ALGORITMOS Y ESTRUCTURAS DE DATOS
SEMANA Nº 3

OBJETIVOS DE LA TERCER SEMANA

Clase teórica
- Clases en java (Estructura general, declaración y definición, cuerpo de una clase. Instancias de una clase (objetos) (acceso a los miembros, la vida de un objeto), declaración de miembros de clases (modificadores de acceso a miembros de clases), atributos y métodos.

Clase práctica
- Ejercicios de clases con código y diagramas de flujo.

CLASE TEORICA

CLASES EN JAVA

INTRODUCCION A CLASES

Como ya dijimos, una clase es un modelo que se utiliza para describir a objetos similares.

Las clases son lo más simple de Java. Todo en Java forma parte de una clase, es una clase o describe como funciona una clase. El conocimiento de las clases es fundamental para poder entender los programas Java.

Todas las acciones de los programas Java se colocan dentro del bloque de una clase o un objeto. Todos los métodos se definen dentro del bloque de la clase, Java no soporta funciones o variables globales. Así pues, el esqueleto de cualquier aplicación Java se basa en la definición de una clase.

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

Las clases soportan el conceptos de encapsulamiento de datos, que se produce cuando la representación interna de un objeto junto con sus operaciones, se encierran en la misma estructura (la clase).

Una clase es un marco que permite crear objetos de su misma estructura. Estos objetos constan de:

Variables de clase y de instancia, que son los descriptores de atributos y entidades de los objetos.

Métodos, que definen las operaciones que pueden realizar esos objetos.

Un objeto se instancia a partir de la descripción de una clase: un objeto es dinámico, y puede tener una infinidad de objetos descritos por una misma clase. Un objeto es un conjunto de datos presente en memoria.

Así, si define una clase Ventana, ésta podrá instanciarse tantas veces como lo desee. Pero todos los objetos creados tendrán la misma descripción y el mismo comportamiento.

¿Qué significa esto? La descripción de un objeto viene dada por sus atributos; en el caso de una Ventana, los atributos serán, por ejemplo, su título, sus coordenadas, su altura y su anchura:

```
class Ventana {
        String titulo;
        int coordenadax;
        int coordenadaY;
        int altura;
        int anchura;
}
```

Cada Ventana así creada tendrá sus propios atributos, pero todas estas Ventana compartirán esta descripción. Observemos que los atributos de un objeto pueden ser tipos simples (enteros, booleanos, etc.) o bien otros objetos como aquí la cadena de caracteres que contiene el nombre de la ventana.

En Java, los objetos se crean de la siguiente forma:

```
Ventana ven;
ven = new Ventana();
```

DESCUBRIMIENTO DE LAS CLASES

Una de las primeras decisiones que se debe tomar al crear una aplicación orientada a objetos es la selección de clases. Las clases en la POO pueden tener tipos diferentes de responsabilidades, en UML (lenguaje unificado de modelo) existen tres estereotipos y se utilizan para ayudar a los desarrolladores a distinguir el ámbito de las diferentes clases. Ellas son:

Clases **de entidad**. Las clases de entidad se utilizan para modelar información que posee una vida larga y que es a menudo persistente. Son clases cuya responsabilidad principal es mantener información de datos o de estado. Con frecuencia se reconocen como los sustantivos en la descripción de un problema y generalmente son los bloques de construcción fundamentales de un diseño. Estas clases modelan la información y el comportamiento asociado a algún concepto, como una persona, un objeto del mudo real.

Clases **de interfaz**. Las clases de interfaz se utilizan para modelar la interacción entre el sistema y sus actores (es decir usuarios y sistemas externos). Esta interacción a menudo implica recibir (y presentar) información y peticiones de (y hacia) los usuarios y los sistemas externos. Por ejemplo, una parte esencial de la mayoría de las aplicaciones es la presentación de la información en un dispositivo de salida, como la pantalla de un terminal. Debido a que el código para llevar a cabo dicha actividad a menudo es complejo, modificado con frecuencia y muy independiente de los datos reales que se están mostrando, es una buena práctica de programación aislar el comportamiento de presentación en clases separadas de aquellas que guardan los datos mostrados. Por ejemplo, en una abstracción de un juego de cartas crearemos una clase Carta que contendrá los datos la misma y una clase VistaDeCatrtas, para encargarse de dibujar la imagen de una carta en la pantalla. Con frecuencia, los datos de base (por ejemplo la clase Carta, que es una clase entidad) son llamados el *modelo*, mientras que la clase que se exhibe (VistaDeCatras que es una clase de interfaz) es llamada la *vista*.

Dado que separamos el modelo de la vista, por lo general se simplifica mucho el diseño del modelo. Idealmente, el modelo no debe requerir ni contener ninguna información acerca de la vista. Ello facilita la reutilización del código, ya que el modelo puede usarse en aplicaciones diferentes. No es raro para un modelo único tener más de una vista. Por ejemplo la información financiera podría presentarse tanto en gráficas de barras o gráficas de pastel, como en tablas o figuras, sin cambiar el modelo subyacente.

En ocasiones es inevitable la interacción entre un modelo y una vista. Por ejemplo, si a las cifras de la tabla financiera que se acaba de describir se les permite cambiar dinámicamente, el programador puede desear que la presentación se actualice en forma instantánea. Entonces es necesario que el modelo avise a la presentación que ha sido cambiado y que ésta se debe actualizar. Algunos programadores se refieren a tal modelo como a un *sujeto* para distinguirlo del modelo que no tiene conocimiento del uso.

Clases **de control**. Las clases de control representan coordinación, secuencia, transacciones, y control de otros objetos. Los aspectos dinámicos del sistema se modelan con clases de control, debido a que ellas manejan y coordinan las acciones y los flujos de control principales, y delegan trabajo a otros objetos (es decir, objetos de interfaz y de entidad).

Si parece que una clase abarca dos o más de estas categorías, con frecuencia podrá dividirse en dos o más clases. Por ejemplo, el primer diseño de la abstracción del juego de cartas tiene una sola clase, llamada carta. Luego esta se fracciona en la clase de datos y en la clase de vista.

EJEMPLO: UN JUEGO DE CARTAS

Usaremos una abstracción de software con una carta de un juego de naipes común. La Carta sabe poco de la intención de su uso y puede incorporarse en cualquier tipo de juegos de cartas. El comportamiento de una carta sería el siguiente:

Mantener palo y rango
Devolver color

Las responsabilidades de la clase Carta son muy limitadas; básicamente, una carta es tan sólo un manejador de datos que mantiene y devuelve valores de rango y de palo. En particular la clase básica Carta no tiene la capacidad para mostrarse a sí misma.

Se crea una clase de vista llamada VistaDeCarta para manejar las tareas de relacionadas con su visualización, el comportamiento de esta clase sería la siguiente:

Dibujar carta en la superficie de juego
Borrar imagen de carta
Mantener estado de bocaarriba o bocaabajo
Mantener ubicación en la superficie de juego
Moverse a la nueva ubicación en la superficie de juego

Si separamos la visualización de la carta de los datos de la carta misma, aislamos el dispositivo principal y las dependencias ambientales de las estructuras menos dependientes. Por ejemplo, la capacidad para dibujar la carta dependerá mucho de la interfaz que elijamos, si cambiamos a un sistema diferente, sólo se necesitará cambiar la clase de vista.

No siempre queda clara la decisión en cuanto a qué clase pertenece un comportamiento. Tras haber separado las tareas manejadoras de datos de las de visualización, resulta fácil decidir que el método color pertenece ala clase Carta y que el método mostrar debe ir en la clase VistaDeCarta, pero, ¿qué pasa con los métodos volver (voltear) y bocaarriba? ¿El estado bocaarriba o bocaabajo de una cata es una propiedad intrínseca de la carta misma o meramente una propiedad que dice cómo se muestra la carta? Se podría defender el punto anterior en ambos sentidos, pero hemos decidido colocar este comportamiento en la clase VistaDeCatra. (Un punto que hay que considerar es si existen usos para la clase Carta que no requieran conocer el estado de bocaarriba. Si es así, entonces no debe asignarse a la abstracción de Carta el mantenimiento de tal información).

En el siguiente paso es refinar nuestras clases, definiendo atributo y métodos, como se muestra a continuación:

Carta

atributos

palo	*- (entero) valor del palo*
rango	*- (entero) valor del rango*

métodos

FijarPaloYRango	*- Fijar palo y rango de carta*
Palo	*- devolver palo de carta*
Rango	*- devolver rango de carta*
Color	*- devolver color de carta*

VistaDeCarta

atributos

laCarta	*– (tipo Carta) el valor de la carta (es un objeto de tipo Carta)*
bocaArriba	*– (booleano) estado de bocaarriba o bocaabajo*
ubicx, ubicy	*– ubicación en superficie de juego*

métodos

Borrar, Dibujar	*– borrar o dibujar imagen de carta*
BocaArriba, Volver	*– probar o voltear carta*
Incluye (entero, entero)	*– probar si el punto está dentro del límite*
MoverA(entero, entero)	*– mover carta a nueva ubicación*

El siguiente paso es traducir a código ejecutable el comportamiento y el estado descritos de las clases. Este código puede ser escrito en cualquier lenguaje de programación que soporte POO y que en nuestro caso usaremos el lenguaje Java.

ESTRUCTURA GENERAL

Una clase contiene elementos, llamados miembros, que pueden ser datos, llamados *atributos,* y funciones que manipulan esos datos llamados *métodos*.

Una clase se define con la palabra reservada class.
La sintaxis de una clase es:

```
[public] [final | abstract] class nombre_de_la_Clase [extends ClaseMadre]
      [implements Interfase1 [, Interfase2 ]...]
{
      [Lista_de_atributos]
      [lista_de_métodos]
}
```

Todo lo que está entre [y] es opcional. Como se ve, lo único obligatorio es class y el nombre de la clase.

public, final, abstract

Definir una clase como pública (**public**) significa que puede ser usada por cualquier clase en cualquier paquete. Si no lo es, solamente puede ser utilizada por clases del mismo paquete (un paquete,

básicamente, es un grupo de clases e interfaces relacionadas, como los paquetes de biblioteca incluidos con Java).

Una clase final (**final**) es aquella que no puede tener clases que la hereden. Esto se utiliza básicamente por razones de seguridad (para que una clase no pueda ser reemplazada por otra que la herede), o por diseño de la aplicación.

Una clase abstracta (**abstract**) es una clase que puede tener derivadas, pero no puede ser instanciada. Es literalmente abstracta. ¿Para qué sirve? Para modelar conceptos. Por ejemplo, la clase Number es una clase abstracta que representa cualquier tipo de números (y sus métodos no están implementados: son abstractos); las clases descendientes de ésta, como Integer o Float, sí implementan los métodos de la madre Number, y se pueden instanciar.

Por todo lo dicho, una clase no puede ser final y abstract a la vez (ya que la clase abstract requiere descendientes).

extends

La instrucción extends indica de qué clase desciende la nuestra. Si se omite, Java asume que desciende de la superclase **object**.

Cuando una clase desciende de otra, significa que hereda sus atributos y sus métodos. Esto quiere decir que, a menos que los redefinamos, sus métodos serán los mismos que los de la clase madre y podrán utilizarse en forma transparente, siempre y cuando no sean *privados* en la clase madre, o protegidos o propios del paquete para subclases de otros paquetes.

DECLARACION Y DEFINICION

La *declaración* lista los miembros de la clase. La definición, también llamada *implementación*, define las funciones de la clase.

La declaración y la definición de la clase van juntas. Por ejemplo:

```
//Implementación de un contador sencillo
//GRABAR EN UN ARCHIVO "Contador.java"
// COMPILAR CON: "javac Contador.Java
public class Contador {    //     Se declara y define la clase Contador
       int cnt;
              public void Inicializa() {
              cnt=0;        //inicializa en 0 la variable cnt
       }
       //Otros métodos
       public int incCuenta() {
              cnt++;
              return cnt;
       }
       public int getCuenta() {
              return cnt;
       }
}
```

EL CUERPO DE LA CLASE

El cuerpo de la clase, encerrado entre { y }, es la lista de atributos (variables) y métodos (funciones) que la constituyen. No es obligatorio, pero en general se listan primero los atributos y luego los métodos.

OBJETOS - INSTANCIAS DE UNA CLASE

Los objetos de una clase son instancias de la misma. Se crean en tiempo de ejecución con la estructura definida en la clase.

Para crear un objeto de una clase se usa la palabra reservada new.

Por ejemplo si tenemos la siguiente clase:

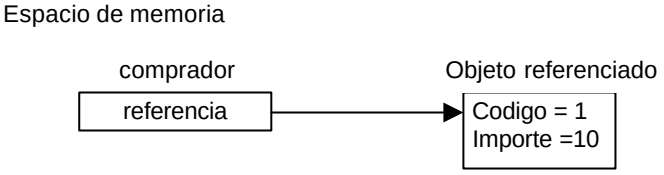
```
       public class Cliente {
              private int codigo;
              private float importe;
              public int   getCodigo()   { return codigo; }
              public float getImporte()  { return importe; }
              public void setImporte(float x) { importe = x; }
```

```
};
```

el objeto o instancia de la clase cliente es:

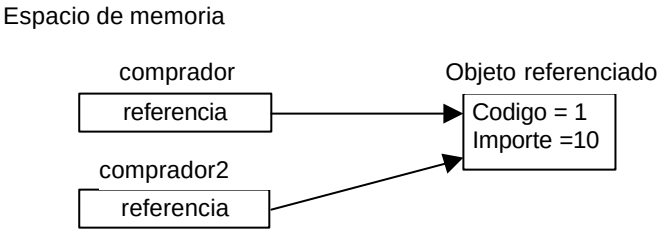
```
Cliente comprador = new Cliente();      //objeto o instancia de la clase
                                        //Cliente
```

El operador new reserva espacio en memoria para los miembros dato y devuelve una referencia que se guarda en la variable comprador de tipo Cliente que denominamos ahora objeto. Dicha sentencia, crea un objeto denominado comprador de la clase Cliente. Gráficamente puede imaginarse una referencia y el objeto referenciado, ubicados en algún lugar del espacio de memoria correspondiente a su aplicación:



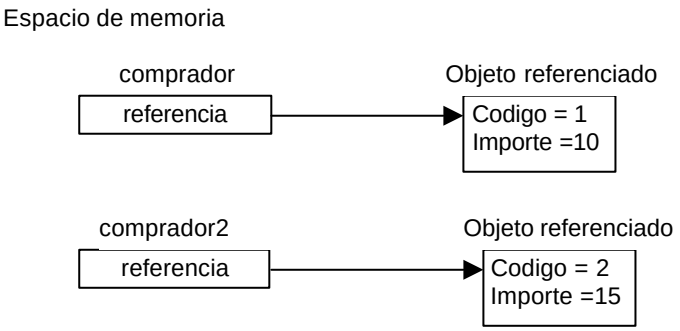
Si usted declara otra variable, por ejemplo, comprador2 y le asigna el objeto comprador, estará creando otra referencia al mismo objeto:

```
Cliente comprador2;
comprador2 = comprador;
```



Si quisiera crear otro objeto deberá usar la sentencia **new** que es la que reserva los espacios en memoria, produciendo la creación del objeto.

```
Cliente comprador2 = new Cliente();
```



Implementando el ejemplo en forma completa:

```
import java.io.*;

Public class ManejaCliente {
       //el punto de entrada del programa
       public static void main(String args[]) {
              Cliente comprador = new Cliente();      //crea un cliente

              comprador.setImporte(100);       //asigna el importe 100

              float adeuda = comprador.getImporte();
              System.out.println("El importe adeudado es "+adeuda);
              Cliente comprador2;
```

```
                comprador2 = comprador     //crea una nueva referencia al mismo cliente
                comprador2 = new Cliente();       //crea un nuevo cliente
        }
}
```

Acceso a los miembros

Desde un objeto se puede acceder a los miembros mediante la siguiente sintaxis

```
objeto.miembro;
```

Por ejemplo, podemos acceder al método setImporte, para cambiar el importe de la deuda de un objeto cliente.

```
comprador.setImporte(100);
```

Si el cliente comprador, por ejemplo, tenía inicialmente un importe de 0, mediante esta sentencia se lo cambiamos a 100.

Desde un objeto llamamos a las funciones miembro para realizar una determinada tarea. Por ejemplo, desde el cliente comprador llamamos a la función getImporte() para obtener el importe de dicho cliente.

```
Comprador.getImporte();
```

La función miembro getImporte() devuelve un número, que guardaremos en una variable adeuda, para luego usar este dato.

```
float adeuda=comprador.getImporte();
System.out.println("El importe adeudado es "+adeuda);
```

Como veremos mas adelante, no siempre es posible acceder a los miembros, el acceso dependerá de los controles de acceso a los mismos.

Ciclo de vida de los objetos

Cuando se ejecuta un programa orientado a objetos ocurren tres sucesos.

- Los objetos se crean a medida que se necesitan.
- Los mensajes se mueven de un objeto a otro (o del usuario a un objeto) a medida que el programa procesa información o responde a la entrada del usuario.
- Cuando los objetos ya no se necesitan, se borran y se libera la memoria, dicha tarea la realiza el recolector de basura (garbage collector).

DECLARACION DE MIEMBROS UNA CLASE

Los datos de una clase se denominan **atributos** y las funciones de una clase se denominan **métodos**.

Los miembros tienen ciertas restricciones en el modo en que se puede manipular los mismos dentro y fuera de la clase, a esto se le llama control de acceso a una clase o visibilidad.

Con estas restricciones se logra la encapsulación que, como vimos en la introducción, consiste en separar los aspectos externos del objeto, a los cuales pueden acceder otros objetos, de los detalles de implementación del mismo, que quedan ocultos para los demás. La encapsulación se basa en la noción de servicios prestados; Una clase proporciona un cierto número de servicios y los usuarios de esta clase no tienen que conocer la forma como se prestan estos servicios. Contrariamente a las cuentas de una asociación caritativa, una clase debe mantenerse opaca.

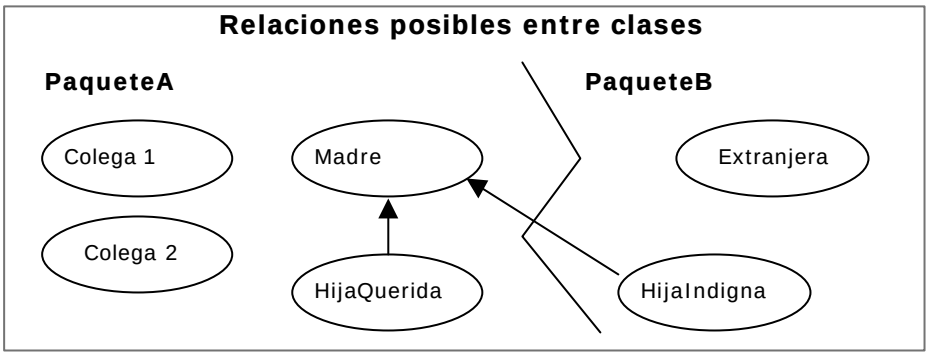
Hay que distinguir pues en la descripción de la clase dos partes:

- la parte pública, accesible por las otras clases.
- a parte privada, accesible únicamente por los métodos de la clase.

Se recomienda encarecidamente poner los atributos de una clase en la parte privada, para respetar el concepto de encapsulamiento.

Modificadores de acceso a miembros de clases

Java proporciona varios niveles de encapsulamiento que vamos a examinar a continuación.



Hemos representado en este dibujo una clase Madre alrededor de la cual gravitan otras clases:

sus hijas **HijaQuerida e HijaIndigna**, la segunda de las cuales se encuentra en otro paquete; estas dos clases heredan de Madre. la herencia se explica en detalle algo más adelante, pero retenga que las clases HijaQuerida e HijaIndigna se parecen mucho a la clase Madre. Los paquetes se detallan igualmente algo más adelante; retenga que un paquete o *package* es un conjunto de clases relacionadas con un mismo tema destinada para su uso por terceros, de manera análoga a como otros lenguajes utilizan las librerías.

sus **colegas**, que no tienen relación de parentesco pero están en el mismo paquete;

una clase **Extranjera**, sin ninguna relación con la clase Madre.

En el esquema anterior, así como en los siguientes, la flecha simboliza la relación de herencia.

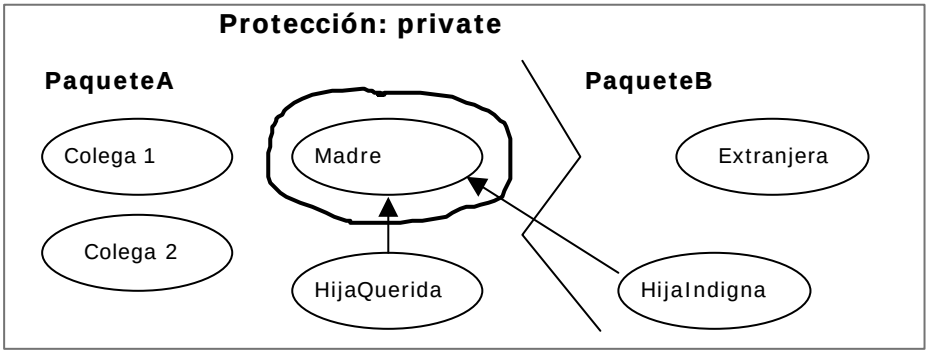
En los párrafos siguientes vamos a examinar sucesivamente los diferentes tipos de protección que Java ofrece a los atributos y a los métodos. Observemos ya desde ahora que hay cuatro tipos de protección posibles y que, en todos los casos, la protección va sintácticamente al principio de definición, como en los dos ejemplos siguientes, donde private y public definen los niveles de protección:

```
private void Metodo ();
public int Atributo;
```

Private

La protección más fuerte que puede dar a los atributos o a un método es la protección **private**.

Esta protección impide a los objetos de otras clases acceder a los atributos o a los métodos de la clase considerada. En el dibujo siguiente, un muro rodea la clase Madre e impide a las otras clases acceder a aquellos de sus atributos o métodos declarados como **private**.



Insistimos en el hecho de que la protección no se aplica a la clase globalmente, sino a algunos de sus atributos o métodos, según la sintaxis siguiente:

```
private void MetodoMuyProtegido () {
      //    ...
}
private int AtributoMuyProtegido;
public int OtraCosa;
```

Observe que un objeto que haya surgido de la misma clase puede acceder a los atributos privados, como en el ejemplo siguiente:

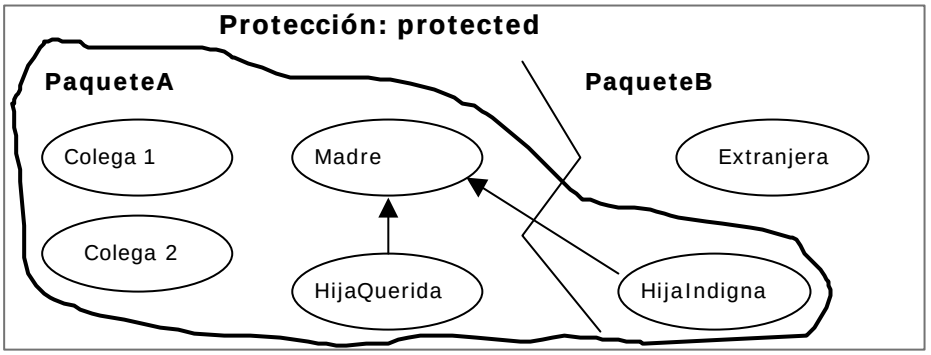
```
class Secreta {
```

```
            private int s;
            void init (Secreta otra) {
                  s = otra.s;
            }
      }
```

La protección se aplica pues a las relaciones entre clases y no a las relaciones entre objetos de la misma clase.

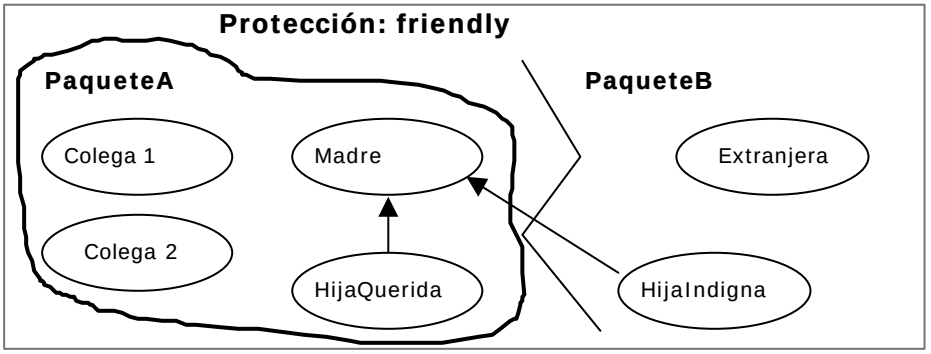
Protected

El tipo de protección siguiente viene definido por la palabra clave **protected** simplemente. Permite restringir el acceso a las subclases y a las clases del mismo paquete.



Friendly

El tipo de protección predeterminado se llama **friendly.** En Java si no se indica explícitamente ningún nivel de protección para un atributo o un método, el compilador considera que lo ha declarado friendly. No tiene por qué indicar esta protección explícitamente. Además, friendly no es una palabra clave reconocida. Esta protección **autoriza el acceso a las clases del mismo paquete**, pero no incluye a las subclases.



Los miembros con este tipo de acceso pueden ser accedidos por todas las clases que se encuentren en el paquete donde se encuentre también definida la clase.

Para recordar:

- Friendly es el tipo de protección asumido por defecto.
- Un paquete se define por la palabra reservada package.
- Un paquete puede ser nominado o no.
- Las clases se codifican en archivos .java.
- Varios archivos pueden pertenecer al mismo paquete.
- Un archivo solo pertenece a un paquete. Solo se permite una cláusula package por archivo.
- Los archivos que no tienen cláusulas package pertenecen al paquete unnamed.

Public

El último tipo de protección es public. Un atributo o un método calificado así es accesible para todo el mundo.

¿Un caso excepcional? No. Muchas clases son universales y proporcionan servicios al exterior de su paquete: las librerías matemáticas, gráficas, de sistema, etc., están destinadas a ser utilizadas por cualquier clase. Por el contrario, una parte de estas clases está protegida, a fin de garantizar la integridad del objeto.

Se suele decir que los miembros públicos conforman la interfaz de la clase con el usuario.

Separación de la interfaz

¿Cuándo debe utilizarse qué? O en otras palabras: ¿cuáles son los diferentes casos de utilización de los mecanismos de protección? Nos proponemos distinguir principalmente dos casos:

- el atributo o el método pertenece a la *interfaz* de la clase: debe ser public;
- el atributo o la clase pertenece al *cuerpo* de la clase: debe ser protegido. Esta protección es diferente según los casos; en general, la protección más fuerte es aconsejable porque es fácil desproteger un atributo, y es mucho más difícil hacerlo inaccesible si ya se utiliza.

La **interfaz** de la que hablamos es pues la interfaz conceptual de la clase, es decir los atributos y las signaturas de los métodos (tipo devuelto + nombre + parámetros), directamente utilizados desde el exterior porque corresponden a un servicio prestado.

El **cuerpo** de la clase es la implementación de dicho servicio.

Es decir, la interfaz de la clase es el ***qué*** -qué hace la clase-, mientras que su cuerpo es el ***cómo*** -cómo lo hace.

Por esto conviene hacer pública la interfaz y proteger el cuerpo.

Ahora, veamos un ejemplo en pseudocódigo sobre cómo crear un objeto reloj que demuestre cuáles serían sus funciones públicas y sus funciones privadas:

Función inicial o Constructor:
Reloj negro, hora inicial 12:00am;
Funciones Públicas:
Apagar
Encender
Poner despertador;
Funciones Privadas:
Mecanismo interno de control
Mecanismo interno de baterías
Mecanismo de manecillas

Al utilizar uno de estos relojes nos importa su operación no su mecanismo interno, por eso existen funciones públicas o privadas. Las funciones públicas son la interfaz que usaremos. El constructor inicializa al objeto en un estado inicial estable para su operación.

Un ejemplo más, esta vez con una computadora:

Función inicial o Constructor:
Computadora portátil compaq, sistema operativo windows98, encendida
Funciones Públicas:
Apagado
Teclado
Pantalla
Impresora
Bocinas
Funciones Privadas:
Caché del sistema
Procesador
Dispositivo de Almacenamiento
Motherboard

Obviamente si se abre (físicamente) la computadora se tendrá acceso a todo, sucede lo mismo si se abre el código fuente del archivo de una clase, se puede modificar todo, pero debe quedar bien claro que la razón por la cual se hace todo lo anterior es para organizar mejor el código, no para impedir el acceso a nadie a ciertas cosas, todo es para mejorar la operación general de una clase ocultando la información que no es necesario que conozcamos y exponiendo la que sí.

ATRIBUTOS DE UNA CLASE

- Todas las variables se declaran dentro del cuerpo de la clase o dentro de un método.
- Las variables declaradas dentro de un método son ***locales*** a él.
- las variables declaradas en el cuerpo de la clase se dice que son ***miembros*** de ella y son accesibles por todos los métodos de la clase.
- Se puede acceder a todos los atributos de la clase de la cual desciende. Por ejemplo, cualquier clase que descienda de la clase **Polygon** hereda los atributos **npoints, xpoints** e **ypoints**. Esto lo veremos con mayor detalle cuando estudiemos herencia.
- Los atributos miembros de la clase pueden ser *atributos de clase* o *atributos de instancia:*
  - Atributos *de clase* si se usa la palabra clave ***static***: en ese caso la variable es única para todas las instancias (objetos) de la clase (ocupa un único lugar en memoria).
  - Atributos *de instancia*, no se usa static, el sistema crea un lugar nuevo para esa variable con cada instancia (o sea que es independiente para cada objeto).
- Los atributos pueden ser:
  - tipos básicos. (no son clases)
  - clases e interfases (clases de tipos básicos, clases propias, de terceros, etc.)
- La lista de atributos sigue el mismo formato de C: se define primero el tipo y luego el nombre del atributo y, finalmente, un ";". Por ejemplo:

```
int n_entero;
float n_real;
char p;
```

- La declaración sigue siempre el mismo esquema:

[**Modificador de acceso**] [**static**] [**final**] [**transient**] [**volatile**] **Tipo NombreVariable [= Valor];**

El modificador de acceso puede ser alguno de los que vimos anteriormente (private, protected, public, etc).

static sirve para definir un atributo como de clase, o sea, único para todos los objetos de ella.

final, como en las clases, determina que un atributo no pueda ser sobreescrito o redefinido, es decir que no se trata de una variable, sino de una *constante.*

transient denomina atributos que no se graban cuando se archiva un objeto, o sea que no forman parte del estado permanente de éste.

volatile se utiliza con variables modificadas en forma asincrónica por objetos en diferentes *threads* (literalmente "hilos", tareas que se ejecutan en paralelo). Básicamente, esto implica que distintas tareas pueden intentar modificar la variable de manera simultánea, y volatile asegura que se vuelva a leer la variable (por si fue modificada) cada vez que se la va a usar.

Atributos estáticos de una clase

Le hemos explicado anteriormente que cada objeto poseía sus propios atributos. Es posible que todos los objetos de una misma clase tengan atributos en común: son los atributos de clase, introducidos por la palabra clave static. Estos atributos son legibles y modificables por todos los objetos de una misma clase. La modificación de un atributo static es tenida en cuenta inmediatamente por los otros objetos, porque lo *comparten.*

Un miembro de una clase se puede declarar estático (static). Para un miembro dato, la designación static significa que existe sólo una instancia de ese miembro en la clase. Un miembro dato estático es compartido por todos los objetos de una clase y existe incluso si ningún objeto de esta clase existe siendo su valor común a la clase completa.

A un miembro dato static se le asigna una zona fija de almacenamiento en tiempo de compilación, al igual que una variable global, pero el identificador de la variable está dentro del ámbito utilizando solamente el operador de resolución con el nombre de la clase.

Ejemplo:

```
import java.io.*

class Participante {
       static int participado = 2;
       int noparticipado = 2;
```

```
        void Modifica () {
            participado = 3;
            noparticipado = 3;
        }
    }

    class demostatic {
        static public void main (String [] arg) {
            Participante p1 = new Participante ();
            Participante p2 = new Participante ();
            System.out.println("pl: " + pl.participado + " " +
                pl.noparticipado);
            pl.Modifica ();
            System.out.println("pl: " + pl.participado + " " +
                pl.noparticipado);
            System.out.println("p2: " + p2.participado + " " +
                p2.noparticipado);
        }
    }
```

dará como resultado:

```
C:\Programasjava\objetos>java demostatic
pl: 2 2
pl: 3 3
p2: 3 2
```

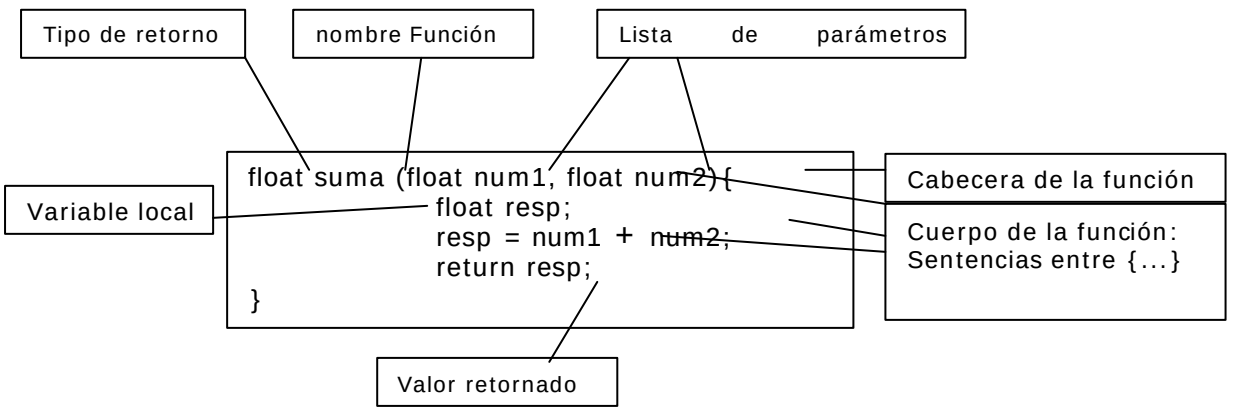
En efecto, la llamada a Modifica () ha modificado el atributo participado. Este resultado se extiende a todos los objetos de la misma clase, mientras que sólo ha modificado el atributo noparticipado del objeto actual.

METODOS DE UNA CLASE

Un método es una colección de sentencias que ejecutan una tarea específica. En Java, un método siempre pertenece a una clase. Es decir, el método siempre se ejecuta sobre un objeto. No se puede ejecutar un método sin precisar el objeto sobre el que se aplica (salvo una excepción que veremos más adelante).

Los métodos de una clase definen las operaciones que un usuario puede realizar con los atributos de la clase. Desde el punto de vista de la POO, el conjunto de todas los métodos definen el conjunto de mensajes a los que los objetos de las citadas clases pueden responder.

Básicamente, los métodos son como las funciones de C: implementan operaciones y estructuras de control, el cálculo de algún parámetro que es el que devuelven al objeto que los llama. Sólo pueden devolver un valor (del tipo TipoDevuelto), aunque pueden no devolver ninguno (en ese caso TipoDevuelto es void). Como ya veremos, el valor de retorno se especifica con la instrucción return, dentro del método.



Definición

La definición de un método consta de una cabecera y del cuerpo del método encerrado entre llaves. La sintaxis es la siguiente:

[**modificador de acceso**] [**static**] [**abstract**] [**final**] [**native**] [**synchronized**] TipoDevuelto NombreMétodo (tipo1 nombre1 [**,** tipo2 nómbre2]...]) [**throws** excepción [**,** excepción2]

```
{
        declaraciones de variables locales;
        sentencias;
        [return [ (] expresión [)] ];
}
```

Las variables declaradas en el cuerpo del método son locales a dicho método y por definición solamente son accesibles dentro del mismo.

El *modificador de acceso* es la palabra clave que modifica el nivel de protección predeterminado del método, puede ser alguno de los que vimos anterioremente (private, protected, public, etc).

El *tipoDevuelto* especifica qué tipo de valor retorna el método. Éste puede ser cualquier tipo primitivo o referecniado. Para indicar que no se devuelve nada, se utiliza la palabra reservada void. El resultado de un método es devuelto a la sentencia que lo invocó, por medio de la siguiente sentencia:

return [(] expresión [)]

En el caso de que el método no retorne un valor (void), se puede omitir o especificar simplemente return. Por ejemplo:

```
void escribir()
{
        //...
        return;
}
```

La *lista de parámetros* de un método son las variables que reciben los valores de los argumentos especificados cuando se invoca al mismo. Consiste en una lista de cero, uno o más identificadores con sus tipos, separados por comas. Estos parámetros pueden ser de cualquiera de los tipos válidos. Si son tipos básicos, el método recibe el *valor* del parámetro; si son arreglos, clases o interfaces, recibe un puntero a los datos *(referencia).* Veamos un pequeño ejemplo:

```
Public int AumentarCuenta(int cantidad) {
        cnt = cnt + cantidad;
        return cnt;
}
```

Este método, si lo agregamos a la clase Contador, le suma cantidad al acumulador cnt. En detalle:

- el método recibe un valor entero (cantidad).
- lo suma a la variable de instancia cnt.
- devuelve la suma (return cnt).

El resto de la declaración

Los métodos estáticos (**static**) son, como los atributos, métodos *de clase:* si el método no es static, es un método *de instancia.* El significado es el mismo que para los atributos: un método static es compartido por todas las instancias de la clase.

Los métodos abstractos (**abstract**) son aquellos de los que se da la declaración pero no la implementación (o sea que consiste sólo en el encabezamiento). Cualquier clase que contenga al menos un método abstracto (o cuya clase madre contenga al menos un método abstracto que no esté implementado en la hija) es una clase abstracta.

Un método es final (**final**) cuando no puede ser redefinido por ningún descendiente de la clase.

Los métodos **native** son aquellos que se implementan en otro lenguaje propio de la máquina (por ejemplo, C o C++). Sun aconseja utilizarlas bajo riesgo propio, ya que, en realidad, son ajenas al lenguaje. Pero existe la posibilidad de usar viejas bibliotecas que uno armó y no tiene ganas de reescribir, ¡a costa de perder portabilidad!

Los métodos **synchronized** permiten sincronizar varios *threads* para el caso en que dos o más accedan concurrentemente a los mismos datos.

Finalmente, la cláusula **throws** sirve para indicar que la clase genera determinadas excepciones.

Parámetros

Los métodos pueden utilizar valores que les pasa el objeto que los llama (parámetros), indicados con tipo1 nombre1, tipo2 nombre2… en el esquema de la declaración.

La lista de parámetros es la lista de nombres de variables separados por comas, con sus tipos asociados, que reciben los valores de los argumentos cuando se llama al método.

Un método puede recibir como parámetros valores prácticamente de cualquier tipo. Estos parámetros serán usados por el método para realizar operaciones que lleven a la acción que se espera generar.

Los parámetros que va a recibir un determinado método habrá que indicarlos en la declaración, tras el identificador y entre paréntesis. La declaración propiamente dicha es similar a la que realizamos cuando definimos una variable, es decir, facilitaremos los tipos y los nombres, usando la coma para separar unos parámetros de otros.

La declaración de los parámetros formales sigue una sintaxis similar a la del resto de los identificadores de Java. Por ejemplo:

void funcion(**int a, float b**);

En Java los parámetros de las funciones pueden tomar *valores por defecto*, que son asignados por el compilador si no se suministran los argumentos actuales. Por ejemplo:

void funcion(**int a=0, float b=7**);

Si una función se declara sin parámetros, Java la considera una función sin argumentos (void).

void funcion();

Los parámetros tienen el ámbito y duración de la función. Esto significa que los parámetros se podrán ver y utilizar sólo dentro de la función.

Tipos de parámetros

Los parámetros pueden ser de cualquiera de los tipos ya vistos. Si son tipos básicos, el método recibe el valor del parámetro; si son arrays, clases o interfases, recibe un puntero a los datos (referencia). En general existen dos tipos de parámetros:

- **Parámetros por valor :** Este método copia el valor de un argumento en el parámetro formal del método. De esta forma los cambios realizados en los parámetros no afectan a la variable original utilizada en la llamada.
- **Parámetros por referencia :** La llamada por referencia es la segunda forma de pasar argumentos a un método. De esta forma se copia la dirección de memoria del argumento en el parámetro. Dentro del método se usa esta dirección de memoria para acceder al contenido. Esto significa que los cambios hechos a los parámetros afectan a las variables usadas en la llamada al método.

El cuerpo de los métodos

En Java dentro de los métodos pueden incluirse:

- Declaración de variables locales.
- Asignaciones a variables.
- Operaciones matemáticas.
- Llamados a otros métodos.
- Estructuras de control.
- Excepciones (try, catch, que veremos más adelante)

Declaración de variables locales

Las variables locales en un método se declaran igual que en C:

```
Tipo NombreVariable [ = Valor];
```

Por ejemplo:

```
int    suma;
float  precio;
Contador laCuenta;
```

Las variables pueden inicializarse al momento de su creación:

```
int    suma=0;
float  precio = 12.3;
Contador laCuenta = new Contador() ;
```

Llamadas a métodos

Se llama a un método de la misma clase simplemente con el nombre del método y los parámetros entre paréntesis, como se ve, entre otros, en el ejemplo siguiente:

Otra vez recordaremos la clase Contador:

```
//Implementación de un contador sencillo
public class Contador {    //     Se declara y define la clase Contador
       int cnt;
       public void Inicializa() {
              cnt=0;       //inicializa
       }
       //Otros métodos
       public int incCuenta() {
              cnt++;
              return cnt;
       }
       public int getCuenta() {
                     return cnt;
       }
}
```

Pueden probar esta clase (mínima) con el siguiente ejemplo de aplicación:

```
import java.io.*;

public class Ejemplollamadas {
       public static void main(String args[]) {
              Contador c = new Contador;
              c.Inicializa();
              System.out.println(c.incCuenta());
              System.out.println(c.getCuenta());
       }
}
```

las líneas en letra resaltada es un ejemplo de un llamado a un método de un objeto de otra clase. Noten que en este caso, es necesario llamar al método para un objeto (instancia) existente, por lo que se indica:

Nombre_del_Objeto<punto>Nombre_del_método(parámetros)

El objeto actual (puntero this)

Nunca se puede llamar una función miembro de una clase a menos que se asocie con un objeto (una instancia de la clase). ¿Cómo sabe una función miembro cuál es la instancia de una clase (el objeto específico) asociada con ella?.

El método utilizado por Java es añadir un argumento extra oculto a las funciones miembro. Este argumento es un puntero al objeto de la clase que los enlaza con la función asociada y recibe un nombre especial denominado **this**.

Dentro de una función miembro, this apunta al objeto asociado con la invocación de la función miembro. Normalmente, el programador no necesita preocuparse por este puntero, ya que el lenguaje realiza la operación automáticamente transparente a las funciones miembro que la utilizan.

Las referencias a los miembros del objeto asociado a la función se realiza con el prefijo this y el operador de acceso punto **.** .

Si tomamos como ejemplo la siguiente clase:

```
public class Cliente {
       private int codigo;
       private float importe;
       public int   getCodigo()   { return codigo; }
       public float getImporte()  { return importe; }
       public void setImporte(float x) { importe = x; }
};
```

Cuando escribimos en el método setImporte

```
importe = x;
```

para asignar un valor a importe, no tuvimos necesidad de indicar a qué objeto pertenecía. Cuando no se pone el objeto antes del atributo, se asume que la variable es un miembro del objeto, o es una variable local o parámetro. Si los nombres colisionan, como podría ser en el siguiente método

```
public void setImporte(float importe)
{
       this.importe = importe;
}
```

usamos this para indicar al objeto actual. Esta palabra reservada siempre está apuntando al objeto actual, y puede usarse como en este caso, para resolver una ambigüedad, o puede usarse, por ejemplo, como parámetro en una llamada a una función para pasar un puntero al objeto asociado:

```
objeto.procesar(this);
```

Métodos especiales

Métodos sobrecargados

Tradicionalmente, un método o una función realizan una tarea específica para la que están programados. Java soporta la *sobrecarga de métodos,* lo que le permite definir versiones de éstos con el mismo nombre en una clase, siempre y cuando las versiones tengan diferentes firmas. Una firma incluye el nombre del método, el número, el orden y los tipos de sus parámetros formales. Como ejemplo simple, considere el reestablecímiento del saldo de cuenta de un objeto Account, esta clase (Account) ya tiene un método

```
public double balanceo()
{
	return saldo;
}
```

que se utiliza para recuperar el saldo de la cuenta. Con la sobrecarga podemos definir un método:

```
public void balanceo(double valor)
{
	saldo = valor;
}
```

que fija el saldo en una cantidad específica. El mismo método recuperará o establecerá el saldo, dependiendo de que se proporcione un argumento; ¡es muy útil!

Tenga en cuenta que, sólo ocurre la sobrecarga, cuando se utiliza varias veces el mismo nombre de método dentro de una clase. No hay un límite práctico para las versiones que es posible apilar en el mismo nombre de método.

Resolución de llamada a un método

Cuando se hace una llamada aun método sobrecargado Java deduce automáticamente, a partir de los argumentos reales, la versión correcta del método que habrá de invocar. A esta actividad se le denomina *resolución de llamada,* Java la realiza al seleccionar un método entre los accesibles *que son aplicables.*

Un método es *aplicable* si toma el mismo número de parámetros que los argumentos dados y cada uno de éstos puede transformarse por *conversión de invocación de método* al tipo del parámetro.

El compilador realiza la resolución de llamada de método al comparar el número y tipo de los argumentos reales con firmas de todos los métodos accesibles y elige un método aplicable que es el *más* especifico.

En resumen, las funciones sobrecargadas tienen el mismo nombre, pero deben tener un número diferente de argumentos o diferentes tipos de argumentos, o ambos. Por ejemplo:

```
void visualizar();
void visualizar(int cuenta);
void visualizar(int cuenta, int max);
```

Ejemplo:

Supongamos que tenemos una clase Media, que calcula la media de dos y tres números, enteros y reales. Para esto tendremos las siguientes funciones:

```
float media (float, float);       //calcula la media de dos valores tipo float
int media (int, int);             //calcula la media de dos valores tipo int
float media (float, float, float); //calcula la media de tres valores tipo float
int media (int, int, int);        // calcula la media de tres valores tipo float
```

Entonces:

```
public class Media {
	public float Cal_Media (float a, float b)
```

```
        { return (a+b)/2.0; }
        public int Cal_Media (int a, int b)
              { return (a+b)/2; }
        public float Cal_Media (float a, float b, float c)
        { return (a+b+c)/3.0;}
        public int Cal_Media (int a, int b, int c)
        { return (a+b+c)/3; }
};

public class demoMedia {
        public static void main (String arg[]) {
              Media M = new Media();
              float x1, x2, x3;
              int y1, y2, y3;
              ...
                    System.out.println(M.Cal_Media (x1, x2));
                    System.out.println(M.Cal_Media (x1, x2, x3));
                    System.out.println(M.Cal_Media (y1, y2));
                    System.out.println(M.Cal_Media (y1, y2, y3));
        }
}
```

Métodos constructores

Para cada clase, pueden definirse uno o más métodos particulares: son los constructores.
Un constructor es una función especial que sirve para construir o inicializar objetos.
En general:

- Tienen el mismo nombre de la clase que inicializa.
- No devuelven valores.
- Pueden admitir parámetros como cualquier otra función.
- Pueden existir más de un constructor, e incluso no existir.
- Si no se define ningún constructor de una clase, el compilador generará un constructor por defecto.
- Se llaman en el momento de la creación del objeto.

```
//Implementación de un contador sencillo
//GRABAR EN UN ARCHIVO "Contador.java"
// COMPILAR CON: "javac Contador.Java
public class Contador {    //     Se declara y define la clase Contador
        int cnt;
        public Contador() {
              cnt = 0;            //inicializa en 0
        }
        public Contador(int c) {
              cnt = c;            //inicializa con el valor de c
        }
        //Otros métodos
        public int getCuenta() {   return cnt;}
        public int incCuenta() {   cnt++;return cnt;
        }
}
```

Pueden probar esta clase (mínima) con el siguiente ejemplo de aplicación:

```
// Archivo:   EjemploConstructor.java
//Compilar con: javac EjemploConstructor.java
//Ejecutar con: java EjemploConstructor
import java.io.*;

public class EjemploConstructor {
        public static void main(String args[]) {
              Contador c1 = new Contador;
              Contador c2 = new Contador(20);
```

```
                System.out.println(c1.getCuenta());
                System.out.println(c2.getCuenta());
        }
}
```

Cuando, desde una aplicación u otro objeto, se crea una instancia de la clase Contador, mediante la instrucción:

```
Contador c1 = New Contador();
```

el compilador busca un método con el mismo nombre de la clase y que se corresponda con la llamada en cuanto al tipo y número de parámetros. Dicho método se llama Constructor, y una clase puede tener más de un constructor (no así un objeto instanciado, ya que, una vez que fue creado, **no** puede "recrearse" a sí mismo).

En tiempo de compilación o ejecución, según corresponda, al encontrarse dicha instrucción, se reserva espacio para el objeto instanciado y se crea su estructura y en tiempo de ejecución se llama al método constructor.

La utilización de las instrucciones vistas, por ejemplo new(), entraña en efecto la creación física del objeto y la llamada a uno de sus constructores. Si hay varios constructores, estos difieren unos de otros por los parámetros que se les pasa mediante new().

Por ejemplo:

```
import java.io.*;
// una clase que tiene dos constructores diferentes
class Ejemplo {
        public Ejemplo (int param) {
                System.out.println ("Ha llamado al constructor");
                System.out.println ("con un parámetro entero");
        }
        public Ejemplo (String param) {
                System.out.println ("Ha llamado al constructor'.);
                System.out.println ("con un parámetro String");
        }
}

// una clase que sirve de main
public class democonstructor {
        public static void main (String arg[]) {
                Ejemplo e;
                e = new Ejemplo (2);
                e = new Ejemplo ("2");
        }
}
```

da el resultado siguiente:

```
c:\Programasjava\objetos>java democonstructor
Ha llamado al constructor con un parámetro entero
Ha llamado al constructor con un parámetro String
```

Los constructores no tienen tipo de retorno. Atención, si por error definimos un constructor que tenga un tipo de retomo, el compilador lo considerará como un método normal. En ese caso tendremos la impresión de que el constructor no se llama en el momento de la creación del objeto. En realidad, se llamará a un constructor predeterminado, porque no habremos definido realmente un constructor.

Tipos de constructores

Constructores por defecto

El constructor por defecto es un constructor que no acepta argumentos. Por ejemplo:

```
class Punto {
        int x;
        int y;
        public Punto()
        {       x = 0;
```

```
                y = 0;
        }
}
```

Para crear objetos usando este constructor se escribiría:

```
public class constructorPorDefecto {
        public static void main (String arg[]) {
                Punto p1;
                p1 = new Punto();
        }
}
```

Constructores con argumentos

El constructor con argumentos, como su nombre lo indica posee argumentos. Por ejemplo:

```
class Punto {
        int x;
        int y;
        public Punto(int x, int y)
        {       this.x = x;
                this.y = y;
        }
}
```

Para crear objetos usando este constructor se escribiría:

```
public class constructorArgumentos {
        public static void main (String arg[]) {
                Punto p2;
                p2 = new Punto(2, 4);
        }
}
```

Constructores copiadores

Un constructor que crea un objeto a partir de uno existente se llama constructor copiador o de copias. Es un constructor que toma como único parámetro otro objeto del mismo tipo. El constructor de copia tiene sólo un argumento: una referencia a un objeto de la misma clase. Por ejemplo:

```
class Punto {
        int x;
        int y;
        public Punto(Punto p)
        {       x = p.x;
                y = p.y;
        }
}
```

Para crear objetos usando este constructor se escribiría:

```
public class constructorCopia {
        public static void main (String arg[]) {
                Punto p2;                   //del ejemplo anterior
                p2 = new Punto(2, 4);

                Punto p3;
                p3 = new Punto(p2);         //p2 sería el objeto creado en el
                                            //ejemplo anterior
        }
}
```

Caso especial

En Java es posible inicializar los atributos indicando los valores que se les darán en la creación del objeto. Estos valores se adjudican tras la creación física del objeto, pero antes de la llamada al constructor.

```
import java.io.*
class Reserva {
```

```
                int capacidad = 2;
                // valor predeterminado
                public Reserva (int lacapacidad) {
                        System.out.println (capacidad);
                        capacidad = lacapacidad;
                        System.out.println (capacidad);
                }
        }

        class demovalor {
                static public void main (String []arg) {
                        new Reserva (1000);
                }
        }
```

visualizará sucesivamente el valor que el núcleo ha dado al atributo capacidad tras la inicialización (2) y posteriormente el valor que le asigna el constructor (1000).

Añadamos que los atributos no inicializados explícitamente por el desarrollador tienen valores predeterminados, iguales a cero. Así, si no se inicializa capacidad:

```
        class Reserva {
                int capacidad:
                public Reserva (int lacapacidad) {
                        System.out.println (capacidad);
                        capacidad = lacapacidad;
                        System.out.println (capacidad);
                }
        }
```

el programa indicará los valores 0 seguido de 1000.

CLASE PRACTICA

Ejercicios prácticos

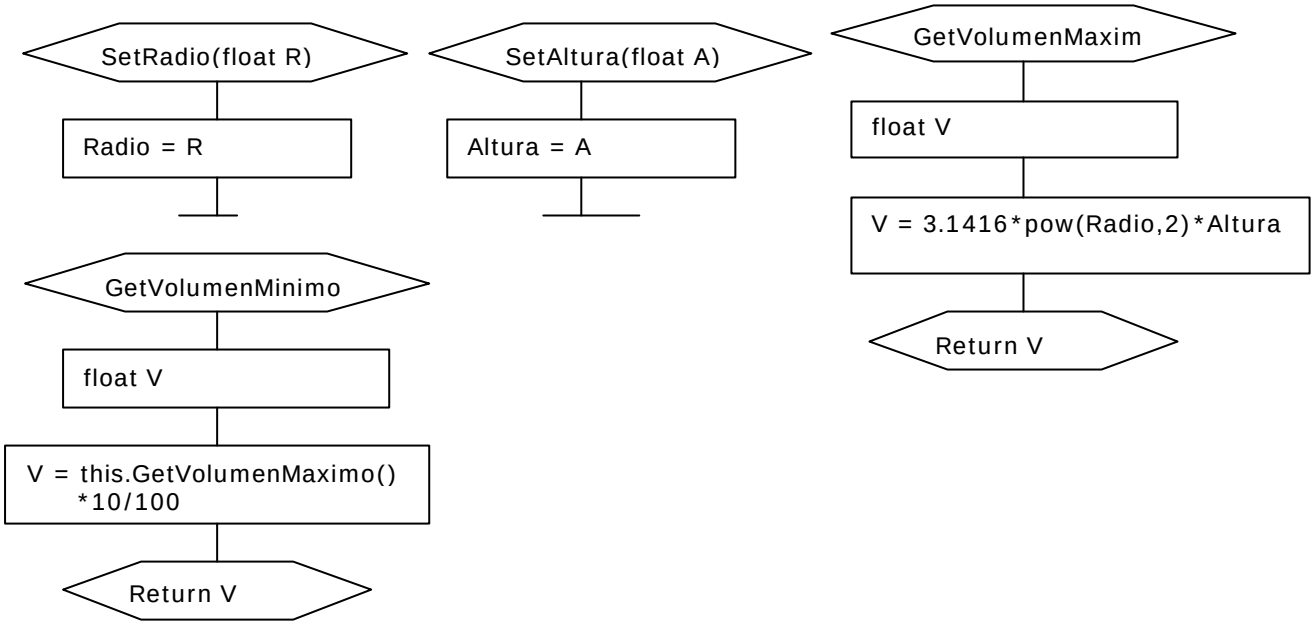
Para la clase práctica se sugiere ejercicios que identifiquen clases y que grafiquen el cuerpo de los métodos mediante diagramas de flujo, comenzar con codificación.

1. El dueño de un campo que almacena sus cosechas de granos en silos cilíndricos le pide que desarrolle un programa que le permita conocer el volumen máximo de un silo a partir de su radio y su altura.
El programa debe ser capaz también de informar el volumen mínimo sabiendo que este se define como el 10% del volumen máximo.
En un cilindro, su volumen interno se calcula como el número pi por el radio al cuadrado por la altura.

Solución.
Diagrama de clase:

Clase Silo
Radio Altura
Silo() SetRadio() SetAltura() GetVolumenMaximo() GetVolumenMinimo()

Diagrama de flujo de los métodos



Código fuente en Java

```
import java.io.*;
import utn.frc.io.In;

public class Silo
{
	private float radio, altura;

	public Silo()
	{
		radio = 0;
		altura = 0;
	}
	public void SetRadio(float r)
	{
		radio = r;
	}
	public void SetAltura(float a)
	{
		altura = a;
	}
	public double GetVolumenMaximo()
	{
		double v;
		v=3.1416*radio*radio*altura;
		return v;
	}
	public double GetVolumenMinimo()
	{
		double v;
		v=this.GetVolumenMaximo()*10/100;
		return v;
	}
}
```

```
class programa
{
 public static void main(String args[])
 {
       Silo unSilo = new Silo();
       float r,a;
       double v;
       System.out.println("Ingrese la altura del silo: ");
       a=In.readInt();
       System.out.println("Ingrese el radio del silo: ");
       r=In.readInt();
       unSilo.SetRadio(r);
       unSilo.SetAltura(a);
       v=unSilo.GetVolumenMaximo();
       System.out.println("El volumen maximo del cilo es: "+v);
       v=unSilo.GetVolumenMinimo();
       System.out.println("El volumen minimo del cilo es: "+v);

 }
}
```

2. Se quiere modelizar la entidad venta que tiene los siguientes datos: fecha, importe bruto, tipo (1-mayorista, 2-minorista) y forma de pago (1- efectivo, 2- cheque, 3- tarjeta). Se sabe que a una venta mayorista se le hace un descuento del 15%, si la forma de pago es efectivo se hace un descuento del 10% y si la forma de pago es con tarjeta se hace un recargo del 10%. Se pide un método que retorne el importe neto de la venta.

3. En un hotel que maneja reservaciones de las habitaciones, se necesita saber el precio final de la reserva, para ello se sabe que:

- Cada reserva tiene: numero, cantidad de personas, precio por persona, numero de habitación y fecha.
- Existe una promoción que si reservan 3 personas pagan 2 y si reservan 5 pagan 4.

Se pide modelizar la entidad reserva y hacer un método que retorne el precio total de la reserva.

4. Construir la clase “fracionario” de manera que permita representar numeros fraccionarios con su respectivo numerador y denominador.
Desarrollar los siguientes métodos:
fracToDec(). Retorna el numero fraccionario en formato decimal.
suma() Retorna la suma de dos fraccionario.
resta() Retorna la resta de dos fraccionarios.
multiplicación() Retorna la multiplicación de dos fraccionarios.
división() Retorna la división de dos fraccionarios.
decToFrac() toma un numero decimal y lo retorna su fraccionario equivalente.

5. Construir la clase “complejo” de manera que permita representar números complejos con su parte real y su parte imaginaria.
Desarrollar los siguientes métodos:
getReal() Retorna la parte real del número complejo
getImaginario() Retorna la parte imaginaria del número complejo.
suma() Retorna el número complejo constituido por la suma de otros dos.
resta()Retorna el número complejo constituido por la resta de otros dos.
multiplicación() Retorna el complejo constituido por la multiplicación de otros dos
división() Retorna el complejo constituido por la división de otros dos.
multiplicaciónPorReal() Retorna el resultado de multiplicar un complejo por un real
devisiónPorReal() Retorna el resultado de dividir un complejo por un real

6. La ecuación de una recta en el plano es de la forma y=ax+b. Desarrolle la clase “recta” de manera que permita representar rectas en un plano y construya los siguientes métodos.
comprarRecta() Retorna verdadero si dos rectas son exactamente iguales.
paralela() Retorna verdadero si dos rectas son paralelas.
corta() Retorna verdadero si una recta cortara a otra en algún punto del plano.

pertenecePunto() Retorna verdadero si un punto pertenece a una recta.
getY() Retorna el valor de y para un cierto valor de x.
getX() Retorna l valor de x para un cierto valor de y.