Unidad II

Algoritmos con TIPO ABSTRACTO DE DATOS

Año 2008

> "Un hombre que quiera ser bueno entre tantos que no lo son llevará al desastre. Por eso es necesario que un príncipe que quiera mantenerse, aprenda a poder no ser bueno y a saber servirse de ello."
>
> Nicolas Maquiavelo

Autor: Ing. Tymoschuk, Jorge
Colaboración: Ing. Guzmán, Analía

Indice (Unidad II – Algoritmos con Tipo Abstracto de Datos

Objetivos de la Unidad

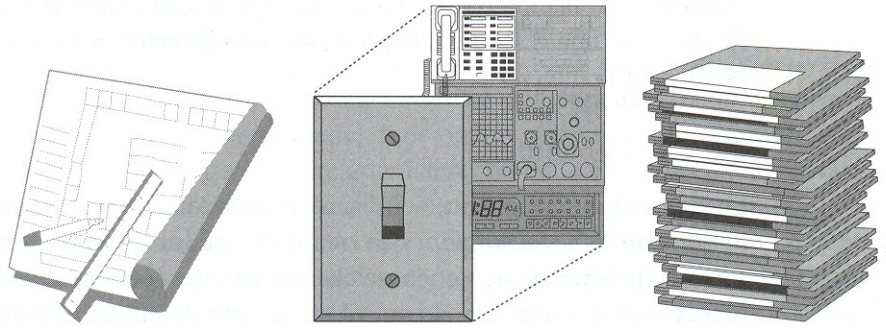
Desarrollar la capacidad de razonamiento y lógica necesarios para identificar problemas algorítmicos que usan tipo abstracto de datos y resolverlos. Esto implica: abordar problemas reales, analizarlos, definir la estrategia de su resolución, explicitar los algoritmos necesarios y codificarlos en un lenguaje de programación.

En esta Unidad trataremos con algoritmos que resuelven problemas usando el comportamiento de una única clase, dejando para la Unidad III el tratamiento de problemas algorítmicos cuya resolución involucra el comportamiento de varias clases.

Programación orientada a Objetos

Principios de diseño

Entre los principios del método orientado a objetos, los más importantes que pretenden llegar a los objetivos que se describieron en la introdución (Unidad I), son los siguientes:



Abstracción Encapsulamiento Modularidad

Abstracción

La noción de abstracción es descomponer un sistema complicado en sus partes más fundamentales, así como describir esas partes con un lenguaje sencillo y preciso. En forma característica, describir las partes de un sistema consiste en nombrar y describir su funcionalidad. Por ejemplo, una interfaz gráfica normal con usuario (GUI) en editor de texto proporciona una abstracción de un menú "editar" que tenga varias operaciones de edición de textos, incluyendo el corte y pegado de partes de texto o de otros objetos gráficos. Sin entrar en detalles sobre las formas en que una GUI representa y muestra texto y objetos gráficos; los conceptos de corte y pegado son sencillos y precisos. Una operación de corte elimina el texto y las gráficas seleccionados y los coloca en una memoria externa. Una operación de pegado inserta el contenido de la memoria externa en determinado lugar del texto. De esta forma, la funcionalidad abstracta de un menú "editar" y sus operaciones de corte y pegado se especifican en un lenguaje suficientemente preciso como para quedar claras, y al mismo tiempo son lo bastante sencillas como para "alejar" detalles innecesarios. Esta combinación de claridad y simplicidad es una ventaja para la robustez, porque conduce a implementaciones inteligibles y correctas.

La aplicación de este paradigma al diseño de las estructuras de datos origina los **tipos de datos abstractos (TDA).**

Tipo Abstracto de Datos

Un TDA es un modelo matemático de una estructura de datos que especifica:

- el tipo de datos que se guardan,
- las operaciones que se hacen con ellos
- los parámetros usados en las operaciones.

Un TDA especifica **qué hace cada operación**, pero **no cómo lo hace.**

En Java, un TDA se puede expresar mediante una interfaz, que no es más que una lista de declaraciones del método.

Luego esta interfaz se implementa en una clase y de esta manera se modela en una estructura concreta de datos. Una clase define a los datos (atributos) que la constituyen y a las operaciones (métodos) permitidas sobre sus objetos constituyentes (instancias). También, a diferencia de las interfaces, las clases especifican cómo se hacen las operaciones.

Se dice que una clase Java implementa una interfaz si sus métodos dan vida a todos los de la interfaz.

Encapsulamiento

Otro principio importante del diseño orientado a objetos es el encapsulamiento u ocultación de información, que establece que los distintos componentes de un sistema de programas no deben revelar los detalles internos de sus implementaciones respectivas. Considérese de nuevo el ejemplo de un menú Editar, con funciones de corte y pegado, en una interfaz gráfica (GUI) del editor de textos con el usuario. Una de las principales razones de que un menú de edición sea tan útil es que se puede comprender totalmente cómo usarlo, sin comprender en forma exacta cómo se implementa. Por ejemplo, no se necesita saber cómo se presenta el menú, cómo se representa el texto a cortar o pegar, cómo se guardan las partes seleccionadas del texto en una memoria externa, o cómo se identifican, guardan y copian, meten y sacan, los textos seleccionados de la memoria externa. En realidad, el programa asociado con el menú Editar debe proporcionar una interfaz suficientemente especificada para que otros componentes del programa usen los métodos con eficacia y, al mismo tiempo, se deben tener interfaces bien definidas de los demás componentes del programa que se necesiten. En términos generales, el principio de encapsulamiento establece que todos los diversos componentes de un sistema grande de programación deben funcionar con la base estrictamente necesaria de conocimiento.

Una de las principales ventajas del encapsulamiento es que permite que el programador tenga libertad de implementar los detalles de un sistema. La única restricción que tiene el programador es mantener la interfaz abstracta que ven las demás personas. Por ejemplo, el programador del menú Editar en una GUI de editor de texto podría implementar primero las operaciones de corte y pegado mandando las imágenes reales de pantalla a una memoria externa y sacándolas de allí. Después podría suceder que esta implementación no gustara al programador, porque no permite un almacenamiento compacto de la selección, y porque no distingue entre texto y objetos gráficos. Si el programador ha diseñado la interfaz de cortar y pegar teniendo en cuenta el encapsulamiento, el cambio de la implementación básica a otra que guarde el texto como objetos de texto y gráficos en un formato compacto y adecuado no debe causar problemas a los métodos que necesitan interconectarse con esta GUI. Es decir, con el encapsulamiento se obtiene adaptabilidad porque permite cambiar la implementación de detalles de las partes del programa, sin afectar a otras partes en forma adversa.

Modularidad

Además de la abstracción y el encapsulamiento, uno de los principios fundamentales del diseño orientado a objetos es la modularidad. Los programas modernos se forman por componentes distintos que deben interactuar en forma correcta, para que todo el sistema funcione bien. Para mantener despejadas esas interacciones se requiere que los distintos componentes funcionen adecuadamente. En el método orientado a objetos, la estructura se centra en el concepto de modularidad, que se refiere a una organización en la que distintos componentes de un sistema de programación se dividen en unidades funcionales separadas. Por ejemplo, se puede considerar que una casa o un departamento consisten en varias unidades que interaccionan: las instalaciones eléctricas, de calefacción y enfriamiento, el servicio sanitario y la estructura. Más que considerar que esos sistemas son un enredo gigantesco de alambres, ventilaciones, tubos y tableros, el arquitecto organizado que diseña una casa o departamento los considera como módulos separados que interactúan en formas bien definidas. Al hacer esa consideración se aplica la modularidad para aclarar las ideas, en una forma natural de organizar las funciones

en unidades distintas y manejables. En forma parecida, el uso de la modularidad en un sistema de programación también puede proporcionar una poderosa estructura organizativa que aporte claridad a una implementación. Trabajaremos en detalle este concepto en la Unidad III – Estrategias de Resolución,

Introducción a Clases (Abstracciones con procedimientos y funciones)

Como ya dijimos, una clase es una implementación de un concepto, y entonces 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. Esto puede desconcertar a los programadores de C++, que pueden definir métodos y variables globales fuera del bloque de la clase. Así pues, el esqueleto de cualquier aplicación Java se basa en la definición de una clase, o más a menudo, varias, muchas.

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 de C) puede colocarse al principio de un archivo, fuera del bloque de la clase. El compilador reemplazará esa sentencia con el contenido del archivo que se indique, que consistirá, como es de suponer, en más clases. Además de import, fuera de la clase podemos tener la palabra clave package y nada mas…

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.

  En una aplicación un objeto se instancia a partir de una descripción de su clase: es dinámico, podemos tener una infinidad de objetos de esa una misma clase.

Estructura General de una clase

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 (package) 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.

Declaración y Definición

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.

Instancias de una clase ((Objetos)

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.

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);
      }
}
```

Acceso a 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.

1. Los objetos se crean a medida que se necesitan.
2. 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.
3. Cuando los objetos ya no se necesitan, se borran y se libera la memoria.

La vida de un objeto

En el lenguaje C++, los objetos que se crean con new se han de eliminar con delete. new reserva espacio en memoria para el objeto y delete libera dicha memoria. En el lenguaje Java no es necesario liberar la memoria reservada, el recolector de basura (garbage collector) se encarga de hacerlo por nosotros, liberando al programador de una de las tareas que más quebraderos de cabeza le producen, olvidarse de liberar la memoria reservada.

Declaración de los Miembros de 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.

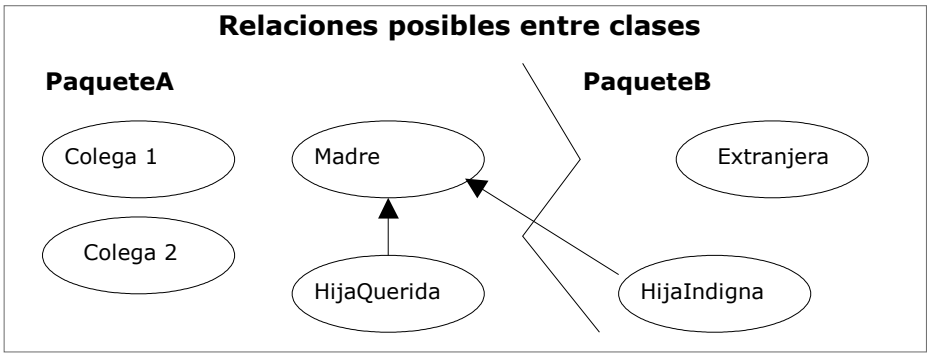
Hay que distinguir pues en la descripción de la clase dos partes:

- la parte pública, accesible por las otras clases;
- la 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.

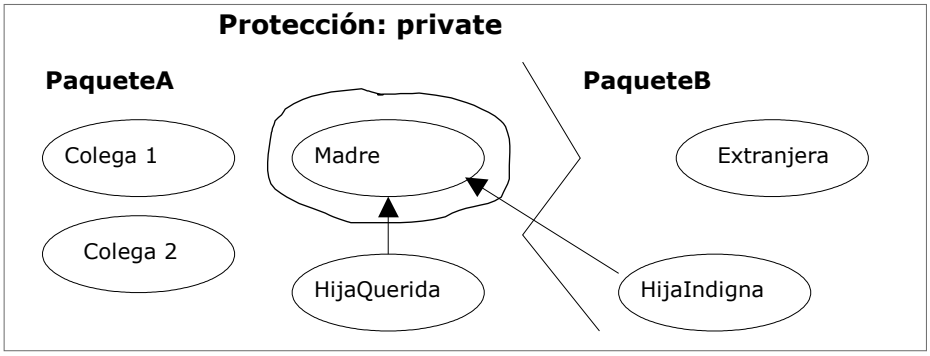
En los párrafos siguientes vamos a examinar sucesivamente los diferentes tipos de protección que Java ofrece a los atributos y a los métodos. Observemos ya desde ahora que hay cuatro tipos de protección posibles y que, en todos los casos, la protección va sintácticamente al principio de definición, como 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;
```

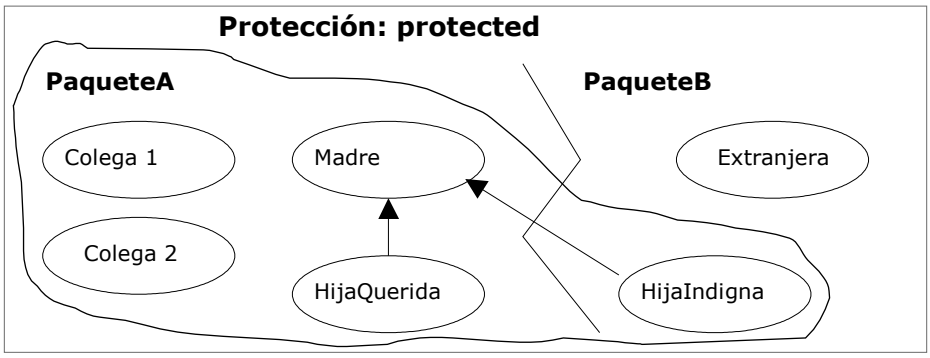
Observe que un objeto que haya surgido de la misma clase puede acceder a los atributos privados, como en el ejemplo siguiente:

```
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.

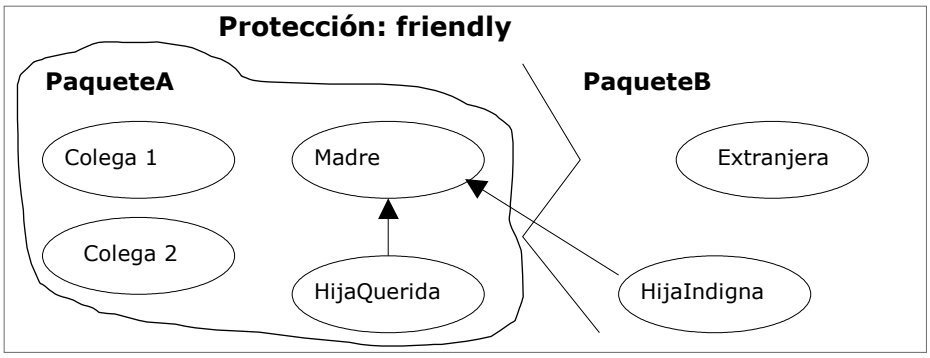
Protected

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.

Por otra parte, además de los atributos de la propia 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.
Finalmente, los atributos miembros de la clase pueden ser *atributos de clase* o *atributos de instancia;*

- ✓ se dice que son 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).
- ✓ Si 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.

Métodos de una clase

Un método es una función que 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 las funciones definen el conjunto de mensajes a los que los objetos de las citadas clases pueden responder.

Declaración y definición

Los métodos, como las clases, tienen una declaración y un cuerpo. La declaración es del tipo:

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

La declaración y definición se realizan juntas, es decir en el cuerpo de la clase. Básicamente, los métodos son como las funciones de C: implementan el cálculo de algún parámetro (que es el que devuelven al método que los llama) a través de funciones, operaciones y estructuras de control. Sólo pueden devolver un valor (del tipo TipoDevuelto), aunque pueden no devolver ninguno (en ese caso TipoDevuelto es void). El valor de retorno se especifica con la instrucción return, dentro del método.

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

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.

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);
```

Parámetros de un método

Veamos en más detalle los conceptos que tratan del paso de parámetros a un método o una función. El término paso por valor significa que el método sólo toma el valor que se pasa desde la llamada. Por el contrario, el paso por referencia implica que ese método obtiene la localización de la variable pasada en la llamada. De esta manera, un método puede modificar el valor almacenado en dicha variable, cosa que no puede hacerse en el paso por valor.
Estas especificaciones de tipo "paso por..." son estándar en la terminología computacional y describen el comportamiento de los parámetros de un método en distintos lenguajes de programación, no sólo en Java (de hecho, también existe un paso por nombre que sólo tiene interés histórico y que es utilizado en Algol, uno de los lenguajes de programación de alto nivel más antiguos).

Java siempre utiliza el paso por valor. Esto significa que el método siempre obtiene una copia de los valores de todos los parámetros. Por tanto, no podrá modificar el contenido de ninguna de las variables que recibe.

Por ejemplo, considere la siguiente llamada:

```
double porcent = 10;
harry.subirSalario(porcent);
```

Independientemente de cómo esté implementado el método, sabemos que tras la llamada al mismo, el valor de porcent sigue siendo 10.

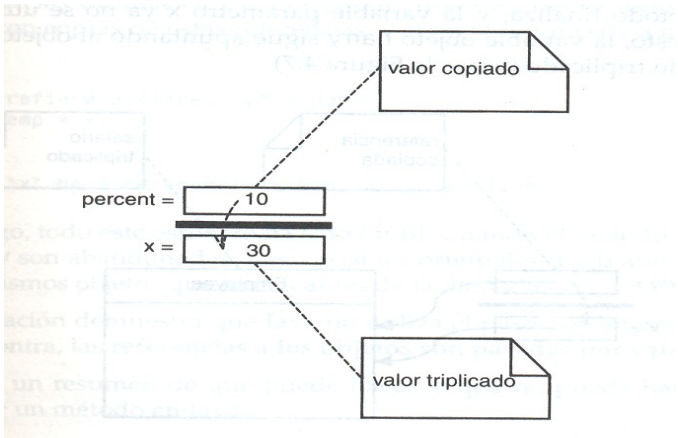
Vamos a profundizar algo más en esta situación. Supongamos un método que intente triplicar el valor de uno de sus parámetros:

```
public static void tripleValor(double x){x = 3 * x;}
double porcent = 10; tripleValor(porcent);
```

Sin embargo, esto no funciona. Tras la llamada, el valor de porcent sigue siendo 10.

Porque? detallamos:

1. x se inicializa con una copia del valor de porcent (en este caso, 10).
2. x se triplica, por lo que ahora vale 30. Pero porcent sigue con su valor en 10
3. El método finaliza, x desaparece…



Sin embargo, existen dos tipos de parámetros de un método:

. Tipos primitivos (números, valores lógicos, etc.);
. Referencias a objetos.

Vimos que resulta imposible para un método modificar un parámetro de tipo primitivo. La situación cambia con los parámetros que son objetos (esto es, parámetros objeto). Se puede implementar de forma fácil un método que triplique el salario de un empleado:

```
public static void tripleSalario(Empleado x){x.subirsalario(200);}
```

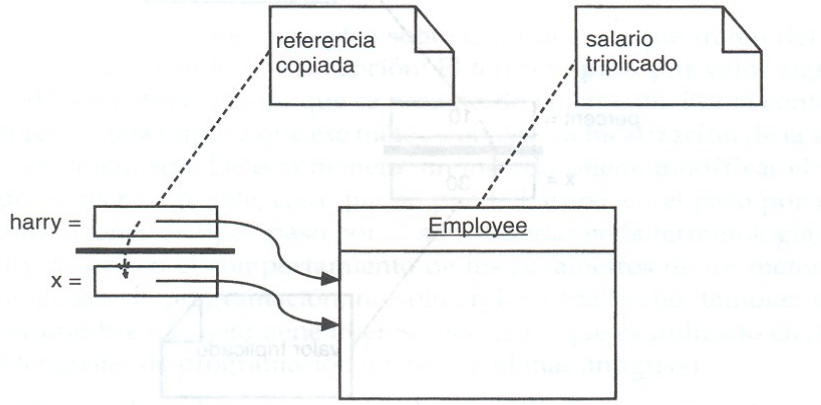
Si invocamos:

```
harry = new Empleado(. . .);
subirSalario(harry); Que sucede?
```

1. x se inicializa con una copia del valor de harry, (referencia a un objeto).

2. El método subirSalario se aplica a dicha referencia. El objeto Empleado al cual se refieren **tanto x como harry** se aumenta un 200%.

3. El método finaliza, y la variable parámetro x no existe más. Por supuesto, la variable objeto harry sigue apuntando al objeto cuyo salario ha sido triplicado



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

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();
      }
}
```

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);
      }
}
```

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
            }
      }

o bien
      //p2 sería el objeto creado en el ejemplo anterior
      public class constructorCopia {
            public static void main (String arg[]) {
                  Punto p2;              //del ejemplo anterior
                  p2 = new Punto(2, 4);

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

En esta asignación no se crea ningún objeto, solamente se hace que p2 y p3 apunten y referencien al mismo objeto.

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 explíc trollador tienen valores predeterminados, iguales a cero. Así, si acidad: el programa indicará los valores 0 seguido de 1000.

Vamos con un **caso concreto resuelto**.(puede resolverse de manera)

Paridad de números. Los números pueden ser pares o impar bles por 2 o no. Diseñe y codifique una clase que informe cuanto nos leído en una secuencia de números finalizada con número = 999

```
//  Contar Pares, Impares en una secuencia de números

public class Paridad {
```

```
run:
Un número, (fin 999)
12
Otro...
24
Otro...
33
Otro...
55
Otro...
66
Otro...
999
Números leidos
 pares:  3
 impares: 2
Demo terminado!!!
```

```
    private int contPares, contImpar;
    public  Paridad(){              // Constructor
        contPares = contImpar = 0;
    }

    public void demo(){
        int numero;
        System.out.println("Un numero, (fin 999)");
        while((numero = In.readInt()) != 999){
            if(esPar(numero))contPares++;
            else contImpar++;
            System.out.println("Otro...");
        }
        System.out.println(this);
    }

    private boolean esPar(int num){
        return num % 2 == 0;
    }

    public String toString(){
        String aux = "Numeros leidos \n";
        aux+= " pares:   "+contPares + " \n";
        aux+= " impares: "+contImpar + " \n";
        aux+= "Demo terminado !!!";
        return aux;
    }

    public static void main(String[] args) {
        Paridad par = new Paridad();
        par.demo();
    }
}
```

Interfaces

Para que interactúen dos objetos, deben "conocer" los diversos mensajes que acepta cada uno, esto es, los métodos que soporta cada objeto. Para forzar ese "conocimiento", el paradigma de diseño orientado a objetos pide que las clases especifiquen la interfaz de programación de aplicación (API, de application programming interface) o simplemente la interfaz que sus objetos presentan a otros objetos. En el método basado en TDA para estructuras de datos, se especifica una interfaz que define un TDA como una definición de tipo y un conjunto de métodos para este tipo, siendo de tipos especificados los argumentos para cada método. A su vez, esta especificación es aplicada por el compilador o sistema en tiempo de ejecución, o que requiere que los tipos de parámetros que se pasan a los métodos, coincidan en forma exacta al tipo especificado en la interfaz. A este requisito se le llama o tipiado fuerte. El tener que definir interfaces para después hacer que esas definiciones sean obligatorias por tipificación fuerte es una carga para el programador, pero está compensada por las ventajas que proporciona porque impone el principio de encapsulamiento, y con frecuencia atrapa errores de programación que de otra forma pasarían inadvertidos.

Implementación de interfaces

El elemento estructural principal de Java que impone una API es **la interfaz**. **Una interfaz es un conjunto de declaraciones de método**, **sin datos ni cuerpos**. Esto es, los métodos de una interfaz siempre están vacíos. Cuando una clase implementa una interfaz, debe implementar todos los métodos declarados en la interfaz. De esta forma, las interfaces imponen una especie de herencia, llamada especificación, donde se requiere que cada método heredado se especifique por completo.

Podríamos además considerar una segunda utilidad a clases que implementan interfaces. Imaginemos que el trabajo de análisis y programación lo hacen diferentes personas, el analista y el programador. Un analista (Minucioso) puede llegar a especificar como debe ser el comportamiento de una clase, método a método. Nombre del método, parámetros, tipo de retorno. El programador tiene la obligación de implementar la interfaz en la clase correspondiente, y codificar los cuerpos de los métodos. Java no le permite otra cosa.

Si aplicamos lo dicho al ejemplo anterior, debemos comenzar por especificar la interfaz para luego implementarla en nuestra clase Paridad.

```
package paridad01;
public interface Parid01 {
    public void demo();
    public boolean esPar(int num);
    public String toString();
}

package paridad01;
//  Contar Pares, Impares en una secuencia de numeros
public class Paridad01 implements Parid01{
    private int contPares, contImpar;
    public  Paridad01(){                  // Constructor
        contPares = contImpar = 0;
    }

    public void demo(){
        int numero;
        System.out.println("Un numero, (fin 999)");
        while((numero = In.readInt()) != 999){
            if(esPar(numero))contPares++;
            else contImpar++;
            System.out.println("Otro...");
        }
        System.out.println(this);
    }

    public boolean esPar(int num){return num % 2 == 0;}

    public String toString(){
        String aux = "Numeros leidos \n";
               aux+= " pares:   "+contPares + " \n";
               aux+= " impares: "+contImpar + " \n";
               aux+= "Demo terminado !!!";
               return aux;
    }
    public static void main(String[] args) {
        Paridad01 par = new Paridad01();
        par.demo();
    }
}
```

Compilando el ejemplo arriba hemos descubierto algunas cosas

1 – En la interfaz no se admiten métodos con nivel de acceso private o protected.
2 – En la interfaz los niveles de acceso son public o no especificados.
3 – En la clase el nivel de acceso de un método debe ser el de la interfaz; si allí no fue especificado puede ser cualquiera, excepto private.
4 – Los métodos private no se declaran en la interfaz y se codifican en la clase. (Son cosa interna de la clase, solo pueden ser llamados por otros métodos de ella).
5 – **Ningún método** descrito en la interfaz puede faltar en la clase.
6 – La clase puede tener más métodos que la interfaz.
7 – En la interfaz no se describen constructores.

Hagamos un ejemplo implementando interfaz con a la clase Carácter, que (la cátedra) viene tratando desde que comenzamos con objetos en C++.

```
package caracter;
public interface Caract01{    //    Interfaz
      boolean esLetra();      // El caracter es letra?
      boolean esLetMay();     //  Es letra mayúscula ?
      boolean esLetMin();     //  Es letra minuscula ?
      boolean esVocal();      //  Es vocal ?
      boolean esConso();      //  Es consonante?
      boolean esDigDec();     //  Es dígito decimal ?
      boolean esDigHex();     //  Es dígito hexadecimal ?
      boolean esSigPun();     //  Es signo de puntuación ?
      boolean lecCar() throws java.io.IOException;  //  Leemos caracter
      int getCar();           //  Retornamos el atributo car.
      void setCar(int cara);  //  Inicializamos el atributo car.
}

package caracter;
import java.io.IOException;
public class Caracter implements Caract01{     //    La clase en cuestión
      private int car;             // Parte interna de la clase, nuestro caracter

      public Caracter(){car=' ';}    // Constructor, inicializa car en ' '

      public Caracter(int cara){car = cara;} // Constructor, inicializa car s/cara

      public boolean esLetra(){
            boolean letra = false;
            if(esLetMay()) letra=true;
            if(esLetMin()) letra=true;
            return letra;      //  Retornemos lo que haya sido
      } //  esLetra()

      public boolean esLetMay(){    //  Es letra mayúscula ?
            boolean mayus = false;
            String mayu = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
            for(int i=0;i < mayu.length();i++)
                    if(mayu.charAt(i)==car) mayus = true;  //  Es mayúscula
            return mayus;
      }

      public boolean esLetMin(){    //  Es letra minuscula ?
            boolean minus = false;
            String minu = "abcdefghijklmnopqrstuvwxyz";
            for(int i=0;i < minu.length();i++)
                if(minu.charAt(i)==car) minus = true;  //  Es una letra minúscula
            return minus;
      }

      public boolean esVocal(){      //  Es vocal ?
            boolean vocal = false;
            String voca = "aeiouAEIOU";
            for(int i=0;i < voca.length();i++)
                if(voca.charAt(i) == car) vocal=true;  //  Es una vocal
            return vocal;                 //  Retornamos lo que sea ...
      }

      public boolean esConso(){     //  Es consonante
            boolean conso = false;
            if(esLetra() && !esVocal()) conso=true;
            return conso;
      }
```

```
      public boolean esDigDec(){    //  Es dígito decimal ?
            boolean digDec = false;
            String dig10 = "1234567890";
            for(int i=0;i < dig10.length();i++)
            if(dig10.charAt(i) == car) digDec=true;
            return digDec;
      }

      public boolean esDigHex(){    //  Es dígito hexadecimal ?
            boolean digHex = false;
            String dig16 = "1234567890ABCDEF";
            for(int i=0;i < dig16.length();i++)
                if(dig16.charAt(i)==car) digHex=true;
            return digHex;
      }

      public boolean esSigPun(){    //  Es signo de puntuación ?
            boolean sigPun = false;
            String punct = ".,;:";
            for(int i=0;i < punct.length();i++)
                if(punct.charAt(i) == car) sigPun=true;
            return sigPun;
      }

      public boolean lecCar() throws java.io.IOException{  //  Lectura
            car = System.in.read(); // leemos caracter
            if(car=='#') return false;
            return true;
      }

      public int getCar(){return car;}   //  Retornamos el atributo car.

      public void setCar(int cara){car = cara;} //  Inicializamos el atributo car.
}

package caracter;
public class Main {     //    Clase que usa objetos Caracter
    int contCar = 0, contLet = 0, contDig = 0, contCon = 0;
    public String toString(){
        String aux = "De los "+contCar+" Caracteres leidos\n";
               aux+= "tenemos   "+contLet+" letras\n";
               aux+= " de ellas "+contCon+" consonantes,\n";
               aux+= "   +      "+contDig+" digitos,\n";
        return aux;
    }

    public void demo()throws java.io.IOException{
        System.out.println("Tipee caracteres, fin '#'");
        Caracter car= new Caracter();
        while(car.lecCar()){    //  Mientras leamos caracteres ...
            contCar++;
            if(car.esLetra()) contLet++;
            if(car.esConso()) contCon++;
            if(car.esDigDec()) contDig++;
        }
    }

    public static void main(String[] args) throws java.io.IOException{
        Main main = new Main();
        main.demo();
        System.out.println(main);   //    Representación del objeto
        System.out.println("demo() terminado!!!");
```

```
    }
}
```

```
run:
Tipee caracteres, fin '#'
Hoy, 18/12/07, mucho calor#
De los 26 Caracteres leidos
tenemos   13 letras
 de ellas 8 consonantes,
   +      6 digitos,
demo() terminado!!!
```