Unidad II

Algoritmos con TIPO ABSTRACTO DE DATOS

Año 2011

> "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. Fritelli, Valerio

Indice (Unidad II – Algoritmos con Tipo Abstracto de Datos

Objetivos de la Unidad

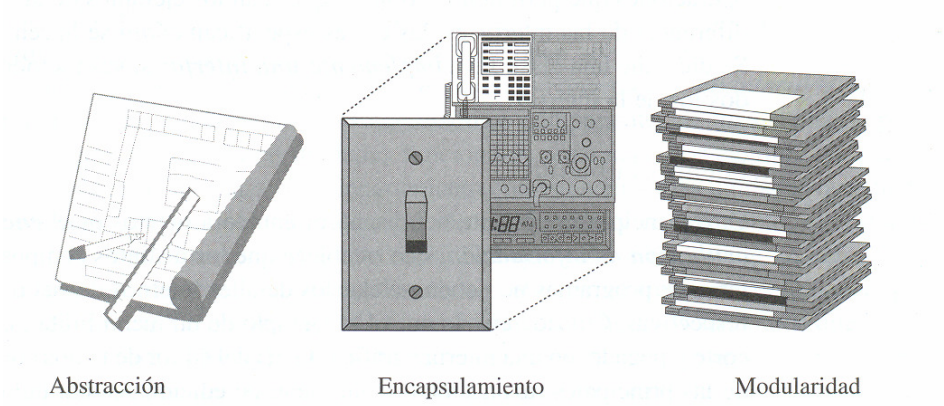
Desarrollar la capacidad de razonamiento y lógica necesarios para identificar problemas algorítmicos que usan tipo abstracto de datos y resolverlos. Esto implica: abordar problemas reales, analizarlos, definir la estrategia de su resolución, explicitar los algoritmos necesarios y codificarlos en un lenguaje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

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.

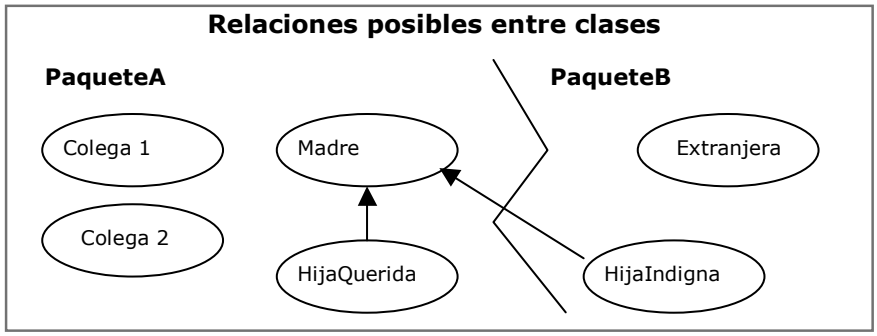
Hay que distinguir pues en la descripción de la clase dos partes:

- 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.

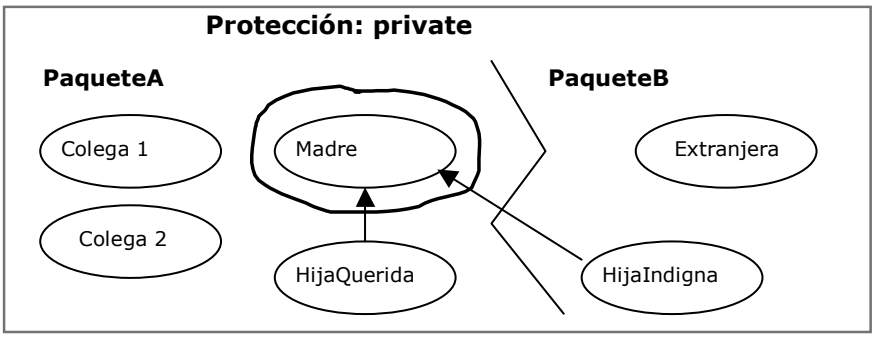
En los párrafos siguientes vamos a examinar sucesivamente los diferentes tipos de protección que Java ofrece a los atributos y a los métodos. Observemos ya desde ahora que hay cuatro tipos de protección posibles y que, en todos los casos, la protección va sintácticamente al principio de definición, como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

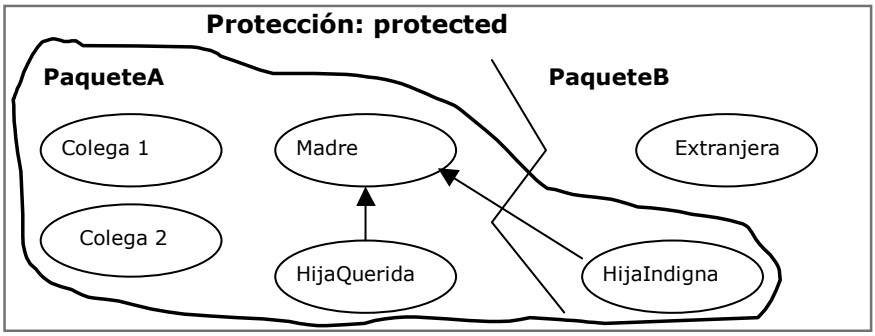
Observe que un objeto que haya surgido de la misma clase puede acceder a los atributos privados, como en el ejemplo siguiente:

```
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.

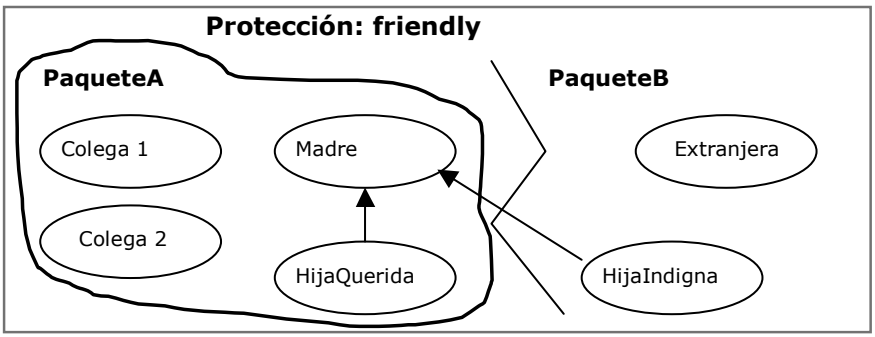
Protected

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.

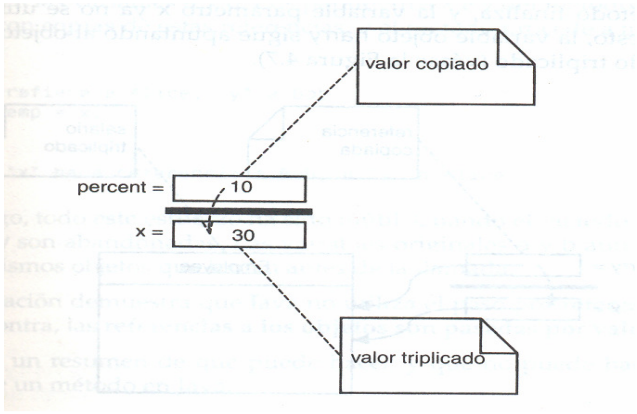
Vamos a profundizar algo más en esta situación. Supongamos un método que intente triplicar el valor de uno de sus parámetros:

```
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);}
```

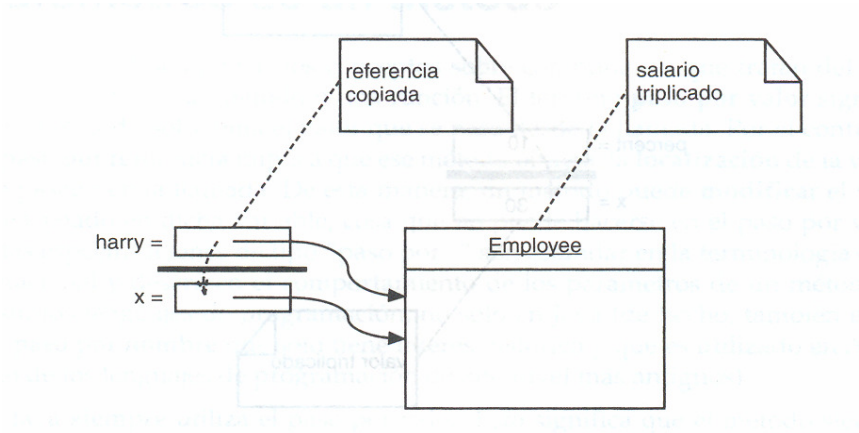
Si invocamos:

```
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 explícitamente por el desarrollador tienen valores predeterminados, iguales a cero. Así, si no se inicializa capacidad: el programa indicará los valores 0 seguido de 1000.

Vamos con un **caso concreto resuelto**.(puede resolverse de otra, quizás mejor manera)

Paridad de números. Los números pueden ser pares o impares, según sean divisibles por 2 o no. Diseñe y codifique una clase que informe cuantos pares e impares hemos leído en una secuencia de números finalizada con número = 999 (centinela)

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

public class Paridad {
```

```
    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();
    }
}
```

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

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!!!
```

Usando GUI Builder

Introducción

El entorno de programación para Java, NetBeans (Que es el que UD debería estar usando a estas alturas) permite construir interfaces graficas para entrada/salida de datos con mucha facilidad.

Este tema se verá con más profundidad en la asignatura Paradigmas de Programación, pero es conveniente que desde ya conozca su uso básico.

Utilizando este material UD aprenderá a:

- Usar el **GUI Builder** (Editor)
- Crear un contenedor de GUI (Grafic user Interface)
- Agregar componentes
- Redimensionar componentes
- Alinear componentes
- Ajustes de ancho
- Posesionar su comportamiento
- Editar sus propiedades

GUI Builder provee una paleta para que gráficamente seleccionemos los componentes que necesitamos y los incorporemos en la interface que estamos construyendo. En paralelo el editor automáticamente genera el código Java, al cual solo necesitaremos agregar, cuando sea necesario, que debemos hacer con el dato, o cual es la acción requerida.
Todo esto es mas fácil hacerlo que escribirlo.

Para comenzar, describiremos paso a paso un proyecto que recibe un texto de una línea, su nombre por ejemplo y lo devuelve invertido. Lo llamaremos EjemploIE01.

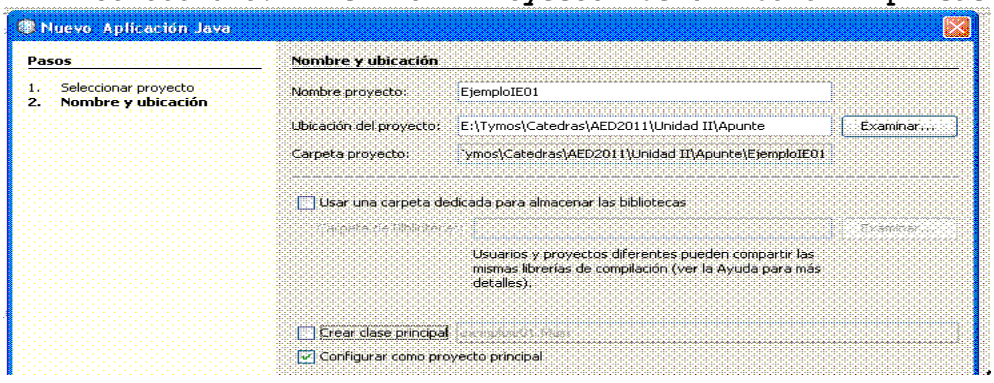
Para ello necesitamos:

- Un contenedor grafico. Sucede que los componentes deben ser distribuidos en una forma, o formulario. Hay diversos contenedores. Usaremos JFrame (J por Java, Frame es marco, bastidor, cuadro…)
- Un campo de texto para entrada de datos (JTextField)
- Un campo de texto para salida de datos (idem)
- Un botón de comando indicando que se debe proceder a la inversión (JButton)

Manos a la obra…

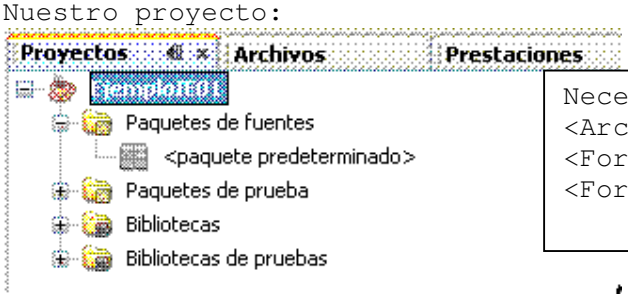
Creando el proyecto. Usando el NetBeans IDE,

- seleccione: **<Archivo> <Proyecto Nuevo> <Java> <Aplicación Java>**

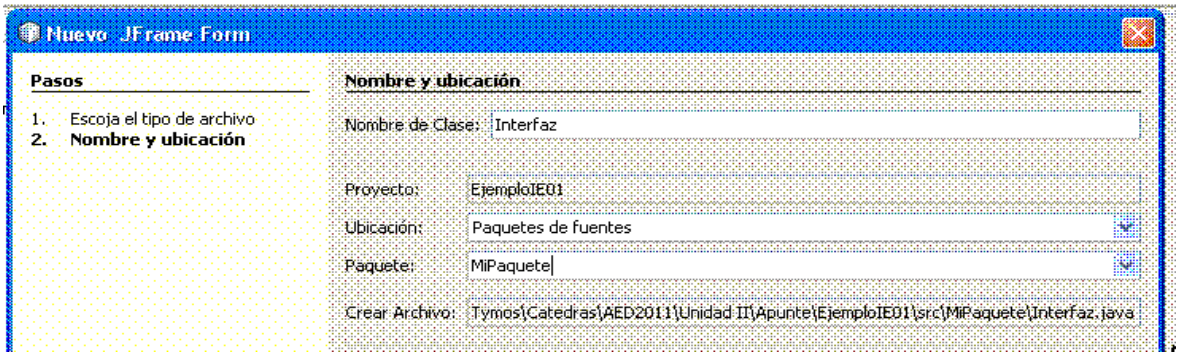


<Siguiente>

Nuestro proyecto:



Necesitamos el contenedor grafico <Archivo> <Archivo Nuevo> <Formularios de Interfaz Grafica> <Formulario Jframe> *<Siguiente>*



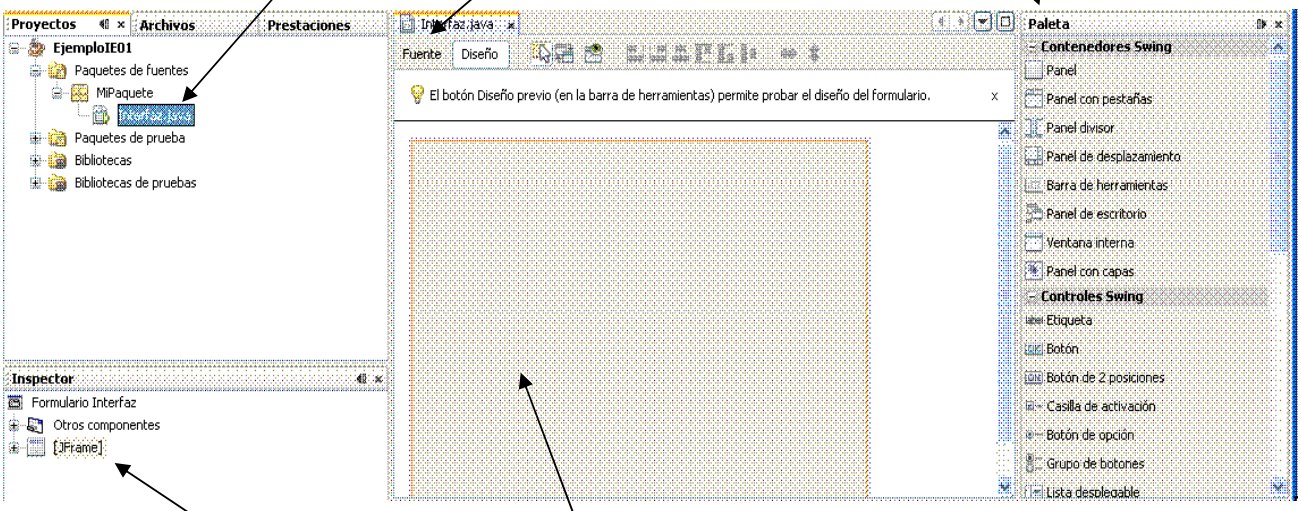
<Terminar>

Como quedó el proyecto **EjemploIG01**?

Interfaz es el archivo .java que hemos incorporado.

Podemos ver su codificación fuente o su diseño.

Aquí esta la paleta para seleccionar componentes



El Inspector nos permite ver como se va estructurando nuestra interfaz y además podemos gráficamente ejecutar acciones como cambiar nombres, borrar, etc

Esta es el área de diseño. Allí previsualizamos lo que estamos construyendo y gráficamente podemos modificar tamaños, posiciones, etc.

Si entramos por <Ventana> <Propiedades> debajo de la paleta aparecerá una ventanita informando las propiedades del componente corrientemente seleccionado.

Si seleccionamos Fuente, vemos el código generado por el Editor.

- **Fondo azul**, son áreas de codificación **no modificables** en este modo. Si algo es necesario modificar, ir al modo diseño, hacer los ajustes. Al salvar el proyecto se actualizan los fuentes.
- **Fondo blanco**, nuestra área de **código a completar**

```
package MiPaquete;

/**
 *
 * @author Usuario
 */
public class Interfaz extends javax.swing.JFrame {

    /** Creates new form Interfaz */
    public Interfaz() {
        initComponents();
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    Generated Code

    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new Interfaz().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify
    // End of variables declaration

}
```

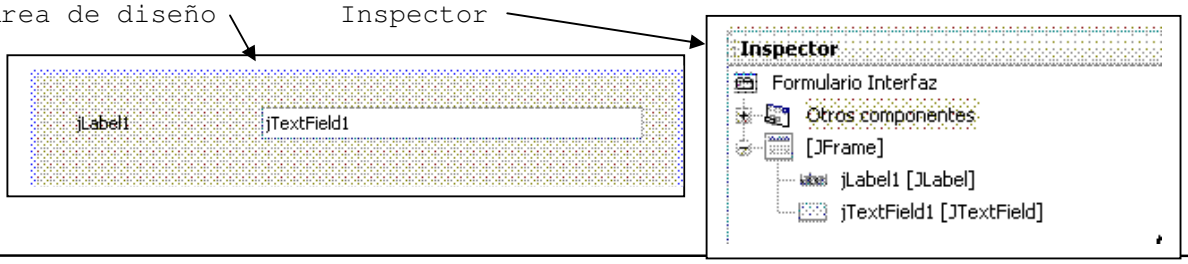
Hasta ahora solo tenemos 2 áreas no modificables.

- Generated Code, que tiene que ver con lo que el editor codifico respecto al contenedor Jframe.
- // Variables declaration, (Aun no tenemos variables declaradas).

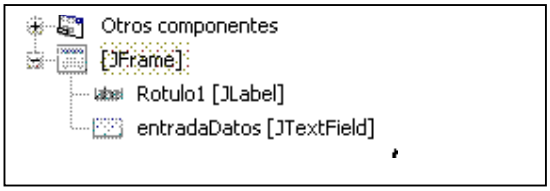
Agregar componentes.

- Comenzamos por una etiqueta (para orientar al usuario) y la entrada de datos.
  - En paleta, Controles Swing, seleccionamos **Etiqueta** y la arrastramos al área de diseño. Gráficamente la ampliamos de manera que contenga el texto "Texto de entrada"
  - En paleta, Controles Swing, seleccionamos Campo de Texto y lo arrastramos ubicándolo al lado de la etiqueta. Lo ampliamos a derecha. Tenemos:

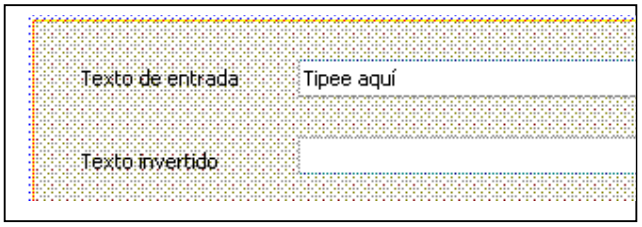
Area de diseño Inspector

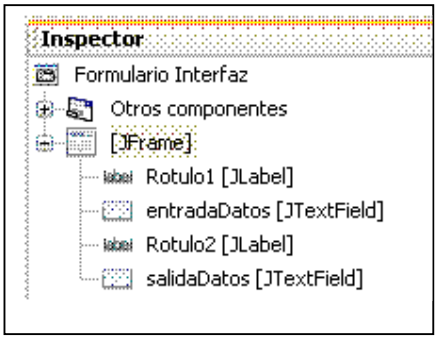


- Modificamos textos default. Lo hacemos de la misma manera que en Windows modificamos nombres.Posicionar cursor, doble clic…
- Modificamos nombres de variables. En Inspector, posicionar cursor, botón derecho. Nos debe quedar:



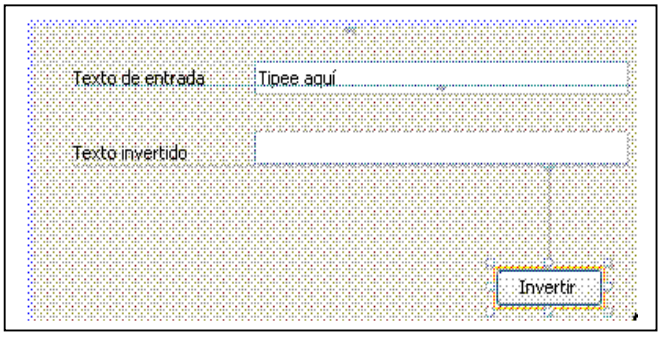
- Repetimos lo hecho para el texto de salida, nos debe quedar:

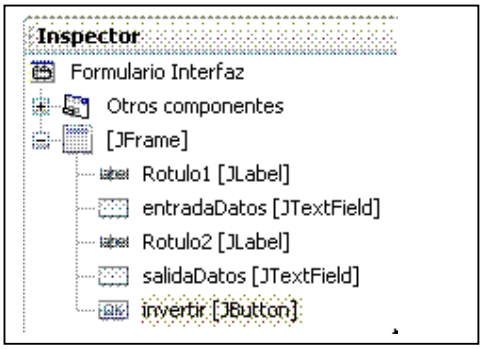




- Nos falta el botón de comando
  - Lo arrastramos, cambiamos su texto y el nombre
  - Debemos indicar al editor que al clic el botón debemos realizar la inversión del texto, o sea una acción, un evento de acción. Posicionamos cursor en Inspector, invertir, <Boton derecho>, <Eventos>, <Acción>, <AccionPerformed>,<Aceptar>

Veamos que tenemos:





En el código generado por el Editor aparecen mas cosas, por ejemplo:

- Las variables que estamos incorporando al contenedor Interfaz

```
    // Variables declaration - do not modify
    private javax.swing.JLabel Rotulo1;
    private javax.swing.JLabel Rotulo2;
    private javax.swing.JTextField entradaDatos;
    private javax.swing.JButton invertir;
    private javax.swing.JTextField salidaDatos;
    // End of variables declaration
```

- El método donde deberemos concretar la acción

```
    private void invertirActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
    }
```

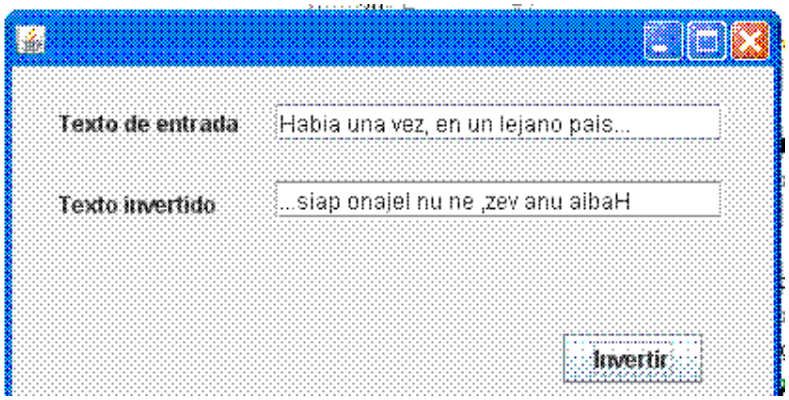
Justamente en este método es donde debemos hacer el trabajo. Y lo haremos aquí mismo, no llamaremos a otro método de otra clase especializada en inversiones de textos, (seria lo mas correcto)…

Definimos una variable String donde almacenamos lo leido, otra para el texto ya invertido y usamos los métodos correspondientes para leer y escribir en campos de texto. Esto lo vemos en <Ayuda> <Búsqueda de índice>, tipear JTextField, <Buscar>

Nos queda:

```
    private void invertirActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        String textoOrig, textoInv = "";
        textoOrig = entradaDatos.getText();
        for(int ind = textoOrig.length()-1; ind >= 0; ind--){
            textoInv+=textoOrig.charAt(ind);
        }
        salidaDatos.setText(textoInv);
    }
```

Probamos.



Perfecto!

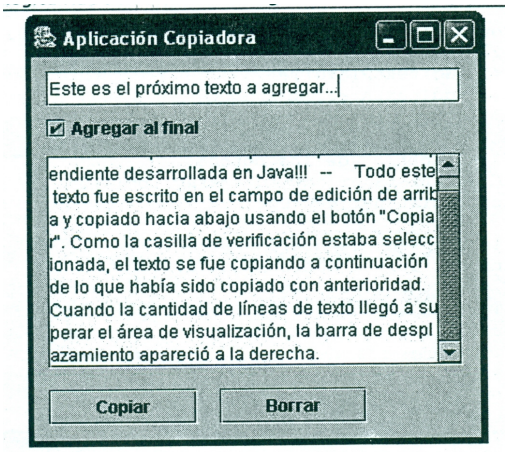
Vamos a un segundo ejemplo. Sea el proyecto EjemploIG02.

Esta aplicación permitirá realizar una copia de texto con las siguientes capacidades:

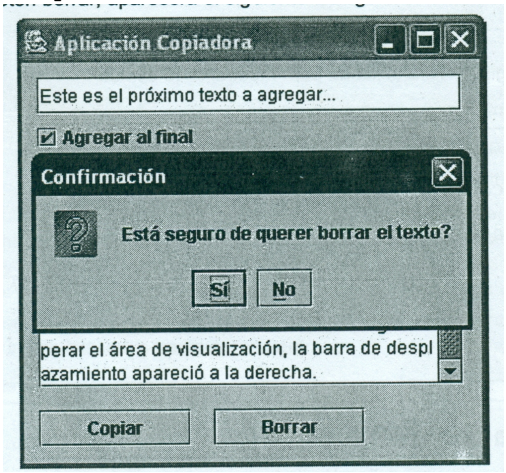
- Una casilla de verificación nos permite indicar si queremos agregar el nuevo texto al final del que se había ingresado anteriormente. Si la casilla no está seleccionada, el texto del área inferior es reemplazado por el nuevo texto introducido cada vez que se presiona el botón copiar.
- Se puede desplazar el área de texto hacia arriba y hacia abajo para poder ver todas las líneas de texto agregadas.
- Se pide confirmación cuando se presiona el botón borrar.

A continuación presentamos un par de imágenes mostrando que se pretende realizar.

resultado obtenido:

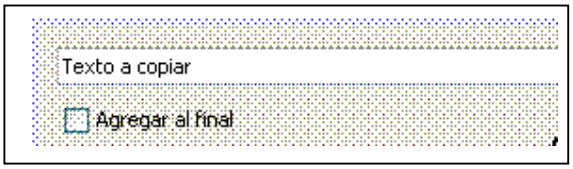


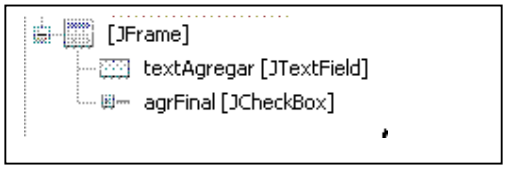
Si presionamos el botón Borrar



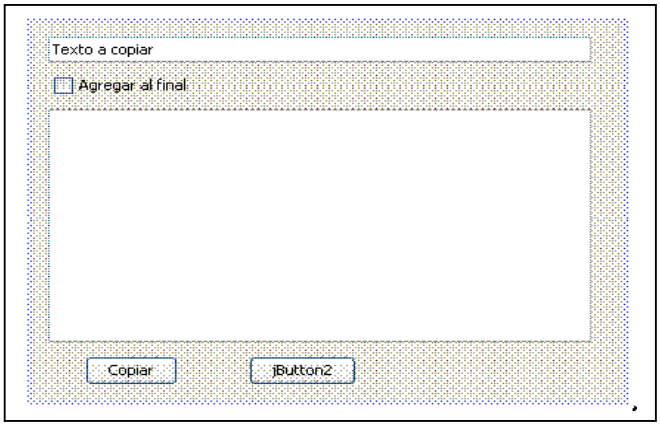
Vamos a la tarea
La primera parte es como en el EjemploIG01. Creamos el proyecto, le incorporamos una ventana, luego los primeros componentes:
Campo de texto: textAgregar
Casilla de activación: agrFinal

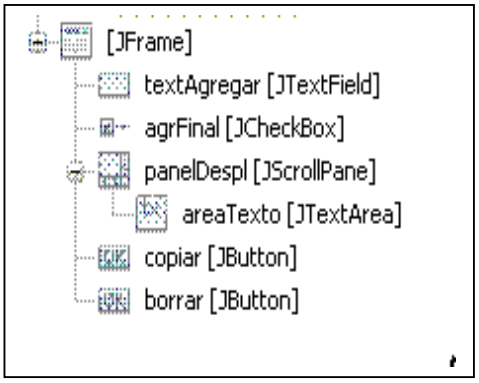
Una vez que modificamos los nombres en Inspector, esto debe verse:





Ahora debemos incorporar el área de texto. Y lo debemos hacer dentro de un panel de desplazamiento, de manera que podamos navegar el texto tanto en sentido vertical como horizontal. El panel de desplazamiento lo incluimos primero, dimensionamos y luego incluimos el área de texto dentro.
Luego incorporamos los botones copiar y borrar, debajo de todo.
Hecho esto, vamos a Inspector,
damos a los componentes los nombres adecuados y
incorporamos eventos de acción a los botones de copiar y borrar.
Echo esto, la interface y el Inspector deben verse:





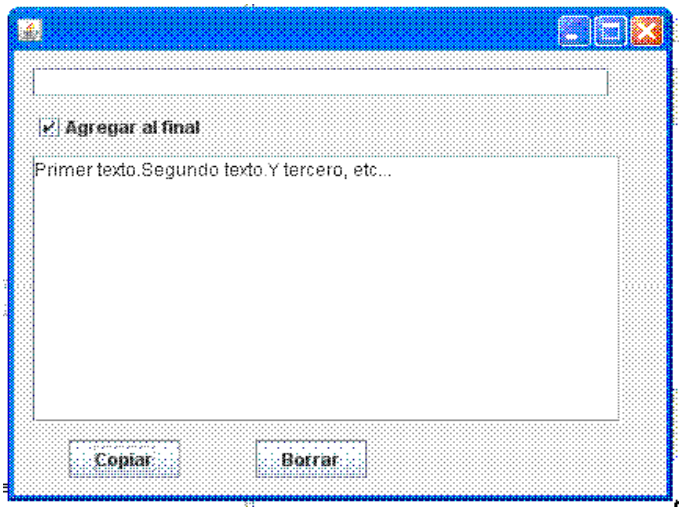
Intefaz.java se ve:

```
public class Interfaz extends javax.swing.JFrame {

    /** Creates new form Interfaz */
    public Interfaz() {
        initComponents();
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    Generated Code

    private void textAgregarActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
    }

    private void copiarActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
    }

    private void borrarActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
    }

    /**
    * @param args the command line arguments
    */
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new Interfaz().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify
    private javax.swing.JCheckBox agrFinal;
    private javax.swing.JTextArea areaTexto;
    private javax.swing.JButton borrar;
    private javax.swing.JButton copiar;
    private javax.swing.JScrollPane panelDespl;
    private javax.swing.JTextField textAgregar;
    // End of variables declaration

}
```

Recordemos que copiar recubre contenido de área o agrega al final, dependiendo de agrFinal.

```
private void copiarActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    String texto ="";
    if(agrFinal.isSelected())
        texto = areaTexto.getText();
     texto += textAgregar.getText();
     areaTexto.setText(texto);
     textAgregar.setText("");
 }
```

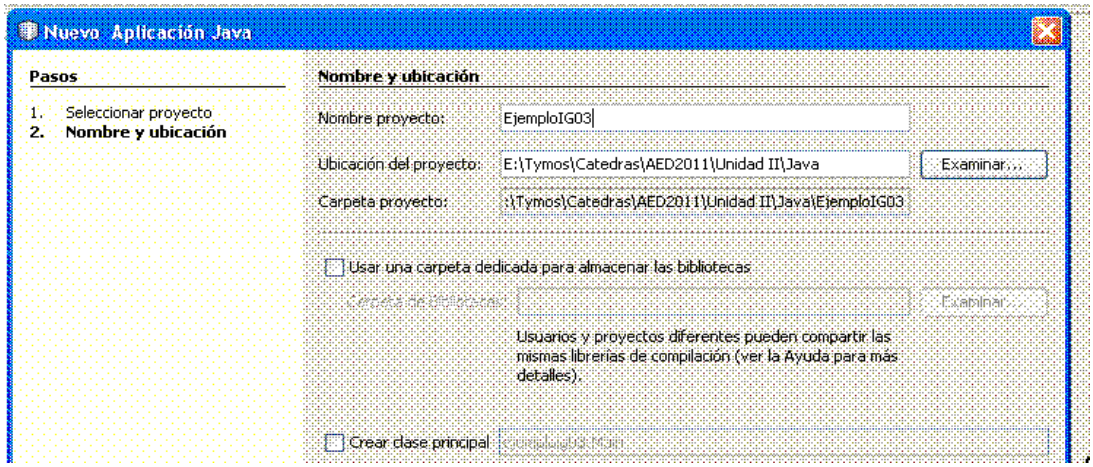
Una instancia del proyecto funcionando



Para terminar, hagamos algo un poco más elaborado, pero todavía bien sencillo. Queremos una interfaz con un panel para la entrada de datos y otro para la salida. En la entrada un campo de texto para identificarnos y una lista desplegable para optar o preguntar por algo, por ejemplo queremos saber que profe dicta AED en determinado curso, sus horarios. En el panel de salida un área de texto informándonos y la opción de salir.

Manos a la obra…
Creando el proyecto. Usando el NetBeans IDE,

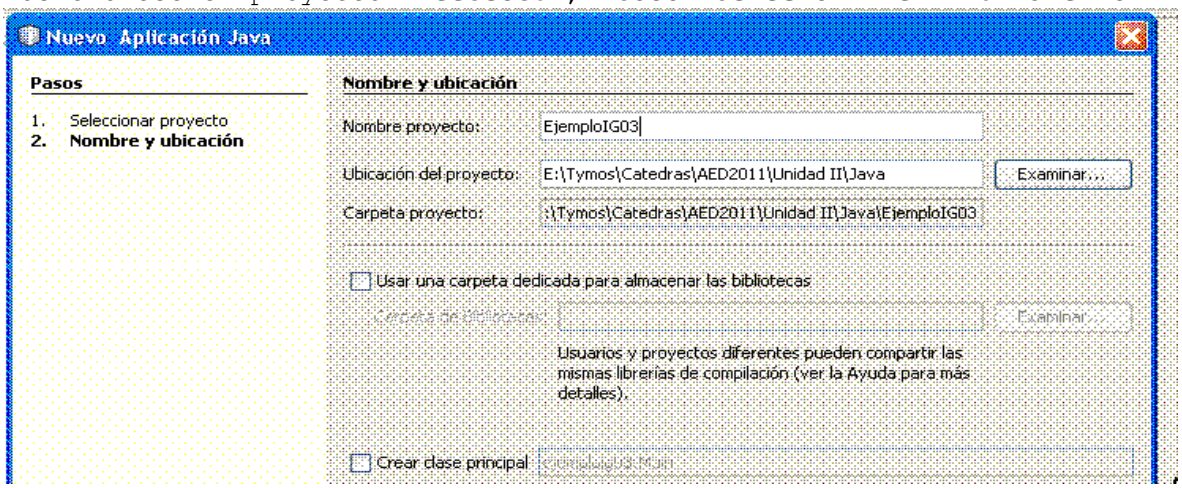
- seleccione: **<Archivo> <Proyecto Nuevo> <Java> <Aplicación Java>**



Creando el contenedor gráfico

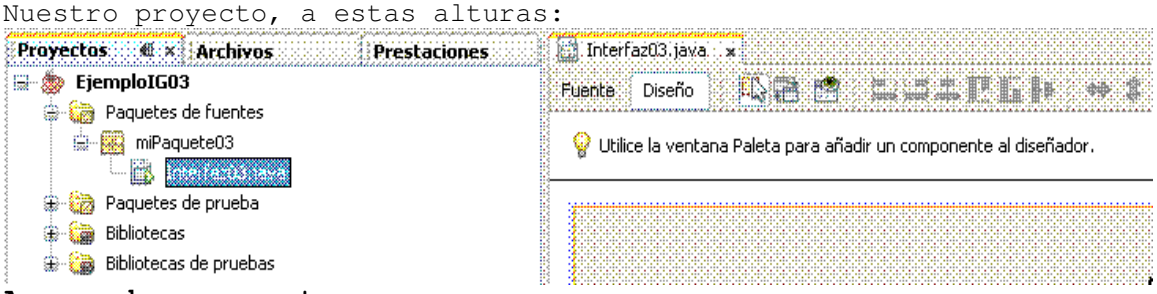
Nuestro contenedor será un JFrame.

Posiciónese en proyecto **TreeSetGUI, <boton derecho> <new> <Jframe Form>**



<Terminar>

Nuestro proyecto, a estas alturas:



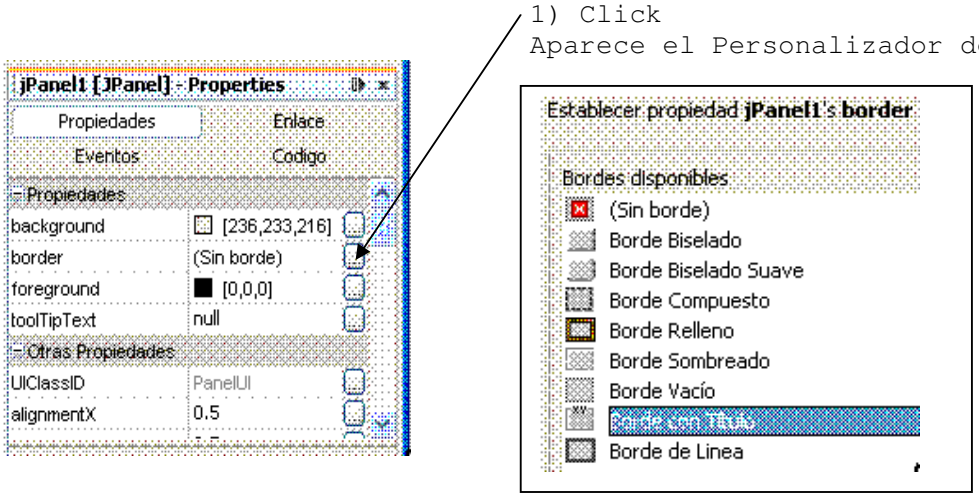
Agregando componentes.

- Tenemos la ventana Interfaz03, es un Jframe, contenedor de máximo nivel (Form's top level container).
- Decidimos usar un par de componentes **Jpanel** (Contenedores intermedios) para agrupar los componentes de nuestro UI (User Interfaz) con bordes titulados (tittled borders)
  - **Preguntas**
  - **Respuestas**

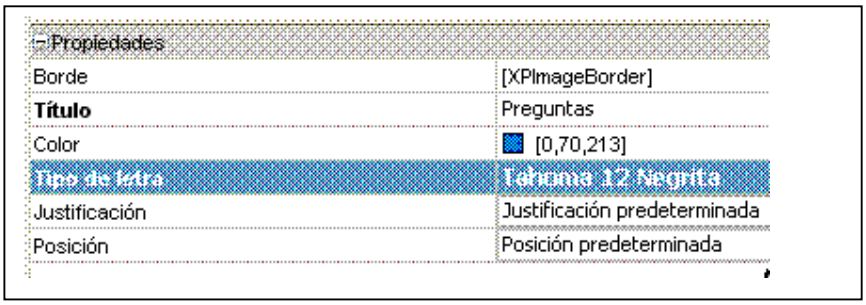
- En la ventana Palette, seleccionar Panel
- Desplace el cursor hacia la esquina superior izquierda del área de diseño. Cuando las líneas guía le muestran una posición adecuada, <click>.
- El Panel ha sido incluido, si bien no es visible. Solo cuando pasa el cursor sobre esa área, aparece en gris claro. Dimensiónelo de manera que ocupe la mitad superior de la interfaz. Este es el panel que llamaremos **Preguntas.**
- Repita los tres puntos anteriores para incorporar el panel Respuestas en la mitad inferior de Interfaz03. Este será el panel **Respuestas.**

Agregando los títulos de borde (title borders) al panel

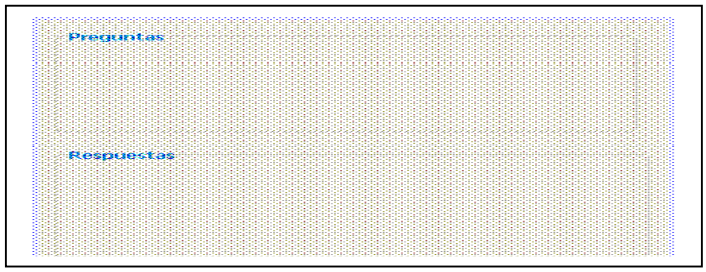
- <Click> sobre el panel Preguntas. Queda seleccionado y visible.
- En el menú tab del entorno, <Ventana>, <Propiedades>. <Click>
- Aparece la ventanita de propiedades de JPanel1 (Ya le modificaremos el nombre para Preguntas, un poquito de paciencia)



2) Aparece propiedades para
Borde con titulo. Optamos por Tahoma 12 Negrita.

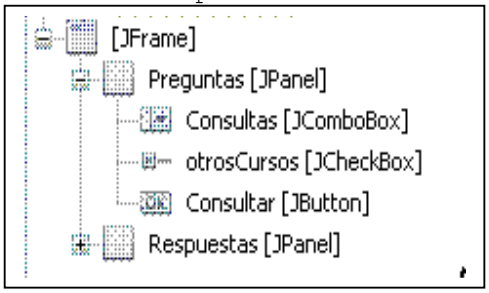


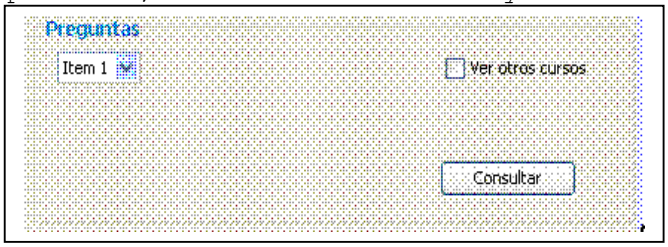
Hacemos lo mismo con los títulos de borde del panel Respuestas.
En el Inspector, cambiamos los nombres de los Jpanel's para Preguntas, Respuestas. Tenemos:



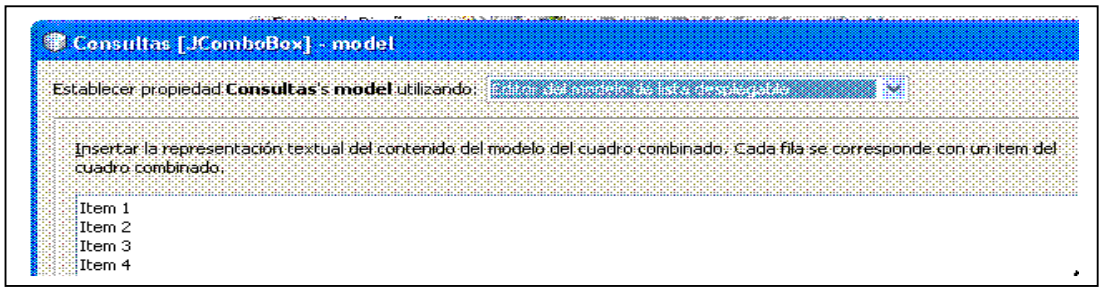
Todo bien. Trabajemos en el panel **Preguntas**.
Le debemos incorporar una lista desplegable para seleccionar el curso y el botón para ejecutar la consulta. Y pensando un poco, agregamos una casilla de verificación que si el usuario la selecciona, informaremos también en que otros cursos el profe del curso seleccionado dicta AED.

- Incorporamos los tres componentes, modificamos nombres y textos:

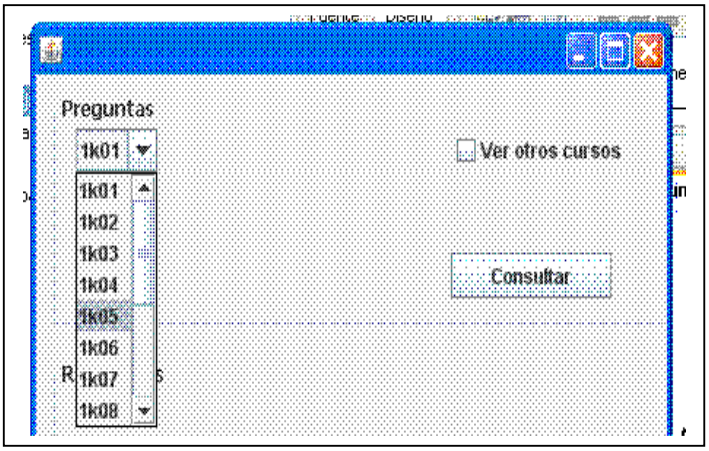




- Debemos incorporar los códigos de curso (1k01, 1k02,…) en la lista desplegable. Seleccionamos la lista desplegable y en propiedades seleccionamos model, clic sobre el botón del fondo, aparece

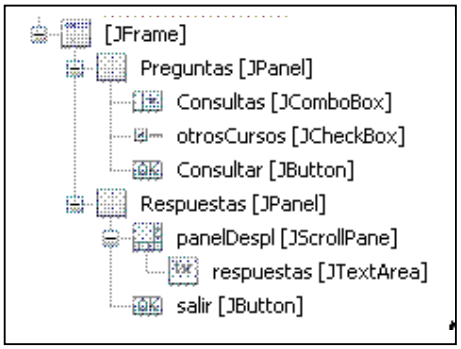


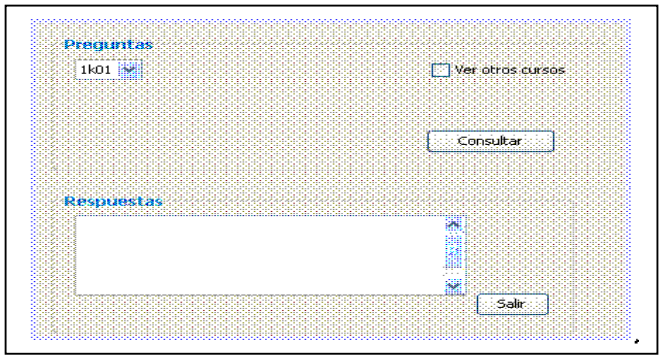
- Reemplazar Ítem 1, Ítem 2, por 1k01, 1k02, … <Aceptar>
- Verificamos si esto anda. Posicionamos cursor sobre Interfaz03, <Botón derecho>, <Ejecutar archivo>



Si clic sobre flechita se despliegan los cursos. OK. Seleccionamos el curso 1k05 y capturamos la imagen.

Listo el panel de **Preguntas**. **Respuestas** es más simple. Solo necesita un área de texto y un botón salir. Incorporamos, modificamos nombres…





Nuestra interfaz esta casi lista. Solo nos falta incorporar el tratamiento de las acciones (eventos) de Consultar y salir.

- En el Inspector posicionamos cursor sobre consultar, <Botón derecho>, <eventos>, <action>, <action performed>
- En el Inspector posicionamos cursor sobre salir, idem.

Si vemos la codificación de interfaz03, a estas alturas tenemos:

La parte de eventos:

```
    private void ConsultarActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
    }

    private void salirActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
    }
```

La parte de variables:

```
    // Variables declaration - do not modify
    private javax.swing.JButton Consultar;
    private javax.swing.JComboBox Consultas;
    private javax.swing.JPanel Preguntas;
    private javax.swing.JPanel Respuestas;
    private javax.swing.JCheckBox otrosCursos;
    private javax.swing.JScrollPane panelDespl;
    private javax.swing.JTextArea respuestas;
    private javax.swing.JButton salir;
    // End of variables declaration
```

Nuestra Interfaz03 esta lista

Pero para que trabaje, claro, debe existir la programación pertinente.
Esta programación es activada desde el método consultarActionPerformed().
Esto no lo genera ninguna paleta, es trabajo nuestro.
En primer lugar, el lugar adecuado de esta programación no es en Interfaz03.
Lo correcto es que definamos otra clase, por ejemplo Profes y allí la desarrollamos. Y mejor todavía si lo hacemos en dos clases:

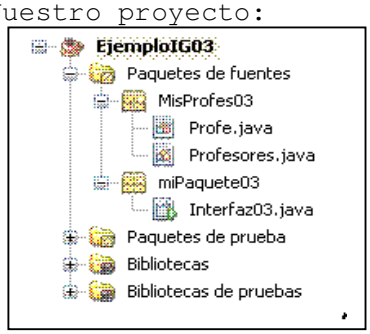
- Profe: Datos específicos de cada profe
- Profesores: Datos del conjunto de objetos profe, almacenados en un arreglo.

Vamos a la tarea.
Incorporemos estas clases en un paquete MisProfes03.
Archivo, Archivo nuevo, Java, Clase Java, <Siguiente>

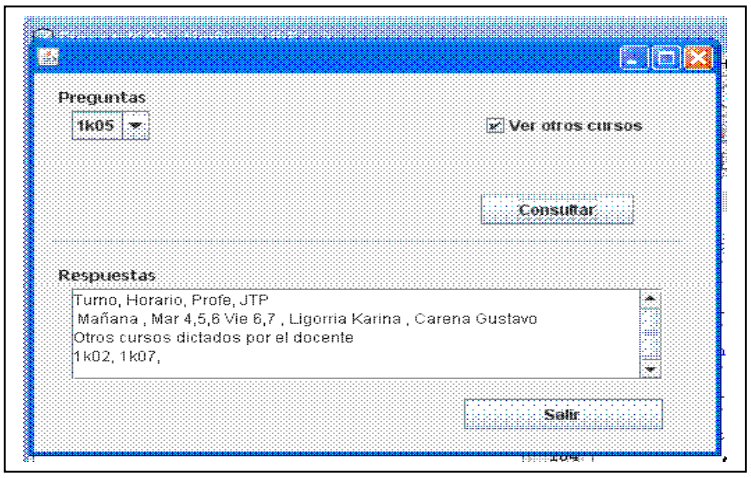
- Nombre de clase: **Profe**
- Paquete: **MisProfes03, <Terminar>**

Hacemos lo mismo con la clase **Profesores.**

Nuestro proyecto:



Nuestra interfaz funcionando



En la clase Interfaz03, el método activado por el botón consultar:

```
    private void consultarActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    //  Definimos objeto Profesores
    MisProfes03.Profesores profesores = new  MisProfes03.Profesores();

    //  Cual es el indice del curso seleccionado?
    int indice = consultas.getSelectedIndex();

    //  Cuales son los datos de ese curso?
    MisProfes03.Profe auxDatos = profesores.getDatos(indice);

    //   armamos la cadena a mostrar
    String auxStr = auxDatos.toString(); //  Representacion del objeto
    auxStr = "Turno, Horario, Profe, JTP\n"  //  Incorporamos título
           + auxStr.substring(6,auxStr.length());// Recortamos curso

    //  Nos preocupamos por si queremos saber otros cursos del profe
    if(otrosCursos.isSelected()){
        auxStr+=profesores.otrosCursos(indice);
    }
    respuestas.setText(auxStr);   //   Mostramos
}
```

Antes de irnos liberamos los recursos usados por el proyecto.

```
private void salirActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    dispose();
    System.exit(0);
}
```

Consola de Gráficos

1.) Presentación de la clase GraphicsConsole (Referencia: proyecto *Consola de Graficos*)

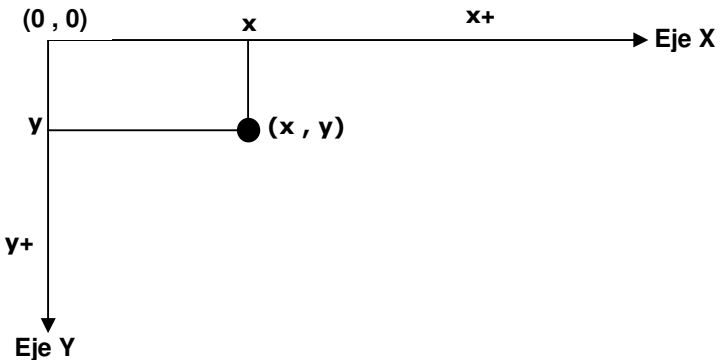
Es bueno recordar que Java provee una compleja estructura de clases y mecanismos de control para el planteo de interfaces de usuario basadas en ventanas, que permiten el desarrollo de programas y sistemas profesionales, cubriendo todos los requisitos que serían de esperar para la interacción con el usuario final. Estas clases y mecanismos se basan en conceptos que hacen imposible su tratamiento en un curso introductivo, por lo cual su estudio siempre se posterga hasta momentos en que los alumnos hayan adquirido esos conocimientos.

Sin embargo, también es cierto que un curso introductivo podría verse muy beneficiado si los estudiantes pudieran utilizar en el corto plazo recursos visuales de alto impacto: la experiencia nos ha mostrado que esos recursos incentivan la creatividad, y constituyen un fuerte impulso de motivación: al fin y al cabo, resulta obvio que un programa resulta mucho más atractivo de usar (y de desarrollar...) si el mismo cuenta con ventanas y gráficos que hagan más sencillo su funcionamiento y más accesibles sus recursos. Por este motivo, así como hemos provisto en su momento las clases *Consola* o *In* para facilitar lecturas por teclado, proveemos ahora una nueva clase *GraphicsConsole*, desarrollada por los profesores especialmente como complemento de estas fichas de clase.

La clase *GraphicsConsole* permite crear una ventana que simplemente contiene un panel central, capaz de desplegar gráficos diversos como líneas, rectángulos, círculos, óvalos y polígonos en diversos colores y estilos, así como también permite desplegar cadenas de caracteres en diversas fuentes y colores. La ventana desplegada contiene algunos elementos adicionales esenciales, como un menú superior con opciones directas para mostrar una ventana de información básica y para permitir la salida o cierre (ocultándola) de la ventana. También dispone de un botón en la parte inferior, que también permite ocultar la ventana. Mostramos en la página siguiente una ventana de la clase *GraphicsConsole* ya generada y con algunos gráficos en su interior.

Lo importante de aquí en más, es saber que la clase *GraphicsConsole* provee numerosos métodos ya listos para usar que permiten desplegar gráficos en el panel central de la consola gráfica, y que combinando las salidas producidas por esos métodos el programador podrá generar gráficas complejas en diversas situaciones. No obstante, es necesario que el estudiante comprenda algunas cuestiones básicas con relación al sistema de gestión de gráficos de Java: es cierto que la clase *GraphicsConsole* implementa todo lo necesario para usar este sistema de gestión de gráficos en forma transparente, pero para poder hacer un uso correcto y eficiente de esos elementos deben incorporarse algunos conceptos y prácticas previas.

Por lo pronto, el primer elemento que debe quedar claro y dominarse rápidamente es el esquema de coordenadas de pantalla que Java supone en la gestión de objetos gráficos. El sistema de coordenadas de Java permite localizar a cada punto de luz o *pixel* de la pantalla en forma directa. Por defecto, el *punto superior izquierdo* de un contenedor capaz de desplegar gráficos (como una ventana o un panel) es el origen de coordenadas de ese contenedor (o sea, el punto de coordenadas ***(0,0)*** del contenedor). Si la distancia de un punto al origen del sistema en el eje horizontal se designa como ***x***, entonces ***x*** será la *coordenada de columna* de ese punto. De forma similar, si la distancia del punto al origen en el eje vertical se designa como ***y***, entonces ***y*** será la *coordenada de fila* del punto:



Note que en este sistema de coordenadas, el *número de filas crece hacia abajo en la pantalla* (y no hacia arriba, como quizás el estudiante esperaría dada la costumbre de los ejes cartesianos que normalmente se usan en matemáticas). Esto quiere decir que una fila que se encuentre más cerca del borde superior de la pantalla o ventana, tendrá un número *menor* que otra fila que se encuentre más abajo. El crecimiento en el número de columnas sigue la misma idea que los gráficos cartesianos normales (es decir, los números de columna crecen hacia la derecha en la pantalla).

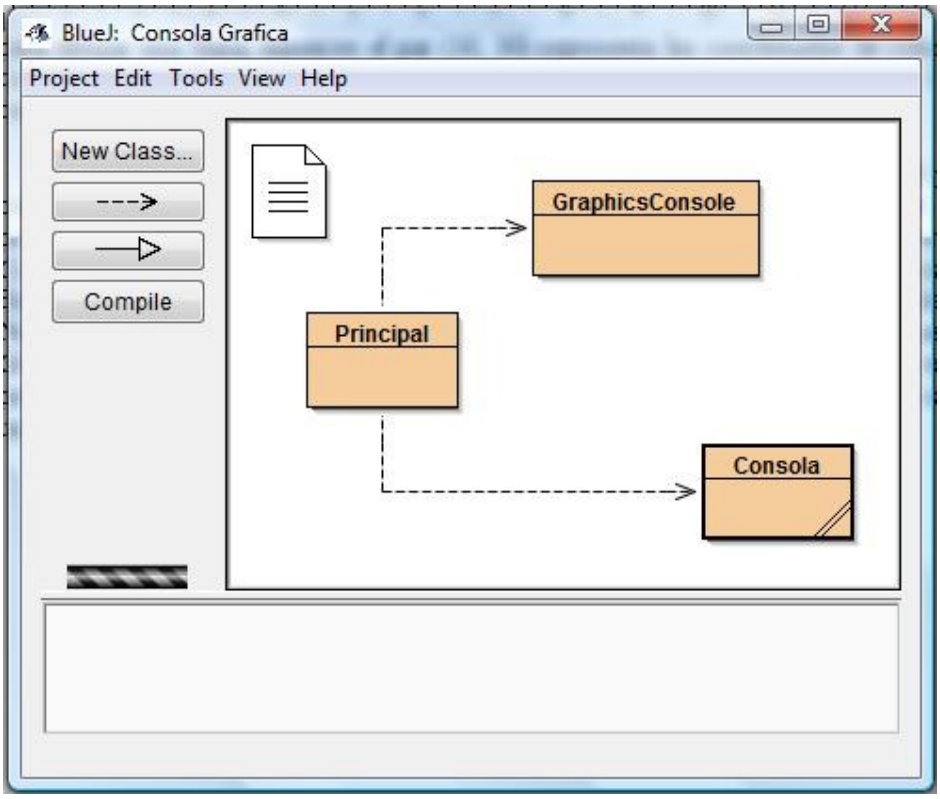
En casi todos los métodos de la clase *GraphicsConsole* se piden parámetros que representan coordenadas de pantalla del objeto a dibujar. Y en todos esos

métodos, se supone que las coordenadas enviadas siguen la convención anterior respecto del origen de coordenadas. Un detalle que no debe dejar de notar, es que además en todos los métodos que piden coordenadas se supone que la coordenada de columna va antes que la coordenada de fila. Así, si se invoca al método:

```
drawLine( 10, 30, 50, 100 )
```

para dibujar una línea, entonces el par (10, 30) representa las coordenadas de columna y fila (o sea, *columna = 10* y *fila = 30*) del punto de partida de la línea a dibujar, y el par (50, 100) representa las coordenadas de columna y fila del punto final de la misma linea.

Finalmente, indicamos ahora un breve resumen de lo que deberá hacer el programador para contar con esta clase y poder usarla en un proyecto. Lo primero, es que esa clase sea incluida en el proyecto. Si está usando BlueJ, puede hacerlo en la misma forma en que normalmente incluye la clase *Consola* o la clase *In* en otros proyectos (opción *Edit* del menú superior, y luego la opción "*Add class from file*"). En el proyecto *Consola de Graficos* que acompaña a esta ficha, la clase ya está incluida y lista para usar. Note que no hay ningún problema en que un mismo proyecto incluya y use tanto la clase *GraphicsConsole* como la clásica clase *Consola* o la clase *In* que permite la carga desde el teclado (de hecho, posiblemente deberá hacer uso de ambas de ahora en adelante, si no opta por hacer carga desde teclado usando *JOptionPane*):



El paso siguiente es crear un objeto de la clase *GraphicsConsole* para poder acceder a la ventana y desplegar gráficos en ella. Un detalle importante aquí, es que esa clase está diseñada de forma que ella misma controla que sólo pueda crearse un único objeto *GraphicsConsole* durante la ejecución de un mismo programa (de esta forma, por ejemplo, se evita que la misma ventana pueda aparecer abierta varias veces al ejecutar el programa). Por lo tanto, el programador no podrá usar el operador *new* para crear un objeto *GraphicsConsole*. En su lugar, deberá invocar al método estático *getInstance()* que la clase provee, y este método será el encargado de retornar una referencia al único objeto que la clase permitirá crear (de hecho, será la propia clase la que cree ese único objeto). La forma de hacerlo, puede verse en el método *main()* de la clase *Principal* del proyecto *Consola de Graficos*:

```
import java.awt.*;
public class Principal
{
    public static void main( String args[] )
    {
        // obtener el objeto GraphicsConsole...
        GraphicsConsole v = GraphicsConsole.getInstance();

        // resto del método main() aquí……
    }
}
```

Con esto, ya disponemos del objeto para manejar la consola gráfica, pero la ventana aún no es visible. Para hacer que la ventana se muestre, debemos invocar al método setVisible() contenido en la clase:

```
GraphicsConsole v = GraphicsConsole.getInstance();
v.setVisible( true );
```

El método *setVisible()* toma un parámetro *boolean*: si su valor es *true*, la ventana se hará visible. Y si el valor es *false*, la ventana se ocultará. Por defecto, la ventana se hará visible con ciertas dimensiones de ancho y alto, y aparecerá en el centro de la pantalla. Note que una vez visible, el usuario puede usar el mouse para "estirar" los bordes de la ventana, y agrandarla o achicarla si fuera necesario. También puede usar el mouse para clickear los botones de minimización, cierre, maximización, etc., que la ventana tiene en su barra de títulos y hacia la derecha. Sin embargo, el programador también puede invocar los métodos *setSize()* y *setLocation()* de la clase *GraphicsConsole*, para cambiar las dimensiones y la ubicación inicial de la ventana (aunque no hemos hecho esto último en el *main()* de la clase *Principal* en el proyecto *Consola Grafica*):

```
GraphicsConsole v = GraphicsConsole.getInstance();
v.setVisible( true );
v.setSize( 400, 300 );
v.setLocation( 0, 0 );
```

El método *setSize()* toma dos parámetros: el primero es el ancho deseado para la ventana (en pixels) y el segundo es la altura deseada (también en pixels). El método *setLocation()* pide también dos parámetros, que son las coordenadas de columna y fila del punto donde se desea *anclar* la ventana (o sea, el lugar donde será colocado el punto superior izquierdo de la ventana o *punto de anclaje* de la misma).

Una vez hecho todo lo anterior, la consola gráfica está visible y ubicada en el lugar deseado. Ahora el programa puede usar los métodos de la clase para generar gráficas en ella. Lo normal es comenzar seleccionando el color con el cual se desea generar la próxima figura, y luego invocar a un método que genere la figura deseada. Va un ejemplo de cómo hacer que aparezca un rectángulo de color azul en el área de dibujo de la ventana:

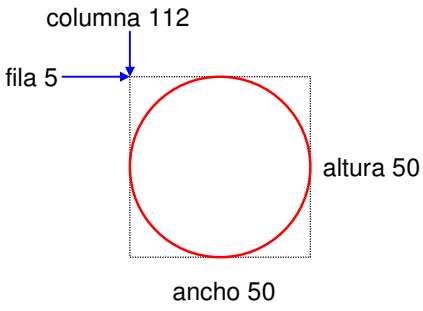
```
GraphicsConsole v = GraphicsConsole.getInstance();
v.setVisible( true );

v.setColor( Color.blue );
v.drawRect( 5, 5, 100, 50 );
```

El método *setColor()* de la clase *GraphicsConsole* fija el color de dibujo de allí en adelante: si se pide dibujar varias figuras, todas saldrán en azul a menos que se vuelva a invocar a *setColor()* para cambiar ese color. Y el método *drawRect()* dibuja el contorno de un rectángulo (sin pintarlo por dentro), de acuerdo a los parámetros enviados: los dos primeros parámetros (5 y 5 en nuestro caso) son las coordenadas de columna y fila del punto superior izquierdo del rectángulo a dibujar. Y los dos últimos, son respectivamente el ancho y el alto del rectángulo deseado, en pixels (ancho = 100 y alto = 50 en nuestro caso). Si ejecuta el *main()* de la clase *Principal* del proyecto, verá el efecto de estas instrucciones: un pequeño rectángulo azul ubicado cerca de la esquina superior izquierda del panel de dibujo de la consola.

En general, así deberá proceder de aquí en más. La siguiente secuencia dibuja el mismo rectángulo azul, y a continuación dibuja el contorno de un círculo de color rojo:

```
GraphicsConsole v = GraphicsConsole.getInstance();
v.setVisible( true );

v.setColor( Color.blue );
v.drawRect( 5, 5, 100, 50 );

v.setColor( Color.red );
v.drawOval( 112, 5, 50, 50 );
```

El método *drawOval()* también toma como parámetro las coordenadas de un rectángulo, pero en lugar de dibujar un rectángulo dibuja un círculo, tal que el mismo será el *máximo círculo que podría inscribirse dentro del rectángulo*. En nuestro caso, el rectángulo dentro del cual se inscribe el círculo tendría su punto superior izquierdo en las coordenadas (columna = 112, fila = 5) y tendría 50 pixels de ancho por 50 pixels de altura. Como el ancho y el alto son iguales, el rectángulo es en realidad un cuadrado. El círculo supone su centro en el centro del cuadrado, y su radio será el máximo posible para caber dentro del cuadrado. Obviamente, el cuadrado NO SE DIBUJA: sólo se dibuja el círculo, suponiendo los límites que impone el cuadrado:



Puede verse que el método *drawOval()* dibujará un *óvalo* en lugar de un círculo si el cuadrilátero es un rectángulo y no un cuadrado.

Los métodos de *GraphicsConsole* cuyo nombre comienza con *draw*, en general dibujan los contornos de alguna figura pero no pintan por dentro a esa figura. La clase provee otros métodos (cuyos nombres comienzan con *fill*) para dibujar las mismas figuras, pero rellenas con el color fijado por *setColor()*. La siguiente secuencia, dibuja un rectángulo pintado por dentro de color naranja:

```
v.setColor( Color.orange );
v.fillRect( 120, 210, 100, 50 );
```

Respecto del uso del color, dijimos que el método *setColor()* permite cambiar el color de dibujo de allí en adelante. Es útil saber que en Java los colores elementales están definidos como constantes de una clase llamada *Color*. Así, el color rojo se representa con la constante *Color.red* de esa clase, mientras que el azul se representa con *Color.blue*. Esas constantes pueden usarse tanto en mayúsculas como en minúsculas (es decir, el color rojo se designa indistintamente como *Color.red* o como *Color.RED*). Mostramos un extracto del archivo de ayuda de la clase *Color*, con los nombres de las principales constantes de colores:

Clase Color: constantes para los colores básicos	
static Color	**black** The color black.
static Color	**BLACK** The color black.
static Color	**blue** The color blue.
static Color	**BLUE** The color blue.
static Color	**cyan** The color cyan.
static Color	**CYAN** The color cyan.
static Color	**DARK_GRAY** The color dark gray.
static Color	**darkGray** The color dark gray.
static Color	**gray** The color gray.
static Color	**GRAY** The color gray.
static Color	**green** The color green.
static Color	**GREEN** The color green.

static Color	**LIGHT_GRAY** The color light gray.
static Color	**lightGray** The color light gray.
static Color	**magenta** The color magenta.
static Color	**MAGENTA** The color magenta.
static Color	**orange** The color orange.
static Color	**ORANGE** The color orange.
static Color	**pink** The color pink.
static Color	**PINK** The color pink.
static Color	**red** The color red.
static Color	**RED** The color red.
static Color	**white** The color white.
static Color	**WHITE** The color white.
static Color	**yellow** The color yellow.
static Color	**YELLOW** The color yellow.

La clase *GraphicsConsole*, como dijimos, provee numerosos métodos para gestionar gráficos complejos. Invitamos a cada estudiante a analizar estos métodos y experimentar con ellos para producir las figuras y los resultados que pudiera requerir. Mostramos aquí un extracto de los principales métodos incluidos en la clase:

GraphicsConsole: resumen de métodos	
void	**clearRect**(int x, int y, int width, int height) Borra el contenido del área rectangular especificada por los parámetros.
void	**copyArea**(int x, int y, int width, int height, int dx, int dy) Copia el contenido del área en el rectángulo indicado, hacia otra área desplazada en dx y dy pixels.
void	**draw3Drect**(int x, int y, int width, int height, boolean raised) Dibuja el contorno de un rectángulo con bordes trabajados para simular un efecto de sobre relieve (raised = true) o bajo relieve (raised = false).
void	**drawArc**(int x, int y, int width, int height, int startAngle, int arcAngle) Dibuja el contorno de un arco circular o elíptico, dentro de los límites del rectángulo indicado.

void	**drawLine**(int x1, int y1, int x2, int y2) Dibuja una línea, comenzando desde el punto (x1, y1) y terminando en el punto (x2, y2).
void	**drawOval**(int x, int y, int width, int height) Dibuja el contorno de un círculo o de una elipse, inscripta en el rectángulo dado.
void	**drawPolygon**(int[ ] xPoints, int[ ] yPoints, int n) Dibuja un polígono cerrado con (n) puntos a modo de vértices.
void	**drawRect**(int x, int y, int width, int height) Dibuja el contorno de un rectángulo.
void	**drawRoundRect**(int x, int y, int width, int height, int arcWidth, int arcHeight) Dibuja el contorno de un rectángulo con bordes redondeados.
void	**drawString**(String s, int x, int y) Dibuja (muestra...) el contenido de la cadena (s) usando la fuente y el color actualmente activados en la ventana.
void	**fill3DRect**(int x, int y, int width, int height, boolean raised) Dibuja y rellena (pinta por dentro) un rectángulo con bordes trabajados para simular un efecto de sobre relieve (raised = true) o bajo relieve (raised = false).
void	**fillArc**(int x, int y, int width, int height, int startAngle, int arcAngle) Dibuja y rellena un arco circular o elíptico, dentro de los límites del rectángulo indicado.
void	**fillOval**(int x, int y, int width, int height) Dibuja un círculo o una elipse, pintada por dentro e inscripta en el rectángulo dado.
void	**fillPolygon**(int[ ] xPoints, int[ ] yPoints, int n) Dibuja un polígono cerrado y pintado por dentro con (n) puntos a modo de vértices.
void	**fillRect**(int x, int y, int width, int height) Dibuja un rectángulo pintado por dentro.
void	**fillRoundRect**(int x, int y, int width, int height, int arcWidth, int arcHeight) Dibuja un rectángulo pintado por dentro, con los bordes redondeados.
java.awt.Color	**getColor**() Retorna el color actualmente usado en esta ventana.
java.awt.Font	**getFont**() Retorna la fuente de caracteres actualmente usada en esta ventana.
java.awt.FontMetrics	**getFontMetrics**() Retorna las métricas de la fuente de caracteres actualmente usada en esta ventana.
static GraphicsConsole	**getInstance**() Obtiene el único objeto existente de la clase GraphicsConsole.
void	**setColor**(java.awt.Color c) Cambia el color de dibujo.
void	**setFont**(java.awt.Font f) Cambia el tipo de fuente usado para mostrar texto en esta ventana.
void	**translate**(int x, int y)

	Cambia origen de coordenadas del contexto gráfico usado, de forma que el nuevo origen coincidirá con (x, y).

A modo de ejemplo complementario, se incluye también un proyecto (que acompaña a esta ficha) llamado *Menu* mediante el cual se muestra la forma de usar la consola de gráficos en un contexto controlado por menú de opciones en consola estándar: desde el menú se pide seleccionar entre dibujar un cuadrado o un círculo, y la figura elegida se dibuja en la consola de gráficos. Lo primero que se puede observar es que cada método encargado de responder a las selecciones del ususario (*mostrarCuadrado()* y *mostrarCirculo()*) obtienen una referencia al objeto *GraphicsConsole* mediante una invocación al método *getInstance()*, como ya se explicó. Como el objeto GraphicsConsole es uno y único, se garantiza con esto que cada método que necesite acceder a la consola de gráficos lo haga en la misma única consola que existe para toda la aplicación.

Por otro lado, si cada opción del menú requiere un nuevo dibujo, eso implica tener que "limpiar" primero el contenido de la consola de gráficos para evitar que la nueva figura pedida se dibuje superpuesta con la última que se haya trazado. A modo de ejemplo, el método *mostrarCuadrado()* de la clase *Principal* del proyecto *Menu* muestra como hacer esa limpieza previa al dibujo:

```
public static void mostrarCuadrado()
{
    GraphicsConsole v = GraphicsConsole.getInstance();
    if( v.isShowing() == false ) v.setVisible( true );

    // borrar el sector donde quiero dibujar...
    // acordarse que el fondo de la consola era blanco...
    v.setColor( Color.white );
    v.clearRect( 100, 100, 100, 100 );

    // cambiar el color y dibujar...
    v.setColor( Color.red );
    v.fillRect( 100, 100, 100, 100 );
}
```

Para hacer el borrado o limpieza de una zona de la consola de gráficos, se usa el método *clearRect()* de la clase *GraphicsConsole*. Este método toma las coordenadas de un rectángulo, y simplemente dibuja ese rectángulo (relleno) usando el *color que haya sido definido con setColor()*. El truco consiste en recordar que antes de invocar a *clearRect()* se invoque primero a *setColor()* para fijar como color de dibujo al *mismo color que tenga el fondo* de la consola de gráficos. El resultado es que cualquier gráfica que hubiese habido en esa área, simplemente desaparecerá.

Note que el mismo método *mostrarCuadrado()* enseña la forma de verificar si la consola de gráficos está ya visible o no, usando el método *isShowing()*: si retorna *true* la ventana está visible, y si no lo está retorna *false*. En el ejemplo, si la ventana no está visible simplemente se invoca al método *setVisible()* en la forma ya explicada en la primera parte de esta ficha, pasándole el valor *true* como parámetro.

2.) La clase *Font* de Java (Referencia: proyecto *Font*)

Nuestra clase *GraphicsConsole* permite desplegar cadenas de caracteres en forma sencilla y muy práctica, mediante el método *drawString()*. Al hacerlo, se puede indicar en qué lugar de la ventana mostrar el texto, y usando previamente *setColor()* se puede configurar el color del texto a mostrar. Sin embargo, no sólo son posibles esas acciones sino que además se puede configurar la consola de gráficos de forma que al desplegar cadenas, se cambie también la fuente de letra a usar.

Java dispone de una clase predefinida llamada *Font*, que permite representar fuentes de letras para el despliegue de texto en entornos visuales y gráficos. Así como la clase *Color* permite representar un color, la clase *Font* permite representar una fuente de texto. Y así como el método *setColor()* permite establecer el color a usar de allí en adelante en los gráficos de la consola de gráficos, el método *setFont()* permite fijar la fuente a usar de allí en adelante para desplegar texto. El siguiente ejemplo muestra una secuencia de instrucciones que despliega en la consola de gráficos el mensaje *"Hola Mundo"*, en color rojo, con una fuente de tipo **Times Roman**, en estilo negrita (**bold**) y de tamaño 12. Básicamente, el texto aparecerá a partir del pixel cuyas coordenadas son (50, 50):

```
GraphicsConsole gc = GraphicsConsole.getInstance();
gc.setVisible( true );

Font f = new Font( "TimesRoman", Font.BOLD, 12 );
gc.setFont( f );
gc.setColor( Color.red );

gc.drawString( "Hola Mundo", 50, 50 );
```

Como se ve, la idea es crear primero un objeto de la clase *Font* y luego pasar ese objeto al método *setFont()* de la clase *GraphicsConsole*. Todas las llamadas al método *drawString()* de la consola de gráficos usarán esa fuente de allí en adelante, y hasta que se vuelva a cambiar la fuente invocando nuevamente a *setFont()*.

La clase *Font* dispone de varios constructores, pero quizás el más usado sea el que pide tres parámetros:

```
public Font ( String name, int style, int size )
```

El primer parámetro (aquí llamado *name*) es una cadena que identifica al nombre de la fuente que se quiere usar. Existen cinco nombres de fuentes que Java incorpora en forma predefinida a través de constantes de la clase *Font*. Esas constantes son: Font.DIALOG, Font.DIALOG_INPUT, Font.MONOSPACED, Font.SERIF y Font.SANS_SERIF. Sin embargo, el programador puede usar cadenas con nombres de fuentes típicas, como "TimesRoman", "Helvetica", "Arial", "Courier", etc.

El segundo parámetro (aquí llamado *style*) es un número entero que indica el estilo a usar en la representación de la fuente. Los estilos típicos son conocidos, y también vienen representados con constantes predefinidas de la clase *Font*, que son las siguientes: Font.BOLD (estilo **negrita**), Font.ITALIC (estilo *cursiva*) y Font.PLAIN (estilo normal). Los estilos bold e italic pueden combinarse mediante el *operador suma*, para formar el estilo ***negrita cursiva***:

```
public Font ( "TimesRoman", Font.BOLD + Font.ITALIC, 14 );
```

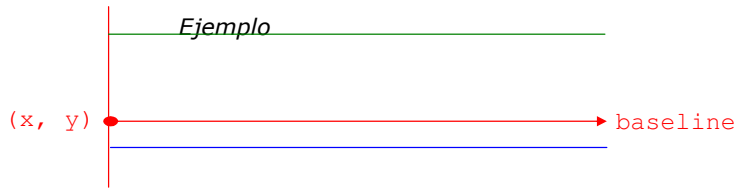
El tercer parámetro del constructor (aquí llamado *size*) configura el tamaño en puntos de la fuente a crear: mientras menor sea el número, menor será el tamaño de la fuente. Como ejemplos rápidos, puede recordar que los tamaños típicos de fuente al editar texto con un programa de edición como Word, son 11, 12 y 14 (este texto está escrito con una fuente de 11 puntos, por ejemplo).

Además de la clase *Font*, Java provee otra clase llamada *FontMetrics*, mediante la cual se pueden obtener las propiedades de una fuente dada. El método *getFontMetrics()* de la clase *GraphicsConsole* retorna un objeto *FontMetrics* para acceder a las propiedades de la fuente actualmente usada en la ventana:

```
FontMetrics fm = gc.getFontMetrics();
```

Cuando una cadena se muestra en un contexto gráfico, hay varias líneas guía en las que se basa el despliegue de la cadena. La línea donde todo el texto se apoya se llama en general la *línea base* del texto (o *baseline*). Las coordenadas (*x*, *y*) que se envían como parámetro al método *drawString()* para desplegar una cadena, constituyen las coordenadas del punto origen de la línea base (el punto donde comenzará la *baseline* que servirá de apoyo a toda la cadena).

Algunos caracteres (como la *j*, la *g* o la *p*, por ejemplo) pasan hacia abajo de la línea base y se terminan apoyando en una segunda línea. Se llama *descenso* (o *descent*) de la fuente a la distancia que hay entre la línea base y la *segunda* línea de apoyo de estos caracteres (en el gráfico que se ve más abajo, el *descenso* es la distancia entre las líneas roja y azul). Por otra parte, cada fuente tiene una *elevación máxima* sobre la línea base, y se llama *ascenso* (o *ascent*) de la fuente a la distancia que existe entre la línea base y la línea de elevación máxima de esa fuente (en el gráfico, el *ascenso* es la distancia entre las líneas roja y verde):



La clase *FontMetrics* provee métodos para obtener los valores de esas distancias entre líneas guía: los métodos *getAscent()* y *getDescent()* retornan respectivamente los valores de *ascenso* y *descenso* de una fuente dada. Otros métodos de la clase permiten obtener otras medidas, tales como los límites de una cadena en esa fuente (*getStringBounds()*), la distancia entre la línea de descenso de una cadena y la línea de elevación de la siguiente línea (*getLeading()*), etc.

En general, la clase *FontMetrics* es muy útil cuando el programador necesita hacer un control fino y especializado respecto de la forma en que una cadena se muestra en un contexto gráfico, aunque por el momento no será necesario profundizar su uso.

3.) La clase *Color* de Java (Referencia: proyecto Font)

Hemos visto que la clase *Color* provee constantes ya declaradas y listas para usar, que representan a los colores principales y más básicos. Sin embargo, al igual que con la clase *Font*, se pueden crear objetos de la clase *Color* que representen *mezclas* de colores, y de esta forma el programador puede definir sus propios colores. Sólo debemos tener en cuenta un detalle esencial: los distintos colores surgen por combinación de tres colores básicos o *primarios*, que en cuanto al monitor de una computadora son el rojo, el verde y el azul (*red*, *green* y *blue* en inglés). Así, una mezcla de ciertas cantidades de *red*, *green* y *blue* se conoce en general como una combinación *RGB* por las iniciales de los tres colores en inglés.

Hay distintas formas de expresar numéricamente una combinación *RGB* para representar un color cualquiera. Una forma común (y muy usada por los programadores Java al crear objetos de la clase *Color*) consiste en expresar cada cantidad de cada color primario como un número entre 0 y 255. Así, una combinación RGB de la forma (0, 200, 50) estaría refiriendo a un color resultante de mezclar 0 partes de rojo, 200 partes de verde y 50 partes de azul. El resultado es este color de tendencia al verde.

La clase *Color* provee numerosos constructores que permiten crear objetos para representar colores por mezcla *RGB*. Quizás el más usado de esos constructores sea el que toma tres parámetros de tipo entero, para expresar en forma simple la combinación *RGB* que se quiere usar:

```
public Color ( int r, int g, int b )
```

Cada uno de los tres parámetros es un número entre 0 y 255, que indica respectivamente la cantidad de rojo, verde y azul que se se quiere mezclar para formar el nuevo color. Algunos ejemplos siguen a continuación:

```
Color cr = new Color( 255, 0, 0);      // el color rojo... se entiende??
Color cv = new Color( 0, 255, 0);      // el color verde...
Color ca = new Color( 0, 0, 255);      // el color azul...
Color cg = new Color( 180, 180, 180);  // un color gris claro...
```

Y obviamente, podemos usar cualquier color creado de esta forma como color base de trabajo en nuestra clase *GraphicsConsole*. El siguiente esquema, muestra una cadena en color gris claro en la consola de gráficos:

```
Color c = new Color( 180, 180, 180 );
gc.setColor( c );
gc.drawString( "Ejemplo", 10, 320 );
```

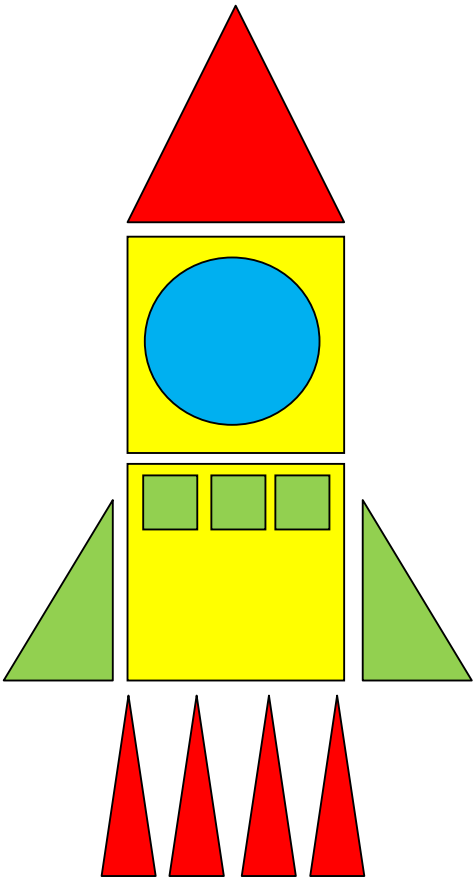
4.) Un ejemplo simple de aplicación (Referencia: proyecto *Cohete*)

Mostramos ahora un ejemplo simple, típico de una primera tarea o aplicación práctica para ser desarrollada por los alumnos en el aula cuando toman un primer contacto con la clase *GraphicsConsole*.

El enunciado fue planteado originalmente en los cursos *1k3* y *1k12* por los docentes a cargo del práctico (Ing. Marcela Tartabini, Ing. Felipe Steffolani, Ing. Romina Teicher) y se adaptó también para ser lanzado como actividad de aula en los cursos *1k1* y *1k4*. Además, este trabajo fue el que cerró la idea para el desarrollo de la clase *GraphicsConsole*: el objetivo inicial de esta actividad era sólo trabajar en el desarrollo de clases (definición de atributos y métodos, creación de objetos, uso de referencias, invocación a métodos), y el dibujo que acompañaba al enunciado era sólo a nivel ilustrativo. Pero luego se pretendió que ese dibujo fuera un objetivo a generar por el programa, y fue entonces que surgió la excusa para programar la clase *GraphicsConsole*, cuya versión 1.0 hemos presentado a lo largo de estas notas.

El enunciado del problema a desarrollar por los alumnos era entonces el siguiente:

Los padres de los niñitos de un jardín de infantes quieren pintar un cohete en la pared del patio de juegos. Después de mucho trabajar prepararon el modelo que ven (¡no vale reírse!) [1]



Ahora necesitan la invaluable ayuda de un programador para saber cuánta pintura de cada color necesitan comprar.

Las medidas que tomaron son:

- Base y altura del triangulo que forma el techo
- Diámetro de la ventana celeste
- Lado de los cuadraditos verdes (todos iguales)

También conocen estas proporciones:

- ✓ Los lados de los cuadrados amarillos tienen la misma longitud que la base del techo
- ✓ Cada alerón es exactamente la mitad del techo
- ✓ Las llamas tienen como base el mismo ancho que los cuadraditos verdes y la misma altura que el techo del cohete.

Lo que los padres necesitan es un programa que realice lo siguiente:

1. Calcular cuál es la superficie que necesitan cubrir con cada uno de los colores elegidos (rojo, celeste, amarillo y verde) ¡Atención! No vale pintar primero con un color y luego colocar otro sobre él. ¡No les hagan gastar pintura de más! [2]

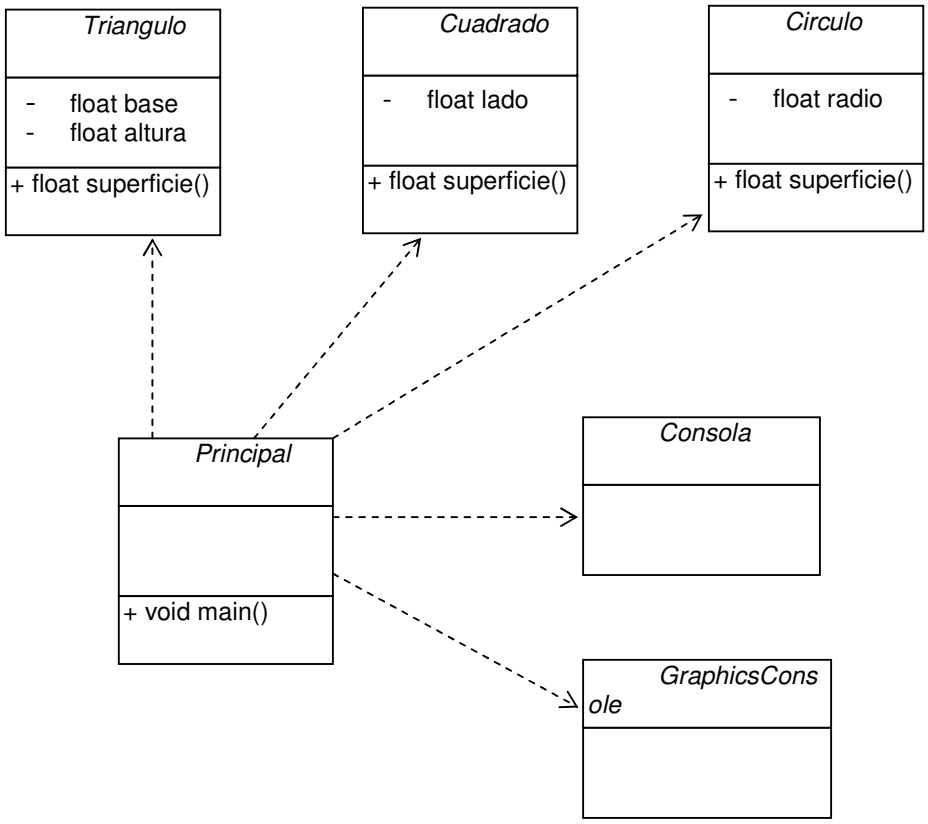
[1] [*sic*] La frase entre paréntesis "(no vale reirse!)" es textual del enunciado original...
[2] [*sic*] Otra vez...

2. Calcular un presupuesto para la compra de pintura. Para eso, consideren que en la pinturería les dijeron que cada litro de pintura cuesta $35 y alcanza para cubrir $4m^2$. Sugerencia: sería ideal que el programa considerara estos datos como variables y que se carguen por teclado.

3. Generar la gráfica del cohete en la consola de gráficos.

Aclaración

El modelo de clases a implementar ya ha sido definido y **debe respetarse**:



Los alumnos en general trabajaron en clases con esta consigna, y produjeron diversas ideas y respuestas. Finalmente, en algún momento se les sugirió una solución, que se publicó en aula virtual y que puede verse en el proyecto *Cohete* que acompaña a esta ficha.

5.) Otro ejemplo de aplicación: diagrama de barras (Referencia: proyecto *Barras*)

Un campo en el que típicamente se usan gráficos es el de la estadística. Es común que en distintas situaciones se pida realizar un gráfico "de barras", o de "sectores circulares", o de cualquier otro tipo, que muestre en forma clara la proporcionalidad entre diversos valores del ámbito de un problema.

El proyecto Barras que acompaña a esta Ficha es un ejemplo simplificado de cómo se podría producir un gráfico de barras verticales para ver la forma

en que actuaron los distintos vendedores de una empresa de ventas. El programa carga por teclado un arreglo con los importes totales facturados por un grupo de n vendedores, y a continuación abre la consola de gráficos y muestra un diagrama de barras verticales, a razón de una por cada vendedor. Cada barra tendrá una altura proporcional al total vendido por el vendedor correspondiente, y debajo del gráfico se muestran líneas de texto que aclaran lo que cada barra significa. El color de cada línea de texto corresponde a su vez al color de la barra que describe.

El modelo está abordado desde una perpectiva puramente algorítmica, y apuntando al manejo de la consola de gráficos para el procesamiento de datos. El diseño de clases en este modelo no es por eso esencial: simplemente se buscó aportar un disparador de creatividad. El análisis del código fuente del modelo es simple y directo, y nos exime de cualquier comentario adicional.

Uso de la clase JOptionPane

1.) Entrada y salida basada en ventanas: la clase JOptionPane. (Referencia: proyecto *JOptionPane*)

La clases *Consola* e *In* provistas por los profesores incluyen un conjunto de métodos para favorecer la carga de valores por teclado desde la consola estándar. La combinación de estos métodos con *System.out.print()* y *System.out.println()* permite entonces la realización de programas con salidas y entradas sencillas (o sea, con *interfaz de usuario* sencilla, basada en consola). El problema es que esta interfaz de usuario basada en consola es muy pobre y poco lucida... Muy pocos sistemas de aplicación profesional basarán su interfaz de usuario en entradas y salidas por consola estándar. Al contrario: la mayoría usarán *ventanas de alto nivel*, vistosas, elegantes y fáciles de usar mediante el mouse y el teclado.

Java provee numerosas clases y métodos dentro de ellas para manejar ventanas y producir programas con interfaz de usuario de muy alto nivel. El problema es que para manejar esas clases y métodos, se requiere de una preparación previa en *programación orientada a objetos* y en *control de eventos* que aún no se espera que tenga un estudiante de un curso introductorio.

Sin embargo, existe una clase muy particular en Java, llamada *JOptionPane* que provee métodos muy útiles para crear ventanas de uso directo llamadas "*cuadros de diálogo*". Esas ventanas ya vienen armadas: sólo es necesario llamar al método adecuado para mostrarla, enviando a ese método ciertos parámetros para configurar la ventana que se abrirá.

Uno de los métodos provistos por *JOptionPane* significa una alternativa para cargar valores por teclado. Se trata del método *showInputDialog()*. Ese método permite desplegar un cuadro de diálogo con dos botones (*aceptar* y *cancelar*) y un campo de edición para que el usuario ingrese un valor. El valor cargado en el campo de edición es retornado como un *String* cuando se presiona el botón *aceptar*. Si se presiona el botón *cancelar* en lugar de *aceptar*, el método retorna lo que se conoce como la *dirección nula*, que se maneja en Java con el valor ***null***. Para poder usar la clase *JOptionPane* en una clase, se debe agregar antes de la declaración de la clase una instrucción adicional de la forma:

```
import javax.swing.*;
```

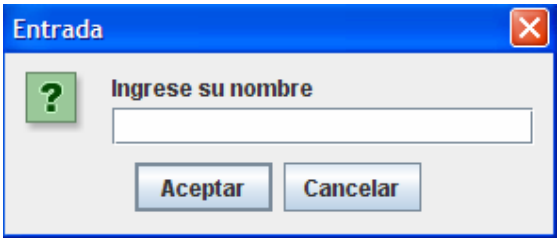
mediante la cual el programa es informado del lugar donde se encuentra grabada la clase. Esa instrucción se conoce como *instrucción de importación de paquete*. En ella, la secuencia *javax.swing* es el nombre de una carpeta que contiene clases predefinidas del lenguaje. Estas carpetas se llaman *paquetes* (o *packages*) de clases. La clase *JOptionPane* está en ese paquete, junto a muchas otras. El asterisco ubicado al final del nombre del paquete indica que el programa tendrá acceso a ***todas*** las clases del mismo.

Suponga que en un programa se desea cargar por teclado el nombre de una persona, para contestarle con un saludo. El siguiente programa muestra cómo hacerlo con el método *showInputDialog()* de la clase *JOptionPane*:

```
import javax.swing.*;
public class Principal01
```

```
{
        public static void main (String args[])
        {
              String nombre;
              nombre = JOptionPane.showInputDialog(null, "Ingrese
      su nombre: ");
              if (nombre != null)
              {
                    System.out.println("Hola " + nombre + "!!!!");
              }
              else
              {
                    System.out.println("Hola amigo... no cargaste tu
      nombre!!!");
              }
              System.exit(0);
        }
}
```

Este programa mostrará una pequeña ventana como ésta (eventualmente deberá buscarla entre las ventanas abiertas de su computadora, minimizando cada uno de los programas que tenga abiertos hasta que la vea):



Al aparecer, el usuario podrá escribir su nombre en el campo de texto que lleva la etiqueta "*Ingrese su nombre:* ", y luego presionar *Aceptar* o *Cancelar*. Cuando cualquiera de ambas cosas sea realizada por el usuario, la ventana se cerrará. Si se apretó el botón *Aceptar*, el método *showInputDialog()* devolverá la cadena que se haya escrito en el campo de texto. Si se apretó *Cancelar*, el método devolverá ***null***. Esa situación debe controlarse, y lo hacemos con un *if* en el programa: si la cadena retornada no es *null*, se pone un mensaje en la consola estandar saludando a la persona, y si volvió *null* se pone otro mensaje avisando que el nombre no fue cargado.

Hay varias versiones del método *showInputDialog()* en la clase *JOptionPane*, y se diferencian por los parámetros que se envían a cada versión. En todos los casos en que usemos este método en ejemplos simples, el primer parámetro (el primer valor que se escribe entre los paréntesis al llamar al método) será el valor *null*. En nuestro caso, luego sigue otro parámetro con la cadena que queremos mostrar encima del campo de edición.

Si se usan los métodos de la clase *JOptionPane*, el programa debería terminar con una invocación al método *exit()* de la clase *System*. Los métodos como *showInputDialog()* lanzan un *hilo de ejecución propio* (o *programa en paralelo*), el cual no termina de manera automática al terminar de ejecutarse el método *main()* del programa o aplicación. El método *System.exit()* termina con la aplicación y con todos los hilos de ejecución lanzados en ella. El método *exit( )* toma como parámetro un valor *int*, que será el código de error devuelto por el método al sistema operativo huésped. La convención es que si

el programa termina sin que haya ocurrido un error, el parámetro para *exit ( )* sea cero. Si el programa termina debido a un error, el parámetro debería ser distinto de cero.

En el ejemplo anterior, se uso este método para cargar por teclado un *String*. Pero en otros programas querremos cargar valores enteros, flotantes, etc. El hecho es que el método retorna *siempre* un *String*, por lo cual si se cargó un número, la cadena retornada deberá ser *convertida* a un *int*, un *float*, un *double*, o el valor numérico que sea. Para eso, existen métodos de conversión de cadenas a números. El siguiente ejemplo muestra cómo hacer esto en un programa que carga dos números: uno entero y otro flotante:

```
public class Principal02
{
      public static void main (String args[])
      {
            int v1 = 0;
            float v2 = 0;

            String cv1 = JOptionPane.showInputDialog(null, "Ingrese un
      valor entero: ");
            if (cv1 != null)
            {
                  v1 = Integer.parseInt(cv1);
            }

            String cv2 = JOptionPane.showInputDialog(null, "Ingrese un
      valor flotante: ");
            if (cv2 != null)
            {
                  v2 = Float.parseFloat(cv2);
            }

            System.out.println("Valor entero:   " + v1);
            System.out.println("Valor flotante: " + v2);
            System.exit(0);
      }
}
```

En el ejemplo, se llama dos veces a *showInputDialog()*: en una se carga un entero y en la otra un flotante. En ambos casos, el valor retornado es una cadena si el usuario presionó *Aceptar*. Si estamos seguros que la cadena contiene caracteres que conforman un número entero, podemos convertir esa cadena en un int mediante el método *Integer.parseInt()*, el cual toma la cadena a convertir y retorna el número *int* formado a partir de ella. Algo similar hace el método *Float.parseFloat()* y en general, existe un método similar para cada uno de los tipos primitivos.

Note que antes de intentar convertir el valor, debemos asegurarnos que se presionó el botón Aceptar (o sea, se retornó una cadena diferente de null). No obstante, si se presiona el botón *Aceptar* pero sin haber cargado ningún valor, o habiendo cargado una cadena que no puede convertirse a número, la conversión a un número fallará lanzando una *excepción*, y el programa se interrumpirá. Por ahora no controlaremos esa situación, pero lo haremos al ver el tema *Excepciones* en un encuentro posterior de este mismo curso.

La clase *JOptionPane* provee otros métodos que permiten mostrar otros tipos de cuadros de diálogo, entre los cuales están los siguientes:

showConfirmDialog(): Pide responder a una pregunta de confirmación, tipo "yes/no/cancel".
showMessageDialog(): Le informa al usuario acerca de algún evento ocurrido.
showOptionDialog(): Unifica los dos anteriores más *showInputDialog()*.

El siguiente ejemplo pide el nombre y la edad del usuario, tal que ahora el mensaje de salida se muestra en una ventana mediante el método *showMessageDialog()*. Se invita al estudiante a incorporar, testear y comprobar el funcionamiento de estos métodos alternativos, más allá de los ejemplos que aquí mostramos:

```
import javax.swing.*;
public class Principal03
{
    public static void main (String args[])
    {
        String nombre = JOptionPane.showInputDialog(null, "Ingrese su nombre:
");
        if(nombre == null) nombre = "Desconocido";

        int edad = 0;
        String cedad = JOptionPane.showInputDialog(null, "Ingrese su edad:
");
        if (cedad != null)
        {
             // convertimos la cadena a un int... si se puede!!
             edad = Integer.parseInt(cedad);
        }

        String r = "Nombre: " + nombre + "  -  Edad: " + edad;
        JOptionPane.showMessageDialog(null,       r,       "Datos:       ",
JOptionPane.INFORMATION_MESSAGE);
        System.exit(0);
    }
}
```

El método *showMessageDialog()* toma varios parámetros, y también hay varias versiones del mismo método de acuerdo a esos parámetros. En el ejemplo, la versión que usamos toma cuatro parámetros: el primero será el clásico *null* que ya hemos citado. El segundo es la cadena que queremos mostrar en el centro de la ventana. El tercero es la cadena que queremos mostrar en la barra de títulos de la ventana. Y el cuarto es una constante (de varias posibles) cuyo valor determina el *tipo de ícono* que se mostrará en la ventana al lado de la cadena que se muestra. Esas constantes pueden ser:

```
JOptionPane.INFORMATION_MESSAGE
JOptionPane.ERROR_MESSAGE
JOptionPane.WARNING_MESSAGE
JOptionPane.QUESTION_MESSAGE
JOptionPane.PLAIN_MESSAGE
```

Se invita al lector a modificar este modelo para probar estas constantes y ver sus resultados.

El siguiente ejemplo, muestra la forma de usar el método *showConfirmDialog()*, para mostrar una ventana en la cual se pedirá que el usuario conteste por sí o por no a una pregunta dada:

```
import javax.swing.*;
public class Principal04
{
    public static void main (String args[])
    {
        String nombre = JOptionPane.showInputDialog(null, "Ingrese su nombre:
");
        if(nombre == null) nombre = "Desconocido";

        String trabaja = "No";
        int resp = JOptionPane.showConfirmDialog(null, "Trabaja?", "Datos",
JOptionPane.YES_NO_OPTION);
        if (resp == JOptionPane.YES_OPTION) trabaja = "Sí";

        String r = "Nombre: " + nombre + "  -  ¿Trabaja?: " + trabaja;
        JOptionPane.showMessageDialog(null,        r,        "Datos:        ",
JOptionPane.WARNING_MESSAGE);
        System.exit(0);
    }
}
```

El método toma varios parámetros: el primero es *null*. El segundo es la pregunta que se quiere que el usuario conteste. El tercero es el título de la ventana. Y el cuarto es una constante (de varias posibles) que indica qué botones deben aparecer:

```
JOptionPane.DEFAULT_OPTION            Un botón Aceptar
JOptionPane.YES_NO_OPTION        Botones Sí - No
JOptionPane.YES_NO_CANCEL_OPTION      Botones Si - No - Cancelar
JOptionPane.OK_CANCEL_OPTION          Botones Ok - Cancelar
```

El método retorna un valor *int*, que indica el botón que presionó el usuario. Estos valores están tipificados con constantes de la clase *JOptionPane*:

```
JOptionPane.YES_OPTION          Se presionó el botón Sí.
JOptionPane.NO_OPTION           Se presionó el botón No.
JOptionPane.CANCEL_OPTION  Se presionó el botón Cancelar.
JOptionPane.OK_OPTION           Se presionó el botón Ok.
JOptionPane.CLOSED_OPTION  Se presionó el botón de cierre de la
ventana.
```

En el programa, la ventana abierta con *showConfirmDialog()* pregunta si el usuario trabaja. Cuando el usuario responde presionando el botón *Sí* o el botón *No*, la ventana se cierra y el método retorna alguna de las constantes indicadas arriba. Con un *if* preguntamos si el valor retornado fue *YES_OPTION*. En caso de serlo, se mostrará un mensaje afirmando este hecho, y si no fue así saldrá un mensaje negándolo.

El uso detallado de todos estos métodos de la clase *JOptionPane* requiere estudio y práctica. El estudiante debería tomarse su tiempo para testear estos métodos, probando diversas configuraciones y usos. La fuente

básica de estudio debería ser la documentación de ayuda de *Sun Microsystems* (que es la que *BlueJ* accede via web cuando se pide el *help*).

Es obvio que tanto los métodos de la clase *Consola*, como los métodos de la clase *JOptionPane*, no permiten generar una interfaz de salida bien diseñada y controlable. La idea aquí es simplemente mostrar la forma básica de hacer carga por teclado y contar con una interfaz de usuario más elegante hasta tanto se entre en el mundo de las clases *AWT* y *Swing* para el diseño de interfaces visuales de alto nivel.