UNIDAD

1

DIPLOMATURA EN PROGRAMACION JAVA

MÓDULO 1: PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA

Fundamentos de Clases y Objetos

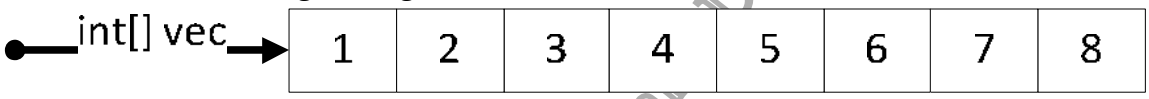
Fundamentos de Clases y Objetos

Este módulo se explica los fundamentos que dieron origen a la creación de clases y objetos en los lenguajes de programación. También se enseña la forma de descubrir objetos y crear clases modelando los objetos descubiertos

Estructuras de Datos

Vectores

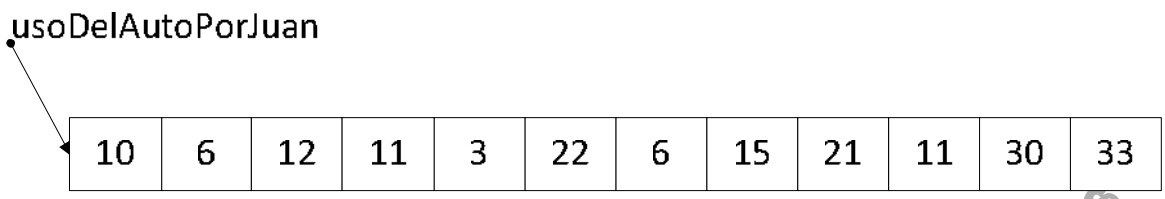
Desde que se comenzaron a utilizar vectores, siempre se buscó agrupar los datos bajo un mismo nombre que definiera a un conjunto de ellos. De esta manera, un vector es un conjunto de datos del mismo tipo (por ejemplo, entero) agrupados con un único nombre que define a dicho vector. De esta manera, si se declara en un programa un vector de 8 enteros con los primeros ocho números naturales, una representación de como se distribuye en la memoria lo muestra el siguiente gráfico:



Los números se almacenan en forma continua uno al lado del otro y el nombre del vector representa la dirección en donde comienzan a almacenarse los elementos. Los lenguajes determinan la cantidad de bytes que se reserva para cada elemento del vector en base a su declaración. En este caso se utilizo un tipo de datos entero. Suponiendo que para el lenguaje en particular, un entero tiene 4 bytes (32 bits), el vector ocuparía 32 bytes de memoria (256 bits). La forma de encontrar en memoria al vector es a través de la dirección de memoria en el que comienza. Si se quiere un elemento en particular, por ejemplo el tercero, basta con desplazarse al tercer lugar, es decir, moverse al **tercer entero** dentro del vector. Esto se hace de la siguiente manera:

1. Si se quiere el primer elemento, sumar 0 bytes a la dirección donde empieza el vector
2. Si se quiere el segundo elemento, sumar 4 bytes (un entero) a la dirección donde empieza el vector (pasarse el primer elemento para leer el segundo)
3. Si se quiere el tercer elemento, sumar 8 bytes (dos enteros) a la dirección donde empieza el vector (pasarse el segundo elemento para leer el tercero)

Un ejemplo de utilización es colocar doce valores enteros dentro de un vector que representa las veces que usó un auto una persona en cada uno de los 12 meses. Así cada elemento es un mes y el valor que almacena es la cantidad de veces que usó el auto. El nombre del vector podría ser, por ejemplo, usoDelAutoPorJuan y el vector sería como el que muestra la figura:



El uso de esta manera de un vector permite hacer una abstracción del uso de la información para que un programa interprete los datos de la siguiente manera:
Juan usó el auto según la siguiente tabla de representación:

Mes	Cantidad de veces
Enero	10
Febrero	6
Marzo	12
Abril	11
Mayo	3
Junio	22
Julio	6
Agosto	15
Septiembre	21
Octubre	11
Noviembre	30
Diciembre	33

Estructuras de datos

Una vez agrupados datos de un mismo tipo como se hizo en el vector, el siguiente desafío es agrupar datos de diferentes tipos. Muchos lenguajes de programación resolvieron esta cuestión incorporando estructuras de datos, las cuales siguen el mismo precepto que el vector, salvo que para cada elemento ahora debe definirse el tipo de datos que es. Cada lenguaje define estas estructuras según su gramática particular, pero en todos los casos, las estructuras de datos son modelos que definen un nuevo tipo de datos. Es decir, una estructura se define para declarar variables **del tipo** de la estructura.
Para no caer en un lenguaje en particular, se utiliza pseudocódigo para mostrar una declaración tipo, según el siguiente formato

ESTRUCTURA Nombre
    DECLARACIONES DE VARIABLES
FIN ESTRUCTURA
ESTRUCTURA Nombre *VARIABLE_ESTRUCTURA*

Notar que no se tiene una variable en memoria hasta que se la declara explícitamente, como se hace al final de la declaración y que al igual que en el vector, el nombre de la estructura representa el lugar de memoria en donde se almacena la misma, pero a diferencia de este, cada vez que se quiera acceder a un elemento de la estructura, se deberá desplazar la dirección tantos bytes como mida el elemento en particular

Ejemplo

```
ESTRUCTURA Persona
    CADENA nombre
    CADENA apellido
    ENTERO edad
```

FIN ESTRUCTURA
ESTRUCTURA Persona variablePersona
Notar que el uso de la abstracción en una estructura de datos es mayor que en el caso de un vector. En estas se agrupan datos más genéricos que en el caso del vector y tiene mayor flexibilidad para realizar asociaciones conceptuales entre el nombre de la variable del tipo estructura y su contenido.
Otro punto importante a tener en cuenta es que **se pueden declarar tantas variables del tipo de la estructura como se necesiten**. La declaración de una estructura define un nuevo tipo de datos (por ejemplo, como si fuera un entero o un punto flotante) y por definición se pueden declarar tantas variables como se deseen de un determinado tipo de datos.

Uso de variables definidas por una estructura de datos

Una vez definida la estructura de datos y declarada una o más variables de su tipo, es necesario acceder a las variables declaradas **dentro** de la estructura de datos. Una notación adoptada por muchos lenguajes de programación es la notación de punto. Del ejemplo anterior, se pueden definir una serie de asignaciones de la siguiente manera:

variablePersona.nombre = “Martín”
variablePersona.apellido = “García”
variablePersona.edad = 33

Se debe tener en cuenta que una variable de tipo estructura es básicamente **una variable**, con lo que tiene todas sus propiedades: se pueden asignar y recuperar valores, se pueden retornar desde procedimientos o funciones y pueden ser sus parámetros.
Se pueden extender algunos conceptos de la siguiente forma:

- Una estructura de datos se puede utilizar para realizar una abstracción de una entidad
- Las variables dentro de la estructura de datos son los atributos de dicha entidad

Así, en el ejemplo, la entidad es Persona, y sus atributos son nombre, apellido y edad.
Si la persona cumple años, se debe cambiar el atributo edad y almacenar, por ejemplo, 34 en lugar de 33. Este hecho demuestra el siguiente importante enunciado:

Los valores que almacena una determinada entidad, definen el estado de la misma

Por lo tanto, las variables declaradas dentro de una estructura son sus atributos y los valores que almacenan definen su **estado**.

Ampliando las estructuras de datos

Las entidades representadas mediante estructuras de datos son un gran paso en la dirección de abstraer elementos de un programa mediante la agrupación de los atributos que pertenecen a dicha entidad. Sin embargo, esto significa que lo único que se está teniendo en cuenta son los datos que se definen para una determinada entidad y no los procedimientos o funciones que operan con dichos datos.
Tomando un ejemplo sencillo, es válido suponer querer realizar una función o procedimiento que verifique que la edad que se asigna a una persona está en un rango entre 0 y 120 años. Con lo que se definió hasta el momento, dicho procedimiento o función debe ser una parte independiente del programa, pero opera con datos específicos de la estructura, es más, opera tan sólo con uno de ellos. Por lo tanto, una invocación incorrecta podría causar errores inesperados si los valores que se reciben como parámetros no son los propios de la estructura.

Otro punto importante es que estas funciones o procedimientos sirven como **servicios** que la entidad brinda para su utilización (por ejemplo, en este caso, el servicio es "no equivocarse al asignar una determinada edad"). Una conclusión importante derivada de esto es el siguiente enunciado:

> *Las entidades poseen atributos que definen su estado y servicios mediante los cuales se comunican con otras entidades*

Se puede concluir entonces que sería provechoso que dichos procedimientos o funciones estuvieran incluidos **dentro** de la estructura y no en una parte independiente del código que se utiliza.

Para declarar un elemento dentro de una estructura, es fundamental conocer el tipo de datos para saber cuántos bytes de memoria se deben reservar para almacenar la información, pero ¿cuántos bytes de memoria hay que reservar si se va a colocar dentro de una estructura una función o procedimiento? En realidad, no es necesario saber el tamaño de una función o procedimiento para colocarlo dentro de una estructura, basta con saber en que lugar de la memoria se encuentra el mismo y guardar una referencia a ese espacio. Además, las direcciones de memoria en un computador son siempre del mismo tamaño ya que este lo fija el bus de direcciones. Por lo tanto, un lenguaje "sabe" el tamaño de una dirección de memoria para una determinada plataforma de trabajo donde se intenta crear un programa. Esto permite definir tamaños fijos dentro de la estructura de datos para indicar cuales son las direcciones de los procedimientos o funciones que operan con los datos declarados dentro de la estructura.

¿Cómo trabaja un programa en memoria?

Todo programa en memoria sigue un formato conceptual de cuatro regiones, determinadas por los registros del procesador que gestionan los datos almacenados en ellas, donde se almacena información en la memoria RAM. Estas cuatro regiones son:

- **Heap (Amontonamiento)**: todo programa tiene asignado un espacio de memoria RAM para su ejecución. Este espacio lo determina el sistema operativo sobre el cuál se ejecuta el programa. Este a la vez gestiona dentro de este espacio las otras tres áreas que lo componen. Sin embargo, si el programa no utiliza todo el espacio que le asigna el sistema operativo queda un lugar sin usar al cual tiene acceso. Los lenguajes de programación permiten el acceso a dicho lugar de la RAM mediante un gestor dinámico de memoria, esto quiere decir, el programa durante su ejecución puede reservar espacios aleatorios de la memoria que le asignó el sistema operativo para usarla. Como la solicitud del espacio de memoria es aleatorio y depende de que sea solicitada durante la ejecución, la memoria se asigna en el primer espacio libre que se encuentra y los diferentes espacios asignados se van amontonando según se van solicitando (de ahí su nombre, Heap)
- **Stack (Pila)**: cada vez que se invoca una función o procedimiento, primero se reserva en la zona de la RAM asignada al Stack una porción de memoria para almacenar en ella los parámetros y variables locales que usará éste durante su ejecución. Cuando el procedimiento o función termina, este espacio de memoria reservado previamente se libera para que otro del mismo tipo pueda reclamar dicho espacio. Como un método puede invocar a otro, y este a su vez a otro y así sucesivamente, los espacios de memoria

se conceden apilando los requerimientos y liberándolos según van terminando (de ahí su nombre).

- **Memoria Estática**: cuando un programa utiliza datos que deben conservarse a lo largo de toda la ejecución del mismo, se deben guardar en un área de memoria que permanezca inalterable. Cómo esto determina que ese espacio se mantenga igual a lo largo de toda la ejecución, no puede variar dinámicamente como en el caso del Stack, por eso este lugar se maneja en un bloque separado de RAM
- **Zona de Texto**: es el lugar de memoria donde se encuentran las instrucciones de un programa. Dentro de la zona de texto están almacenados las funciones o procedimientos que pertenecen a un programa y cada una esta diferenciada por un punto de entrada. Dicho punto de entrada es un desplazamiento (offset) calculado desde el comienzo del área de la RAM asignada a la Zona de Texto

El siguiente diagrama de bloque muestra como queda conformada **lógicamente** la memoria RAM para la ejecución de un programa:

Creación de Instancias

Cada vez que se declara una variable del tipo de una estructura se crea una instancia y cada instancia puede tener su propio estado porque cada nueva variable se almacena en un lugar diferente de la memoria. Entonces, cabe la siguiente pregunta ¿Cómo encuentra cada instancia diferente los atributos que definen su estado?

Es obvio que de alguna manera hay que tener una referencia al lugar de memoria que se debe utilizar. Estas referencias se resuelven mediante el uso del nombre de la variable y la notación de punto cuando se acceden a los atributos, pero ¿cómo sabe un servicio cuáles son los atributos sobre los cuales operar? La solución lógica es pasarle cada vez que se invoca al servicio la dirección de memoria en donde se encuentran los atributos.
Como esta es una tarea repetitiva y promueve errores de programación si los programadores debieran hacerlo cada vez que invocan a un servicio, los lenguajes de programación resuelven el tema "tras bambalinas" y pasan por defecto esta referencia en cada invocación.
Para los lenguajes orientados a objetos las estructuras se denominan **clases** y las variables del tipo de la estructura se denominan **objetos**. Por lo tanto un objeto se obtiene como consecuencia de crear una instancia de una clase y tiene **todas las propiedades de las variables**.
Los atributos de un objeto se denominan **variables de instancia** y las funciones o procedimientos definidos en la clase se denominan **servicios, métodos o funciones de instancia**. Las variables locales o parámetros definidos dentro de los servicios no pertenecen a la estructura o clase, por lo tanto **no son variables de instancia**.

Definiciones

Para empezar a separar conceptos y clarificarlos, este es el punto donde las definiciones empiezan a ser necesarias:
Clase: es una construcción que se utiliza como un modelo (o plantilla) para crear objetos de ese tipo. Define un nuevo tipo de datos creado por el usuario (programador) que es justamente el de la clase. A este modelo se lo denomina también **abstracción del objeto que representa**
Miembros de una Clase: son los atributos y servicios declarados dentro de la clase
Objeto: es la unidad que en tiempo de ejecución realiza las tareas de un programa y se define mediante la creación de una instancia de una determinada clase
Variables de Instancia o Atributos: son las variables declaradas dentro de una clase. No se almacena espacio de memoria para ellas hasta que no se cree una instancia de la clase (de ahí su nombre)
Servicios: son los procedimientos o funciones declarados dentro de una clase. Muchos lenguajes los denominan como método o función miembro
Variables locales: son las variables declaradas como parámetros de los servicios o dentro de ellos. Sólo se las puede acceder dentro del método que las declara, por lo cual no son variables de instancia
Módulo: es una unidad de programación. Muchos lenguajes de programación establecen que un módulo es una clase. Sin embargo, como los módulos se comunican entre mediante el acceso a sus atributos o servicios, este concepto se suele extender en otros lenguajes a un conjunto de clases que interactúan con otros módulos (como es el caso de las librerías de enlace dinámico – dll, dynamic link library – utilizadas por los sistemas operativos).

Visibilidad en las Clases

Una clase es algo más que una simple estructura. En los ejemplos expuestos anteriormente se trataron los servicios como agentes internos que cumplen una función específica, como validar un ingreso de información. Sin embargo es claro que se necesita definir de forma

estricta que se puede acceder desde adentro de la misma clase o desde fuera de ella. En otras palabras, definir cuáles servicios son internos (como por ejemplo validar) y cuales son externos (como conocer el estado de un objeto por medio de sus atributos).
Si se accede a los atributos de un objeto sin ningún control, se permite que cualquier código que accede a un objeto modifique su valor y esto puede derivar en errores. Si se utiliza como ejemplo una clase del tipo Persona igual a la estructura de datos previamente definida, se puede dar el caso que a la variable edad se le asigne un valor que no lo controla el servicio que valida que las edades estén dentro del rango de 0 a 120, con lo cual el atributo almacena un valor que es inconsistente con el diseño de la clase, causando una anomalía en tiempo de ejecución.
Se hace evidente entonces que deben existir controles de acceso a los elementos que componen una clase. En los lenguajes de programación esto se define mediante una técnica llamada **visibilidad**. La visibilidad define el **alcance** de una variable, es decir, los lugares dentro del código en los cuales es válido su acceso. Cuando una variable sale del alcance es porque pierde su visibilidad y no puede ser invocada.

Principio del iceberg: ocultamiento de la información

Para poder dominar, como una clase se comunica con otra, debe intervenir la parte de diseño de la misma ya que la comunicación se establecerá por el desarrollo de algún medio de acceso que ésta declare.
Una primera definición para realizar es la diferencia entre los miembros que son accesibles desde fuera de la clase cuando se crea una instancia de ella (objeto) y los que no:

> *Las clases definen miembros **públicos** y **privados**. Los primeros son accesibles desde fuera de la clase por cualquier código que lo requiera. Los segundos sólo se pueden acceder desde dentro de la clase, por lo tanto tiene su visibilidad circunscripta a la clase*

Muchas veces se denomina a dicho medio **interfaz de la clase**, y es la definición que establece como se acceden sus elementos. Si se piensa en una interfaz, bien se podría decir que la clase consta de ciertos elementos internos que se comunicarán a través de esta con otras clases, actuando como un canal que controla dicha comunicación. Lo que se quiere señalar con esto es que, los elementos privados de un módulo no deben ser accesibles desde afuera de este por ningún otro, salvo por medio de la interfaz definida para hacerlo, que es la parte pública del mismo.
Resumiendo, el principio se puede enunciar como:

> *En cada módulo debe existir una división entre sus componentes en secciones que sean públicas y privadas de manera que toda información acerca del mismo sea privada a éste, a no ser que se especifique como pública explícitamente.*

Si se alteran los elementos privados de un módulo, pero su interfaz se mantiene constante, la interacción con otros módulos no variará ya que estos acceden mediante ellas, evitando que los cambios se propaguen a lo largo del código sin control.

> *Para asegurarse de cumplir el principio de ocultamiento de la información, se deben declarar privados todos los atributos de una clase para que los mismos sean invariables. Sólo pueden ser públicos aquellos atributos que sean CONSTANTES*

Para clarificar el principio, se puede pensar en una clase que posee un vector interno que almacena números enteros y un servicio que permite realizar una búsqueda sobre el vector para acceder a cada uno de sus elementos.

Como interfaz se puede brindar un servicio que busque un determinado elemento y retorne el índice del vector (la posición) del elemento. Si se cambia el elemento del vector, mientras se lleva a cabo la búsqueda, del valor almacenado que es justamente el que se deseaba hallar, el servicio retornará que encontró un valor que ya no esta almacenado en el vector porque otro código lo accedió y lo cambió sin que se pueda realizar ninguna acción de control previa. Esto es, se cambia un valor almacenado sin que un servicio propio del objeto registre el acceso a la variable de instancia (el elemento del vector) y pueda realizar alguna acción en consecuencia. Si por el contrario, el acceso a la variable de instancia se realiza mediante un servicio, este puede verificar que otro servicio de la clase no esté accediendo al elemento y la búsqueda no retornará un valor erróneo o viceversa, que no se realice una búsqueda mientras este en proceso un cambio de valor.
Se suele llamar a este principio "del iceberg" puesto que la interfaz de la clase, que es lo que se ve desde afuera del módulo, es la punta del iceberg, mientras que los elementos privados, que son por lo general el grueso de la implementación, se encuentran ocultos al mundo exterior del módulo.
La importancia de este principio radica en el control que permite del flujo de datos hacia y desde el módulo, además de los servicios brindados por él. Respecto de los servicios que brinda, al estar oculta la forma en que se los brinda, la comunicación con el módulo estará garantizada a través de su parte pública ocultando detalles y uso no deseado de los componentes del módulo que se encuentren en la parte privada

Encapsulado

Siguiendo el ejemplo enunciado, ¿qué pasa si el servicio que realiza la verificación del rango de valores del atributo edad de la clase Persona es una interfaz de la clase? Como la verificación se hace sobre los atributos dentro de un objeto del tipo Persona, puede dejar a los mismo en una condición inconsistente, es decir, se puede cometer el error de invocar la verificación usándola como un servicio diferente a como se pensó originalmente.
Supongamos que la verificación recibe como parámetro el índice del elemento que debe verificar y cierto código ajeno a la clase confunde ese parámetro con el valor que se debe verificar en la validación. El código tratará de acceder a un elemento que probablemente no exista causando un error.
Para evitar este tipo de situaciones, se debe diseñar un servicio interno de la clase que sea utilizado sólo por los miembros de esta y que acceda a los atributos privados de la misma. Esto permite mantener el control de la prestación de servicios de manera que sólo se utilicen para los fines que fueron creados.
Como se mencionó anteriormente para los atributos, existe una técnica utilizada por los lenguajes de programación que limita el acceso y alcance de las variables llamado visibilidad. Esta se puede aplicar de manera análoga a los servicios para que se accedan sólo desde el interior de la clase. Por lo tanto, el servicio interno sólo se invocará desde otros elementos de la clase y su comportamiento estará restringido a la visibilidad de la clase en la que está definido.

> *Cuando una clase tiene definido un servicio interno, esto es, una función o procedimiento con visibilidad* ***privada****, se define un comportamiento* ***encapsulado*** *dentro del objeto en tiempo de ejecución.* ***Para que el comportamiento del servicio sea correcto, los atributos deben estar ocultos de manera que un cambio en los mismos no provoque un error al ejecutarlo***

Descubriendo los objetos

Todos los enunciados que se vienen haciendo hasta el momento encontraron su motivación en las limitaciones que se encontraban en la programación tradicional o estructurada.
En los lenguajes tradicionales, la programación se basa en funciones y procedimientos, prevaleciendo éstos sobre los datos que manejan.
Si prevalecen las funciones, la solución será hacer tan importantes a los datos como a las operaciones que se realizan con ellos, agruparlos dentro de un módulo y que las operaciones sean los servicios antes explicados.
A partir de esta afirmación, las cosas se complican, porque normalmente se pensaba en los datos sólo como elementos a transformar por diferentes procesos.
Además, no se pueden pensar a los datos como entidades aisladas, puesto que los mismos transformados por ciertos procesos pueden generar nuevos datos de los que se alimente el sistema.
La respuesta al dilema se encuentra en poder unir bajo una misma estructura tanto a los datos como a las operaciones que manejan a los mismos. Cuando las transformaciones sufridas a través de procesos generan nuevos datos, pero estos están representados como atributos dentro de una entidad, sus modificaciones provocan sólo cambios de estado dentro de los objetos a los que pertenecen.
Por lo tanto, la clave del diseño de la programación orientada a objetos radica en detectar los objetos que sirven para la solución del problema en particular que se aborda.
Todo objeto esta representado de forma natural en el lenguaje por medio de los **sustantivos**.
Por ejemplo, tomando la frase "Juan come una manzana" se pueden detectar los objetos separando los sustantivos y analizando la frase. De esta manera los sustantivos son:

- Juan
- Manzana

La frase tiene una sola acción a analizar:

- Come

Esta información permite un primer análisis. Existen dos objetos, Juan y manzana, y una operación, comer. Como se puede apreciar en la frase, los objetos se detectan en tiempo de ejecución (Juan come - **ahora** - una manzana). A dichos objetos se los denomina **candidatos** porque se debe corroborar que se puede generar una abstracción **única** a partir de ellos. Por ejemplo, si dos objetos candidatos son **sinónimos**, ambos tendrán como abstracción a la misma clase.
Una vez detectados, se debe crear una abstracción de ellos (o sea, una clase). Se puede realizar la siguiente abstracción para representar al objeto:

Juan es una instancia de la clase Persona

Al realizar semejante afirmación se está indicando que si se genera una clase que se llame Persona, los atributos y servicios definidos dentro de ella pueden representar perfectamente a un objeto como Juan cuando se cree una instancia de la misma. Análogamente, se puede crear una abstracción de manzana por medio de la clase Fruta.
Sin embargo, se puede extraer aún más información de la frase analizada. Queda por ver que pasa con la acción que se define en ella. Las acciones en el mundo de la programación orientada a objetos se representan como servicios que los diferentes objetos ofrecen en

tiempo de ejecución. Por lo tanto se debe definir ***que objeto tiene la responsabilidad de ofrecer el servicio*** comer.
Este último razonamiento nos permite crear la siguiente definición:

> **Responsabilidad de un objeto**: es el conjunto de servicios que debe prestar un objeto

Por lo tanto, la determinación de responsabilidades es un punto crucial en el diseño de clases como abstracciones de objetos a crear.
Esto puede inducir a errores de razonamiento. Si no se es cuidadoso y se plantea bien a que objeto pertenece la responsabilidad del servicio en dichos términos, se puede caer en la tentación de deducir que el mismo pertenece a Juan "porque es Juan el que come". Esto se un error. Juan *puede comer la manzana porque* ***la manzana tiene el servicio que permite a Juan comerla***. Por lo tanto, la responsabilidad del servicio recae sobre el objeto manzana.
Cuando se genera una abstracción de la misma por medio de la clase Fruta, se puede asegurar que Fruta posee un servicio que es "dejarseComer" y que el mismo pertenece a la interfaz de la clase.
Sin embargo, Juan también posee un servicio, que puede ser interno o externo, que recibe un parámetro del tipo fruta e invoca a través de su método público "dejarseComer" el servicio de la instancia en particular, en este caso, manzana.
Resumiendo, de una simple frase se pudo analizar la siguiente información para diseñar las abstracciones:

- Existen dos abstracciones, las clases Persona y Fruta
- La clase Fruta tiene un servicio público que es dejarseComer
- La clase Persona tiene un servicio público o privado que es comer

Manteniendo esto último presente se puede definir:

> ***Los objetos pueden invocar servicios de otros objetos para llevar a cabo aquellos servicios que son sus responsabilidades***

Todavía se puede sacar algunas conclusiones más para realizar un buen modelo por medio de la clase. Cuando se obtiene la abstracción, se puede inferir cuales son los atributos que la misma posee. En el caso de persona se puede determinar rápidamente sin miedo a cometer un error, que se debe incluir entre sus atributos nombre y apellido. En particular, el objeto analizado, Juan, guarda en el atributo nombre este valor y define así su **estado**. No confundir el nombre de la instancia del objeto con el valor del atributo. En este caso en particular, el nombre de la instancia coincide con el valor que asume el atributo nombre en tiempo de ejecución.

Abstracciones principales

En el ejemplo anterior rápidamente se encontró en la frase analizada dos objetos candidatos. Cuando se detectan objetos rápidamente en un enunciado es porque los mismos son los más importantes, ya que pueden haber muchos otros objetos que se determinen cuando se profundice el análisis, ya sea por medio de obtener más información o porque descubrimos que ciertos atributos de los objetos deben tratarse como objetos en si mismo.
En el ejemplo se determinó que las abstracciones eran Persona y Fruta y como fueron las primeras en detectarse se las denomina **abstracciones principales**. En el caso de la primera se puede determinar la necesidad de un atributo que se llame dirección. Sin embargo un análisis más minucioso revela que el atributo puede tener servicios en si mismo, por

ejemplo uno que permita realizar cambios de dirección. Cuando un atributo se detecta como un posible objeto se lo denomina **objeto derivado** y a su modelado como **abstracción derivada**

Información relevada

La información relevada es aquella a partir de la cual se comienza a analizar cuáles son las abstracciones principales. Por lo general se encuentra en forma de una serie de párrafos que se obtienen al tratar de entender el problema a resolver y que se obtiene de un cliente, interesado o definición que indican en forma narrativa sus necesidades de solución. Por ejemplo, cuando un profesor dicta un problema a los alumnos en una clase, el enunciado pasa a ser la información relevada.
Por lo general la información de este tipo no sigue el orden que suele tener el enunciado de un problema que dicta un profesor sino que viene en un formato no secuencial, desordenado y confuso. También parte de ella se puede descartar si no es relevante a la solución que se desea obtener
A la simplificación de la información relevada se la denomina extracción de requisitos y ordena en forma sencilla las necesidades de solución planteadas en la información relevada. Cuando el conjunto de requerimientos se ordena en un enunciado claro y conciso, sin redundancias ni perdida de información de necesidades a resolver, se lo llama **enunciado de un problema**, al igual que el dictado por un profesor a sus alumnos.

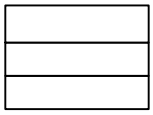
Contexto de un problema

Cada vez que se realiza un análisis de un problema es importante determinar cuales son los límites de la solución. Es muy fácil cuando se crean modelos concluir que otros objetos se pueden modelar a partir de las abstracciones principales. De alguna manera hay que establecer donde se debe parar y hay que tener presente en todo momento dichas limitaciones, mientras se analiza o se diseña una clase.
Muchas veces estas limitaciones se establecen como una lista de lo que la solución hace y lo que no. Al conjunto de elementos que pertenece a dicha lista se lo denomina contexto de un problema.
En el ejemplo anterior, se modela como una Persona come una Fruta. En la abstracción creada no se tiene en cuenta todas las cosas que puede comer una persona o de cuantas formas diferentes se puede "dejar comer" una fruta. Estas limitaciones pueden no ser muy claras durante el análisis pero si no se pierde de vista el contexto que se define para la solución del problema, se sabe exactamente donde para el análisis para no modelar más de lo que el problema requiere para su solución.
El contexto de un problema no es estático y puede variar a medida que avanza el análisis. Mientras se determinan que elementos son parte de la solución o no, se va extendiendo la lista que pertenece al contexto como una limitación del alcance de la solución.
Como el alcance de la solución lo determina el contexto de la misma, muchas veces se denomina al contexto como **dominio de la solución**. Sin embargo, vale la pena aclarar que cuando el problema no se subdivide en varios problemas de menor tamaño, el contexto coincide con el dominio de la solución. Caso contrario, el dominio de la solución es la suma de los contextos de las diferentes soluciones que se realizaron para resolver el problema.
En el análisis y diseño orientado a objetos se suele llamar a un problema a resolver **problema de negocio**, ya que por lo general la información relevada era para solucionar un

problema de ese estilo. Hoy en día se utilizan estas palabras para englobar el enunciado de cualquier tipo de problema, más allá si es de un negocio en particular, científico, etc.

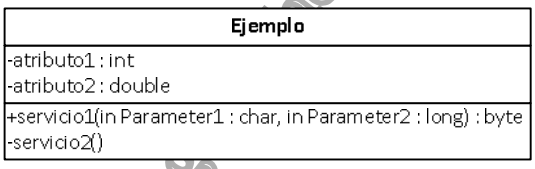
Diagramas UML para las clases

Las clases tienen una representación gráfica en un lenguaje universal de modelado llamado UML (Universal Modelling Language).
Las clases se representan en un rectángulo con tres divisiones en su interior, como muestra la siguiente figura



- En la primera división se coloca el nombre de la clase
- En la segunda división se coloca los atributos de la clase
- En la tercera división se colocan los servicios de la clase

Dejando de lado la división que lleva el nombre de la clase, las otras dos son opcionales y pueden no estar presentes en el diagrama, ya sea una o ambas. Sin embargo, por una cuestión de claridad visual, se recomienda usar todas las divisiones aunque alguna de ellas esté vacía.
Los diagramas de clases se utilizan tanto para el análisis como para el diseño, por lo cual es normal que dependiendo de la etapa del análisis en la que se encuentre, una de las divisiones opcionales o ambas estén vacías. Como ejemplo, se puede observar el siguiente gráfico:



En el diagrama se modela una clase que se llama Ejemplo, que posee dos atributos, y dos servicios. A continuación se explica la forma de leer las declaraciones que tienen los miembros.
Para los atributos

- **Visibilidad**: en este caso pueden ser públicas, privadas o protegidas (las visibilidades más comunes en los lenguajes de programación aunque no las únicas). Si un atributo es público se representa con el símbolo **+**, si es privado con **–** y si es protegido con **#**. En el ejemplo tanto atributo1 como atributo2 tienen visibilidad privada
- **Nombre**: el nombre que se le asigna a la variable de instancia. En este caso, las variables de instancias se llaman atributo1 y atributo2
- **Tipo**: es el tipo de variable definida. Para el caso del ejemplo, atributo1 es un **int** y atributo2 es un **double**. La declaración del tipo de variable va siempre a continuación del símbolo "**:".**

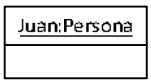
Para los servicios

- **Visibilidad**: en este caso pueden ser públicos, privados o protegidos (las visibilidades más comunes en los lenguajes de programación aunque no las únicas). Si un servicio es público

se representa con el símbolo **+**, si es privado con **–** y si es protegido con **#**. La declaración de visibilidad es igual que para el caso de los atributos

- **Lista de parámetros**: se declaran entre paréntesis separados por comas con el mismo formato de los atributos pero sin incluir la visibilidad. Según el lenguaje a utilizar, pueden agregarse declaraciones propias de estos, como la palabra "**in**" que se puede apreciar en el gráfico, aunque este es opcional.
- **Nombre**: el nombre que se le asigna a la variable de instancia. En este caso, las variables de instancias se llaman atributo1 y atributo2
- **Tipo del valor retornado**: es el tipo de datos que retorna el método (en caso que retorne un valor). Para el ejemplo, servicio1 retorna un **byte** y tiene dos parámetros. En el caso de servicio2 no retorna valor ni recibe parámetros. La declaración del tipo de valor retornado va siempre a continuación del símbolo "**:".**

También se puede modelar gráficamente un objeto o instancia de una clase. Utilizando el ejemplo de la clase Persona y la instancia Juan, se conoce para dicho caso tanto el nombre de la clase como el del objeto y el diagrama es como el que se muestra a continuación



Pero si se debe modelar una instancia de una clase que todavía no tiene asignado un nombre, se lo puede hacer como muestra la siguiente figura