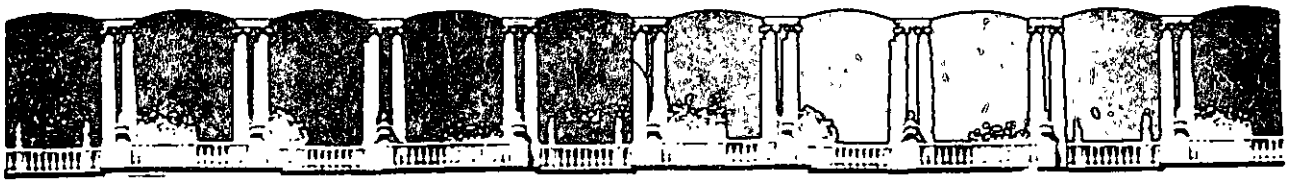


FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CENTRO DE INFORMACION Y DOCUMENTACION
"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.**
- **Noticias técnicas.**
- **Publicaciones periódicas.**
- **Publicaciones de la Academia Mexicana de Ingeniería.**
- **Notas de los cursos que se han impartido de 1980 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 19:30 horas de lunes a viernes.

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.

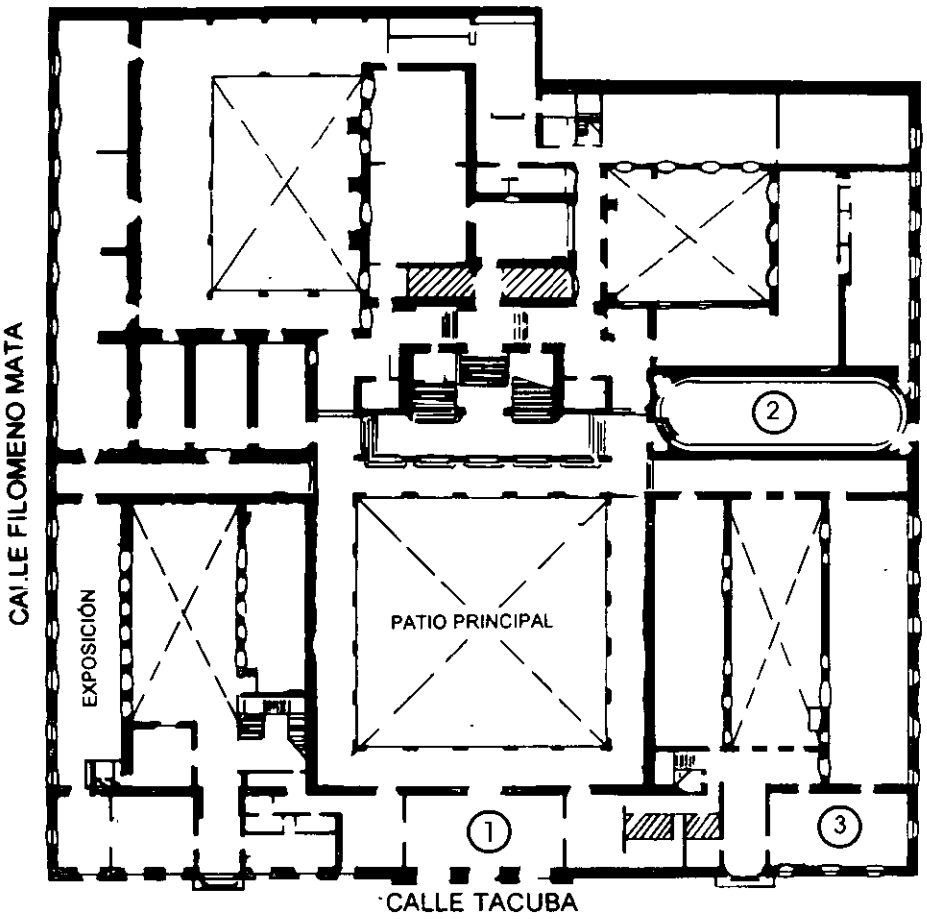
Se recomienda llenar dicha evaluación conforme los profesores impartan sus clases, a efecto de no llenar en la última sesión las evaluaciones y con esto sean más fehacientes sus apreciaciones.

Atentamente

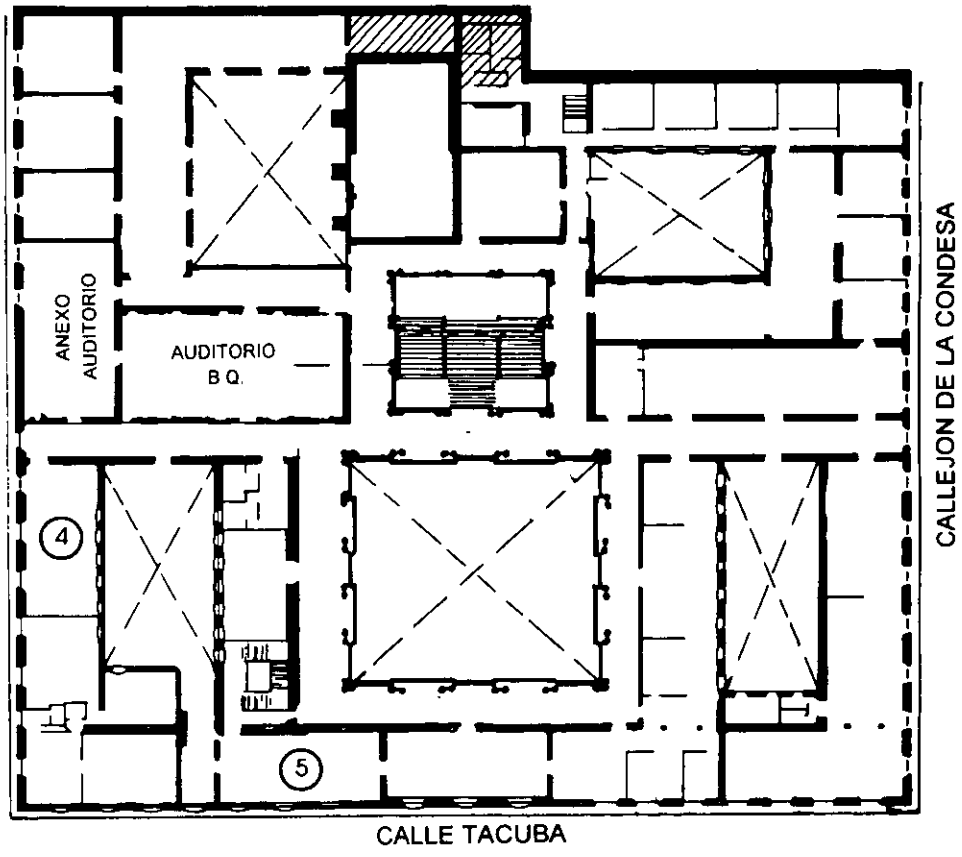
División de Educación Continua.

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

PALACIO DE MINERIA

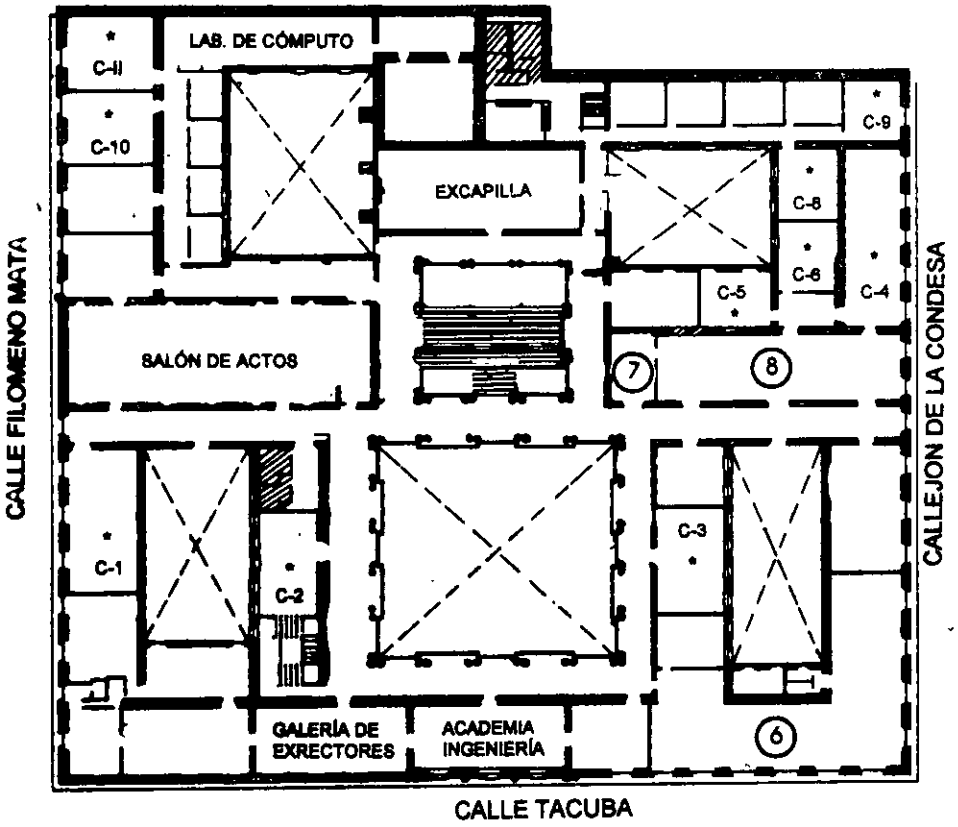


PLANTA BAJA



MEZZANINNE

PALACIO DE MINERIA



Ier. 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

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

Diplomado en Multimedia

Programación Básica en Java

(CC078)

(25/09/98 al 02/10/98)

Profesor:
Ing. Jaime Arturo Ruíz García

Programación Básica en JAVA.

OBJETIVO:

El alumno conocerá los fundamentos del lenguaje de programación JAVA, así como también aplicará este nuevo conocimiento en el desarrollo de aplicaciones que integren esta tecnología en el desarrollo de su trabajo.

PRESENTACIÓN:

En fechas recientes se ha dado el fenómeno llamado internet y consigo han surgido un conjunto de nuevas tecnologías, de las cuales destaca la aparición del lenguaje JAVA, esta nueva herramienta ha logrado en poco tiempo acaparar la atención de un gran número de personas, y todo se debe al hecho de que con su uso se pueden lograr nuevas formas de interacción, así como la utilización de productos multimedios en la construcción de nuevos sitios en internet.

Por lo anterior, este curso permite conocer las bases que conforman esta tecnología, con el único fin de proporcionar a los usuarios de nuevas

herramientas que logren que su trabajo sea más creativo y productivo.

DIRIGIDO A:

Toda persona que desee conocer y aplicar esta tecnología en el desarrollo de su trabajo.

TEMARIO:

1. ***Introducción a Java.***
2. ***Instalación del JDK.***
3. ***Conceptos Básicos de Java.***
4. ***Programación básica en Java (Aplicaciones y Applets).***
5. ***Conceptos Avanzados de Java.***

Desarrollo:

1. **Introducción a Java**
   - 1.1 Origen de Java
   - 1.2 Características de Java
     - Simple
     - Orientado a Objetos
     - Distribuido
     - Robusto
     - De Arquitectura Neutral
     - Seguro
     - Portable
     - Interpretado
     - Multithreaded
     - Tipos de aplicaciones.
     - Lo nuevo

2. Instalación del JDK

- Obtención del Software
- Otros ambientes de desarrollo
- Donde obtener información
- Instalación en Windows 95
- Herramientas del JDK
- Proceso de construcción de Applets y Aplicaciones independientes Java
- Primera Aplicación independiente
- Primer Applet

3. Conceptos Básicos de Java

3.1 Fundamentos del lenguaje Java

- Comentarios
- Palabras Reservadas
- Separadores
- Espacios en blanco
- Identificadores
- Literales
- Operadores
  - Aritméticos
  - Relacionales
  - Lógicos
  - De desplazamiento de bits
- Variables
  - Identificador de una variable
  - Ambito de una variable
  - Inicialización de una variable
  - Variables final
- Tipos de datos
- Expresiones
- Declaraciones
- Condiciones
- Sentencias y bloques
- Estructuras de control
  - Sentencias de Decisión.
  - Sentencias de Repetición.
  - Sentencias de Salto.
  - Otras.

- Arrays
- Cadenas

3.2 Introducción a P.O.O.

3.3 Conceptos de P.O.O. en Java

- Clases
- Tipos de Clases
- Creación de un objeto (Instancia de Clase).
  - Referencias
  - Operador "new"
- Variables y Métodos de Instancia
  - Ambito de una variable.
  - Métodos
  - Sobrecarga de métodos
  - Constructores
  - Finalizadores
- Alcance de Objetos y Reciclado de Memoria
- Herencia
  - Sobreescritura de métodos.
- Control de acceso
- Variables y Métodos Estáticos
- This y Super
- Clases Abstractas
- Interfaces
- Paquetes
  - Sentencia "import"
  - Paquetes de Java

4. Programación básica en Java (Aplicaciones y Applets).

4.1 Compilación y Ejecución de aplicaciones (stand alone)

- Fichero Fuente Java
- Método "main"
- Compilación y Ejecución
- Problemas de compilación

4.2 Compilación y Ejecución de Applets

- Fichero Fuente Applet.java
- Ciclo de vida de un applet
- Componentes básicos de un applet
  - Clases incluidas

- ➢ La clase Applet
  - ➢ Métodos de Applet
- Compilación de un Applet
- La etiqueta APPLET de html
  - ➢ Atributos de APPLET
  - ➢ Paso de parámetros a applets
- Ejecución y prueba de un applet
  - ➢ Uso de "appletviewer"
- Esquema de seguridad de los applets

5. Conceptos Avanzados de Java

5.1 Clases Java.
5.2 Ficheros y Streams.
5.3 Excepciones.
5.4 Threads y Multithreading.
5.5 Código Nativo.

1. Introducción a Java.

1.1 Origen de Java.

En 1990 un grupo de investigadores (Proyecto llamado Green) de Sun Microsystems, estudian la forma de desarrollar sistemas avanzados para una gran variedad de dispositivos en red (networked devices) y para sistemas empotrados o embarcados (embedded systems). En esos tiempos el lenguaje C++ y la metodología por objetos tienen una gran aceptación en el mundo de los investigadores. Sin embargo, después de un primer intento de adoptar este lenguaje para el proyecto Green, se decide crear uno nuevo. Este nuevo lenguaje se basa en las mismas ideas del mundo de objetos y sus lenguajes: Eiffel, SmallTalk, Objetive-C, Cedar/Mesa, y C++; pero adaptando el paradigma al mundo de redes y el desarrollo de aplicaciones distribuidas.

Así surge la idea de un lenguaje y su ambiente de desarrollo en 1991 con la idea de desarrollar productos electrónicos de consumo. En un principio se le bautiza con el nombre de lenguaje Oak y después en el otoño de 1992, se crea un proyecto llamado Star 7* (*7).

Después de un intento fallido de realizar un proyecto para el desarrollo de un conjunto de TV para Time-Warner Inc en 1993, el grupo de desarrollo se enfoca, en 1994, al creciente y popular mundo WWW.

Así, dado que el contrato para Time-Warner Inc no se obtuvo, el grupo de desarrollo encabezado por James Gosling aplican su tecnología al mundo Internet y en particular a generar un hojeador (browser) para WWW, el hojeador HotJava. Así durante 1994 se desarrolla una gran idea. Aplicar la tecnología del proyecto Star 7* al mundo WWW. Así tenemos que en mayo de 1995 surge oficialmente Java y HotJava como grandes proyectos para Sun Microsystems y para todo el mundo Internet.

James Gosling uno de los integrantes destacados en la construcción de Java tiene una gran experiencia trabajando en Sun desde mucho tiempo antes en 1984. Este estudiante graduado BS en ciencias de la computación de la Universidad de Calgary en Canadá y después graduado PhD de la Universidad de Carnegie-Mellon tiene entre sus desarrollos para Sun en UNIX versiones de editor Emacs escritas en C y aspectos postcript del ambiente SunOs NeWS.

Después las fechas abundan y todos los grandes constructores de sistemas adoptan este lenguaje de programación y grandes constructores de chips principian la generación de chips Java y el mundo entero de sistemas adopta Java. Ha pasado de ser una especificación formal de un lenguaje de programación por objetos a un ambiente completo de programación con un compilador, intérprete, depurador, visor de applets, sistema de ejecución del lenguaje (lenguage run-time), bibliotecas de clases, y lenguaje de guiones para facilitar generación de aplicaciones (JavaScript). Hoy en día Java es tema indispensable en todos los medios de información.

Hoy en día el revuelo en los programadores radica en la característica única de Java en compilar programas en un formato binario que pueden ser ejecutados en una gran variedad de plataformas sin tener que recompilar el código.

1.2 Características de Java.

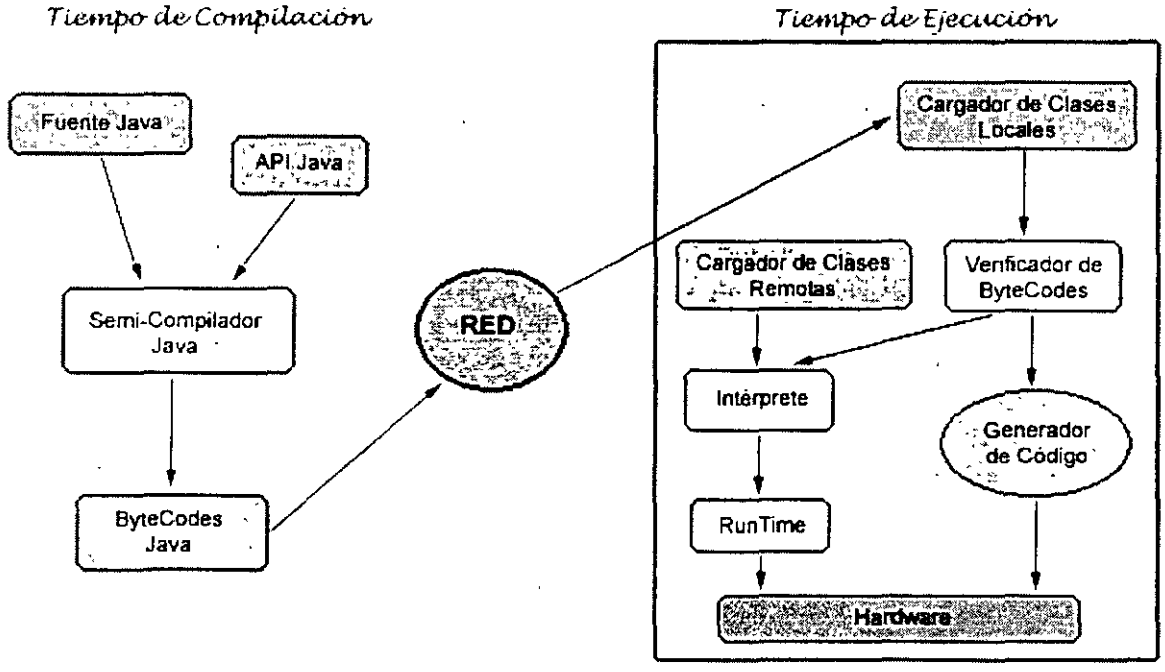
- Simple. Java es un lenguaje que se puede aprender rápidamente una vez que se comprendan los conceptos básicos de la programación orientada a objetos. Sólo se necesitan comprender unos cuantos conceptos para escribir programas productivos y satisfactorios. Java realiza un esfuerzo para no tener características sorprendentes. Hay muchos sistemas de programación que se enorgullecen de su versatilidad a la hora de proporcionar docenas de maneras de realizar lo mismo. Si un lenguaje muestra lo suficiente las interioridades de la máquina, será libre de hacer casi cualquier cosa, de la manera que desee. Aunque es cierto que esto ofrece un rendimiento impresionante para el programador muy cuidadoso y experto, la libertad tiene un precio en cuanto a complejidad y comprensión. En Java, hay un número reducido de maneras de realizar una tarea dada.

- Orientado a Objetos. Java fue diseñado partiendo de cero. No es un derivado directo de otro lenguaje de programación, y no es compatible con ninguno de ellos. Debido a que existió la libertad de diseñar a partir de una pizarra en blanco, se eligió un enfoque para los objetos limpio, utilizable y pragmático, Java ha tomado prestadas con libertad ideas de muchos entornos de software de objetos pioneros de las últimas décadas, con lo que consigue tener un modelo de objetos simple y fácil de ampliar. Soporta las tres características de este tipo de programación: encapsulación, herencia y polimorfismo. Además incluye un conjunto de librerías de clases para obtener los tipos básicos, procedimientos de entrada/salida, comunicaciones a través de red y clases para desarrollar interfaces gráficas de usuario. Por último Java evita las características más problemáticas de lenguajes más antiguos como él "C++" , por ejemplo: no existen punteros, ni aritmética de punteros y el manejo de memoria es automático entre otras cosas.

- Distribuido. Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de clases para acceder e interactuar con protocolos como http y ftp. Esto permite a los programadores acceder a la información a través de la red con tanta facilidad como a los archivos locales. Java en sí no es distribuido, sino que proporciona las librerías y herramientas para que los programas puedan ser distribuidos.

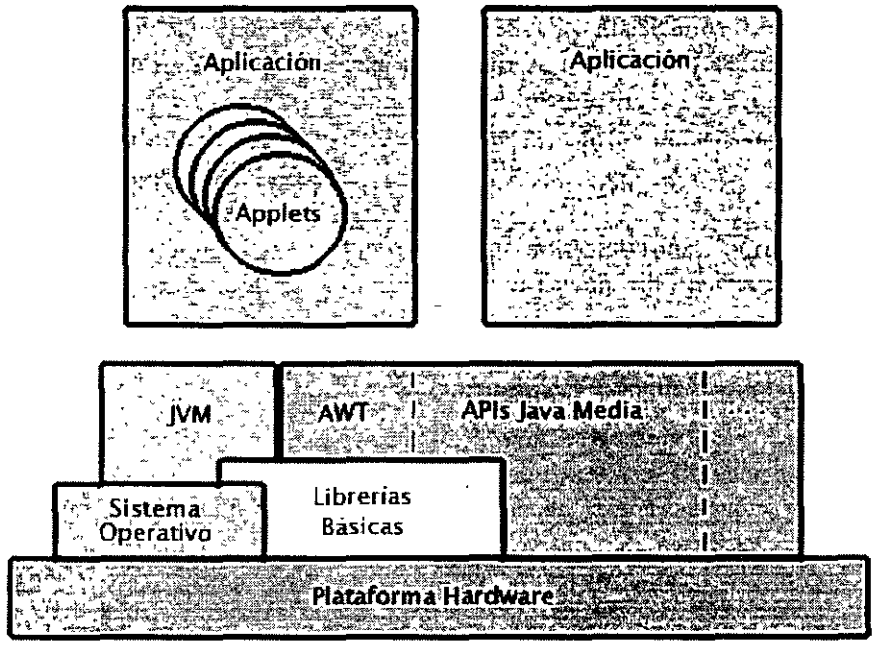
- Robusto. Podría pensar que la robustez es un lujo en estos tiempos de aventuras en Internet. Dado que todos ejecutamos versiones "alfa" o "beta" de la mayoría de las herramientas de Internet, ¿por qué preocuparnos por un lenguaje de programación robusto? ¿No está eso reservado para los programadores de los transbordadores espaciales? Ya no. Java le restringe en unas cuantas áreas clave para obligarle a encontrar pronto sus errores en el desarrollo del programa. A la vez, Java le libera de tener que preocuparse por muchas de las causas principales de error del programador en la mayoría de los otros lenguajes. Muchos de los errores difíciles de encontrar que se convierten a menudo en situaciones en tiempo de ejecución difíciles de reproducir son imposibles de crear en Java.

- De Arquitectura Neutral. Para establecer Java como parte integral de la red, el compilador Java compila su código a un archivo objeto con formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución ("run-time" Java Virtual Machine) puede ejecutar ese código objeto, sin importar en modo alguno la máquina en que ha sido generado. Actualmente existen sistemas run-time para Solaris 2.x, SunOs 4.1.x, Windows '95, Windows NT, Linux, Irix, Aix, Mac, Apple y probablemente haya grupos de desarrollo trabajando en el porting a otras plataformas.



El código fuente Java se "compila" a un código de bytes de alto nivel independiente de la máquina. Este código (ByteCode) está diseñado para ejecutarse en una máquina hipotética que es implementada por un sistema run-time, que sí es dependiente de la máquina.

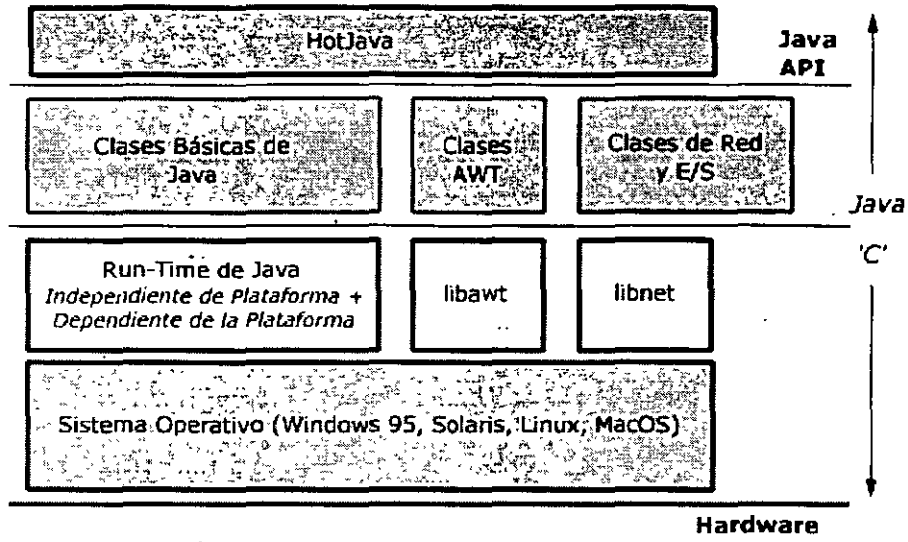
En una representación en que tuviésemos que indicar todos los elementos que forman parte de la arquitectura de Java sobre una plataforma genérica, obtendríamos una imagen como la siguiente:



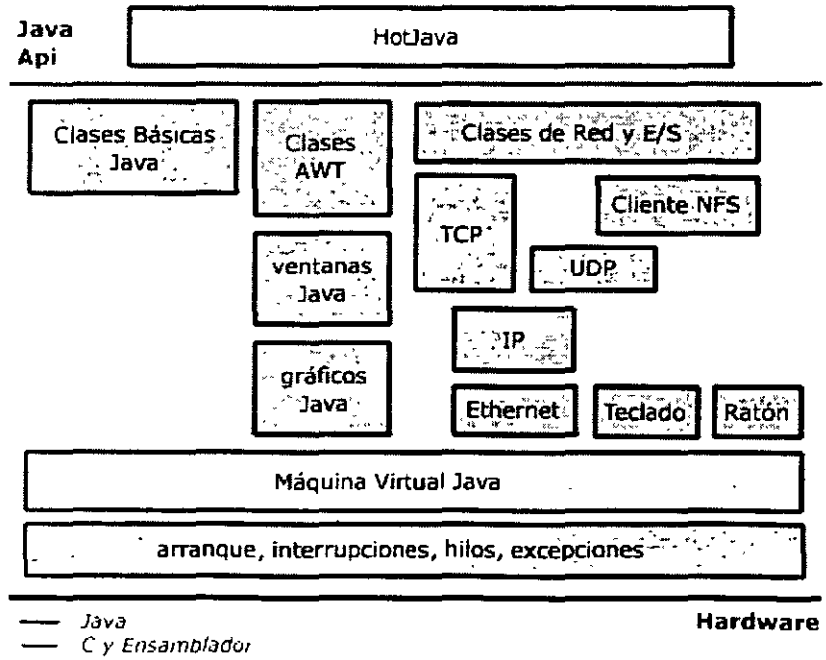
En ella podemos ver que lo verdaderamente dependiente del sistema es la Máquina Virtual Java (JVM) y las librerías fundamentales, que también permitirían acceder directamente al hardware de la máquina. Además, siempre habrá APIs de Java que también entren en contacto directo con el hardware y serán dependientes de la máquina, como ejemplo de este tipo de APIs podemos citar:

- Swing. Mejora de las herramientas para la implantación de interfaces de usuario.
- Java 2D. Gráficos 2D y manipulación de imágenes
- Java Media Framework. Elementos críticos en el tiempo: audio, video...
- Java Animation. Animación de objetos en 2D
- Java Telephony. Integración con telefonía
- Java Share. Interacción entre aplicaciones multiusuario.
- Java 3D. Gráficos 3D y su manipulación

La siguiente figura ilustra la situación en que se encuentra Java cuando se ejecuta sobre un sistema operativo convencional. En la imagen se observa que si se exceptúa la parte correspondiente al Sistema Operativo que ataca directamente al hardware de la plataforma, el resto está totalmente programado por Javasoft o por los programadores Java, teniendo en cuenta que HotJava es una aplicación en toda regla, al tratarse de un navegador y estar desarrollado en su totalidad en Java. Cualquier programador podría utilizar las mismas herramientas para levantar un nuevo desarrollo en Java, bien fuesen applets para su implantación en Internet o aplicaciones para su uso individual.



Y la imagen que aparece a continuación muestra la arquitectura de Java sin un Sistema Operativo que lo ampare, de tal forma que el kernel tendría que proporcionar suficientes posibilidades como para implementar una Máquina Virtual Java y los drivers de dispositivos imprescindibles como pantalla, red, ratón, teclado y la base para poder desarrollar en Java librerías como AWT, ficheros, red, etc., Con todo ello estaría asegurado el soporte completo del API de Java, quedando en mano de los desarrolladores la implantación de aplicaciones o applets o cualquier otra librería.



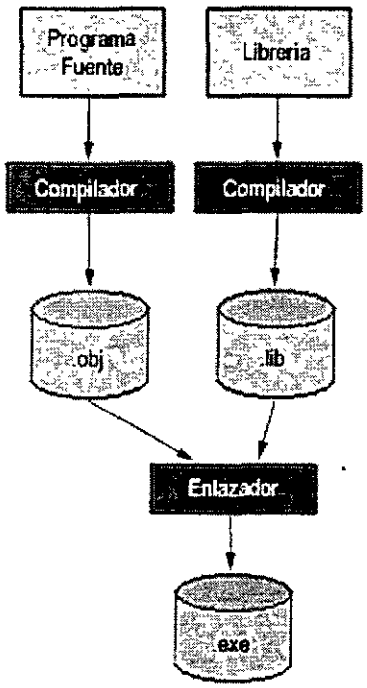
- Seguro. Actualmente es muy popular en la prensa él termino seguridad, especialmente cuando se refiere a Internet. Todo el mundo esta preocupado porque piensa que el introducir el comercio en Internet es tan seguro como dibujar su número de tarjeta de crédito en la lateral de un autobús. La amenaza de virus o caballos de Troya está en todas partes. Incluso sus documentos de procesamiento de textos pueden contener virus. La mayoría de estos problemas se derivan de sistemas más antiguos que en su principio nunca fueron diseñados teniendo en cuenta el concepto de seguridad en Internet. Uno de los principios de diseño claves de Java es la seguridad. Java nunca ha tenido características inseguras que ahora se tengan que remediar para hacerlo seguro. La seguridad en Java tiene dos facetas. En el lenguaje, características como los punteros o el casting implícito que hace el compilador de C y C++ se eliminan para prevenir el acceso ilegal a la memoria. El código Java pasa muchos tests antes de ejecutarse en una máquina. El código se pasa a través de un verificador de ByteCode que comprueba el formato de los fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código ilegal -código que falsea punteros, viola derechos de acceso sobre objetos o intenta cambiar el tipo o clase de un objeto-. Si los ByteCodes pasan la verificación si generar ningún mensaje de error, entonces sabemos que:

    - El código no produce desbordamiento de operandos en la pila.
    - El tipo de los parámetros de todos los códigos de operación son conocidos y correctos.
    - No ha ocurrido ninguna conversión ilegal de datos, tal como convertir enteros en punteros.
    - El acceso a los campos de un objeto se sabe que es legal: public, private, protected.
    - No hay ningún intento de violar las reglas de acceso y seguridad establecidas.

- Portable. Más allá de la portabilidad básica por ser de arquitectura independiente, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Los enteros son siempre enteros y además, enteros de 32 bits en complemento a 2. Además, Java construye sus interfaces de usuario a través de un sistema abstracto de ventanas de forma que las ventanas puedan ser implantadas en entornos Unix, Pc o Mac.

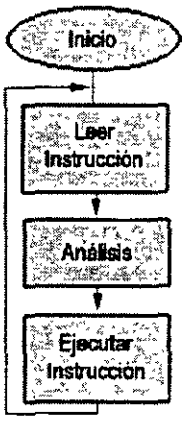
- Interpretado. Java es más lento que otros lenguajes de programación, como C++, ya que debe ser interpretado y no ejecutado como sucede en cualquier programa tradicional. Se dice que Java es de 10 a 30 veces más lento que C, y que tampoco existen en Java proyectos de gran envergadura como en otros lenguajes. La verdad es que ya hay comparaciones ventajosas entre Java y el resto de los lenguajes de programación. Java para conseguir ser un lenguaje independiente del sistema operativo y del procesador que incorpore la máquina utilizada, es tanto interpretado como compilado. Y esto no es contradictorio, el código fuente escrito con cualquier editor se compila generando el ByteCode.

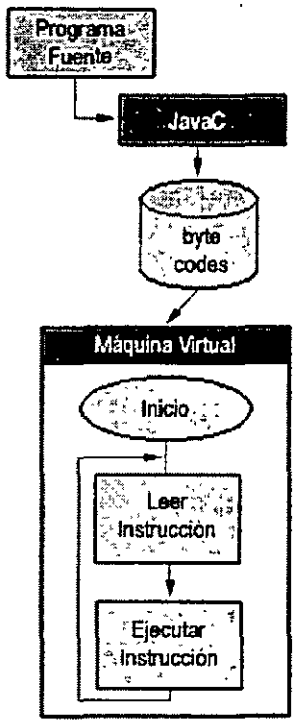
Este código intermedio es de muy bajo nivel, pero sin alcanzar las instrucciones máquina propias de cada plataforma. El ByteCode corresponde al 80% de las instrucciones de la aplicación. Ese mismo código es el que se puede ejecutar sobre cualquier plataforma. Para ello hace falta el runtime, que sí es completamente dependiente de la máquina y del sistema operativo que interpreta dinámicamente el ByteCode y añade el 20% de instrucciones que faltaban para su ejecución. Con este sistema es fácil crear aplicaciones multiplataforma, pero para ejecutarlas es necesario que exista el runtime correspondiente al sistema operativo utilizado.



La imagen de la izquierda muestra las acciones correspondientes a un compilador tradicional. El compilador traslada las sentencias escritas en lenguaje de alto-nivel a múltiples instrucciones, que luego son enlazadas junto con el resultado de múltiples compilaciones previas que han dado origen a librerías, y juntando todo ello, es cuando se genera un programa ejecutable.

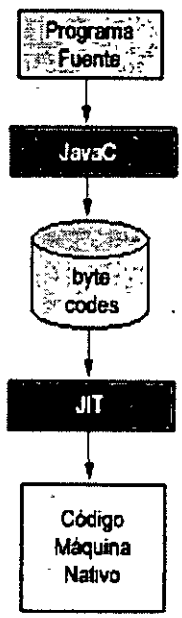
La imagen de la derecha muestra la forma de actuación de un intérprete. Básicamente es un enorme bucle, en el cual se va leyendo o recogiendo cada una de las instrucciones del programa fuente que se desea ejecutar, se analiza, se parte en trozos y se ejecuta. Luego se va a recoger la siguiente instrucción que se debe interpretar y se continúa con este proceso hasta que se terminan las instrucciones o hasta que entre las instrucciones hay alguna que contiene la orden de detener la ejecución de las instrucciones que componen el programa fuente.

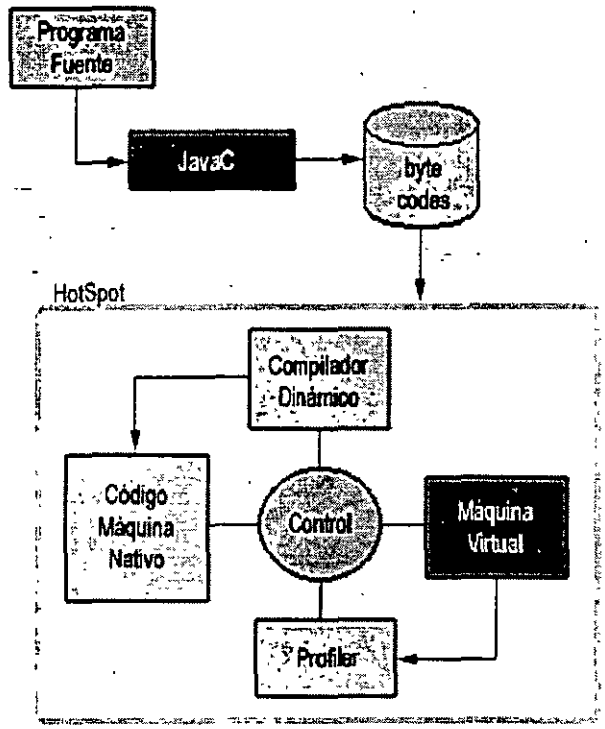




La imagen siguiente, situada a la izquierda, muestra un tipo de intérprete más eficiente que el anterior, el intérprete de ByteCodes, que fue popularizado hace más de veinte años por la Universidad de California al crear el UCSD Pascal. En este caso, el intérprete trabaja sobre instrucciones que ya han sido trasladadas a un código intermedio en un paso anterior. Así, aunque se ejecute en un bucle, se elimina la necesidad de analizar cada una de las instrucciones que componen el programa fuente, porque ya lo han sido en el paso previo. Este es el sistema que Java utiliza, y la Máquina Virtual Java es un ejemplo de este tipo de intérprete. No obstante, sigue siendo lento, aunque se obtenga un código independiente de plataforma muy compacto, que puede ser ejecutado en cualquier ordenador que disponga de una máquina virtual capaz de interpretar los ByteCodes trasladados.

Un paso adelante en el rendimiento del código Java lo han representado los compiladores Just-In-Time, que compilan el código convirtiéndolo a código máquina antes de ejecutarlo. Es decir, un compilador JIT va trasladando los ByteCodes al código máquina de la plataforma según los va leyendo, realizando un cierto grado de optimización. El resultado es que cuando el programa se ejecute, habrá partes que no se ejecuten y que no serán compiladas, y el compilador JIT no perderá el tiempo en optimizar código que nunca se va a ejecutar. No obstante, los compiladores JIT no pueden realizar demasiadas optimizaciones, ya que hay código que ellos no ven, así que aunque siempre son capaces de optimizar la parte de código de inicialización de un programa, hay otras partes que deben ser optimizadas, según se van cargando, con lo cual, hay una cierta cantidad de tiempo que inevitablemente ha de perderse.





Y, finalmente, se presenta la última tendencia en lo que a compilación e intérpretes se refiere. Lo último en que trabaja Sun es HotSpot, una herramienta que incluye un compilador dinámico y una máquina virtual para interpretar los ByteCodes, tal como se muestra en la figura superior. Cuando se cargan los ByteCodes producidos por el compilador por primera vez, éstos son interpretados en la máquina virtual. Cuando ya están en ejecución, el profiler mantiene información sobre el rendimiento y selecciona el método sobre el que se va a realizar compilación. Los métodos ya compilados se almacenan en un caché en código máquina nativo. Cuando un método es invocado, esta versión en código máquina nativo es la que se utiliza, en caso de que exista: en caso contrario, los ByteCodes son reinterpretados. La función control que muestra el diagrama es como un salto indirecto a través de la memoria que apunta tanto al código máquina como al interpretado, aunque Sun no a proporcionado muchos detalles sobre este extremo

- Multithreaded. Al ser MultiHilo (o multihilvanado, mala traducción de multithreaded), Java permite muchas actividades simultáneas en un programa. Los hilos -a veces llamados, procesos ligeros, o hilos de ejecución- son básicamente pequeños procesos o piezas independientes de un gran proceso. Al estar estos hilos construidos en el mismo lenguaje, son más fáciles de usar y más robustos que sus homólogos en C o C++. El beneficio de ser multihilo consiste

en un mejor rendimiento interactivo y mejor comportamiento en tiempo real. Aunque el comportamiento en tiempo real está limitado a las capacidades del sistema operativo subyacente (Unix, Windows, etc.) de la plataforma, aún supera a los entornos de flujo único de programa (single-threaded) tanto en facilidad de desarrollo como en rendimiento. Cualquiera que haya utilizado la tecnología de navegación concurrente, sabe lo frustrante que puede ser esperar por una gran imagen que se está trayendo de un sitio interesante desde la red. En Java, las imágenes se pueden ir trayendo en un hilo de ejecución independiente, permitiendo que el usuario pueda acceder a la información de la página sin tener que esperar por el navegador.

En consecuencia Java puede trabajar con sistemas operativos de alto nivel que soporten multienhebramiento. De esta forma, un programa Java puede tener más de una hebra en ejecución. Por ejemplo podría realizar un cálculo largo en una hebra, mientras otras interactúan con el usuario. Así los usuarios no tienen que dejar de trabajar mientras los programas Java completan las operaciones más largas.

- Tipos de Aplicaciones. Existen diferentes tipos de aplicaciones que podemos desarrollar en Java:

Modelo de Aplicación.	Ambiente.	Ventajas.	Desventajas.	**Usos.**
Applet.	Corre dentro de un Navegador.	Muy seguro. Cargado via red.	No tiene acceso a los recursos locales. No puede escribir o leer archivos locales. Debe existir un Navegador que le de el contexto de ejecución.	Contenido dinámico e interactivo en páginas HTML. Front-end para aplicaciones cliente-servidor.
Ventanas. (Aplicaciones Independientes).	Los archivos .class son cargados desde el sistema de archivos local, y son ejecutados por la JVM local. Los usuarios interactuan con la aplicación con ventanas.	Acceso a todos los recursos locales. Pueden usar métodos nativos para acceder al hardware o integrarse con otras aplicaciones no hechas en Java.	No hay un esquema de seguridad en el acceso a los recursos. Los usuarios deben tener una copia local para poder usar la aplicación.	Puedes ser usados como cualquier aplicación de ventanas.

Modelo de Aplicación.	Ambiente.	Ventajas.	Desventajas.	Usos.
Consola. (Aplicaciones Independientes)	Los archivos .class son cargados desde el sistema de archivos local, y son ejecutados por la JVM local. Los usuarios interactuan con la aplicación mediante comandos tecleados en la línea de comandos.	Acceso a todos los recursos locales. Pueden usar métodos nativos para acceder al hardware o integrarse con otras aplicaciones no hechas en Java.	No hay un esquema de seguridad en el acceso a los recursos. Los usuarios deben tener una copia local para poder usar la aplicación.	Usada donde la interacción con el usuario es simple o innecesaria. Puede usarse como un servidor en una aplicación cliente servidor.
Cliente-Servidor. (Aplicaciones Independientes).	Una aplicación, el servidor, corre en una computadora especial. Esta aplicación controla el acceso a la información y los recursos, así como también responde solicitudes de alguno de estos recursos que maneja. Otra aplicación, el cliente, que corre en otra maquina interactúa con el usuario, además de mandar solicitudes de información o recursos a la aplicación servidor.	Múltiples clientes pueden acceder a los recursos de un solo servidor, eliminando la necesidad de que cada cliente cuente con sus propios recursos. El servidor centraliza el control sobre los recursos así como la seguridad con lo cual se consigue un control de acceso y una mayor utilización de los recursos.	Clientes y servidores deben ser mantenidos. Si el servidor fallá provocará que muchos clientes no realizan su trabajo. Debe existir un medio de comunicación entre las dos aplicaciones el cual debe ser seguro y fiable.	Aplicaciones Multiusuario. Acceso a información en bases de datos centrales.

- Lo nuevo. El lenguaje Java es algo en evolución y a lo largo de su corta vida a recibido algunas modificaciones a su estructura así como nuevas funciones, en este espacio mencionaremos algunos de esos cambios.

  - El lenguaje Java había permanecido estable en su estructura hasta el momento en que SUN dio a conocer la versión 1.1 del lenguaje la cual contenía una serie de mejoras así como nueva funcionalidad de la cual lo más importante ha sido:

    1. El A.W.T. (las clases para el manejo de interfaces gráficas) para Win32 ha sido reescrito totalmente, mejorando su rendimiento.
    2. Se cambio el modelo de manejo de eventos dentro del A.W.T. por considerar al anterior de bajo rendimiento.
    3. Aparecen distintos conjuntos de clases denominados API´s, todos con un propósito bien definido:

API	Propósito
JDBC	Acceso a Base de Datos.
RMI	Capacidad para crear aplicaciones distribuidas totalmente en Java.
JavaBeans	Modelo de componentes de Java.
Seguridad	Clases para la incorporación de elementos de seguridad como: criptografía, firmas digitales, etc.
IDL	Provee interoperabilidad con CORBA.
JFC	Clases para multimedia.

Lo malo con estos cambios ha sido que no todos los desarrolladores de Java y todas las aplicaciones que soportan Java como los Navegadores los han llevado a cabo, lo que nos coloca en el dilema de no poder utilizar algunas de estas nuevas Api's ya que no existe soporte para ellas y por consiguiente los programas que tengan algunos de estos rasgos no podrán ejecutarse.

2. Instalación del JDK.

- **Obtención del Software.**

Sin lugar a dudas en la actualidad ha surgido una gran cantidad de software para desarrollar programas en Java, sin embargo, realmente se necesita poco para lograr este propósito, lo más indispensable es obtener el J.D.K. (entorno de desarrollo Java), el cuál se puede adquirir de forma gratuita a través de la red, Sun ha adoptado la misma estrategia de mercado que adoptó en su momento Netscape, la distribución gratuita de sus productos, creando así una masa de usuarios que permita el establecimiento de un estándar de facto. También es necesario el contar con la documentación adecuada de las clases que proporciona Java, un editor de textos y un Navegador (Explorer, Netscape). En consecuencia todo lo necesario para empezar a desarrollar en Java puede ser encontrado en:

http://java.sun.com.

En donde se debe bajar en particular el JDK y la documentación del API de Java.
Con lo que respecta al editor de textos muy bien puede servir el "edit de Ms-dos" o cualquier otro que guarde los documentos en formato de texto ascci. El Navegador se puede obtener de la red en:

http://www.netscape.com.

- **Otros ambientes de desarrollo.**

Muchas herramientas de desarrollo para Java han surgido, y continuamente surgen más el tratar de mencionar todas es algo complicado, sin embargo en revistas han salido artículos comparativos de diferentes herramientas. En la sección donde obtener información se encuentran algunos lugares donde existe este tipo de estudios. Así es que lo que viene a continuación en una lista de las herramientas más famosas de desarrollo:

Herramienta	Empresa.
SUN	Java Workshop.
Microsoft	Visual J++
Symantec	Café
Borland	JBuilder

- **Donde obtener información.**

Información acerca de Java existe por todas partes, aquí mencionaremos algunos de los lugares más famosos en el WEB que hablan acerca de Java. También podrá encontrar más información en revistas así como en libros, con lo que respecta al último medio este material proporciona una amplia bibliografía al final.

Estas son las direceiones:

http://www.gamelan.com
http://www.javaworld.com
http://java.sun.com

- **Instalación del J.D.K. en Windows 95.**

1. Obtener la versión correcta del software del J.D.K. de la dirección mencionada (http://java.sun.com.), así como el archivo de la documentación en caso de querer tener una copia local.

2. Crear un directorio bajo "C:/" llamado "Java" y ahí guardar el archivo del J.D.K.

3. Ejecutar el archivo de instalación.

4. Borrar el archivo de instalación para liberar espacio.

5. Poner en el Path la ruta donde se encuentran las herramientas del J.D.K. en este caso C:\JAVA\BIN.

6. Hacer un test de la instalación tratando de ejecutar cualquiera de los demos que vienen dentro del J.D.K

- **Herramientas del J.D.K.**

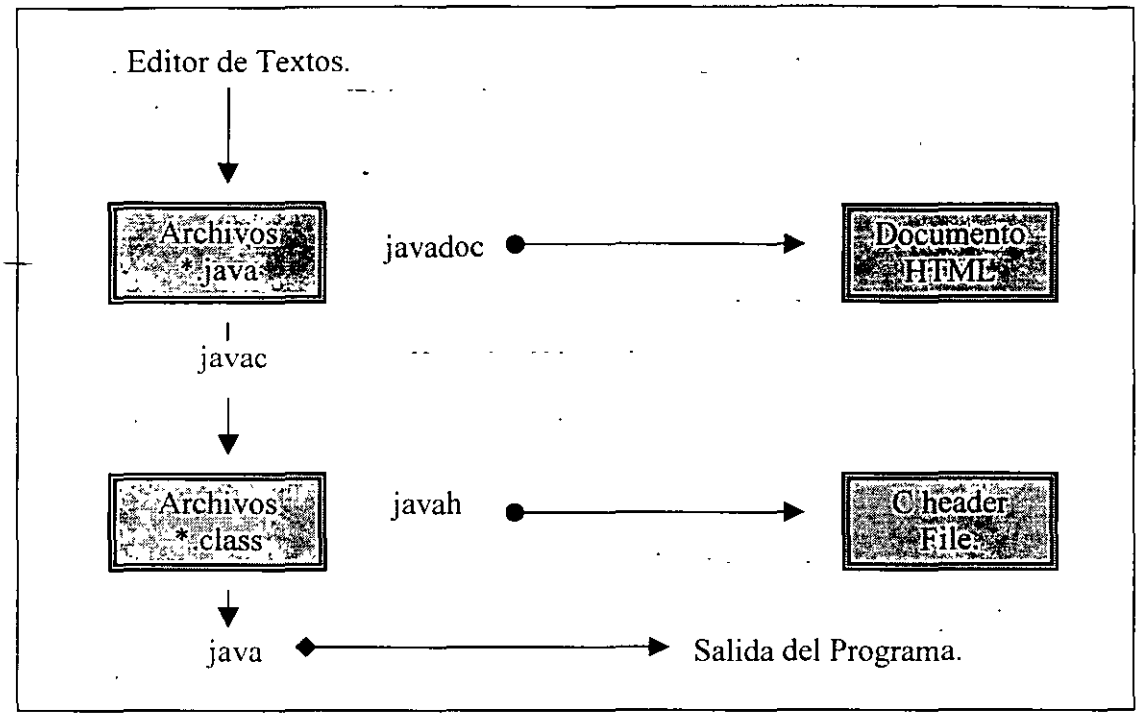
El J.D.K. viene con un conjunto de herramientas que nos ayudan en el proceso de compilación, ejecución, depuración y documentación de nuestros programas. A continuación se muestra una tabla de las herramientas con que dispone el J.D.K.

Aplicación.	Nombre de la herramienta.	Descripción.
appletviewer	El visor de applets Java.	Usado para ver un aplett sin la necesidad de un navegador.
java	El interprete de Java.	Es el encargado de ejecutar las aplicaciones independientes de Java, esto lo hace en forma interpretada.
javac	El compilador de Java.	Compila los archivos fuente de Java "*.java", generando los archivos de byte-code cuya extensión es "*.class".
javadoc	El generador de documentación.	Crea paginas de documentacion en formato HTML a partir de los archivos fuente de Java.
javah	El generador de header files para C.	Generador de header files para c con lo que se logra la interacción entre estos lenguajes.
javap	El desensamblador de archivos "*.class" de Java.	Desensambla archivos "*.class" cuyo contenido son los byte-codes para mostrar los métodos y las variables miembro accesibles en una clase compilada.
jdb	El debugger de lenguaje Java.	Ayuda a encontrar y corregir problemas en el código java.

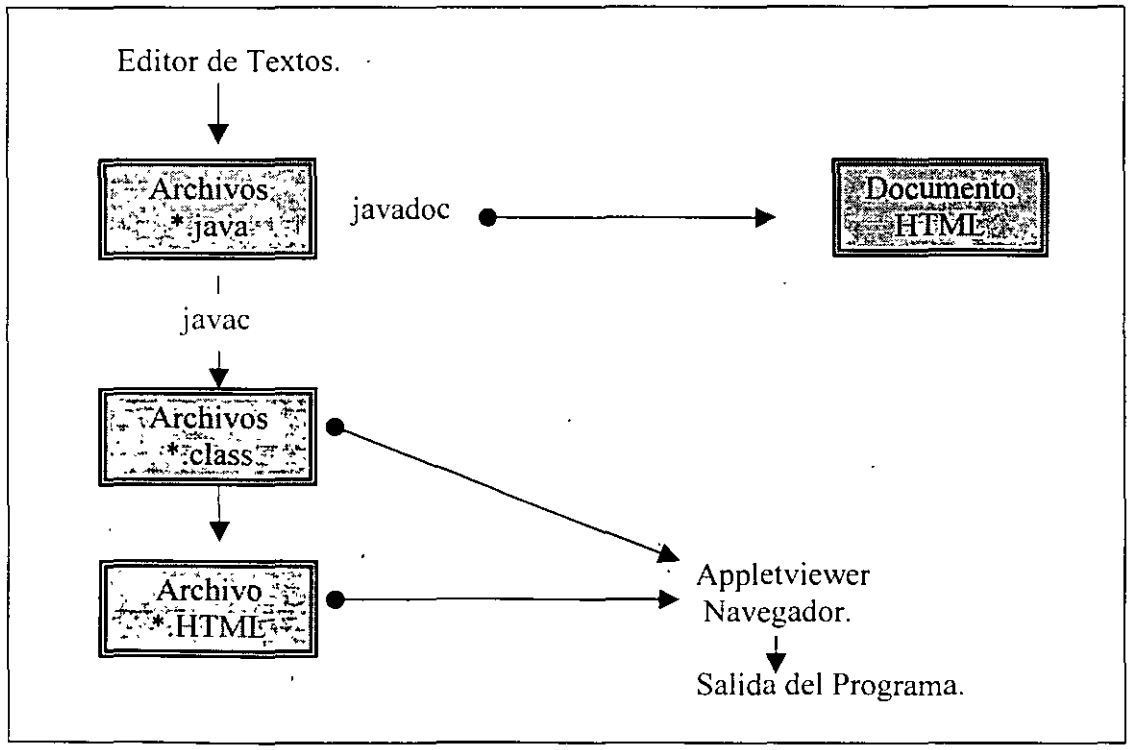
- **Proceso de construcción de Applets y Aplicaciones independientes Java.**

Es tiempo ahora de establecer el conjunto de pasos que se siguen cuando se desea crear un programa en Java. Como hemos visto existen dos grandes tipos de aplicaciones: los Applets y las Aplicaciones independientes, por consiguiente a continuación se muestran estos pasos junto con las herramientas que se utilizan en cada uno de ellos.

- Construcción de Aplicaciones Java.



- Construcción de Applets Java.



A continuación se expresan algunas consideraciones a tomar en cuenta cuando se desarrolla el proceso de construcción de programas en Java.

- Java requiere que todos los archivos fuente tengan la extensión ".java".
- Java requiere que el nombre del archivo fuente sea exactamente igual al nombre de la primera clase declarada dentro del archivo.

- **Primera Aplicación independiente.**

Para reafirmar lo que llevamos hasta ahora realizaremos una pequeña aplicación independiente tipo consola en donde lo más importante será el poner en práctica los pasos para la construcción de un programa en Java y no tanto la funcionalidad ni la sintaxis del mismo.

1. Primero crearemos un directorio donde guardaremos todos nuestros programas. En este caso el directorio estará bajo "C:\users\mi_nombre\java_apps.
2. Como segundo paso editaremos nuestro archivo fuente utilizando un procesador de textos que guarde documentos como texto simple. Nuestro archivo se llamara PrimeraApps.java y por consiguiente el nombre de la clase dentro del archivo debe ser PrimeraApps para que no existan errores. Este es el contenido del archivo:

```
class PrimeraApps {
        public static void main (String args[ ]) {
                System.out.println("Mi primera Aplicación Independiente es un éxito.");
        }
}
```

3. Después de haber editado nuestro archivo fuente y haberlo guardado es tiempo de compilarlo. Para compilar nuestro archivo desde la línea de comandos de una pantalla Ms-Dos y estando en el mismo directorio donde se encuentra el archivo fuente hay que teclear lo siguiente:

```
Javac PrimeraApps.java
```

Si todo va bien el compilador hará su trabajo, es decir creará el archivo de byte-Code llamado PrimeraApps.class en el mismo directorio y entonces nos devolverá la línea de prompt. Si no se genera el archivo ".class" quiere decir que nuestro programa tiene algún error de sintaxis y por consiguiente será necesario revisarlo nuevamente para corregir el problema.

4. Finalmente es tiempo de ejecutar nuestro programa, como en este caso es una Aplicación independiente utilizaremos el interprete java para ejecutarla. En la línea de comandos hay que teclear lo siguiente:

   Java PrimeraApps

Y como podrás ver la computadora ejecuta el programa desplegando el mensaje:

Mi primera Aplicación Independiente es un éxito.

Pues bien, felicidades por tu primer programa en Java, si ya se que no es mucho, sin embargo, si partimos de cosas sencillas podrá entender los conceptos de una mejor forma con lo que a la larga podrá desarrollar aplicaciones realmente sorprendentes y complejas.

- **Primer Applet.**

Continuando con nuestros primeros pasos, vamos a hacer un Applet que a final de cuentas son una de las cosas por las cuales el lenguaje Java es tan famoso. Nuevamente le recuerdo que no se preocupe por la funcionalidad o sintaxis del Applet ya que con lo que hemos visto hasta ahora todavía no esta en la posibilidad de entenderlo y por consecuencia los posteriores capítulos tendrán como objetivo que usted llegue a comprender todo lo que envuelve la programación en Java. Dicho lo anterior pasemos a la construcción de nuestro primer Applet:

1. Al igual que una Aplicación independiente lo primero que haremos será editar nuestro archivo fuente utilizando un procesador de textos que guarde documentos como texto simple. Nuestro archivo se llamará PrimerAppt.java y por consiguiente el nombre de la clase dentro del archivo debe ser PrimerAppt para que no existan errores. Este es el contenido del archivo:

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

public  class PrimerAppt extends Applet {
        Image NuevaImagen;

        public void init( ) {
                resize(400,400);
                NuevaImagen = getImage(getCodeBase( ), " imagen ");
        }

        public void paint(Graphics g) {
                g.drawImage(NuevaImagen, 0, 0, this);
                play(getCodeBase( ), "sonido ");
                g.drawString("Mi primer Applet", 10, 350);
        }
}
```

2. Después de haber editado nuestro archivo fuente y haberlo guardado es tiempo de compilarlo. Para compilar nuestro archivo desde la línea de comandos de una pantalla Ms-Dos y estando en el mismo directorio donde se encuentra el archivo fuente hay que teclear lo siguiente:

```
Javac PrimerAppt.java
```

Si todo va bien el compilador hará su trabajo, es decir creará el archivo de byte-Code llamado PrimerAppt.class en el mismo directorio y entonces nos devolverá la línea de prompt. Si no se genera el archivo ".class" quiere decir que nuestro programa tiene algún error de sintaxis y por consiguiente será necesario revisarlo nuevamente para corregir el problema.

3. En este punto es donde se hace la diferencia entre un Applet y una Aplicación independiente, ya que es necesario crear una página HTML que posea una etiqueta especial llamada Applet y cuyo propósito es incrustar nuestro Applet en la página para que el navegador lo pueda manipular. El archivo HTML de nuestro ejemplo es el siguiente:

```
<HTML>
<HEAD>
<TITLE> MI PRIMER APPLET </TITLE>
</HEAD>
<BODY>
<APPLET CODE="PrimerAppt"  WIDTH=400 HEIGHT=400> </APPLET>
</BODY>
</HTML>
```

4. Finalmente es tiempo de ejecutar nuestro programa, como en este caso es un Applet utilizaremos el visor de Applets de Java llamado appletviewer para ejecutarla. Esta herramienta toma como entrada un archivo HTML así que entonces en la línea de comandos hay que teclear lo siguiente:

```
appletviewer  PrimerAppt.html
```

Y como podrás ver la computadora habré una ventana donde ejecuta el Applet.

3. Conceptos Básicos de Java.

3.1 Fundamentos del lenguaje Java.

El código que se genera en Java esta compuesto de un número de diferentes piezas. A estos distintos tipos de piezas se les denomina Tokens. Los Tokens son como los átomos. En consecuencia un Token no puede ser dividido en piezas más pequeñas sin alterar su significado fundamental.

Existen siete tipos de Tokens diferentes: palabras reservadas, comentarios, separadores, espacios en blanco, operadores, identificadores y literales. Estos distintos tipos de Tokens son combinados de diversas formas para formar estructuras más complejas del lenguaje que son las expresiones, declaraciones y sentencias, a su vez estas formas se combinan con las estructuras de control y variables y entonces son agrupadas en bloques para formar los atributos (variables miembro de un objeto), las operaciones (los métodos de un objeto) y otro tipo de entes que en su conjunto forman las clases de nuestro programa. De estas clases podremos crear los objetos que a través de la interacción con otros objetos realizarán las tareas que nuestro programa debe realizar.

- Comentarios.

Cuando se realiza un programa la documentación del mismo es indispensable, ya que siempre es útil el almacenar información en el mismo código que describa lo que ese programa hace y como lo hace. Esto se consigue mediante el uso de los comentarios. Una característica fundamental de los comentarios es que su contenido es ignorado por el compilador y en consecuencia no tienen efecto sobre el código elaborado, ya sea en su lógica o en su ejecución, sin embargo son una valiosa herramienta cuando se hace mantenimiento a los programas.

En Java existen tres tipos diferentes de comentarios:

Sintaxis:

```
/*   */
```

Uso:

Son utilizados para hacer comentarios de varias líneas.

Ejemplo:

```
/* Este es un comentario, y como pueden ver  puede abarcar varias
   líneas sin ningún tipo de problema */
```

Sintaxis:

//

Uso:

Son utilizados para hacer comentarios de una sola línea.

Ejemplo:

```
// Con este tipo de comentarios no puedo abarcar más de una línea
// a menos que inicie cada línea con la marca de comentario.
```

Sintaxis:

/** */

Uso:

Son comentarios con un significado especial, ya que son utilizados por una de las herramientas del J.D.K. (javadoc) con propósito de generar una página HTML de documentación del programa.

- Palabras Reservadas.

Son palabras que tienen un significado especial dentro de Java y por consiguiente el programador no las puede utilizar para otros propósitos como nombrar variables o métodos. Estas palabras siempre están en minúsculas. Se utilizan para expresar las construcciones básicas del lenguaje.

Palabras Reservadas Para tipos de datos.	
boolean	Declara una variable o un retorno de tipo booleano.
byte	Declara una variable o un retorno de tipo byte.
char	Declara una variable o un retorno de tipo char.
double	Declara una variable o un retorno de tipo double.
float	Declara una variable o un retorno de tipo float.
int	Declara una variable o un retorno de tipo int.
long	Declara una variable o un retorno de tipo long.
short	Declara una variable o un retorno de tipo short.
Palabras Reservadas Para estructuras de control de repetición.	
break	Sale de un ciclo en forma prematura.
continue	Prematuramente regresa al inicio de un ciclo.
do	Inicia un ciclo do-while.
for	Inicia un ciclo for.
while	Inicia un ciclo while o termina un ciclo do-while.
Palabras Reservadas Para estructuras .de control de decisión.	
case	Un caso en una estructura de decisión switch.

else	Señala el código a ser ejecutado si la condición de una estructura de decisión if no es verdadera.
if	Inicia una estructura de decisión if, la cual es ejecutada si la condición es verdadera.
switch	Inicio de una estructura de decisión switch.
Palabras Reservadas Para el manejo de excepciones.	
catch	Maneja una excepción.
finally	Declara un bloque de código que se garantiza que siempre se va a ejecutar dentro de una estructura de excepción.
throw	Tira o dispara una excepción.
try	Intenta realizar una opearcion que podría tirar o disparar una excepción
Palabras Reservadas Para la declaración de clases.	
abstract	Declara que una clase o un método es abstracto.
class	Señala el inicio de una declaración de una clase.
default	La acción a realizar por default en una estructura de decisión switch.
extends	Especifica la clase de la cual se va a heredar.
implements	Declara que esta clase implementa una interface dada.
instanceof	Verifica si un objeto es una instancia de una clase dada.
interface	Señala el inicio de una definición de una inteface
Palabras Reservadas Para modificadores y control de acceso.	
final	Declara que una clase no puede ser heredada, o que un método o una variable miembro no pueden ser sobrescritos.
native	Declara que un método es implantado en código nativo.
new	Crea un nuevo objeto de una clase dada.
private	Declara que un método o variable miembro tendrá un control de acceso del tipo privado.
protected	Declara que una clase, método o variable miembro tendrá un control de acceso del tipo protegido
public	Declara que una clase, método o variable miembro tendrá un control de acceso del tipo público.
static	Declara que un método o variable pasará a formar parte de la clase en ves de ser parte de cada objeto, a estos tipos de métodos y variables se les llama estáticos.
synchonized	Indica que una sección de código debe tener un acceso sincronizado por parte de diferentes threads.
transient	Declara que un campo no debería ser serializado.
void	Declara que un método no retorna ningún tipo de valor.

Palabras Reservadas Para otras funciones.	
import	Permite el acceso a una clase o grupo de clases en un paquete determinado.
null	
package	Define el paquete en el cual será almacenada la clase.
return	Es utilizado para que un método retorne algún valor al lugar donde fue invocado.
super	Es una referencia a la clase padre del objeto en utilización.
this	Es una referencia al objeto en uso.

- Separadores.

Los separadores te ayudan a definir la estructura del programa, así como también en algunos casos a forzar cierta precedencia en la evaluación de expresiones. Entre sus usos te ayudan a delimitar el contenido de las clases de los métodos, etc.

Separador.	Propósito.
( )	Encierra los parámetros en la declaración de un método. Ajusta la precedencia en expresiones aritméticas. Delimita las condiciones en estructuras de control..
{ }	Define bloques de código. Agrupa los elementos que automáticamente inicializan un arreglo.
[ ]	Es usado en la declaración de arreglos así como en la manipulación de los mismos.
;	Finaliza una sentencia.
,	Separa sucesivos identificadores en una declaración. Encadena condiciones en un ciclo for.
.	Selecciona una variable miembro o un método de un objeto. Separa nombres de paquetes de subpaquetes y nombres de clases.

- Espacios en blanco.

En Java a excepción de las cadenas los espacios en blanco tienen el propósito de crear un código bien estructurado y legible, es decir, separando lo suficiente y alineando las sentencias del programa mediante el uso de espacios en blanco se logra tener una mejor organización del código fuente de los programas.

Para introducir espacios en blanco en las cadenas no suele haber mucho problema cuando se trata de espacios sencillos sin embargo cuando tratamos de incorporar tabulaciones, retornos de carro u otros caracteres que Java maneja de manera especial es cuando surge la necesidad de incorporar una forma de lograrlo, es así como entran en

escena las secuencias de escape que son formadas por el carácter de escape \ y algún otro carácter que define la operación:

\b	Backspace
\t	Tabulador
\n	Linefeed
\f	Formfeed
\r	Carriage return
\"	Comillas dobles
\'	Comillas simples
\\	backslash

- Identificadores.

Un identificar es un nombre dado a un elemento de un programa. Así tenemos que en Java los identificadores son las palabras reservadas, los nombres dados a las variables, métodos, clases, paquetes e interfaces creados por el programador o provenientes dentro del J.D.K.

Las reglas para crear identificadores validos en un programa Java son:

- Los identificadores deben estar compuestos solamente de letras, números, el signo de dólar "$" o el signo de underscore "_"
- Los identificadores solo pueden iniciar con una letra, el underscore o el signo de dólar pero nunca con un número.
- No deben ser palabras reservadas ni literales booleanas como true o false.
- La longitud no esta limitada.
- No pueden existir dos identificadores iguales en la misma zona de influencia.

A partir de estas reglas se pueden crear cualquier tipo de identificador sin embargo existen ciertas recomendaciones de cómo generar buenos identificadores:

- Los identificadores deben ser lo más descriptivo posible con respecto a lo que identifican.
- Los nombres de clases deben tener Mayúsculas al principio de cada palabra del identificador.
- Los nombres de métodos deben tener Mayúsculas al principio de cada palabra dentro del identificador , excepto la primera.
- Los nombres de variables deben tener Mayúsculas al principio de cada palabra dentro del identificador, excepto la primera.
- Los nombres de constantes deben tener Todo en mayúsculas y el carácter de underscore separando cada palabra.

- Literales.

Son piezas de código que significan exactamente lo que dicen. Existen diferentes tipos de literales y cada uno de ellos es relacionado con un tipo de dato en particular. Así como consecuencia tenemos tipos de literales enteras, de punto flotante, booleanas, de carácter, y de cadena.

Enteros:
Por ejemplo:
2 21 077 0xDC00
Reales en coma flotante:
Por ejemplo:
3.14 2e12 3.1E12
Booleanos:
true
false
Caracteres:
Por ejemplo:
'a' 'j' '\t' \u???? [????] número unicode
Cadenas:
Por ejemplo:
"Esto es una cadena literal"

Cuando se inserta un literal en un programa, el compilador normalmente sabe exactamente de qué tipo se trata. Sin embargo, hay ocasiones en la que el tipo es ambiguo y hay que guiar al compilador proporcionándole información adicional para indicarle exactamente de qué tipo son los caracteres que componen el literal que se va a encontrar. En el ejemplo siguiente se muestran algunos casos en que resulta imprescindible indicar al compilador el tipo de información que se le está proporcionando:

```
class literales {
  char c = 0xffff;  // mayor valor de char en hexadecimal
  byte b = 0x7f;    // mayor valor de byte en hexadecimal
  short s = 0x7fff; // mayor valor de short en hexadecimal
  int i1 = 0x2f;    // hexadecimal en minúsculas
  int i2 = 0X2F;    // hexadecimal en mayúsculas
  int i3 = 0177;    // octal (un cero al principio)
  long l1 = 100L;
  long l2 = 100l;
  long l3 = 200;
  float f1 = 1;
  float f2 = 1F;
  float f3 = 1f;
  float f4 = 1e-45f; // en base 10
  float f5 = 1e+9f;
  double d1 = 1d;
  double d2 = 1D;
```

```
double d3 = 47e47d; // en base 10
}
```

Un valor hexadecimal (base 16), que funciona con todos los tipos enteros de datos, se indica mediante un 0x o 0X seguido por 0-9 y a-f, bien en mayúsculas o minúsculas. Si se intenta inicializar una variable con un valor mayor que el que puede almacenar, el compilador generará un error. Por ejemplo, si se exceden los valores para char, byte y short que se indican en el ejemplo, el compilador automáticamente convertirá el valor a un int e indicará que se necesita un casting para la asignación.

Los números octales (base 8), se indican colocando un cero a la izquierda del número que se desee. No hay representación para números binarios ni en en C, ni en C++, ni en Java.

Se puede colocar un carácter al final del literal para indicar su tipo, ya sea una letra mayúscula o minúscula. L se usa para indicar un long, F significa float y una D mayúscula o minúscula es lo que se emplea para indicar que el literal es un double.

La exponenciación se indica con la letra e, tomando como referente la base 10. Es decir, que hay que realizar una traslación mental al ver estos números de tal forma que 1.3e-45f en Java, en la realidad es 1.3 x 10-45.

No es necesario indicarle nada al compilador cuando se puede conocer el tipo sin ambiguedades. Por ejemplo, en

```
long l3 = 200;
```

No es necesario colocar la L después de 200 porque resultaría de más. Sin embargo, en el caso:

```
float f4 = 1e-45f; // en base 10
```

sí es necesario indicar el tipo, porque el compilador trata normalmente los números exponenciales como double, por lo tanto, si no se coloca la f final, el compilador generará un error indicando que se debe colocar un cast para convertir el double en float.

Cuando se realizan operaciones matemáticas o a nivel de bits con tipos de datos básicos más pequeños que int (char, byte o short), esos valores son promocionados a int antes de realizar las operaciones y el resultado es de tipo int. Si se quiere seguir teniendo el tipo de dato original, hay que colocar un moldeo, teniendo en cuenta que al pasar de un tipo de dato mayor a uno menor, es decir, al hacer un moldeo estrecho, se puede perder información. En general, el tipo más grande en una expresión es el que determina el tamaño del resultado de la expresión; si se multiplica un float por un double, el resultado será un double, y si se suma un int y un long, el resultado será un long.

- Operadores.

Un operador es un símbolo que opera sobre uno o más argumentos (operandos). Un operador que actúa sobre un solo operando es un operador unario, y un operador que actúa sobre dos operandos es un operador binario. Existen distintos tipos de operadores y su propósito es ayudar al programador a hacer las operaciones que su programa necesita hacer.

1. Operadores Aritméticos

Java soporta varios operadores aritméticos que actúan sobre números enteros y números en coma flotante. Los operadores aritméticos binarios soportados por Java son:

Operador	Operación	Ejemplo
+	Suma los operandos.	G + H
-	Resta el operando de la derecha al de la izquierda	G – H
*	Multiplica los operandos	G * H
/	Divide el operando de la izquierda entre el de la derecha	G / H
%	Residuo de la división del operando izquierdo entre el derecho.	G % H

El operador más (+), se puede utilizar para concatenar cadenas, como se observa en el ejemplo siguiente:

"miVariable tiene el valor " + miVariable + " en este programa"

Los operadores aritméticos unarios que soporta Java son:

Operador	Operación	Ejemplo
+	Indica un valor positivo.	+ H
-	Negativo, o cambia el signo algebraico.	– H
++	Suma 1 al operando, como prefijo o sufijo.	
--	Resta 1 al operando, como prefijo o sufijo	

En los operadores de incremento (++) y decremento (--), en la versión prefijo, el operando aparece a la derecha del operador, ++x; mientras que en la versión sufijo, el operando aparece a la izquierda del operador, x++. La diferencia entre estas versiones es el momento en el tiempo en que se realiza la operación representada por el operador si éste y su operando aparecen en una expresión larga. Con la versión prefijo, la variable se incrementa (o decrementa) antes de que sea utilizada para evaluar la expresión en que se encuentre, mientras que en la versión sufijo, se utiliza la variable para realizar la evaluación de la expresión y luego se incrementa (o decrementa) en una unidad su valor.

2. Operadores Relacionales y Condicionales.

El lenguaje Java soporta los siguientes operadores relacionales. Los operadores relacionales en Java devuelven un tipo booleano, true o false.

Operador	Operación.	Ejemplo.	Significado.
>	Mayor que	G>H	¿Es G mayor que H?
>=	Mayor o igual que	G>=H	¿Es G mayor o igual que H?
<	Menor que	G<H	¿Es G menor que H?
<=	Menor o igual que	G<=H	¿Es G menor o igual que H?
==	Igual	G==H	¿Es G igual a H?
!=	Desigual	G!=H	¿Es G desigual a H?

Los operadores condicionales que soporta Java son:

Operador	Uso.	Retorna true si
&&	Op1 && op2	Op1 y op2 son ambos true
||	OP1 || op2	Si cualquiera de los dos Op1 o Op2 es true
!	!Op	Si Op es false

Al igual que los operadores relacionales los operadores condicionales devuelven valores booleanos.

Java soporta otro operador condicional. El operador ?: . Este operador es un operador terciario y es basicamente una versión corta de una sentencia if-else.

expresion ? op1 : op2

El operador ?: evalúa la expresión y devuelve op1 si la expresión es verdadera y op2 si la expresión es falsa.

3. Operadores de desplazamiento de bits.

Un operador de este tipo te permite hacer manipulaciones sobre los bits de los datos en forma individual.

Operador	Operación
&	AND a nivel de bits
|	OR a nivel de bits
^	XOR a nivel de bits
<<	Movimiento a la izquierda
>>	Movimiento a la derecha
~	Complemento a nivel de bits

4. Operadores de asignación.

El operador = es un operador binario de asignación de valores. El valor almacenado en la memoria y representado por el operando situado a la derecha del operador es copiado en la memoria indicada por el operando de la izquierda.

Java soporta otro tipo de operadores de asignación que se componen con otros operadores para realizar la operación que indique ese operador y luego asignar el valor obtenido al operando situado a la izquierda del operador de asignación. De este modo se pueden realizar dos operaciones con un solo operador.

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

Por ejemplo, las dos sentencias que siguen realizan la misma función:

```
x += y;
x = x + y;
```

- Precedencia de Operadores.

1.	Operadores postfix	[] . (params) expr++ expr--
2.	Operadores unarios	++expr --expr +expr -expr ~ !
3.	Creación o cast	new (type)expr
4.	Multiplicativos	* / %
5.	Aditivos	+ -
6.	Movimiento	<< >> >>>
7.	Relacionales	< > <= >= instanceof
8.	Igualdad	== !=
9.	AND de bits	&
10.	OR exclusiva de bits	^
11.	OR inclusiva de bist	|
12.	AND lógico	&&
13.	OR lógico	||
14.	Condicional	?:
15.	Asignación	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Cuando operadores de igual precedencia aparecen en la misma expresión, algunas reglas deben gobernar cual es evaluado primero. En Java, todos los operadores binarios excepto el operador de asignación son evaluados de izquierda a derecha. El operador de asignación es evaluado de derecha a izquierda.

- Variables.

El lenguaje Java permite declarar variables dentro de un programa. Estás se usan para contener datos que pueden cambiar durante la ejecución del programa. Todas las variables en Java tienen un tipo de dato, un identificador y un ámbito. Una variable siempre se debe declarar antes de usarse. Los componentes de una declaración de variable son los siguientes:

```
Tipo_de_dato  Identificador [= Literal del mismo Tipo ] { , Identificador [= Literal del mismo Tipo ] };
                              Valor de la variable                          Valor de la variable
```

Lo que se encuentra entre [ ] es opcional.
Lo que se encuentra entre { } significa que se puede repetir.

La localización de la declaración de la variable, es decir donde la declaración aparece con relación a otros elementos del código determina su ámbito.

- Identificador de una variable.

  Un programa se refiere al valor de una variable a través de su identificador. Los identificadores de las variables siguen las mismas reglas que cualquier otro identificador.

- Ambito de una variable.

  El ámbito de una variable es el bloque de código dentro del cual la variable es accesible y determina cuando la variable es creada y destruida. La localización de la declaración de la variable dentro del programa establece su ámbito el cual puede caer dentro de estas cuatro categorías:

  - Variable miembro.
  - Variable local.
  - Parámetro de un método.
  - Parámetro de un manejador de excepción.

Una variable miembro es un miembro de una clase o un objeto. Puede ser declarada en cualquier parte de la clase excepto dentro de los métodos de la clase. Las variables miembro son utilizables dentro de todo el código de la clase.

Las variables locales se pueden declarar en cualquier parte dentro de un método o dentro de un bloque de código en un método. En general una variable local es accesible desde el sitio donde se encuentra su declaración hasta el final del bloque de código donde se encuentra declarada.

Los otros tipos de ámbitos de variables serán analizados posteriormente.

- Inicialización de una variable.

  Tanto las variables locales como las variables miembro pueden ser inicializadas con una sentencia de asignación al momento de ser declaradas aunque esto es opcional. Los tipos de dato de ambos lados de la sentencia de asignación deben ser iguales. Los parámetros de un método y los parámetros de un manejador de excepción no pueden ser inicializados de esta forma, ya que los valores para los parámetros son puestos por el objeto que invoca a esos métodos.

- Variables final.

  Se puede declarar una variable en cualquier ámbito con un modificador especial llamado final. El valor de una variable final no puede ser cambiado después de que esta a sido inicializada.

  Para declarar una variable como final, debemos usar la palabra reservada final en la declaración de la variable, por ejemplo:

```
final int numero = 5;
```

  La sentencia anterior declara una variable como final y la inicializa. Todos los posteriores intentos de querer asignar un nuevo valor a la variable resultarán en un error de compilación. Sin embargo, si es necesario, podrías no inicializar una variable final al momento de declararla, solamente declararías la variable y la inicializarias después como en este ejemplo:

```
final int variableNoInicializada ;

. . .

variableNoInicializada = 0;
```

  Una variable final que ha sido declarada pero no inicializada y después se inicializa recibe la misma protección que cualquier otra variable final.

- Tipos de datos.

Un tipo de dato determina los valores que una variable puede contener, el rango de estos valores y las operaciones que pueden ser realizadas sobre ella. Por ejemplo, la declaración

int contador;

declara que contador es un entero (int), los enteros pueden contener solo valores de números enteros (tanto positivos como negativos), y se les pueden aplicar los operadores aritméticos (+,-,* y /) para realizar las operaciones aritméticas estándar (adición, sustracción, multiplicación y división respectivamente).

Existen dos clases de categorías de tipos de datos en Java: primitivos y de referencia.

Los tipos primitivos son los siguientes:

Tipo	Tamaño / Formato	Descripción	Rango
	Números	Enteros	
byte	8 bits complemento a 2	Enteros de longitud de 8 bits	-128 a 127
short	16 bits complemento a 2	Enteros cortos	-32768 a 32767
int	32 bits complemento a 2	Entero	-2147483648 a 2147483647
long	64 bits complemento a 2	Entero largo	-9223372036854775808 a 9223372036854775807
	Números	Reales	
float	32 bits IEEE 754	Números de punto flotante de precisión simple.	
double	64 bits IEEE 754	Números de punto flotante de precisión doble.	
	Otros	Tipos	
char	16 bits Caracteres Unicode	Un carácter sencillo	
bolean	true o false	Un valor booleano	true o false

Nota: En otros lenguajes, el formato y tipo de los datos primitivos es dependiente de la plataforma sobre la cual corre el programa. En contraste en Java el mismo lenguaje especifica el tamaño y formato de sus tipo de datos primitivos. Con lo que se elimina la dependencia de plataforma.

Una variable de tipo primitivo contiene un valor simple del tamaño y formato apropiado de su tipo: un numero, un carácter, o un valor booleano. Por ejemplo el valor de un int es un número entero, el valor de un char es un carácter de 16 bits Unicode y así con

los otros tipos. Los arreglos, clases e interfaces son tipos referencia. El valor de la variable de un tipo referencia, en contraste con los tipos primitivos, es una referencia al actual valor o conjunto de valores representados por la variable. Una referencia es como la dirección de una casa, es decir la dirección no es la casa, pero es la forma de encontrarla. Una variable de tipo referencia no es el arreglo o el objeto sino una forma de tener acceso a él.

- Expresiones.

Una expresión es una determinada combinación de operadores y operandos que se evalúan para obtener un resultado particular. Los operandos pueden ser variables, literales o llamadas a métodos.

Los operadores son usados en expresiones para operar sobre las operandos. Todo resultado de una expresión es un valor. Los operadores le dicen al compilador javac como manipular las variables y demás datos para dar el resultado apropiado.

Las expresiones no solo utilizan tipos de datos numéricos sino también char, arrays. etc. Ejemplos de expresiones son los siguientes:

```
Libro = 4
Resultado = (3 * 2 + 3) + (6 + 4 / 3)
KeyboardChar = (char) System.in.read( )
```

- Declaraciones.

Las declaraciones son un tipo especial de expresiones que definen el tipo de datos de una variable. A continuación hay unos ejemplos de declaraciones de variables de diferentes tipos:

Declaración de variables de números enteros.

```
byte ByteVariable; // 8 bits de tamaño
short ShortVariable; // 16 bits de tamaño
int IntVariable; // 32 bits de tamaño
long LongVar; // 64 bits de tamaño
```

Declaración de variables de números de punto flotante.

```
float FloatVariable; //32 bits de tamaño
double DoubleVariable; //64 bits de tamaño
```

Declaración de variables de tipo char.

```
char MiCaracter;  // Es capaz de almacenar un carácter.
char MiOtroCaracter = 'y'; //declara la variable MiOtroCaracter y le asigna
                           //la literal y
```

Declaración de variables booleanas.

```
boolean BooleanVariable; //Es capaz de almacenar el valor true o false.
boolean BoolanOtraVariable = true;
```

- Condiciones.

Las condiciones son un tipo de expresiones que están formadas por operadores relacionales, operadores lógicos, variables, literales, etc. Otra de sus características es que devuelven un valor de tipo booleano.

Este tipo de expresiones se utilizan para tomar decisiones dentro de los programas.

- Sentencias y bloques.

Una sentencia es una línea de código terminada por el terminador de ";". Es una operación completa entendida por el compilador de java. Una sentencia puede ser una expresión, una declaración, una condición u otro tipo de estructura.

Se pueden utilizar los separadores para agrupar un conjunto de sentencias y formar una única sentencia compuesta. Este tipo de sentencias compuestas se utilizan para tener mayor orden sobre el código pero sobre todo para que las estructuras de control tomen a este conjunto de sentencias como una sola al momento de ejecutar su lógica. A este tipo de agrupaciones de sentencias se les conoce como bloques.

- Bloques.

Un bloque es un grupo de sentencias que son consideradas en forma lógica como una sola sentencia. Un bloque esta delimitado por los separadores "{ y }"; El primero de estos símbolos inicia el bloque, y el segundo lo termina. Un bloque puede ser usado en cualquier lugar donde una sentencia es usada y es tratado en la misma forma.

Este último punto se podrá apreciar mejor cuando veamos las estructuras de control y la forma en como manipulan las sentencias.

- Estructuras de control.

Las expresiones, operadores, y separadores pueden ser combinados para manipular datos en una variedad de formas, sin embargo algo nos hace falta, la habilidad de tomar decisiones sobre el flujo de ejecución del programa. Estas decisiones conocidas como estructuras de control instruyen al programa sobre cuales sentencias deberá procesar en base a esta decisión

Las piezas que constituyen la forma de las estructuras de control son los delimitadores de bloques { y } así como las palabras reservadas if, while, do, for, switch, else, case, brake y continue

Estas sentencias le permiten a su programa hacer diferentes cosas dependiendo del resultado de la evaluación de una condición. Los resultados de estas evaluaciones son devueltos como booleanos.

- Sentencias de Decisión.

1. if/else

```
if(expresión-booleana-true)
   sentencia;
else
   sentencia;

if( expresión-booleana-true ) {
   sentencias;
}
else {
   sentencias;
}
```

Esta construcción evalua la expresión booleana y si el resultado es true ejecuta las sentencias dentro del if, en caso de ser false ejecuta las sentencias dentro del else. La cláusula else es opcional. Cada una de las sentencias puede ser una sentencia compuesta y la expresión-booleana podría ser una variable simple declarada como boolean, o una expresión que utilice operadores relacionales o lógicos (condición) para generar el resultado de una comparación.

2. switch

```
switch( expresión ) {
case valor1:
   sentencias;
   break;
case valor2:
   sentencias;
   break;
[default:
   sentencias;]
}
```

La sentencia switch proporciona una forma limpia de enviar la ejecución a partes diferentes del código en base al valor de una única variable o expresión. La expresión puede devolver cualquier tipo entero (byte short, int) o un tipo char, y cada uno de los valores especificados en las sentencias case debe ser de un tipo compatible.

En Java no se puede realizar una comprobación de un tipo numérico contra caracteres como en C++, porque el tipo char en C++ es en realidad un entero de 8 bits, lo que no es cierto en Java.

La sentencia switch funciona de la siguiente manera: el valor de la expresión se compara con cada uno de los valores literales de las sentencias case. Si coincide con alguno, se ejecuta el código que sigue a la sentencia case. Si no coincide con ninguno de ellos, entonces se ejecuta la sentencia default (por defecto), que es opcional. Si no hay sentencia default y no coincide con ninguno de los valores, no hace nada. Al igual que en otros lenguajes, cada valor de cada case debe ser único.

El compilador de Java inspeccionara cada uno de los valores que pueda tomar la expresión en base a las sentencias case que se proporcionen, y creará una tabla eficiente que utiliza para ramificar el control del flujo al case adecuado dependiendo del valor que tome la expresión. Por lo tanto, si se necesita seleccionar entre un gran grupo de valores, una sentencia switch se ejecutará mucho más rápido que la lógica equivalente codificada utilizado sentencias if-else.

La palabra clave break se utiliza habitualmente en sentencias switch, para que la ejecución salte tras el final de la sentencia switch. Si no se pone el break, la ejecución continuará en el siguiente case. Es un error habitual a la hora de programar el olvidar un break; dado que el compilador no avisa de dichas omisiones, es una buena idea poner un comentario en los sitios en los que normalmente se pondría el break, diciendo que la intención es que el case continúe en el siguiente, por convenio este comentario es simplemente, "Cóntinúa".

- Sentencias de Repetición.

1. Ciclos for

```
for( inicialización; terminación; iteración )
  sentencia;
```

o

```
for( inicialización; terminación; iteración ) {
  sentencias;
  }
```

Un bucle for, normalmente involucra a tres acciones en su ejecución:

Inicialización de la variable de control
Comprobación del valor de la variable de control en una expresión condicional
Actualización de la variable de control

La cláusula de inicio y la cláusula de incremento pueden estar compuestas por varias expresiones separadas mediante el operador coma (,), que en estos ciclos Java también soporta.

```
for( a=0,b=0; a < 7; a++,b+=2 )
```

El operador coma garantiza que el operando de su izquierda se ejecutará antes que el operando de su derecha. Las expresiones de la cláusula de inicio se ejecutan una sola vez, cuando arranca el ciclo.

Cualquier expresión legal se puede emplear en esta cláusula, aunque generalmente se utiliza para inicialización. Las variables se pueden declarar e inicializar al mismo tiempo en esta cláusula:

```
for( int cnt=0; cnt < 2; cnt++ )
```

La segunda cláusula, que es una condición, consiste en una única expresión que debe evaluarse a false para que el bucle concluya. En este caso, Java es mucho más restrictivo que C++, ya que en C, C++ cualquier expresión se puede evaluar a cero, que equivale a false. Sin embargo, en Java, esta segunda expresión debe ser de tipo booleano, de tal modo que se pueden utilizar únicamente expresiones relacionales o expresiones relacionales y condicionales, es decír condiciones.

El valor de la segunda cláusula es comprobado cuando la sentencia comienza la ejecución y en cada una de las iteraciones posteriores.

La tercera cláusula, de incremento, aunque aparece físicamente en la declaración de ciclo, no se ejecuta hasta que se han ejecutado todas las sentencias que

componen el cuerpo del bucle for; por ello, se utiliza para actualizar la variable de control. Es importante tener en cuenta que si utilizamos variables incluidas en esta tercera cláusula en las sentencias del cuerpo del bucle, su valor no se actualizará hasta que la ejecución de todas y cada una de las sentencias del cuerpo del bucle se haya completado. En esta cláusula pueden aparecer múltiples expresiones separadas por el operador coma, que serán ejecutadas de izquierda a derecha.

El siguiente trocito de código Java que dibuja varias líneas en pantalla alternando sus colores entre rojo, azul y verde, utiliza un bucle for para dibujar un número determinado de líneas y una sentencia switch para decidir el color de la línea. Este fragmento sería parte de una función Java (método):

```
int contador;
        for( contador=1; contador <=12; contador++ ) {
          switch( contador % 3) {
            case 0:
              setColor( Color.red );
              break;
            case 1:
              setColor( Color.blue );
              break;
            case 2:
              setColor( Color.green );
              break;
          }
          g.drawLine(10,contador*10,80,contador*10);
        }
```

También se soporta, como ya se ha indicado, el operador coma (,) en los ciclos for, aunque su uso es una decisión de estilo, no es la única forma de codificar una sentencia lógica en particular. En ocasiones se utiliza como atajo, pero en otras se prefiere la utilización de sentencias múltiples dentro del cuerpo del bucle for.

```
for( a=0,b=0; a < 7; a++,b+=2 )
```

La primera y tercera cláusulas del bucle for pueden encontrarse vacías, pero deben estar separadas por punto y coma (;). Hay autores que sugieren incluso que la cláusula de evaluación puede estar vacía, aunque para el programador que está escribiendo esto, salvando el caso de que se trate de implementar un ciclo infinito, si esta cláusula de comprobación se encuentra vacía, el método de terminación del ciclo no es nada obvio, al no haber una expresión condicional que evaluar, por lo que debería recurrirse a otro tipo de sentencia, en vez de utilizar un bucle for.

2. Ciclo while

```
[inicialización;]

while( expresión-booleana-true ) {
   sentencias;
   [iteración;]
   }
```

El ciclo while es la sentencia de bucle más básica en Java. Ejecuta repetidamente una vez tras otra una sentencia, mientras una expresión booleana sea verdadera. Las partes de inicialización e iteración, que se presentan entre corchetes, son opcionales.

Esta sentencia while se utiliza para crear una condición de entrada. El significado de esta condición de entrada es que la expresión condicional que controla el bucle se comprueba antes de ejecutar cualquiera de las sentencias que se encuentran situadas en el interior del bucle, de tal modo que si esta comprobación es false la primera vez, el conjunto de las sentencias no se ejecutará nunca.

Un ejemplo típico de utilización de este bucle es el cálculo de la serie de números de Fibonacci, cuyo código se muestra a continuación:

```
class Fibinacci {
        public static void main( String args[] ){
           int max = 20;
           int bajo = 1;
           int alto = 0;

           System.out.println( bajo );
           while( alto < 50 ) {
              System.out.println( alto );
              int temp = alto;
              alto = alto + bajo;
              bajo = temp;
              }
           }
        }
```

3. Bucles do/while

```
[inicialización;]
do {
   sentencias;
   [iteración;]
}while( expresión-booleana );
```

A veces se puede desear el ejecutar el cuerpo de un bucle while al menos una vez, incluso si la expresión booleana tiene el valor false la primera vez. Es decir, si se desea evaluar la expresión de terminación al final del bucle en vez de al principio como en el bucle while. Esta construcción do-while hace eso exactamente.

- Sentencias de Salto.

1. break

```
break [etiqueta];
```

La sentencia break puede utilizarse en una sentencia switch o en un ciclo. Cuando se encuentra en una sentencia switch, break hace que el control del flujo del programa pase a la siguiente sentencia que se encuentre fuera del entorno del switch. Si se encuentra en un ciclo, hace que el flujo de ejecución del programa deje el ámbito del ciclo y pase a la siguiente sentencia que venga a continuación del ciclo.

Java incorpora la posibilidad de etiquetar la sentencia break, de forma que el control pasa a sentencias que no se encuentran inmediatamente después de la sentencia switch o del ciclo, es decir, saltará a la sentencia en donde se encuentre situada la etiqueta. La sintaxis de una sentencia etiquetada es la siguiente:

```
etiqueta: sentencia;
```

2. continue

```
continue [etiqueta];
```

La sentencia continue no se puede utilizar en una sentencia switch, sino solamente en ciclos. Cuando se encuentra esta sentencia en la ejecución normal de un programa Java, la iteración en que se encuentre el ciclo finaliza y se inicia la siguiente.

Java permite el uso de etiquetas en la sentencia continue, de forma que el funcionamiento normal se ve alterado y el salto en la ejecución del flujo del programa se realizará a la sentencia en la que se encuentra colocada la etiqueta.

3.2 Introducción a P.O.O.

♦ Sistemas de Software.

En la práctica del desarrollo de programas, la gente empieza realizando aplicaciones de un tamaño moderado, que por lo regular son especificadas, construidas, mantenidas y usadas por la misma persona, que en la mayoría de las ocasiones trabaja en forma aislada, la utilización y periodo de existencia de éstos trabajos es limitada, en consecuencia podemos afirmar que este tipo de sistemas de software son poco complejos y que la utilización de alguna técnica de diseño en su construcción no es muy importante ya que debido a la naturaleza de ellos es viable y en cierto sentido más práctico reemplazarlos por nuevos sistemas de software en vez de intentar rehusar, reparar o extender la funcionalidad de los sistemas antiguos.

Sin embargo conforme crece el tamaño, la importancia y el conjunto de conductas que debe exhibir una aplicación, surgen sistemas que tienden a tener un periodo de vida más largo sobre el tiempo, así como también que muchos usuarios dependen de su buen funcionamiento para realizar su trabajo, en una palabra que sean realmente "COMPLEJOS", como por ejemplo, sistemas de control de vuelo, de control de procesos industriales, sistemas que mantienen la integridad y el acceso a cientos o miles de registros de información mientras se permiten concurrentes inspecciones y alteraciones a los datos, aplicaciones en ambientes distribuidos que además de sus servicios normales deben mantener ciertos estándares de seguridad, sistemas que modelan procesos físicos.

La característica fundamental de este tipo de software es que es intensamente difícil o imposible para un programador individual comprender todas las características de diseño y desarrollo de estos sistemas, de ahí que surja la necesidad de buscar mecanismos que nos ayuden a manejar y tratar esta complejidad inherente de los sistemas de software de dimensiones grandes y características importantes.

Se nos plantean las preguntas :

- ¿Cuáles son los atributos de los sistemas complejos ?
- ¿Cuál es la forma estructural de los sistemas complejos ?
- ¿Con qué mecanismos contamos para manejar la complejidad?
- ¿Cuáles paradigmas existen para la construcción de sistemas de software ?
- ¿Qué es la tecnología Orientada a Objetos?
- ¿Cómo utiliza la tecnología Orienta a Objetos los mecanismos para el manejo de la complejidad ?

- **Atributos de los sistemas complejos**

1. Frecuentemente, los sistemas complejos toman una forma jerárquica, en donde esta jerarquía esta compuesta por subsistemas interrelacionados que a su vez tienen sus propios subsistemas y así sucesivamente hasta el más bajo nivel de componentes elementarios.

2. El establecimiento de cuales son los componentes elementarios o primitivos en un sistema es una decisión relativamente arbitraria y por lo regular depende del punto de vista del observador así como de sus objetivos. (Diferentes tipos de objetos).

3. Las uniones de los componentes de un subsistema de un sistema complejo son más fuertes entre los componentes miembros de ese mismo subsistema que con los componentes miembros de otros subsistemas. (Cohesión).

4. Los subsistemas de un sistema complejo están compuestos de solo algunos tipos diferentes de componentes primitivos combinados en varios arreglos y formas. (En el caso de sistemas de software son los tokens).

5. Un sistema complejo que trabaja es indudable que evolucionó de sistemas más simples que también trabajaban.

- **Manejo de la Complejidad.**

- Descomposición.

Se basa en la idea "Divide y Vencerás", cuando desarrollamos sistemas de software complejos es esencial descomponerlos en pequeñas partes o subsistemas, cada una de las cuales pueda ser manejada hasta cierto grado en forma independiente, así de esta forma solo necesitamos entender solo algunas partes a la vez.

- Abstracción.

El hombre incapaz de manejar la total complejidad de un sistema ha ideado una forma de tratarla, se ha escogido ignorar los detalles menos significantes y solo trabajar con los de mayor importancia a nuestro punto de vista y establecer un modelo general e idealizado del sistema.

Puntualizando la abstracción es la descripción o especificación simplificada de un sistema que hace énfasis en algunos detalles o propiedades y suprime otros.

- Jerarquización.

Como habíamos visto cuando descomponemos un sistema lo dividimos en partes o subsistemas, sin embargo, como es la relación entre éstas y que organización guarda una con respecto a la otra, es aquí donde la jerarquización entra en juego ya que en base a ella podemos establecer las relaciones y con ello crear una estructura jerárquica que nos presente dependencias así como componentes comunes entre subsistemas, con lo cual, podemos hacer generalizaciones entre subsistemas y evitar redundancias.

- Encapsulación.

El encapsulamiento consiste en ocultar los detalles de implantación de un objeto, a la vez que provee una interfaz pública que contiene los únicos medios por los cuales se puede interactuar con este ente.

Aquí se maneja el concepto de caja negra, en donde no nos importa lo que haya dentro sino los mecanismos que existen para tratar con ella.

- Modularidad.

Es otro concepto muy relacionado con la descomposición y que algunos establecen que es el producto de ella, sin embargo, se aplica a una representación física del sistema y no lógica, lo que se obtiene son partes del sistema que se pueden manejar por separado de manera física.

♦ Paradigmas de construcción de software

Descomposición Algorítmica (Programación estructurada).

En este enfoque esta basada la programación estructurada y consiste en descomponer al dominio del problema en un conjunto de procesos y acciones a realizar, las cuales se estructuran en una jerarquía funcional de módulos que a su vez están formados por un conjunto de algoritmos. Otro aspecto que se toma en cuenta en esta forma de construir un sistema de software son los flujos de datos a través del sistema. Se hace una fuerte división entre los datos y las funciones que se aplican a los mismos. Por lo regular el lenguaje y los modelos utilizado en el dominio del problema deben ser trasladados al lenguaje y modelos de las herramientas de programación, lo que no es un proceso sencillo, y que además provoca un aumento de complejidad y es una fuente mayor de errores.

Descomposición Orientada a Objetos.

En este enfoque se ve al dominio del problema como un conjunto de objetos autónomos que colaboran entre sí para efectuar una tarea de más alto nivel, desde esta perspectiva un objeto es simplemente una entidad la cuál exhibe una conducta y estado bien definidos. Los objetos hacen cosas y se comunican con otros objetos en base a envió de mensajes para realizar acciones y completar trabajos.

Una idea fundamental detrás de esta nueva forma de realizar sistemas es la que se refiere al hecho de manejar el mismo lenguaje y los mismos modelos tanto en el dominio del problema como en el dominio de programación, es decir si por ejemplo vamos a realizar un sistema sobre una nómina y en ella manejamos objetos como personas, cheques, pagos, etc. , es necesario bajo este paradigma que las herramientas de programación nos brinden las facilidades para crear estos mismos objetos (tanto atributos como conducta) en el dominio de programación, es decir poder seguir hablando de personas. Cheques, pagos. etc., al momento de estar programando el sistema y no tener que estar utilizando solo tipos de datos primitivos para representar nuestros objetos reales. Todo esto nos lleva a la necesidad de poder crear nuestros propios tipos de datos (objetos) cuando hacemos programas utilizando el paradigma orientado a objetos y las herramientas que lo soportan.

En esta forma se ven a los sistemas como una colección de objetos, que cooperan entre sí para realizar las funciones del sistema, cada uno de los cuales representa una instancia de alguna clase, las cuáles se encuentran organizadas en una estructura jerárquica unidas vía relaciones de herencia.

Surge la duda de que paradigma se debe utilizar para la construcción de sistemas de software, y como veremos posteriormente el enfoque Orientado a Objetos es considerado mejor ya que se adapta de una forma más natural a la forma en como percibimos al mundo y como estructuramos nuestro conocimiento.

"Tecnología Orientada a Objetos."

El ser humano ha desarrollado una gran capacidad para clasificar, generalizar y abstraer objetos, de tal forma que puede tratar con el complejo mundo que le rodea. De un mundo lleno de perros individuales, hemos desarrollado el concepto de perro como una clase abstracta que sintetiza los atributos y el comportamiento que todos los perros del mundo comparten. Esto nos permite desarrollar ideas acerca de los perros, sin pensar en los detalles de un perro en particular. Esta es la forma como aplicamos la capacidad de clasificar, generalizar y abstraer.

En O.O., el mundo de un problema a resolver se compone entonces de objetos altamente especializados que cooperan entre sí para lograr un objetivo. En esos mundos OO, el conocimiento se descentraliza en todos los objetos que lo componen, cada objeto sabe hacer lo suyo y no le interesa saber cómo el vecino hace su trabajo, pero sabe que lo hace y qué es lo que puede hacer.

Como bien lo definió Dan Ingalls de Smalltalk con las siguientes palabras:
"La orientación a objetos proporciona una solución que conduce a un Universo de Objetos 'bien educados' que se piden de manera cortés, concederse mutuamente sus deseos".

¿Por qué O.O.?

La tendencia O.O. se inició como una búsqueda de un paradigma que permitiera tratar los cada vez más complejos problemas de ingeniería de software, de una forma más natural, más práctica, y, sobre todo, que brindara la posibilidad de llegar en el campo del software, a lo que ya se tiene en otras ingenierías: una verdadera industria. La meta es dejar la etapa en la que la construcción del software es una labor de artesanos, y pasar a la etapa en la que se pueda tener fábricas de software, con gran capacidad de reutilización de código y con metodologías eficientes y efectivas que se apliquen al proceso de producción.

Aunque la descentralización y el aislamiento del conocimiento especializado ha sido planteado y usado de diversas formas por tecnologías diferentes a la O.O., es esta la que ofrece un soporte integral y completo a estos conceptos, permitiendo a los desarrolladores obtener mejores sistemas.

Entonces, O.O. por:

- Mantenimiento.

La altá modularidad facilita la extensión del software. Hay mayor facilidad para realizar los cambios.

•Rápido Desarrollo.

Por la alta reutilización que se puede hacer. La idea es industrializar la construcción de software, a través del uso intensivo de librerías de clases (hechas en casa o compradas).

Esto también acorta el tiempo de pruebas, ya que los objetos de estas librerías han sido previamente probados y se puede confiar en su buen funcionamiento. Pero, hay reutilización también en la herencia, que permite extender o especializar clases de objetos previamente desarrollados.

•Equipo de Desarrollo.

Como los objetos son entidades independientes que existen por sí solas, con un comportamiento conocido, ellos pueden ser desarrollados y probados separadamente.

•Diseño Conceptual.

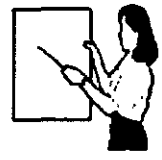
Los modelos O.O. se aproximan a la realidad, tratando de reflejar en el modelo los objetos que se perciben en el problema. El modelamiento es, por tanto, más intuitivo y fácil de comprender.

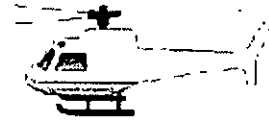
"Conceptos O.O."

- Objeto (Instancia de Clase).

Un Objeto es cualquier cosa, real o abstracta, que posee atributos y un conjunto de operaciones que manipulan esos atributos; atributos y operaciones le dan al objeto un comportamiento particular. Por ejemplo, en una biblioteca podemos distinguir los objetos biblioteca, libros, revistas, usuarios, ficheros, fichas, etc., ejemplos de objetos abstractos pueden ser los números.


Antena Parabólica


Objeto Profesor


Helicóptero

a + b*i*
Número Complejo


Cuenta Bancaria

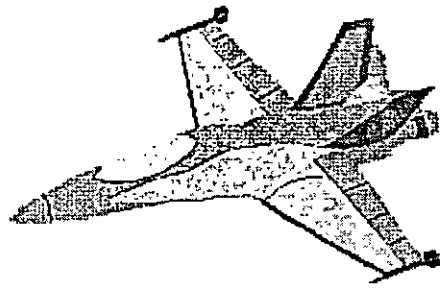

Automóvil

- Identidad de un Objeto.

El mundo está dividido en entidades discretas y distinguibles llamadas OBJETOS. Identidad es la propiedad única de un objeto que lo distingue sin ambigüedad de los otros objetos en el universo en el curso de todo el tiempo. Piense, por ejemplo, en la computadora que usa para leer este documento; esta computadora es única en el universo en este instante de tiempo, y no existió antes de él otra computadora que fuera ella, y no existirá en el futuro otra computadora que sea ella; en el mundo orientado por objetos no existe la reencarnación.

- Atributos.

Los atributos son las características de un objeto. Son un conjunto de datos (valores) y calificadores para aquellos datos. Por ejemplo, para un objeto de la clase libro, uno de sus atributos es Título (calificador) cuyo valor es Cien Años de Soledad; para un objeto de la clase persona, uno de sus atributos es Color de Ojos (calificador), cuyo valor es Verde.



velocidad... color... tamaño... coste...

- Operaciones, Métodos y Mensajes.

Una Operación es una función que un objeto ejecuta o una transformación a la que se ve sujeto.

Un Método es una implantación particular de una operación. Una operación puede tener varias implantaciones, es decir, puede tener varios métodos que la representan.

Un Mensaje es el mecanismo por el cual se invoca una operación de un objeto. Es la forma como se le solicita a un objeto que ejecute una de sus operaciones.

- Estado.—

El estado de un objeto está dado por el conjunto de valores de sus atributos en un instante dado.

Los objetos son dinámicos, los valores de sus atributos (su estado) pueden cambiar durante su período de vida.

•Comportamiento.

El comportamiento es la forma como actúa o reacciona un objeto en términos de cambio de estado, envío y recepción de mensajes. Esta formado por la definición de las operaciones que puede realizar ese objeto. Los tipos más comunes de operaciones son :

1. modificar.
2. seleccionar.
3. construir.
4. destruir.

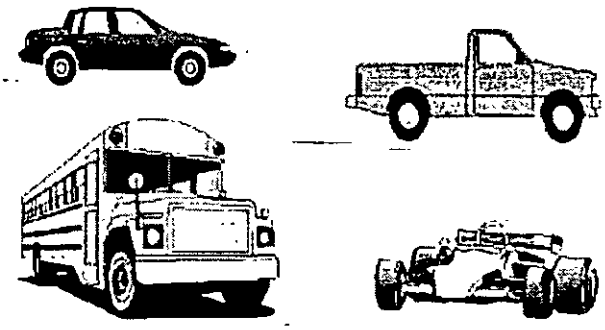
Las operaciones y estado de un objeto son sus miembros o partes

- Constructores y Destructores.

Existen dos operaciones especiales que todo objeto debe poseer: una constructora y una destructora. La constructora es la operación que se invoca automáticamente en el momento de instanciar un objeto, para propósitos de inicialización de sus atributos. La destructura es la operación que se invoca cuando un objeto deja de existir, y su propósito es liberar memoria (generalmente).

- Clase.----

Objetos con la misma estructura de datos (atributos) y que exhiben un mismo comportamiento son agrupados en CLASES.

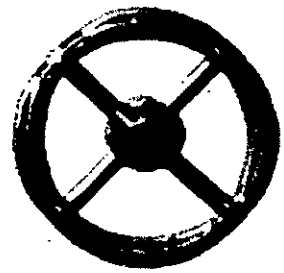


Objetos de la clase **Coche**

Una clase se asimila a un molde de hacer galletas; a partir de ese molde se puede crear infinidad de objetos. Al proceso de crear objetos a partir de la clase se denomina instanciar. Los objetos son, entonces, instancias de una clase.

Por otro lado las clases se ven como el tipo de dato del objeto, por ejemplo mi perro bob es del tipo de clase perro.

La clase tiene dos vistas : la exterior o interfaz, en la cual se hace énfasis en la abstracción y se oculta la estructura y secretos de comportamiento y solo se muestra la forma en como se puede interactuar con el objeto; y la vista interior o implantación en donde se hace énfasis en los algoritmos que constituyen los métodos del objeto y las estructuras de datos que soportan los atributos del mismo. Aquí se nota que es indispensable hacer uso del concepto de encapsulación.



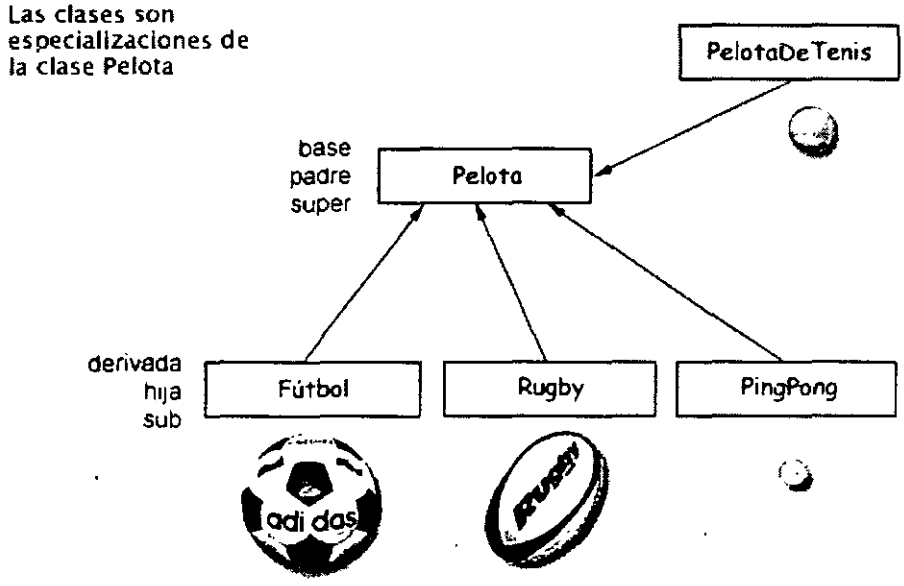
El volante representa un interfaz público hacia el mecanismo de giro de un coche

La implementación del volante es privada y sobre ella sólo puede actuar el propio volante

- Generalización y Herencia (Tipo de).

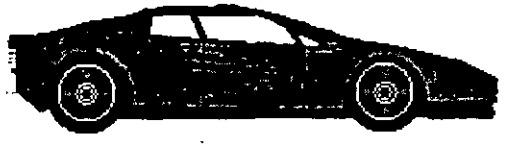
Al proceso de "factorizar" de varias clases,(a las que llamaremos Subclases) las características comunes y dárselas a otra clase (a la que llamaremos Superclase) se denomina Generalizar; el mecanismo que permite a las subclases poseer las características de una superclase se denomina Herencia.

Por ejemplo podemos agrupar las clases perro, gato, hombre y abstraer sus características comunes dándoselas a una clase llamada mamíferos que en este caso sería la generalización. Y si lo vemos en forma inversa usaríamos la herencia y tomaríamos a mamíferos como una superclase que gracias al mecanismos de herencia la podemos heredar para crear clases más especializadas como son: perro, gato y hombre, las cuales poseerían todos los atributos y operaciones de la clase mamíferos y aumentarían los atributos y operaciones específicos de su clase.



- Composición (es parte de).

Un objeto puede ser "ensamblado" a partir de otros objetos. Por ejemplo, un carro posee llantas, motor, puertas, etc. Esta relación entre objetos se conoce como "es parte de", porque en nuestro ejemplo el motor es parte del carro. Para implementar esta relación, generalmente los objetos que son parte de uno mayor son atributos de este.



- Polimorfismo.

Polimorfismo significa múltiples formas; en programación orientada por objetos esto significa que una operación puede tener más de un método que la implanta.

En otras palabras cuando existen operaciones que pueden ser aplicadas a diferentes tipos de datos u objetos, es decir que pueden recibir distintos tipos de parámetros se debe hacer uso de esta característica. Un ejemplo claro en Java es el operador "+" que puede ser aplicado a números ya sea enteros o de coma flotante así como a cadenas.

Por lo regular existen dos formas de implantar el Polimorfismo:

- Sobrecarga.
- Sobreescritura.

Cada uno de ellos es utilizado en diferente tipo de situaciones, cuando lleguemos al apartado siguiente veremos como es que se usan estas dos formas.

La propiedad de polimorfismo utiliza un truco para poder ser implantada en los lenguajes de programación, se decide en tiempo de ejecución que método se invoca. Este concepto se conoce como LATE BINDING, o lo que es lo mismo, encadenamiento tardío. En lenguajes tradicionales estructurados estamos acostumbrados al EARLY BINDING, o encadenamiento temprano, lo que significa que el encadenador sabe resolver todos los llamados a rutinas en tiempo de compilación/encadenamiento; el código con Late Binding puede ser más "lento", pero mucho más poderoso que el código con Early Binding.

- Clases Abstractas.

Hay ocasiones, cuando estamos definiendo una clase, en que sabemos que la clase debe tener una operación, sabemos QUE HACE la operación, pero NO sabemos COMO LO HACE. Por ejemplo, cuando estamos definiendo la clase FiguraGeometrica, sabemos que a todas las figuras geométricas se les puede calcular el área, se les puede calcular el perímetro, se pueden dibujar; pero no hay un método general para hallar el área o dibujar cualquier figura geométrica.

Para este tipo de problemas, las metodologías O.O. nos permiten definir clases Abstractas, que no son más que Clases con operaciones a las cuales no se les ha provisto método de implantación (estas operaciones se denominan operaciones abstractas) y solo son declaradas. Como estas clases poseen operaciones sin método, su definición no es completa, y por lo tanto no se pueden sacar instancias directas de estas clases. El objetivo de su existencia es, entonces, definir un comportamiento mínimo que deben poseer las subclases suyas y dejar el problema de implantación para las subclases, que es en donde se puede atacar.

Una clase concreta es una clase completamente definida y que por tanto puede tener instancias directas. Así, para que una subclase sea concreta, debe proveer los métodos de todas las operaciones abstractas de sus superclases.

3.3 Conceptos de P.O.O. en Java.

- **Creación de Objetos**

Un objeto es (de nuevo) una instancia de una clase. Tanto en Java como en C++, la creación de un objeto se realiza en tres pasos (aunque se pueden combinar):

Declaración: proporcionar un nombre al objeto
Instanciación: asignar memoria al objeto
Inicialización: opcionalmente se pueden proporcionar valores iniciales a las variables de instancia del objeto

Cuando se trata de Java, es importante reconocer la diferencia entre objetos y variables de tipos básicos. Para instanciar objetos a partir de clases en Java se utiliza el operador new.

La síntaxis es la siguiente:

```
Tipo_de_clase identificador_de_objeto = new Constructor de la clase(parámetros);
```

- **Constructor.**

Tanto Java como C++ soportan la sobrecarga de métodos, es decir, que dos o más métodos puedan tener el mismo
nombre, pero distinta lista de argumentos en su invocación. Si se sobrecarga un método, el compilador determinará
ya en tiempo de compilación, en base a lista de argumentos con que se llame al método, cual es la versión del
método que debe utilizar.

Java soporta la noción de constructor. El constructor es un tipo específico de método que siempre tiene el mismo nombre que la clase y se utiliza para construir objetos de esa clase. No tiene tipo de dato específico de retorno, ni siquiera void. Esto se debe a que el tipo específico que debe devolver un constructor de clase es el propio tipo de la clase.

En este caso, pues, no se puede especificar un tipo de retorno, ni se puede colocar ninguna sentencia que devuelva un valor. Los constructores pueden sobrecargarse, y aunque puedan contener código, su función primordial es inicializar el nuevo objeto que se instancia de la clase. En C++, el constructor se invoca automáticamente a la hora de crear un objeto. En Java, ha de hacerse una llamada explícita al constructor para instanciar un nuevo objeto.

Cuando se declara una clase en Java, se pueden declarar uno o más constructores opcionales que realizan la inicialización cuando se instancia (se crea una ocurrencia) un objeto de dicha clase.

La palabra clave new se usa para crear una instancia de la clase. Antes de ser instanciada con new no consume memoria, simplemente es una declaración de tipo.

En Java, cuando se instancia un objeto, siempre se hace una llamada directa al constructor como argumento del operador new. Este operador se encarga de que el sistema proporcione memoria para contener al objeto que se va a crear.

En Java como en C++, si no se proporciona explícitamente un constructor, el sistema proporciona uno por defecto que inicializará automáticamente todas las variables miembro a cero o su equivalente, en Java.

Se puede pensar en el constructor de defecto en Java como un método que tiene el mismo nombre que la clase y una lista de argumentos vacía.

Las dos sentencias siguientes muestran cómo se utiliza el constructor en Java para declarar, instanciar y, opcionalmente, inicializar un objeto:

```
MiClase miObjeto = new MiClase();
MiClase miObjeto = new MiClase( 1,2,3 );
```

Las dos sentencias devuelven una referencia al nuevo objeto que es almacenada en la variable miObjeto. También se puede invocar al constructor sin asignar la referencia a una variable. Esto es útil cuando un método requiere un objeto de un tipo determinado como argumento, ya que se puede incluir una llamada al constructor de este objeto en la llamada al método:

```
miMetodo( new MiConstructor( 1,2,3 ) );
```

Aquí se instancia e inicializa un objeto y se pasa a la función. Para que el programa compile adecuadamente, debe existir una versión de la función que espere recibir un objeto de ese tipo como parámetro.

En el siguiente ejemplo, se ilustran algunos de los conceptos sobre constructores que se han planteado en esta sección.

- **Clases.**

Las clases son lo más simple de Java. Todo en Java forma parte de una clase, es una clase o describe como funciona una clase. El conocimiento de las clases es fundamental para poder entender los programas Java.Todas las acciones de los programas Java se colocan dentro del bloque de una clase o un objeto. Un objeto es una instancia de una clase. Todos los métodos se definen dentro del bloque de la clase, Java no soporta funciones o variables globales.

Esto puede despistar a los programadores de C++, que pueden definir métodos fuera del bloque de la clase, pero esta posibilidad es más un intento de no separarse mucho y ser compatible con C, que un buen diseño orientado a objetos. Así pues, el esqueleto de cualquier aplicación Java se basa en la definición de una clase.

Todos los datos básicos, como los enteros, se deben declarar en las clases antes de hacer uso de ellos. En C la unidad fundamental son los ficheros con código fuente, en Java son las clases. De hecho son pocas las sentencias que se pueden colocar fuera del bloque de una clase. La palabra clave import (equivalente al #include) puede colocarse al principio de un fichero, fuera del bloque de la clase. Sin embargo, el compilador reemplazará esa sentencia con el contenido del fichero que se indique, que consistirá, como es de suponer, en más clases.

La definición de una clase consta de dos partes, la declaración y el cuerpo, según la siguiente sintaxis:

```
DeclaracionClase {
  CuerpoClase
  }
```

sin el punto y coma (;) final característico de C++ para señalar el final de la definición de una clase.

La declaración de la clase indica al compilador el nombre de la clase, la clase de la que deriva (su superclase), los privilegios de acceso a la clase (pública, abstracta, final) y si la clase implementa o no, uno o varios interfaces. El nombre de la clase debe ser un identificador válido en Java. Por convención, muchos programadores Java empiezan el nombre de las clase Java con una letra mayúscula.

Cada clase Java deriva, directa o indirectamente, de la clase Object. La clase padre inmediatamente superior a la clase que se está declarando se conoce como superclass. Si no se especifica la superclase de la que deriva una clase, se entiende que deriva directamente de la clase Object (definida en el paquete java.lang).

En la declaración de una clase se utiliza la palabra clave extends para especificar la superclase, de la forma:

```
class MiClase extends SuperClase {
  // cuerpo de la clase
```

```
}
```

Los programadores C++ observarán que no se indica nada sobre los privilegios de acceso a la clase, es decir, si es pública, privada o protegida. O sea, la herencia de los métodos de acceso a una clase no se pueden utilizar en Java para modificar el control de acceso asignado a un miembro de la clase padre.

Al igual que en C++, una clase hereda las variables y métodos de su superclase y de la superclase de esa clase, etc.; es decir, de todo el árbol de jerarquía de clases desde la clase que estamos declarando hasta la raíz del árbol:
Object. En otras palabras, un objeto que es instanciado desde una clase determinada, contiene todas las variables y métodos de instancia definidos para esta clase y sus antecesores; aunque los métodos pueden ser modificados (sobreescritos) en algún lugar.

Una clase puede implementar uno o más interfaces, declarándose esto utilizando la palabra clave implements, seguida de la lista de interfaces que implementa, se paradas por coma (,), de la forma:

```
class MiClase extends SuperClase implements MiInterfaz,TuInterfaz {
  // cuerpo de la clase
  }
```

Cuando una clase indique que implementa un interfaz, se puede asegurar que proporciona una definición para todos y cada uno de los métodos declarados en ese interfaz; en caso contrario, el compilador generará errores al no poder resolver los métodos del interfaz en la clase que lo implementa.

Hay cierta similitud entre un interfaz y una clase abstracta, aunque las definiciones de métodos no están permitidas en un interfaz y sí se pueden definir en una clase abstracta. El propósito de los interfaces es proporcionar nombres, es decir, solamente declara lo que necesita implementar el interfaz, pero no cómo se ha de realizar esa implementación; es una forma de encapsulación de los protocolos de los métodos sin forzar al usuario a utilizar la herencia.

Cuando se implementa un interfaz, los nombres de los métodos de la clase deben coincidir con los nombres de los métodos que están declarados en es interfaz, todos y cada uno de ellos.

El cuerpo de la clase contiene las declaraciones, y posiblemente la inicialización, de todos los datos miembros, tanto variables de clase como variables de instancia, y la definición completa de todos los métodos.

Las variables pueden declararse dentro del cuerpo de la clase o dentro del cuerpo de un método de la clase. Sin embargo, éstas últimas no son variables miembro de la clase, sino variables locales del método en la que están declaradas.

Un objeto instanciado de una clase tiene un estado definido por el valor actual de las variables de instancia y un entorno definido por los métodos de instancia. Es típico de la programación orientada a objetos el restringir el acceso a las variables y proporcionar métodos que permitan el acceso a esas variables, aunque esto no es estrictamente necesario.

Alguna o todas las variables pueden declararse para que se comporten como si fuesen constantes, utilizando la palabra clave final.

Los objetos instanciados desde una clase contienen todos los métodos de instancia de esa clase y también todos los métodos heredados de la superclase y sus antecesores. Por ejemplo, todos los objetos en Java heredan, directa o indirectamente, de la clase Object; luego todo objeto contiene los miembros de la clase Object, es decir, la clase Object define el esta y entorno básicos para todo objeto Java. Esto difiere significativamente de C++, en donde no hay una clase central de la que se derive todo, por lo que no hay una definición de un estado básico en C++.

Las características más importantes que la clase Object cede a sus descendientes son:

Posibilidad de cooperación consigo o con otro objeto
Posibilidad de conversión automática a String
Posibilidad de espera sobde una variable de condición
Posibilidad de notificación a otros objetos del estado de la variable de condición

La sintaxis general de definición de clase se podría extender tal como se muestra en el siguiente diagrama, que debe tomarse solamente como referencia y no al pie de la letra:

```
NombreDeLaClase {
  // declaración de las variables de instancia
  // declaración de las variables de la clase
  metodoDeInstancia() {
    // variables locales
    // código
    }
  metodoDeLaClase() {
    // variables locales
    // código
    }
}
```

Tipos de Clases

Hasta ahora sólo se ha utilizado la palabra clave public para calificar el nombre de las clases que hemos visto, pero hay tres modificadores más. Los tipos de clases que podemos definir son:

- public

Las clases public son accesibles desde otras clases, bien sea directamente o por herencia, desde clases declaradas fuera del paquete que contiene a esas clases públicas, ya que, por defecto, las clases solamente son accesibles por otras clases declaradas dentro del mismo paquete en el que se han declarado. Para acceder desde otros paquetes, primero tienen que ser importadas. La sintaxis es:

```
public class miClase extends SuperClase implements miInterface,TuInterface {
  // cuerpo de la clase
  }
```

Aquí la palabra clave public se utiliza en un contexto diferente del que se emplea cuando se define internamente la clase, junto con private y protected.

- abstract

Una clase abstract tiene al menos un método abstracto. Una clase abstracta no se instancia, sino que se utiliza como clase base para la herencia.

- final

Una clase final se declara como la clase que termina una cadena de herencia, es lo contrario a una clase abstracta. Nadie puede heredar de una clase final. Por ejemplo, la clase Math es una clase final. Aunque es técnicamente posible declarar clases con varias combinaciones de public, abstract y final, la declaración de una clase abstracta y a la vez final no tiene sentido, y el compilador no permitirá que se declare una clase con esos dos modificadores juntos.

- synchronizable

Este modificador especifica que todos los métodos definidos en la clase son sincronizados, es decir, que no se puede acceder al mismo tiempo a ellos desde distintos threads; el sistema se encarga de colocar los flags necesarios para evitarlo. Este mecanismo hace que desde threads diferentes se puedan modificar las mismas variables sin que haya problemas de que se sobreescriban.

Si no se utiliza alguno de los modificadores expuestos, por defecto, Java asume que una clase es:

No final
No abstracta
Subclase de la clase Object

No implementa interfaz alguna

- **Variables miembro.**

Una clase en Java puede contener variables y métodos. Las variables pueden ser tipos primitivos como int, char, etc.
Los métodos son funciones.

Por ejemplo, en el siguiente trozo de código podemos observarlo:

```
public class MiClase {
    int i;

    public MiClase() {
        i = 10;
    }
    public void Suma_a_i(int j) {
        int suma;
        suma = i + j;
    }
}
```

La clase MiClase contiene una variable (i) y dos métodos, MiClase() que es el constructor de la clase y Suma_a_i(int j).

La declaración de una variable miembro aparece dentro del cuerpo de la clase, pero fuera del cuerpo de cualquier método de esa clase. Si se declara dentro de un métodos, será una variable local del método y no una variable miembro de la clase. En el ejemplo anterior, i es una variable miembro de la clase y suma es una variable local del método Suma_ a_i().

El tipo de una variable determina los valores que se le pueden asignar y las operaciones que se pueden realizar con ella.

El nombre de una variable ha de ser un identificador válido en Java. Por convenio, los programadores Java empiezan los nombres de variables con una letra minúscula, pero no es imprescindible. Los nombres de las variables han de ser únicos dentro de la clase y se permite que haya variables y métodos con el mismo nombre, a diferencia de C++, en donde se produce un error de compilación si se da este caso.

Java permite la inicialización de variables miembro de tipos primitivos en el momento de la declaración .

La sintaxis completa de la declaración de una variable miembro de una clase en Java sería:

[especificador_de_acceso][static][final][transient][volatile]tipo nombreVariable [= valor_inicial];

- Ambito de una Variable

Los bloques de sentencias compuestas en Java se delimitan con dos llaves. Las variables de Java sólo son válidas desde el punto donde están declaradas hasta el final de la sentencia compuesta que la engloba. Se pueden anidar estas sentencias compuestas, y cada una puede contener su propio conjunto de declaraciones de variables locales.

Sin embargo, no se puede declarar una variable con el mismo nombre que una de ámbito exterior.

El siguiente ejemplo intenta declarar dos variables separadas con el mismo nombre. En C y C++ son distintas, porque están declaradas dentro de ámbitos diferentes. En Java, esto es ilegal.

```
class Ambito {
    int i = 1; //ámbito exterior
    {          //crea un nuevo ámbito
      int i=2; //error de compilación
    }
}
```

- Variables de Instancia

La declaración de una variable miembro dentro de la definición de una clase sin anteponerle la palabra clave static, hace que sea una variable de instancia en todos los objetos de la clase. El significado de variable de instancia sería, más o menos, que cualquier objeto instanciado de esa clase contiene su propia copia de toda variable de instancia. Si se examinara la zona de memoria reservada a cada objeto de la clase, se encontraría la reserva realizada para todas las variables de instancia de la clase. En otras palabras, como un objeto es una instancia de una clase, y como cada objeto tiene su propia copia de un dato miembro particular de la clase, entonces se puede denominar a ese dato miembro como variable de instancia.

En Java, se accede a las variables de instancia asociadas a un objeto determinado utilizando el nombre del objeto, el
operador punto (.) y el nombre de la variable:

```
miObjeto.miVariableDeInstancia;
```

- Variables Estáticas

La declaración de un dato miembro de una clase usando static, crea una variable de clase o variable estática de la clase. El significado de variable estática es que todas las instancias de la clase (todos los objetos instanciados de la clase) contienen la mismas variables de clase o estáticas. En otras palabras, en un momento determinado se puede querer crear una clase en la que el valor de una variable de instancia sea el mismo (y de hecho sea la misma variable) para todos los objetos instanciados a partir de esa clase. Es decir, que exista una única copia de la variable de instancia, entonces es cuando debe usarse la palabra clave static.

```
class Documento extends Pagina {
    static int version = 10;
    }
```

El valor de la variable version será el mismo para cualquier objeto instanciado de la clase Documento. Siempre que un objeto instanciado de Documento cambie la variable version, ésta cambiará para todos los objetos.

Si se examinara en este caso la zona de memoria reservada por el sistema para cada objeto, se encontraría con que todos los objetos comparten la misma zona de memoria para cada una de las variables estáticas, por ello se llaman también variables de clase, porque son comunes a la clase, a todos los objetos instanciados de la clase.

En Java, se puede acceder a las variables de clase utilizando el nombre de la clase y el nombre de la variable, no es necesario instanciar ningún objeto de la clase para acceder a las variables de clase.

En Java, a las variables de clase se accede utilizando el nombre de la clase, el nombre de la variable y el operador punto (.). La siguiente línea de código, ya archivista, se utiliza para acceder a la variable out de la clase System.

En el proceso, se accede al método println() de la variable de clase que presenta una cadena en el dispositivo estándar de salida.

```
System.out.println( "Hola, Mundo" );
```

Es importante recordar que todos los objetos de la clase comparten las misma variables de clase, porque si alguno de ellos modifica alguna de esas variables de clase, quedarán modificadas para todos los objetos de la clase. Esto puede utilizarse como una forma de comunicación entre objetos.

- Constantes

En Java, se utiliza la palabra clave final para indicar que una variable debe comportarse como si fuese constante, significando con esto que no se permite su modificación una vez que haya sido declarada e inicializada.

Como es una constante, se le ha de proporcionar un valor en el momento en que se declare, por ejemplo:

```
class Elipse {
  final float PI = 3.14159;
  . . .
  }
```

Si se intenta modificar el valor de una variable final desde el código de la aplicación, se generará un error de compilación.

Si se una la palabra clave final con una variable o clase estática, se pueden crear constantes de clase, haciendo de esto modo un uso altamente eficiente de la memoria, porque no se necesitarían múltiples copias de las constantes.

La palabra clave final también se puede aplicar a método s, significando en este caso que los métodos no pueden ser sobreescritos.

- **Métodos.**

Los métodos son funciones que pueden ser llamadas dentro de la clase o por otras clases. La implementación de un método consta de dos partes, una declaración y un cuerpo. La declaración en Java de un método se puede expresar esquemáticamente como:

```
tipoRetorno nombreMetodo( [lista_de_argumentos] ) {
  cuerpoMetodo
  }
```

En Java, la definición completa del método debe estar dentro de la definición de la clase y no se permite la posibilidad de métodos inline, por lo tanto, Java no proporciona al programador distinciones entre métodos normales y métodos inline.

Los métodos pueden tener numerosos atributos a la hora de declararlos, incluyendo el control de acceso, si es estático o no estático, etc. Los parámetros, o argumentos, se utilizan para pasar información al cuerpo del método.

La sintaxis de la declaración completa de un método es la que se muestra a continuación con los items opcionales en itálica y los items requeridos en negrilla:

especificadorAcceso static abstract final native synchronized tipoRetorno
nombreMetodo(lista_de_argumentos) throws listaEscepciones

EspecificadorAcceso, determina si otros objetos pueden acceder al método y cómo pueden hacerlo. Está soportado en Java y en C++, pero la sintaxis e interpretación es considerablemente diferente.

static, indica que los métodos pueden ser accedidos sin necesidad de instanciar un objeto del tipo que determina la clase. C++ y Java son similares en el soporte de esta característica.

abstract, indica que el método no está definido en la clase, sino que se encuentra declarado ahí para ser definido en una subclase (sobreescrito). C++ también soporta esta capacidad con una sintaxis diferente a Java, pero con similar interpretación.

final, evita que un método pueda ser sobreescrito.

native, son métodos escritos es otro lenguaje. Java soporta actualmente C y C++.

synchronized, se usa en el soporte de multithreading.

lista_de_argumentos, es la lista opcional de parámentros que se pueden pasar al método

throws listaExcepciones, indica las excepciones que puede generar y manipular el método. También se verán en este Tutorial a fondo las excepciones en Java.

- Valor de Retorno de un Método

En Java es imprescindible que a la hora de la declaración de un método, se indique el tipo de dato que ha de devolver.
Si no devuelve ningún valor, se indicará el tipo void como retorno.

Los métodos y funciones en C++ pueden devolver una variable u objeto, bien sea por valor (se devuelve una copia), por puntero o por referencia. Java no soporta punteros, así que no puede devolver nada por puntero. Todos los tipos primitivos en Java se devuelven por valor y todos los objetos se devuelven por referencia. El retorno de la referencia a un objeto en Java es similar a devolver un puntero a un objeto situado en memoria dinámica en C++, excepto que la sintaxis es mucho más simple en Java, en donde el item que se devuelve es la dirección de la posición en memoria dinámica donde se encuentra almacenado el objeto.

Para devolver un valor se utiliza la palabra clave return. La palabra clave return va seguida de una expresión que será evaluada para saber el valor de retorno. Esta expresión puede ser compleja o puede ser simplemente el nombre de un objeto, una variable de tipo primitivo o una constante.

El siguiente ejemplo ilustra el retorno por valor y por referencia.

```
// Un objeto de esta clase sera devuelto por referencia
class miClase {
   int varInstancia = 10;
   }
```

Si un programa Java devuelve una referencia a un objeto y esa referencia no es asignada a ninguna variable, o utilizada en una expresión, el objeto se marca inmediatamente para que el reciclador de memoria en su siguiente ejecución devuelve la memoria ocupada por el objeto al sistema, asumiendo que la dirección no se encuentra ya almacenada en ninguna otra variable. En C++, si un programa devuelve un puntero a un objeto situado en memoria dinámica y el valor de ese puntero no se asigna a una variable, la posibilidad de devolver la memoria al sistema se pierde y se producirá un memory leak, asumiendo que la dirección no está ya disponible para almacenar ninguna otra variable.

Tanto en Java como en C++ el tipo del valor de retorno debe coincidir con el tipo de retorno que se ha indicado en la declaración del método; aunque en Java, el tipo actual de retorno puede ser una subclase del tipo que se ha indicado en la declaración del método, lo cual no se permite en C++. En Java esto es posible porque todas las clases heredan desde un objeto raíz común a todos ellos: Object.

En general, se permite almacenar una referencia a un objeto en una variable de referencia que sea una superclase de ese objeto. También se puede utilizar un interfaz como tipo de retorno, en cuyo caso, el objeto retornado debe implementar dicho interfaz.

- Nombre del Método

El nombre del método puede ser cualquier identificador legal en Java. Java soporta el concepto de sobrecarga de métodos, es decir, permite que dos métodos compartan el mismo nombre pero con diferente lista de argumentos, de forma que el compilador pueda diferenciar claramente cuando se invoca a uno o a otro, en función de los parámetros que se utilicen en la llamada al método.

El siguiente fragmento de código muestra una clase Java con cuatro métodos sobrecargados, el último no es legal porque tiene el mismo nombre y lista de argumentos que otro previamente declarado:

```
class MiClase {
  . . .
  void miMetodo( int x,int y ) { . . . }
  void miMetodo( int x ) { . . . }
  void miMetodo( int x,float y ) { . . . }
  // void miMetodo( int a,float b ) { . . . } // no válido
  }
```

Todo lenguaje de programación orientado a objetos debe soportar las características de encapsulación, herencia y polimorfismo. La sobrecarga de métodos es considerada por algunos autores como polimorfismo en tiempo de compilación.

En Java, los métodos sobrecargados siempre deben devolver el mismo tipo.

- Métodos de Instancia

Cuando se incluye un método en una definición de una clase Java sin utilizar la palabra clave static, estamos generando un método de instancia. Aunque cada objeto de la clase no contiene su propia copia de un método de instancia (no existen múltiples copias del método en memoria), el resultado final es como si fuese así, como si cada objeto dispusiese de su propia copia del método.

Cuando se invoca un método de instancia a través de un objeto determinado, si este método referencia a variables de instancia de la clase, en realidad se están referenciando variables de instancia específicas del objeto específico que se está invocando.

La llamada a los métodos de instancia en Java se realiza utilizando el nombre del objeto, el operador punto y el nombre del método.

```
miObjeto.miMetodoDeInstancia();
```

Los métodos de instancia tienen acceso tanto a las variables de instancia como a las variables de clase, tanto en Java como en C++.

- Métodos Estáticos

Cuando una función es incluida en una definición de clase C++, o un método e incluso en una definición de una clase Java, y se utiliza la palabra static, se obtiene un método estático o método de clase.

Lo más significativo de los métodos de clase es que pueden ser invocados sin necesidad de que haya que instanciar ningún objeto de la clase. En Java se puede invocar un método de clase utilizando el nombre de la clase, el operador punto y el nombre del método.

```
MiClase.miMetodoDeClase();
```

En Java, los métodos de clase operan solamente como variables de clase; no tienen acceso a variables de instancia de la clase, a no ser que se cree un nuevo objeto y se acceda a las variables de instancia a través de ese objeto.

Si se observa el siguiente trozo de código de ejemplo:

```
class Documento extends Pagina {
   static int version = 10;
   int numero_de_capitulos;
   static void annade_un_capitulo() {
      numero_de_capitulos++; // esto no funciona
      }
   static void modifica_version( int i ) {
      version++;          // esto si funciona
      }
   }
```

la modificación de la variable numero_de_capitulos no funciona porque se está violando una de las reglas de acceso al intentar acceder desde un método estático a una variable no estática.

Todas las clases que se derivan, cuando se declaran estáticas, comparten la misma página de variables; es decir, todos los objetos que se generen comparten la misma zona de memoria. Los métodos estáticos se usan para acceder solamente a variables estáticas.

```
class UnaClase {
   int var;
   UnaClase() {
      var = 5;
      }
   unMetodo() {
      var += 5;
      }
   }
```

En el código anterior, si se llama al método unMetodo() a través de un puntero a función, no se podría acceder a var, porque al utilizar un puntero a función no se pasa implícitamente el puntero al propio objeto (this). Sin embargo, sí se podría acceder a var si fuese estática, porque siempre estaría en la misma posición de memoria para todos los objetos que se creasen de la clase UnaClase.

- Paso de parámetros

En Java, todos los métodos deben estar declarados y definidos dentro de la clase, y hay que indicar el tipo y nombre de los argumentos o parámetros que acepta. Los argumentos son como variables locales declaradas en el cuerpo del método que están inicializadas al valor que se pasa como parámetro en la invocación del método.

En Java, todos los argumentos de tipos primitivos deben pasarse por valor, mientras que los objetos deben pasarse por referencia. Cuando se pasa un objeto por referencia, se está pasando la dirección de memoria en la que se encuentra almacenado el objeto.

Si se modifica una variable que haya sido pasada por valor, no se modificará la variable original que se haya utilizado para invocar al método, mientras que sis se modifica una variable pasada por referencia, la variable original del método de llamada se verá afectada de los cambios que se produzcan en el método al que se le ha pasado como argumento.

En Java, los métodos tienen acceso directo a las variables miembro de la clase. El nombre de un argumento puede tener el mismo nombre que una variable miembro de la clase. En este caso, la variable local que resulta del argumento del método, oculta a la variable miembro de la clase.

Cuando se instancia un método se pasa siempre una referencia al propio objeto que ha llamado al método, es la referencia this.

- **this**

Al acceder a variables de instancia de una clase, la palabra clave this hace referencia a los miembros de la propia clase en el objeto actual; es decir, this se refiere al objeto actual sobre el que está actuando un método determinado y se utiliza siempre que se quiera hace referencia al objetoactual de la clase. Volviendo al ejemplo de MiClase, se puede añadir otro constructor de la forma siguiente:

```
public class MiClase {

  int i;
  public MiClase() {
    i = 10;
    }
  // Este constructor establece el valor de i
  public MiClase( int valor ) {
    this.i = valor; // i = valor ;
    }
  // Este constructor también establece el valor de i
```

```
public MiClase( int i ) {
   this.i = i;
   }
public void Suma_a_i( int j ) {
   i = i + j;
   }
}
```

Aquí this.i se refiere al entero i en la clase MiClase, que corresponde al objeto actual. La utilización de this en el tercer constructor de la clase, permite referirse directamente al objeto en sí, en lugar de permitir que el ámbito actual defina la resolución de variables, al utilizar i como parámetro formal y después this para acceder a la variable de instancia del objeto actual.

La utilización de this en dicho contexto puede ser confusa en ocasiones, y algunos programadores procuran no utilizar variables locales y nombres de parámetros formales que ocultan variables de instancia. Una filosofía diferente dice que en los métodos de inicialización y constructores, es bueno seguir el criterio de utilizar los mismos nombres por claridad, y utilizar this para no ocultar las variables de instancia. Lo cierto es que es más una cuestión de gusto personal que otra cosa el hacerlo de una forma u otra.

- **super**

Si se necesita llamar al método padre dentro de una clase que ha reemplazado ese método, se puede hacer referencia al método padre con la palabra clave super:

```
import MiClase;
public class MiNuevaClase extends MiClase {
   public void Suma_a_i( int j ) {
      i = i + ( j/2 );
      super.Suma_a_i( j );
   }
}
```

En el siguiente código, el constructor establecerá el valor de i a 10, después lo cambiará a 15 y finalmente el método Suma_a_i() de la clase padre MiClase lo dejará en 25:

```
MiNuevaClase mnc;
mnc = new MiNuevaClase();
mnc.Suma_a_i( 10 );
```

super es un concepto que no existe en C++, al menos no con una implementación similar a Java. Si un método sobreescribe un método de su superclase, se puede utilizar la palabra clave super para eludir la versión sobreescrita de la clase e invocar a la versión original del método en la supreclase. Del mismo modo, se puede utilizar super para acceder a variables miembro de la superclase.