LENGUAJES DE PROGRAMACIÓN

(Sesión 4)

2. PROGRAMACIÓN ORIENTADA A OBJETOS

2.3. El lenguaje de programación Java

2.4. Otros lenguajes orientados a objetos

Objetivo: Comparar la estructura general de JAVA con algunos otros lenguajes de programación también orientados a objetos.

Paradigma:	Orientado a objetos
Apareció en:	1991
Diseñado	Sun Microsystems
Tipo de dato:	Fuerte, Estático
Implementacio	Numerosas
Influido por:	Objective-C, C++,
Ha influido a:	C#, J#, JavaScript,PHP
Sistema	Multiplataforma
Licencia de	GNU GPL / Java

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 90. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria.

Las aplicaciones Java están típicamente compiladas en un *bytecode*, aunque la compilación en código máquina nativo también es posible. En el tiempo de ejecución, el *bytecode* es normalmente interpretado o compilado a código nativo para la ejecución, aunque la ejecución directa por hardware del *bytecode* por un procesador Java también es posible.

La implementación original y de referencia del compilador, la máquina virtual y las bibliotecas de clases de Java fueron desarrolladas por Sun Microsystems en 1995. Desde entonces, Sun ha controlado las especificaciones, el desarrollo y evolución del lenguaje a través del Java Community Process, si bien otros han desarrollado también implementaciones alternativas de estas tecnologías de Sun, algunas incluso bajo licencias de software libre.

Entre noviembre de 2006 y mayo de 2007, Sun Microsystems liberó la mayor parte de sus tecnologías Java bajo la licencia GNU GPL, de acuerdo con las especificaciones del Java Community Process, de tal forma que prácticamente todo el Java de Sun es ahora software libre (aunque la biblioteca de clases de Sun que se requiere para ejecutar los programas Java aún no lo es).Historia

La tecnología Java se creó como una herramienta de programación para ser usada en un proyecto de set-top-box en una pequeña operación denominada *the Green Project* en Sun Microsystems en el año 1991. El equipo (*Green Team*), compuesto por trece personas y dirigido por James Gosling, trabajó durante 18 meses en Sand Hill Road en Menlo Park en su desarrollo.

El lenguaje se denominó inicialmente *Oak* (por un roble que había fuera de la oficina de Gosling), luego pasó a denominarse *Green* tras descubrir que *Oak* era ya una marca comercial registrada para adaptadores de tarjetas gráficas y finalmente se renombró a *Java*.

El término Java fue acuñado en una cafetería frecuentada por algunos de los miembros del equipo. Pero no está claro si es un acrónimo o no, aunque algunas fuentes señalan que podría tratarse de las iniciales de sus creadores: *James Gosling, Arthur Van Hoff, y Andy Bechtolsheim*. Otros abogan por el siguiente acrónimo, *Just Another Vague Acronym* ("sólo otro acrónimo ambiguo más"). La hipótesis que más fuerza tiene es la que Java debe su nombre a un tipo de café disponible en la cafetería cercana, de ahí que el icono de java sea una taza de cafe caliente.

Un pequeño signo que da fuerza a esta teoría es que los 4 primeros bytes (el *número mágico*) de los archivos .class que genera el compilador, son en hexadecimal, 0xCAFEBABE. Otros simplemente dicen que el nombre fue sacado al parecer de una lista aleatoria de palabras.

Los objetivos de Gosling eran implementar una máquina virtual y un lenguaje con una estructura y sintaxis similar a C++. Entre junio y julio de 1994, tras una sesión maratoniana de tres días entre John Gaga, James Gosling, Joy Naughton, Wayne Rosing y Eric Schmidt, el equipo reorientó la plataforma hacia la Web. Sintieron que la llegada del navegador web Mosaic, propiciaría que Internet se convirtiese en un medio interactivo, como el que pensaban era la televisión por cable. Naughton creó entonces un prototipo de navegador, WebRunner, que más tarde sería conocido como HotJava.

En 1994, se les hizo una demostración de HotJava y la plataforma Java a los ejecutivos de Sun. Java 1.0a pudo descargarse por primera vez en 1994, pero hubo que esperar al 23 de mayo de 1995, durante las conferencias de SunWorld, a que vieran la luz pública Java y HotJava, el navegador Web. El acontecimiento fue anunciado por John Gage, el Director Científico de Sun Microsystems. El acto estuvo acompañado por una pequeña sorpresa adicional, el anuncio por parte de Marc Andreessen, Vicepresidente Ejecutivo de Netscape, que Java sería soportado en sus navegadores. El 9 de enero del año siguiente, 1996, Sun fundó el grupo empresarial JavaSoft para que se encargase del desarrollo tecnológico. Dos semanas más tarde la primera versión de Java fue publicada.

La promesa inicial de Gosling era *Write Once, Run Anywhere* (Escríbelo una vez, ejecútalo en cualquier lugar), proporcionando un lenguaje independiente de la plataforma y un entorno de ejecución (la JVM) ligero y gratuito para las plataformas más populares de forma que los binarios (bytecode) de las aplicaciones Java pudiesen ejecutarse en cualquier plataforma.

El entorno de ejecución era relativamente seguro y los principales navegadores web pronto incorporaron la posibilidad de ejecutar applets Java incrustadas en las páginas web.

Java ha experimentado numerosos cambios desde la versión primigenia, JDK 1.0, así como un enorme incremento en el número de clases y paquetes que componen la biblioteca estándar.

Desde J2SE 1.4, la evolución del lenguaje ha sido regulada por el JCP (Java Community Process), que usa *Java Specification Requests* (JSRs) para proponer y especificar cambios en la plataforma Java. El lenguaje en sí mismo está especificado en la *Java Language Specification* (JLS), o Especificación del Lenguaje Java.

- J2SE 1.4 (6 de febrero de 2002) — Nombre Clave *Merlin*. Este fue el primer lanzamiento de la plataforma Java desarrollado bajo el Proceso de la Comunidad Java como JSR 59. Los cambios más notables fueron: comunicado de prensa lista completa de cambios

- J2SE 5.0 (30 de septiembre de 2004) — Nombre clave: *Tiger*. (Originalmente numerado 1.5, esta notación aún es usada internamente.[16]) Desarrollado bajo JSR 176, Tiger añadió un número significativo de nuevas características comunicado de prensa

- Java SE 6 (11 de diciembre de 2006) — Nombre clave *Mustang* . Estuvo en desarrollo bajo la JSR 270

- Java SE 7 — Nombre clave *Dolphin*. En el año 2006 aún se encontraba en las primeras etapas de planificación.

Además de los cambios en el lenguaje, con el paso de los años se han efectuado muchos más cambios dramáticos en la biblioteca de clases de Java (*Java class library*) que ha crecido de unos pocos cientos de clases en JDK 1.0 hasta más de tres mil en J2SE 5.0. APIs completamente nuevas, como Swing y Java2D, han sido introducidas y muchos de los métodos y clases originales de JDK 1.0 están obsoletas.

En el 2005 se calcula en 4,5 millones el número de desarrolladores y 2.500 millones de dispositivos habilitados con tecnología Java.

Filosofía

El lenguaje Java se creó con cinco objetivos principales:

1. Debería usar la metodología de la programación orientada a objetos.
2. Debería permitir la ejecución de un mismo programa en múltiples sistemas operativos.
3. Debería incluir por defecto soporte para trabajo en red.
4. Debería diseñarse para ejecutar código en sistemas remotos de forma segura.
5. Debería ser fácil de usar y tomar lo mejor de otros lenguajes orientados a objetos, como C++.

Para conseguir la ejecución de código remoto y el soporte de red, los programadores de Java a veces recurren a extensiones como CORBA (Common Object Request Broker Architecture),
Internet Communications Engine o OSGi respectivamente.

Orientado a Objetos

La primera característica, orientado a objetos (“OO”), se refiere a un método de programación y al diseño del lenguaje. Aunque hay muchas interpretaciones para OO, una primera idea es diseñar el software de forma que los distintos tipos de datos que usen estén unidos a sus operaciones. Así, los datos y el código (funciones o métodos) se combinan en entidades llamadas objetos.

Un objeto puede verse como un paquete que contiene el “comportamiento” (el código) y el “estado” (datos). El principio es separar aquello que cambia de las cosas que permanecen inalterables. Frecuentemente, cambiar una estructura de datos implica un cambio en el código que opera sobre los mismos, o viceversa.

Esta separación en objetos coherentes e independientes ofrece una base más estable para el diseño de un sistema software. El objetivo es hacer que grandes proyectos sean fáciles de gestionar y manejar, mejorando como consecuencia su calidad y reduciendo el número de proyectos fallidos.

Otra de las grandes promesas de la programación orientada a objetos es la creación de entidades más genéricas (objetos) que permitan la reutilización del software entre proyectos, una de las premisas fundamentales de la Ingeniería del Software. Un objeto genérico “cliente”, por ejemplo, debería en teoría tener el mismo conjunto de comportamiento en diferentes proyectos, sobre todo cuando estos coinciden en cierta medida, algo que suele suceder en las grandes organizaciones.

En este sentido, los objetos podrían verse como piezas reutilizables que pueden emplearse en múltiples proyectos distintos, posibilitando así a la industria del software a construir proyectos de envergadura empleando componentes ya existentes y de comprobada calidad; conduciendo esto finalmente a una reducción drástica del tiempo de desarrollo. Podemos usar como ejemplo de objeto el aluminio. Una vez definidos datos (peso, maleabilidad, etc.), y su “comportamiento” (soldar dos piezas, etc.), el objeto “aluminio” puede ser reutilizado en el campo de la construcción, del automóvil, de la aviación, etc.

La reutilización del software ha experimentado resultados dispares, encontrando dos dificultades principales: el diseño de objetos realmente genéricos es pobremente comprendido, y falta una metodología para la amplia comunicación de oportunidades de reutilización. Algunas comunidades de "código abierto" (open source) quieren ayudar en este problema dando medios a los desarrolladores para diseminar la información sobre el uso y versatilidad de objetos reutilizables y bibliotecas de objetos.

Independencia de la plataforma

La segunda característica, la independencia de la plataforma, significa que programas escritos en el lenguaje Java pueden ejecutarse igualmente en cualquier tipo de hardware. Este es el significado de ser capaz de escribir un programa una vez y que pueda ejecutarse en cualquier dispositivo, tal como reza el axioma de Java, '"write once, run everywhere"'.

Para ello, se compila el código fuente escrito en lenguaje Java, para generar un código conocido como "bytecode" (específicamente Java bytecode)—instrucciones máquina simplificadas específicas de la plataforma Java. Esta pieza está "a medio camino" entre el código fuente y el código máquina que entiende el dispositivo destino. El bytecode es ejecutado entonces en la máquina virtual (JVM), un programa escrito en código nativo de la plataforma destino (que es el que entiende su hardware), que interpreta y ejecuta el código. Además, se suministran bibliotecas adicionales para acceder a las características de cada dispositivo (como los gráficos, ejecución mediante hebras o threads, la interfaz de red) de forma unificada. Se debe tener presente que, aunque hay una etapa explícita de compilación, el bytecode generado es interpretado o convertido a instrucciones máquina del código nativo por el compilador JIT (Just In Time).

Hay implementaciones del compilador de Java que convierten el código fuente directamente en código objeto nativo, como GCJ. Esto elimina la etapa intermedia donde se genera el bytecode, pero la salida de este tipo de compiladores sólo puede ejecutarse en un tipo de arquitectura.

La licencia sobre Java de Sun insiste que todas las implementaciones sean "compatibles". Esto dio lugar a una disputa legal entre Microsoft y Sun, cuando éste último alegó que la implementación de Microsoft no daba soporte a las interfaces RMI y JNI además de haber añadido características "dependientes" de su plataforma. Sun demandó a Microsoft y ganó por daños y perjuicios (unos 20 millones de dólares) así como una orden judicial forzando la acatación de la licencia de Sun. Como respuesta, Microsoft no ofrece Java con su

versión de sistema operativo, y en recientes versiones de Windows, su navegador Internet Explorer no admite la ejecución de applets sin un conector (o plugin) aparte. Sin embargo, Sun y otras fuentes ofrecen versiones gratuitas para distintas versiones de Windows.

Las primeras implementaciones del lenguaje usaban una máquina virtual interpretada para conseguir la portabilidad. Sin embargo, el resultado eran programas que se ejecutaban comparativamente más lentos que aquellos escritos en C o C++. Esto hizo que Java se ganase una reputación de lento en rendimiento. Las implementaciones recientes de la JVM dan lugar a programas que se ejecutan considerablemente más rápido que las versiones antiguas, empleando diversas técnicas, aunque sigue siendo mucho más lento que otros lenguajes.

La primera de estas técnicas es simplemente compilar directamente en código nativo como hacen los compiladores tradicionales, eliminando la etapa del bytecode. Esto da lugar a un gran rendimiento en la ejecución, pero tapa el camino a la portabilidad. Otra técnica, conocida como compilación JIT (Just In Time, o '''compilación al vuelo'''), convierte el bytecode a código nativo cuando se ejecuta la aplicación. Otras máquinas virtuales más sofisticadas usan una
'''recompilación dinámica''' en la que la VM es capaz de analizar el comportamiento del programa en ejecución y recompila y optimiza las partes críticas.

La recompilación dinámica puede lograr mayor grado de optimización que la compilación tradicional (o estática), ya que puede basar su trabajo en el conocimiento que de primera mano tiene sobre el entorno de ejecución y el conjunto de clases cargadas en memoria. La compilación JIT y la recompilación dinámica permiten a los programas Java aprovechar la velocidad de ejecución del código nativo sin por ello perder la ventaja de la portabilidad en ambos.

La portabilidad es técnicamente difícil de lograr, y el éxito de Java en ese campo ha sido dispar. Aunque es de hecho posible escribir programas para la plataforma Java que actúen de forma correcta en múltiples plataformas de distinta arquitectura, el gran número de estas con pequeños errores o inconsistencias llevan a que a veces se parodie el eslogan de Sun, "Write once, run anywhere" como "Write once, debug everywhere" (o “Escríbelo una vez, ejecútalo en cualquier parte” por “Escríbelo una vez, depúralo en todas partes”)

El concepto de independencia de la plataforma de Java cuenta, sin embargo, con un gran éxito en las aplicaciones en el entorno del servidor, como los Servicios Web, los Servlets, los Java Beans, así como en sistemas empotrados basados en OSGi, usando entornos Java empotrados.

El recolector de basura

Un argumento en contra de lenguajes como C++ es que los programadores se encuentran con la carga añadida de tener que administrar la memoria solicitada dinámicamente de forma manual:

En C++, el desarrollador puede asignar memoria en una zona conocida como *heap* (montículo)

para crear cualquier objeto, y posteriormente desalojar el espacio asignado cuando desea borrarlo.

Un olvido a la hora de desalojar memoria previamente solicitada puede llevar a una *fuga de memoria*, ya que el sistema operativo seguirá pensando que esa zona de memoria está siendo usada por una aplicación cuando en realidad no es así. Así, un programa mal diseñado podría consumir una cantidad desproporcionada de memoria. Además, si una misma región de memoria es desalojada dos veces el programa puede volverse inestable y llevar a un eventual *cuelgue*. No obstante, se debe señalar que C++ también permite crear objetos en la pila de llamadas de una función o bloque, de forma que se libere la memoria (y se ejecute el destructor del objeto) de forma automática al finalizar la ejecución de la función o bloque.

En Java, este problema potencial es evitado en gran medida por el recolector automático de basura (o *automatic garbage collector*). El programador determina cuándo se crean los objetos y el entorno en tiempo de ejecución de Java (Java runtime) es el responsable de gestionar el ciclo de vida de los objetos.

El programa, u otros objetos pueden tener localizado un objeto mediante una referencia a éste (que, desde un punto de vista de bajo nivel es una dirección de memoria). Cuando no quedan referencias a un objeto, el recolector de basura de Java borra el objeto, liberando así la memoria que ocupaba previniendo posibles fugas (ejemplo: un objeto creado y únicamente usado dentro de un método sólo tiene entidad dentro de éste; al salir del método el objeto es eliminado). Aun así, es posible que se produzcan fugas de memoria si el código almacena referencias a objetos que ya no son necesarios—es decir, pueden aún ocurrir, pero en un nivel conceptual superior. En definitiva, el recolector de basura de Java permite una fácil creación y eliminación de objetos, mayor seguridad y puede que más rápida que en C++ [*cita requerida*].

La recolección de basura de Java es un proceso prácticamente invisible al desarrollador. Es decir, el programador no tiene conciencia de cuándo la recolección de basura tendrá lugar, ya que ésta no tiene necesariamente que guardar relación con las acciones que realiza el código fuente. Debe tenerse en cuenta que la memoria es sólo uno de los muchos recursos que deben ser gestionados.

Sintaxis

La sintaxis de Java se deriva en gran medida de C++. Pero a diferencia de éste, que combina la sintaxis para programación genérica, estructurada y orientada a objetos, Java fue construido desde el principio para ser completamente orientado a objetos. Todo en Java es un objeto (salvo algunas excepciones), y todo en Java reside en alguna clase (recordemos que una clase es

un molde a partir del cual pueden crearse varios objetos).

Hola Mundo

Aplicaciones autónomas

```
//
Hola.java
public class
Hola
{
    public static void main(String[] args)throws
        IOException { System.out.println("¡Hola,
        mundo!");
    }
}
```

Este ejemplo necesita una pequeña explicación.

- Todo en Java está dentro de una clase, incluyendo programas autónomos.
- El código fuente se guarda en archivos con el mismo nombre que la clase que contienen y con extensión ".java".

  Una clase (class) declarada pública (public) debe seguir este convenio. En el ejemplo anterior, la clase es Hola, por lo que el código fuente debe guardarse en el fichero "Hola.java"
- El compilador genera un archivo de clase (con extensión ".class") por cada una de las clases definidas en el archivo fuente. Una clase anónima se trata como si su nombre fuera la concatenación del nombre de la clase que la encierra, el símbolo "$", y un número entero.
- Los programas que se ejecutan de forma independiente y autónoma, deben contener el método "main()".
- La palabra reservada "void" indica que el método main no devuelve nada.
- El método main debe aceptar un array de objetos tipo String. Por acuerdo se referencia como "args", aunque puede emplearse cualquier otro identificador.
- La palabra reservada "static" indica que el método es un método de clase, asociado a la clase en vez de a instancias de la misma. El método main debe ser estático o "de clase".
- La palabra reservada public significa que un método puede ser llamado desde otras clases, o que la clase puede ser usada por clases fuera de la jerarquía de la propia

clase. Otros tipos de acceso son ”private” o ”protected”.

- La utilidad de impresión (en pantalla por ejemplo) forma parte de la biblioteca estándar de Java: la clase “’System’” define un campo público estático llamado “’out’”. El objeto out es una instancia de “’PrintStream’”, que ofrece el método “’println (String)’” para volcar datos en la pantalla (la salida estándar).
- Las aplicaciones autónomas se ejecutan dando al entorno de ejecución de Java el nombre de la clase cuyo método main debe invocarse. Por ejemplo, una línea de comando (en Unix o Windows) de la forma java - cp .
  Hola ejecutará el programa del ejemplo (previamente compilado y generado “Hola.class”) . El nombre de la clase cuyo método main se llama puede especificarse también en el fichero “MANIFEST” del archivo de empaquetamiento de Java (.jar).

Applets

Las applets de Java son programas incrustados en otras aplicaciones, normalmente una página
Web que se muestra en un navegador.

```java
// Hola.java
import java.applet.Applet;
import java.awt.Graphics;


public class Hola extends Applet {
    public void paint(Graphics gc) {
        gc.drawString("Hola, mundo!", 65, 95);
    }
}
```

```html
 <!-- Hola.html -->
<html>
  <head>
    <title>Applet Hola Mundo</title>
  </head>
  <body>
    <applet code="Hola" width="200" height="200">
    </applet>
  </body>
</html
>
```

La sentencia **import** indica al compilador de Java que incluya las clases java.applet. Applet y java.awt. Graphics, para poder referenciarlas por sus nombres, sin tener que anteponer la ruta completa cada vez que se quieran usar en el código fuente.

La clase **Hola** extiende (**extends**) a la clase **Applet**, es decir, es una subclase de ésta. La clase **Applet** permite a la aplicación mostrar y controlar el estado del applet. La clase **Applet** es un componente del AWT (Abstract Window Toolkit), que permite al applet mostrar una inteterfaz gráfica de usuario o GUI (Graphical User Interface), y

responder a eventos generados por el usuario.

La clase Hola sobrecarga el método paint (Graphics) heredado de la superclase contenedora (Applet en este caso), para acceder al código encargado de dibujar. El método paint() recibe un objeto Graphics que contiene el contexto gráfico para dibujar el applet. El método paint() llama al método drawString (String, int, int) del objeto Graphics para mostrar la cadena de caracteres Hola, mundo! en la posición (65, 96) del espacio de dibujo asignado al applet.

La referencia al applet es colocada en un documento HTML usando la etiqueta <applet>. Esta etiqueta o tag tiene tres atributos: code="Hola" indica el nombre del applet, y width="200" height="200" establece la anchura y altura, respectivamente, del applet. Un applet también pueden alojarse dentro de un documento HTML usando los elementos object, o embed, aunque el soporte que ofrecen los navegadores Web no es uniforme.[30][31]

Servlets

Los servlets son componentes de la parte del servidor de Java EE, encargados de generar respuestas a las peticiones recibidas de los clientes.

```
// Hola.java
import java.io.*;
import javax.servlet.*;


public class Hola extends
GenericServlet
{
    public void service(ServletRequest request,
ServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/ht
        ml"); PrintWriter pw =
        response.getWriter();
        pw.println("Hola, mundo!");
        pw.close();
    }
}
```

Las sentencias import indican al compilador de Java la inclusión de todas las clases públicas e interfaces de los paquetes java.io y javax.servlet en la compilación.

La clase Hola extiende (extends), es heredera de la clase GenericServlet. Esta clase proporciona la interfaz para que el servidor le pase las peticiones al servlet y el mecanismo para controlar el ciclo de vida del servlet.

La clase Hola sobrecarga el método service (ServletRequest, ServletResponse), definido por la interfaz servlet para acceder al manejador de la petición de servicio.

El método service() recibe un objeto de tipo ServletRequest que contiene la petición del cliente y un objeto de tipo ServletResponse, usado para generar la respuesta que se devuelve al cliente. El método service() puede *lanzar* (throws) excepciones de tipo ServletException e IOException si ocurre algún tipo de anomalía.

El método setContentType (String) en el objeto respuesta establece el tipo de contenido MIME a "text/html", para indicar al cliente que la respuesta a su petición es una página con formato HTML. El método getWriter() del objeto respuesta devuelve un objeto de tipo PrintWriter, usado como una *tubería* por la que viajarán los datos al cliente. El método println (String) escribe la cadena "Hola, mundo!" en la respuesta y finalmente se llama al método close() para cerrar la conexión, que hace que los datos escritos en la tubería o stream sean devueltos al cliente.

Aplicaciones con ventanas

Swing es la biblioteca para la interfaz gráfica de usuario avanzada de la plataforma Java SE.

```
// Hola.java
import javax.swing.*;


public class Hola extends
    JFrame { Hola() {
        setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE)
        ;
        add(new JLabel("Hola, mundo!"));
        pack();
    }


    public static void main(String[] args) {
        new Hola().setVisible(true);
    }
}
```

Las instrucciones **import** indican al compilador de Java que las clases e interfaces del paquete javax.swing se incluyan en la compilación.

La clase **Hola** extiende (**extends**) la clase javax.swing.JFrame, que implementa una ventana con una barra de título y un control para cerrarla.

El constructor Hola() inicializa el marco o frame llamando al método setDefaultCloseOperation (int) heredado de JFrame para establecer las operaciones por defecto cuando el control de cierre en la barra de título es seleccionado al valor WindowConstants.DISPOSE_ON_CLOSE. Esto hace que se liberen los recursos tomados por la ventana cuando es cerrada, y no simplemente ocultada, lo que permite a la máquina virtual y al programa acabar su ejecución. A continuación se crea un objeto de tipo JLabel con el texto "Hola, mundo!", y se añade al marco mediante el método add (Component), heredado de la clase Container. El método pack(), heredado de la clase Window, es invocado para dimensionar la ventana y distribuir su contenido.

El método main() es llamado por la JVM al comienzo del programa. Crea una instancia de la clase Hola y hace la ventana sea mostrada invocando al método setVisible (boolean) de la superclase (clase de la que hereda) con el parámetro a true. Véase que, una vez el marco es dibujado, el programa no termina cuando se sale del método main(), ya que el código del que depende se encuentra en un hilo de ejecución independiente ya lanzado, y que permanecerá activo hasta que todas las ventanas hayan sido destruidas.

Entornos de funcionamiento

El diseño de Java, su robustez, el respaldo de la industria y su fácil portabilidad han hecho de Java uno de los lenguajes con un mayor crecimiento y amplitud de uso en distintos ámbitos de la industria de la informática.

En dispositivos móviles y sistemas empotrados

Desde la creación de la especificación J2ME (Java 2 Platform, Micro Edition), una versión del entorno de ejecución Java reducido y altamente optimizado, especialmente desarrollado para el mercado de dispositivos electrónicos de consumo se ha producido toda una revolución en lo que a la extensión de Java se refiere.

Es posible encontrar microprocesadores específicamente diseñados para ejecutar bytecode Java y software Java para tarjetas inteligentes (JavaCard), teléfonos móviles, buscapersonas, set-top-boxes, sintonizadores de TV y otros pequeños electrodomésticos.

El modelo de desarrollo de estas aplicaciones es muy semejante a las *applets* de los navegadores salvo que en este caso se denominan *MIDlets*.

En el navegador web

Desde la primera versión de java existe la posibilidad de desarrollar pequeñas aplicaciones (Applets) en Java que luego pueden ser incrustadas en una página HTML para que sean descargadas y ejecutadas por el navegador web. Estas mini-aplicaciones se ejecutan en una JVM que el navegador tiene configurada como extensión (*plug-in*) en un contexto de seguridad restringido configurable para impedir la ejecución local de código potencialmente malicioso.

El éxito de este tipo de aplicaciones (la visión del equipo de Gosling) no fue realmente el esperado debido a diversos factores, siendo quizás el más importante la lentitud y el reducido ancho de banda de las comunicaciones en aquel entonces que limitaba el tamaño de las applets que se incrustaban en el navegador. La aparición posterior de otras alternativas (aplicaciones web dinámicas de servidor) dejó un reducido ámbito de uso para esta tecnología, quedando hoy relegada fundamentalmente a componentes específicos para la intermediación desde una aplicación web dinámica de servidor con dispositivos ubicados en la máquina cliente donde se ejecuta el navegador.

Las *applets* Java no son las únicas tecnologías (aunque sí las primeras) de componentes complejos incrustados en el navegador. Otras tecnologías similares pueden ser: ActiveX de Microsoft, Flash, Java Web Start, etc.

En sistemas de servidor

En la parte del servidor, Java es más popular que nunca, desde la aparición de la especificación de
Servlets y JSP (Java Server Pages).

Hasta entonces, las aplicaciones web dinámicas de servidor que existían se basaban fundamentalmente en componentes CGI y lenguajes interpretados. Ambos tenían diversos inconvenientes (fundamentalmente lentitud, elevada carga computacional o de memoria y propensión a errores por su interpretación dinámica).

Los servlets y las JSPs supusieron un importante avance ya que:

- El API de programación es muy sencilla, flexible y extensible.

- Los servlets no son procesos independientes (como los CGIs) y por tanto se ejecutan dentro del mismo proceso que la JVM mejorando notablemente el rendimiento y reduciendo la carga computacional y de memoria requeridas.
- Las JSPs son páginas que se compilan dinámicamente (o se pre-compilan previamente a su distribución) de modo que el código que se consigue una ventaja en rendimiento substancial frente a muchos lenguajes interpretados.

La especificación de Servlets y JSPs define un API de programación y los requisitos para un contenedor (servidor) dentro del cual se puedan desplegar estos componentes para formar aplicaciones web dinámicas completas.

Hoy día existen multitud de contenedores (libres y comerciales) compatibles con estas especificaciones.

A partir de su expansión entre la comunidad de desarrolladores, estas tecnologías han dado paso

a modelos de desarrollo mucho más elaborados con frameworks (pe Struts, Webwork) que se sobreponen sobre los servlets y las JSPs para conseguir un entorno de trabajo mucho más poderoso y segmentado en el que la especialización de roles sea posible (desarrolladores, diseñadores gráficos, ...) y se facilite la reutilización y robustez de código.

A pesar de todo ello, las tecnologías que subyacen (Servlets y JSPs) son substancialmente las mismas.

Este modelo de trabajo se ha convertido en un estándar *de-facto* para el desarrollo de aplicaciones web dinámicas de servidor y otras tecnologías (pe. ASP) se han basado en él.

En aplicaciones de escritorio

Hoy en día existen multitud de aplicaciones gráficas de usuario basadas en Java. El entorno de ejecución Java (JRE) se ha convertido en un componente habitual en los PC de usuario de los sistemas operativos más usados en el mundo. Además, muchas aplicaciones Java lo incluyen dentro del propio paquete de la aplicación de modo que se ejecuten en cualquier PC.

En las primeras versiones de la plataforma Java existían importantes limitaciones en las APIs de desarrollo gráfico (AWT). Desde la aparición de la biblioteca Swing la situación mejoró substancialmente y posteriormente con la aparición de bibliotecas como SWT hacen que el desarrollo de aplicaciones de escritorio complejas y con gran dinamismo, usabilidad, etc. sea relativamente sencillo.

Plataformas soportadas

Una versión del entorno de ejecución Java JRE (Java Runtime Environment) está disponible en la mayoría de equipos de escritorio. Sin embargo, Microsoft no lo ha incluido por defecto en sus sistemas operativos. En el caso de Apple, éste incluye una versión propia del JRE en su sistema operativo, el Mac OS. También es un producto que por defecto aparece en la mayoría de las distribuciones de GNU/Linux. Debido a incompatibilidades entre distintas versiones del JRE, muchas aplicaciones prefieren instalar su propia copia del JRE antes que confiar su suerte a la aplicación instalada por defecto. Los desarrolladores de applets de Java o bien deben insistir a los usuarios en la actualización del JRE, o bien desarrollar bajo una versión antigua de Java y verificar el correcto funcionamiento en las versiones posteriores.

Industria relacionada

Sun Microsystem, como creador del lenguaje de programación Java y de la plataforma JDK, mantiene fuertes políticas para mantener una especificación del lenguaje así como de la máquina virtual a través del JCP. Es debido a este esfuerzo que se mantiene un estándar de facto.

Son innumerables las compañías que desarrollan aplicaciones para Java y/o están volcadas con esta tecnología:

- La industria de la telefonía móvil está fuertemente influenciada por la tecnología Java.
- El entorno de desarrollo Eclipse ha tomado un lugar importante entre la comunidad de desarrolladores Java.
- La fundación Apache tiene también una presencia importante en el desarrollo de bibliotecas y componentes de servidor basados en Java.
- IBM, BEA, IONA, Oracle,... son empresas con grandes intereses y productos creados en y para Java.

Críticas

Harold dijo en 1995 que Java fue creado para abrir una nueva vía en la gestión de software complejo, y es por regla general aceptado que se ha comportado bien en ese aspecto. Sin embargo no puede decirse que Java no tenga grietas, ni que se adapta completamente a todos los estilos de programación, todos los entornos, o todas las necesidades.

General

- Java no ha aportado capacidades estándares para aritmética en punto flotante. El estándar IEEE 754 para "Estándar para Aritmética Binaria en Punto Flotante" apareció en 1985, y desde entonces es el estándar para la industria. Y aunque la aritmética flotante de Java *(cosa que cambió desde el 13 de noviembre de 2006, cuando se abrió el código fuente y se adoptó la licencia GNU, aparte de la ya existente)* se basa en gran medida en la norma del IEEE, no soporta aún algunas características. Más información al respecto puede encontrarse en la sección final de enlaces externos.

El lenguaje

- En un sentido estricto, Java no es un lenguaje absolutamente orientado a objetos, a diferencia de, por ejemplo, Ruby o Smalltalk. Por motivos de eficiencia, Java ha relajado en cierta medida el paradigma de orientación a objetos, y así por ejemplo, no todos los valores son objetos.
- El código Java puede ser a veces redundante en comparación con otros lenguajes. Esto es en parte debido a las frecuentes declaraciones de tipos y conversiones de tipo manual (casting). También se debe a que no se dispone de operadores sobrecargados, y a una sintaxis relativamente simple. Sin embargo, J2SE 5.0 introduce elementos para tratar de reducir la redundancia, como una nueva construcción para los bucles '''foreach'''.
- A diferencia de C++, Java no dispone de operadores de sobrecarga definidos por el usuario.

  Sin embargo esta fue una decisión de diseño que puede verse como una ventaja, ya que esta característica puede hacer los programas difíciles de leer y mantener.

Apariencia

La apariencia externa (el '''look and feel''') de las aplicaciones GUI (Graphical User Interface) escritas en Java usando la plataforma Swing difiere a menudo de la que muestran aplicaciones nativas. Aunque el programador puede usar el juego de herramientas AWT (Abstract Windowing Toolkit) que genera objetos gráficos de la plataforma nativa, el AWT no es capaz de funciones gráficas avanzadas sin sacrificar la portabilidad entre plataformas; ya

que cada una tiene un conjunto de APIs distinto, especialmente para objetos gráficos de alto nivel. Las herramientas de Swing, escritas completamente en Java, evitan este problema construyendo los objetos gráficos a partir de los mecanismos de dibujo básicos que deben estar disponibles en todas las plataformas.

El inconveniente es el trabajo extra requerido para conseguir la misma apariencia de la plataforma destino. Aunque esto es posible (usando GTK+ y el Look-and-Feel de Windows), la mayoría de los usuarios no saben cómo cambiar la apariencia que se proporciona por defecto por aquella que se adapta a la de la plataforma.

Rendimiento

El rendimiento de una aplicación está determinado por multitud de factores, por lo que no es fácil hacer una comparación que resulte totalmente objetiva. En tiempo de ejecución, el rendimiento de una aplicación Java depende más de la eficiencia del compilador, o la JVM, que de las propiedades intrínsecas del lenguaje. El bytecode de Java puede ser interpretado en tiempo de ejecución por la máquina virtual, o bien compilado al cargarse el programa, o durante la propia ejecución, para generar código nativo que se ejecuta directamente sobre el hardware. Si es interpretado, será más lento que usando el código máquina intrínseco de la plataforma destino. Si es compilado, durante la carga inicial o la ejecución, la penalización está en el tiempo necesario para llevar a cabo la compilación.

Algunas características del propio lenguaje conllevan una penalización en tiempo, aunque no son

únicas de Java. Algunas de ellas son el chequeo de los límites de arrays, chequeo en tiempo de ejecución de tipos, y la indirección de funciones virtuales.

El uso de un recolector de basura para eliminar de forma automática aquellos objetos no requeridos, añade una sobrecarga que puede afectar al rendimiento, o ser apenas apreciable, dependiendo de la tecnología del recolector y de la aplicación en concreto. Las JVM modernas usan recolectores de basura que gracias a rápidos algoritmos de manejo de memoria, consiguen que algunas aplicaciones puedan ejecutarse más eficientemente.

El rendimiento entre un compilador JIT y los compiladores nativos puede ser parecido, aunque la distinción no está clara en este punto. La compilación mediante el JIT puede consumir un tiempo apreciable, un inconveniente principalmente para aplicaciones de corta duración o con gran cantidad de código. Sin embargo, una vez compilado, el rendimiento del programa puede ser comparable al que consiguen compiladores nativos de la plataforma destino, inclusive en tareas numéricas. Aunque Java no permite la expansión manual de llamadas a métodos, muchos compiladores JIT realizan esta optimización durante la carga de la aplicación y

pueden aprovechar información del entorno en tiempo de ejecución para llevar a cabo transformaciones eficientes durante la propia ejecución de la aplicación. Esta recompilación dinámica, como la que proporciona la máquina virtual HotSpot de Sun, puede llegar a mejorar el resultado de compiladores estáticos tradicionales, gracias a los datos que sólo están disponibles durante el tiempo de ejecución.

Java fue diseñado para ofrecer seguridad y portabilidad, y no ofrece acceso directo al hardware de la arquitectura ni al espacio de direcciones. Java no soporta expansión de código ensamblador, aunque las aplicaciones pueden acceder a características de bajo nivel usando bibliotecas nativas (JNI, Java Native Interfaces).

JRE

El JRE (Java Runtime Environment, o Entorno en Tiempo de Ejecución de Java) es el software necesario para ejecutar cualquier aplicación desarrollada para la plataforma Java. El usuario final usa el JRE como parte de paquetes software o plugins (o conectores) en un navegador Web. Sun ofrece también el SDK de Java 2, o JDK (Java Development Kit) en cuyo seno reside el JRE, e incluye herramientas como el compilador de Java, Javadoc para generar documentación o el depurador. Puede también obtenerse como un paquete independiente, y puede considerarse como el entorno necesario para ejecutar una aplicación Java, mientras que un desarrollador debe además contar con otras facilidades que ofrece el JDK.

Componentes

- Bibliotecas de Java, que son el resultado de compilar el código fuente desarrollado por quien implementa la JRE, y que ofrecen apoyo para el desarrollo en Java. Algunos ejemplos de estas bibliotecas son:
  - Las bibliotecas centrales, que incluyen:
    - Una colección de bibliotecas para implementar estructuras de datos como listas, arrays,
      árboles y conjuntos.
    - Bibliotecas para análisis de XML.
    - Seguridad.
    - Bibliotecas de internacionalización y localización.
  - Bibliotecas de integración, que permiten la comunicación con sistemas externos. Estas bibliotecas incluyen:
    - La API para acceso a bases de datos JDBC (Java DataBase Conectivity).
    - La interfaz JNDI (Java Naming and Directory Interface) para servicios de directorio.
    - RMI (Remote Method Invocation) y CORBA para el desarrollo de aplicaciones distribuidas.
  - Bibliotecas para la interfaz de usuario, que incluyen:
  - El conjunto de herramientas nativas AWT (Abstract Windowing Toolkit), que ofrece

componentes GUI (Graphical User Interface), mecanismos para usarlos y manejar sus eventos asociados.

- Las Bibliotecas de Swing, construidas sobre AWT pero ofrecen implementaciones no nativas de los componentes de AWT.
- APIs para la captura, procesamiento y reproducción de audio.

•Una implementación dependiente de la plataforma en que se ejecuta de la máquina virtual deJava (JVM), que es la encargada de la ejecución del código de las bibliotecas y las aplicaciones externas.

•Plugins o conectores que permiten ejecutar applets en los navegadores Web.

- Java Web Start, para la distribución de aplicaciones Java a través de Internet.
- Documentación y licencia.

APIs

Sun define tres plataformas en un intento por cubrir distintos entornos de aplicación. Así, ha distribuido muchas de sus APIs (Application Program Interface) de forma que pertenezcan a cada una de las plataformas:

- Java ME (Java Platform, Micro Edition) o J2ME — orientada a entornos de limitados recursos, como teléfonos móviles, PDAs (Personal Digital Assistant), etc.
- Java SE (Java Platform, Standard Edition) o J2SE — para entornos de gama media y estaciones de trabajo. Aquí se sitúa al usuario medio en un PC de escritorio.
- Java EE (Java Platform, Enterprise Edition) o J2EE — orientada a entornos distribuidos empresariales o de Internet.

Las clases en las APIs de Java se organizan en grupos disjuntos llamados paquetes. Cada paquete contiene un conjunto de interfaces, clases y excepciones relacionadas. La información sobre los paquetes que ofrece cada plataforma puede encontrarse en la documentación de ésta.

El conjunto de las APIs es controlado por Sun Microsystems junto con otras entidades o personas a través del programa JCP (Java Community Process). Las compañías o individuos participantes del JCP pueden influir de forma activa en el diseño y desarrollo de las APIs, algo que ha sido motivo de controversia.

En 2004, IBM y BEA apoyaron públicamente la idea de crear una implementación de código abierto (open source) de Java, algo a lo que Sun, a fecha de 2006, se ha negado.

Extensiones y arquitecturas relacionadas

Las extensiones de Java están en paquetes que cuelgan de la raíz javax: `javax.*`. No se incluyen en la JDK o el JRE. Algunas de las extensiones y arquitecturas ligadas estrechamente al lenguaje Java son:

- Java EE (Java Platform, Enterprise Edition; antes J2EE) —para aplicaciones distribuidas orientadas al entorno empresarial

Java en código abierto

Java se ha convertido en un lenguaje con una implantación masiva en todos los entornos (personales y empresariales). El control que mantiene Sun sobre éste genera reticencias en la comunidad de empresas con fuertes intereses en Java (IBM, Oracle) y obviamente en la comunidad de desarrolladores de software libre. La evolución basada en un comité en el que participen todos los implicados no es suficiente y la comunidad demandaba desde hace tiempo la liberación de las APIs y bibliotecas básicas de la JDK.

Características de Java

Las características principales que nos ofrece Java respecto a cualquier otro lenguaje de programación, serían:

Simple

Java ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos. C++ no es un lenguaje conveniente por razones de seguridad, pero C y C++ son los lenguajes más difundidos, por ello Java se diseñó para ser parecido a C++ y así facilitar un rápido y fácil aprendizaje.

Java elimina muchas de las características de otros lenguajes como C++, para mantener reducidas las especificaciones del lenguaje y añadir características muy útiles como el *garbage collector* (reciclador de memoria dinámica). No es necesario preocuparse de liberar memoria, el reciclador se encarga de ello y como es de baja prioridad, cuando entra en acción, permite liberar bloques de memoria muy grandes, lo que limita en mucho la fragmentación de la memoria.

Java reduce en un 50% los errores más comunes de programación con lenguajes como C y C++ al eliminar muchas de las características de éstos, entre las que destacan:

- aritmética de punteros
- no existen referencias
- registros (struct)
- definición de tipos (typedef)
- macros (#define)
- necesidad de liberar memoria (free)

Aunque, en realidad, lo que hace es eliminar las palabras reservadas (struct, typedef), ya que las clases son algo parecido.

Además, el intérprete completo de Java que hay en este momento es muy pequeño, solamente ocupa 215
Kb de RAM.

Orientado a Objetos

Java implementa la tecnología básica de C++ con algunas mejoras y elimina algunas cosas para mantener el objetivo de la simplicidad del lenguaje. Java trabaja con sus datos como objetos y con interfaces a esos objetos. Soporta las tres características propias del paradigma de la orientación a objetos: encapsulación, herencia y polimorfismo. Las plantillas de objetos son llamadas, como en C++, *clases* y sus copias, *instancias*. Estas instancias, como en C++, necesitan ser construidas y destruidas en espacios de memoria.

Java incorpora funcionalidades inexistentes en C++ como por ejemplo, la resolución dinámica de métodos. Esta característica deriva del lenguaje Objective C, propietario del sistema operativo Next. En C++ se suele trabajar con librerías dinámicas (DLLs) que obligan a recompilar la aplicación cuando se retocan las funciones de su interior.

Este inconveniente es resuelto por Java mediante una interfaz específica llamada RTTI (*RunTime Type Identification*) que define la interacción entre objetos excluyendo variables de instancias o implementación de métodos. Las clases en Java tienen una representación en el *runtime* que permite a los programadores interrogar por el tipo de clase y enlazar dinámicamente la clase con el resultado de la búsqueda.

Distribuido

Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de rutinas 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 ficheros locales.

Java en sí no es distribuido, sino que proporciona las librerías y herramientas para que los programas puedan ser distribuidos, es decir, que se corran en varias máquinas, interactuando.

Robusto

Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador de la liberación o corrupción de memoria.

También implementa los *arrays* auténticos, en vez de listas enlazadas de punteros, con comprobación de límites, para evitar la posibilidad de sobreescribir o corromper memoria resultado de punteros que señalan a zonas equivocadas. Estas características reducen drásticamente el tiempo empleado en el desarrollo de aplicaciones Java.

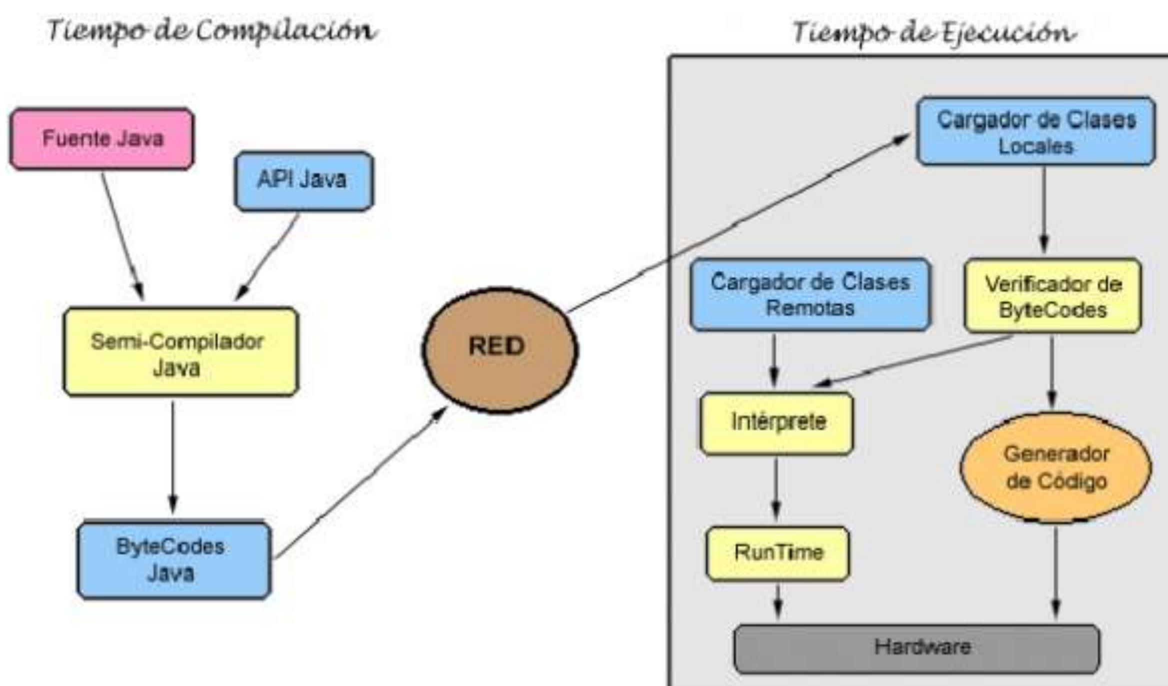
Además, para asegurar el funcionamiento de la aplicación, realiza una verificación de los *ByteCodes*, que son el resultado de la compilación de un programa Java. Es un código de máquina virtual que es interpretado por el intérprete Java. No es el código máquina directamente entendible por el hardware, pero ya ha pasado todas las fases del compilador: análisis de instrucciones, orden de operadores, etc., y ya tiene generada la pila de ejecución de órdenes.

Java proporciona, pues:

- Comprobación de punteros
- Comprobación de límites de arrays
- Excepciones
- Verificación de ByteCodes

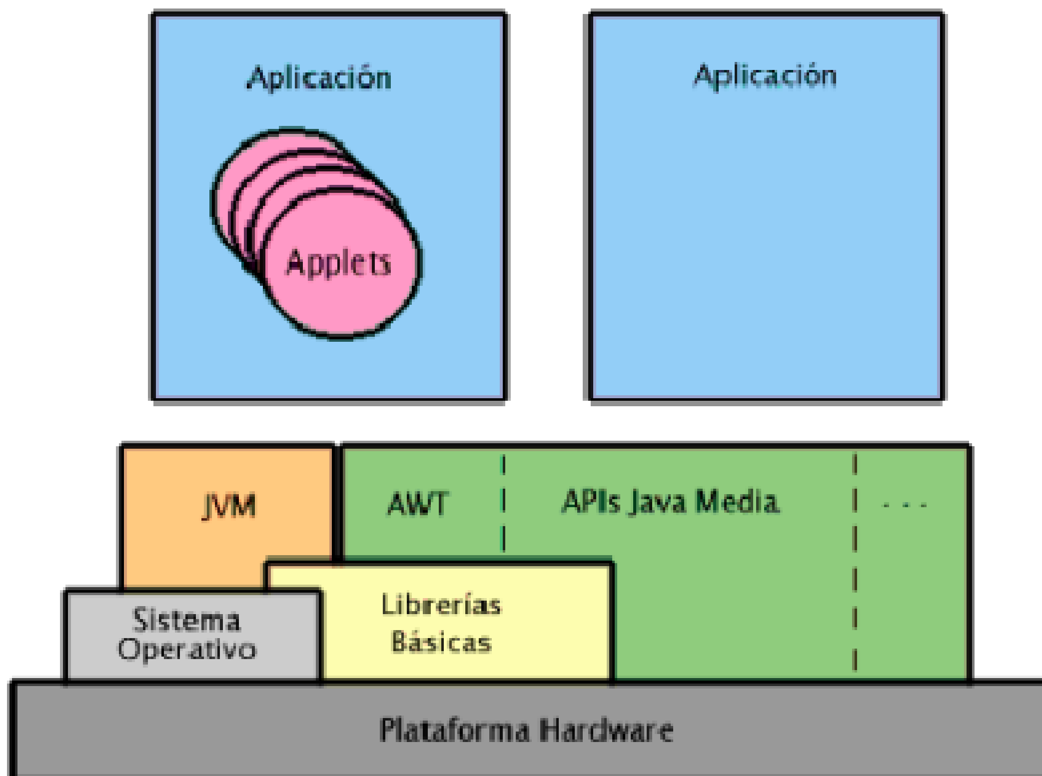
Arquitectura Neutral

Para establecer Java como parte integral de la red, el compilador Java compila su código a un fichero objeto de formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (*run-time*) 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:



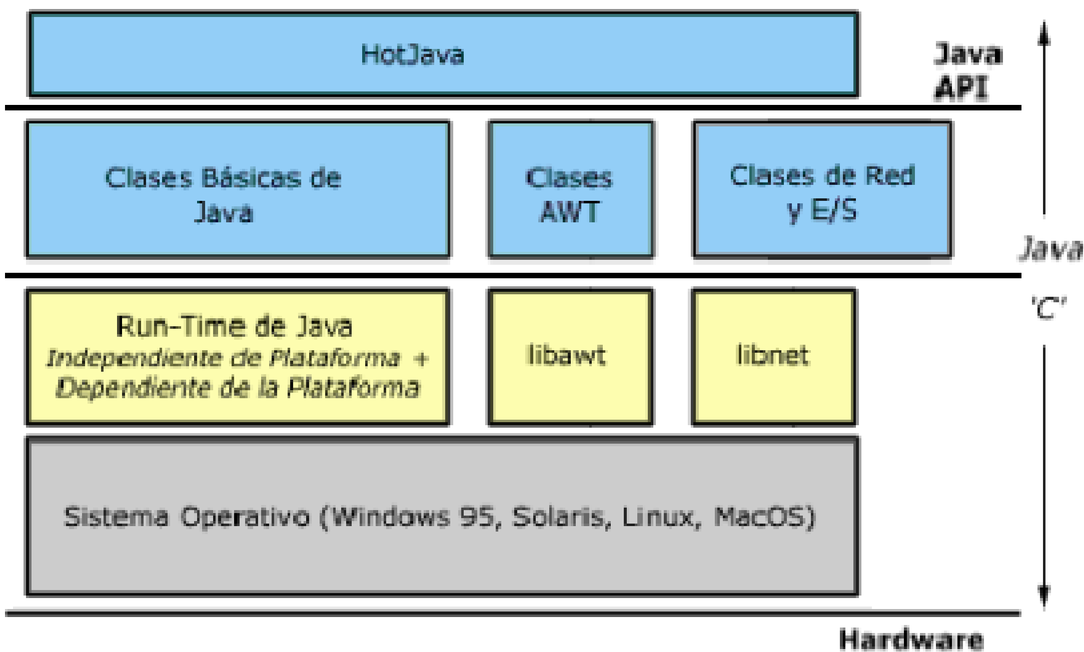
En ella podemos ver que lo verdaderamente dependiente del sistema es la *Máquina Virtual Java* (JVM) y las librerías fundamentales, que también permitirían acceder directamente al hardware de la máquina. Además, siempre habrá APIs de Java que también entren en contacto directo con el hardware y serán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

- Swing : Mejora de las herramientas para la implementación de interfaces de usuario

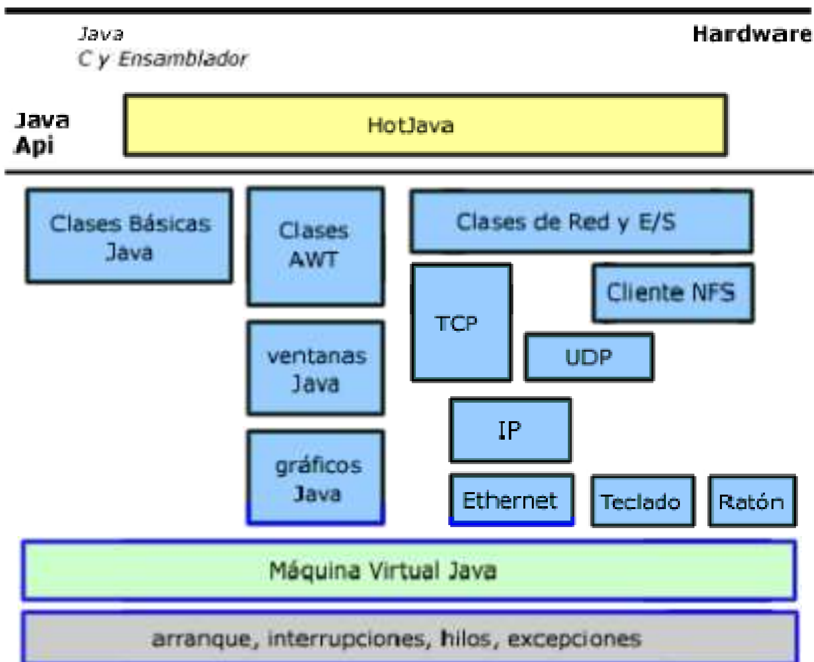
- Java 2D : gráficos 2D y manipulación de imágenes

- 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 implementación de aplicaciones o applets o cualquier otra librería.



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. Cuando se usa Java para crear un navegador, se combinan las características del lenguaje con protecciones de sentido común aplicadas al propio navegador.

El lenguaje C, por ejemplo, tiene lagunas de seguridad importantes, como son los *errores de alineación*. Los programadores de C utilizan punteros en conjunción con operaciones aritméticas. Esto le permite al programador que un puntero referencie a un lugar conocido de la memoria y pueda sumar (o restar) algún valor, para referirse a otro lugar de la memoria. Si otros programadores conocen nuestras estructuras de datos pueden extraer información confidencial de nuestro sistema. Con un lenguaje como C, se pueden tomar números enteros aleatorios y convertirlos en punteros para luego acceder a la memoria:

```
printf( "Escribe un valor entero: " );
scanf( "%u",&puntero );
printf( "Cadena de memoria: %s\n",puntero );
```

Otra laguna de seguridad u otro tipo de ataque, es el *Caballo de Troya*. Se presenta un programa como una utilidad, resultando tener una funcionalidad destructiva. Por ejemplo, en UNIX se visualiza el contenido de un directorio con el comando *ls*. Si un programador deja un comando destructivo bajo esta referencia, se puede correr el riesgo de ejecutar código malicioso, aunque el comando siga haciendo la funcionalidad que se le supone, después de lanzar su carga destructiva. Por ejemplo, después de que el caballo de Troya haya enviado por correo el /etc/shadow a su creador, ejecuta la funcionalidad de *ls* presentando el contenido del directorio. Se notará un retardo, pero nada inusual.

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

El Cargador de Clases también ayuda a Java a mantener su seguridad, separando el espacio de nombres del sistema de ficheros local del de los recursos procedentes de la red. Esto limita cualquier aplicación del tipo Caballo de Troya, ya que las clases se buscan primero entre las locales y luego entre las procedentes del exterior.

Las clases importadas de la red se almacenan en un espacio de nombres privado, asociado con el origen. Cuando una clase del espacio de nombres privado accede a otra clase, primero se busca en las clases predefinidas (del sistema local) y luego en el espacio de nombres de la clase que hace la referencia. Esto imposibilita que una clase suplante a una predefinida.

En resumen, las aplicaciones de Java resultan extremadamente seguras, ya que no acceden a zonas delicadas de memoria o de sistema, con lo cual evitan la interacción de ciertos virus. Java no posee una semántica específica para modificar la pila de programa o la memoria libre o utilizar objetos y métodos de un programa sin los privilegios del kernel del sistema operativo. Además, para evitar modificaciones por parte de los crackers de la red, implementa un método ultraseguro de autentificación por clave pública. El Cargador de Clases puede verificar una firma digital antes de realizar una instancia de un objeto. Por tanto ningún objeto se crea y almacena en memoria, sin que se validen los privilegios de acceso. Es decir, la seguridad se integra en el momento en que se interpreta, con el nivel de detalle y de privilegio que sea necesario. Con código compilado no sería posible establecer estos niveles de seguridad entre dos objetos pertenecientes al mismo proceso, porque al compartr un único espacio de direcciones, el código de uno podría acceder tranquilamente a cualquier dirección del otro.

Dada, pues la concepción del lenguaje y si todos los elementos se mantienen dentro del estándar marcado por Sun, no hay peligro. Java imposibilita, también, abrir ningún fichero de la máquina local (siempre que se realizan operaciones con archivos, éstas trabajan sobre el disco duro de la máquina de donde partió el applet), no permite ejecutar ninguna aplicación nativa de una plataforma e impide que se utilicen otros ordenadores como puente, es decir, nadie puede utilizar nuestra máquina para hacer peticiones o realizar operaciones con otra. Además, los intérpretes que incorporan los navegadores de la Web son aún más restrictivos. Bajo estas condiciones (y dentro de la filosofía de que el único ordenador seguro es el que está apagado, desenchufado, dentro de una cámara acorazada en un bunker y rodeado por mil soldados de los cuerpos especiales del ejército), se puede considerar que Java es un lenguaje

seguro y que los applets están libres de virus.

Respecto a la seguridad del código fuente, no ya del lenguaje, el propio JDK proporciona un desemsamblador de ByteCode, lo cual hace que cualquier programa pueda ser convertido a código fuente, lo que para el programador significa una vulnerabilidad total a su código. Utilizando *javap* no se obtiene el código fuente original, pero sí desmonta el programa mostrando el algoritmo que se utiliza, que es lo realmente interesante. La protección de los programadores ante esto es utilizar llamadas a programas nativos, externos (incluso en C o C++) de forma que no sea descompilable todo el código; aunque así se pierda portabilidad. Es otra de las cuestiones que Java tiene pendientes.

Portable

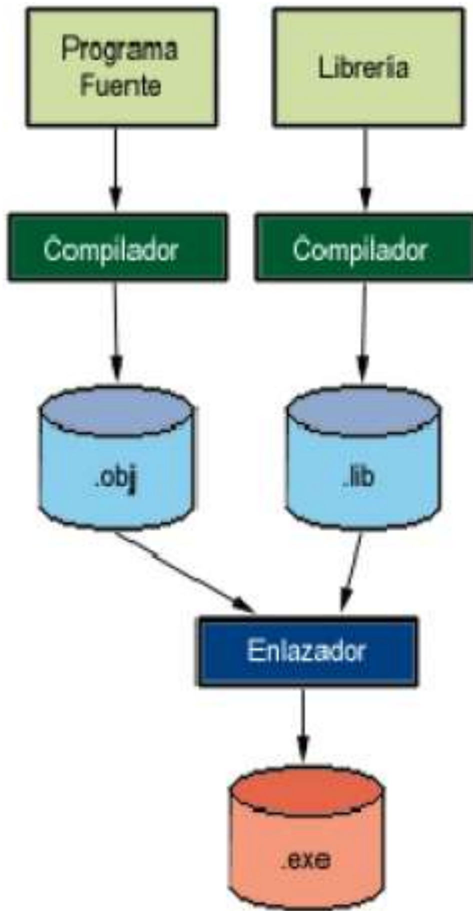
Más allá de la portabilidad básica por ser de arquitectura independiente, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Los enteros son siempre enteros y además, enteros de 32 bits en complemento a 2. Además, Java construye sus interfaces de usuario a través de un sistema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

Interpretado

El intérprete Java (sistema *run-time*) puede ejecutar directamente el código objeto. Enlazar (*linkar*) un programa normalmente consume menos recursos que compilarlo, por lo que los desarrolladores con Java pasarán más tiempo desarrollando y menos esperando por el ordenador. No obstante, el compilador actual del JDK es bastante lento. Por ahora, que todavía no hay compiladores específicos de Java para las diversas plataformas, 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.

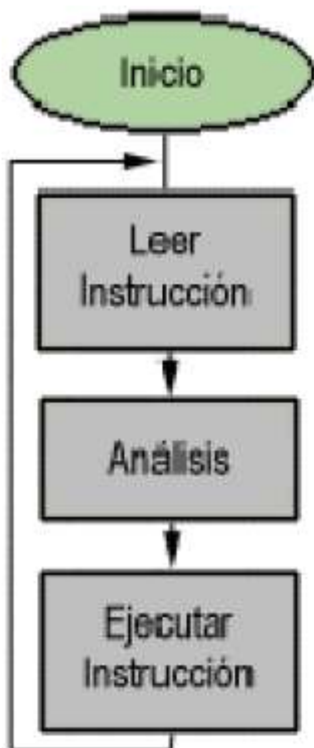
Se dice que Java es de 10 a 30 veces más lento que C, y que tampoco existen en Java proyectos de gran envergadura como en otros lenguajes. La verdad es que ya hay comparaciones ventajosas entre Java y el resto de los lenguajes de programación, y una ingente cantidad de folletos electrónicos que supuran fanatismo en favor y en contra de los distintos lenguajes contendientes con Java. Lo que se suele dejar de lado en todo esto, es que primero habría que decidir hasta que punto Java, un lenguaje en pleno desarrollo y todavía sin definición definitiva, está maduro como lenguaje de programación para ser comparado con otros; como por ejemplo con Smalltalk, que lleva más de 20 años en cancha.

La verdad es que 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 ningún contrasentido, me explico, 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 y no tiene nada que ver con el *p-code* de Visual Basic. 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.

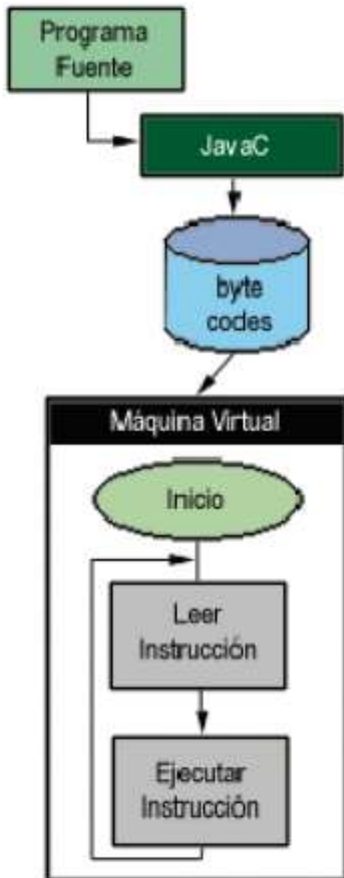


No obstante, este panorama está cambiando a pasos agigantados, y aunque Java sigue siendo básicamente un lenguaje interpretado, la situación se acerca mucho a la de los programas compilados, sobre todo en lo que a la rapidez en la ejecución del código se refiere. Para comprender el funcionamiento de ***HotSpot***, que es el
último lanzamiento que hace Sun y que promete interpretar los ByteCodes más rápido que un programa compilado, las siguientes imágenes muestran cómo actúa el sistema *runtime* en los diversos casos que hasta ahora pueden darse.

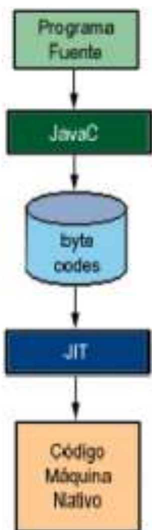
La imagen de arriba muestra las acciones correspondientes a un **compilador tradicional**. El compilador traslada las sentencias escritas en lenguaje de alto-nivel a múltiples instrucciones, que luego son enlazadas junto con el resultado de múltiples compilaciones previas que han dado origen a librerías,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



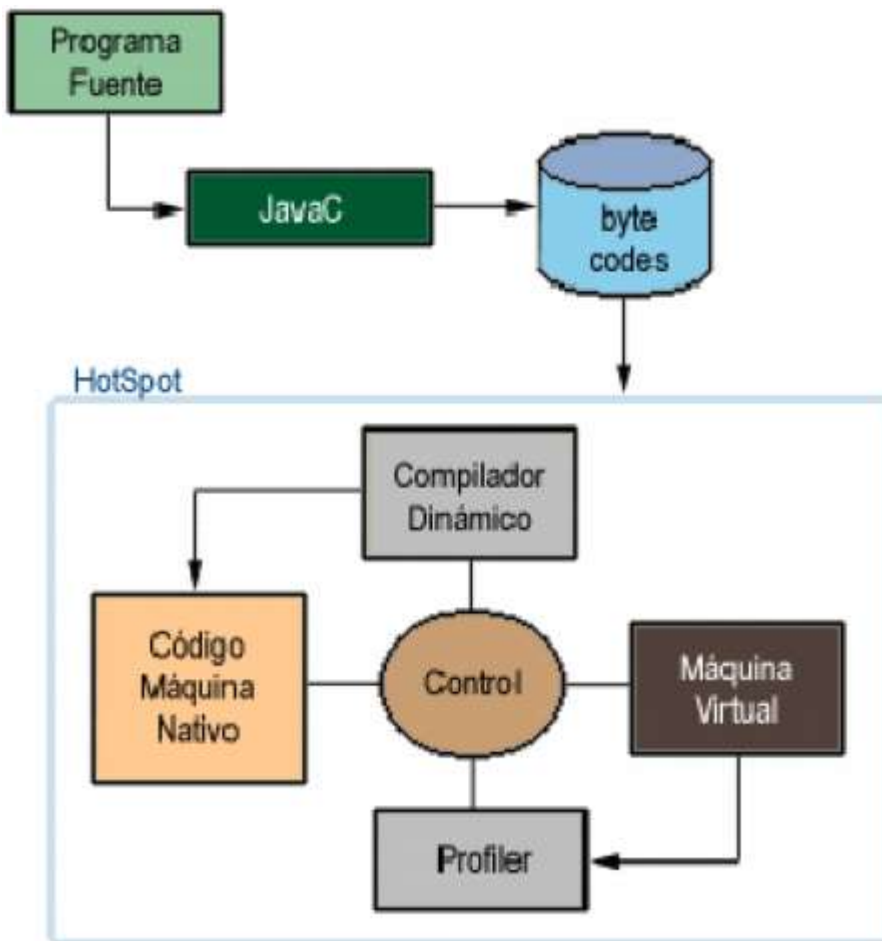
La imagen de arriba muestra la forma de actuación de un intérprete. Básicamente es un enorme bucle, en el cual se va leyendo o recogiendo cada una de las instrucciones del programa fuente que se desea ejecutar, se analiza, se parte en trozos y se ejecuta. Luego se va a recoger la siguiente instrucción 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 anterior, 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 siguiente, situada a la izquierda de la página. 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 la 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 ha proporcionado muchos detalles sobre este extremo.

Multihilo

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++.

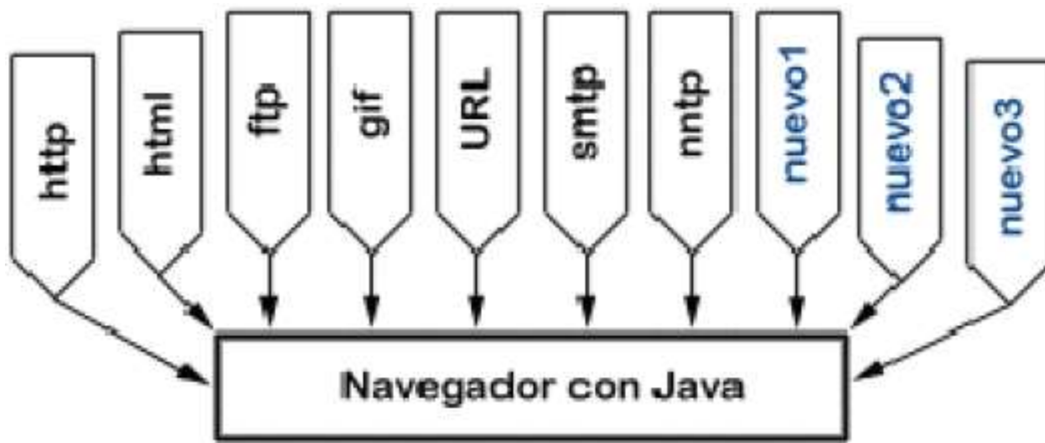
El beneficio de ser multihilo consiste en un mejor rendimiento interactivo y mejor comportamiento en tiempo real. Aunque el comportamiento en tiempo real está limitado a las capacidades del sistema operativo subyacente (Unix, Windows, etc.) de la plataforma, aún supera a los entornos de flujo único de 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.

Dinámico

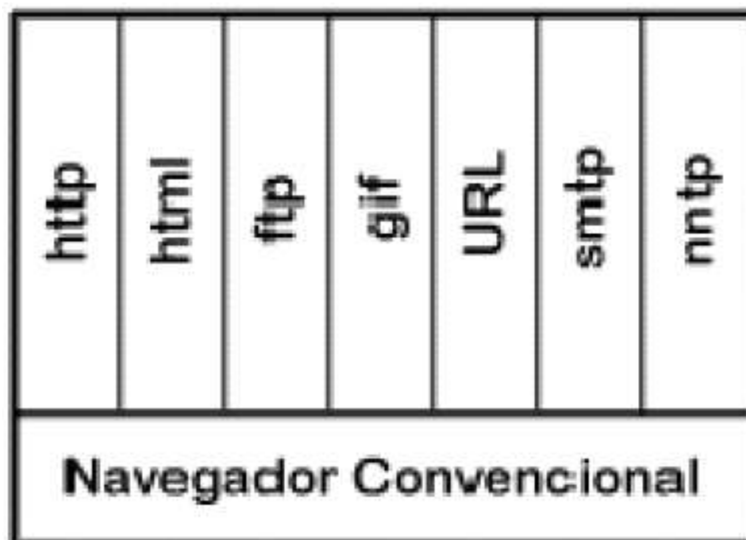
Java se beneficia todo lo posible de la tecnología orientada a objetos. Java no intenta conectar todos los módulos que comprenden una aplicación hasta el mismo tiempo de ejecución. Las librería nuevas o actualizadas no paralizarán la ejecución de las aplicaciones actuales - siempre que mantengan el API anterior.

Java también simplifica el uso de protocolos nuevos o actualizados. Si su sistema ejecuta una aplicación Java sobre la red y encuentra una pieza de la aplicación que no sabe manejar, tal como se ha explicado en párrafos anteriores, Java es capaz de traer automáticamente cualquier pieza que el sistema necesite para funcionar.

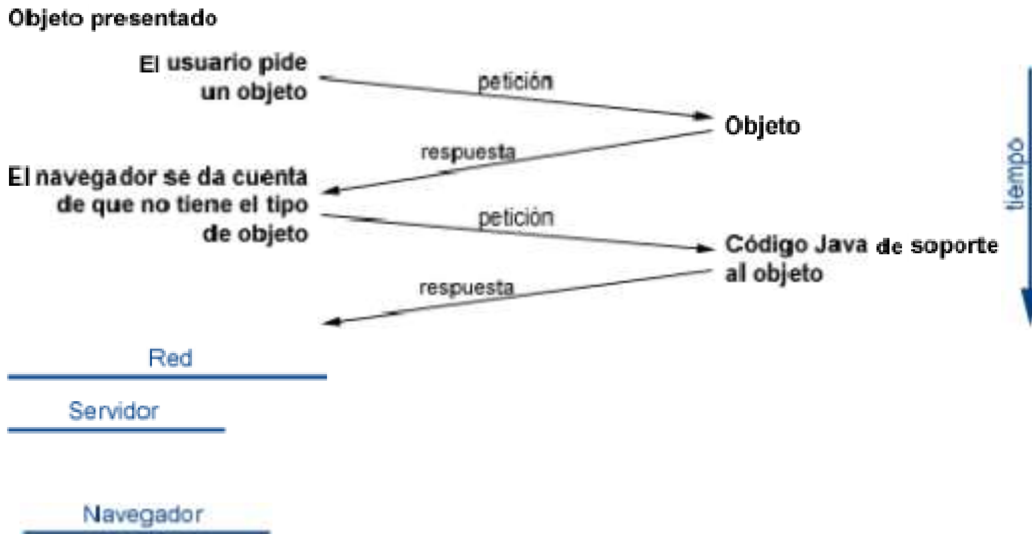


Sistema Federado: el navegador es un coordinador de piezas, y cada pieza es responsable de una función. Las piezas se pueden añadir dinámicamente a través de la red

Monolito: cada pieza de código se compacta dentro del código del navegador



Java, para evitar que los módulos de ByteCode o los objetos o nuevas clases, no haya que estar trayéndolas de la red cada vez que se necesiten, implementa las opciones de persistencia, para que no se eliminen cuando de limpie la caché de la máquina.



2.4 Otros Lenguajes orientados a objetos

Python	
Paradigma:	multiparadigma: orientado a objetos, imperativo, funcional
Apareció en:	1991
Diseñado por:	Guido van Rossum
Última	3.1.1 (17 de agosto de 2009)
Tipo de dato:	fuertemente tipado, dinámico
Implementacion	CPython, Jython, IronPython, PyPy
Dialectos:	Stackless Python, RPython
Influido por:	ABC, Tcl, Perl, Modula-3, Smalltalk, ALGOL 68, C,
Ha influido a:	Ruby, Boo, Groovy, Cobra, D
Sistema	Multiplataforma
Licencia de	Python Software Foundation License

Web	http://www.python.org/

Python es un lenguaje de programación interpretado creado por Guido van Rossum en el año

1991. Se compara habitualmente con Tcl, Perl, Scheme, Java y Ruby. En la actualidad Python se desarrolla como un proyecto de código abierto, administrado por la Python Software Foundation. La última versión estable del lenguaje es la 3.1.1. Python es considerado como la "oposición leal" a Perl, lenguaje con el cual mantiene una rivalidad amistosa.

Los usuarios de Python consideran a éste mucho más limpio y elegante para programar. Python permite dividir el programa en módulos reutilizables desde otros programas Python. Viene con una gran colección de módulos estándar que se pueden utilizar como base de los programas (o como ejemplos para empezar a aprender Python). También hay módulos incluidos que proporcionan E/S de ficheros, llamadas al sistema, sockets y hasta interfaces a GUI (interfaz gráfica con el usuario) como Tk, GTK, Qt entre otros.

Python se utiliza como lenguaje de programación interpretado, lo que ahorra un tiempo considerable en el desarrollo del programa, pues no es necesario compilar ni enlazar. El intérprete se puede utilizar de modo interactivo, lo que facilita experimentar con características del lenguaje, escribir programas desechables o probar funciones durante el desarrollo del programa.

El nombre del lenguaje proviene de la afición de su creador original, Guido van Rossum, por los humoristas británicos Monty Python. El principal objetivo que persigue este lenguaje es la facilidad, tanto de lectura, como de diseño.

Historia

Python fue creado a finales de los ochenta por Guido van Rossum en CWI en los Países Bajos como un sucesor del lenguaje de programación ABC, capaz de manejar excepciones e interactuar con el sistema operativo Amoeba.

Van Rossum es el principal autor de Python, y su continuo rol central en decidir la dirección de Python es reconocido, refiriéndose a él como *Benevolente dictador vitalicio* o *Benevolent Dictator for Life* (BDFL).

En 1991, van Rossum publicó el código (versión 0.9.0) en lt.sources alt.sources [6]. En esta etapa del desarrollo ya estaban presentes clases con herencia, manejo de excepciones, funciones, y los tipos modulares: `list`, `dict`, `str` y así sucesivamente. Además en este lanzamiento inicial aparecía un sistema de módulos adoptado de Modula-3; van Rossum
describe el módulo como "uno de las mayores unidades de programación de Python".

El modelo de excepciones en Python es parecido al de Modula-3, con la adición de una cláusula `else` . En el año 1994 se formó [news:comp.lang.python comp.lang.python], el foro de discusión principal de Python, marcando un hito en el crecimiento del grupo de usuarios de este lenguaje.

Python alcanzó la versión 1.0 en enero de 1994. Una característica de este lanzamiento fueron las herramientas de la programación funcional: `lambda`, map, `filter` y `reduce`. Van Rossum explicó que "Hace 12 años, Python adquirió lambda, reduce(), filter() and map(), cortesía de un hacker de Lisp que las extrañaba y que envió parches." El donante fue Amrit Prem; no se hace ninguna mención específica de cualquier herencia de Lisp en las notas de lanzamiento.

La última versión liberada proveniente de CWI fue Python 1.2. En 1995, van Rossum continuó su trabajo en Python en la Corporation for National Research Initiatives (CNRI) en Reston, Virginia, donde lanzó varias versiones del software.

Durante su estancia en CNRI, van Rossum lanzó la iniciativa *Computer Programming for Everybody* (CP4E), con el fin de hacer la programación más accesible a más gente, con un nivel de 'alfabetización' básico en lenguajes de programación, similar a la alfabetización básica en inglés y habilidades matemáticas necesarias por muchos trabajadores. Python tuvo un papel crucial en este proceso: debido a su orientación hacia una sintaxis limpia, ya era idóneo, y las metas de CP4E presentaban similitudes con su predecesor, ABC.

El proyecto fue patrocinado por DARPA. En el año 2007, el proyecto CP4E está inactivo, y mientras Python intenta ser fácil de aprender y no muy arcano en su sintaxis y semántica, alcanzando a los no-programadores, no es una preocupación activa.

En el año 2000, el principal equipo de desarrolladores de Python se cambió a BeOpen.com para formar el equipo BeOpen PythonLabs. CNRI pidió que la versión 1.6 fuera pública, continuando su desarrollo hasta que el equipo de desarrollo abandonó CNRI; su programa de lanzamiento y el de la versión 2.0 tenían una significativa cantidad de traslapo. Python 2.0 fue el primer y único lanzamiento de BeOpen.com. Después que Python 2.0 fuera publicado por BeOpen.com, Guido van Rossum y los otros desarrolladores PythonLabs se unieron en Digital Creations.

Python 2.0 tomó una característica mayor del lenguaje de programación funcional Haskell: list comprehensions. La sintaxis de Python para esta construcción es muy similar a la de Haskell, salvo por la preferencia de los caracteres de puntuación en Haskell, y la preferencia de Python por palabras claves alfabéticas. Python 2.0 introdujo además un sistema de recolección de basura capaz de recolectar referencias cíclicas.

Posterior a este doble lanzamiento, y después que van Rossum dejó CNRI para trabajar con desarrolladores de software comercial, quedó claro que la opción de usar Python con software disponible bajo GPL era muy deseable. La licencia usada entonces, la Python License, incluía una cláusula estipulando que la licencia estaba gobernada por el estado de Virginia, por lo que, bajo la óptica de los abogados de Free Software Foundation (FSF), se hacía incompatible con GNU GPL. CNRI y FSF se relacionaron para cambiar la licencia de software libre de Python para hacerla compatible con GPL. En el año

2001, van Rossum fue premiado con FSF Award for the Advancement of Free Software.

Python 1.6.1 es esencialmente el mismo que Python 1.6, con unos pocos arreglos de bugs, y con una nueva licencia compatible con GPL.

Código Python

Una innovación mayor en Python 2.2 fue la unificación de los tipos en Python (tipos escritos en C), y clases (tipos escritos en Python) dentro de una jerarquía. Esa unificación logró un modelo de objetos de Python puro y consistente. También fueron agregados los generadores que fueron inspirados por el lenguaje Icon.

Características y paradigmas

Python es un lenguaje de programación multiparadigma. Esto significa que más que forzar a los programadores a adoptar un estilo particular de programación, permite varios estilos: programación orientada a objetos, programación estructurada y programación funcional. Otros muchos paradigmas más están soportados mediante el uso de extensiones. Python usa tipo de dato dinámico y reference counting para el manejo de memoria. Una característica importante de Python es la resolución dinámica de nombres, lo que enlaza un método y un nombre de variable durante la ejecución del programa (también llamado ligadura dinámica de métodos).

Otro objetivo del diseño del lenguaje era la facilidad de extensión. Nuevos módulos se pueden escribir fácilmente en C o C++. Python puede utilizarse como un lenguaje de extensión para módulos y aplicaciones que necesitan de una interfaz programable.

Aunque el diseño de Python es de alguna manera hostil a la programación funcional tradicional del Lisp, existen bastantes analogías entre Python y los lenguajes minimalistas de la familia Lisp como puede ser Scheme.

Filosofía

Los usuarios de Python se refieren a menudo a la Filosofía Python que es bastante

análoga a la filosofía de Unix. El código que sigue los principios de Python de legibilidad y transparencia se dice que es "pythonico". Contrariamente, el código opaco u ofuscado es bautizado como "no pythonico" ("unpythonic" en inglés).

Desde la versión 2.1.2, Python incluye estos puntos (en su versión original en inglés) como un huevo de pascua que se muestra al ejecutar `import this`.

Modo interactivo

El intérprete de Python estándar incluye un *modo interactivo*, en el cual se escriben las instrucciones en una especie de shell: las expresiones pueden ser introducidas una a una, pudiendo verse el resultado de su evaluación inmediatamente. Esto resulta útil tanto para las personas que se están familiarizando con el lenguaje como también para los programadores más avanzados: se pueden probar porciones de código en el modo interactivo antes de integrarlo como parte de un programa.

Módulos

Existen muchas propiedades que se pueden agregar al lenguaje importando módulos, que son minicódigos (la mayoría escritos también en Python) que llaman a los recursos del sistema. Un ejemplo es el módulo Tkinter, que permite crear interfaces gráficas que incluyan botones, cajas de texto, y muchas cosas que vemos habitualmente en el sistema operativo. Otro ejemplo es el módulo que permite manejar el sistema operativo desde el lenguaje. Los módulos se agregan a los códigos escribiendo `import` seguida del nombre del módulo que queramos usar. En este código se muestra como apagar el ordenador desde Windows.

```
apagar =
"shutdown /s"
import os
os.system(apaga
r
)
```

Sistema de objetos

En Python, todo es un objeto (incluso las clases). Las clases, al ser objetos, son instancias de una metaclase. Python además soporta herencia múltiple y polimorfismo.

Licencias

Python posee una licencia de código abierto, denominada *Python Software Foundation License*, que es compatible con la licencia GPL a partir de la versión 2.1.1, e incompatible en ciertas versiones anteriores. Esta licencia no obliga a liberar el código fuente al distribuir los archivos binarios.

Ruby

Paradigma:	multiparadigma, orientado
Apareció	1995
Diseñado	Yukihiro Matsumoto
Última	1.9.1 (Febrero de 2009)
Tipo de	fuerte, dinámico
Implementaci	Ruby, JRuby, Rubinius,
Influido	Perl, Smalltalk, Python,
Ha influido	Groovy

Ruby es un lenguaje de programación interpretado, reflexivo y orientado a objetos, creado por el programador japonés Yukihiro "Matz" Matsumoto, quien comenzó a trabajar en Ruby en 1993, y lo presentó públicamente en

1995. Combina una sintaxis inspirada en Python, Perl con características de programación orientada a objetos similares a Smalltalk.

Comparte también funcionalidad con otros lenguajes de programación como Lisp, Lua, Dylan y CLU. Ruby es un lenguaje de programación interpretado en una sola pasada y su implementación oficial es distribuida bajo una licencia de software libre.

Historia

El lenguaje fue creado por Yukihiro "Matz" Matsumoto, quien empezó a trabajar en Ruby el

24 de febrero de 1993, y lo presentó al público en el año 1995. En el círculo de amigos de Matsumoto se le puso el nombre de "Ruby" (en español *rubí*) como broma aludiendo al lenguaje de programación "Perl" (*perla*). La última versión estable es la 1.8.6, publicada en diciembre de 2007. El 26 de ese mismo mes salió Ruby 1.9.0, una versión en desarrollo que incorpora mejoras sustanciales en el rendimiento del lenguaje, que se espera queden reflejadas en la próxima versión estable de producción del lenguaje, Ruby 1.9.0.1 Diferencias en rendimiento entre la actual implementación de Ruby (1.8.6) y otros lenguajes de programación más arraigados han llevado al desarrollo de varias máquinas virtuales para Ruby. Entre ésas se encuentra JRuby, un intento de llevar Ruby a la plataforma Java, y Rubinius, un intérprete modelado basado en las máquinas virtuales de Smalltalk. Los principales desarrolladores han apoyado la máquina virtual proporcionada por el proyecto YARV, que se fusionó en el árbol de código fuente de Ruby el 31 de diciembre de 2006, y se dará a conocer como Ruby 1.9.0.1.

Objetivo

El creador del lenguaje, Yukihiro "Matz" Matsumoto, ha dicho que Ruby está diseñado para la productividad y la diversión del desarrollador, siguiendo los principios de una buena interfaz de usuario. Sostiene que el diseño de sistemas necesita enfatizar las necesidades humanas más que las de la máquina:

A menudo la gente, especialmente los ingenieros en informática, se centran en las máquinas. Ellos piensan, "Haciendo esto, la máquina funcionará más rápido. Haciendo esto, la máquina funcionará de manera más eficiente. Haciendo esto..." Están centrados en las máquinas, pero en realidad necesitamos centrarnos en las personas, en cómo hacen

programas o cómo manejan las aplicaciones en los ordenadores. Nosotros somos los jefes. Ellos son los esclavos.

Ruby sigue el "principio de la menor sorpresa", lo que significa que el lenguaje debe comportarse de tal manera que minimice la confusión de los usuarios experimentados. Matz ha dicho que su principal objetivo era hacer un lenguaje que le divirtiera él mismo, minimizando el trabajo de programación y la posible confusión.

Él ha dicho que no ha aplicado el principio de menor sorpresa al diseño de Ruby, pero sin embargo la frase se ha asociado al lenguaje de programación Ruby. La frase en sí misma ha sido fuente de controversia, ya que los no iniciados pueden tomarla como que la características de Ruby intentar ser similares a las características de otros lenguajes conocidos. En mayo de
2005 en una discusión en el grupo de noticias comp.lang.ruby, Matz trató de distanciar Ruby de la mencionada filosofía, explicando que cualquier elección de diseño será sorprendente para alguien, y que él usa un estándar personal de evaluación de la sorpresa. Si ese estándar personal se mantiene consistente habrá pocas sorpresas para aquellos familiarizados con el estándar.

Matz lo definió de esta manera en una
entrevista:

Todo el mundo tiene un pasado personal. Alguien puede venir de Python, otro de Perl, y ellos pueden verse sorprendidos por distintos aspectos del lenguaje. Entonces ellos podrían decir
'Estoy sorprendido por esta característica del lenguaje, asi que Ruby viola el principio de la menor sorpresa.' Esperad, esperad. El principio de la menor sorpresa no es solo para ti. El principio de la menor sorpresa significa el principio de 'mi' menor sorpresa. Y significa el principio de la menor sorpresa después de que aprendes bien Ruby. Por ejemplo, yo fui un programador de C++ antes de empezar a diseñar Ruby. Yo programé solamente en C++ durante dos o tres años. Y después de dos años de programar en C++, todavía me sorprendía.

Semántica

Ruby es orientado a objetos: todos los tipos de datos son un objeto, incluidas las clases y tipos que otros lenguajes definen como primitivas, (como enteros, booleanos, y "nil"). Toda función es un método. Las variables siempre son referencias a objetos, no los objetos mismos. Ruby soporta herencia con enlace dinámico, mixins y métodos singleton (pertenecientes y definidos por un sola instancia más que definidos por la clase).

A pesar de que Ruby no soporta herencia múltiple, la clases pueden importar módulos como mixins. La sintaxis procedural está soportada, pero todos los métodos definidos fuera del
ámbito de un objeto son realmente métodos de la clase Object. Como esta clase es padre de todas las demás, los cambios son visibles para todas las clases y objetos.

Ruby ha sido descrito como un lenguaje de programación multiparadigma: permite programación procedural (definiendo funciones y variables fuera de las clases haciéndolas parte del objeto raíz Object), con orientación a objetos, (todo es un objeto) o funcionalmente (tiene funciones anónimas, clausuras o closures, y continuations; todas las sentencias tiene valores, y las funciones devuelven la última evaluación). Soporta introspección, reflexión y metaprogramación, además de soporte para hilos de ejecución gestionados por el interprete. Ruby tiene tipificado dinámico, y soporta polimorfismo de tipos (permite tratar a subclases utilizando el interfaz de la clase padre). Ruby no requiere de polimorfismo de funciones (sobrecarga de funciones) al no ser fuertemente tipado (los parámetros pasados a un método pueden ser de distinta clase en cada llamada a dicho método). De acuerdo con las preguntas frecuentes de Ruby, "Si te gusta Perl, te gustará Ruby y su sintaxis. Si te gusta Smalltalk, te gustará Ruby y su semántica. Si te gusta Python, la enorme diferencia de diseño entre Python y Ruby/Perl puede que te convenza o puede que no."

Características

- orientado a objetos

- cuatro niveles de ámbito de variable: global, clase, instancia y local.

- manejo de excepciones

- iteradores y clausuras o closures (pasando bloques de código)

- expresiones regulares nativas similares a las de Perl a nivel del lenguaje

- posibilidad de redefinir los operadores (sobrecarga de operadores)

- recolección de basura

automática

- altamente portable

- Hilos de ejecución simultáneos en todas las plataformas usando *green threads*

- Carga dinámica de DLL/librerías compartidas en la mayoría de las plataformas

- introspección, reflexión y metaprogramación

- amplia librería estándar

- soporta inyección de dependencias

- soporta alteración de objetos en tiempo de ejecución

- continuaciones y generadores

Ruby actualmente no tiene soporte completo de Unicode, a pesar de tener soporte parcial para

UTF-8.

Smalltalk

Paradigma:	orientado a objetos
Apareció	Desarrollo comenzado en 1969. Públicamente
Diseñado	Alan Kay
Tipo de	dinámico
Implementaci	múltiples
Influido	Simula, Sketchpad, LISP
Ha influido	Objective-C, Java, Self, Python, Ruby,

Smalltalk es un sistema informático que permite realizar tareas de computación mediante la interacción con un entorno de objetos virtuales. Metafóricamente, se puede considerar que un Smalltalk es un mundo virtual donde viven objetos que se comunican mediante el envío de mensajes.

Un sistema Smalltalk está compuesto por:

- Máquina virtual
- Imagen virtual que contiene todos los objetos del sistema
- Lenguaje de programación (también conocido como Smalltalk)
- Biblioteca de Objetos reusables
- Opcionalmente un entorno de desarrollo que funciona como un sistema en tiempo de ejecución.

Historia

Los orígenes de Smalltalk se encuentran en las investigaciones realizadas por Alan Kay, Dan Ingalls, Ted Kaehler, Adele Goldberg y otros durante los años setenta en el Palo Alto Research Institute de Xerox (conocido como *Xerox PARC*), para la creación de un

sistema informático orientado a la educación. El objetivo era crear un sistema que permitiese expandir la creatividad de sus usuarios, proporcionando un entorno para la experimentación, creación e investigación.

Terminología

Un programa Smalltalk consiste únicamente de objetos, un concepto que se utiliza universalmente dentro de todo sistema Smalltalk. Prácticamente todo, desde un número natural como el 4 hasta un servidor web es considerado un objeto.

Los objetos Smalltalk presentan características comunes

- Tienen una memoria propia.

- Poseen capacidad para comunicarse con otros objetos.

- Poseen la capacidad de heredar características de objetos ancestros.

- Tienen capacidad de procesamiento.

Los objetos se comunican entre sí mediante el envío de mensajes. Un mensaje se envía entre un objeto emisor y un receptor, el objeto emisor pide una operación que el objeto receptor puede proporcionar. Asimismo, un objeto puede proveer muchas operaciones (actualmente esto está determinado por cada implementación).

Las definiciones de estas operaciones en los objetos son llamadas métodos. Un método especifica la reacción de un objeto cuando recibe un mensaje que es dirigido a ese método. La resolución (en el sentido de ligado) de un mensaje a un método es dinámica. La colección entera de métodos de un objeto es llamada protocolo de mensajes o interfaz de mensajes del objeto. Los mensajes pueden ser parametrizados, estos parámetros serán objetos, y el resultado o respuesta del mismo también será un objeto.

Las características comunes de una categoría de objetos está capturado bajo la noción de clase, de tal forma que los objetos agrupados bajo una clase son llamados instancias de ella. Las instancias son creadas durante la ejecución de un programa con algún propósito y son barridos automáticamente en el momento que no son necesitados más por el recolector de basura. Exceptuando algunos objetos especiales como los muy simples, llamados literales (números, cadenas, etc), cada objeto tiene su propio estado local y representa una instancia diferente de su clase.

Características

Se busca que todo lo que se pueda realizar con un ordenador se pueda realizar mediante un

Sistema Smalltalk, por lo que en algunos casos se puede considerar que incluso sirve como

Sistema Operativo.

Smalltalk es considerado el primero de los lenguajes orientados a objetos (OOP). En Smalltalk

todo es un objeto, incluidos los números reales o el propio entorno Smalltalk.

Como lenguaje tiene las siguientes características:

- Orientación a Objetos Pura
- Tipado dinámico
- Interacción entre objetos mediante envío de mensajes
- Herencia simple y con raíz común
- Reflexión computacional completa
- Recolección de basura
- Compilación en tiempo de ejecución o Interpretado (dependiendo de la distribución o del proveedor)
- Múltiples Implementaciones

Smalltalk ha tenido gran influencia sobre otros lenguajes como Java o Ruby, y de su entorno han surgido muchas de las prácticas y herramientas de desarrollo promulgadas actualmente por las metodologías ágiles (refactorización, desarrollo incremental, desarrollo dirigido por tests, etc.).

El entorno Smalltalk

Las implementaciones de Smalltalk de mayor peso (VisualWorks, Squeak, VisualSmalltalk, VisualAge y Dolphin) poseen un entorno de interacción muy diferente al entorno de desarrollo típico de otras tecnologías como Microsoft Visual Studio .Net o Eclipse. El entorno o ambiente Smalltalk es primordialmente gráfico y funciona como un sistema en tiempo de ejecución que integra varias herramientas de programación (Smalltalk), utilidades multimedia, interfaces para ejecutar código no nativo a Smalltalk y servicios del sistema operativo.

Estas posibilidades, que han influído en la metodología de trabajo y concepción de la programación, se traducen en la tendencia a considerar a Smalltalk más que un

simple lenguaje de programación. La forma de programar en Smalltalk no consiste en el ciclo típico de las tecnologías tradicionales: Arrancar un editor de texto, compilar y ejecutar y terminar la aplicación. En Smalltalk se manipula el entorno mismo, comúnmente mediante el Navegador del Sistema.

Sintaxis

Tradicionalmente, Smalltalk no posee una notación explícita para describir un programa entero. Sí se utiliza una sintaxis explícita para definir ciertos elementos de un programa, tales como métodos, pero la manera en que tales elementos están estructurados dentro de un
programa entero generalmente es definida por las múltiples implementaciones.

El estándar mismo no promueve otra dirección, por lo que define una sintaxis abstracta de programas Smalltalk, que define todos los elementos que constituyen un programa Smalltalk y la manera en que esos elementos están lógicamente compuestos por otros elementos, sin embargo, cada implementación es libre de definir y utilizar las muchas sintaxis posibles que están conformes a la sintaxis abstracta estándar. Un ejemplo de una sintaxis concreta es el Formato de Intercambio Smalltalk (o SIF, de Smalltalk Interchange Format) definida en el mismo estándar.

La sintaxis de Smalltalk-80 tiende a ser minimalista. Esto significa que existen un grupo chico de palabras reservadas y declaraciones en comparación con la mayoría de los lenguajes populares. Smalltalk posee un grupo de 5 palabras reservadas: self, super, nil, true y false.

Recolección de basura

En Smalltalk no es necesario desalocar objetos explícitamente, por lo tanto no proporciona mecanismos para ello. Las implementaciones utilizan técnicas de recolección de basura para detectar y reclamar espacio en memoria asociado con objetos que ya no se utilizarán más en el sistema. En Smalltalk la recolección de basura es integrada configurable. La forma de ejecución del recolector de basura es en *background*, es decir, como un proceso de baja prioridad no interactivo, aunque en algunas implementaciones es posible ejecutarlo a demanda, siendo posible definir configuraciones de memoria especiales para cada sistema mediante políticas (por ejemplo en VisualWorks). La frecuencia y características de la recolección depende de la técnica utilizada por la implementación. Adicionalmente algunas implementaciones de Smalltalk proporcionan soporte para mecanismos de finalización como el uso de Ephemerons.

Reflexión computacional

Smalltalk-80 provee reflexión computacional estructural y comportacional, ya que es un sistema implementado en sí mismo. La reflexión estructural se manifiesta en que las clases y métodos que define el sistema son en sí mismos objetos también y forman parte del sistema mismo. La mayoría de las implementaciones de Smalltalk tienden a exponer el compilador Smalltalk al entorno de programación, permitiendo que dentro del sistema se compile código fuente (textual), transformándose en objetos métodos, que son comúnmente instancias de la clase *CompiledMethod*. El sistema generalmente incorpora estos métodos en las clases,

almacenándolos en el diccionario de métodos de la clase a la cual se quiera agregar el comportamiento que realiza el método.

Esto, así como la incorporación de nuevas clases al sistema, es realizado dentro del sistema mismo; aunque la mayor parte de las implementaciones poseen herramientas visuales que ocultan la complejidad de interactuar con la clase que usualmente se encarga de tales tareas, el *ClassBuilder*.

La reflexión comportacional de Smalltalk-80 se manifiesta en la posibilidad de observar el estado computacional del sistema. En los lenguajes derivados del Smalltalk-80 original, durante el envío de mensajes entre objetos, cada objeto receptor de un mensaje consulta su clase para tener acceso a los métodos que define. En caso de encontrarse el método en la clase, se dice que se "activa" el método. Esta activación de un método actualmente en ejecución, es accesible mediante una palabra clave llamada thisContext. Enviando mensajes a thisContext se puede consultar cuestiones tales como "¿quién me envió este mensaje?".

Estas facilidades hacen posible implementar co-rutinas, continuaciones o back-tracking al estilo Prolog sin necesidad de modificar la máquina virtual. Uno de los usos más interesantes de esta facilidad, se da en el framework de web Seaside de Avi Bryant.

Ejemplos de Smalltalk

En Smalltalk todo es un objeto, y a un objeto se le envían mensajes. Por ejemplo:

```
1 + 1
```

Significa que al objeto "1" le enviamos el mensaje "+" con el colaborador externo, otro objeto, "1". Este ejemplo entonces resulta en el objeto "2".

```
Transcript show: '¡Hola, mundo!'
```

En el típico Hola mundo, el objeto es Transcript, que recibe el mensaje show con el colaborador externo '¡Hola, Mundo!'.

Para crear una instancia de un objeto, sólo hay que mandar un mensaje new a una clase: Objeto new

Para obtener las vocales de una cadena de texto:

```
'Esto es un texto' select: [:aCharacter | aCharacter isVowel].
```