CLASE 1: EL MODELO DE OBJETOS Y EL LENGUAJE JAVA

En esta primera clase veremos los fundamentos del modelo de objetos y conoceremos como surgió el lenguaje Java y cuales son sus características más salientes.

Ahora comenzaremos analizando el modelo de objetos, para esto, tendremos que definir que es la Programación el Diseño y el Análisis Orientado a Objetos, y podremos concluir que muchos lenguajes que dicen ser Orientados a Objetos no son tal.

Bueno, vamos a empezar.

1.1 FUNDAMENTOS DEL MODELO DE OBJETOS

Los métodos de diseño estructurados, surgieron para guiar a los desarrolladores que intentaban construir sistemas complejos utilizando los algoritmos como bloques fundamentales para su construcción.

Análogamente, los métodos de diseño orientados a objetos han surgido para ayudar a los desarrolladores a explotar la potencia expresiva de los lenguajes de programación basados en objetos y orientados a objetos, utilizando las clases y los objetos como bloques básicos de construcción.

En realidad, el modelo de objetos a recibido la influencia de una serie de factores, no solo de la programación orientada a objetos. Por el contrario, el modelo de objetos ha mostrado ser un concepto unificador en la informática, aplicable no solo a los lenguaje de programación, si no también al diseño de interfaces de usuario, bases de datos e inclusos arquitecturas de computadores. La razón para este gran atractivo es simplemente que una orientación a objetos ayuda a combatir la complejidad inherente a muchos tipos de sistemas diferentes.

El diseño orientado a objetos representa así un desarrollo evolutivo, no revolucionario; no rompe con los avances del pasado, si no que se basa en avances ya probados.

Desgraciadamente, hoy en día la mayoría de los programadores han sido educados formal e informalmente solo en los principios del diseño estructurado. Sin embargo, existen límites para la cantidad de complejidad que se puede manejar utilizando solo descomposición algorítmica; por tanto hay que volverse hacia la descomposición orientada a objetos. Es más, si se intenta utilizar lenguajes orientados a objetos como si fuesen lenguajes tradicionales orientados algorítmicamente, no solo se pierde la potencia de la que se disponía, si no que habitualmente se acaba en una situación peor que si se hubiese utilizado un lenguaje estructurado.

1.1.1 Programación Orientada a Objetos (POO)

¿Qué es entonces la Programación Orientada a Objeto? Aquí va a definirse como sigue:

La programación orientada a objetos es un método de implementación en el que los programas se organizan como conjuntos cooperativos de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son, todas ellas, miembros de una jerarquía de clases unidas mediante relaciones de herencia.

Hay tres partes importantes en esta definición: la programación orientada a objetos

> **(1)** utiliza Objetos, no algoritmos, como sus bloques lógicos de construcción fundamentales (la jerarquía "parte de") ;
>
> **(2)** cada objeto es una instancia de alguna clase;
>
> **(3)** las clases están relacionadas con otras clases por medio de relaciones de herencia (la jerarquía "de clases").

Un programa nos puede parecer orientado a objetos, pero si falta cualquiera de estos elementos, no lo es. La programación sin herencia es explícitamente no orientada a objetos y se denomina Programación con tipos abstractos de datos.

Según esta definición algunos lenguajes son orientados a objetos y otros no lo son. Podemos sugerir que "si el término "Orientado a Objetos" significa algo, debe significar un lenguaje que tiene mecanismos que soportan bien el estilo de programación orientada a objetos... Un lenguaje soporta bien un estilo de programación si proporciona capacidades que hacen conveniente utilizar tal estilo. Un lenguaje no soporta una técnica si exige un esfuerzo o habilidad excepcionales escribir tales programas. En ese caso, el lenguaje se limita a permitir a los programadores el uso de esas técnicas".
Desde una perspectiva teórica, podemos fingir programación orientada a objetos en lenguajes de programación no orientados a objetos, pero no tiene ningún sentido hacerlo. Debemos destacar que cierto lenguaje es orientado a objetos si y solo si satisface los siguientes requisitos:

> Soporta objetos que son abstracciones de datos con una interfaz de operaciones con nombres y un estado local oculto.
> Los objetos tienen un tipo asociado [clases].
> Los tipos [clases] pueden heredar atributos a los supertipos [superclases].

Para un lenguaje, el soportar la herencia significa que es posible expresar relaciones "es un" entre tipos, por ejemplo, una rosa roja es un tipo de flor, y una flor es un tipo de planta. Si un lenguaje no ofrece soporte directo para la herencia, entonces no es Orientado a Objetos.

1.1.2 Diseño Orientado a Objetos (DOO)

El énfasis en los métodos de programación está puesto principalmente en el uso correcto y efectivo de mecanismos particulares del lenguaje que se utiliza. Por contraste, los métodos de diseño enfatizan la estructuración correcta y efectiva de un sistema complejo ¿Qué es entonces el Diseño Orientado a Objetos?

Podemos sugerir que:

> *El Diseño Orientado a Objetos es un método de diseño que abarca el proceso de descomposición orientada a objetos y una notación para describir los modelos lógico y físico, así como los modelos estático y dinámico del sistema que se diseña.*

Hay dos partes importantes en esta definición: el Diseño Orientado a Objetos

> **(1)** da lugar a una Descomposición Orientada a Objetos y
>
> **(2)** utiliza diversas notaciones para expresar diferentes modelos del diseño lógico (estructura de clases y objetos) y físico (arquitectura de módulos y procesos) de un sistema, además de los aspectos estáticos y dinámicos del sistema.

El soporte para la Descomposición Orientado a Objetos es lo que hace al Diseño Orientado a Objetos bastante diferente del Diseño Estructurado: el primero utiliza abstracciones de clases y objetos para estructurar lógicamente los sistemas y el segundo utiliza abstracciones algorítmicas. Se utilizará el término Diseño Orientado a Objetos para referirse a cualquier método que encamine a una Descomposición Orientado a Objetos.

1.1.3 Análisis Orientado a Objetos (AOO)

El modelo de objetos ha influido incluso en las fases iniciales del ciclo de vida del desarrollo del software. Las técnicas de análisis estructurado tradicionales, con extensiones para tiempo real, se centran en el flujo de datos dentro de un sistema. El Análisis Orientado a Objetos enfatiza la construcción de modelos del mundo real, utilizando una visión del mundo orientado a objetos:
El Análisis Orientado a Objetos es un método de análisis que examina los registros desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema.
¿Cómo se relacionan AOO, DOO y POO? Básicamente, los productos del Análisis Orientado a Objetos sirven como modelos de los que se pueden partir para un Diseño Orientado a Objetos; los productos del Diseño Orientado a Objetos pueden utilizarse entonces como anteproyectos para la implementación completa de un sistema utilizando métodos de Programación Orientado a Objetos.

1.2 ELEMENTOS DEL MODELO DE OBJETOS

Luego de analizar las características del Modelo de Objetos, ahora nos detendremos en las ventajas

1.2.1 Tipos de Paradigmas de Programación

La mayoría de los programadores trabajan en un lenguaje y utilizan sólo un estilo de programación. Programan bajo un paradigma apoyado por el lenguaje que usan. Frecuentemente no se les ha mostrado vías alternativas para pensar sobre un problema, y por tanto tienen dificultades para apreciar las ventajes de elegir un estilo más apropiado para el problema que tienen entre manos.

Un estilo de programación es: Una forma de organizar programas sobre las bases de algún modelo conceptual de programación y un lenguaje apropiado para que resulten claros los programas escritos en ese estilo.

Hay cinco tipos principales de estilos de programación que se listan aquí con los tipos de abstracciones que emplean:

Orientados a Procedimientos Algoritmos
Orientados a Objetos Clases y Objetos
Orientados a Lógica Objetivos, a menudo expresados como Cálculo de Predicados
Orientados a Reglas Reglas si-entonces (if/then)
Orientados a Restricciones Relaciones Invariantes

No hay un estilo de programación que sea el mejor para todo tipo de aplicaciones. Por ejemplo, la Programación Orientada a Reglas sería la mejor para el diseño de una Base de Conocimientos y la Programación Orientada a Procedimientos sería la más indicada para el diseño de operaciones de cálculo intensivo. Por nuestra experiencia, el estilo Orientado a Objetos es el más adecuado para el más amplio conjunto de aplicaciones; realmente, este paradigma de programación sirve con frecuencia como el marco de referencia arquitectónico en el que se emplean otros paradigmas.

Cada uno de esos estilos de programación se basa en su propio marco de referencia conceptual. Cada uno requiere una actitud mental diferente, una forma distinta de pensar en el problema. Para todas las cosas Orientadas a Objetos, el marco de referencia conceptual es el Modelo de Objetos. Hay cuatro elementos fundamentales en este modelo:

Abstracción
Encapsulamiento
Modularidad
Jerarquía

Al decir Fundamentales, quiere decirse que un modelo que carezca de cualquiera de estos elementos no es Orientado a Objetos.

Hay tres elementos secundarios del modelo de objetos:

Tipos (Tipificaciones)
Concurrencia
Persistencia

Por Secundarios quiere decirse que cada uno de ellos es una parte útil del modelo de objetos, pero no es esencial.

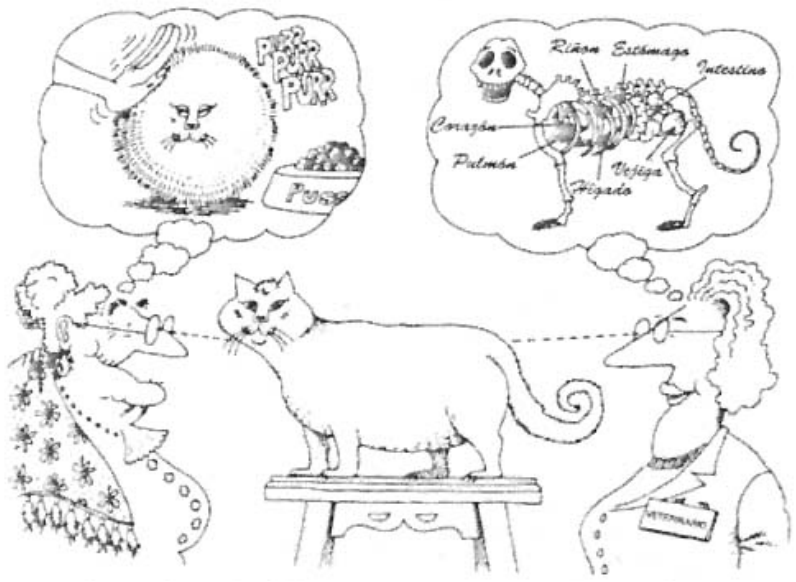
A continuación analizaremos cada uno de los elementos del modelo Orientado a Objetos.

1.2.2 Abstracción

La abstracción es una de las vías fundamentales por la que los humanos combatimos la complejidad. La abstracción surge de un reconocimiento de las similitudes entre ciertos objetos, situaciones o procesos del mundo real, y la decisión de concentrarse en esas similitudes e ignorar por el momento las diferencias.

También pueden ver la Abstracción como una descripción simplificada o una especificación de un sistema que enfatiza algunos de los detalles o propiedades del mismo mientras suprime otros. Una buena abstracción es aquella que enfatiza detalles significativos al lector o usuario y suprime detalles que son, al menos por el momento, irrelevantes o causa de distracción. Un ejemplo de lo que estamos afirmando puede verse a continuación.

En esta imagen podemos observar como dos personas ven un mismo objeto (un gato) pero cada una lo abstrae según las características que conoce del mismo.



Combinando estas ideas, definimos una abstracción del modo siguiente:

> *Una Abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipo de objetos y*

proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador.

Una abstracción se centra en la visión externa de un objeto y, por tanto, sirve para separar el comportamiento esencial de un objeto de su implementación. La decisión sobre el conjunto adecuado de abstracciones para determinado dominio es el problema central del diseño orientado a objetos.

Existe una gama de abstracción, desde los objetos que modelan muy de cerca entidades del dominio del problema a objetos que no tienen una verdadera razón de existir. De mayor a menor utilidad, estos tipos de abstracción incluyen:

Abstracción de entidades.	Un objeto que representa un modelo útil de una entidad del dominio del problema o del dominio de la solución.
Abstracción de acciones.	Un objeto que proporciona un conjunto generalizado de operaciones, y todas ellas desempeñan funciones del mismo tipo.
Abstracción de Máquinas Virtuales	Un objeto que agrupa operaciones, todas ellas virtuales utilizadas por algún nivel superior de control, u operaciones que utilizan todas algún conjunto de operaciones de nivel inferior.
Abstracción de Coincidencia	Un objeto que almacena un conjunto de operaciones que no tiene relación entre sí.

Se persigue construir abstracciones de entidades, porque imitan directamente el vocabulario de un determinado dominio de problema.

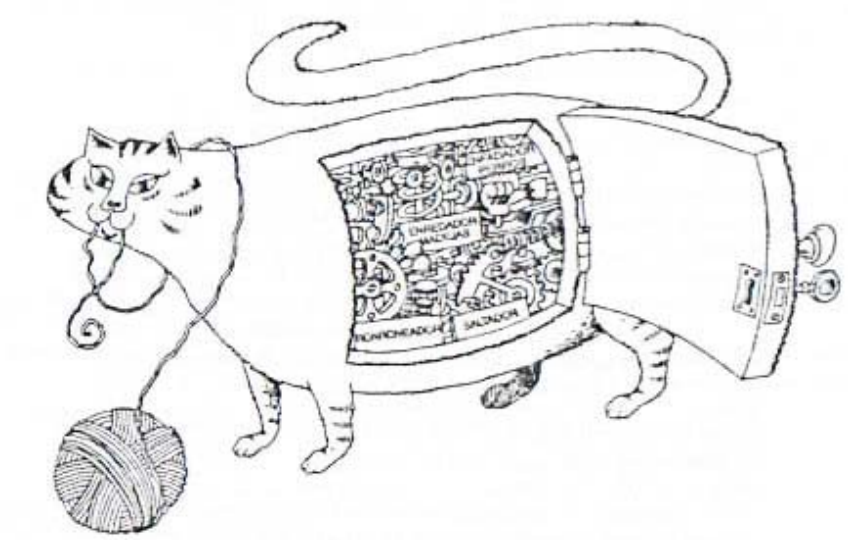
1.2.3 Encapsulamiento

Pasemos ahora a Encapsulamiento. La abstracción y el encapsulamiento son conceptos complementarios: la abstracción se centra en el comportamiento observable de un objeto, mientras que el encapsulamiento se centra en la implementación que da lugar a este comportamiento. El encapsulamiento se consigue a menudo mediante el ocultamiento de información, que es el proceso de ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales es decir, la estructura de un objeto está oculta, así como la implementación de sus métodos.

El Encapsulamiento proporciona barreras explicitas entre abstracciones diferentes y por tanto conduce a una clara separación de interés por ejemplo, considérese la estructura de una planta. Para comprender como funciona la fotosíntesis a un nivel alto de abstracción se pueden ignorar detalles como los cometidos de las raíces de la planta o la química de las paredes de las células. Análogamente, al diseñar una aplicación de bases de datos, es práctica común el escribir programas de forma que no se preocupen de la representación física de los datos, si no que dependan sólo de un esquema que denota la vista

lógica de los mismo. En ambos casos, los objetos a un nivel de abstracción están protegidos de los detalles de implementación a niveles más bajos de abstracción.

En la figura se muestra un objeto que exteriormente se muestra como un gato pero interiormente es un robot, a esto apunta el encapsulamiento.



Para que la abstracción funcione, la implementación debe estar encapsulada. En la práctica, esto significa que cada clase debe tener dos partes: una interfaz y una implementación. La interfaz de una clase captura solo su vista interna abarcando la abstracción que se ha hecho del comportamiento común de todas las instancias de la clase. La implementación de una clase comprende la representación de la abstracción así como los mecanismos que consiguen el comportamiento deseado. La interfaz de una clase es el único lugar en el que se declaran todas las suposiciones que un cliente puede hacer acerca de todas las instancias de la clase; la implementación encapsula detalles acerca de los cuales ningún cliente puede realizar suposiciones.

Para resumir, se define el Encapsulamiento como sigue:

> *El Encapsulamiento es el proceso de almacenar en un mismo compartimiento los elementos de una abstracción que constituyen su estructura y su comportamiento; sirve para separar la interfaz de una abstracción y su implementación.*

1.2.4 Modularidad

El acto de fragmentar un programa en componentes individuales puede reducir su complejidad en algún grado... Aunque la fragmentación de programas es útil por esta razón una justificación más poderosa para esta fragmentación es que crea una serie de fronteras bien definidas y documentadas dentro del programa.

Estas fronteras o interfaces, tiene un incalculable valor cara a la comprensión del programa. En algunos lenguajes no existe el concepto de módulo, y así la

clase es la única unidad física de descomposición. En muchos otros, incluyendo Object Pascal, C++ y Ada, el módulo es una construcción adicional del lenguaje y por lo tanto, justifica un conjunto separado de decisiones de diseño. En estos lenguajes, las clases y los objetos forman la estructura lógica de un sistema; se sitúan estas abstracciones en módulos para producir la arquitectura física del sistema. Especialmente para aplicaciones más grandes, en las que puede haber varios cientos de clases, el uso de módulos es esencial para ayudar a manejar la complejidad.

La Modularidad consiste en dividir un programa en módulos que pueden compilarse separadamente, pero que tienen conexiones con otros módulos. Las conexiones entre módulos son las suposiciones que cada módulo hace acerca de todos los demás. La mayoría de los lenguajes que soportan el módulo como un concepto adicional distinguen también entre la interfaz de un módulo y su implementación. Así, es correcto decir que la Modularidad y el Encapsulamiento van de la mano. Como en el Encapsulamiento, los lenguajes concretos soportan la Modularidad de formas diversa.

La decisión sobre un conjunto adecuado de módulos para determinado problema es un problema casi tan difícil como decidir sobre el conjunto adecuado de abstracciones.

1.2.5 Jerarquía

La abstracción es algo bueno, pero excepto en las aplicaciones más triviales, puede haber muchas más abstracciones diferentes de las que se pueden comprender simultáneamente. El Encapsulamiento ayuda a manejar esta complejidad ocultando la visión interna de las abstracciones. La Modularidad también ayuda, ofreciendo una vía para agrupar abstracciones relacionadas lógicamente. Esto sigue sin ser suficiente.

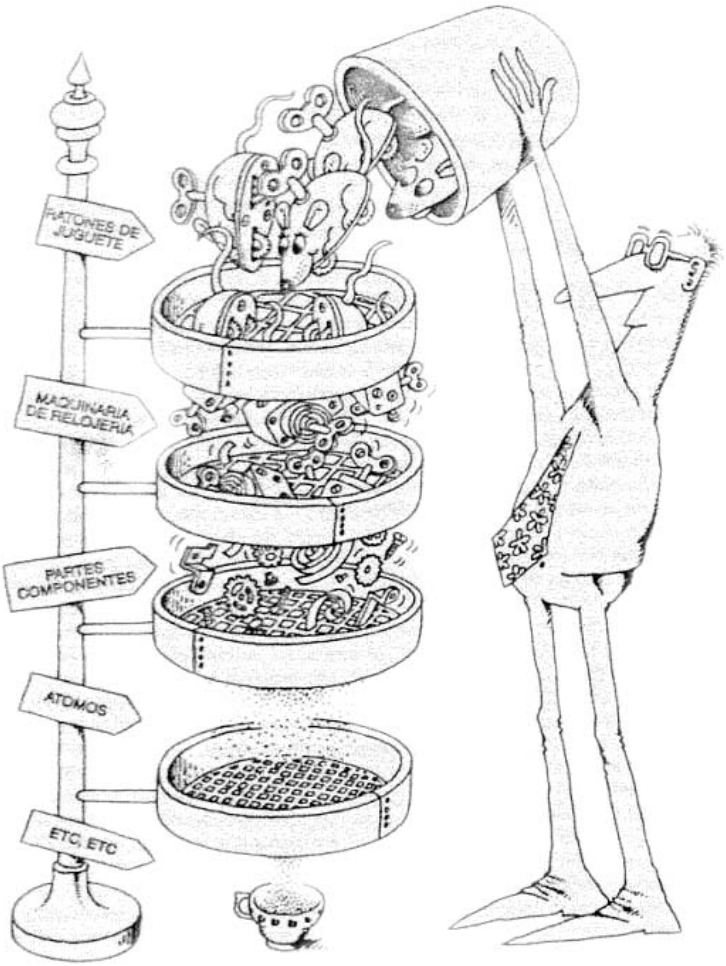
Frecuentemente un conjunto de abstracciones forma una jerarquía, y la identificación de esas jerarquías en el diseño simplifica en gran medida la comprensión del problema.

Se define la jerarquía de la siguiente manera:

La jerarquía es una clasificación u ordenación de abstracciones.

Las dos jerarquías más importantes en un sistema complejo son su estructura de clases (la jerarquía "de clases") y su estructura de objetos (la jerarquía "de partes").

La herencia es la jerarquía "de clases" más importantes y, como se apuntó anteriormente, es un elemento esencial de los sistemas orientados a objetos. Básicamente, la herencia define una relación entre clases, en la que una clase comparte la estructura de comportamiento definida en una o más clases (lo que se denomina herencia simple o herencia múltiple, respectivamente). La herencia representa así una jerarquía de abstracciones, en la que una subclase hereda de una o más superclases. Típicamente una subclase aumenta o redefine la estructura y el comportamiento de sus superclases.



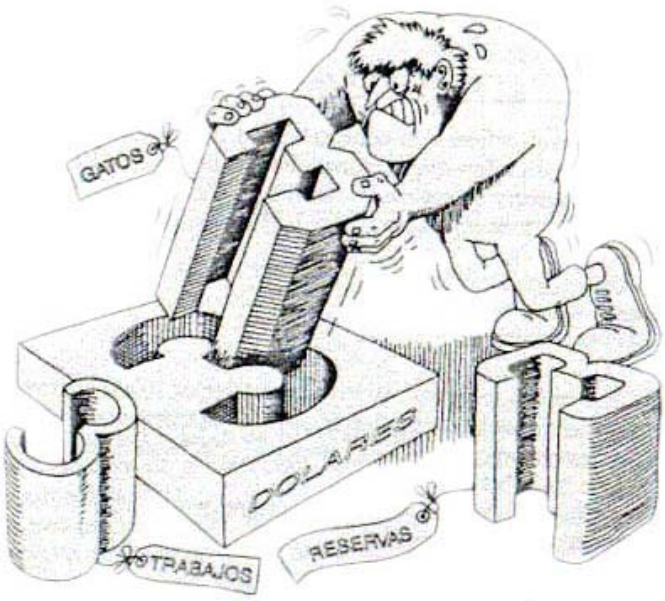
A medida que se desarrolla la jerarquía de herencias, la estructura y comportamiento comunes a diferentes clases tenderá a migrar hacia superclases comunes. Por esta razón se habla a menudo de la herencia como una jerarquía de generalización / especialización. Las superclases representan abstracciones generalizadas, y las subclases representan especializaciones en que los campos y métodos de la superclase sufren añadidos, modificaciones o incluso ocultaciones. De este modo, la herencia permite declarar las abstracciones con economía de expresión. De hecho, el ignorar las jerarquías

"de clases" que existen pueden conducir a diseños deformados y poco elegantes. Sin herencia, cada clase sería una unidad independiente desarrollada partiendo de cero. Las distintas clases no guardarían relación entre sí, puesto que el desarrollador de cada clase proporcionaría métodos según le viniese en gana. Toda consistencia entre clases es el resultado de una disciplina por parte de los programadores. La herencia posibilita la definición de nuevo software de la misma forma en que se presenta un concepto a un recién llegado, comparándolo con algo que ya le resulte familiar.

Existe una conveniente tensión entre los principios de abstracción, encapsulamiento y jerarquía. La abstracción de datos intenta proporcionar una barrera opaca tras de la cual se ocultan los métodos y el estado; la herencia requiere abrir esta interfaz en cierto grado y puede permitir el acceso a los métodos y al estado sin abstracción. Para una clase dada, habitualmente existen dos tipos de cliente: objetos que invocan operaciones sobre instancias de la clase, y subclases que heredan de esta clase. Con la herencia puede violarse el encapsulamiento en tres formas: la subclase podría acceder a una variable de instancia de su superclase llamar a una operación privada de su superclase o referenciar directamente a superclases de su superclase. Los distintos lenguajes de programación hacen concesiones entre el apoyo del Encapsulamiento y de la herencia de diferentes formas, la interfaz de una clase puede tener tres partes: partes private (privadas), que declaran miembros accesibles solo a la propia clase, partes protected (protegidas) que declaran miembros accesibles solo a la clase y sus subclases, y partes public (publicas), accesibles a todos los cliente.

1.2.6 Tipos(tipificación)

El concepto de tipo se deriva en primer lugar de las teorías sobre los tipos abstractos de datos. Para nuestros propósitos, se utilizaran los términos tipo y clase de manera intercambiable. Aunque los conceptos de clase y tipo son similares se incluyen los tipos como elemento separado del modelo de objetos porque el concepto de tipo pone énfasis en el significado de la abstracción en un sentido muy distinto.



En concreto, se establece lo siguiente:

Los tipos son la puesta en vigor de la clase de los objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restringidas.

Los tipos permiten expresar las abstracciones de manera que el lenguaje de programación en que se implanta puede utilizarse para apoyar las decisiones de diseño.

1.2.7 Concurrencia

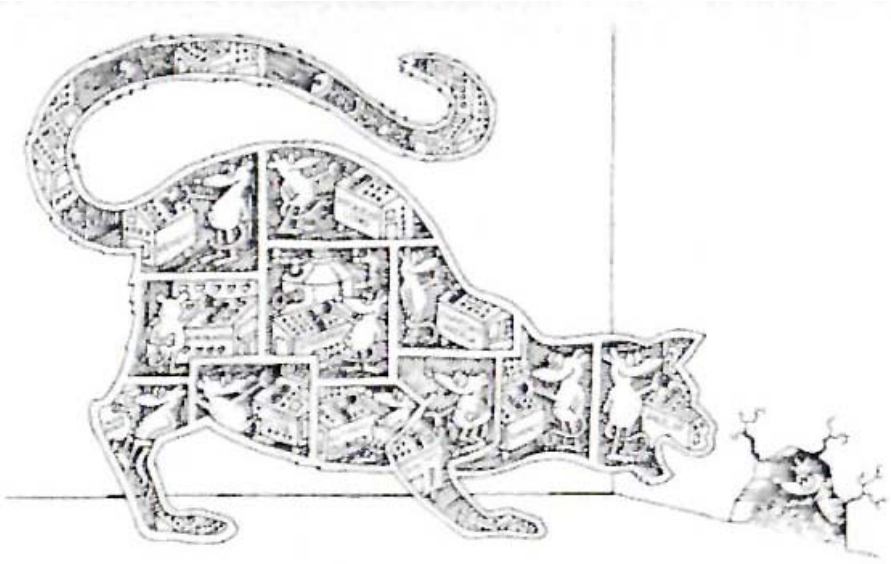
Para ciertos tipos de problemas, un sistema automatizado puede tener que manejar muchos eventos diferentes simultáneamente. Otros problemas pueden implicar tantos cálculos que excedan la capacidad de cualquier procesador individual. En ambos casos, es natural considerar el uso de un conjunto distribuido de computadores para la implementación que se persigue o utilizar procesadores capaces de realizar multitarea.

Un solo proceso - denominado hilo de control - es la raíz a partir de la cual se producen acciones dinámicas independientes dentro del sistema. Todo programa tiene al menos un hilo de control, pero un sistema que implique concurrencia puede tener muchos de tales hilos: algunos son transitorios, y otros permanecen durante todo el ciclo de vida de la ejecución del sistema. Los sistemas que se ejecutan en múltiples CPUs permiten hilos de control verdaderamente concurrentes, mientras que los sistemas que se ejecutan en una sola CPU solo pueden conseguir la ilusión de hilos concurrentes de control, normalmente mediante algún algoritmo de tiempo compartido.

Se distinguen también entre concurrencia pesada y ligera. Un proceso pesado es aquel típicamente manejado de forma independiente por el sistema

operativo de destino, y abarca su propio espacio de direcciones. Un proceso ligero suele existir dentro de un solo proceso del sistema operativo en compañía de otros procesos ligeros, que comparten el mismo espacio de direcciones. La comunicación entre procesos pesados suele ser costosa, involucrando a alguna forma de comunicación inter-proceso; la comunicación entre procesos ligeros es menos costosa, y suele involucrar datos compartidos.

Mientras que la programación orientada a objetos se centra en la abstracción de datos, Encapsulamiento y herencia, la concurrencia se centra en la abstracción de procesos y de sincronización. El objeto es un concepto que unifica estos dos puntos de vista distintos: cada objeto(extraído de una abstracción del mundo real) puede representar un hilo separado de control (una abstracción de un proceso). Tales objetos se llaman activos. En un sistema basado en Diseño orientado a objetos, se puede conceptuar el mundo como un conjunto de objetos cooperativos, algunos de los cuales son activos y sirven así como centros de actividad independientes. Partiendo de esta concepción se define la concurrencia como sigue:
La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo



1.2.8 Persistencia

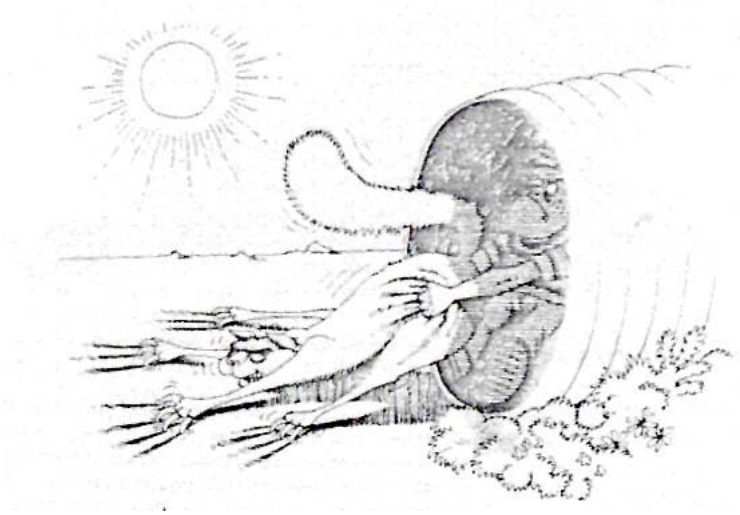
Un objeto de software ocupa una cierta cantidad de espacio, y existe una cierta cantidad de tiempo.
La unificación de los conceptos de concurrencia y objetos da lugar a los lenguajes concurrentes de programación orientada a objetos. De manera similar, la introducción de conceptos de persistencia en el modelo de objetos da lugar a la aparición de bases de datos orientadas a objetos.

Muy pocos lenguajes de programación orientados a objetos ofrecen soporte directo para la persistencia. Frecuentemente la persistencia se consigue a través de un pequeño número de bases de datos orientadas a objetos disponibles comercialmente. Otro enfoque razonable para la persistencia es proporcionar una piel orientada a objetos bajo la cual se oculta una base de datos relacional. Esta aproximación es más atractiva si existe una gran

inversión de capital en tecnología de bases de datos relacionales que sería arriesgado o demasiado caro reemplazar.

La persistencia abarca algo más que la mera duración de los datos. En las bases de datos orientadas a objetos, no solo persiste el estado de un objeto, sino que su clase debe trascender también a cualquier programa individual, de forma que los programas interpreten de la misma manera el estado almacenado.
Para resumir, se define la persistencia como:

La persistencia es la propiedad de un objeto por la que su existencia trasciende el tiempo (es decir, el objeto continua existiendo después de que su creador deja de existir) y/o el espacio (es decir, la posición del objeto varía con respecto al espacio de direcciones en el que fue creado).



1.3 APLICACIÓN DEL MODELO DE OBJETOS

El modelo de objetos es fundamentalmente diferente de los modelos adoptados por los métodos más tradicionales de análisis estructurado, diseño estructurado y programación estructurada. Esto no significa que el modelo de objetos abandone todos los sanos principios y experiencias de métodos más viejos. En lugar de eso, introduce varios elementos novedosos que se basan en estos modelos anteriores. Así, el modelo de objetos proporciona una serie de beneficios significativos que otros modelos simplemente no ofrecen.

Para terminar esta introducción al modelo de objetos, sería interesante que escriban en el Block de notas de la clase una opinión sobre el tema.

Continuaremos ahora, analizando como se gestó el lenguaje Java y las características que lo transforman en un verdadero lenguaje de Programación Orientada a Objetos.

1.4 ORIGEN DE JAVA

Sun Microsystems, líder en servidores para Internet, uno de cuyos lemas desde hace mucho tiempo es "the network is the computer" (lo que quiere dar a

entender que el verdadero ordenador es la red en su conjunto y no cada máquina individual), es quien ha desarrollado el lenguaje Java, en un intento de resolver simultáneamente todos los problemas que se le plantean a los desarrolladores de software por la proliferación de arquitecturas incompatibles, tanto entre las diferentes máquinas como entre los diversos sistemas operativos que funcionaban sobre una misma máquina, añadiendo la dificultad de crear aplicaciones distribuidas en una red como Internet.
Hace algunos años, Sun Microsystems decidió intentar introducirse en el mercado de la electrónica de consumo y desarrollar programas para pequeños dispositivos electrónicos. Tras unos comienzos dudosos, Sun decidió crear una filial, denominada FirstPerson Inc., para dar margen de maniobra al equipo responsable del proyecto. El mercado inicialmente previsto para los programas de FirstPerson eran los equipos domésticos: microondas, tostadoras y, fundamentalmente, televisión interactiva. Este mercado, dada la falta de pericia de los usuarios para el manejo de estos dispositivos, requería interfaces mucho más cómodas e intuitivas que los sistemas que proliferaban en el momento. Otros requisitos importantes a tener en cuenta eran la fiabilidad del código y la facilidad de desarrollo. James Gosling, el miembro del equipo con más experiencia en lenguajes de programación, decidió que las ventajas aportadas por la eficiencia de C++ no compensaban el gran costo de pruebas y depuración. Gosling había estado trabajando en su tiempo libre en un lenguaje de programación que él había llamado Oak, el cual, aún partiendo de la sintaxis de C++, intentaba remediar las deficiencias que iba observando.

Los lenguajes al uso, como C o C++, deben ser compilados para un chip, y si se cambia el chip, todo el software debe compilarse de nuevo. Esto encarece mucho los desarrollos. La aparición de un chip más barato y, generalmente, más eficiente, conduce inmediatamente a los fabricantes a incluirlo en las nuevas series de sus cadenas de producción, por pequeña que sea la diferencia en precio ya que, multiplicada por la tirada masiva de los aparatos, supone un ahorro considerable. Por tanto, Gosling decidió mejorar las características de Oak y utilizarlo.
El primer proyecto en que se aplicó este lenguaje recibió el nombre de proyecto Green y consistía en un sistema de control completo de los aparatos electrónicos en el entorno de un hogar. Para ello se construyó un ordenador experimental denominado *7 (Star Seven- Estrella Siete). El sistema presentaba una interfaz basada en la representación de la casa de forma animada y el control se llevaba a cabo mediante una pantalla sensible al tacto. En el sistema aparecía Duke, la actual mascota de Java. Posteriormente se aplicó a otro proyecto denominado VOD (Video On Demand) en el que se empleaba como interfaz para la televisión interactiva. Ninguno de estos proyectos se convirtió nunca en un sistema comercial, pero fueron desarrollados enteramente en un Java primitivo y fueron su bautismo de fuego. Una vez que en Sun se dieron cuenta de que a corto plazo la televisión interactiva no iba a ser un gran éxito, urgieron a FirstPerson a desarrollar con rapidez nuevas estrategias que produjeran beneficios. No lo consiguieron y FirstPerson cerró en la primavera de 1994.

Pese a lo que parecía ya un olvido definitivo, Bill Joy, cofundador de Sun y uno de los desarrolladores principales del Unix de Berkeley, juzgó que Internet

podría llegar a ser el campo de juego adecuado para disputar a Microsoft su primacía casi absoluta en el terreno del software, y vio en Oak el instrumento idóneo para llevar a cabo estos planes. Tras un cambio de nombre y modificaciones de diseño, el lenguaje Java fue presentado en sociedad en agosto de 1995.

1.5 El entorno de desarrollo de Java

Existen distintos programas comerciales que permiten desarrollar código Java. El que usaremos a lo largo del curso es el que la compañía Sun distribuye gratuitamente y se denomina Java(tm) Development Kit (JDK). Se trata de un conjunto de programas y librerías que permiten desarrollar, compilar y ejecutar programas en Java. Incorpora además la posibilidad de ejecutar parcialmente el programa, deteniendo la ejecución en el punto deseado y estudiando en cada momento el valor de cada una de las variables (con el denominado Debugger). Cualquier programador con un mínimo de experiencia sabe que una parte muy importante (muchas veces la mayor parte) del tiempo destinado a la elaboración de un programa se destina a la detección y corrección de errores. Existe también una versión reducida del JDK, denominada JRE (Java Runtime Environment) destinada únicamente a ejecutar código Java (no permite compilar).

Los IDEs (Integrated Development Environment), tal y como su nombre indica, son entornos de desarrollo integrados. En un mismo programa es posible escribir el código Java, compilarlo y ejecutarlo sin tener que cambiar de aplicación. Algunos incluyen una herramienta para realizar Debug gráficamente, frente a la versión que incorpora el JDK basada en la utilización de una consola (denominada habitualmente ventana de comandos de MS-DOS, en Windows NT/95/98) bastante difícil y pesada de utilizar. Estos entornos integrados permiten desarrollar las aplicaciones de forma mucho más rápida, incorporando en muchos casos librerías con componentes ya desarrollados, los cuales se incorporan al proyecto o programa. Como inconvenientes se pueden señalar algunos fallos de compatibilidad entre plataformas, y ficheros resultantes de mayor tamaño que los basados en clases estándar.

1.6 CARACTERÍSTICAS DE JAVA

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

1.6.1 Simple

Java ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos. C y C++ son lenguajes muy 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 para mantener reducidas las especificaciones del lenguaje y añadir características muy útiles como el garbage collector (recolector de memoria dinámica). No es necesario preocuparse de liberar memoria, el recolector se encarga de ello

1.6.2 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 nos permitira acceder a la información a través de la red con tanta facilidad como a los ficheros locales.

La verdad es que 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.

1.6.3 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.

1.6.4 Arquitectura Neutral

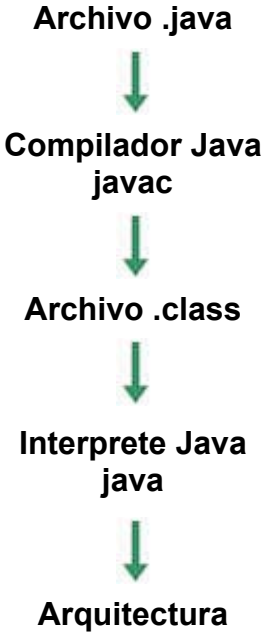
Algunos deben haber escuchado que Java es portable o multiplataforma o independiente de la plataforma o de arquitectura neutral, ¿Qué significa esto? Una vez que escribimos un programa en el lenguaje Java tenemos un archivo con extensión .java.

Ese archivo debe ser compilado, esto es, se debe verificar que no contenga errores de sintaxis ni algunos lógicos y se transforma el código que habíamos escrito a un código entendible para la máquina, esto es lo que sucede en casi todos los lenguajes, en Java esto es sutilmente diferente. En realidad, cuando se compila mediante la sentencia
javac nombredelarchivo.java
donde javac es la abreviatura de Java Compiler, lo que se obtiene es un archivo con extensión .class que no está escrito en el código de una máquina particular porque, si lo estuviera, al querer llevarlo a otra máquina con un sistema operativo o una arquitectura distinta no andaría, habría que volver a compilarlo para obtener un archivo .class que anduviera para esa máquina.

¿Cómo hace Java para eliminar ese problema y transformarse en un lenguaje independiente de la Arquitectura de la máquina? Sencillamente, el archivo .class está en un código que va a ser ejecutado por un interprete (interpreta el código del archivo .class denominado bytecode) que sí varía según la arquitectura y el sistema operativo. Ese interprete se lo conoce como la Máquina Virtual de Java (JVM).

Con este funcionamiento, se escribe un programa, se lo compila, y ese archivo puede funcionar en cualquier sistema (para el que haya una máquina virtual).

Para aclarar un poco más el tema vean el siguiente modelo:



1.6.5 Seguro

La seguridad en Java tiene dos facetas. 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.

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.

El código Java pasa muchos tests antes de ejecutarse en una máquina. El código se pasa a través de un verificador de byte-codes 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-.
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. Además, para evitar modificaciones por parte de los crackers de la red, implementa un método ultraseguro de autentificación por clave pública. 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 de compilación, con el nivel de detalle y de privilegio que sea necesario.
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 ficheros de la máquina local (siempre que se realizan operaciones con archivos, éstas trabajan sobre el disco duro de la máquina de donde partió el programa), no permite ejecutar aplicaciones nativa 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, se puede considerar que Java es un lenguaje seguro.
Respecto a la seguridad del código fuente, no ya del lenguaje, JDK proporciona un desemsamblador de byte-code, que permite 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. Esta es otra de las cuestiones que Java tiene pendientes.

1.6.6 Interpretado

El intérprete Java puede ejecutar directamente el código objeto. Enlazar 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. Por ahora 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.

La verdad es que ya hay comparaciones ventajosas entre Java y el resto de los lenguajes de programación,.

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.

1.6.7 Multithreaded

Al ser multithreaded (permite muchos hilos de control, ver Concurrencia), Java permite muchas actividades simultáneas en un programa. Los threads (a veces llamados, procesos ligeros), son básicamente pequeños procesos o piezas independientes de un gran proceso.

El beneficio de ser miltithreaded 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.), 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. En Java, las imágenes se pueden ir trayendo en un thread independiente, permitiendo que el usuario pueda acceder a la información en la página sin tener que esperar por el navegador.

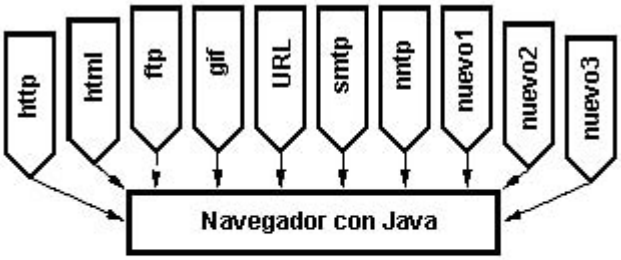
1.6.8 Dinámico

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 tiempo de ejecución. Las librería nuevas o actualizadas no paralizarán 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 cualquiera de esas piezas que el sistema necesita para funcionar.

Java, para evitar que los módulos de byte-codes o los objetos o nuevas clases, haya que estar trayéndolos 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.

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



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