Java:

del Grano a su Mesa

Andrés Muñoz O.

VERSIÓN 1.3

Tabla de Contenidos

Introducción

Este documento está orientado al apoyo de personas que no tengan conocimiento con el lenguaje Java, ni tampoco con conceptos de programación.

Todos los capítulos están realizados con un formato de clases, plantenado primero una motivación que generalmente es un problema que es deseable resolver y que tiene su solución con los conceptos obtenidos a través del capítulo, luego el desarrollo del tema y por último algunos problemas resueltos y propuestos para practicar con los conceptos obtenidos.

Como está recopilado por clases de cátedra de un profesor de la Universidad de Chile, existe una posibilidad que hayan inconsistencias entre los capítulos, que serán solucionados en posteriores ediciones de este apunte.

Su uso es liberado a nivel académico, tanto por alumnos del curso de Computación I como profesores que deseen usar este material como apoyo a sus clases.

También, la idea es dar más herramientas en español para aquellos programadores inexpertos que están ingresando en el mundo de Java.

Agradecimientos

A mi esposa, Verónica, quien comprendió eternamente las largas noches en los cuales preparaba las clases del año 2001 y 2002 (las que faltaban).

Además, para mis alumnos del curso de CC10A – Computación I sección 03 del año 2001, de la Escuela de Ingeniería de la Universidad de Chile, quienes motivaron el desarrollo del contenido de este apunte.

También a mis alumnos del curso de CC10A – Computación I sección 03 del año 2002, gracias a quienes logré mejorar, completar información y compilarla en este apunte.

De manera especial, agradezco a Daniel Muñoz, quien me hizo reflexionar en el nombre de este apunte y cambiarlo desde su versión original "Java: Programación y Lenguaje" a "Java: del Grano a su Mesa".

A Giselle, Pablo y Claudia quienes hicieron control de calidad de mi apunte.

Por último a los profesores y alumnos quienes utilizan este apunte, ya que gracias a ellos podré recibir comentarios y aportes a mi correo electrónico (andmunoz@dcc.uchile.cl) para mejorar más aún su ayuda académica y computacional.

A todos, gracias.

Andrés Muñoz O.
Ingeniero de Software
Profesor Universidad de Chile

Capítulo I: Principios Básicos de Java

Principios Básicos

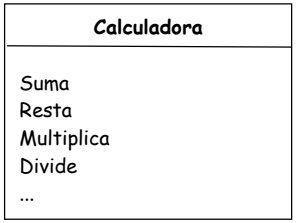
¿Qué es Java?

JAVA es un lenguaje de programación creado por SUN Microsystems (http://www.sun.com), muy parecido al estilo de programación del lenguaje "C" y basado en lo que se llama Programación Orientada al Objeto.

Programación Orientada al Objeto

Los principios de la programación al objeto es "modelar" y representar, a través de elementos de programación, objetos que existen en la realidad (tangible o intangibles). Por ejemplo, un lenguaje en donde se pueda modelar una calculadora con todas las operaciones que pueda realizar.

Es así como se encapsula y representa un objeto de la siguiente forma (notación UML[1]):

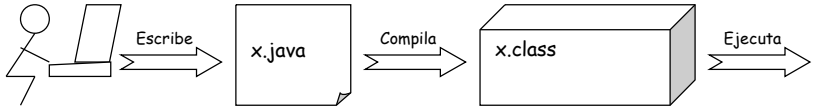


Este dibujo representa las funciones que uno puede realizar con una calculadora.

[1] Unified Model Language: Un lenguaje para modelamiento orientado al objeto que veremos más adelante en el Capítulo XXIV: Diseño de Software UML de la pagina 262.

Programación en Java

Los programas en Java son archivos de texto planos con extensión .java (se utiliza cualquier programa que escriba archivos de texto como son el Notepad de Windows, Wordpad, Textpad e inclusive Word cuando guarda en formato texto) que deben ser compilados con una JVM (Java Virtual Machine) y convertidos en un archivo .class, con el cuál pueden ser ejecutados.



Los archivos .class son binarios, por lo que no se pueden abrir con ningún programa.

Un Archivo Java

Un archivo .java debe contener la siguiente estructura base:

```
// Area de inclusión de librerías (package) [Opcional]
import <package>.*;

// Definición de la clase
[public] class <nombre de la clase> {
       // Definición de métodos
       [static] [public/protected/private] <tipo de dato> <nombre>
                     (<param1>, <param2>, ..., <paramN>) {
              ...
       }
}
```

Clase

La estructura anteriormente vista la llamaremos **Clase** y representará algún objeto o entidad, tangible o intagible que queramos modelar.

En un principio escribiramos clases que sean nuestros *Programas Principales*, es decir, aquellos que resuelven los problemas que nos planteemos. Por ejemplo:

```
public class HolaMundo {
       static public void main (String[] args) {
              Console c = new Console();
              c.println ("Hola Mundo");
       }
}
```

Esta clase representa mi programa llamado **HolaMundo** y solo escribe en pantalla la frase "Hola Mundo".

Tipos de Clases
Existen 2 tipos básicos de clases: Las clases ejecutables y las que no lo son.

La diferencia entre estas 2 clases radica físicamente en que algunas de ellas son invocadas desde otras clases. Por ejemplo, **Console** es una clase (**Console.class**) que es invocada o utilizada por la clase **HolaMundo** anteriormente vista.

Esta clase puede tener muchos métodos o funcionalidades, que son utilizadas dentro de los otros programas.

```
public class Console {
       public void print(String s) { ... }
       public void println(String s) { ... }
       public int readInt() { ... }
       public double readDouble() { ... }
       public long readLong() { ... }
       ...
}
```

La clase **HolaMundo** es del tipo que se puede ejecutar, es decir, al tener el método **main** significa que lo que está dentro de los paréntesis de llave corresponde a lo ejecutable:

```
public class HolaMundo {
       static public void main (String[] args) {  // EJECUTABLE
              Console c = new Console();
              c.println("Hola Mundo");
       }
}
```

Instrucciones

Cada línea en Java es conocida como una **Instrucción**, y significa que es lo que uno manda a realizar al computador en un determinado momento. Por ejemplo:

```
c.println("Hola Mundo");  // Indica al computador que debe
                          // imprimir en pantalla la frase
                          // "HOLA MUNDO"
```

Todas las instrucciones llevan un separador al final: ";" (punto y coma). Este separador funciona como si se le indicara al computador DONDE termina una línea, ya que Java permite que en 1 línea física escribir más de una instrucción:

```
c.print("Hola"); c.println(" Mundo");
```

Esta línea es equivalente a:


```
c.print("Hola");
c.println(" Mundo");
```

ya que el ";" indica el fin de instrucción.

También es normal que uno pueda escribir una instrucción en más de una línea física, ya que el ";" es nuestro fin de instrucción:

```
c.println("Hola" +
          " Mundo");
```

Estas 2 línea son interpretadas por el computador como si fuesen una sola:

```
c.println("Hola" + " Mundo");
```

Bloques
En Java se pueden agrupar las instrucciones en **Bloques** de instrucciones. Los bloques son delimitados por los paréntesis de llaves ("{" y "}").

Los **métodos** son bloques de programas que se ejecutan al ser invocados:

```
static public void main (String[] args) {
       Console c = new Console();
       c.println("Hola Mundo");
}
```

BLOQUE

Las **clases** se componen de 2 partes: un encabezado y un cuerpo:

```
public class HolaMundo {                                    ENCABEZADO
       static public void main (String[] args) {
              Console c = new Console();
              c.println("Hola Mundo");                       CUERPO
       }
}
```

El cuerpo de una clase es considerado un bloque de programa, ya que almacena múltiples bloques (métodos) y otras instrucciones. Los bloques también pueden almacenar otros bloques como en este caso.

Ejecución de un Programa Java

Definiremos **plan de ejecución** a las líneas que realmente son leídas y ejecutadas cuando uno invoca la clase que desea correr.

La ejecución de un programa en Java es lineal, y comienza SIEMPRE por el bloque de programa escrito dentro del método main. Al momento de ejecutar la clase, el bloque main comienza a ejecutar. Luego se realizan las llamadas necesarias a cualquier bloque o método:

```
import java.io.*;

class ejemplo{
	//este programa calcula el factorial de un numero.
	static public int fact(int numero){
		int factorial=1;

		for(int i=1; i<numero; i++){
			factorial=factorial*i;
		}

		return factorial;
	}

	static public void main(String args[]){
		Console c=new Console();

		c.print("Ingrese el numero a calcular: ");
		int numero=c.readInt();

		int factorial = fact(numero);

		c.print("El factorial de:"+ numero+ " es:"+ factorial);
	}
}
```

En este ejemplo, el plan de ejecución sería el siguiente:

1) . **[INICIO DEL PROGRAMA]**
2) Console c=new Console();
3) c.print("Ingrese el numero a calcular: ");
4) int numero=c.readInt();
5) int factorial = **[resultado del bloque siguiente]**
   a) int factorial=1;
   b) for(int i=1; i<numero; i++) **[repite el siguiente bloque]**
      i) factorial=factorial*i;
   c) return factorial; **[devuelve el valor del bloque al 4]**
6) c.print("El factorial de:"+ numero+ " es:"+ factorial);
7) . **[FIN DEL PROGRAMA]**

Podemos notar que la ejecución es muy distinta a el órden en el cual se han escrito las líneas de código (instrucciones).

En la Práctica

La idea es que ahora que conocemos como se escriben los programas, podamos saber también como llegar a que se ejecuten también.

Qué necesitas para programar

Es muy necesario utilizar el **Java Develpment Kit** (JDK) adecuado para las necesidades que tengas. Para efectos del curso, nosotros utilizaremos el JDK 1.1.8 o 1.3.0, pero nada superior a eso, ya que las funcionalidades de la Console pueden verse afectadas por otras versiones[2].

El JDK es un compilador y ejecutor de programas Java a través de algo que se llama **Java Virtual Machine** (JVM) y que es como un computador pero que solo entiende Java.

Para que tu computador tenga una JVM, es necesario que descargues el JDK y lo instales en tu computador. En el caso de la escuela se encuentra instalado el JDK 1.3.0 y las funcionalidades se ven bastante bien, pero recomendamos que instales el JDK 1.1.8 o 1.2.2.

El JDK lo puedes descargar de la página oficial de Java de Sun Microsystems (http://java.sun.com) o desde la página de ayuda de la página del curso.

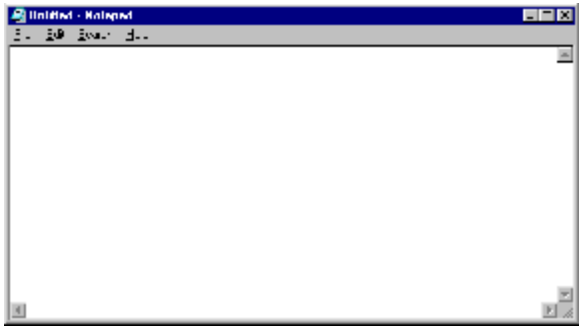
Luego que lo tengas en tu computador, instala el software y "voila". Con esto ya tienes lo necesario para comenzar tu aventura.

Como escribir un programa

Para escribir un programa lo primero que se necesita es un editor de texto. El mas simple y que viene de manera predeterminada con windows es **Notepad** (Bloc de Notas), pero cada quien es libre de usar el que mas le acomode (UltraEdit, Textpad, Jolie, etc).

El icono de notepad se encuentra en el menú: **MENU INICIO ACCESORIOS Bloc de Notas (Notepad)**

Y el abrirlo se ve una ventana como la siguiente:

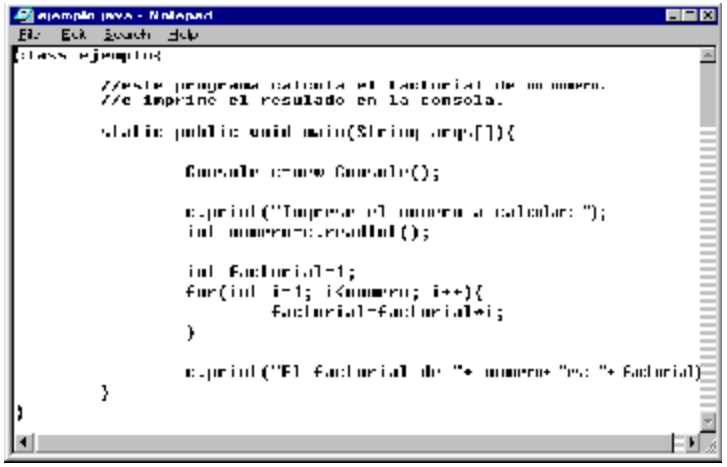


[2] Si te atreves a probar en otra versión del JDK queda estrictamente bajo tu responsabilidad.

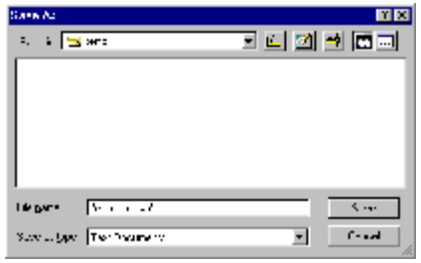
En esta ventana se puede escribir el código de algún programa. Por ejemplo el siguiente código podemos escribirlo dentro de Notepad, para que probemos:

```
class ejemplo{
	static public void main(String args[]){
		Console c=new Console();

		c.print("Ingrese el numero a calcular: ");
		int numero=c.readInt();

		int factorial=1;
		for(int i=1; i<numero; I++){
			factorial=factorial*i;
		}

		c.print("El factorial de:"+ numero+ " es:"+ factorial);
	}
}
```

Que en notepad quedaría como:



Luego, para guardar eso se debe ir al menú **Archivo/Guardar como**…



Se guarda, para ahorrar confusión, con el mismo nombre de la clase (*class ejemplo{…}*).

Las comillas aquí son necesarias o de lo contrario el archivo se guardara con el nombre como: **ejemplo.java.txt** cosa que esta muy mal porque no podremos utilizarlo como programa Java.

Existen otros editores que puedes usar. Algunos conocidos son los llamados IDE. La mayoría debes pagarlos, pero puedes encontrar versiones en la internet. Estos son:

- IBM Visual Age for Java
- IBM Eclipse (gratis)
- Borland JBuilder
- JCreator (La versión Lite es gratis)
- Sun Forte
- Jade
- Jolie (un desarrollo gratis del profesor Kurt Schwarze)

Algunos editores de texto que pueden ayudar marcando colores y estructura son:

- Textpad
- UltraEdit

Como compilar un programa
Una vez guardado el programa debemos asegurarnos que nuestro CLASSPATH este apuntando correctamente a las librerías de Java. Para ello debemos abrir una ventana de MS-DOS y poner:

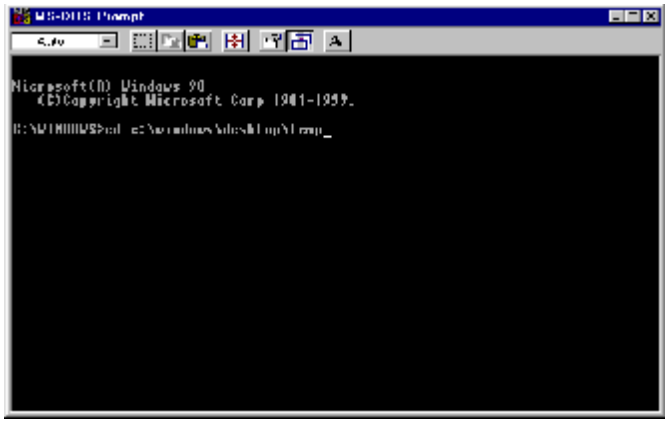
set CLASSPATH = .;C:\jdk1.1.8\lib

el directorio **jdk1.1.8** es, en este caso, la versión de Java que instalamos. Si tu versión es la 1.2.2, deberías cambiarlo en el CLASSPATH.

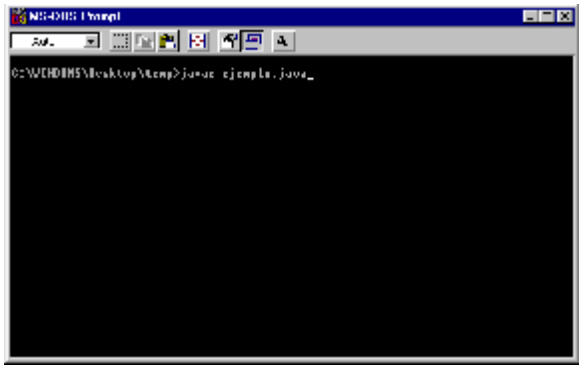
Después debemos colocar los 4 archivos que usan la consola en el mismo directorio que el programa que hemos escrito llamado **ejemplo.java:**

Console.class
ConsoleCanvas.class
Message.class
FatalError.class

Luego debemos abrir una ventana de MS-DOS y colocarnos en el directorio donde se encuentra nuestro archivo ejemplo.java mediante el comando "cd". Ejemplo:



Una vez ubicados en el directorio llamamos al comando compilador del programa:



El cual, si todo esta bien, y generalmente no lo esta (karma computin), generara el archivo ejemplo.class que contendrá el programa ya compilado, es decir, en un lenguaje que el computador si puede comprender.

Si todo está bien, volverá a mostrar la línea de comando sin poner nada en pantalla:

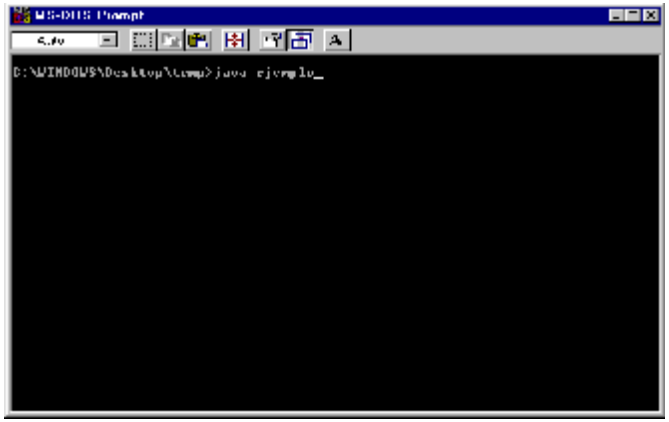
C:\WINDOWS\Desktop\temp>

y en el directorio donde están los programas, generar el archivo **ejemplo.class**.

En caso de que no se ejecute el comando javac, es probable que sea o porque java no esta instalado en el computador, o porque nuestro PATH, no apunta al directorio donde esta el comando java. En este caso, deberás poner lo siguiente y volver a compilar:

set PATH = %PATH%;C:\jdk1.1.8\bin

Como ejecutar un programa.
Una vez compilado el programa y ahora que tenemos el archivo "ejemplo.class", podemos proceder a ejecutarlo. Aquí suelen ocurrir cosas inesperadas, pero con paciencia también se pueden arreglar.



Lo cual ejecutará el programa ejemplo. Debemos hacer hincapié en que el nuevo comando se llama **java**, y no **javac** (javac java compiler), y que no se ejecuta: "java ejemplo.class", sino que solamente "java ejemplo".

Los Errores en Java

Lo más común cuando escribimos programas en Java es que tengamos errores. Existen 2 tipo de errores: **de compilación** y **de ejecución**:

Errores de Compilación

Los errores de compilación son aquellos que salen cuando estamos compilando (usando **javac**) nuestro archivo .java. Por ejemplo, veamos el siguiente programa con errores:

```
clas EjemploDeError {
      static void main (String[] args) {
            Console c = new Console();
            c.println ("Aquí le falta un punto y coma")
            c.println ("Esta línea está ok");
      }
}
```

Nótese que hemos deliberadamente puesto **clas** en vez de **class** en la definición de la clase, y en la línea 4 hemos dejado sin ; al final de ella. También no hemos puesto las clases de la consola para que veamos qué ocurre.

Si lo escribimos y lo intentamos compilar en java obtendremos el siguiente error:

```
C:\WINDOWS\Desktop\temp>javac EjemploDeError.java
EjemploDeError.java:1: 'class' or 'interface' expected
clas EjemploDeError {
^
1 error

C:\WINDOWS\Desktop\temp>_
```

En general, los errores de compilación son aquellos que son causados por un problema en la sintaxis o que no dependen de los valores de variables, métodos e ingresos de datos que ocurran al momento de ejecutar.

Al ver este caso podemos decier inmediatamente que "clas" está mal escrito. De hecho el compilador nos lo dice:

EjemploDeError.java:1	Indica el archivo y el número de la línea del error.
'class' or 'interface' expected	Éste es el error.

El que nos salió en este momento nos dice que "esperaba un class o interface", lo que significa que esperaba la palabra "class" que es claramente el error en que incurrimos. Lo que está abajo de la primera línea de error es la línea y está marcado el inicio de la palabra que está mal escrita:

```
clas EjemploDeError {      Esta es la línea errada.
^                          Indica dónde está el error en la línea.
```


Si lo corregimos y lo volvemos a compilar, el error cambiará:

```
C:\WINDOWS\Desktop\temp>javac EjemploDeError.java
EjemploDeError.java:4: ';' expected
                c.println ("Aquí le falta un punto y coma")
                                                          ^
EjemploDeError.java:3: cannot resolve symbol
symbol  : class Console
location: class EjemploDeError
                Console c = new Console();
                ^
EjemploDeError.java:3: cannot resolve symbol
symbol  : class Console
location: class EjemploDeError
                Console c = new Console();
                            ^
3 errors

C:\WINDOWS\Desktop\temp>_
```

Ahora tenemos 3 errores (como dice la última línea). 2 de ellos ocurren por causa de la **Console** (2° y 3°) y otro ocurre por falta del ;.

```
EjemploDeError.java:4: ';' expected
                c.println ("Aquí le falta un punto y coma")
                                                          ^
```

El primer error nos dice que en la línea 4 esperaba que hubiera un ";" al final de la línea. Este era el problema que habíamos dejado para probar.

```
EjemploDeError.java:3: cannot resolve symbol
symbol  : class Console
location: class EjemploDeError
                Console c = new Console();
                ^
EjemploDeError.java:3: cannot resolve symbol
symbol  : class Console
location: class EjemploDeError
                Console c = new Console();
                            ^
```

En los otros dos casos aparece la frase "cannot resolve symbol" y destacando la palabra "Console". Esto ocurre porque no encuentra el archivo **Console.class** en el CLASSPATH, es decir, o no está definido el CLASSPATH como indicamos anteriormente o faltan las clases.

Corrijamos todos los errores y veamos como queda:

```
C:\WINDOWS\Desktop\temp>javac EjemploDeError.java

C:\WINDOWS\Desktop\temp>_
```

Lo que significa que fue compilado satisfactoriamente.

Errores de Ejecución

Los errores de ejecución creo que son los más difíciles de justificar y también los más complejos de leer en Java. El **intérprete de java**[3] ejecuta un programa y al momento de encontrar un error (que no fue detectado por el compilador) lo envía a pantalla en forma de **Excepción** (Exception).

Las excepciones son algo súmamente delicado, ya que pueden darnos dolores de cabeza como nos pueden salvar la vida en otros casos, ya que se pueden manipular, a diferencia de otros lenguajes de programación[4].

Veamos un programa ejemplo que tenga errores de ejecución:

```
class EjemploDeError {
      static void main (String[] args) {
            Console c = new Console();
            c.println ("Ingresa una letra?");
            int x = c.readInt();

            String s = "Un texto";
            c.println(s.charAt(-1));
      }
}
```

En este caso hay 1 error obvio que está en el **charAt(-1)**, ya que no se puede sacar un valor menor a 0 de un string, pero veamos el otro que puede ocurrir también. Si ejecutamos este programa nos sale en pantalla:

```
Ingresa una letra?_
```

Si nosotros hacemos caso e ingresamos "a" ocurre:

```
Unable to convert to int
```

Esto nos dice que está mal lo que intentamos ingresar y el programa se pega y debemos presionar Control-C en la ventana de MS-DOS que invocamos "**java EjemploDeError**".

Probemos ahora ingresando un número (correctamente) para que caiga en la otra línea:

```
C:\WINDOWS\Desktop\temp>java EjemploDeError
Exception in thread "main"
java.lang.StringIndexOutOfBoundsException: String index out of
range: -1
        at java.lang.String.charAt(Unknown Source)
        at EjemploDeError.main(EjemploDeError.java:7)
```

[3] El Intérprete de Java (Linker) es quien se encarga de ejecutar el .class en la JVM.
[4] Esto se encuentra en el Capítulo XVII: Excepciones y Control de Errores de la pagina 160.

y nuevamente debemos presionar Control-C para terminar la ejecución del programa. Ahora tenemos algo distinto al error de conversión, pero igualmente una excepción que debemos interpretar.

Si notamos bien en la línea (ya que por tema de espacio, las 3 primera líneas son en realidad 1 sola) esta nos indica qué ocurrió:

java.lang.StringIndexOutOfBoundsException	Tipo de excepción que ocurrió.
String index out of range	Pequeña descripción de la excepción.
-1	Valor que causó la excepción.

Con estos valores podemos percatarnos que el error ocurrió porque traté de acceder con el –1 fuera del rango de un String.

Sigamos más abajo. Lo que sigue es llamado **Stack de Ejecución** y nos indica cuántos métodos hemos llamado desde que se ejecutó el primer **main**. Su lectura es de abajo hacia arriba, comenzando por el **main** en la última línea y acabando por el método más interno en la primera.

También, cada línea nos indica en qué archivo y línea ocurrió el error.

En este caso, el stack es:

```
at java.lang.String.charAt(Unknown Source)
at EjemploDeError.main(EjemploDeError.java:7)
```

Si lo analizamos detenidamente (de arriba hacia abajo) dice:

a) Se cayó en el método **charAt** de la clase **java.lang.String**.
b) No tenemos el código fuente de esa clase.
c) El que llamó a **charAt** es el **main** de la clase **EjemploDeError**.
d) El archivo que posee esta clase es **EjemploDeError.java**.
e) La línea que está haciendo esta invocación es la línea 7.

Por lo que si vamos a la línea 7 del archivo encontraremos:

```
c.println(s.charAt(-1));
```

que es exactamente el problema que queríamos encontrar.

Más adelante en el curso utilizaremos las excepciones como un apoyo a la programación en Java al momento de incurrir en errores de ejecución.

Principios Avanzados

Uso de PUBLIC/PROTECTED/PRIVATE:

Estas sentencias (que pueden estar en las firmas de los métodos o en las definiciones de las clases) nos indican el nivel de "PRIVACIDAD" que tiene la clase.

METODOS:
La privacidad (a nivel de métodos) indica desde DONDE puedo llamar un método. Es decir:

- PUBLIC: Lo puedo llamar desde cualquier parte (clases dentro del mismo directorio o de otro directorio).
- PROTECTED: Solo las clases "hijas" pueden llamarlo (concepto de herencia más avanzado que veremos luego).
- PRIVATE: Solo la misma clase que lo define puede llamarlo.
- Sin nada: Para las clases del mismo directorio es pública, es decir la pueden llamar. Desde otro directorio no existe.

Estas palabras no son OBLIGATORIAS, ya que por definición si no ponemos nada, es accesible.

CLASES:
A nivel de clases, la privacidad se refleja casi de la misma forma que los métodos. Es decir:

- PUBLIC: Se puede utilizar esta clase desde cualquier lugar. OBLIGA a guardar la clase en un archivo con el MISMO NOMBRE (ojo con las mayúsculas y minúsculas):

```
public class Ejemplo {
      ...
}
```

  se debe guardar en Ejemplo.java y no en ejemplo.java.

- Sin nada: Se puede utilizar en el mismo directorio. Aquí el nombre del archivo JAVA no importa, es decir:

```
class Ejemplo {
      ...
}
```

  se puede guardar en Ejemplo.java, ejemplo.java o miprograma.java

Uso de STATIC:

El uso de STATIC en los métodos solo nos explica cuando utilizamos un "OBJETO" o cuando utilizamos una "CLASE" para invocar el método.

OBJETO:
Un objeto es una variable de un tipo Clase que es creada con un new:

```
Console c = new Console();
```

en este caso "c" es un objeto de tipo "Console". Todos los métodos de la clase Console están declarados sin la palabra STATIC, por ejemplo, la firma de "println" sería:

```
public void println(String s) {
      ...
}
```

y es por esta razón que necesitamos al objeto "c" para invocarlo como:

```
c.println("Esto usa un objeto");
```

CLASE:
Las llamadas a métodos que se realizan a través del nombre de la clase REQUIEREN la palabra "static" en su definición. Por ejemplo, en la clase Math tenemos el método:

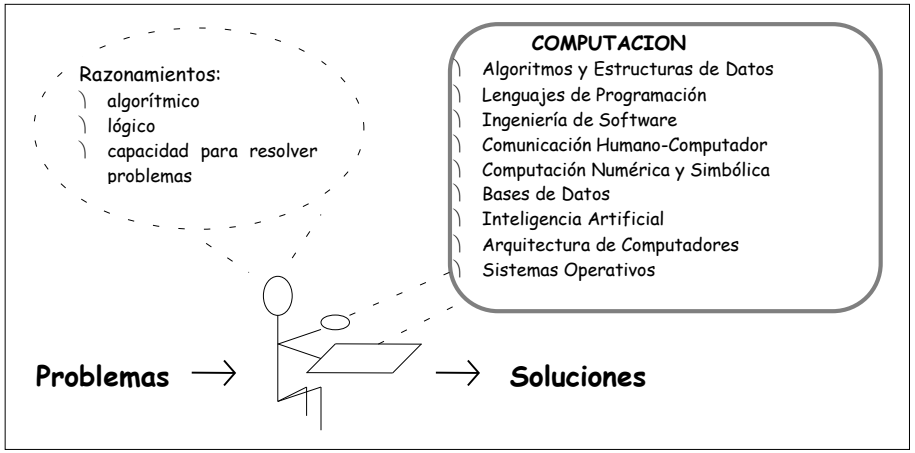
```
public static double random() {
      ...
}
```

y como posee el "static", su invocación sería del tipo:

```
double x = Math.random();
```

Capítulo II: Conceptos Básicos

Motivación



Escribir un programa que instruya al computador para que establezca el siguiente diálogo con una persona (usuario):

```
Por favor ingresa un N°:  123
Gano yo con el 124
```

Conceptos

Algoritmo

*Se define como "**Algoritmo**" a la serie de pasos o "Etapas" (que debe realizar el computador) para resolver un problema*

En el problema planteado arriba, un "algoritmo" para resolverlo se describe a continuación:

1. Escribir (mostrar) en la pantalla la frase "**Por favor ingresa un N°:**"
2. Leer (obtener, recuperar) el n° ingresado por la persona usando el teclado.
3. Escribir en la pantalla:
   - "Gano yo con el "
   - el número (ingresado por la persona en el paso 2) sumándole uno.
4. Terminar el programa

Programa

*Traducción de un "**Algoritmo**" en un lenguaje de programación que el computador pueda interpretar para resolver el problema.*

Para efectos de estudio en este curso, el lenguaje de programación utilizado para interpretar nuestros algoritmos es **Java**.

Traduzcamos el algoritmo que definimos para el problema inicial y veamos como queda en lenguaje Java:

```
C.print("Por favor ingresa un N°:");

int n;
n = C.readInt();

C.print("Gano yo con el ");
C.print(n+1);
```

Para comprender mejor el lenguaje, iremos línea a línea analizando qué significa y cómo se utilizó para traducir el algoritmo anterior.

C.print("Por favor ingresa un N°:");
- Escribe en la pantalla la frase encerrada entre comillas ("").
- **C** es un objeto que representa la consola (pantalla y teclado) del computador.
- **print** es un método (función) que se aplica al objeto **C** y que escribe su argumento en la pantalla.

int n;
- Declara **n** como una variable entera.
- **Variable**
  - Representación simbólica de un valor (número).
  - Representa una ubicación (celda) en la memoria del computador.
  - Posee un nombre que la identifica (letra seguida de latras, dígitos o _).
  - Capacidad: **un** valor del tipo indicado.
  - **int** indica que el tipo de número que puede almacenar la variable es un entero (números sin punto decimal).

n = C.readInt();
- Lee un número entero desde el teclado y lo asigna (guarda) a (en) la variable **n**.
- **readInt()** es un método (función sin argumentos) que:
  - Espera que el usuario ingrese un número (tecleando dígitos y ENTER).
  - Lee (obtiene, reconoce) el número.
  - Entrega (retorna) el número como resultado.
- Para abreviar las dos líneas anteriores se puede utilizar **int n = C.readInt();**

C.print("Gano yo con el ");
- Escribe en la pantalla la frase "**Gano yo con el**".

C.print(n+1);
- Escribe en la pantalla el valor de la variable **n** más uno.
- **n+1** Expresión aritmética (**n**: variable, **1**: constante, **+**: operador).

- Operadores Aritméticos válidos:
  - Adición: (+)
  - Substracción: (-)
  - Producto: (*)
  - Cuociente: (/)
- Para abreviar las dos líneas anteriores se puede utilizar **C.print("Gano yo con el " + (n+1));**
  - (+) : Este operador se convierte en un concatenador (añade) de caracteres.
- Mejora: **C.println("Gano yo con el " + (n+1));** escribe y después posiciona el cursor al comienzo de la siguiente línea en la pantalla

Con estos conceptos, se pueden realizar muchos más programas que este sencillo ejemplo. La mayor cantidad de los problemas planteados a la computación se basan en esta básica interacción entre usuario-computador, para hacerla sencilla y que el computador pueda obtener la información necesaria para resolver el problema.

Solución al Problema

Una solución completa al problema planteado, la cual sería fácilmente probada en el laboratorio, se presenta a continuación.

```
// Jalisco: programa que nunca pierde
import java.awt.*;
class Jalisco {
      static public void main(String[] args) {
            Console C = new Console();
            C.print("Por favor ingresa un N°:");
            int n = C.readInt();
            C.println("Gano yo con el "+ (n+1));
      }
}
```

De la misma forma que en la anterior explicación se presentó, se detalla línea a línea la ejecución del programa con la traducción que el computador realiza.

// Jalisco: programa que nunca pierde
- Comentario hasta el final de la línea.
- El computador no traduce esta línea, puesto que es solo una línea informativa y no influye en la ejecución del programa.

import java.awt.*;
- Inserta declaraciones necesarias para que programa lea/escriba

class Jalisco {…}
- Define la clase de nombre **Jalisco**.
- Todo programa Java debe estar contenido en una clase.

static public void main(String[] args) {…}
- Método que contiene instrucciones del programa principal (**main**).
- Encabezamiento estándar (significados se explicarán posteriormente).

Console C = new Console();

- Abre una ventana para leer y escribir.
- **C**: objeto de clase **Console**.
- **Console**: clase de interacción predefinida con métodos para leer y ecribir.

Problemas

Escribir un programa que establezca el siguiente diálogo.

```
Calcular perímetro y area de circulo
Diametro ? 3
Perimetro = ...
Area = ...
```

Solución 1.

Esta solución considera que el diámetro es un número **entero** y que PI es un número **real**:

```
// Se declara la constante pi
final double pi = 3.1416;

// Se crea una Consola para entrada y salida de datos
Console C = new Console();

// Se obtiene el valor del diámetro del círculo
C.println("Calcular perímetro y area de circulo");
C.print("Diametro ? ");
int d = C.readInt();

// Se calcula y despliega los resultados de la operación
C.println("Perímetro = " + (pi * d));
C.println("Area = " + ( pi * Math.pow(d/2, 2));
```

Solución 2

Esta solución considera que el diámetro es un número **real** y que PI es un número **real**:

```
// Se declara la constante pi
final double pi = 3.1416;

// Se crea una Consola para entrada y salida de datos
Console C = new Console();

// Se obtiene el valor del diámetro del círculo
C.println("Calcular perímetro y area de circulo");
C.print("Diametro ? ");
double d = C.readDouble();

// Se calcula y despliega los resultados de la operación
C.println("Perímetro = " + (pi * d));
C.println("Area = " + ( pi * Math.pow(d/2, 2));
```

Los resultados que se obtienen para el perímetro son iguales, sin embargo para el área son complétamente distintos:

Caso **d** entero:

```
Perímetro = 9.4248
```

```
Area = 3.1416
```

Caso **d** real:

```
Perímetro = 9.4248
Area = 7.0686
```

Este fenómeno ocurre normalmente en Java, puesto que al operar un cuociente (caso que puede dar un resultado con decimales) con ambos valores **enteros**, el resultado SIEMPRE será **entero**.

Por ejemplo:	3 / 2 = 1	entero
	3 / 2.0 = 1.5	real

Problemas Propuestos

1. Programa para el siguiente diálogo

```
Calcular perímetro y area de rectángulo
Largo ? 2
Ancho ? 3
Perímetro = ...
Area = ...
```

2. Diseñar el diálogo y escribir un programa que calcule la velocidad de un móvil en km/hora, dadas la distancia (en metros) y el tiempo (en segundos).

3. Inventar un problema, diseñar el diálogo y escribir.

Capítulo III: Entrada y Salida

Motivación

La I/O (Entrada/Salida) estándar es, por definición, cómo se comunica nativamente Java con el usuario.

Es así como clases hechas como **Console** evitan que uno conozca realmente donde está el ingreso de datos y el despliegue de ellos en la pantalla o interfaz (gráfica de texto) que Java provee. De hecho, **Console** es un Applet que utiliza un Canvas en donde se escriben las líneas de texto para la salida estándar y que permite el ingreso de datos a través del teclado, para luego pasárselo a los programas que la utilizan.

Sintaxis

Clase Console[5]

Esta clase es simplemente un encapsulamiento de la Entrada y Salida estándar de Java. En este documento se utiliza la mayor parte de los capítulos por simplicidad, no obstante la utilización de System para entrada y salida estándar también se detalla en caso de desear utilizarla.

Le definición de Console permite realizar las siguientes funcionalidades:

Método	Descripción
public Console();	Constructores de la clase Console por defecto, con título, y con tamaño y título respectivamente.
public Console(String);	
public Console(int,int, String);	
public int maxcol();	Obtener el tamaño de columnas y filas que puede contener la consola abierta.
public int maxrow();	
public void clear();	Limpiar la consola.
public void showCursor();	Mostrar y ocultar el cursor.
public void hideCursor();	
public void print(String);	Imprimir en la consola.
public void println(String);	
public boolean readBoolean();	Leer un valor desde el teclado utilizando la consola.
public byte readByte();	
public short readShort();	
public int readInt();	
public long readLong();	
public double readDouble();	
public float readFloat();	

[5] Ver http://www.holtsoft.com/java/hsa_package.html#Console para mayor información.

public char readChar();	
public String readString();	
public String readLine();	
public void setFont(Font);	Dar tipografía (Font) y color al texto de la consola[6].
public void setTextBackgroundColor(Color);	
public void setTextColor(Color);	

Y también existen formas de utilizarla como un lienzo gráfico para dibujar[7]:

Método	Descripción
public int maxx();	Obtener tamaño del lienzo.
public int maxy();	
public void setColor(java.awt.Color);	Dar color al pincel.
public void clearRect(int, int, int, int);	Limpiar un trozo del lienzo.
public void copyArea(int, int, int, int, int, int);	Copiar un trozo del lienzo.
public void draw3DRect(int, int, int, int, boolean);	Dibujar todo tipo de figuras.
public void drawArc(int, int, int, int, int, int);	
public void drawLine(int, int, int, int);	
public void drawMapleLeaf(int, int, int, int);	
public void drawOval(int, int, int, int);	
public void drawPolygon(int[], int[], int);	
public void drawRect(int, int, int, int);	
public void drawRoundRect(int, int, int, int, int, int);	
public void drawStar(int, int, int, int);	
public void drawString(String, int, int);	
public void fill3DRect(int, int, int, int, boolean);	
public void fillArc(int, int, int, int, int, int);	
public void fillMapleLeaf(int, int, int, int);	
public void fillOval(int, int, int, int);	
public void fillPolygon(int[], int[], int);	
public void fillRect(int, int, int, int);	
public void fillRoundRect(int, int, int, int, int, int);	
public void fillStar(int, int, int, int);	

Clase System

En algún lado ya se ha utilizado la clase **System**, la cual se basa en una serie de funcionalidades *estáticas* (definidas como **static**) que permiten interactuar nativamente entre el sistema y el usuario (o los periféricos que tenga). Por ejemplo:

System.out es un objeto que posee una referencia a la pantalla de salida estándar de Java.

[6] Para ver cómo funcionan los colores, referirse a Canvas en página 151.
[7] Para ver cómo funcionan algunos de los métodos gráficos, ver sección Canvas en página 151.

Ahondando en este ejemplo, **System.out** posee las funcionalidades de imprimir en pantalla que han trascendido a objetos como la **Console** o **PrintWriter**:

- **System.out.println(...)**: Imprime en pantalla lo que está entre paréntesis (literal o expresión) y salta una línea después de hacerlo.
- **System.out.print(...)**: Imprime en pantalla lo que está entre paréntesis (literal o expresión) pero sin saltar la línea al final.

Por ejemplo, lo que con **Console** imitaba el siguiente diálogo:

```
Hola mucho gusto.
```

era:

```
Console c = new Console();
c.println("Hola mucho gusto");
```

con **System.out** es mucho más sencillo: al ser una variable estática de la clase **System**, no necesita de una referencia a dicha clase:

```
System.out.println("Hola mucho gusto");
```

Bastante sencillo. Veamos ahora como se leen datos con **System**.

El caso de entrada de datos es más complejo y es lo que lleva a los programadores a crear objetos como **Console**. Es decir, no es tan sencillo como poner:

```
System.in.readInt();  // ESTO ESTA MALO
```

Aunque les gustaría mucho. ♍

System.in es una referencia estándar a la entrada desde el teclado. Sin embargo su uso es un poco distinto, pero similar a lo que pasa con los archivos de lectura, ya que también son entradas, esto es:

```
BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
```

Esta línea reemplazaría a la declaración de la consola en el caso de utilizar **Console**. Es decir, lo que antes imitaba a:

```
Cómo te llamas? Juan
Hola Juan, mucho gusto
```

y que con **Console** quedaba más o menos así:

```
Console c = new Console();
c.print("Como te llamas?");
```

```
String nombre = c.readLine();
c.println("Hola " + nombre + ", mucho gusto");
```

Ahora cambiaría un poquito como:

```
BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
System.out.print("Como te llamas?");
String nombre = in.readLine();
System.out.println("Hola " + nombre + ", mucho gusto");
```

Hey, no es tan distinto. Pero el cambio viene al trabajar con números, ya que **BufferedReader** solo puede leer líneas de texto, por lo que métodos **readInt** y **readDouble** quedan completamente fuera de lugar. Solo se puede utilizar **readLine()** y leer solo Strings (hay que hacer un cast o conversión explícita[8]).

Problema Propuesto

Tenemos el siguiente programa desarrollado con Console:

```
Console c = new Console();

while (true) {
      c.print("Pregunta: ");
      String p = c.readLine();
      c.print("Respuesta: ");
      switch (p.charAt(0).toUpperCase()) {
            case "A":
                  c.println("Si, toda la razón");
            case "E":
                  c.println("No es cierto");
            case "I":
                  c.println("Es muy probable");
            case "O":
                  c.println("Nunca");
            case "U":
                  c.println("Siempre");
            default:
                  c.println("Quizás");
      }
}
```

Trate de convertir este código para que utilice **System.out** y **System.in** como entrada y salida de datos

[8] Conversiones **String > int** y **String > double**.

Capítulo IV: Asignaciones, Expresiones y Tipos de Datos

Motivación

Nos gustaría mucho realizar el cálculo de porcentajes de elección de 2 candidatos instruyendo al computados para que establezca el siguiente diálogo con el usuario:

```
Calcular porcentajes
Votos candidato 1? _
Votos candidato 2? _
Total de votos = N°
Candidato 1 = xxx %
Candidato 2 = xxx %
```

Concepto

A continuación haremos las definiciones básicas necesarias para comenzar con los conceptos de programación y que nos serán útiles a través del curso:

Expresión

> *Es una combinación de operadores, variables y constantes del lenguaje que, al ser evaluada, permite la obtención de un valor reutilizable en otro lugar del programa.*

Para ver claramente esto, la siguiente línea es una expresión:

$$((6 * a) - (5 + b) / c) * x^2$$

La evaluación de las expresiones sigue las mismas reglas del álgebra. Estas son:

1. Expresiones Parentizadas
2. Operadores Unarios
3. Operadores Multiplicativos (*, /)
4. Operadores Aditivos (+, -)

Y en el caso de existir operadores de igual prioridad, se evalúan de izquierda a derecha.

Asignación

> *La asignación es una instrucción en la cual el computador da un valor para ser almacenado dentro de una variable o espacio de memoria física.*

La sintaxis de una asignación es:

```
<variable> = <expresión>;
```

En donde:

<expresión> se escribe en una línea (hacia el lado) y no en varios niveles. La comprenden variables, constantes, operadores binarios (+, -, *, /), operadores unarios (+, -), métodos y expresiones entre paréntesis. Antes de ser asignada a una variable, esta expresión es EVALUADA y el valor obtenido reside en el espacio reservado de memoria para dicha variable.

Por ejemplo, podemos realizar las siguientes asignaciones

```
a = v1 + v2;
a = (5 – a) * 3 / v1;
a = 2;
```

Tipos de Datos

Son definiciones de espacios de memoria en los cuales se almacenan valores específicos.

Esta definición es bastante técnica, por lo que simplificaremos un poquito a la siguiente definición de tipo de dato:

Es una clase de números (o datos) con los cuales se pueden definir las variables que los almacenarán.

En Java existen los tipos de datos numéricos:

Tipo	Nº Bits	Mayor Valor	Menor Valor	Precisión (dígitos)
Byte	8	127	-128	3
Short	16	32.767	-32.768	5
Int	32	2.147.483.647	-2.147.483.648	10
Long	64	$2^{63} - 1$	-2^{63}	19
Float	32	$3{,}4 \times 10^{38}$	$-3{,}4 \times 10^{38}$	7
Double	64	$1{,}7 \times 10^{301}$	$-1{,}7 \times 10^{301}$	15

Para definir o declarar una variable con cierto tipo, se debe utilizar la siguiente sintaxis:

<tipo de dato> <nombre de la variable>;

en donde

<nombre de la variable> es un nombre cualesquiera que le permita identificar a ese espacio de memoria separado para el tipo de dato indicado.

Por ejemplo:

```
int i;
i = 5;
```


Conversión de Datos

Para convertir tipos de datos se usa un CAST explícito o implícito que transforma, convierte y/o trunca desde un tipo de dato a otro.

Los **cast implícitos** se pueden hacer desde tipos de más pequeño tamaño (ver tabla) hacia tipos más grandes. Por ejemplo:

```
int a = 6;
double b = a;       // Cast implícito
```

Los **cast explícitos** se deben hacer desde tipos de más grande tamaño (ver tabla) hacia tipos más pqueños. Por ejemplo:

```
double a = 10.0;
double b = (double) a;     // Cast explícito
```

Solución

```
Console c = new Console();
c.println("Calcular porcentajes");

c.print("Votos candidato 1");
int v1 = c.readInt();

c.print("Votos candidato 2");
int v2 = c.readInt();

int total;
total = v1 + v2;

c.println("Total de votos = " + total);
c.println("Candidato 1 = " + 100.0*v1/total);
c.println("Candidato 2 = " + 100.0*v2/total)
```

Capítulo V: Métodos y Funciones

Motivación

Existen muchas funciones matemáticas y no matemáticas en Java que se encuentran pre-definidas. Estas funciones están escritas para que los programadores las utilicen en sus códigos sin volver a escribir los subprogramas que realizan esas funcionalidades.

Concepto

Funciones/Métodos

> *Trozos de código que pueden ser reutilizados a través de una llamada a su definición con ciertos parámetros. También reciben el nombre de* ***Métodos****.*

En Java por ejemplo existe una gran librería matemática que incluye métodos para la raíz cuadrada, potencias, exponenciales, logaritmos, senos, cosenos, tangentes, máximos, mínimos, redondeo de valores y números aleatorios.

Invocación o Llamada de un Método

> *La llamada es la forma en que el lenguaje invoca o permite la ejecución del trozo de código escrito dentro de un método. En general, requiere de un nombre (del método) y sus parámetros (si es que tiene).*

Para que veamos como se invocan los métodos, veamos aquellos que se encuentran en la librería matemática:

Función	Significado	Tipo Argumento	Tipo Resultado	Ejemplo	Resultado
sqrt(x)	$\sqrt{x}$, $x \geq 0$	double	double	sqrt(4.0)	2.0
abs(x)	|x|	i, l, f, d	del arg	abs(-3)	3
pow(x,y)	x^y	d	d	pow(2.0,3)	8.0
exp(x)	e^x	d	d	exp(1)	**Math.E**[9]
log(x)	$\log_e x$	d	d	log(Math.E)	1.0
sin(x)	seno de < x (x en radianes)	d	d	sin(Math.PI/2)	1.0
cos(x)	coseno de < x	d	d	cos(Math.PI)	-1.0
tan(x)	tangente de < x	d	d	tan(Math.pi/4)	1.0
asin(x)	arco-seno de x	d	d	asin(1.0)	**Math.PI**[10]/2
acos(x)	arco-coseno de x	d	d	acos(-1.0)	**Math.PI**

[9] Math.E está definida en la librería matemática.
[10] Math.PI está definida en la librería matemática.

Función	Significado	Tipo Argumento	Tipo Resultado	Ejemplo	Resultado
atan(x)	arco-tangente de x	d	d	atan(1.0)	**Math.PI**/4
round(x)	redondear x	d, f	l, i	round(4.5)	5L
floor(x)	n / n ®x < n+1	d	d	floor(4.9)	4.0
ceil(x)	n / n-1 < x ®n	d	d	ceil(4.1)	5.0
max(x,y)	mayor entre x e y	i,l,f,d	de arg	max(4.1, 6.5)	6.5
min(x,y)	menor entre x e y	i,l,f,d	de arg	min(4.1, 6.5)	4.1
random( )	N° al azar en [0,1[		d	random( )	0.x...

Veamos unos ejemplos de utilización de funciones de la librería matemática:

```
// Cálculo de Area y Perímetro
c.print("Ingrese el radio de la circunferencia?");
double r = c.readDouble();

c.print("El perímetro de la circunferencia es: ");
c.println(2 * Math.PI * r);
c.print("El area de la circunferencia es: ");
c.println(Math.PI * Math.pow(r, 2));
```

En este ejemplo podemos ver como utilizamos un método de la clase matemática dentro de una expresión.

```
// Cálculo de un radio a partir del área
c.print("Ingrese ahora un Area de circunferencia?");
double a = c.readDouble();

c.print("El radio de la circunferencia es: ");
c.println(Math.sqrt(a / Math.PI));
```

En este ejemplo podemos ver como se puede utilizar una expresión como argumento de un método de la clase matemática.

```
// Cálculo de la tangente a partir de otras funciones
c.print("Ingrese un ángulo en radianes? ");
double ang = c.readDouble();

c.print("La tangente original es: ");
c.println(Math.tan(ang));

c.print("La tangente calculada es: ");
c.println(Math.sin(ang)/Math.cos(ang));
```

En este ejemplo podemos ver que se pueden componer en una misma expresión distinto métodos matemáticos sin necesidad de utilizar variables adicionales.

Motivación

Quisiéramos escribir trozos de código que permitan realizar las operaciones básicas de la aritmética (sumar, resta, división y multiplicación), para utilizarlos a distintos niveles.

Es por eso que nos gustaría crear una estructura de programación que nos permita almacenar subprogramas pre-hechos para ser utilizados dentro de nuestros propios programas.

Concepto

Declaración de un Método

La declaración de un método es la escritura del subprograma que resuelve la funcionalidad del mismo.

La forma general de declaración de un método es la siguiente:

```
static <tipo> <nombre> (<arg1>, <arg2>, ...) {
      <instrucciones>
      return <valor de retorno>;
}
```

En donde:

<tipo>	Tipo de dato del valor que se retorna al final de la ejecución del método.
<nombre>	Nombre con el cuál es identificado el método.
<argn>	Argumento n-ésimo del método. Puede tener entre 0 y cualquier número de argumentos.
<instrucciones>	Trozo de código que se procesa durante la ejecución del método.
<valor de retorno>	Valor que retorna el método a su línea llamadora y que es resultado del procesamiento realizado en el trozo de código o cuerpo del método.

La combinación entre tipo, nombre y argumentos se le llama **Firma del Método**.

Por ejemplo, definamos un método que retorne el valor de una raíz quinta:

```
static double raizQuinta (double x) {
      return Math.pow(x, 1/5);
}
```

y su llamada será (en negritas):

```
double rq = raizQuinta(26);
```

Solución

Con esto definido, podemos escribir el código de los métodos de la segunda motivación:


```
static double suma (double a, double b) {
      return a + b;
}

static double resta (double a, double b) {
      return suma(a, -b);
}

static double multiplica (double a, double b) {
      return a * b;
}

static double division (double a, double b) {
      return multiplica (a, 1/b);
}
```

Y para utilizar esta divertida versión es:

```
double val1 = 5;
double val2 = 10;

c.println (val1 + " + " + val2 + " = " + suma (val1, val2));
c.println (val1 + " - " + val2 + " = " + resta (val1, val2));
c.println (val1 + " * " + val2 + " = " + multiplica (val1, val2));
c.println (val1 + " / " + val2 + " = " + division (val1, val2));
```

Problema

Escribir una función que calcule el máximo de 3 números y otra el mínimos de 3 números reales.

```
static double max3 (double val1, double val2, double val3) {
      return Math.max(val1, Math.max(val2, val3));
}

static double min3 (double val1, double val2, double val3) {
      return Math.min(val1, Math.min(val2, val3));
}
```

Desafíate: Saca ahora el de en medio

Propuesto

Escribir una función que calcule el número aleatorio entre una cota inferior x y otra cota superior y. Es decir, que tenga la siguiente firma:

```
static int aleatorio (int x, int y)
```

el resultado [x, y]

Capítulo VI: Condicionales

Motivación

Escribir subprogramas que realicen los operaciones de adición y cuociente de número reales, considerando que no se puede dividir por 0.

Algoritmo

Veamos primero el algoritmo para la adición a modo de práctica:

1. Recibir los valores de los sumandos dentro del subprograma (llamada del método).
2. Guardar temporalmente el valor de la suma de los valores obtenidos en la llamada.
3. Retornar el resultado guardado en el paso 2.

Para este algoritmo no hay problema de escribir la solución:

```
static double suma (double a, double b) {
      double r = a + b;
      return r;
}
```

Una versión alternativa y avanzada sería:

```
static double suma (double a, double b) {
      return a + b;
}
```

Ok. Veamos el algoritmo para la división:

1. Recibir los valores de los operandos del cuociente.
2. **Verificar si el dividendo es igual a 0**
    a. **Retornar 0 (indicando que era 0 para que no haya error)**
3. Guardar temporalmente el valor de la división de los valores obtenidos en la llamada.
4. Retornar el resultado guardado en el paso 3.

Como podemos ver, necesitamos algo que nos permita ejecutar **(a)** solo si el dividendo es 0.

Concepto

Condicionales

> *Un **condicional** es una instrucción que permite ejecutar un trozo de código si y solo si cumple con una condición o valor de verdad.*

El concepto de condicional es bastante básico. Las condiciones son expresiones que retornan valores de verdad (verdadero o falso) y que condicionan la ejecución de instrucciones indicadas en el condicional.

Algo como:

```
Si la condición es cierta
      Ejecuta las instrucciones cuando es cierta
Si no es cierta
      Ejecuta estas otras instrucciones.
```

Sintaxis

Los condicionales utilizan un comando especial llamado **if ... else**. Su forma general es como sigue:

```
if (<condición>) {
      <instrucciones si es verdadero>
}
else {
      <instrucciones si es falso>
}
```

La primera parte del condicional corresponde a la condición, la segunda parte (o **else**) corresponde a decir "si la condición NO se cumple" o el famoso "si no" o "de lo contrario". Esta segunda parte es opcional.

Antes de continuar, debemos insertar dos nuevos concepto:

Valor de Verdad

Todos saben cuál es la definición e este concepto. Así que abordaremos un nuevo tipo de dato que soporta almacenar esto: **bolean**.

El **bolean** es un tipo de dato especial que solo puede almacenar VERDADERO o FALSO. Por ejemplo:

```
boolean var1 = true;
boolean var2 = false;
```

Es sumamente sencillo. La gracia es que se puede usar como condición sola en un if. Por ejemplo:

```
if (var1)
...
else
...
```

Operadores Lógicos

Un operador lógico es un comparador que permite operar valores, variable o expresiones y retornar un valor de verdad (verdadero o falso).

Existe muchos operadores lógicos que describiremos a continuación:

Comparador de Valores Numéricos: Permite comparar valores numéricos indicando si es mayor, menor, mayor o igual, menor o igual, distinto o simplemente igual.

```
A > 5        OPERADOR MAYOR QUE
A < 3        OPERADOR MENOR QUE
A >= 1       OPERADOR MAYOR O IGUAL QUE
A <= 0       OPERADOR MENOR O IGUAL QUE
A == 4       OPERADOR IGUAL QUE
A != 4       OPERADOR DISTINTO QUE
```

Operador de Negación: Este operador permite NEGAR un valor de verdad, es decir cambiar el valor de VERDADERO a FALSO o de FALSO a VERDADERO.

```
! (A == 5)   SE TRADUCE POR A != 5
```

Conjunciones: Son operadores especiales que permiten unir distintos valores de verdad. Hablando un poco en español sería como el Y y el O (bueno, son eso).

```
A > 5 && A < 10     A mayor que 5 y menor que 10
A < 9 || A > 10     A menor que 9 o mayor que 10
```

Un ejemplo que utiliza todo lo anterior sería condicionar si es par o impar un número:

```
c.print("Número?");
int n = readInt();
if ( n % 2 == 0 ) {
      c.println(n + "Es par");
}
else {
      c.println(n + "Es impar");
}
```

Esto es muy fácil de visualizar. Veamos un ejemplo más complejo considerando 3 casos distintos:

```
c.print("Número?");
int n = readInt();
if ( n > 0 ) {
      c.println(n + " es mayor que 0");
}
else if ( n < 0 ) {
      c.println(n + " es menor que 0");
}
else {
      c.println(n + " es igual que 0");
}
```


En este ejemplo podemos ver 3 casos y como se pueden mezclar los distintos if ... else.

Por último un ejemplo sencillo de que hacer solo si se cumple la condición:

```
c.print("Número?");
int n = readInt();
if ( n == 123 ) {
      c.println( "BINGO");
}
```

En este último ejemplo podemos ver que el ELSE era completamente opcional.

Solución

La solución con este concepto se pone muuuuy sencilla y queda como:

```
static double division (double a, double b) {
      if (b == 0) {
            return 0;
      }
      double r = a / b;
      return r;
}
```

También, podemos utilizar una solución alternativa como:

```
static double division (double a, double b) {
      double r;
      if (b == 0) {
            r = 0;
      }
      else {
            r = a / b;
      }
      return r;
}
```

O una muy similar, pero algo mas inteligente:

```
static double division (double a, double b) {
      double r = 0;
      if (b != 0) {
            r = a / b;
      }
      return r;
}
```

Caso Especial

Hay un problema muy claro, cuando se utilizan muchos IF para distinguir distintos trozos con comparadores de igualdad. Por ejemplo:

```
c.print("Opcion?");
int op = c.readInt();

if (op == 1)
      <instrucciones 1>
else if (op == 2)
      <instrucciones 2>
else if (op == 3)
      <instrucciones 3>
else if (op == 4)
      <instrucciones 4>
else if (op == 5)
      <instrucciones 5>
else if (op == 6)
      <instrucciones 6>
else
      <instrucciones 7>
```

Al observar con detención esto, se puede tornar engorroso, pues si indentáramos quedaría realmente asqueroso. Existe una instrucción que nos puede salvar: **switch**.

```
c.print("Opcion?");
int op = c.readInt();

switch op {
      case 1:
            <instrucciones 1>
            break;
      case 2:
            <instrucciones 2>
            break;
      case 3:
            <instrucciones 3>
            break;
      case 4:
            <instrucciones 4>
            break;
      case 5:
            <instrucciones 5>
            break;
      case 6:
            <instrucciones 6>
            break;
      case:
            <instrucciones 7>
            break;
}
```

en donde los case deben tener valores constantes (es decir, que no sean variables).

Problemas

(a) Se desea escribir un método que calcula la raíz n-ésima de un número considerando que solo se pueden calcular raíces para números > 0 (caso de raíces pares). La firma del método es:

public static double raizN (double base, int raiz)


```
public static double raizN (double base, int raiz) {
      if (raiz % 2 = 0 && base < 0)
            return 0;     // Indica que no se puede calcular

      return Math.pow(base, 1/raiz);
}
```

(b) Escriba un programa que imite el siguiente diálogo:

```
Escriba el número que desea obtener raíz? _
Indique la raíz? _

La raíz <n-ésima> del número es = XXXX.XXX
```

Considere que el término "<n-ésima>" debe indicar:

- "cuadrada" si la raíz es 2
- "cuarta" si la raíz es 4
- "quinta" si la raíz es 5
- "n-ésima" en cualquier otro caso, reemplazando "n" por el valor de la raíz

```
import java.io.*;

public class Raices {

      // Aquí se inserta el método raizN
      public static double raizN (double base, int raiz) {
            if (raiz % 2 = 0 && base < 0)
                  return 0;     // Indica que no se puede calcular

            return Math.pow(base, 1/raiz);
      }

      // Programa principal (solución parte b)
      public static void main (String[] args) {
            Console c = new Console("Raices");

            // Se imprime en pantalla el diálogo
            c.print("Ingrese el número que desea obtener raiz?");
            double x = c.readDouble();

            c.print("Ingrese la raiz?");
            int n = c.readInt();

            // Se calcula la raiz
            double raiz = raizN(x, n);

            // Se escribe el resultado
            switch r {
                  case 2:
                        c.print("La raíz cuadrada ");
                  case 4:
                        c.print("La raíz cuarta ");
                  case 5:
                        c.print("La raíz quinta ");
                  case:
```

```
                        c.print("La raíz " + n + "-ésima ");
                }
                c.println(" del número es = " + raiz);
        }
}
```


Capítulo VII: Ciclos de Programa

Motivación

Escriba un programa que permita calcular el valor de la hipotenusa de triángulos con los valores de los catetos dados por el usuario, según la fórmula de pitágoras.

$$a^2 + b^2 = c^2$$

Para ello, trate de realizar el siguiente diálogo:

```
Cálculo de la Hipotenusa de un Triángulo

Triángulo 1
a?_
b?_
Hipotenusa = ...

Desea Continuar (1=Si/2=No)? _          Suponemos que pone 1

Triángulo 2
a?_
b?_
Hipotenusa = ...

Desea Continuar (1=Si/2=No)? _          Suponemos que pone 2

Se han calculado 2 triángulos
Gracias por usar el programa
```

Algoritmo:

1. Escribir texto de bienvenida en pantalla
2. Iniciar contador del nº de triángulos calculados en 1
3. **Iniciar ciclo de programa**
4. Escribir texto "Triángulo" con el valor del triángulo (1)
5. Pedir y almacenar cateto "a"
6. Pedir y almacenar cateto "b"
7. Calcular e escribir en pantalla el valor de la hipotenusa del triángulo (1)
8. Incrementar el contador
9. Escribir en pantalla la pregunta "Desea Continuar (1=Si/2=No)?"
10. Pedir y almacenar el resultado a la pregunta del punto 9
11. **Si el usuario ingresa 1, volver al punto 4 para el triángulo (contador+1)**
12. Escribir en pantalla la cantidad de triángulos calculados
13. Escribir en pantalla "Gracias por usar el programa"

Este algoritmo se ve un poco más largo de lo común, pero lo más importante son los items 3 y 11 destacados, porque esto definen un ciclo.

Conceptos

Ciclo

*Un **Ciclo** es un trozo de programa que se repite según una **Condición** que evaluada puede entregar verdadera o falsa.*

En Java existe una instrucción para realizar este tipo de código, y se llama **While**.

Sintaxis

```
while (<condición>) {
      <trozo de código que se repite>
}
```

en donde:

<condición> Valor de verdad que debe ser verdadero para que se ejecute el trozo de código escrito dentro del while.

Por ejemplo, el siguiente código se repite **mientras** (while) a sea mayor o igual a 0:

```
Console C = new Console();
int a = 10;
while (a >= 0) {
      C.println("Se repite este texto en pantalla");
      a--; // esto es lo mismo que escribir a=a-1;
}
```

Otra forma de representar un ciclo es con un **for** (por o para) que es más completo que el anterior:

```
for (<inicio>; <condición>; < incremento>) {
      <trozo de código que se repite>
}
```

en donde:

<inicio> Es una instrucción que se ejecuta solo al comenzar el for.
<condición de salida> Es la condición que debe ser TRUE para que el for no termine.
<incremento> Es una instrucción que se ejecuta cada vez que el ciclo es realizado (en cada iteración).

Por ejemplo, el siguiente código se repite **por** (for) 10 veces:

```
Console C = new Console();
for (int a=10; a>0; a--) {
      C.println("Se repite este texto en pantalla");
}
```

Observa la equivalencia directa entre el for y el while de ejemplo, ya que ambos hacen exactamente lo mismo.

Solución

```
import java.awt.*;

public class Triangulos {
      public static void main (String[] args) {
            Console C = new Console();

            C.println("Cálculo de la Hipotenusa de un
                  Triángulo");

            int respuesta = 1;

            int contador = 1;
            while (respuesta == 1) {
                  C.println("Triángulo " + contador);

                  C.print("a?");
                  double a = C.readDouble();
                  C.print("b?");
                  double b = C.readDouble();

                  C.println("Hipotenusa = " +
                        Math.sqrt(Math.pow(a, 2) +
                              Math.pow(b, 2))
                        );

                  contador++;

                  C.print("Desea continuar (1=Si/2=No)?");
                  resultado = C.readInt();
            }

            C.println("Se han calculado " + (contador - 1) +
                  Triángulos");
            C.println("Gracias por usar el programa");
      }
}
```

Problemas

(a) Escribir un programa que pida al usuario 2 número enteros y que luego los multiplique utilizando *SOLO* sumas.

Nota: Recuerda que X por Y es lo mismo que sumar Y veces el valor de X. Se recomienda que utilices un ciclo **for** para resolver este problema

```
import java.awt.*;

public class Producto {
      public static void main (String[] args) {
            // Se crea la consola
            Console c = new Console();

            // Se piden los valores de los operandos
            c.print("Ingrese el operando X?");
            int X = c.readInt();
            c.print("Ingrese el operando Y?");
            int Y = c.readInt();
```

```
            // Se calcula el producto entre X e Y como  sumas
            int producto = 0;
            for (int i=1; i<=Y; i++) {
                  producto = producto + X;
            }

            // Se despliega el resultado
            c.println("X por Y = " + producto);
      }
}
```

(b) **Propuesto**. Haz lo mismo que en el problema anterior (X por Y) pero ahora considerando las potencias (X^Y) y utilizando solo PRODUCTOS. Recuerda que esta operación es lo mismo que multiplicar Y veces el valor de X.

(c) **Desafío**. ¿Puedes hacer lo mismo a b pero solo con sumas? Trata de hacerlo con tus palabras, y luego compáralo con la siguiente solución:

```
import java.awt.*;

public class Potencias {
      public static int productoXY (int X, int Y) {
            int producto = 0;
            for (int i=1; i<=Y; i++) {
                  producto = producto + X;
            }
            return producto;
      }

      public static int potenciaXY (int X, int Y) {
            int potencia = 0;
            for (int i=1; i<=Y; i++) {
                  potencia = productoXY(X, X);
            }
            return potencia;
      }

      public static void main (String[] args) {
            Console c = new Console();

            c.print("Ingrese la base X?");
            int X = c.readInt();
            c.print("Ingrese el exponente Y?");
            int Y = c.readInt();

            c.println("X elevado a Y = " + potenciaXY(X, Y));
      }
}
```


Capítulo VIII: Cadenas de Texto y Literales

Motivación

Hasta ahora conocemos los números y todo. Pero, ¿cómo utilizar las cadenas de caracteres?

Queremos realizar el juego del "repondón" simulando el siguiente diálogo:

```
Hazme preguntas y YO respondo.
P? Esto es un entero
R: NO
P? Eres un computador
R: SI
P? Yo soy un usuario
R: NO
P? Sabes computación
R: SI
```

Concepto

Cadena de Texto

> *Las **Cadenas de Texto** son un grupo de caracteres uno tras otro y que representan palabras, frases u oraciones (gramaticales) con y sin sentido.*

Según esta definición:

- **"Ana"** es una Cadena de Texto.
- **"Yo soy genial"** es una Cadena de Texto.
- **"23499 ldslñññsdfñ"** también es una Cadena de Texto.

Sintaxis

En **Java**, las cadenas de texto tiene un nombre especial y se llaman **String**.

Existen 2 tipos de Strings, los **Literales** y los tipos de dato **String**.

Literales: Son aquellos Strings, representados por cadenas dentro de comillas dobles (" "), y que representan fielmente su contenido. Por ejemplo:

```
c.print("Este es un LITERAL");
```

Claramente el parámetro del método **print** es un literal.

String: Es un tipo de dato especial que permite almacenar literales en variables. Por ejemplo:

```
String a = "Esta es una variable STRING";
c.print(a);
```

Al igual que en el anterior ejemplo, **print** recibe en este caso una variable String en vez de un literal. Mucha atención, que al inicializar una variable String, es necesario hacerlo con un LITERAL.

Algunas funcionalidades útiles de los Strings son:

Concatenación de Strings

Para concatenar o juntar dos o más Strings se utiliza el operador aritmético de adición (+). Por ejemplo:

```
String a = ""; // VACIO
a = a + " primero"
a = a + " segundo"
a = a + " otro" + a + "otro";
```

Esto concatenaría al final en a la frase: "primero segundo otro primero segundo otro".

Lectura de un String desde el Teclado

Para comenzar, debemos saber como leer un String desde la entrada de datos. Para ello utilizaremos el método de la **CONSOLA** llamado **readLine()**:

```
String a;
a = c.readLine();
```

Este ejemplo guardaría lo ingresado por el usuario en la variable **a**, sin importar si son número o letras.

Al igual que los readInt() y readDouble(), readLine() no posee parámetros y su forma general sería la siguiente:

<variable String> = c.readLine();

Largo de un String

Es muy util conocer cuántos caracteres posee un String (variable o literal). Para ello existen un método dentro de la clase **String** que se utiliza, llamado **length**:

```
String a = "Hola mundo";
c.print (a + " posee " + a.length() + " caracteres");
```

Este ejemplo mostraría en la consola "*Hola mundo* posee 10 caracteres"

Como podemos ver en el ejemplo, length se utiliza con la variable **punto** el método length (sin parámetros). Es decir:

<variable String>.length();

Obtención de un Carácter

Al igual que para el largo, existe la forma de sacar 1 sola letra del String. Esto se realizar con el método de la clase String **charAt(i)** en donde **i** es el i-ésimo menos 1 carácter de la cadena. Por ejemplo:

```
String alfabeto = "abcdefghijklmnopqrstuvwxyz";
c.println("La letra 3 es " + alfabeto.charAt(3));
```

Como podemos ver, este ejemplo muestra en la consola la frase: "La letra 3 es d". Ooooops. ¿Qué está mal? No hay nada malo. Lo que pasa es que las posiciones comienzan desde 0, es decir, si quiero la 5 letra, esta sería **alfabeto.charAt(4)** y no **alfabeto.charAt(5)** como sería lo común.

Su forma general queda como:

<variable String>.charAt(<posición i>);

Búsqueda de un Carácter o Texto

Para buscar un trozo de texto o carácter dentro de otro se utiliza el método **indexOf("texto")** el cual retorna la posición i-ésima menos uno (es decir retorna el índice del carácter correcto dentro del String). Por ejemplo:

```
String s = "Texto contiene un ; dentro del cuerpo";
c.println("En " + s.indexOf(";") + " hay un punto y coma");
```

Este ejemplo retorna "En 18 hay un punto y coma". Pero si contamos el número desde 1, es el 19avo carácter.

Su forma general que busca la primera ocurrencia de un texto, es:

<variable String>.indexOf(<texto buscado>);

Nota: Si el String buscado no está dentro del String s, la posición retornada es –1.

Existen algunas variantes de esté método. Para buscar una ocurrencia de un texto a partir de una posición específica, se utiliza:

<variable String>.indexOf(<texto buscado>, <posición a buscar>);

Con esto entonces podemos buscar desde una posición específica. Por ejemplo:

```
String s = "esta es una casa";
int n = s.indexOf("es");   // es lo mismo decir: s.indexOf("es", 0)
```

Esto retorna la posición 0, pero si buscamos desde otra posición

```
int n = s.indexOf("es", 2);
```

Retornará la posición 5, porque a partir de la posición 2, la primer ocurrencia de "es" está en la posición 5.

Nota: Si anteponemos la palabra "last" a ambas formas del método, es decir, "lastIndexOf", podemos obtener no la primera ocurrencia, si no que la última ocurrencia del texto buscado.

Obtención de Trozos de un String

Por último para la clase, veremos como sacamos un trozo de un String, es decir, algo como sacar un pedacito que nos sirva. El siguiente ejemplo sacar lo que va entre sexta y la undécima posición es:

```
String s = "Este string está completito";
String a = s.substring(5, 11);
c.println(a);
```

Este ejemplo pone en pantalla la palabra **string**. Su forma general queda:

<variable String>.substring(<pos inicial>, <pos final>);

Nótese que NO INCLUYE la posición final, es decir, saca hasta el último caracter antes de **<pos final>**.

También existe una forma adicional de usarlo y es sin indicar el largo, es decir:

<variable String>.substring(<pos inicial>);

De esta forma estaríamos pidiendo desde la posición inicial hasta el final del String.

Comparación de Strings

Los strings no pueden ser comparados como los números con operadores lógicos tradicionales. Para ello posee sus propios comparadores, métodos de la clase String:

Existen 2 métodos que nos ayudan a comparar strings:

<variable String 1>.equals(<variable String 2>);

Este método retorna TRUE si los strings son EXACTAMENTE iguales (incluyendo mayúsculas y minúsculas[11]). Si los strings difieren en algo (largo o caracteres) el resultado es FALSE.

La otra forma compara los strings según la tabla ASCII:

<variable String 1>.compareTo(<variable String 2>);

Este método retorna

[11] A esto se le llama Case

- número > 0 si el string 1 es MAYOR que el string 2
- 0 si el string 1 es IGUAL que el string 2
- número < 0 si el string 1 es MENOR que el string 2

¿Cuándo un string es mayor o menor que otro?

Un string es mayor que otro cuando en el primer caracter que difieren, el caracter del string 1 es mayor según ASCII que el string 2. Según la tabla ASCII de valores internacionales, que representa cada caracter con un número entero entre 0 y 255, nos dice que:

- Los números (0 .. 9) se representan por códigos entre el 48 y 57 (respectivamente)
- Las mayúsculas (A .. Z) se representan por códigos entre el 65 y 90
- Las minúsculas (a .. z) se representan por código entre el 97 y 122
- Todos los demás caracteres poseen distintos códigos

Entonces, un caracter es mayor que otro cuando su código ASCII es mayor.

Solución

Recordando un poquito tenemos que imitar el diálogo siguiente:

```
Hazme preguntas y YO respondo.
P? Esto es un entero
R: NO
P? Eres un computador
R: SI
P? Yo soy un usuario
R: NO
P? Sabes computación
R: SI
```

Pero además, nos falta saber como responde el computador:

- Si la pregunta termina con vocal, el computador responde **NO**.
- En cualquier otro caso responde **SI**.

Ahora veamos la solución:

```
Console c = new Console();
c.println("Hazme preguntas y YO respondo:");
while (true) {
      c.print("P? ");
      String pregunta = c.readLine();
      if ("aeiou".indexOf(pregunta.charAt(pregunta.length()-1))>= 0)
            c.println("R: NO");
      else
            c.println("R: SI");
}
```

Oooops. Creo que es demasiado complicado ver la línea del **if** por entero. Analicemos lo que vamos haciendo paso a paso y separado por líneas distintas (equivalente al IF de arriba):

```
// Obtenemos el largo del STRING
int L = pregunta.length();

// Calculamos el ultimo caracter del string
L = L - 1;

// Sacamos el último carácter
String ultimo = pregunta.charAt(L);

// Vemos si es vocal
int vocal = "aeiou".indexOf(ultimo);
if (vocal >= 0)                      // Solo si ultimo es vocal
      c.println("R: NO");
else
      c.println("R: SI");
```

Problemas

(a) Se desea escribir un programa que solicite una línea de texto al usuario y que simplemente la despliegue en forma inversa, es decir, la de vuelta.

```
// Primero, pediremos el texto al usuario
c.println("Escribe un texto para invertir?");
String texto = c.readLine();

// Ahora tomamos el texto y lo invertimos
// guardándolo en una nueva variable
String invertido = "";                  // Se inicia vacío
for (int i=0; i<texto.length(); i++) { // Ciclo de recorrido texto
      // Se va guardando al revés, caracter a caracter
      invertido = texto.charAt(i) + invertido;
}

// Se muestra en pantalla
c.println("El texto invertido es: " + invertido);
```

Es muy importante notar que esta no es la única solución posible, porque si somos astutos, podemos ir mostrando en pantalla caracter a caracter en vez de utilizar una nueva variable:

```
// Primero, pediremos el texto al usuario
c.println("Escribe un texto para invertir?");
String texto = c.readLine();

// Ahora lo invertimos
for (int i=texto.length()-1; i>=0; i--) {
      c.print(texto.charAt(i));
}
```

(b) **Propuesto.** Escribir un programa que cuente cuántas palabras son verbos (en infinitivo) en un texto ingresado por el usuario. Recuerde que los infinitivos terminan TODOS en una vocal + "r".

(c) **Propuesto**. ¿Podría escribir un método que entregue aleatoriamente un String de **n** caracteres?. La firma del método es:

public static String textoAzar (int n);

Nota: Utiliza un string que tenga TODAS las letras del abecedario.

(d) **Propuesto.** Escribe un programa que, utilizando e indicando los patrones de programación usados, pueda simular el siguiente diálogo:

```
Contador de letras en una frase
Escriba la frase que desea contar? Esta es la frase
a = 3 veces
e = 3 veces
f = 1 veces
l = 1 veces
r = 1 veces
s = 2 veces
t = 1 veces
7 letras usadas
Escriba la frase que desea contar? _
```

Note que lo que hace el programa es contar cuántas veces se repite una letra del abecedario en la frase que ingresa el usuario, y cuenta cuántas letras distintas tiene la frase.

```
Console c = new Console();
c.println("CONTADOR DE LETRAS");
// Ciclo de programa,
// nada que ver con Patrones
while (true) {
      c.print("Escriba la frase?");

      // PATRON DE LECTURA DE DATOS
      String frase = c.readLine();

      // Trasnformamos toda la frase a minúsculas
      frase = frase.toLowerCase();
      int cont_letras = 0;

      for (int i=0; i<abcdario.length(); i++;) {

            // Sacamos la letra i-ésima del abcdario
            String letra = abcdario.charAt(i);

            int cont_veces = 0;

            for (int j=0; j<frase.length(); j++) {
                  // Verificamos que sea la letra escogida
                  if (frase.charAt(j).equals(letra)) {
                        cont_veces++;
                  }
            }

            // Una vez que cuenta cuántas veces se repite la letra
            // la mostramos en la Console si es que es != 0.
            if (cont_veces>0) {
                  c.println(letra + " = " + cont_veces +
                        " veces");
                  cont_letras++;
            }
      }
```

```
        // Ahora decimos cuántas letras encontramos en la frase
        c.println("Hay " + cont_letras + " letras en la frase");
}
```


Capítulo IX: Patrones de Programación

Motivación

Los programas no se parten a desarrollar desde cero. Existen "elementos" que permiten traducir los algoritmos mentales en códigos de programación y que son comunes entre una solución y otra. A ellos se les llama *Patrones de Programación*.

Concepto

> *Un Patrón de Programación es un elemento de programa ya conocido que se puede utilizar como parte de la solución de muchos problemas.*

Como es normal en lo que son los patrones, serían elementos que se repiten en distintas soluciones. A continuación veremos algunos patrones básicos más utilizados.

Patrón de Acumulación

Se utiliza para realizar cálculos como la suma de varios valores acumulados en distintas iteraciones de un ciclo:

```
suma = val1 + val2 + val3 + ... + valn
producto = fac1 * fac2 * fac3 * ... * facm
```

La forma general de este patrón es:

```
<tipo> <variable> = <valorinicial>;
...
while (<condicion>) {
        ...
        <variable> = <variable> <operador> <expresión>;
        ...
}
```

Un ejemplo concreto sería realizar el siguiente diálogo:

```
Valores
? 4.0
? 5.0
? 6.0
? 0.0
Contador = 3
```

La solución del programa anterior, marcando en negritas aquellas que corresponden al patrón de acumulación, sería:

```
c.println("Valores");
int cont = 0;
c.print("? ");
double nota = c.readDouble();
while (nota != 0.0) {
```

```
        cont = cont + 1;
        c.print("? ");
        nota = c.readDouble();
}
c.println ("Contador = " + cont);
```

Patrón de Lectura de Datos

Este patrón involucra las acciones de lectura de valores o datos. Es bastante sencillo y su forma general es:

```
<tipo> <variable>;
...
<variable> = c.read<Tipo>();
...
```

Es muy útil porque es necesario para la interacción con el usuario. Ejemplo:

```
c.print("Ingrese su nombre? ");
String nombre = c.readString();
c.println("Hola " + nombre);
```

Patrón de Recorrido de Intervalo

El patrón de recorrido de intervalo es aquél que permite realizar un ciclo de repetición entre un intervalo de valores enteros. Su forma general es:

```
int i = <valor inicial>;
while (i <= <valor final>) {
        . . .
        i++;
}
```

y en forma de **for** sería:

```
for (int i = <valor inicial>; i <= <valor final>; i++) {
        . . .
}
```

Un ejemplo sencillo es la resolución de una sumatoria de la forma:

$$\sum_{i=0}^{n} i$$

La solución a este problema es bastante sencilla con la forma **for**:

```
int suma = 0;
for (int i = 0; i <= n; i++) {
        suma += i;
}
```


Problema

(a) En el siguiente problema asocie con números las líneas de código que pertenecen a un patrón de programación. Use la siguiente simbología:

1. Patrón de Acumulación
2. Patrón de Lectura de Datos
3. Patrón de Recorrido de Intervalo

```
Console c = new Console();
c.println("Ingrese las notas de los 100 alumnos.");
double suma = 0;
for (int i = 0; i < 100; i++) {
        c.print("Alumno " + i + "?");
        double nota = c.readDouble();
        suma += nota;
}
c.println("El promedio del curso es: " + suma/100);
```

Solución

```
        Console c = new Console();

        c.println("Ingrese las notas de los 100 alumnos.");

(1)     double suma = 0;

(3)     for (int i = 0; i < 100; i++) {

              c.print("Alumno " + i + "?");

(2)           double nota = c.readDouble();

(1)           suma += nota;

        }

        c.println("El promedio del curso es: " + suma/100);
```

Complemento a la Clase

Es muy importante recalcar que estos patrones son los básicos que existen. También los hay en muchas ocasiones y tipos distintos dependiendo de la solución.

Probablemente, a través del curso encuentres otros patrones de programación. Por ello, búscalos y encuéntralos, pues te servirán para todos tus programas (incluso las tareas).

Capítulo X: Arreglos y Matrices

Motivación

Hasta ahora hemos hecho programas que pueden funcionar hasta con 3 valores distintos, y no hemos podido guardar más información, a veces variables, ni nada.

Imaginemos el siguiente diálogo:

```
Votaciones Mundiales por la Paz
Candidatos = 10

Emisión de votos:
Voto? 1
Voto? 3

... (El diálogo continúa mientras no se ponga 0)

Voto? 0

Resultados de las elecciones:
Candidato 7 = 58 votos
Candidato 3 = 55 votos
Candidato 1 = 33 votos

Fin del programa.
```

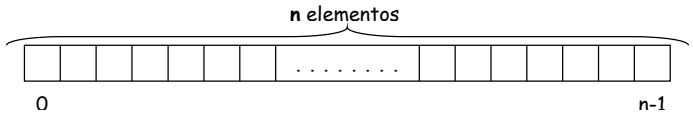
Una forma sencilla sería declarar 10 variables que guarden los votos de cada candidato pero ¿qué sentido tiene cuando el número de candidatos es 100 o 1.000?. Hoy, encontraremos otra solución.

Conceptos

Arreglos:

> *Lista de elementos (espacios en memoria) de un mismo tipo que son referenciados por una misma variable subindicada a partir de 0 (primera posición) hasta el número de elementos menos 1 (n-1 es la última posición).*

En efecto. Un **Arreglo** puede ser muy útil cuando se utilizan este tipo de problemas ya que permiten un trabajo mucho más *dinámico* que utilizando variables estáticas. También, es un ahorro de espacios de memoria cuando se almacenan.



Es muy importante ver que los arreglos parten desde **0** y terminan en **n-1** (si el largo es de **n** por supuesto). Por ejemplo, un arreglo de largo 10 posee las "casillas" 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9.

Dimensiones de los Arreglos

Los arreglos pueden ser uni-dimensionales o multi-dimensionales. Para nosotros solo nos serán aplicables los arreglos uni-dimensionales (a los que llamaremos solo **arreglos**) y bi-dimensionales (a las que llamaremos **matrices**).

Sintaxis

Los arreglos no poseen un tipo especial para ser declarados. Lo único que hay que hacer es agregar un modificador especial a cada uno de los tipos ya definidos.

Por ejemplo:

```
int a[];
a = new int[100]; // Arreglo de 100 enteros

double[] b;
b = new double[10]; // Arreglo de 10 reales

String s[][];
s = new String[23][50]; // Matriz de 23 filas x 50 columnas Strings.

MiTipo[] c;
c = new MiTipo[3]; // Arreglo de 3 objetos de MiTipo
```

Como podemos ver hay formas distintas (pero equivalentes) para crear arreglos. La forma (o formas) general es:

```
<tipo>[] <var>;
<var> = new <tipo>[<largo>];

o

<tipo> <var>[];
<var> = new <tipo>[<largo>];
```

o

```
<tipo>[][] <var>;
<var> = new <tipo>[<filas>][<cols>];

o

<tipo> <var>[];
<var> = new <tipo>[<filas>][<cols>];
```

Cualquiera de las dos formas es correcta y hacen exactamente lo mismo. La diferencia está en las combinaciones que se pueden realizar cuando se programa. Por ejemplo:

```
int[] a, b;
```

En este caso se declaran 2 arreglos de enteros.

```
int a[], b;
```

En este otro caso se declara 1 arreglo de enteros (**a**) y un entero (**b**).

Pero cuando queremos leer un elemento, o almacenar algo en una posición del arreglo, se utilizan subíndices enteros para indicar que queremos sacar:

```
String s[] = new String[50] ;
s[3] = "Esta es la posición 4";
s[10] = "Esta es la posición 11";
c.println("Se imprime la 15ava posición del String: " + s[14]);
```

Un ejemplo de utilización de arreglos puede ser realizar una lectura de texto por líneas, es decir:

```
Escriba el texto de maximo 100 líneas que desea almacenar en el
archivo. Para finalizar ponga un punto aislado (.)

> Este es un texto
> que es ingresado por el usuario
> a través del teclado.
> ...
(Llegar a 100)
> última línea.

Listo. Fue guardado en archivo.txt
```

Ok. Una forma sencilla sería hacerlo que cada vez que escribiera una línea la enviáramos al archivo, pero si lo hiciéramos con arreglos solo abriríamos el archivo al final:

```
Console c = new Console();
c.println("Escriba el texto de maximo 100 líneas que desea almacenar
         en el archivo. Para finalizar ponga un punto aislado(.)  ");
String texto[] = new String[100];
int i = 0;
while(i < 100) {
       c.print("> ");
       texto[i] = c.readLine();
       if (texto[i].equals(".")) break;
       i++;
}

// Ahora se graba en el archivo
PrintWriter pw = new PrintWriter(new FileWriter( "archivo.txt"));
for(int j = 0; j < i; i++) {
      pw.println() = texto[j];
}
pw.close();
c.println("Listo. Fue guardado en archivo.txt");
```

Otro ejemplo, pero ahora con matrices:

```
Ingrese las notas de los 130 alumnos del curso:
```



```
Alumno 1:
P1? _
P2? _
P3? _

...

Alumno 131:
P1? _
P2? _
P3? _

El mejor promedio fue el alumno 57 con un promedio de control X.X.
```

A diferencia del problema anterior, aquí se realiza una búsqueda después de ingresado los datos a la matriz. El código sería:

```
Console c = new Console();
c.println("Ingrese las notas de los 110 alumnos del curso: ");
double notas[][] = new double[110][3];
for(int i = 0; i < 110; i++) {
       c.println("Alumno "+ i);
       for(int j = 0; i < 3; i++) {
              c.print("P" + j + "? ");
              notas[i][j] = c.readDouble();
       }
}

// Ahora se busca el mejor promedio
double maxProm = 0;        // Nadie tiene peor promedio
double mejorAlumno = -1;  // Nadie
for (int i=0; i<110; i++) {
       double promedio = (notas[i][0] + notas[i][1] + notas[i][2])/3;
       if (maxProm < promedio) {
              maxProm = promedio;
              mejorAlumno = i;
       }
}

c.println("El mejor promedio fue el alumno " + mejorAlumno +
          " con un promedio de control " + maxProm);
```

Como podemos ver, la utilización es completamente similar en ambos casos.

¿Qué pasa si necesito un arreglo en un método? Bueno este problema es bastante sencillo, ya que el "tipo del arreglo" está definido tanto por el tipo de dato que almacena + los corchetes que indican que es un arreglo. Siguiendo con esta regla:

1. Para pasar por parámetro un arreglo o matriz:

```
public void producto (double[] arreglo, double[][] matriz)
```

2. Para retornar un arreglo o matriz:

```
public double[] sumaArreglo (...)
```

```
public double[][] sumaMatriz (...)
```

No hay mucha diferencia.

Para aclarar más los conceptos hasta aquí vistos, podemos tratar de hacer un método que permita trasponer una matriz, entregándole por parámetro la matriz original y las dimensiones de ella:

```
public double[][] trasponer (double[][] M, int filas, int cols) {
      double valTemp = 0;
      for (int i = 0; i < filas; i++)  {
            for (int j = 0; j < i; j++)  {     // Es óptimo
                  valTemp = M[i][j];
                  M[i][j] = M[j][i];
                  M[j][i] = valTemp;
            }
      }
      return M;
}
```

Hecho. ♍

Otra nota interesante es saber el largo de un arreglo sin tener la necesidad de conocerlo previamente. Por ejemplo:

```
public double promedioNotas (double[] notas) {
      double suma = 0;
      for (int i = 0; i < notas.length; i++)
            suma += notas[i];
      return (suma / notas.length);
}
```

En este caso **notas** trae una lista con valores numéricos, pero no sabemos cuánto. Para ello utilizamos .**length** sin paréntesis (a diferencia de length() de String obviamente).

Solución al Problema

Recordemos el diálogo del problema:

```
Votaciones Mundiales por la Paz
Candidatos = 10

Emisión de votos:
Voto? 1
Voto? 3

... (El diálogo continúa mientras no se ponga 0)

Voto? 0

Resultados de las elecciones:
Candidato 7 = 58 votos
Candidato 3 = 55 votos
Candidato 1 = 33 votos

Fin del programa.
```


La solución a este problema es muy similar al problema de las notas:

```
Console c = new Console();
c.println("Votaciones Mundiales por la Paz");

c.println("Candidatos = 10");
int votos[] = new int[10];

// Se inicializan los votos en 0
for (int i=0; i<votos.length; i++)
      votos[i] = 0;

// Se ejecuta la votación
c.println("Emisión de Votos:");
while (true) {
      c.print("Voto?");
      int candidato = c.readInt();
      if (candidato = 0)
            break;
      votos[candidato]++;
}

// Se inicializan al mayor como el primero
int[] maximas = new int[3];
for (int i=0; i<maximas.length; i++)
      maximas[i] = 0;

// Ahora se buscan los mejores candidatos
for (int i=0; i<votos.length; i++) {
      // Buscamos si es mayor que el primer lugar
      if (votos[i] >= votos[maximas[0]]) {
            maximas[2] = maximas[1];
            maximas[1] = maximas[0];
            maximas[0] = i;
      }
      // Buscamos si es el segundo
      else if (votos[i] >= votos[maximas[1]]) {
            maximas[2] = maximas[1];
            maximas[1] = i;
      }
      // Buscamos si es el tercero
      else if (votos[i] >= votos[maximas[2]]) {
            maximas[2] = i;
      }
}

// Se imprimen los lugares
c.println("Resultados de las elecciones:");
c.println("Candidato "+maxima[0]+"="+votos[maxima[0]]+" votos");
c.println("Candidato "+maxima[1]+"="+votos[maxima[1]]+" votos");
c.println("Candidato "+maxima[2]+"="+votos[maxima[2]]+" votos");

c.println("Fin del programa");
```

Otra opción hubiese sido haber declarado 3 variables para las máximas votaciones, modificando solo el trozo de código por el siguiente:

```
...
// Se inicializan al mayor como el primero
```

```
int lugar1 = 0;
int lugar2 = 0;
int lugar3 = 0;

// Ahora se buscan los mejores candidatos
for (int i=0; i<votos.length; i++) {
      // Buscamos si es mayor que el primer lugar
      if (votos[i] >= votos[lugar1]]) {
            lugar3 = lugar2;
            lugar2 = lugar1;
            lugar1 = i;
      }
      // Buscamos si es el segundo
      else if (votos[i] >= votos[lugar2]) {
            lugar3 = lugar2;
            lugar2 = i;
      }
      // Buscamos si es el tercero
      else if (votos[i] >= votos[lugar3]) {
            lugar3 = i;
      }
}

// Se imprimen los lugares
c.println("Resultados de las elecciones:");
c.println("Candidato "+lugar1+"="+votos[lugar1]+" votos");
c.println("Candidato "+lugar2+"="+votos[lugar2]+" votos");
c.println("Candidato "+lugar3+"="+votos[lugar3]+" votos");
...
```

Pero la elegancia va por dentro. ♍

Problemas

Una multitienda de Burundí, El Negro Judío, se ha dado cuenta de que las utilidades que recibe mensualmente no son tan buenas como lo eran antes.

Un estudio con los especialistas económicos de la tribu ha arrojado que la clave de las utilidades está en la atención que los vendedores daban a los clientes. En una encuesta anónima se dieron cuenta de que los 25 vendedores de la tienda no tenían incentivos para atender mejor a los clientes porque poseían un sueldo fijo.

Entonces, al genial Yoghu-Rtuh Mghe, se le ocurrió dar incentivos dependiendo de lo que venda cada vendedor. Para ello le piden a usted algunos favorcitos:

(a) Desarrolle un programa que simule el siguiente diálogo:

```
Tienda de 25 vendedores
Porcentaje de comisión? _       (Para todos los vendedores es igual)

Inicio de Ventas:

Vendedor? _
Monto vendido? _

...
```



```
Vendedor? 0
Fin de Ventas

Resumen de comisiones
Vendedor 1 = $ XXXXX
...
Vendedor 25 = $ XXXXX
```

Solución

```
Console c = new Console();

c.println("Tienda de 25 vendedores");
int nVendedores = 25;

// Se declara el arreglo con lo vendido por el vendedor
int[] ventas = new int[nVendedores];
for (int i=0; i<ventas.length; i++)
      ventas[i] = 0;

// Se pide el % de comisión
c.print ("Porcentaje de comisión? ");
double comision = c.readDouble();

// Ciclo de venta de la tienda
c.println("Inicio de Ventas:");
while (true) {
      c.print("Vendedor?");
      int v = c.readInt();
      if (v == 0)
            break;
      c.print("Monto vendido?");
      ventas[v - 1] += c.readInt();
}
c.println("Fin de Ventas");

// Calculo de comisiones por vendedor
c.println("Resumen de comisiones");
for (int i = 0; i < ventas.length; i++) {
      c.print ("Vendedor " + (i + 1) + " = $ ");
      c.println ((ventas[i] * comision / 100));
}
```

(b) Hágalo ahora con un número de vendedores variable, es decir, que el usuario ingrese el número de vendedores, ya que a través del tiempo no es seguro que siempre tengan 25 vendedores.

Solución

Bueno, de la forma en que se hizo en la parte (a) la solución es bastante sencilla. Veremos solo un trozo de lo que necesitamos para ver la diferencia:

```
Console c = new Console();

c.print("Número de Vendedores?");
int nVendedores = c.readInt();

// Se declara el arreglo con lo vendido por el vendedor
int[] ventas = new int[nVendedores];
```

```
for (int i=0; i<ventas.length; i++)
      ventas[i] = 0;

...
```

Problemas Propuestos

Escriba un problema de control de inventario:

1. Tiene un archivo de entrada (inventario.in) en donde guarda una lista con los códigos de los productos (5 caracteres) y sus stocks iniciales (resto de la línea). Por ejemplo:

```
39578 27
84990 32
12948 40
38380 0
92329 109
.
.
.
```

   Note el espacio que hay entre el 5to carácter y el stock del producto

2. Al final del procesamiento del programa tiene que guardar los nuevos stocks en un archivo (inventario.out) con la misma estructura anterior.

3. Proceso de la siguiente forma:

```
Control de Inventario

Inventario Inicial:
  Producto XXXX = 5
  Producto XXXX = ...

Movimientos:
  Tipo (1 = entrada, 2 = salida, 0 = terminar)? _
  Código del producto? _
  ...
  Tipo (1 = entrada, 2 = salida, 0 = terminar)? 0
Fin del día.

Inventario Final:
  Producto XXXX = 5
  Producto XXXX = ...

El inventario fue guardado en inventario.out
```

Importante

Cuando tu creas un arreglo con una clase pre-definida, no puedes llegar y crear el arreglo sin crear cada uno de los objetos (casillas) del arreglo. Por ejemplo con la clase **Complejo**:

```
Complejo[] c = new Complejo [20];       ▫ Casillas del Arreglo
for (int i = 0; i < c.length; i++) {
      c[i] = new Complejo();            ▫ Constructor de la Clase
}
```

Capítulo XI: Recursividad

Motivación

Sabemos perfectamente calcular el factorial de un número en forma secuencial. Además, podemos verlo en forma de método estático (dentro de la clase principal del programa):

```
static int factorial (int n) {
	int resultado = 1;
	for (int i = 1; i <= n; i++)
		resultado = resultado * i;
	return resultado;
}
```

¿Se podría solucionar este mismo problema pero sin utilizar un ciclo explícito como **for** o **while**?

Conceptos

Recursividad

> *La Recursividad es un patrón de programación que permite llamar sucesivamente la misma funcionalidad o método, con una condición de salida y variando su parámetro de entrada, para llegar al resultado final.*

El concepto de **Recursividad** es un poco complejo de entender de buenas a primeras, ya que es un patrón bastante abstracto cuando se trata de visualizarlo físicamente.

Veamos un ejemplo.

Generar una secuencia (String) de números enteros a partir de 1 y hasta llegar a **n** (dado por el usuario (como parámetro) en forma recursiva e inversa, es decir como { n, n-1, n-2, ..., 2, 1 }.

El encabezado del método quedaría como:

```
static String concatenaRecursivo (int n) {
	// Recibe como parámetro el n del máximo en el conjunto.
```

Para realizar este problema, se debe pensar en el caso inicial, es decir cuando n = 1 (menor valor posible para n):

```
	// Caso base retornamos solo el uno.
	if (n == 1)
		return "1";
```

Ok… ahora que el caso base está realizado, suponemos el paso k-1 cierto, es decir que la llamada de **concatenaRecursivo(k-1)** nos retorna el conjunto de valores entre k-1 y 1 concatenados. Esto quiere decir que debemos concatenarle k.

Ahora bien, como no podemos sacar un k, sabemos por hipótesis inductiva que:

concatenaRecursivo(n) n + concatenaRecursivo(n-1)

Esto se ve si hubiésemos usado **n** en vez de **k** en la hipótesis.

En Java se utiliza la hipótesis para **n-1** cierta de la misma forma, quedando:

```
	// Caso general con hipótesis inductiva
	return n + concatenaRecursivo(n-1);
} // Fin del método
```

y, entonces, en una visión general nuestro método quedaría como:

```
// Encabezado del método
static String concatenaRecursivo (int n) {

	// Caso base retornamos solo el uno.
	if (n == 1)
		return "1";

	// Caso general con hipótesis inductiva
	return n + concatenaRecursivo(n-1);

}
```

¡¡¡Es recursivo!!!

Y la llamada sería:

```
String serie = concatenaRecursivo(45);
```

(Guarda en **serie** el resultado "45 44 43 42 41 40 ... 3 2 1")

Entonces, resumiendo lo aprendido, para hacer un método recursivo se necesita de 2 pasos muy importantes:

1. Conocer el caso (o casos) base, es decir, la condición de salida de la recursividad.
2. El paso general que se ejecuta procesando y llamando a la siguiente iteración.

Esto es muy similar a los ciclos.

Solución al Problema

Para solucionar el problema, definamos el caso base y el caso general.

Caso Base:

La condición base para el problema del factorial, bastaría decomponer la fórmula:

N! = N * (N-1) * (N-2) * ... * 1! * 0!

Es claro ver que la condición de salida sería **0! = 1** ya que sin esta definición no termina la serie.

Caso General:

Si realizamos una equivalencia matemática de la fórmula obtendremos el caso general:

N! = N * (N-1)!

Bueno, esta equivalencia es muy común en soluciones matemáticas de problemas de este tipo, ya que utilizamos la inducción para suponer que conocemos **(N-1)!** y solo lo multiplicamos con nuestro factor incógnita **N**.

Entonces, con estos dos pasos claramente definidos, podemos dar la solución al problema:

```
static int factorialRecursivo (int n) {

      // Caso base
      if ( n == 0 )
            return 1;

      // Caso general
      return n * factorialRecursivo(n-1);

}
```

Un concepto muy importante para problemas matemáticos.,

Problema Propuesto

(a) Escriba un método recursivo para resolver la siguiente sumatoria:

$$\prod_{i=0}^{n} i * e^{i}$$

En donde **n** es el valor entregado por el usuario (como parámetro). Recuerde que la exponencial está en la clase matemática y funciona como **Math.exp(i)**.

(b) Escriba el programa que utilice lo anterior y simule el siguiente diálogo:

```
Ingrese el número de iteraciones para la suma
n? 35
La sumatoria vale = ...
```


Capítulo XII: Clases y Objetos

Motivación

En Java solo existen números enteros y reales. Realmente es algo interesante modelar como se podrían realizar operaciones con números complejos.

Para este problema es necesario primero definir conceptos generales para la utilización de tipos especiales de datos.

Conceptos

Clase

> *Se define como **Clase** a una estructura de programación que permite definir características y funcionalidades especiales a un grupo de objetos que poseen el mismo comportamiento (en la vida real).*

Es así como **2+i**, **3.5+9i**, **(1,4.5)i** que son notaciones para los números complejos pertenecen a la **CLASE COMPLEJO**.

Características: Poseen una parte real y una parte imaginaria, ambas partes son números reales.

Funcionalidades: Pueden ser escritos en forma binomial o en forma de vector, se suman, se restan, se dividen, se multiplican, etc.

Objeto

> *Se define como **Objeto** a una referencia específica de un elemento que de cierta clase, es decir, es un elemento que cumple con las características de una clase.*

Como podemos observar, si **COMPLEJO** es una clase, cada uno de sus números complejos serían objetos de esa clase, es decir, **2+i** es un objeto.

Un ejemplo mucho más concreto que permite ver la diferencia entre CLASE y OBJETO es el clásico ejemplo de **PERSONA**.

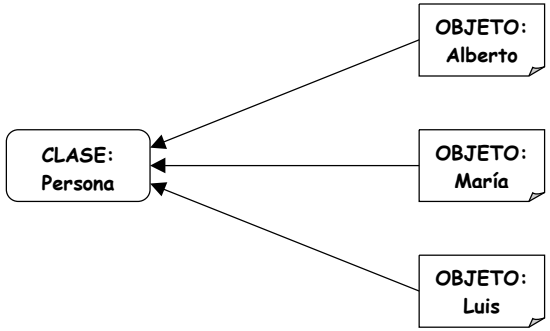
Persona es una clase, pues agrupa a una serie de individuos que cumplen con un conjunto de características comunes (tienen sentidos, son bípedos, de dos sexos, etc) y pueden realizar una cantidad de acciones (funcionalidades) comunes (caminar, hablar, saludar, comer, etc).

Ahora bien, ¿qué diferencia una persona de otra?. Enumeremos algunas de las características que hacen a cada individuo distinto de otro: Color de Pelo, Color de Ojos, Color de Piel, Talla, Peso, Idioma, Nombre, entre otras.

Con estas definiciones, veremos un **Objeto** de la clase **Persona**:

```
Color de Pelo: Castaño
Color de Ojos: Verde
Color de Piel: Blanca
Talla: 1.76 mts.
Peso: 73 Kgs.
Idioma: Español
Nombre: Alberto Díaz
```

Como podremos ver, Alberto Díaz posee las características de cualquier persona, es decir, posee color de pelo, de ojos, de piel, talla, peso, idioma y nombre, pero es un **Objeto** de la clase **Persona** porque podemos *distinguirlo* del grupo de personas.



En la figura anteriormente desplazada, podemos observar definidos 2 objetos adicionales al que anteriormente llamamos **Alberto**: **María** y **Luis**. Los tres objetos son de tipo persona porque poseen características comunes como ser bípedos y seres humanos (para ser de tipo **Persona**), pero cada uno se distingue de otro por tener distintos nombres, tallas, pesos, color de piel, etc.

Creación y Referencia de Objetos en Java

Para crear un objeto (de tipo **Persona** para ejemplos), se utiliza la sentencia **NEW**.

```
Persona alberto = new Persona();
```

En esta simple línea le indicamos al ordenador que nos cree un **Objeto** de tipo **Persona** y le llamaremos (a la variable) **alberto**.

Ahora bien, si existiera que todas las personas pueden caminar (funcionalidad), esto lo representaríamos como:


```
alberto.caminar();
```

Como podemos ver, para llamar una funcionalidad de la clase del tipo de la variable, se utiliza el separador "." (punto).

Es muy importante decir que las funcionalidades de una clase son definidos como **métodos**, es decir, pueden recibir parámetros y pueden retornar valores. Por ejemplo:

```
String frase = alberto.hablar("hola maria");
maria.escuchar(frase);
```

En este último ejemplo, se está simulando una conversación entre Alberto y María. Otro ejemplo de esto se puede ver más claramente con los números complejos:

```
// Declaramos el complejo 3+5i
Complejo a = new Complejo(3, 5);

// Declaramos el complejo -1+9i
Complejo b = new Complejo(-1, 9);

// Declaramos el complejo c como la suma de los complejos
// a y b
Complejo c = a.sumar(b);

// Imprime en pantalla la forma binomial del complejo c
c.println(c.binomial());
```

Es así como nace un nuevo concepto:

Referencia:

*Una **Referencia** a un objeto es la variable que indica el objeto deseado.*

En el ejemplo de los complejos, a, b y c son referencias a objetos de tipo complejo. En el ejemplo de las personas, alberto y maria son los nombres de las variables que son las referencias.

Declaración de una Clase

La declaración de una clase no es muy distinta a lo que ya hemos hecho para poder compilar y ejecutar nuestros programas y ejemplos.

La forma general de declaración de una clase es la siguiente:

```
[public] class <nombre de la clase> {
      // Variables de instancia
      [declaración de variables de instancia]

      // Constructor de la clase
      public <nombre de la clase> ( [parámetros] ) {
            <instrucciones>
      }
```

```
        // Inicio de las funcionalidades
        [public] <tipo de dato del retorno> <nombre del método>
                        ( [parámetros] ) {
                <instrucciones>
        }
}
```

Como podemos ver se parece mucho al programa principal. La diferencia se encuentra en los modificadores de los métodos que utilizamos. Notemos la ausencia del modificador **static** frente a la definición de los métodos.

Por ejemplo, usemos la clase **Persona**:

```
public class Persona {
        // Variables de Instancia
        public String nombre;

        // Constructor
        public Persona(String nombre) {
                this.nombre = nombre;
        }

        // Funcionalidades
        public String decirNombre() {
                return this.nombre;
        }

        public int sumarEnteros (int n1, int n2) {
                return n1 + n2;
        }
}
```

y para este ejemplo, podremos utilizar el siguiente código para referenciarlo:

```
// Al hacer NEW se llama al constructor de la clase
Persona alberto = new Persona ("Alberto Díaz");

// Le pedimos que nos de su nombre
c.println("Yo me llamo " + alberto.decirNombre());

// Le pedimos que sume dos números
int suma = alberto.sumaEnteros(1, 5);          // suma es 6
```

Variables de Instancia: Son variables que representan las características que diferencian los distintos tipos de objetos de la clase.

Constructor: Son métodos especiales que permiten el "seteo" inicial del objeto. En particular la mayor parte de las veces se utilizan para setear variables de instancia o darle algunos valores iniciales al objeto.

Funcionalidades (Métodos): Son las acciones que el objeto puede realizar. Estas acciones pueden o no devolver datos. A veces se utilizan solo para darle valores a variables de instancia o simplemente para imprimir en pantalla (pasándole como parámetro la consola).

Es importante destacar que la instrucción **THIS** se utiliza dentro de la declaración de una clase como una referencia por defecto a ella misma, para llamar a las variables de instancia. También, y en forma práctica, la clase declarada se guarda en un archivo con el mismo nombre que el nombre de la clase, es decir para el ejemplo, **Persona.java** (ojo con la mayúscula del principio que es igualita a la del nombre de la clase).

Solución al Problema

Revisemos la solución del problema planteado a la motivación para mayor claridad a estos conceptos.

```
public class Complejo {
        // Variables de instancia
        public double parteReal;
        public double parteImag;

        // Constructor
        public Complejo (double real, double imaginario) {
                this.parteReal = real;
                this.parteImag = imaginario;
        }

        // Funcionalidades (algunas)
        public String formaBinomial () {
                return this.parteReal + "+" + this.parteImag + "i";
        }

        public String formaVectorial () {
                return "(" + this.parteReal + ", " +
                        this.parteImag + ")i";
        }

        public Complejo suma (Complejo sumando) {
                Complejo resultado = new Complejo
                        (this.parteReal + sumando.parteReal,
                         this.parteImag + sumando.parteImag);
                return resultado;
        }
}
```

Y para mayor claridad, como se utiliza en un programa principal sería algo como lo siguiente:

```
<...>
Complejo a = new Complejo (2, -9);
Complejo b = new Complejo (0, -1);

Complejo suma = a.suma (b);

c.println("La suma de " + a.formaBinomial() + " y " +
        b.formaBinomial() + " es: ");
c.println(c.formaVectorial());
<...>
```

Problemas

(a) Definir la clase **Triángulo** que guarde los valores de los catetos a y b (variables de instancia) y que permita realizar las siguientes operaciones sobre un triángulo rectángulo:

- Calcular el perímetro del triángulo
- Calcular el área del triángulo

```
public class Triangulo {
	// Variables de instancia que almacenan los catetos
	public double a;
	public double b;

	// Constructor que recibe los catetos de un triángulo
	public Triangulo(double a, double b) {
		this.a = a;
		this.b = b;
	}

	// Cálculo del perímetro
	public double perimetro() {
		// Calculamos primero la hipotenusa
		double c = Math.sqrt(Math.pow(this.a, 2) +
					Math.pow(this.b, 2));

		// Retornamos el perímetro
		return this.a + this.b + c;
	}

	// Cálculo del área
	public double area() {
		// Retornamos el área
		return (this.a * this.b) / 2;
	}
}
```

(b) Escribir el programa principal que use lo anterior y que simule el siguiente diálogo:

```
Ingrese un Triángulo:
a? _
b? _
El perímetro es = []
El área es = []
```

(me parece conocido de unas clases atrás)

```
<...>
// Pedimos los datos
Console c = new Console();
c.println("Ingrese un Triángulo:");
c.print("a?");
double a = c.readDouble();
c.print("b?");
double b = c.readDouble();

// Creamos el triángulo con estos catetos
Triangulo tr = new Triangulo(a, b);
```



```
// Retornamos los valores pedidos
c.println("El perímetro del triángulo es = " + tr.perimetro());
c.println("El área del triángulo es = " + tr.area());
<...>
```

(c) **(Propuesto)** Definir una clase llamada **Pantalla** que permita almacenar una **Console** (consola) y que esconda a la consola que hemos utilizado hasta ahora.

Para ello defina los métodos:

public Pantalla(Console c)
que es el constructor de la pantalla y que permite referenciar la consola externa.

public void escribir(String s)
que permite escribir en la consola (de instancia) el string **s** con un retorno de línea al final (println).

public String leeString()
que permite leer desde el teclado un String cualquiera.

(d) **(Propuesto)** Use la clase **Pantalla** para escribir el siguiente programa:

```
Hola. Yo soy el computador.
¿Cómo te llamas? ALBERTO
Hola ALBERTO mucho gusto en conocerte.
```

Herencia

Existe la clase **Dado** ya definida como:

int nCaras;	Variable de instancias que posee el número de caras del dado.
Dado(int n)	Constructor que permite crear un dado con **n** caras.
int tirarDado(int veces)	Simula que se tira **veces** veces el dado de **nCaras** caras y suma los resultados.
int tirarDado()	Llama a **tirarDado(1)**.

Se pide realizar una clase **Dado6** que simule un dado de 6 caras a partir de la clase **Dado**.

Conceptos

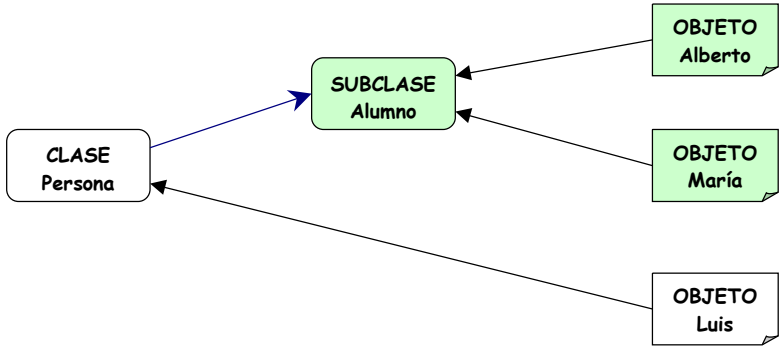
Herencia

*Una clase **Hereda** de otra cuando posee las mismas características y funcionalidades que su padre, y que se le pueden agregar otras características y funcionalidades particulares para ella.*

El concepto de herencia es otro paradigma de la programación orientada al objeto que la hacen una herramienta poderosa para el desarrollo.

La idea es no re-construir características ni funcionalidades comunes para muchas clases de objetos, por el contrario, es reutilizar lo que se va escribiendo para poder realizar mejores y más estructurados programas.

Por ejemplo, volviendo al ejemplo de **Persona**.



En el diagrama (un poco modificado del original visto 2 clases atrás) podemos ver que tenemos los mismos objetos: **María**, **Alberto** y **Luis**. Estos 3 objetos son **Personas** porque pertenecen a esa clase, pero hemos insertado un elemento adicional que es la clase **Alumno**:

¿Cuál es la idea de esta clase?

Alumno es una especialización de **Persona** porque "*todos los alumnos deben ser personas*". Esto quiere decir que un **Animal** no puede ser un alumno (se parece a la vida real),

Entonces, podemos decir que **Maria** y **Alberto** son **Personas** y **Alumnos** porque todas las funcionalidades y características de **Persona** son *heredadas* por la clase **Alumno**.

Veamos un ejemplo concreto.

Definamos la clase **Calculadora** que permite realizar las operaciones matemáticas básicas (suma, resta, multiplicación y división):

```
public class Calculadora {
	// Esta clase no tiene variables de instancia

	// Constructor vacío y no hace nada
	public Calculadora() {
	}

	// Métodos que permiten operar números reales
```



```
	public double sumar(double a, double b) {
		return (a + b);
	}

	public double restar(double a, double b) {
		return (a - b);
	}

	public double multiplicar(double a, double b) {
		return (a * b);
	}

	public double dividir(double a, double b) {
		if (b == 0) return 0;
		return (a / b);
	}
}
```

Este ejemplo (y a modo de repaso) se utilizaría de la siguiente forma:

```
//...
Console c = new Console();
Calculadora cal = new Calculadora();

// Pedimos datos
c.print("a?");
double a = c.readDouble();
c.print("+,-,*,/?");
String op = c.readLine();
c.print("b?");
double b = c.readDouble();

// Hagamos un IF múltiple de otra forma
double resultado = 0;
switch op {
	case "+":
		resultado = cal.sumar(a, b);
	case "-":
		resultado = cal.restar(a, b);
	case "*":
		resultado = cal.multiplicar(a, b);
	case "/":
		resultado = cal.dividir(a, b);
}

// Mostramos el resultado
c.println(a + " " + op + " " + b + " = " + resultado);
```

Y si mostramos la pantalla, saldría algo como:

```
a? 5
+, -, *, /? *
b? -3
5 * -3 = -15
```

Ahora bien. Necesitamos una clase **CalculadoraCientifica** que calcule además el SEN y COS. Para ello no tenemos que programar sumar. Restar, multiplicar y dividir de nuevo, solo debemos *heredarlos* de calculadora:

```
public class CalculadoraCientifica extends Calculadora {
	// Se heredan los métodos de Calculadora y las variables
	// de instancia.

	// Constructor de CalculadoraCientifica
	public CalculadoraCientifica() {
		// Invoca el constructor de Calculadora
		super();
	}

	// Solo los métodos que nos faltaban
	public double seno(double x) {
		return Math.sin(x);
	}

	public double coseno(double x) {
		return Math.cos(x);
	}
}
```

Y el mismo programa anterior podemos cambiar y agregar a **CalculadoraCientifica**:

```
//...
Console c = new Console();
CalculadoraCientifica cal = new CalculadoraCientifica();

// Pedimos datos
c.print("a?");
double a = c.readDouble();
c.print("+, -, *, /, seno, coseno?");
String op = c.readLine();
if (op != "seno" && op != "coseno") {
	c.print("b?");
	double b = c.readDouble();
}

// Todos los casos
double resultado = 0;
switch op {
	case "+":
		resultado = cal.sumar(a, b);
	case "-":
		resultado = cal.restar(a, b);
	case "*":
		resultado = cal.multiplicar(a, b);
	case "/":
		resultado = cal.dividir(a, b);
	case "seno":
		resultado = cal.seno(a);
	case "coseno":
		resultado = cal.coseno(a);
}

// Mostramos el resultado
if (op != "seno" && op != "coseno")
	c.println(a + " " + op + " " + b + " = " + resultado);
else
	c.println(op + "(" + a + ") = " + resultado);
```

¡Esto funciona!

Solución al Problema

Veremos que la solución es mucho más corta de lo que pensamos.

```
public class Dado6 extends Dado {
	// Hereda las variables de instancia de Dado

	// Constructor que no necesita parámetros
	public Dado6() {
		// Lo creamos con un dado de 6 caras
		super(6);
	}

	// Los métodos se mantienen por lo que no es necesario
	// declarar ninguno de los de la clase Dado
}
```

y si pensamos como funcionaría esto, quedaría un programa como este:

```
// ...
Console c = new Console();

// Creamos un dado de 6 caras
Dado6 d = new Dado6();

// Tiramos 3 veces el dado
c.println(d.tirar(3));

// Tiramos 1 vez el dado
c.println(d.tirar());

// Tiramos 10 veces el dado
c.println(d.tirar(10));
```

Es muy importante destacar que se están utilizando los métodos declarados en Dado y no cualquier otro método.

Problemas

(a) **Propuesto**. Escribir la clase **Dado** descrita en la motivación.

Existe una clase **Figura** definida de la siguiente forma:

Método	Descripción
Figura ()	Constructor vacío. Crea una figura sin puntos.
void agregaPunto (double x, double y)	Agrega un punto (x, y) a la figura.
void dibuja (Console c)	Dibuja la figura en la consola c.
Figura copia ()	Retorna una copia de la misma figura
boolean compara (Figura fig)	Retorna TRUE si la figura fig es igual a la figura (this) y FALSE si no.

(b) Escriba la clase **Cuadrado** que herede de **Figura** y que represente un cuadrado. Solo debe implementar el constructor **Cuadrado (double x, double y, double lado)** en donde (x, y) indica el centro del cuadrado y (lado) es el largo de cada lado.

(c) Haga lo mismo que en b pero para la clase **Círculo**, que recibe en su constructor el centro (x, y) y el radio del círculo. Nótese que debe dibujar punto a punto el círculo.

Enlace Dinámico

Recordando conceptos pasados, se desea organizar archivos a través de clases. Para esto se tiene la clase **Archivo** que es capaz de representar un archivo cualesquiera con las siguientes funcionalidades:

- **Archivo()**: Constructor de la clase que no hace nada.
- **void serNombre()**: Le asigna un nombre al archivo.
- **String getNombre()**: Obtiene el nombre del archivo.

Un ejemplo de utilización sería:

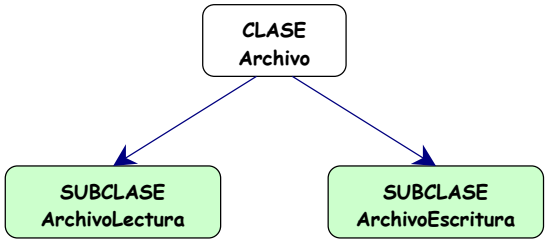
```
Archivo arch = new Archivo();
arch.setNombre("Curriculum.doc");
```

La clase **Archivo** está representando cualquier tipo de archivo y posee las operaciones genéricas de ellos, sin embargo no se sabe si es de *Lectura* o *Escritura*.

Para aliviar esto, podemos definir 2 subclases más llamadas **ArchivoLectura** y **ArchivoEscritura**, con los métodos respectivos:

	ArchivoLectura	**ArchivoEscritura**
Constructor	ArchivoLectura()	ArchivoEscritura()
Para Leer Datos	String leeLinea()	-
Para Escribir Datos	-	void escribeLinea(String s)
Identifica el Fin de Archivo	boolean esEOF()	-
Para Abrir el Archivo	void abrir()	void abrir()
Para Cerrar el Archivo	void cerrar()	void cerrar()

Como podemos ver, ambas clases se diferencian por lo métodos, pero ambas deben heredar las características y funcionalidades definidas para **Archivo**. Gráficamente esto se vería como:



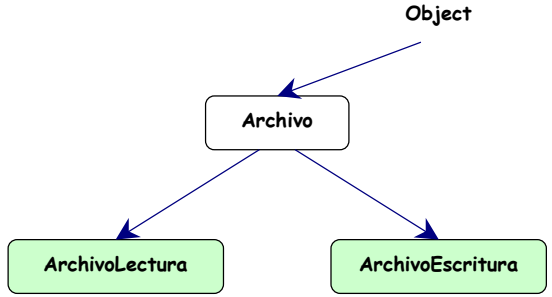
Y como sabemos, podemos crear objetos del tipo de la subclase y utilizar métodos de la subclase como de la superclase:

```
ArchivoLectura al = new ArchivoLectura();
al.setNombre("misdatos.txt");   // Método de la Superclase
al.abrir();                     // Método de la Subclase
```

Para que practiques, trata de implementar estas 3 clases.

Conceptos

Lo que hemos visto en la motivación ha sido lo que hasta ahora sabemos de las clases y de las subclases (aparte de su sintaxis). Pero el tema no llega hasta ahí, pues por ejemplo no se puede implementar una clase que no tenga superclase, porque TODAS las clases son subclases de una mayor: **Object**.



En Java, el modelo jerárquico de clase prediseñadas nace a partir de Object (del paquete **java.lang**) la cual posee algunas funcionalidades básicas y un constructor genérico. Es por eso que una clase que implementemos nosotros con nuestras propias manos no requiere un constructor por obligación, ya que por defecto llama al de Object.

Pero veamos los conceptos básicos para el modelo jerárquico de clases que posee Java:

Tipo Estático

> *El **Tipo Estático** de una variable es el tipo con el cual es declarada dicha variable, es decir, es la clase o tipo primitivo que es usado para indicar a la computadora cuánta memoria reservar.*

Esta forma compleja de ver la declaración de una variable se resume solo en la primera oración, es decir: ... **el tipo con el cual se declara** ... Por ejemplo:

```
String s;
Archivo arch;
ArchivoLectura fd;
```

En estos 3 casos es llamado *Tipo Estático* a la clase que hasta ahora hemos llamado **TIPO** de la variable. String, Archivo y ArchivoLectura sería la respuesta correcta.

Tipo Dinámico

> *El **Tipo Dinámico** de una variable es el tipo referenciado por ella, es decir, es el tipo con el cual se instancia la variable. Este tipo puede coincidir con el **Tipo Estático** como puede ser una subclase de él.*

Este concepto nos explica un poco más sobre la instanciación de una variable. Por ejemplo:

```
Archivo arch;
arch = new ArchivoLectura();
```

Mirando este tan básico ejemplo podemos ver que el tipo estático de **arch** es **Archivo**. Sin embargo, al momento de su instanciación (segunda línea) estamos llamando al constructor de la clase **ArchivoLectura**, es decir, estamos creando una instancia de esta clase. ¿Cómo es eso?.

El **Tipo Dinámico** nos permite realizar esta "barbaridad" ya que la subclase es una especialización de su padre (superclase) y es posible decir que: "Todo ArchivoLectura también es un Archivo".

Estos dos conceptos hacen el nacimiento del tercero y no menos importante término:

Enlace Dinámico

> *El **Enlace Dinámico** es una situación que ocurre cuando se invoca un método de un objeto referenciado por una variable, el método que efectivamente es ejecutado es el que corresponde al tipo dinámico de la variable.*

Este concepto nos abre muchas puertas, pero también nos pone algunas restricciones. Para entenderlo un poco mejor, veamos algunos ejemplos:

Primero, declararemos el tipo estático de **arch** como **Archivo** y el dinámico como **ArchivoLectura**.

```
Archivo arch;
arch = new ArchivoLectura();
```

Hasta ahora nada nuevo. Pero qué pasa si hacemos lo siguiente:

```
arch.setNombre("miarchivo");
```

Bueno, diremos que estamos bautizando nuestro archivo de lectura. Pero lo importante es el enlace dinámico que ocurre aquí, ya que aunque el método **setNombre** esté declarado en la

clase Archivo, Java ejecuta el método que es heredado (virtualmente escrito, no es el método original).

No obstante a que podamos hacer esto, también hay una limitante grande en el siguiente ejemplo:

```
Archivo arch = new ArchivoEscritura();
arch.setNombre("salida.txt");
arch.abrir();
```

Este trozo de código que al parecer no tiene nada de malo, no funciona.

¿Por qué?

Si analizamos un poco nos daremos cuenta que **arch** es de tipo estático **Archivo** y no **ArchivoEscritura** que es en donde el método **abrir** existe. En este caso prevalece la condición del tipo estático, por lo que el compilador no encontrará como válido pedirla a **Archivo** el método **abrir**. Java no hace un cast implícito, así que esto lo podríamos solucionarlo con un cast explícito a la subclase (que si está permitido).

```
Archivo arch = new ArchivoEscritura();
arch.setNombre("salida.txt");
( (ArchivoEscritura) arch).abrir();
```

Conclusión

El enlace dinámico nos da la facultad de escribir métodos genéricos para un grupo de clases "del mismo tipo", reciclando subprogramas.

¿Y para qué sirve hacerlo?

Es útil porque si no pudiéramos hacerlo, tendríamos que escribir el mismo código una y otra vez dependiendo de los distintos tipos de objetos que necesitáramos.

Sintaxis

Un elemento muy peculiar en Java es el llamado **Instanceof**.

Con esta instrucción uno puede comparar referencias a objetos y poder comparar el tipo dinámico sin conocerlo explícitamente, esto quiere decir, que podremos saber si una variable es instancia de una clase o de otra. Por ejemplo si tuviéramos un arreglo de archivos, que no conocemos si son de lectura o escritura:

```
Archivo archs[];
// ... en alguna parte los crean ...
for (int i=0; i<archs.length; i++) {
      c.print (archs[i].getNombre() + " es de ");
      if (archs[i] instanceof ArchivoLectura)
            c.println("Lectura");
```

```
        else if (archs[i] instanceof ArchivoEscritura)
                c.println("Lectura");
        else // caso que es instancia de Archivo
                c.println("Desconocido");
}
```

Interface

Clase especial que permite definir las firmas de los métodos que DEBE poseer toda clase heredera de ella misma, es decir, permite definir un patrón para la creación de las clases hija de una interface.

Este concepto bastante extraño es muy útil, ya que permite dar pautas para escribir clases de cierto tipo. Por ejemplo, podemos declarar una **interface** llamada **Nodo** que permite almacenar un elemento cualquiera con características no definidas. Entonces la interface Nodo quedaría así:

```
public interface Nodo {
        public Object sacarValor();
        public void ponerValor(Object o);
}
```

Como podemos ver no hemos construido ni declarado nada dentro de **Nodo**, solo hemos puesto las firmas de los métodos. Fíjate que todos son públicos.

Todos los métodos que se definen en una interface DEBEN ser públicos, porque están definiendo la forma de comunicación que tendrán los programas con las clases que IMPLEMENTEN esta interface.

Por ejemplo, escribamos un Nodo que nos permita almacenar una clase **Archivo**:

```
public class NodoArchivo implements Nodo {
        private Archivo arch;

        public Object sacarValor() {
                return arch;
        }

        public void ponerValor(Object o) {
                // Debemos suponer que o es de tipo dinámico Archivo
                arch = (Archivo) o;
        }
}
```

Aquí estamos implementando un elemento llamado **NodoArchivo** que almacena un archivo e implementa los métodos de la interface **Nodo**. Esto no limita a que podamos escribir más métodos, por el contrario. Siempre nacerán funcionalidades específicas que solo NodoArchivo tendrá, y que no son comunes a Nodo.

Una Interface NO PUEDE tener objetos, es decir:

```
Nodo n = new Nodo();            // Esto está errado
```



```
Nodo n = new NodoArchivo();      // Esto está correcto
```

Un ejemplo práctico es:

```
public interface Arreglo {
        // Dimensionar un arreglo
        public void dimensionar (int n);

        // Largo de un arreglo
        public int largo();
}
```

Esta **Interface** define la estructura (interface) general que tienen todos los arreglos: ver y dimensionar el tamaño del arreglo (cantidad de elementos). Implementemos ahora entonces un arreglo de Strings a partir de esta interface:

```
public class ArregloStrings implements Arreglo {
        private String els[];

        public ArregloString() {
        }

        // Se implementan los métodos de la interface
        public void dimensionar(int n) {
                this.els = new String[n];
        }

        public int largo() {
                return this.els.length;
        }

        // Los siguientes métodos no corresponden a la definición
        // de la interface
        public String sacar(int i) {
                return this.els[i];
        }

        public void poner(int i, String x) {
                this.els[i] = x;
        }
}
```

Como podemos observar, tenemos la clase que implementa la interface. Sin embargo la clase **ArregloString** posee métodos que no son comunes con otros tipos de arreglos (**ArregloEntero**, **ArregloArchivo**, etc), ya que su definición depende del tipo de la variable de instancia de la clase propia (**els**).

Clase Abstracta

Una Clase Abstracta es un híbrido entre una Interface y una Clase normal que permite definir métodos y variables de instancia (como en las clases normales) pero también dejar por definir en los hijos de la clase algunos métodos (como en las interfaces).

Las clases abstractas en general son utilizadas como interfaces más especializadas, pero con líneas de código útil y que hacen cosas.

La sintaxis de una clase abstracta es muy similar a la de las clases normales, sin embargo posee algunos elementos adicionales:

```
abstract class Nodo {
	// Puede tener variables de instancia
	Object elto;

	// Para que se definan en los hijos, igual que INTERFACE
	public abstract Object sacarValor();

	// y para definir como en clases normales
	public void ponerValor(Object o) {
		elto = o;
	}
}
```

Ahora bien, las clases que hereden de **Nodo** en este caso NUNCA sabrán que su padre es una clase abstracta. Su única regla es que DEBEN implementar aquellos métodos definidos como abstractos en la superclase. Es decir:

```
public class NodoArchivo extends Nodo {
	// Ya no se necesitan variables de instancia

	// Se DEBE implementar
	public Object sacarValor() {
		return arch;
	}

	// El método ponerValor() no es necesario implementarlo,
	// pues lo está en la superclase Nodo.
}
```

Al igual que las interfaces, NO SE PUEDE tener un objeto del tipo de la clase abstracta.

Veamos el caso de los arreglos pero con clase abstracta (más fácil):

```
public abstract class Arreglo {
	// Se presentan los métodos que serán implementados
	public abstract void dimensionar(int n);

	public abstract int largo();

	// En este caso no tiene métodos propios
}
```

Vemos claramente que cambia un poco el punto de vista que se puede utilizar para este tipo de casos. Sin embargo, queda claro como la clase se implementa luego:

```
public class ArregloStrings extends Arreglo {
	private String els[];

	public ArregloString() {
```



```
	}

	// Se implementan los métodos de la clase abstracta
	public void dimensionar(int n) {
		this.els = new String[n];
	}

	public int largo() {
		return this.els.length;
	}

	// Los siguientes métodos no corresponden a la definición
	// de la clase abstracta
	public String sacar(int i) {
		return this.els[i];
	}

	public void poner(int i, String x) {
		this.els[i] = x;
	}
}
```

¿Alguna diferencia?

Solución al Problema

Es muy interesante ver el problema de los Archivos de Lectura y Escritura con clases abstractas e interfaces. Veamos primero la implementación con **Interface**:

```
public interface Archivo {
	public void darNombre(String nombre);
	public void abrir();
	public void cerrar();
}
```

Solo eso, ya que los archivos de lectura/escritura se diferencian en lo que hacen entre el abrir y cerrar.

```
public class ArchivoLectura implements Archivo {
	private BufferedReader bf;
	private String nombre;

	// Constructor vacío
	public ArchivoLectura() {
	}

	// Los métodos de la interface
	public void darNombre (String nombre) {
		this.nombre = nombre;
	}

	public void abrir () {
		this.bf = new BufferedReader (
			new FileReader( this.nombre ) );
	}

	public void cerrar () {
		this.bf.close();
	}
```

```
	// Métodos propios
	public String leeLinea() {
		return this.bf.readLine();
	}
}

public class ArchivoEscritura implements Archivo {
	private PrintWriter pw;
	private String nombre;

	// Constructor vacío
	public ArchivoEscritura() {
	}

	// Los métodos de la interface
	public void darNombre (String nombre) {
		this.nombre = nombre;
	}

	public void abrir () {
		this.pw = new PrintWriter (
			new FileWriter( this.nombre ) );
	}

	public void cerrar () {
		this.pw.close();
	}

	// Métodos propios
	public void escribeLinea(String linea) {
		this.bf.printLine(linea);
	}
}
```

Mucho más bonito que lo que alguna vez vimos ¿no?. Bueno la utilización de estas clases queda a discreción de ustedes, ya que si repasamos más ese tema quedaremos más complicados... ♍

Ahora veamos la implementación de Archivos con **Clase Abstracta**.

```
public abstract class Archivo {
	// Declaremos el común de ambos tipos
	// No es privado porque quiero que lo lean los hijos
	protected String nombre

	// Los por implementar son abrir y cerrar
	public abstract void abrir ();
	public abstract void cerrar ();

	// El genérico
	public void darNombre (String nombre) {
		this.nombre = nombre;
	}

	public Archivo (String nombre) {
		this.darNombre(nombre);
		this.abrir();
	}
}
```


Hey, pudimos implementar el **darMombre** sin tener el problema del tipo de archivo. Veamos como quedan los tipos:

```
public class ArchivoLectura extends Archivo {
	private BufferedReader bf;

	// Constructor
	public ArchivoLectura (String nombre) {
		super(nombre);
	}

	// Los métodos de la clase abstracta
	public void abrir () {
		this.bf = new BufferedReader (
			new FileReader( this.nombre ) );
	}

	public void cerrar () {
		this.bf.close();
	}

	// Métodos propios
	public String leeLinea() {
		return this.bf.readLine();
	}
}

public class ArchivoEscritura extends Archivo {
	private PrintWriter pw;

	// Constructor vacío
	public ArchivoEscritura() {
	}

	// Los métodos de la clase abstracta
	public void abrir () {
		this.pw = new PrintWriter (
			new FileWriter( this.nombre ) );
	}

	public void cerrar () {
		this.pw.close();
	}

	// Métodos propios
	public void escribeLinea(String linea) {
		this.bf.printLine(linea);
	}
}
```

Se nos simplifica más la solución.

Pero es cosa de gustos y elegir cuál es mejor no es el problema.

Problemas

Se tiene la clase **Carta** que permite modelar una carta de un mazo de cartas inglesas (de póker). Su definición es la siguiente:

```
public class Carta {
	private String pinta;
	private String numero;

	public Carta (int p, String n) {
		this.pinta = (String) p;
		this.numero = n;
	}

	public String verCarta () {
		return this.numero + "-" + this.pinta
	}

	static public int compararCarta (Carta c1, Carta c2) {
		...
		// 1 si c1 > c2
		// 0 si c1 = c2
		// -1 si c1 < c2
		...
	}
}
```

(a) Escriba la clase **Mazo** que permite modelar el mazo de cartas. Para ello utilice la siguiente definición de la clase:

`Mazo ()`	(Constructor) Crea un nuevo mazo de cartas. Llena el mazo.
`void poner(Carta c)`	Pone la carta c en el mazo (en la primera posición)
`Carta sacar()`	Saca la primera carta del mazo. Esto elimina la carta del mazo.

Nota: En este caso considere las pintas como un número, es decir:

1. Diamantes
2. Corazones
3. Tréboles
4. Picks

Solución

```
public class Mazo {
	private Cartas m[];
	private int indice;

	public Mazo () {
		this.m = new Cartas[52];
		this.indice =  0;
		for (int pinta=1; pinta<5; pinta++) {
			for (int n=1; n<14; n++) {
				String numero = (String) n;
				if (n > 10)
					numero = "J";
				if (n > 11)
					numero = "Q";
				if (n > 12)
					numero = "K";
				this.m[this.indice] =
```



```
					new Carta(pinta, numero);
				this.indice++;
			}
		}
	}

	public void poner (Carta c) {
		this.m[this.indice] = c;
		this.indice++;
	}

	public Carta sacar () {
		this.indice--;
		return this.m[this.indice];
	}
}
```

(b) Escriba el método privado **void mezclar()** que permite aleatoriamente mezclar el mazo.

Solución

Tiene muchas soluciones, pero veremos solo una:

```
private void mezclar() {
	// Cortamos el mazo actual
	int corte = Math.trunc(Math.random()*52);

	// Creamos los 2 mazoz
	Cartas c1[] = new Cartas[corte];
	Cartas c2[] = new Cartas[52 - corte];

	// Luego repartimos los submazos
	for (int i=0; i<52; i++) {
		if (i < corte)
			c1[i] = this.sacar();
		else
			c2[i - corte] = this.sacar();
	}

	// Y luego volvemos una a una las cartas del mazo actual
	int j1 = 0, j2 = 0;
	while (j1 > c1.length && j2 > c2.length) {
		if (i % 2 == 0) {
			this.poner(c1[j1]);
			j1++;
		}
		else {
			this.poner(c2[j2]);
			j2++;
		}
	}

	while (j1 > c1.length) {
		this.poner(c1[j1]);
		j1++;
	}

	while (j2 > c2.length) {
		this.poner(c2[j2]);
		j2++;
	}
```

```
}
```

(c) Utilizando (b), escriba el método **revolver** para los siguientes casos (firmas):

- **void revolver()**: Revuelve una vez el mazo una vez.
- **void revolver(int n)**: Revuelve n veces el mazo.

Solución

```
public void revolver() {
       mezclar();
}

public void revolver(int n) {
       for (int i=0; i<n; i++)
              mezclar();
}
```

(d) Se desea implementar la clase abstracta **Figura** que permita modelar una figura genérica a partir de segmentos de rectas unidas por vértices.

La definición de la clase es la siguiente:

`Variables de Instancia`	Representación con 2 arreglos de **double** para almacenar las componentes **x** e **y**.
`public double Perimetro()`	Calcula el perímetro del polígono.
`public abstract double Area()`	Firma del método que definirá cada uno de las áreas de figuras que hereden de esta clase.

Solución

```
public abstract class Figura {
       protected double x[];
       protected double y[];

       private double distancia (double x1, double y1,
                                 double x2, double y2) {
              return Math.sqrt(Math.pow(x2 - x1) +
                               Math.pow(y2 - y1));
       }

       public double perimetro () {
              double p = 0;
              int n = this.x.length;
              for (int = 0; i < n - 1; i++) {
                     p += this.distancia(this.x[i], this.y[i],
                                         this.x[i+1], this.y[i+1]);
              }
              p += this.distancia(this.x[n - 1], this.y[n - 1],
                                  this.x[0], this.y[0]);
              return p;
       }

       public abstract double area();
}
```


(e) Construya la clase **Triangulo** a partir de la clase **Figura** para que permita modelar un triángulo. Para ello considere que esta clase posee el siguiente constructor:

public Triangulo (double x[], double y[])

en donde **x** e **y** son arreglos que traen 3 valores para cada una de las coordenadas de los puntos del triángulo, es decir, **(x[i], y[i])** es un punto.

Nota: No olvide implementar lo que sea necesario.

Solución

```
public class Triangulo extends Figura {
       public Triangulo (double x[], double y[]) {
              super.x = new double[x.length];
              super.x = x;

              super.y = new double[y.length];
              super.y = y;
       }

       // Se declara el método abstracto
       public double area() {
              // PROPUESTO: Calculo a partir del perimetro
       }
}
```

(f) Rehaga el problema pero ahora utilizando una Interface en vez de una Clase Abstracta.

Solución

```
public interface Figura {
       public double perimetro();
       public abstract double area();
}

public class Triangulo implements Figura {
       protected double x[];
       protected double y[];

       public Triangulo (double x[], double y[]) {
              super.x = new double[x.length];
              super.x = x;

              super.y = new double[y.length];
              super.y = y;
       }

       private double distancia (double x1, double y1,
                                 double x2, double y2) {
              return Math.sqrt(Math.pow(x2 - x1) +
                               Math.pow(y2 - y1));
       }

       public double perimetro () {
              double p = 0;
              int n = this.x.length;
              for (int = 0; i < n - 1; i++) {
                     p += this.distancia(this.x[i], this.y[i],
```

```
                                        this.x[i+1], this.y[i+1]);
                }
                p += this.distancia(this.x[n - 1], this.y[n - 1],
                                    this.x[0], this.y[0]);
                return p;
        }

        public double area () {
                // PROPUESTO: Calculo a partir del perimetro
        }
}
```

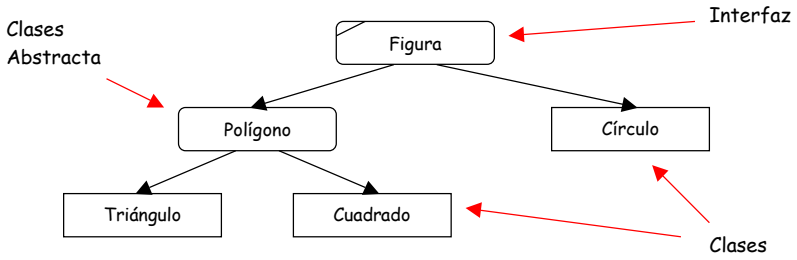
(d) ¿Qué ventajas tiene utilizar una con respecto a la otra solución?

Solución

Al utilizar **Interface** tenemos la capacidad de, en este caso, incluir dentro de la definición a **Circulo** como una clase que implementa **Figura**. Mientras que en el caso de la utilización de **Clase Abstracta** el círculo lo DEBEMOS representar como un polígono.

Sin embargo lo anterior, usar **Clase Abstracta** queda más apropiado según las especificaciones ya que la repetición de algunos métodos no pasa (por ejemplo el del perímetro para figuras como Cuadrados, Triángulos y demases.

El modelo general (y más adecuado) sería:



Otro Problema

(Sacado de un control de hace un par de años atrás) Un problema bastante real sería suponer que existe una clase **Figura** con los siguientes métodos:

- **Figura()**: Constructor vacío que no hace nada
- **void agregarPunto(double x, double y)**: Agrega un punto a la figura
- **void sacarPunto()**: Saca el último punto agregado de la figura
- **double perímetro()**: Obtiene el perímetro de la figura (suma de sus lados)

Con esta definición, la clase puede representar cualquier figura en un plano, a partir de trozos de rectas que definen su contorno (hasta un círculo si lo trozos son pequeñitos).

(a) Se pide implementar una subclase **Rectangulo** que tenga los siguiente métodos:

- **Rectangulo()**: Constructor que tampoco hace nada
- **void definirRectangulo(double x1, double y1, double x2, double y2)**: Define un rectángulo a partir de dos vértices diametralmente opuestos.

Solución

```
public class Rectangulo extends Figura {
        public Rectangulo() {
                super();
        }
        public void definirRectangulo(double x1, double y1,
                                      double x2, double y2) {
                super();
                super.agregarPunto(x1, y1);
                super.agregarPunto(x1, y2);
                super.agregarPunto(x2, y2);
                super.agregarPunto(x2, y1);
        }
}
```

(b) Implementar la subclase **Cuadrado** a partir de **Rectangulo** solo con el constructor:

- **Cuadrado(double x, double y, double d)**: Define un cuadrado desde el punto **(x, y)** y de lado **d**.
- **Cuadrado()**: Constructor que no hace nada.

Nota: La superclase de Cuadrado es Rectángulo.

Solución

```
public class Cuadrado extends Rectangulo {
        public Cuadrado() {
                super();
        }
        public Cuadrado(double x1, double y1, double d) {
                super.definirRectangulo(x1, y1, x1+d, y1+d);
        }
}
```

(c) Escriba un método que permita leer desde un archivo de puntos formateado como:

- **Tipo** (1 carácter): (C)uadrado, (R)ectángulo o (F)igura
- **Puntos**: Par de coordenadas del tipo (X,Y) y separadas hasta el fin de línea.

Y que las almacene en un arreglo de **Figura**.

Por ejemplo:

```
C 1,1 1,2 2,2 2,1
R 3,4 5,4 5,-2 3, -2
F 1,5 -4,7 -1,-1 0,0 9,10 -1,-2 10,10
...
```

(Estas son 3 figuras, pero una es un cuadrado, otra un rectángulo y otra una figura cualquiera. Es muy importante fijarse en los espacios que vienen entre cada par de datos)

Nota: El nombre de archivo viene por parámetro y vienen 100 figuras en el archivo.

Solución

```
public Figura[] leerArchivo(String nombre) {
	// Declaramos el arreglo de resultados
	Figura figs[];

	// Inicializamos el arreglo
	// Ojo que cada casilla requiere un posterior NEW
	figs = new Figura[100];

	// Leemos el archivo
	BufferedReader bf = new BufferedReader(
				new FileReader(nombre));
	String linea;
	int i=0;
	for (int i=0; i<100; i++) {
		linea = bf.readLine();

		// Discriminamos el tipo de figura
		String tipo = linea.charAt(0);
		switch tipo {
			case "C":
				figs[i] = new Cuadrado();
			case "R":
				figs[i] = new Rectangulo();
			case "F":
				figs[i] = new Figura();
		}

		// Y ahora ponemos los puntos de la figura
		linea = linea.substring(2);
		int espacio, coma;
		double x, y;
		while ((espacio = linea.indexOf(" ")) > 0) {
			coma = linea.indexOf(",");
			x = new Double(
				linea.substring(0, coma)
				).doubleValue();
			y = new Double(
				linea.substring(coma + 1, espacio - coma)
				).doubleValue();
			figs[i].agregarPunto(x, y);
			linea = linea.substring(espacio + 1);
		}
		coma = linea.indexOf(",");
		x = new Double(
			linea.substring(0, coma)
			).doubleValue();
		y = new Double(
			linea.substring(coma + 1, length(linea) - coma)
			).doubleValue();
		figs[i].agregarPunto(x, y);
	}
	bf.close();

	// Retornamos el arreglo de figuras
	return figs;
```



```
}
```

(d) Escribir un programa que utilice (c) para simular el siguiente diálogo:

```
Nombre de archivo de entrada? puntos.txt
Leyendo puntos... Ok!

Contando cuántas figuras hay:
Cuadrados: 10
Rectángulos: 2
Otras figuras: 27
```

Solución

```
Console c = new Console();
c.print("Nombre de archivo de entrada?");
String nombre = c.readLine();

// Se leen los puntos
c.print("Leyendo puntos... ");
Figura figs[] = leerArchivo(nombre);
c.print("Ok!");

// Ahora se cuentan las figuras
c.println("Contando cuántas figuras hay:");
int nC = 0, nR = 0, nF = 0;
for (int i=0; i<figs.length; i++) {
	if (figs[i] instanceOf Cuadrado)
		nC++;
	else if (figs[i] instanceOf Rectangulo)
		nR++;
	else		// figs[i] instanceOf Figura
		nF++;
}
c.println("Cuarados: " + nC);
c.println("Rectángulos: " + nR);
c.println("Otras Figuras: " + nF);
```

Problemas Propuestos

(a) Volviendo al tema de la motivación, crear a partir de un **Mapa** (superclase), la clase **MapaEnteros** (subclase) que permita almacenar solo números enteros en el mapa. Para ello son necesarios los siguientes métodos:

`MapaEnteros (int n)`	(Constructor) Crea un nuevo mapa de enteros vacío de n elementos.
`void agregar (int i, int x)`	Agrega el entero x a la posición i del mapa.
`int sacar (int i)`	Saca el elementos de la posición i del mapa.

Note claramente que el **agregar** no ha cambiado. ¿Deberá implementarlo entonces?

Capítulo XIII: Ordenamiento y Búsqueda

Motivación

El MTT ha decidido almacenar las mediciones que realiza a los vehículos en un lugar de Santiago. Para esto requiere que se le realice un programa que haga el siguiente diálogo:

```
Bienvenido al Sistema de Control de Contaminación
Ingrese el código del Inspector? 538

Bienvenido señor.

Ingrese patente del vehículo? HG 5857
Índice registrado? 3.8
Clave Verde

Ingrese patente del vehículo? ZZ 3481
Índice registrado? 2.6
Clave Verde

Ingrese patente del vehículo? AS 2216
Índice registrado? 5.1
Clave Roja

...

Ingrese patente del vehículo? 0

Se han registrado 53 mediciones:
 (1) índice de 7.8
 (2) índice de 7.8
 (3) índice de 7.7
 (4) índice de 7.2
...
(53) índice de 2.6
```

Como se ve en el diálogo, el programa consta de 2 partes:

1. Recolectar los datos de cada medición durante el día. Estas mediciones son al azar, es decir, no se sabe cuál es el índice que se está ingresando (mayor que 5.0 es clave roja). La lectura de datos termina con una patente 0 y no e sabe el número de mediciones, sin embargo, puede ser un máximo de 100 (para que use un arreglo).
2. Desplegar en pantalla las mediciones realizadas, pero esta vez ordenadas de mayor a menor, para tener una estadística más clara.

La solución a este problema es bastante sencilla:

```
Console c = new Console();
c.println("Bienvenido al Sistema de Control de Contaminación");
c.print("Ingrese el código del Inspector?");
int code = c.readInt();

c.println("Bienvenido señor.");
```



```
double mediciones[] = new double[100];
int n = 0;
while(true);
c.print("Ingrese patente del vehículo?");
String pat = c.readLine();

if (pat.equals("0") || n == 100)
break;

c.print("Índice registrado?");
mediciones[n] = c.readDouble();

if (mediciones[n] > 5.0)
      c.println("Clave Roja");
else
      c.println("Clave Verde");

n++;
}

// n sale con el número del siguiente elemento o 100
// que es lo mismo que el NÚMERO de elementos
// por lo tanto enviaremos a un método el arreglo
// entre 0 y el máximo + 1 para que quede completa-
// mente ordenado.
ordenarArreglo(mediciones, 0, n);

// Ahora que está ordenado, desplegamos
c.println("Se han registrado " + n + " mediciones:");

for (int i=0; i<n; i++) {
      c.println("(" + (i+1) + ") índice de " + mediciones[i]);
}
```

Solo nos queda pendiente como ordenar el arreglo... ♍

Conceptos

Ordenamiento

> *Es un algoritmo que nos permite cambiar el orden (de posición) los elementos para dejarlos ordenados según un criterio fijo (numéricamente, lexicográficamente, de menor a mayor, de mayor a menor, etc).*

Esta definición bastante básica es la que nos permite entender que es un algoritmo de ordenamiento, pero no tiene mucho sentido si no hay una aplicación tangible.

El ordenamiento de arreglos es muy útil para realizar búsquedas de elementos, puesto que cuando buscamos dentro de un arreglo un elemento específico (por ejemplo el nombre de una persona) deberíamos usar la "fuerza bruta" para encontrarlo. Por ejemplo:

```
String nombres[] = new String[100];
// en alguna parte de se ingresan los nombres
// desordenadamente
String nombreBuscado = "Morales Gutierrez Carlos Alberto";
```

```
int i = 0;
while ( ! nombres[i].equals(nombreBuscado) ) {
       i++;
}
```

Si nos ponemos a analizar este programita para buscar un nombre, tenemos el problema que si la persona hubiese sido la última en el arreglo, tuvimos que recorrerlo completo para encontrarlo. ¿Se imaginan que pasaría si la guía de teléfono estuviera desordenada y ustedes quisieran encontrar el teléfono de un tío? ¡¡Qué locura!!.

Si la lista de nombres está ordenada, a "Morales" no lo buscaremos al principio, porque lo más probable que allí estén "Astorga", "Araya" y todos los primeros apellidos, por lo tanto empezamos de un punto más en la mitad, en donde están los "Lorca", "Martinez" y "Ortega".

Bueno, pero para eso, el arreglo **hay** que ordenarlo. ¿Cómo se hace?.

Existen algoritmos de ordenamiento, partiendo del base que es una forma **exhaustiva**, es decir, de la siguiente forma:

1. Buscar el mayor de todos
2. Ponerlo en la primera (o última) posición de un arreglo auxiliar
3. "Anular" el mayor
4. Volver a hacer desde 1 hasta que no quede ninguno

Esta es la forma básica. Pero ¿qué diferencia hay con los algoritmos de ordenamiento?, en el fondo nada, pues al final siempre ordenan un arreglo (de lo que sea). Sin embargo definiremos lo que es la **eficiencia** de los algoritmos, algo que puede ser un punto crítico al momento de elegir una forma de ordenar una serie de datos.

Eficiencia

> *En los algoritmos, la eficiencia se mide como el **tiempo** que toma un algoritmo en realizar su finalidad.*

Pero no es tan sencillo medir ese tiempo como tomar un cronómetro y ver cuánto se demora en hacer un algoritmo su trabajo con 10, 100 o 1.000 datos, pues los tiempos que puede tomar son milisegundos.

Para medir la eficiencia existe un examen llamado **análisis de tiempo** que nos permite obtener el valor algebraico del tiempo que nos da la idea de la magnitud del tiempo que puede tomar el algoritmo es su ejecución expresado en función de la cantidad de elementos que posee (cantidad de elementos del arreglo en el caso de ordenamiento).

Veamos cómo analizar el caso de ordenamiento exhaustivo:

```
public void ordenar (int[] arreglo, int nEls) {
       int auxiliar[] = new int[nEls];

       // Ciclo 1
       for (int i=nEls-1; i>0; i--) {
              int maximo = 0

              Ciclo 2
              for (j=0; j<nEls; j++) {
                     // Ponemos el máximo en la última posición
                     if (arreglo[maximo] < arreglo[j])
                            máximo = j;
              }

              auxiliar[i] = arreglo[máximo];
              arreglo[máximo] = -1; // Se elimina como posible sgte
       }

       arreglo = auxiliar;
}
```

Definamos como T_0 el tiempo de ejecución de toda línea de código fuera de los ciclos, T_1 el tiempo que toma las líneas del ciclo 1 y T_2 las del ciclo 2. Todas se miden en forma independiente de las otras, es decir no considera los valores dentro de su ejecución. Entonces, ordenar un elemento tomará el siguiente tiempo:

$$T(1) = T_0 + 1 * T_1 + 1 * T_2$$

T0 es siempre constante, ya que siempre se ejecutará 1 vez independiente de la cantidad de veces que ejecutemos el método, sin embargo T1 y T2 están dependiendo de la cantidad de elementos, ya que el for repite esas instrucciones como número de elementos tengamos, es decir, para n elementos nos quedaría:

$$T(n) = T_0 + n * T_1 + n * (n * T_2)$$

Desarrollando un poquito:

$$T(n) = T_2 * n^2 + T_1 * n + T_0$$

Diremos entonces que el **orden del algoritmo** es el término variable de mayor exponente dentro de la cuadrática, esto quiere decir (y lo anotaremos así) que:

$$T(n) = O(n^2)$$

Se leerá como: "*El tiempo que toma el algoritmo es de orden n^2*". Esto significa que si tomamos un arreglo de 100 elementos, el orden de magnitud del algoritmo será de 10.000 (5 ceros algo alto cuando hablamos de 1.000.000 de elementos). Pero no todo es tan malo, pues esto es en teoría. En la práctica (aunque parezca un chiste) podemos tomar 3 casos:

- **Mejor Caso** (arreglo ordenado): $O(n^2)$
- **Peor Caso** (arreglo muuuuuy desordenado): $O(n^2)$
- **Caso Promedio** (otro caso normal): $O(n^2)$

Existen muchos algoritmos de ordenamiento definidos que nos permiten eficiencia en el ordenamiento. Se han definido como base los siguientes:

Algoritmo de Selección y Reemplazo
Algoritmo de la Burbuja (BubbleSort)
QuickSort
MergeSort

Advertencia: Lo algoritmos de ordenamiento en general se pueden implementar en forma iterativa (for, while) pero son mucho más eficiente cuando se utiliza la recursividad. Es recomendable que repacen el capítulo de **Recursividad** para entender mejor este.

Algoritmo de Selección y Reemplazo

Probablemente este sea el algoritmo más intuitivo que existe, pues utiliza una metodología bastante básica que se puede analizar. Es muy eficiente y fácil de implementar, pero solo cuando los arreglos son pequeños.

Veamos como lo haría una persona inexperta y sin conocimientos para ordenar un arreglo (solo valores positivos por simplicidad):

```
void ordenarArreglo (int arreglo[], int nEls) {
      // Crearemos un arreglo auxiliar del mismo tamaño
      int auxiliar[] = new int[nEls];

      // Vamos sacando cada uno de los elementos y
      // los vamos almacenando el el auxiliar ordenadamente
      // (Supongamos valores positivos)
      for (int i=nEls-1; i>0; i--) {
            int maximo = 0
            for (j=0; j<nEls; j++) {
                  // Ponemos el máximo en la última posición
                  if (arreglo[maximo] < arreglo[j])
                        máximo = j;
            }

            auxiliar[i] = arreglo[máximo];
            arreglo[máximo] = -1; // Se elimina como posible sgte
      }

      arreglo = auxiliar;
}
```

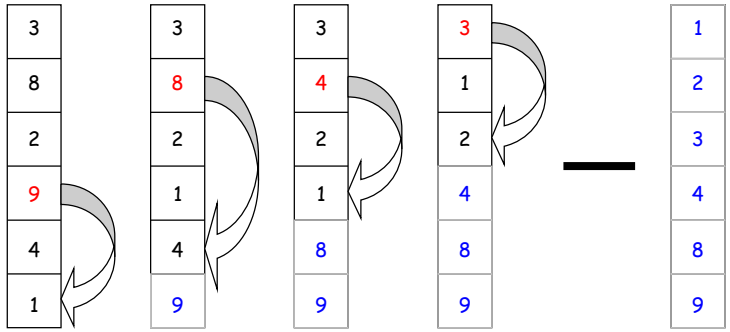
Este método lo que hace es ordenar en FORMA BRUTA un arreglo no es muy eficiente, ya que no lleva un patrón fijo de búsqueda del máximo valor (lo que pasa cuando buscamos en un arreglo desordenado, como ya mencionamos). Veamos como funciona el algoritmo se selección que no utiliza un arreglo auxiliar.

1. El arreglo se ordena entre 0 y nEls
   a. Se busca donde está el máximo
   b. Se intercambia el último elemento por donde está el máximo
   c. Olvidarse del máximo ahora y seguir.

2. El arreglo se ordena entre 0 y nEls-1
   a. Se busca donde está el siguiente máximo
   b. Se intercambia el penúltimo elemento por donde está el siguiente máximo
   c. Olvidarse del siguiente máximo ahora y seguir.
3. ...

Ultima. El arreglo se ordena entre 0 y 0
   a. Terminar, porque un arreglo de un elemento está ordenado.

Gráficamente podemos ver esto como:



Así, no repasamos todo el arreglo en cada iteración... ☺

Veamos ahora cómo escribiríamos eso en forma iterativa:

```
void ordenarArreglo (int arreglo[], int nEls) {
      for (int i=nEls-1; i>0; i--) {
            int maximo = 0
            for (j=0; j<i; j++) {
                  // Buscamos el máximo
                  if (arreglo[maximo] < arreglo[j])
                        máximo = j;
            }

            int auxiliar = arreglo[máximo];
            arreglo[máximo] = arreglo[i];
            arreglo[i] = auxiliar;
      }
}
```

¿No hay demasiada diferencia con la "fuerza bruta" como le llamamos?... Pues sí... porque si observamos con algo de detención nos daremos cuenta que el segundo for (j) solo recorre hasta el elemento en donde pondremos el máximo –1, es decir, hasta antes de donde debe estar.

Pero de todas formas hicimos 2 ciclos. Veamos como se vería recursivamente:

```
void ordenarArreglo (int arreglo[], int nEls) {
      if (nEls == 1)
            return;

      int maximo = 0
      for (j=0; j<nEls-2; j++) {
            // Buscamos el máximo
            if (arreglo[maximo] < arreglo[j])
                  máximo = j;
      }

      int auxiliar = arreglo[máximo];
      arreglo[máximo] = arreglo[nEls-1];
      arreglo[nEls-1] = auxiliar;

      ordenarArreglo(arreglo, nEls-1);
}
```

Acá podemos verlo que hace exactamente lo que explicamos arriba como algoritmo, pues utiliza el caso general (ordenar entre 0 y nEls) y luego va y vuelva a llamar al caso general con un elemento menos (ordenar entre 0 y nEls-1). Así sucesivamente quedaría al final ordenando el arreglo HASTA que quede un elemento.

Veamos como se comporta la eficiencia de este algoritmo:

```
void ordenarArreglo (int arreglo[], int nEls) {
      // Ciclo 1
      for (int i=nEls-1; i>0; i--) {
            int maximo = 0

            // Ciclo 2
            for (j=0; j<i; j++) {
                  if (arreglo[maximo] < arreglo[j])
                        máximo = j;
            }

            int auxiliar = arreglo[máximo];
            arreglo[máximo] = arreglo[i];
            arreglo[i] = auxiliar;
      }
}
```

- T_0 : En este caso es 0
- T_1 : Se ejecuta n veces
- T_2 : Se ejecuta una cantidad de veces variable dependiendo del valor de i

Hagamos la ecuación general de inmediato:

$$T(n) = T_0 + n * T_1 + A$$

donde A es la cantidad de veces que se ejecutó el ciclo 2 y se expresa como:

$$A = T_2 * n + T_2 * (n-1) + ... + T_2 * 2 + T_2 * 1 <= T_2 * n^2$$

Con esta última igualdad podemos decir que:

$$T(n) <= T_2 * n^2 + T_1 * n$$
$$T(n) = O(n^2)$$

Nuevamente volvemos a ver que es un orden cuadrático. Y veamos en la práctica:

- **Mejor Caso:** $O(n^2)$
- **Peor Caso:** $O(n^2)$
- **Caso Promedio:** $O(n^2)$

Malo malo... no ha sido mejor que el fuerza bruta.

Desafío: Realizar el intercambio de posiciones, es decir:

```
      int auxiliar = arreglo[máximo];
      arreglo[máximo] = arreglo[nEls-1];
      arreglo[nEls-1] = auxiliar;
```

Pero sin utilizar una variable auxiliar. Juega con las matemáticas.

Algoritmo de la Burbuja (Bubblesort)

La desventaja de utilizar selección y reemplazo es que en arreglos muy grandes, tarda mucho en ordenarlos. Así que los grandes genios de la computación estuvieron buscando otro similar y tan fácil de aprender como el de selección para así dar a luz este algoritmo llamado de la Burbuja.

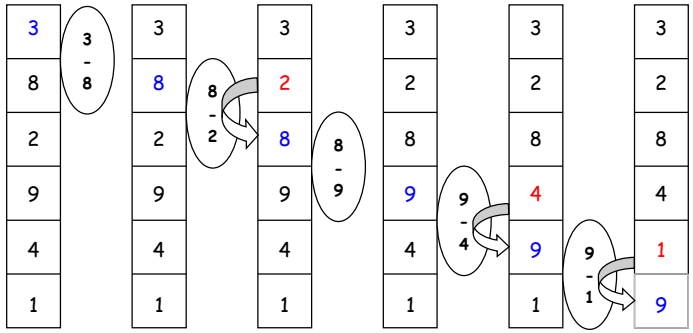
La idea es muy similar al de selección, de ir intercambiando los valores hasta dejar el máximo en el último lugar del arreglo. La diferencia está en que no solo intercambia el máximo con el último, sino que va "desplazando" los candidatos a máximo hasta posiciones más avanzadas de las que se encuentran, es decir, va ordenando parcialmente.

El algoritmo (paso general) quedaría algo como:

1. Ordenar entre 0 y nEls:
   a. Tomar el primero como potencial máximo dentro de una burbuja.
   b. Si el segundo es menor que el que tengo en la burbuja, intercambiar por el potencial máximo. Si el segundo es mayor que el de la burbuja, poner el segundo ahora como potencial máximo.
   c. Si el tercero es menor que el que tengo en la burbuja, intercambiar por el potencial máximo. Si el tercer es mayor que el de la burbuja, poner el tercero ahora como potencial máximo.
   d. ...
   e. Si el siguiente es menor que el que tengo en la burbuja, intercambiar por el potencial máximo. Si el siguiente es mayor que el de la burbuja, poner el siguiente ahora como potencial máximo.

f. Al último, reventar la burbuja, y volver a repetir todo el procedimiento hasta nEls-1.

¡Hey, es como super raro!... Veamoslo gráficamente:



A modo explicativo suele ser complejo escribir una descripción verbal... Pero no es difícil de entender al programarlo. Veámoslo en forma iterativa primero:

```
void ordenarArreglo (int arreglo[], int nEls) {
      for (int i=nEls-1; i>0; i--) {
            for (j=0; j<i-1; j++) {
                  if (arreglo[j] > arreglo[j+1]) {
                        int auxiliar = arreglo[j];
                        arreglo[j] = arreglo[j+1];
                        arreglo[j+1] = auxiliar;
                  }
            }
      }
}
```

Como podemos ver usamos mucho menos código, pues lo que vamos haciendo es ir desplazando al máximo desde donde esté, encontrándolo en el camino, y dejándolo al final, sin andarlo "buscando" antes.

La forma recursiva es bastante sencilla, y es:

```
void ordenarArreglo (int arreglo[], int nEls) {
      // Arreglo de 1 elemento está ordenado
      if (nEls == 1)
            return;

      for (j=0; j<nEls-2; j++) {
            if (arreglo[j] > arreglo[j+1]) {
                  int auxiliar = arreglo[j];
                  arreglo[j] = arreglo[j+1];
                  arreglo[j+1] = auxiliar;
            }
      }
```



```
      ordenarArreglo(arreglo, nEls-1);
}
```

Difiere en casi nada la forma general. De hecho, el for de i que estaba antes casi ni se nota que no está, cosa que en el algoritmo de **selección** no ocurría, porque en el intercambio si influía.

Midamos la eficiencia de este algoritmo:

```
void ordenarArreglo (int arreglo[], int nEls) {
      // Ciclo 1
      for (int i=nEls-1; i>0; i--) {
            // Ciclo 2
            for (j=0; j<i-1; j++) {
                  if (arreglo[j] > arreglo[j+1]) {
                        int auxiliar = arreglo[j];
                        arreglo[j] = arreglo[j+1];
                        arreglo[j+1] = auxiliar;
                  }
            }
      }
}
```

Veamos el análisis:

$$T(n) = T_0 + n * T_1 + A$$

donde A es la cantidad de veces que se ejecutó el ciclo 2 y se expresa como:

$$A = T_2 * n + T_2 * (n-1) + ... + T_2 * 2 + T_2 * 1 <= T_2 * n^2$$

Con esta última igualdad podemos decir que:

$$T(n) <= T_2 * n^2 + T_1 * n$$
$$T(n) = O(n^2)$$

Nuevamente volvemos a ver que es un orden cuadrático. Pero en este caso, si vemos la práctica:

- **Mejor Caso:** $O(n)$
- **Peor Caso:** $O(n^2)$
- **Caso Promedio:** $O(n^2)$

¿Por qué ha cambiado el mejor caso? Pues porque ese caso está condicionado por un **if** que puede anular T_2 cuando está ordenado el arreglo (jamás entra). No es tan malo después de todo, ya que en arreglos semi-ordenados, burbuja puede ser más eficiente, ya que no pasará el 100% de las veces por el código del ciclo 2.

Pero veamos realmente en serio cómo podemos ser eficaces al momento de ordenar un arreglo:

MergeSort

Este algoritmo se basa en el principio de "dividir para reinar", es decir, va dividiendo el arreglo en problemas más pequeños para realizar el ordenamiento.

Es recursivo y funciona bastante bien. Veamos cómo lo hace:

Caso Base:

- Si el arreglo tiene uno o ningún elemento, está ordenado.

Caso General:

- Corta por la mitad el arreglo
- Ordena cada mitad con MergeSort
- Mezcla las mitades para que queden ordenadas.

Ahora, escribamos el algoritmo en líneas de código con Java:

```
void mergeSort (int a[], int iMin, int iMax) {
	// Caso Base
	if(iMin >= iMax) {
		return;
	}

	// Cortamos para aplicar mergeSort recursivamente
	int k = (iMin+iMax) / 2;
	mergeSort(a, iMin, k);
	mergeSort(a, k+1, iMax);

	// Utilizamos un arreglo temporal
	int l = iMax-iMin+1;
	int temp[] = new int[l];
	for(int i = 0; i < l; i++) {
		temp[i] = a[iMin+i];
	}

	// Mezclamos
	int i1 = 0;
	int i2 = k-iMin+1;
	for(int i = 0; i < l; i++) {
		if(i2 <= iMax-iMin) {
			if(i1 <= k-iMin) {
				if(temp[i1] > temp[i2]) {
					a[i+iMin] = temp[i2++];
				}
				else {
					a[i+iMin] = temp[i1++];
				}
			}
			else {
				a[i+iMin] = temp[i2++];
			}
		}
		else {
			a[i+iMin] = temp[i1++];
		}
	}
}
```


¡Es como mezclar 2 trozos de una baraja de cartas (como en Las Vegas)!

Si vemos el análisis de orden llegaremos a:

$$T(n) = O(n \log_2 n)$$ en el caso promedio

QuickSort

Este algoritmo, más eficiente que los anteriores, se ha desarrollado bajo recursividad. Su nombre nos indica (y hasta ahora es así) que es el más óptimo que existe.

La idea es que QuickSort toma un elemento dentro de el arreglo como pivote y luego pasa todos los elementos menores que el pivote a la izquierda y los mayores que el pivote a la derecha. Es decir debe cumplir que:

$A[i] < \text{pivote}$, para todo $i < k$
$A[i] = \text{pivote}$, para $i = k$
$A[i] > \text{pivote}$, para todo $i > k$

Pero no significa que a cada lado del pivote esté ordenado. Veamos como funciona el código:

```
void quickSort (int arreglo[], int iMin, int iMax) {
	// Caso base
	if (iMin >= iMax)
		return;

	// Caso general
	int k = particionar(arreglo, iMin, iMax);
	quickSort(arreglo, iMin, k-1);
	quickSort(arreglo, k+1, iMax);
}
```

Este método ordena un arreglo entre los índices **iMin** y **iMax**. Es sumamente corto, ya que la mayor parte del trabajo la hace el método particionar:

```
int particionar (int a[], int iMin, int iMax) {
	int iPiv = iMin;
	int k = iMin;
	int j = k + 1;

	while(j <= iMax) {
		if (a[j] < a[iPiv] ) {
			// Vamos poniendo el pivote en medio
			k = k+1;
			int aux = a[k];
			a[k] = a[j];
			a[j] = aux;
		}
		j++;
	}
	int aux = a[k];
	a[k] = a[iPiv];
```

```
        a[iPiv] = aux;
        return k;
}
```

El cómo funciona es bien simple.

1. Se toma la posición **iMin** como pivote.
2. En cada iteración con j mayor que el pivote y menor que **iMax**, se compara el a[j] con el pivote.
    a. Si es menor, se hace crecer k en una posición y se intercambia el elemento k con el elemento j, para que se cumpla la situación de:

$$a[i] < \text{pivote si } i < k$$
$$a[i] >= \text{pivote si } i >= k$$

3. En ambos caso se incrementa j para continuar y se vuelve a 2.
4. Al final del ciclo se intercambia el a[k] por el pivote y queda la situación requerida.
5. Retornamos el pivote.

Con esto vamos ordenando "relativamente" cada trozo de arreglo. En cada recursión el tamaño de los trozos se va achicando, hasta llegar al mínimo: 1 elemento ordenado.

Si analizamos este algoritmo, tendremos que:

$$T(n) = O(n \log_2 n)$$

Y si vemos la práctica:

- **Mejor Caso:** ?
- **Peor Caso:** $O(n^2)$
- **Caso Promedio:** $O(n \log_2 n)$

Es realmente mejor en el caso promedio que los anteriores. Hasta ahora el mejor.

Ejemplo

Para todos los casos de ordenamiento que ya hemos visto, cada uno tiene su firma distintiva. Veamos cada una:

- **void selectSort (int[] a, int n)**: Algoritmo de selección donde **a** es el arreglo y **n** es la cantidad de elementos.
- **void bubbleSort (int[] a, int n)**: Algoritmo de burbuja donde **a** es el arreglo y **n** es la cantidad de elementos.
- **void mergeSort (int[] a, int iMin, int iMax)**: Algoritmo MergeSort donde **a** es el arreglo, **iMin** es donde empieza el arreglo (0) y **iMax** es donde termina el arreglo (cantidad de elementos-1).

- **void quickSort (int[] a, int iMin, int iMax)**: Algoritmo QuickSort donde **a** es el arreglo, **iMin** es donde empieza el arreglo (0) y **iMax** es donde termina el arreglo (cantidad de elementos-1).

Podemos hacer una ligera modificación a los algoritmos iniciales, para que los 4 sean similares. Veamos como es eso:

- **void selectSort (int[] a, int iMin, int iMax)**
- **void bubbleSort (int[] a, int iMin, int iMax)**
- **void mergeSort (int[] a, int iMin, int iMax)**
- **void quickSort (int[] a, int iMin, int iMax)**

Es decir, con estos 4 método puedo ordenar trozos de un arreglo. En su defecto, bastaría que hiciera:

```
selectSort(a, 0, a.length-1);
bubbleSort(a, 0, a.length-1);
mergeSort(a, 0, a.length-1);
quickSort(a, 0, a.length-1);
```

Para ordenar el arreglo a de las 4 formas distintas.

Solución al Problema

Con los algoritmos de ordenamiento y sus formas iterativas o recursivas, tenemos en total 8 soluciones al problema que pueden ser utilizadas en forma indistinta (solo debemos cambiar el INT por DOUBLE y listo).

Ya no es necesario pensar en un problema específico para resolverlo, pues si el problema hubiese sido otro, no importa cuál es, porque los métodos de ordenamiento son completamente IGUALES.

Conceptos

Ahora que ya hemos revisado como ordenar un arreglo, es muy importante aprender como también buscar dentro de un arreglo. Por eso definimos:

Búsqueda

> *Algoritmo que permite encontrar valores dentro de una lista (ordenada) de ellos.*

Es sencillo, ya que lo hemos utilizado antes.

Búsqueda Secuencial

Veamos como encontrar un valor dentro de un arreglo en forma ineficiente:

```
int buscar (int a[], int nEls, int x) {
      int i = -1;
```

```
        for (int j = 0; j < nEls; j++)
                if (a[j] == x)
                        i = j;
        return i;
}
```

Este pequeño código busca en forma secuencial un valor **x** dentro del arreglo **a** y retorna la posición en la cual se encuentra, Sencillo de ver, entender y escribir, pero es ineficiente. Fíjense en el mejor de los casos (está en la primera posición) igual se recorre TODO el arreglo para encontrarlo.

- **Mejor Caso:** n iteraciones
- **Peor Caso:** n iteraciones
- **Caso Promedio:** n iteraciones

Vamos a ver algunas optimizaciones para corregir esto.

```
int buscar (int a[], int nEls, int x) {
        int i = -1;
        int j = 0;
        while(j < nEls) {
                if (a[j] == x) {
                        i = j;
                        break;
                }
                j++;
        }
        return i;
}
```

Con este pequeño cambio podemos bajar el tiempo de búsqueda:

- **Mejor Caso:** 1 iteración
- **Peor Caso:** n iteraciones
- **Caso Promedio:** n/2 iteraciones aproximadas

Esto mejora un poco nuestro algoritmo. Pero aún existe una forma más óptima de buscar un elemento.

Búsqueda Binaria

Este tipo de búsquedas se basa en el mismo concepto que utiliza MergeSort y QuickSort: El hecho de subdividir el arreglo para optimizar la búsqueda. Su solución es recursiva, así que añadiremos un parámetro adicional en la firma:

```
int buscar (int a[], int iMin, int iMax, int x) {
        if (iMin > iMax)
                return -1;

        int iCentro = (imin + imax) / 2;
        if (a[iCentro] == x)
                return iCentro;
        else if (a[iCentro] > x)
```



```
                return buscar(a, iMin, iCentro-1, x);
        else
                return buscar(a, iCentro+1, iMax, x);
}
```

En este caso se utiliza como pivote el cortar por la mitad el trozo de búsqueda.

Es muy fácil ver que:

- **Mejor Caso:** 1 iteración
- **Peor Caso:** n/2 iteraciones
- **Caso Promedio:** $\log_2 n$ iteraciones

Si lo vemos de una forma práctica, la búsqueda binaria en muchísimo más rápida que la secuencial.

Problemas

Tenemos un archivo "notas" con las notas del control 1 con el siguiente formato:

- Código del alumno (3 caracteres)
- Nota Pregunta 1 (3 caracteres, como 6.1, 5.0, 3.9, etc)
- Nota Pregunta 2 (3 caracteres)
- Nota Pregunta 3 (3 caracteres)

Además, posee un archivo llamado "alumnos.txt" que contiene los nombres de los alumnos asociados a los códigos, es decir:

- Código del alumno (3 caracteres)
- Nombre del alumno (el resto de la línea)

Nota:

- El archivo "alumnos.txt" está ordenado LEXICOGRÁFICAMENTE.
- El archivo "notas.txt" no está ordenado.
- En total son 110 alumnos ("alumnos.txt"), pues es la lista completa. No todos los alumnos tienen notas ("notas.txt") y a ellos se les recomienda que le ponga un 1.0
- Los códigos de los alumnos parten desde 001 y terminan en 110 (1 a 110 para ser más claro).

(a) Escriba un método que modifique la burbuja para que ordene de MAYOR A MENOR el arreglo (no importa si es RECURSIVO o ITERATIVO).

Solución

Para hacer esto es cosa de cambiar la condición de orden del método:

```
void ordenarArreglo (int arreglo[], int nEls) {
        if (nEls == 1)
                return;
```

```
      for (j=0; j<nEls-2; j++) {
            if (arreglo[j] < arreglo[j+1]) {
                  int auxiliar = arreglo[j];
                  arreglo[j] = arreglo[j+1];
                  arreglo[j+1] = auxiliar;
            }
      }
      ordenarArreglo(arreglo, nEls-1);
}
```

(b) Escriba un programa que permita leer las notas de los alumnos y escriba en pantalla los promedios ordenados de MAYOR A MENOR indicando Código del Alumno y Promedio, utilizando el método anterior.

Solución

```
Console c = new Console();

// Creamos el arreglo
double notas[] = new double[110];

// Inicialmente TODOS tienen un 1.0
for (int i=0; i<110; i++)
      notas[i] = 1.0;

// Leemos el archivo
BufferedReader bf = new BufferedReader (
new FileReader("notas.txt"));
while ((String linea = bf.readLine()) != null) {
      int codigo = Integer.parseInt(linea.substring(0, 3));
      double p1 = new Double(linea.substring(3, 3)).doubleValue();
      double p2 = new Double(linea.substring(6, 3)).doubleValue();
      double p3 = new Double(linea.substring(9, 3 )).doubleValue();
      notas[codigo] = (p1 + p2 + p3) / 3;
}
bf.close();

// Ordenamos
ordenarArreglo(notas, 110);

// Desplegamos en pantalla ordenadamente
for (int i=0; i<110; i++) {
      c.println(i + " sacó un " + notas[i]);
}
```

(c) Siga con el programa para que luego permita crear un nuevo archivo llamado "lista.txt" en el cual se escriban los siguientes datos:

- Nombre del Alumno
- Espacio (1 carácter)
- Promedio del Control 1

Nótese que usted YA debe tener ordenado el arreglo de MAYOR A MENOR con las notas de los alumnos.

Solución

Continuando con el programa, no declaramos nada de nuevo, pues está todo listo.

```
PrintWriter pw = new PrintWriter (
      new FileWriter("lista.txt");

BufferedReader bf2 = new BufferedReader (
      new FileReader("alumnos.txt"));

while ((String linea = bf2.readLine()) != null) {
      // Obtenemos el código del alumno
      int codigo = Integer.parseInt(linea.substring(0, 3));

      // Escribimos en lista.txt
      pw.print(linea.substring(3));
      pw.print(" ");
      pw.println(notas[codigo]);
}

bf2.close();
pw.close();
```

(d) Escribir una interfaz llamada **EsComparable** que contenga una firma que permite a cualquier clase heredada compararse con otro de su mismo tipo. Debe poseer entonces esta firma:

public int compararCon(EsComparable b);

y que retorna:

- nº > 0 si el objeto es mayor que **b**
- 0 si el objeto es igual a **b**
- nº < 0 si el objeto es menor que **b**

Solución

```
public interface EsComparable {
      public int compararCon(EsComparable b);
}
```

(e) Escribir una clase **Entero** que implemente **EsComparable** y que permita realizar el siguiente método estático:

public static int comparar (int a, int b);

Este método internamente crea un objeto **Entero** (que debe poseer un constructor que reciba un entero y que lo almacena como variable de instancia) y lo compara con otro entero del mismo tipo (crea otro objeto de la clase **Entero** con b), retornando lo que entrega el método de la parte (a).

Solución

```
public class Entero extends EsComparable{
      protected int n;
```

```
	public Entero(int x) {
		this.n = x;
	}

	public int compararCon(EsComparable b) {
		return this.n - (Entero b).n;
	}

	static public int comparar(int a, int b) {
		Entero A = new Entero (a);
		Entero B = new Entero (b);
		return A.comparaCon(B);
	}
}
```

(f) Escribir el **QuickSort** en forma genérica para que compare con el método **comparar** que de la parte (e).

Solución

```
void quickSort (int arreglo[], int iMin, int iMax) {
	// Caso base
	if (iMin >= iMax)
		return;

	// Caso general
	int k = particionar(arreglo, iMin, iMax);
	quickSort(arreglo, iMin, k-1);
	quickSort(arreglo, k+1, iMax);
}

int particionar (int a[], int iMin, int iMax) {
	int iPiv = iMin;
	int k = iMin;
	int j = k + 1;

	while(j <= iMax) {
		if (Entero.comparar(a[j], a[iPiv]) < 0) {
			// Vamos poniendo el pivote en medio
			k = k+1;
			int aux = a[k];
			a[k] = a[j];
			a[j] = aux;
		}
		j++;
	}
	int aux = a[k];
	a[k] = a[iPiv];
	a[iPiv] = aux;
	return k;
}
```

(g) Escribir la Búsqueda Binaria (igual que el la parte c).

Solución

```
int buscar (int a[], int iMin, int iMax, int x) {
	if (iMin > iMax)
		return -1;
```



```
	int iCentro = (imin + imax) / 2;
	if (Entero.comparar(a[iCentro], x) == 0)
		return iCentro;
	else if (Entero.comparar(a[iCentro], x) > 0)
		return buscar(a, iMin, iCentro-1, x);
	else
		return buscar(a, iCentro+1, iMax, x);
}
```

Problemas Propuestos

(a) Desarrolle el método de SELECCIÓN para que en vez de ordenar de MENOR A MAYOR lo haga en forma inversa, es decir de MAYOR A MENOR.

(b) Desarrolle el método de la BURBUJA para que en vez de ordenar llevando el MAYOR dentro de la burbuja, haga el proceso inverso de llevar el MENOR a la primera posición.

Capítulo XIV: Archivos de Texto

Motivación

En Java se pueden utilizar distintos tipos de archivos para lectura y escritura. Los de más fácil acceso son los llamados Archivos de Texto.

La idea es poder obtener datos desde un archivo de texto guardado en el disco duro en vez del teclado y/o escribirlos en otro archivo de texto que también se guarda en disco duro, en vez de la famosa ventana de la Consola.

Sin embargo la utilización de los archivos es un poco engorrosa y complicada.

Sintaxis

Lectura de un Archivo de Texto

Para leer un archivo de texto en Java, existen 2 clases que son muy importantes:

1. **BufferedReader** es el tipo de dato que guarda la referencia a una ENTRADA de datos (que también se utiliza tanto para archivos como para teclado en el Java estándar).
2. **FileReader** es una clase que permite obtener un LECTOR para BufferedReader desde un Archivo de Texto.

Es así como abrir un archivo de texto para la lectura quedaría de la siguiente forma:

```
BufferedReader <var> = new BufferedReader(
                           new FileReader("<nombre de archivo>"));
```

Algo bastante feo ☹. Pero eso no es todo, sino que existe un método muy utilizado para la lectura y es **readLine()** (si, al igual que la consola).

Por lo tanto, para leer un archivo de texto de inicio a fin, tenemos el siguiente ejemplo (incompleto por supuesto):

```
// se abre el archivo
BufferedReader arch = new BufferedReader(
                          new FileReader("archivo.txt"));

// se lee la primera línea del archivo
String linea = arch.readLine();

// se repite mientras no esté en el final del archivo
while (linea != null) {

      // se procesa la línea leída desde el archivo
      <instrucciones>

      // se lee siguiente línea
      linea = arch.readLine();
}

// Se cierra el archivo
arch.close();
```


Otra forma de hacerlo es:

```
// se abre el archivo
BufferedReader arch = new BufferedReader(
                          new FileReader("archivo.txt"));

// se repite mientras no esté en el final del archivo
while (true) {
      // se lee siguiente línea
      linea = arch.readLine();

      // se verifica que no se esté al final del archivo
      if (linea == null) {
            break;
      }

      // se procesa la línea leída desde el archivo
      <instrucciones>
}

// Se cierra el archivo
arch.close();
```

Escritura de un Archivo de Texto

Para escribir un archivo de texto en Java, existen 2 clases que son muy importantes:

1. **PrintWriter** es el tipo de dato que guarda la referencia a una SALIDA de datos (que también se utiliza tanto para archivos como para pantalla en el Java estándar).
2. **FileWriter** es una clase que permite obtener un ESCRITOR para PrintWriter a un Archivo de Texto.

Es así como abrir un archivo de texto para la escritura quedaría de la siguiente forma:

```
PrintWriter <var> = new PrintWriter(
                        new FileWriter("<nombre de archivo>"));
```

Algo igualmente feo ☹. Pero eso no es todo, sino que existe dos métodos muy utilizado para la escritura y son **print(String)** y **println(String)** (si, al igual que la consola).

Por lo tanto, para escribir un archivo de texto tenemos el siguiente ejemplo:

```
// se abre el archivo
PrintWriter arch = new PrintWriter(
                       new FileWriter("archivo.txt"));

// se repite mientras hayan datos
while (<condición para terminar el archivo>) {
      // se obtiene los datos para una línea
      <instrucciones>
```

```
      // se escribe la línea en el archivo
      arch.println(datos);
}

// se cierra el archivo
arch.close();
```

Ejemplo de Archivos

Escribir un programa que lea desde el teclado una serie de notas y las escriba en un archivo. Luego, debe utilizar este archivo para obtener el promedio del curso completo.

```
//...
Console c = new Console("Lector de Notas");
c.println("Ingrese las notas de los alumnos. Termine con un 0 ");

// Trozo de programa que lee de pantalla y
// almacena en archivo c:\notas.txt
PrintWriter pw = new PrintWriter(
new FileWriter("c:\\notas.txt"));
double nota = c.readDouble();
while (nota != 0) {
      pw.println(nota);
      nota = c.readDouble();
}
pw.close();

// Trozo de programa que lee el archivo de notas
BufferedReader br = new BufferedReader(
      new FileReader("c:\\notas.txt"));
int n = 0;
double suma = 0;
String linea = br.readLine();
while(linea != null) {
      suma += Double.parseDouble(linea);
      linea = br.readLine();
      n++;
}
br.close();

// Despliega la nota
c.println("El promedio es " + (suma / n));
```

Problema

(a) Escribir un programa que simule la conexión de un usuario con nombre y clave. Realice el siguiente diálogo:

```
Inicio de Sesión

Nombre de usuario? jperez
Clave? jp
ERROR: Su clave no corresponde

Nombre de usuario? jperez
Clave? Jpa
INGRESO ACEPTADO
```


Además, considere que los nombres de usuario y claves se encuentran en un archivo llamado **claves.txt** y tiene la siguiente estructura:

```
amendoza:lskksa
jperez:Jpa
nromero:nata1.0
...
```

Hint: Para separar nombre de usuario y clave de acceso puedes utilizar:

```
// ...
// Suponemos que tenemos en linea lo leido
int i = linea.indexOf(":");
String nombre = linea.substring(0, i);
String clave = linea.substring(i+1);
// Con nombre y clave comparas los ingresados
// por el usuario
// ...
```

Solución 1

Esta solución utiliza el HINT que se entrega,

```
Console c = new Console();
c.println("Inicio de Sesión");

// Iniciamos el ciclo de sesión
while (true) {
      c.print("Nombre de Usuario?");
      String sunombre = c.readLine();
      c.print("Clave de Acceso?");
      String suclave = c.readLine();

      // Se abre el archivo de claves
      BufferedReader bf = new BufferedReader(
            new FileReader("claves.txt"));
      String linea = bf.readLine();

      String nombre;
      String clave;

      while (linea != null) {
            // Se obtiene el nombre y la clave del archivo
             int i = linea.indexOf(":");
             nombre = linea.substring(0, i);
             clave = linea.substring(i+1);

            // Se compara solo el nombre
            if (nombre.equals(sunombre))
                  break;

            // Siguiente usuario
            linea = bf.readLine();
      }
      bf.close();

      // Ahora verificamos por qué salió.
      if (linea == null)
            c.println("ERROR: El usuario no existe");
      else {
```

```
            if (clave.equals(suclave))
                  break;
            c.println("ERROR: La clave es incorrecta");
      }
}

// Como solo sale si la clave es correcta
c.println("INGRESO ACEPTADO!!!!");
```

Solución 2

Esta otra solución también funciona, pero no utiliza el juego con substrings y es algo más corta.

```
Console c = new Console();
c.println("Inicio de Sesión");

// Iniciamos el ciclo de sesión
while (true) {
      c.print("Nombre de Usuario?");
      String sunombre = c.readLine();
      c.print("Clave de Acceso?");
      String suclave = c.readLine();

      // Se abre el archivo de claves
      BufferedReader bf = new BufferedReader(
            new FileReader("claves.txt"));
      String linea = bf.readLine();

      String nombre;
      String clave;

      while (linea != null) {
            // Se compara la linea completa
            if (linea.equals(sunombre + ":" + suclave))
                  break;

            // Siguiente usuario
            linea = bf.readLine();
      }
      bf.close();

      // Ahora verificamos por qué salió.
      if (linea == null)
            c.println("ERROR: La clave es incorrecta");
      else
            break;
}

// Como solo sale si la clave es correcta
c.println("INGRESO ACEPTADO!!!!");
```

(b) Escriba un programa que reemplace textos en un archivo, es decir, que simule el siguiente diálogo:

```
Ingrese el nombre del archivo? micarta.txt
Ingrese el patrón a reemplazar? @
Ingrese valor a reemplazar? Juanita
El resultado quedó en Juanita_micarta.txt
```

La dea es que, por ejemplo si el archivo micarta.txt tiene el siguiente texto:

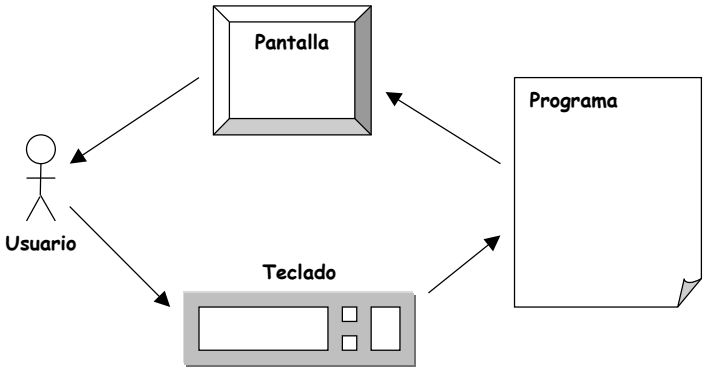

```
Mi amada @:

A través de la presente carta he querido invitarte para que mañana
podamos ir al cine y luego tal vez quien sabe.

Así que, por favor @, contéstame este mail y te espero...

Siempre tuyo
Tu amado

PS: @, no olvides llevar plata, porque no tengo... :)
```

El resultado de cambiar "@" por "Juanita" entregaría el siguiente archivo Juanita_micarta.txt:

```
Mi amada Juanita:

A través de la presente carta he querido invitarte para que mañana
podamos ir al cine y luego tal vez quien sabe.

Así que, por favor Juanita, contéstame este mail y te espero...

Siempre tuyo
Tu amado

PS: Juanita, no olvides llevar plata, porque no tengo... :)
```

Capítulo XV: Interfaces Gráficas AWT

Motivación



Hasta ahora, todos los programas interactúan con el usuario tal como lo dice la figura[12]: Mostrar datos en pantalla y pedir datos por el teclado.

Nuestro foco ha estado todo este tiempo en lo que son los programas y solo hemos hecho interacciones sencillas con el usuario utilizando una herramienta llamada "Console". Pero ¿Qué es Console? ¿Cómo funciona? ¿Podemos hacer otro tipo de ventanas más bonitas?

Es hora de llevar nuestra atención a realizar interfaces distintas a Console, gráficas, y que puedan sernos de mucha utilidad.

Conceptos

Interfaz Gráfica

> *Programa (en Java) que permite una interacción con el usuario a través de una ventana con botones y otras componentes haciendo más amigable la interacción con el Usuario.*

Las interfaces gráficas son programas son componentes que nos permiten trabajar directamente con gráficos como botones, textos, etc.

[12] Se refiere solo a la interacción humano-computador, es decir, usuario del programa con el computador. Existen más interacciones como por ejemplo con Archivos y Bases de Datos, pero que no involucran al usuario directamente.

Sintaxis

Para la utilización de interfaces gráficas es obligatorio utilizar unos paquetes de clases que se encuentran en la API del JDK (Java Delevopment Kit). Es por eso que todos los programas Java que utilizan interfaces gráficas empiezan con la importación de:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class MiInterfazGrafica {
       ...

       public MiInterfazGrafica() {
              // Creamos la ventana y las componentes
              ...
       }

       ...
}
```

Con esto nos aseguramos que nuestra clase Java pueda crear ventana gráficas y escribir en ellas (como lo hacemos en la Console) o dibujar sobre ellas.

Para ejecutar esta interfaz gráfica es importante recalcar que se debe crear una clase que cree nuestra interfaz personalizada (al igual que la console):

```
public class MiPrograma {
       static public void main (String args[]) {
              MiInterfazGrafica ig = new MiInterfazGrafica();
       }
}
```

Como se ve en el ejemplo, lo que hacemos realmente es crear una clase que almacenará nuestra interfaz dentro de si las componentes y todo. La idea es que el contructor de la clase sea el constructor de la interfaz y que permita que se refleje en pantalla.

Conceptos

Componente

> *Elemento u objeto (en Java) que se utiliza para construir una interfaz gráfica.*

Los componentes pueden ser botones, áreas de texto, áreas de dibujo, imágenes, listas de valores y cualquier tipo de elemento gráfico (de Java) que se puede insertar dentro de una interfaz.

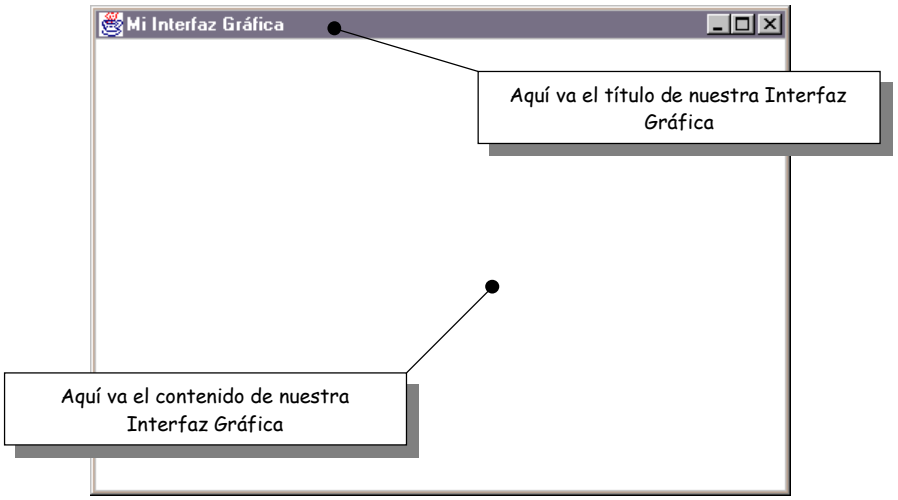
Estos elementos son sensibles a "ocurrencias" de la interfaz llamadas "eventos".

Sintaxis

Existen muchos componentes para las interfaces gráficas[13]:

Frame

Un frame es un área que nos permitirá dentro de ella crear la interfaz gráfica. Practicamente es la ventana que se abre al ejecutar nuestra clase:



Esta ventana es producida por el siguiente código:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class MiInterfazGrafica {
	private Frame ventana;
	public MiInterfazGrafica() {
		ventana = new Frame("Mi Interfaz Gráfica");
		ventana.pack();
		ventana.show();
	}
}
```

Entonces, podemos ver que la componente Frame posee varios métodos que aquí se describen[14]:

Método	Descripción
Frame()	Constructor sin parámetros que crea una ventana de tamaño 0x0 (sin área de contenido) y sin título definido.

[13] Mayor información en la API del JDK: http://java.sun.com/j2se/1.3/docs/api/index.html
[14] Los parámetros se encuentran en el mismo orden de aparición que en la descripción.

Método	Descripción
Frame(String)	Constructor que crea una ventana de tamaño 0x0 y con el título definido por el parámetro.
void setSize(int, int)	Dimensiona el frame para que tenga un tamaño (x, y) en pixeles reflejados en los 2 parámetros del método.
void setResizable(boolean)	Hace que el frame quede sin permiso para redimensionar el tamaño con el mouse.
void setLayout(Layout)	Le indica la grilla que va a utilizar el frame para contener los componentes. Estas grillas serán descritas más adelante.
void add(Component)	Pone la componente dentro del frame en la primera posición libre del mismo (dependiendo de la grilla).
void add(String, Component)	Pone la componente dentro del frame y en la posición de la grilla indicada en el parámetro String.
void addXListener(XListener)	Agrega un listener para ejecutar una acción cuando ocurre el evento X. X va variando dependiendo del listener (lo veremos más adelante).
void pack()	Prepara el despliegue del frame en pantalla.
void show()	Muestra el frame en pantalla.

Layout

Las grillas o Layout son utilizados dentro de los frames para darle "espacios disponibles" para poner un Componente. Para que un frame pueda contener "componenter" es necesario definir cuántos puede contener y luego empezar a generarlos.

Existen distintos tipos de grilla (LayoutManager es la clase que define una grilla) y las veremos con dibujos a continuación:

GridLayout: Este layout corresponde a una distribución cuadriculada (tipo planilla excel) y que ocupan el mismo tamaño todos los cuadritos.

Por ejemplo, esta grilla tiene 6 posiciones. Para crear un Frame con esa distribución, se debe escribir:

```
...
f. setLayout(new GridLayout(3,2));
...
```

donde **f** es el frame que quiero darle esa grilla. Una vez que se setea, se van poniendo en orden las componentes con cada **void add(Component)** que se va

realizando en el constructor de la IG.

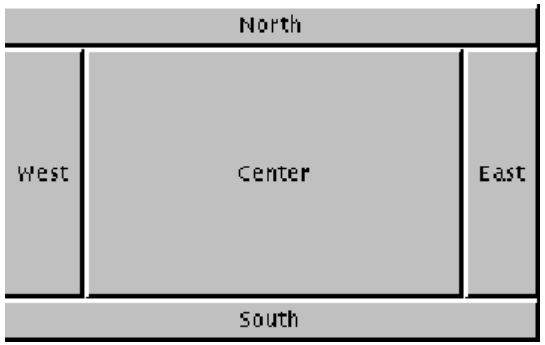
FlowLayout: Este layout permite en general poner un arreglo de componentes uno al lado del otro pero con tamaño variable de cada uno, como si se estuviera poniendo una cajita al lado de otra.

Por ejemplo, este arreglo de botones se puede utilizar dentro de una IG y se declara de la forma:

```
...
f.setLayout(new FlowLayout());
...
```

donde **f** es el frame que se quiere usar. Una vez que se setea, se van poniendo en orden las componentes con cada **void add(Component)** que se va realizando en el constructor de la IG.

BorderLayout: Este layout es el más elaborado y utiliza una distribución de puntos cardinales para las componentes que se van insertando.



Para crear una grilla como esta se debe hacer:

```
...
f.setLayout(new BorderLayout());
...
```

donde **f** es el frame al cual se le asigna esa grilla. Una vez que se setea la grilla, las componentes deben ser asignadas según dónde van con **add(pos, Componente)**. La posición es un String que corresponde al área deseada "North", "South", etc.

Existen otros Layouts más complejos, pero con una composición de estos 3 se puede crear lo que uno desee.

Panel

Los paneles son componentes "contenedoras" que nos permiten crear sub-interfaces dentro de nuestros Frames. El uso de paneles es una forma muy útil, ya que para componer o mezclar distintos Layouts de pantalla, utilizamos paneles.

También, dentro del panel podemos trabajar como si fuese un frame, es decir, agregamos componentes a libre elección según el layout seleccionado.

La definición de un panel se hace a través de objetos de la clase **Panel**[15]:

Método	Descripción
Panel()	Constructor del panel.
void setLayout(Layout)	Le indica la grilla que va a utilizar el panel para contener los componentes.
void add(Component)	Pone la componente dentro del panel en la primera posición libre del mismo (dependiendo de la grilla).
void add(String, Component)	Pone la componente dentro del panel y en la posición de la grilla indicada en el parámetro String.

Un ejemplo de uso sería el siguiente programa:

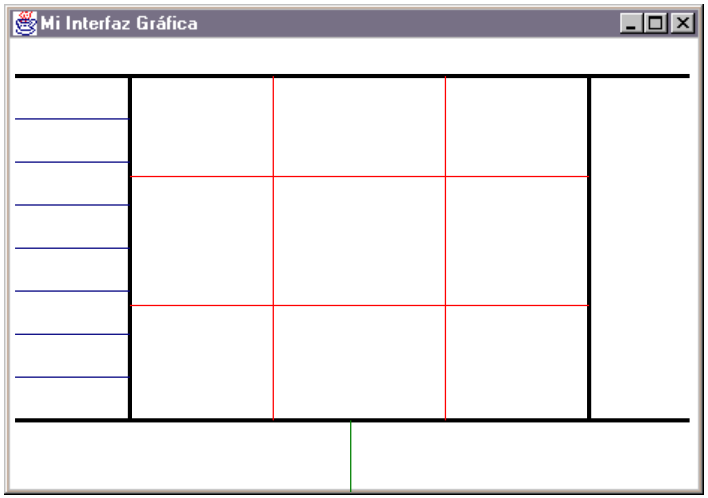
```
public class MiInterfazGrafica {
	// Frame
	private Frame ventana;

	// Paneles
	private Panel p;

	public MiInterfazGrafica() {
		ventana = new Frame("Mi Interfaz Gráfica");
		ventana.setLayout(new GridLayout(4, 4));

		p = new Panel();
		p.setLayout(new BorderLayout());

		...  // Creación de la interfaz

		ventana.add(p);

		ventana.pack();
		ventana.show();
	}
}
```

Como podemos ver en el ejemplo, estamos creando un frame de distribución cuadrada de 4x4 y con un panel en la primera casilla con distribución de BorderLayout.

[15] Panel posee más métodos, pero para efectos académicos no nos serán útiles.

Importante: Recuerda que para crear interfaces más complejas, la idea es siempre ir componiendo a través de paneles para llegar a los Layouts básicos.

Imaginemos que queremos la siguiente distribución:



Podemos distinguir claramente varios paneles dentro de este frame[16]:

- El Layout del frame es BorderLayout (marcado más grueso)
- En el centro hay un Panel con GridLayout de 3 x 3.
- En la izquierda hay un Panel con GridLayout de 8 x 1.
- Abajo puede ser un Panel con GridLayout de 1 x 2 o FlowLayout.
- A la derecha y arriba no es necesario un Panel.

y eso quedaría escrito cómo:

```
public class MiInterfazGrafica {
	private Frame ventana;
	private Panel p1, p2, p3;

	public MiInterfazGrafica() {
		ventana = new Frame("Mi Interfaz Gráfica");
		ventana.setLayout(new BorderLayout());

		p1 = new Panel();
		p1.setLayout(new GridLayout(3, 3));
		ventana.add("Center", p1);
```

[16] Todas estas líneas son imaginarias, ya que la distribución se ve reflejada en las componentes que cada panel contiene realmente.


```
		p2 = new Panel();
		p2.setLayout(new GridLayout(8, 1));
		ventana.add("West", p2);

		p3 = new Panel();
		p3.setLayout(new FlowLayout());
		ventana.add("South", p3);

		ventana.pack();
		ventana.show();
	}
}
```

Ahora veamos otras componentes para que vayamos creando nuestras interfaces:

Label

Las áreas de texto o etiquetas son súmamente útil en todas las interfaces, ya que pueden servirnos de informativos o más bien de títulos para algunas cosas. La clase **Label** posee los siguientes (reducidos) métodos:

Método	Descripción
Label(String)	Constructor del label que pone el texto en su contenido.
Label(String, int)	Constructor del label que pone el texto en su contenido y lo alínea según el segundo parámetro. El alineamiento que va como parámetro es identificado como: • Label.CENTER • Label.LEFT • Label.RIGHT
String getText()	Retorna el texto que posee el label.
void setText(String)	Cambia el texto original por otro que va como parámetro.
void setAlignment(int)	Alinea el texto dentro del label. El alineamiento que va como parámetro es identificado como: • Label.CENTER • Label.LEFT • Label.RIGHT

Ejemplo:

```
// Crear un label con un texto fijo
Label titulo = new Label("Este es un título");

// Cambiar el texto del label
titulo.setText("Cambio de título");

// Alinea a la derecha el label
titulo.setAlignment(Label.RIGHT);

// Crea un nuevo label, centrado, con el texto del otro
Label tit2 = new Label(titulo.getText(), Label.CENTER);
```

TextField

Son campos de ingreso de datos de una sola línea.

La clase **TextField** posee los siguientes (reducidos) métodos:

Método	Descripción
TextField()	Constructor de un textfield vacío.
TextField(int)	Constructor de un textfield de largo definido.
TextField(String)	Constructor de un textfield con un texto definido.
TextField(String, int)	Constructor de un textfield con un text y largo definido.
void setColumns(int)	Fija el largo del textfield.
void setText(String)	Pone el texto como contenido del textfield.
String getText()	Retorna el texto del textfield.
void setEditable(boolean)	Configura para que el textfield sea editable (TRUE) o solo de lectura (FALSE).
boolean isEditable()	Retorna si el textfield es editable (TRUE) o no (FALSE).

Ejemplo:

```
// Crea un textfield de 20 caracteres de largo
TextField tf = new TextField(20);

// Lo pone como solo de lectura
tf.setEditable(false);

// Escribe en el textfield un texto
tf.setText("Este texto es fijo");
```

Button

Las componentes buttons son sencillos botones de acción que se utilizan en las interfaces gráficas como gatilladores de eventos específicos (ver sección de eventos más adelante).

La clase **Button** posee los siguientes métodos:

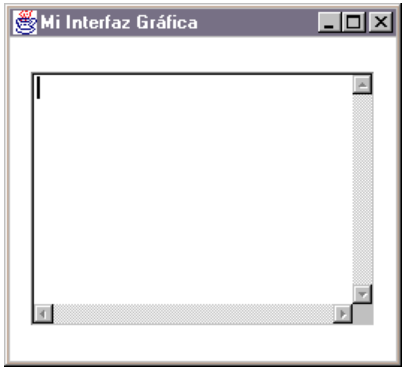
Método	Descripción
Button()	Constructor de un botón vacío.
Button(String)	Constructor que da a un botón un texto como etiqueta.
void setLabel(String)	Asigna a un botón una etiqueta específica.
void setEnable(boolean)	Activa (TRUE) o desactiva (FALSE) el botón.
boolean isEnabled()	Retorna si el botón está activado (TRUE) o desactivado (FALSE).

Por ejemplo:

```
// Creamos un botón con texto
Button b = new Button ("Aceptar");

// Desactivamos el botón
b.setEnable(false);
```

TextArea

Los textarea son áreas de texto en donde se pueden escribir múltiples líneas, a diferencia de los textfield.



La clase **TextArea** posee los siguientes métodos:

Método	Descripción
TextArea()	Constructor de un textarea vacío.
TextArea(int, int)	Constructor de un textarea de largo definido como filas x columnas.
TextArea(String)	Constructor de un textarea con un texto definido.
TextArea(String, int, int)	Constructor de un textarea con un text y largo definido con filas x columnas.
TextArea(String, int, int, int)	Constructor de un textarea con un text, largo definido con filas x columnas y los scrollbars: ① TextArea.SCROLLBARS_BOTH

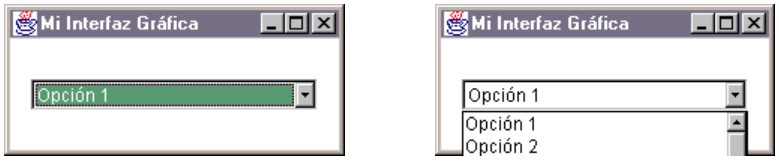
Método	Descripción
	• TextArea.SCROLLBARS_HORIZONTAL_ONLY • TextArea.SCROLLBARS_VERTICAL_ONLY • TextArea.SCROLLBARS_NONE
void setColumns(int)	Fija la cantidad de columnas del textarea.
void setRows(int)	Fija la cantidad de filas del textarea.
void setText(String)	Pone el texto como contenido del textarea.
String getText()	Retorna el texto del textarea.
void setEditable(boolean)	Configura para que el textarea sea editable (TRUE) o solo de lectura (FALSE).
boolean isEditable()	Retorna si el textarea es editable (TRUE) o no (FALSE).

Por ejemplo, para general la ventana que aparece como ejemplo, es necesario crear el textarea de la siguiente forma:

```
Textarea area = new TextArea(10, 30);
```

Choice

El choice es un elemento de selección que puede ser tomado desde una lista más grande de datos.



Una vez que uno presiona la flecha del costado derecho, aparece la lista completa de valores del choice. La clase **Choice** se define como:

Método	Descripción
Choice()	Constructor de un choice vacío.
void add(String)	Agrega un nuevo elemento al choice (al final).
void insert(String, int)	Inserta un nuevo elemento al choice en la posición indicada.
int getItemCount()	Retorna el número de elementos que tiene el choice.
int getSelectedIndex()	Retorna el índice del elemento seleccionado.
String getSelectedItem()	Retorna el valor del elemento seleccionado.
String getItem(int)	Retorna el elemento de la posición indicada.
void select(int)	Selecciona el elemento de la posición indicada.
void select(String)	Selecciona el elemento indicado.

Por ejemplo

```
Choice c = new Choice();
```

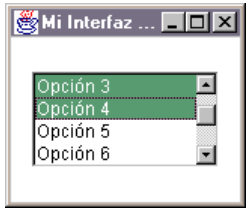


```
for (int i=2000; i<2100; i++) {
      c.add("Año " + i);
}
```

Esta lista entregaría todos los años entre el 2000 y el 2099.

List

Las listas son uno de los elementos más útiles después de los botones y las áreas de texto.



La clase **List** del awt posee la siguiente definición:

Método	Descripción
List()	Constructor de una lista vacía con un número de líneas visibles igual a 4 con selección única.
List(int)	Constructor de una lista con un número de líneas visibles definido y selección única.
List(int, boolean)	Constructor de una lista con un número de líneas visibles definido y con opción de selección única (FALSE) o múltiple (TRUE).
void add(String)	Agrega una nueva opción al final de la lista.
void add(String, int)	Agrega una nueva opción en la posición indicada.
void select(int)	Selecciona el elemento de la posición indicada.
boolean isIndexSelected(int)	Indica si una posición está (TRUE) o no (FALSE) seleccionada.
int getItemCount()	Retorna la cantidad de elementos de la lista.
int getSelectedIndex()	(Modo de selección simple) Entrega el índice del elemento seleccionado.
int[] getSelectedIndexes()	(Modo de selección múltiple) Entrega un arreglo con todos los índices seleccionados.

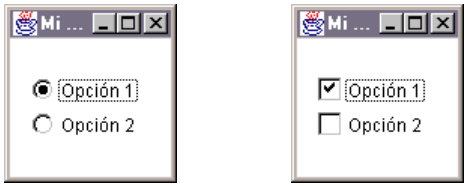
Por ejemplo, si queremos una lista con muchas opciones:

```
List l = new List(10, false);
for (int i=0; i<100; i++) {
      l.add("Opción " + i);
}
```

Esta lista entregaría 100 opciones con selección simple.

Checkbox y CheckboxGroup

Los checkboxes son opciones que permiten marcar un texto específico y que le permiten al programador interactuar con opciones (alternativas). Dependiendo si se usan la opción de grupo checkboxgroup, es dependiendo si se puede o no marcar múltiples opciones.



La clase **Checkbox** identifica solo una opción de selección con estos dispositivos y posee la siguiente definición:

Método	Descripción
Checkbox()	Crea una opción vacía.
Checkbox(String)	Crea una opción con un texto definido.
Checkbox(String, boolean)	Crea una opción con un texto y le indica si está o no marcada.
Checkbox(String, boolean, CheckboxGroup) o bien Checkbox(String, CheckboxGroup, boolean)	Crea una opción con un texto, indicando si está o no marcada y además agrupada según un grupo de opciones. Al asignarlas al mismo grupo, las opciones quedan con la posibilidad de marcar SOLO 1 de ellas, y su forma cambia a la de selector redondo en vez del cuadrado (como se ve en la figura). Si no se desea ningún grupo, esta opción debe ser "null".
void setLabel(String)	Pone el texto a la etiqueta que acompaña al selector.
boolean getState()	Retorna si la opción si está seleccionada (TRUE) o no (FALSE).
void setCheckboxGroup(CheckboxGroup)	Asigna al grupo la opción.
CheckboxGroup getCheckboxGroup()	Obtiene el grupo al que pertenece la opción.

Por ejemplo, si queremos un selector de sexo de una persona (si es Masculino o Femenino) hacemos:

```
CheckboxGroup sexo = new ChekboxGroup();
Checkbox sexoM = new Checkbox("Masculino", sexo, true);
Checkbox sexoF = new Checkbox("Femenino", sexo, false);
```


Pero si queremos sabes por ejemplo los grupos de interés de un usuario, podemos hacer un grupo de selección diferenciada:

```
Checkbox gr01 = new Checkbox("Deportes", null, false);
Checkbox gr02 = new Checkbox("Música", null, false);
Checkbox gr03 = new Checkbox("Televisión", null, false);
...
```

Canvas

Un canvas es un rectángulo blanco dentro de la interfaz en donde se puede dibujar cualquier cosa (texto, líneas, polígonos) y se puede atrapar eventos sobre él.

La clase **Canvas** está definida como:

Método	Descripción
Canvas()	Constructor que crea un canvas.
void paint(Graphics)	Permite re-dibujar en el canvas lo que esté almacenado en él.

Por ser un tema especial, canvas lo dejaremos para más adelante.

Ejemplos de Componentes

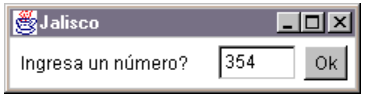
Un primer ejemplo de utilización de componentes es el juego del gato (solo interfaz):

Esta interfaz sencillita, es resuelta por el siguiente programa:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Gato {
	private Frame ventana;

	// Acá van solo las componentes que son útiles
	// para el programa
	private Choice jugadas[][];
	private Button salir;
```

```
        public MiInterfazGrafica() {
                ventana = new Frame("Juego del Gato");
                ventana.setLayout(new GridLayout (3, 1));

                // Le ponemos el título a nuestro ejemplo
                Label titulo = new Label("Gato", Label.CENTER);
                ventana.add(titulo);

                // Dibujamos el tablero para jugar gato
                Panel gato = new Panel();
                gato.setLayout(new GridLayout(3, 3));

                Choice jugadas[][] = new Choice[3][3];
                for(int i=0; i<3; i++) {
                        for (int j=0; j<3; j++) {
                                jugadas[i][j] = new Choice();
                                jugadas[i][j].add(" ");
                                jugadas[i][j].add("X");
                                jugadas[i][j].add("O");
                                gato.add(jugadas[i][j]);
                        }
                }

                ventana.add(gato);

                // Y el botón de salida del juego
                salir = new Button("Terminar");
                ventana.add(salir);

                ventana.pack();
                ventana.show();
        }
}
```

Y por supuesto, nuestro amigo y hermano ejemplo: El programa Jalisco.



Que se resuelve con el siguiente código:

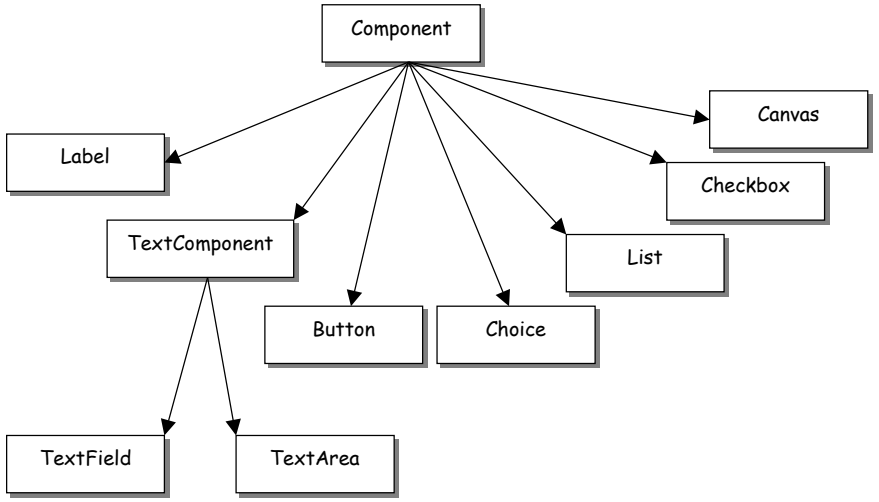
```
public class Jalisco {
        private Frame ventana;

        private TextField numero;
        private Button ok;

        public MiInterfazGrafica() {
                ventana = new Frame("Jalisco");
                ventana.setLayout(new FlowLayout());

                // Esto nos puede evitar tener una variable para el
                // texto que no nos sirve más.
                ventana.add(new Label("Ingresa un número? "));

                numero = new TextField(4);
                ventana.add(numero);
```



```
                ok = new Button("Ok");
                ventana.add(ok);

                ventana.pack();
                ventana.show();
        }
}
```

Propiedades heredadas de Component

Todas las componentes están definidas como clases dentro de la estructura de la API. Es por eso que todas ellas tienen una superclase llamada **Component** y que posee algunas propiedades y funcionalidades especiales que afectan a todas.



La clase **Component** posee las siguientes funcionalidades útiles[17]:

Método	Descripción
void setVisible(boolean)	Hace aparecer (TRUE) o desaparecer (FALSE) la componente.
boolean isVisible()	Verifica si la componente está (TRUE) o no (FALSE) visible.
void setSize(int, int)	Le da un tamaño dado por largo x ancho para la componente.
void setSize(Dimension)	Le da un tamaño dado por un objeto a un tipo **Dimension**.
Dimension getSize()	Obtiene en un objeto **Dimension** el tamaño de la componente.
void setLocation(int, int)	Pone la componente en la posición (x, y).
void setLocation(Point)	Pone la componente en la posición dada por el objeto **Point**.
Point getLocation()	Obtiene la posición de la componente en un objeto **Point**.

[17] Todos estos métodos son solo un extracto de los que realmente tiene la clase **Component**. Para mayor información se recomienda consultar la documentación de la API de JDK 1.3 en: http://java.sun.com/j2se/1.3/docs/api/java/awt/Component.html

Método	Descripción
void setFont(Font)	A la componente le da un estilo de letra. Para darle un font, uno puede utilizar el constructor de la clase **Font**: x.setFont(new Font("Arial", Font.ITALIC, 12));
Font getFont()	Obtiene el estilo que tiene el componente.

void setBackground(Color)
y también
void setForeground(Color)
	Le da color al fondo (Back) o al font (Fore) de la componente. Para utilizar el color, se usa la clase **Color**:
	
	x.setBackground(Color.black);
	
	o también se puede obtener gracias a la representación de colores en codificación RGB (cantidad de Rojo, Verde y Azúl):
	
	x.setForeground((new Color(100, 100, 100)).getColor());

Color getBackground() y también Colo getForegrount()	Obtiene el color que está definido para la componente en el fondo (Back) o en el font (Fore).
void repaint()	Redibuja toda la componente.

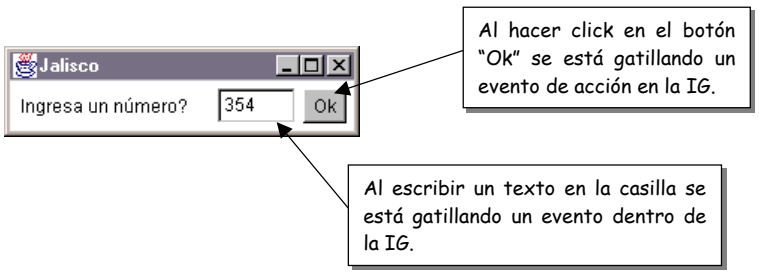
Conceptos

Evento

Acción que ocurre dentro en una componente y que es utilizada para gatillar acciones sobre la interfaz (al presionar un botón, al escribir un texto o al hacer click en un área específica).

El funcionamiento de las interfaces gráficas no es automático como lo hace cualquier programa, pues debe haber alguna interacción con el usuario en algún momento. Hasta ahora habíamos visto esa interacción como un ingreso por teclado de datos por parte del usuario. En las IG (Interfaces Gráficas) existen otras formas de ingreso de datos que corresponden a dispositivos comunes como son el teclado y el mouse.

Esos ingresos o esa interacción se hace a través de los eventos.



Listener

Elemento que le permite a un componente detectar cuándo ocurre un evento sobre él e indicar qué debe hacer en la interfaz.

Tal como dice la definición, los listeners son asignados a objetos o componentes de una IG y le permiten darse cuenta cuándo les están enviando un evento a ellos.

Al momento de decirle al componente: "escucha el evento", se le dice también que cuando ocurra realice ciertas acciones. Esas acciones pueden ser cambios en la interfaz, acciones sobre archivos, bases de datos, etc.

Sintaxis

En las interfaces gráficas para Java existen distintos tipos de eventos, que son capturados por los listener. Algunos de estos eventos son:

Eventos de Acción (ActionEvent)

Cualquier acción que es gatillada por una componente se le llama evento de acción. Estos eventos son representados por la clase ActionEvent y es ella la que se encarga de almacenar a qué componente corresponde el evento.

Para capturar estos eventos, es necesario utilizar un listener del tipo **ActionListener**:

```
...
Button b = new Button("Activa");
b.addActionListener(new <Listener de Acción Personalizado>);
...
```

de esta forma, le decimos a la IG que, al ocurrir un evento de acción sobre la componente (activar el botón en este caso) que capture ese evento y ejecute el "Listener de Acción Personalizado".

Este listener debe ser declarado dentro de la IG como una nueva clase (¡sí! una clase dentro de otra) que implemente un **ActionListener**. Por ejemplo:

```
...
public class MiInterfazGrafica {
        ...
        public MiInterfazGrafica() {
                ...
                t = new TextField(20);
                t.addActionListener(new MiActionListener);
                ...
        }
        ...
        class MiActionListener implements ActionListener  {
                ...
        }
        ...
}
```

Así, le estaríamos indicando a nuestra IG que el listener que queremos utilizar se llama **MiActionListener**. Este es solo el nombre y podrías haber utilizado el que te guste más.

Todas las clases que implementen un ActionListener DEBEN tener esta estructura fija:

```
class <nombre> implements ActionListener {
      public void actionPerformed(ActionEvent e) {
            // Aquí van las acciones que se ejecutan cuando ocurre
            // un ActionEvent
      }
}
```

Las acciones que se ejecutan pueden utilizar elementos de la IG libremente, ya que por estar dentro de la misma clase pueden acceder a todos los elementos de ella sin necesitar de un objeto.

Veamos el típico ejemplo del programa Jalisco con eventos:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

class Jalisco {
      // Acá esta nuestra IG físicamente referenciada
      private Frame programa;

      // Estos son los componentes que van a actuar cuando ocurre el
      // ActionEvent respectivo
      private Label respuesta;
      private TextField numero;
      private Button salir;

      public Jalisco() {
            // Creamos nuestra nueva ventana
            programa = new Frame("Jalisco");
            programa.setLayout(new GridLayout(3, 1));

            // Ponemos el texto antes del campo de datos
            programa.add(new Label("Ingresa un número? ",
                        Label.CENTER));

            // Ponemos el cuadro de ingreso del usuario
            numero = new TextField(4);
            numero.addActionListener(new UnEscuchador());
            programa.add(numero);

            // Ponemos el area en donde el computador  da
            // su respuesta
            respuesta = new Label("", Label.CENTER);
            programa.add(respuesta);

            // Mostramos la IG
            programa.pack();
            programa.show();
      }
...
```


Hasta aquí vamos bien, ya que la única diferencia con lo que ya habíamos visto es la línea en la cual le indicamos el ActionListener a la IG. En esta oportunidad, el ActionListener lo ponemos en el TextField (ingreso de datos) para que se active SOLO SI el usuario presiona ENTER dentro de esa casilla (simulando que ingresa número y presiona ENTER).

```
...
      class UnEscuchador implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                  int n = Integer.parseInt(numero.getText());
                  respuesta.setText((n+1) + " te gané!!!");
                  numero.setText("");
            }
      } // Fin del ActionListener
} // Fin de la IG
```

Este ejemplo produce esta ventana:

suponiendo la situación de que el usuario ingrese 27 como número de entrada. Cada vez que el usuario presione ENTER en la casilla de texto dejada para que ingrese su número, el computador tomará ese número (con el comando **numero.getText()**) y lo pondrá como valor de entrada en la casilla de respuestas sumado en 1, es decir:

```
// Toma el número de la casilla de texto (numero) y lo transforma
// a un entero dejándolo en la variable n
int n = Integer.parseInt(numero.getText());

// Escribe en el area de respuesta el número que ingreso el usuario
// (n) incrementado en 1 (n+1) y el texto "te gané!!!"
respuesta.setText((n+1) + " te gané!!!");

// Borra el contenido de la casilla de texto.
numero.setText("");
```

Así de simple funciona.

Nota: Para que un applet se cierre al presionar un botón, en el ActionListener definido para ese botón se debe usar **System.exit(0)**.

Eventos del Mouse (MouseEvent)

Los eventos del mouse son eventos que pueden ocurrir sobre una componente cuando con el mouse uno realiza algo: ingresar a la componente, presionar el botón, soltar el botón, salir de la componente, moverse, etc.

Para capturar los MouseEvents es necesario utilizar un escuchador del tipo **MouseListener** (caso obvio después de ver los ActionListeners) o uno del tipo **MouseMotionListener**.

```
...
Canvas c = new Canvas();
b.addMouseListener(new <Listener de Ratón Personalizado>);
...
```

A diferencia de los ActionListeners, los MouseListeners tienen una estructura más compleja, porque los eventos posibles por el mouse son más que uno solo. Es por eso que la estructura de los MousListeners cambia:

```
class <nombre> implements MouseListener {
	public void mouseClicked(MouseEvent e) {
		// Se ejecuta solo cuando se hace un click del mouse
	}

	public void mouseEntered(MouseEvent e) {
		// Se ejecuta solo cuando el mouse ingresa a una
		// componente (el puntero ingresa al área de la
		// componente)
	}

	public void mouseExited(MouseEvent e) {
		// Se ejecuta solo cuando el mouse abandona a una
		// componente (el puntero abandona del área de la
		// componente)
	}

	public void mousePressed(MouseEvent e) {
		// Se ejecuta solo cuando el botón del mouse es
		// presionado y se mantiene así (no es un click rápido)
	}

	public void mouseReleased(MouseEvent e) {
		// Se ejecuta solo cuando el botón del mouse es
		// soltado después de ser presionado (caso anterior)
	}
}
```

Como pueden ver la cantidad de cosas que hay que implementar es más grande. Pero ¿qué pasa si solo quiero hacer algo cuando el usuario hace CLICK en el mouse? Pues simplemente debes dejar los otros métodos en blanco y escribir solo en el método que corresponde al evento de CLICK del mouse:

```
class UnEventoClick implements MouseListener {
	public void mouseClicked(MouseEvent e) {
		// ACA SE ESCRIBE CODIGO
	}

	public void mouseEntered(MouseEvent e) { }
	public void mouseExited(MouseEvent e) { }
	public void mousePressed(MouseEvent e) { }
	public void mouseReleased(MouseEvent e) { }
}
```

y así funciona. Veamos un ejemplo sencillo. Juguemos al Gato entre 2 personas:

```
import java.applet.*;
```



```
import java.awt.*;
import java.awt.event.*;

public class Gato {
	// IG
	private Frame ventana;

	// Para despliegue de mensajes de sistema
	private Label mensaje;

	// El GATO
	private TextField[][] gato;

	// Quien juega
	private boolean jugador1;

	public Gato() {
		ventana = new Frame("Gato");
		ventana.setLayout(new BorderLayout());

		ventana.add("North",
			new Label("Juego del Gato", Label.CENTER));

		mensaje = new Label("Jugador 1", Label.CENTER);
		jugador1 = true;
		ventana.add("South", mensaje);

		// Creamos el arreglo para el gato y
		// el panel que lo contiene
		Panel pgato = new Panel();
		gato = new TextField[3][3];
		pgato.setLayout(new GridLayout(3, 3));
		for (int i=0; i<3; i++) {
			for (int j=0; j<3; j++) {
				gato[i][j] = new TextField(1);
					// A cada casilla le ponemos un
					// mouselistener
				gato[i][j].addMouseListener(
					new JuegaGato());
				pgato.add(gato[i][j]);
			}
		}
		ventana.add("Center", pgato);

		// Está listo el juego
		ventana.pack();
		ventana.show();
	}
...
```

Hasta aquí tenemos la interfaz definida para quedar de la siguiente forma:

Ahora manipulemos que, cuando el usuario haga click sobre los campos en blanco se ponga una "X" o una "O" dependiendo si es el jugador 1 o 2 respectivamente:

```
...
      class JuegaGato implements MouseListener {
            public void mouseClicked(MouseEvent e) {
                  // Buscamos casilla seleccionada
                  for (int i=0; i<3; i++) {
                        for (int j=0; j<3; j++) {
                              if (e.getSource() == gato[i][j]) {
                                 if (jugador1) {
                                     gato[i][j].setText("X");
                                     jugador1 = false;
                                     mensaje.setText("Jugador 2");
                                 }
                                 else {
                                     gato[i][j].setText("O");
                                     jugador1 = true;
                                     mensaje.setText("Jugador 1");
                                 }
                              }
                        }
                  }
            }

            public void mouseEntered(MouseEvent e) { }
            public void mouseExited(MouseEvent e) { }
            public void mousePressed(MouseEvent e) { }
            public void mouseReleased(MouseEvent e) { }
      }
}
```

Como podemos ver, solo implementamos el método **mouseClicked(MouseEvent e)** para solucionar el tema de marcar el gato. Así evitamos hacer los demás métodos.

Nota: Utilizamos algo nuevo para identificar qué componente se está haciendo click. Si pensamos un poco, son 9 casillas distintas que gatillan el mismo evento. Podríamos hacer 9 eventos distintos, pero eso es poco óptimo. Así que diferenciamos dentro de los listeners a cuál componente corresponde con el método **getSource()** del evento.

e.getSource() : Retorna una referencia a la componente a la cual se le accionó el evento.

Es por eso que podemos compararlo con la componente en forma directa como

```
...
if (e.getSource() == gato[i][j]) {
      ...
}
...
```

ya que le estamos diciendo implícitamente "*si el evento fue gatillado sobre la componente gato[i][j]…*". Este método sirve no solo para los MouseEvents, si no que para cualquier evento (ActionEvent, ItemEvent, etc.).

El otro tipo de listener para el mouse es el **MouseMotionListener** quien se preocupa de los movimientos libres que el mouse puede realizar:

```
class <nombre> implements MouseMotionListener {
      public void mouseDragged(MouseEvent e) {
            // Se ejecuta solo cuando se presiona el botón en una
            // componente y luego se arrastra.
      }

      public void mouseMoved(MouseEvent e) {
            // Se ejecuta solo cuando el mouse se muerve dentro
            // del área de una componente, cualquiera sea.
      }
}
```

Para estos últimos eventos es muy importante destacar que la variable **e**, además de identificar la componente seleccionada, posee otras características (métodos) que pueden ser de gran importancia:

Método	Descripción
int getClickCount()	Retorna el número de clicks que se realizaron.
int getX() y int getY()	Retorna el valor de la posición horizontal (X) y vertical (Y) donde ocurrió el evento, relativo a la componente asignada.

Todo esto resulta útil cuando estamos tratando de dibujar en un Canvas, sin embargo, estos métodos se pueden utilizar dentro del MouseListener sin problemas (si fuera necesario).

Eventos de Selección (ItemEvent)

Los eventos de selección son eventos que ocurren cuando en una componente de selección (Checkbox, List o Choice) cambia su estado, es decir, se selecciona un item o se activa o desactiva una opción.

Para capturar los ItemEvents es necesario utilizar un escuchador del tipo **ItemListener**.

```
...
List c = new List(10);
b.addItemListener(new <Listener de Selección Personalizado>);
...
```

Al igual que los MouseEvents, los listeners de este tipo poseen un método que DEBE ser implementado para poder capturar los ItemEvents:

```
class <nombre> implements ItemListener {
      public void itemStateChanged(ItemEvent e) {
            // Aquí van las acciones que se ejecutan cuando ocurre
            // un ItemEvent
      }
}
```

Más Eventos

Otros eventos que se pueden utilizar se detallan en la siguiente tabla:

Evento	Listener	Métodos a Implementar en el Listener
de Texto	TextListener	void textValueChanged(TextEvent e)
del Teclado	KeyListener	void keyPressed(KeyEvent e)
		void keyReleased(KeyEvent e)
		void keyTyped(KeyEvent e)
de Ventana	WindowListener	void windowActivated(WindowEvent e)
		void windowClosed(Event e)
		void windowClosing(Event e)
		void windowDeactivated(Event e)
		void windowDeiconified(Event e)
		void windowOpened(Event e)

Sin dejar de considerar que toda la información que se ha desplegado es simplemente referencial y no necesariamente está completa. Para mayor información se recomienda visitar la página de la API de JDK en la página de la empresa SUN Microsystems[18].

Canvas

Ahora veamos como se utiliza la componente Canvas en las interfaces gráficas.

En el Canvas, al igual que en otras componentes, pueden ocurrir eventos de lo más normales. De hecho, una forma de poner un Canvas en una IG es igual que cualquier otra componente, indicando eso si el tamaño de él:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Paint {
	private Frame programa;
	private Canvas area;

	public Paint() {
		programa = new Frame("Java Paint");
		programa.setLayout(new FlowLayout());

		area = new Canvas();
		area.setSize(800, 600);
		area.addMouseListener(new Pincel());

		programa.add(area);
		programa.pack();
		programa.show();
	}
...
}
```

[18] API del JDK: http://java.sun.com/j2se/1.3/docs/api/index.html

Como se ve en el ejemplo, lo que estamos haciendo es creando un Canvas de 800 x 600 pixeles de tamaño, solo en la IG con la intención de que al ocurrir un evento de mouse, el listener llamado **Pincel** lo capture.

Pero ¿cuál es la gran diferencia con los componentes que ya hemos visto? hasta aquí, ninguna, pues cambia cuando tenemos que dibujar sobre el Canvas algun elemento.

Graphics

Esta clase es utilizada por el canvas para dibujar. Graphics a diferencia de las componentes de las IG es una clase que controla el contenido de un Canvas y no tiene nada que ver con su layout o eventos que ocurran porque no es un espacio físico en pantalla.

Se definen los métodos de la clase Graphics:

Método	Descripción
setColor(Color)	Cambia el pincel al color de tipo **Color**. Los colores se definen como constantes en la clase Color: • Color.black = Negro • Color.white = Blanco • Color.blue = Azúl • Color.red = Rojo • Color.green = Verde • Color.yellow = Amarillo • etc Una vez que se setea el color, todas las figuras que se realicen quedan con ese color definidas. Si quieres definir más colores, busca la documentación de la clase Color en el sitio de la API del JDK.
drawLine(int, int, int, int)	Dibuja una línea entre el punto 1 (x, y) y el punto 2 (x, y).
drawOval(int, int, int, int)	Dibuja el contorno de una elipse en la posición centro (x, y) con un tamaño horizontal y vertical definido.
fillOval(int, int, int, int)	Dibuja una elipse rellena en la posición centro (x, y) con un tamaño horizontal y vertical definido.
drawRect(int, int, int, int)	Dibuja el contorno de un rectángulo con vértice superior izquierdo (x, y) y un tamaño horizontal y vertical definido.
fillRect(int, int, int, int)	Dibuja un rectángulo relleno con vértice superior izquierdo (x, y) y un tamaño horizontal y vertical definido.
setFont(Font)	Pone un estilo de letra especifico a los elementos de texto que se escriban en adelante.
drawString(String, int, int)	Escribe el texto en la posición (x, y) de la IG.

Método	Descripción
clearRect(int, int, int, int)	Limpia un área definida por el rectángulo asociado (ver drawRect).

Con estos métodos, podemos hacer muchas cosas. Veamos un ejemplo de utilización del canvas:

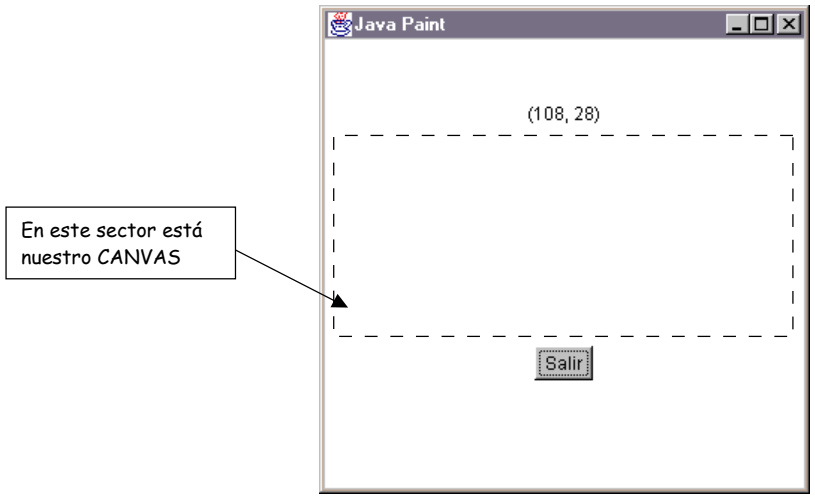
```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Paint {
	// Las componentes importates de la IG
	private Frame programa;
	private Canvas area;
	private Label pos;
	private Button salir;

	public Paint() {
		// Creamos primero la ventana de nuestro Paint
		programa = new Frame("Java Paint");
		programa.setLayout(new GridLayout(3, 1));

		// Le ponemos un LABEL para la posición del Mouse
		pos = new Label("", Label.CENTER);
		programa.add(pos);

		// Creamos el Canvas que nos permitirá dibujar en él
		area = new Canvas();
		area.setSize(800, 600);
		area.addMouseListener(new Pincel());
		area.addMouseMotionListener(new Movimiento());
		programa.add(area);

		// Ponemos un botón para terminar y cerrar la ventana
		salir = new Button("Salir");
		salir.addActionListener(new AccionBoton());
		Panel p = new Panel();
		p.setLayout(new FlowLayout());
		p.add(salir);
		programa.add(p);

		// Mostramos la IG creada
		programa.pack();
		programa.show();
	}
...
```

Como podemos ver en estas pocas líneas, lo que queremos hacer es un área de dibujo utilizando un Canvas (muy similar al ejemplo que habíamos visto anteriormente).

Es muy importante destacar varios puntos:

- El Canvas recibe acciones de MouseListener y MouseMotionListener en este ejemplo.
- Existe un botón que permitirá terminar la ejecución del programa.
- Hay un label que nos indicará la posición (en pixeles) de donde se encuentra el puntero.

Todo esto queda distribuído y nos muestra algo como lo siguiente:



Pero como sabemos que el Canvas no es facil de trabajar, prepararemos algunos métodos que nos permitirán hacer "algo" dentro de él. En este caso particular, nuestro Java Paint solo dibujará puntos (por ahora):

```
...
	// Acciones que se realizan dentro del Canvas
	private void dibujaPunto(int x, int y) {
		// Se obtiene primero el pincel del canvas
		Graphics g = area.getGraphics();

		// Seteamos el color azúl para dibujar el punto
		g.setColor(Color.blue);

		// Dibujamos un punto, es decir, un óvalo de radio 5
		// pixeles x 5 pixeles en el punto x, y indicado en
		// el parámetro.
		g.fillOval(x, y, 5, 5);
	}
...
```

Ahora bien, como podemos ver no tenemos idea donde se está dibujando cada punto. Ese si sería trabajo del listener que, cuando se presione el botón del mouse nos diga en qué posición (en pixeles) se hizo el click para dibujar el punto.

Entonces, debemos implementar el primero de los listener, que es el **MouseListener**, el cual nos permitirá hacer que si se presiona el botón del mouse, dibujar un punto en la posición aquella, utilizando el método anteriormente definido:

```
...
	class Pincel implements MouseListener {
		public void mouseClicked(MouseEvent e) {
```

```
                dibujaPunto(e.getX(), e.getY());
            }

            public void mouseEntered(MouseEvent e) {
            }

            public void mouseExited(MouseEvent e) {
            }

            public void mousePressed(MouseEvent e) {
                dibujaPunto(e.getX(), e.getY());
            }

            public void mouseReleased(MouseEvent e) {
            }
        }
...
```

Como podemos ver suponemos en ambos eventos en los cuales se presiona el botón (click y pressed) para que dibuje el punto en x, y dado por e.getX() y e.getY.

Ahora implementemos el que nos indicará en qué posición se encuentra el mouse para mostrarla en el label que definimos sobre el área de dibujo:

```
...
        class Movimiento implements MouseMotionListener {
            public void mouseMoved(MouseEvent e) {
                pos.setText("(" + e.getX() + ", " +
                            e.getY() + ")");
            }

            public void mouseDragged(MouseEvent e) {
            }
        }
...
```

En este caso, mouseMoved nos indica si el mouse se mueve o no. Pero en este caso no tenemos que dibujar, pero si indicar en qué posición está el puntero del mouse. Es por eso que ponemos esa posición como un par ordenado en el label definido sobre el área llamado **pos**.

Por último, completamos con el ActionListener para el botón salir, el cual cierra la IG:

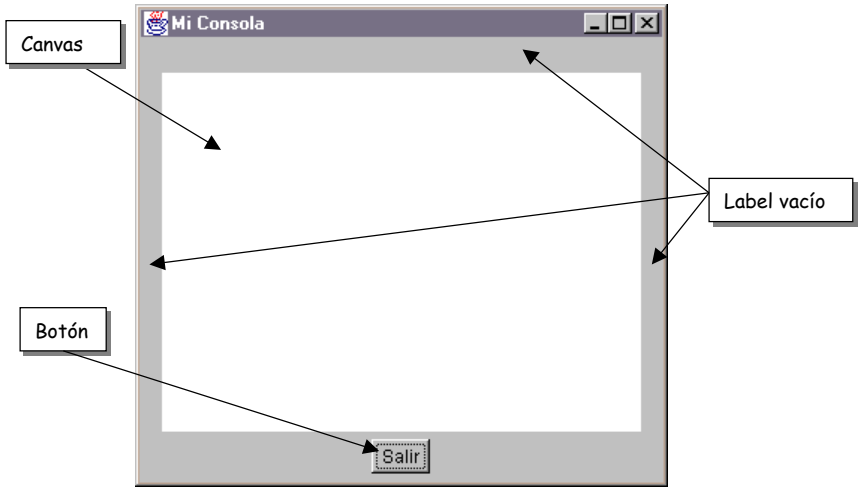
```
...
        class AccionBoton implements ActionListener {
            public vois actionPerformed(ActionEvent e) {
                // La IG se cierra si el programa termina
                System.exit(0);
            }
        }
}
```

Problemas

(a) Escribir la clase **Consola** que es una interfaz gráfica similar a la clase Console que hemos utilizado hasta ahora. Solo programe los siguientes métodos:

Método	Descripción
Consola()	Crea una consola con el título "Consola" y de tamaño 320 x 240.
Consola(String)	Crea una consola con el título indicado y de tamaño 320 x 240.
void imprimir(String)	Imprime dentro de la consola el texto indicado en la posición actual sin salto de línea.
void imprimirsl(String)	Imprime dentro de la consola el texto indicado en la posición actual saltando una línea después de hacerlo.
void limpiar()	Limpia la consola.

La IG que debe programar debe ser similar a esta:



Nota: Los label vacío son solo para darle un toque de margen al cuento. Ojo con el Layout que está casi directo con el gráfico.

Solución

Partamos primero por la parte de dibujo de la IG:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Consola {
      // Primero van las componentes necesarias para controlar
      // esta nueva consola.
      private Frame ventana;
      private Canvas area;
      private Button salir;

      // También necesitaremos un punto para indicar en donde nos
      // encontraremos escribiendo dentro del Canvas
      private int linea, caracter;
```

```
	public Consola(String titulo) {
		// Creamos la ventana con BorderLayout, con el título
		// pasado por parámetro y color de fondo gris claro.
		ventana = new Frame(titulo);
		ventana.setLayout(new BorderLayout());
		ventana.setBackground(Color.lightGray);

		// Creamos el Canvas de tamaño definido 320x240, con
		// color de fondo blanco (diferenciarlo del frame) y
		// puesto al centro del layout del frame.
		area = new Canvas();
		area.setSize(320, 240);
		area.setBackground(Color.white);
		ventana.add("Center", area);

		// Creamos el botón salir y lo ponemos al sur del
		// layout del frame. Ojo que para que quede pequeñito
		// usamos un panel con FlowLayout.
		salir = new Button("Salir");
		Panel p = new Panel();
		p.setLayout(new FlowLayout());
		p.add(salir);
		ventana.add("South", p);

		// Le damos el listener para que cierre la ventana
		// al click en el botón "Salir".
		salir.addActionListener(new Finalizar());

		// Ponemos los labels vacío de margen al rededor del
		// Canvas, es decir, al norte, este y oeste.
		ventana.add("North", new Label());
		ventana.add("East", new Label());
		ventana.add("West", new Label());

		// Mostramos la ventana.
		ventana.pack();
		ventana.show();

		// Inicializamos como inicio de escritura.
		linea = 1;
		caracter = 1;
	}

	public Consola() {
		// Llama al otro constructor, pero con el título fijo.
		this("Consola");
	}

	// Ahora le damos el listener del botón.
	class Finalizar implements ActionListener {
		public void actionPerformed(ActionEvent e) {
			System.exit(0);
		}
	}
```

Una vez que la parte gráfica está lista, podemos poner los métodos que trabajarán sobre el canvas y escribirán o borrarán algo:

```
	// Método que limpia la pantalla
	public void limpiar() {
		// Obtenemos el pincel
```



```
		Graphics g = area.getGraphics();

		// Limpia la pantalla completa.
		g.clearRect(0, 0, 320, 240);

		// La posición vuelve al inicio.
		linea = 1;
		caracter = 1;
	}

	// Métodos que imprimen en pantalla
	public void imprimir(String s) {
		// Obtenemos el pincel
		Graphics g = area.getGraphics();

		// Escribimos el texto en el canvas
		g.drawString(s, (caracter - 1)*7, linea * 12);

		// Dejamos el cursor al final
		caracter += s.length() + 1;
	}

	public void imprimirsl(String s) {
		// Usamos el método anterior
		imprimir(s);

		// Movemos el lapiz a la línea siguiente
		linea++;
		caracter = 1;
	}
}
```

Como los caracteres son de un tamaño de 12 pixeles de alto y 7 pixeles de ancho, podemos simular que escribimos el string en la posición (caracter-1)*7, linea*12 (considerando que para escribir necesitamos indicar la base del string).

Ahora si probamos algo, podremos saber como queda:

```
public class MiPrograma {
	static public void main(String[] args) {
		Consola c = new Consola("Mi Consola");
		c.imprimirsl("1234567890");
		c.imprimir("123");
		c.imprimir("123");
	}
}
```

Capítulo XVI: Interfaces Gráficas SWING

(en construcción)

Capítulo XVII: Excepciones y Control de Errores

Motivación

Lo más común al hacer programas en Java son los errores que aparecen en la "pantalla de la muerte" o salida de errores.

Cómo manejarlos, entenderlo y lograr prever los posibles problemas en tiempo de ejecución es un trabajo de lo que se llaman **Excepciones**. Lo más importante es que Java provee una forma para que el programador controle fácilmente estos errores sin conocer las condiciones en las que ocurren previamente, bajo solo suposiciones del estilo "el archivo puede tener problemas de lectura/escritura, no existir o simplemente estar malo".

Conceptos

Existen 2 clases de errores:

Errores de Compilación

> *Los errores de compilación son aquellos errores que son detectados por el compilador (javac) en el momento en que se genera la clase ejecutable (archivo .class) deseada.*

Estos errores comúnmente ocurren cuando existe un error de sintaxis o falta alguna clase que es llamada en los archivos que crea el programador.

Lo interesante de estos errores es que se pueden detectar rápidamente, pues el compilador indica exactamente qué pasó y donde ocurrió, lo que los hace muy fácil de controlar y corregir. Sin embargo, lo interesante no es esta clase de errores sino la que viene a continuación.

Errores de Ejecución (Runtime)

> *Los errores que ocurren en tiempo de ejecución o runtime son problemas que, al momento de ejecutar las clases ya compiladas, suelen ocurrir por ingreso de datos, manipulación de dispositivos de entrada/salida, condiciones de borde, conversión de tipos de datos, etc.*

Existe una infinidad de razones de por qué el programa se cae en tiempo de ejecución. Veamos un pequeño ejemplo:

```
public class UnArreglo {
      static public void main (String[] args) {
            int[] ar = new int[10];
            ar[10] = 25;
      }
}
```

Este sencillísimo programa lo que hace es asignar fuera del rango del arreglo un valor. Bueno, si compilan este programita, se darán cuenta que no hay error detectable. Pero al momento de ejecutar la clase **UnArreglo**, lanzará el siguiente error:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at UnArreglo.main(UnArreglo.java:4)
```

Este texto indica el error o **Excepción** que ha ocurrido al ejecutar la clase. Analicemos un poco la excepción para saber cómo solucionarla:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
```

El texto destacado **ArrayIndexOutBoundsException** indica qué ha ocurrido. En este caso (y solo bastaría utilizar un diccionario de Inglés-Español) podemos darnos cuenta que nos dice:

Excepción de Índice Fuera del Rango del Arreglo.

Tenemos identificado cuál es el error (que obviamente era el que predijimos al escribir el programa). Pero ¿dónde y por qué ocurrió?. Bueno, continuemos el análisis.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:  10
```

Este pequeño numerito casualmente coincide con el valor del rango que queríamos sobrepasar. Bueno, no es casualidad, porque en este caso ese valor indica la posición del arreglo que hemos querido asignar o referenciar y que está fuera del rango. El rango en este caso terminaba en la posición 9 y queríamos acceder a la 10. Por eso obtuvimos el error. Siguiendo:

```
        at UnArreglo.main(UnArreglo.java:4)
```

Esta línea nos indica dónde ocurrió. En general esta línea no es una, sino varias, dependiendo cuántos métodos estemos llamando. Pero lo que indica es que en el método **main** de **UnArreglo** (esto lo dice en la parte **UnArreglo.main**) fue la excepción. Más específicamente, en la línea **4** del archivo **UnArreglo.java**.

Ahora que sabemos todo, sinteticemos nuestro análisis:

1. Tenemos una asignación fuera del rango de un arreglo.
2. Tratamos de poner o referenciar la posición 10 del arreglo.
3. El error está en el método main de la clase UnArreglo.
4. La línea es la 4 del archivo UnArreglo.java.

Con esta información es fácil corregir el problema.

Analicemos otro problema algo más complejo:

```
public class Programa {
      static public void main (String[] args) {
            double[] ar = new double[10];
            ar[0] = new Double("ALFA1").doubleValue();
      }
}
```


El stack de excepción quedaría:

```
Exception in thread "main" java.lang.NumberFormatException: ALFA1
        at java.lang.FloatingDecimal.readJavaFormatString(FloatingDecimal.
java:1180)
        at java.lang.Double.valueOf(Double.java:172)
        at java.lang.Double.<init>(Double.java:244)
        at Programa.main(Programa.java:4)
```

Bueno, ahora es un poquito más grande que en el ejemplo anterior, pero es analizable.

Paso 1: Tomemos la línea que indica el tipo de excepción:

```
Exception in thread "main" java.lang.NumberFormatException: ALFA1
```

Esto nos indica que la excepción es **NumberFormatException** o traducida **Excepción de Formato Numérico** (o de Número). ¿Por qué?. El valor que viene a continuación "ALFA1" es el problema, puesto que (como pueden ver) no es numérico. Ese es el problema.

Paso 2: Veamos el resto de la excepción para ver si logramos obtener donde ocurrió:

```
        at java.lang.FloatingDecimal.readJavaFormatString(FloatingDecimal.
java:1180)
        at java.lang.Double.valueOf(Double.java:172)
        at java.lang.Double.<init>(Double.java:244)
        at Programa.main(Programa.java:4)
```

Lo que anteriormente habíamos dicho se cumple, pues ya no es una línea, sino que son 4 líneas. Pero es sencillo descubrir donde está el error, pues buscamos aquellos programas que hemos hecho para buscar el error, en este caso, **Programa.java** es nuestro (los demás ni siquiera sabíamos que existían). Esta última línea nos dice nuevamente que el error está en el método **main** de la clase **Programa,** en la línea **4** del archivo **Programa.java**.

Sencillo ahora que sabemos cómo hacerlo.

Sintaxis

Identificar Excepciones en Java

Las excepciones en general pueden ser de 2 tipos:

- **Errores de Programación**: Aquellos errores en donde el programador puede evitarlo porque es un error al codificar el programa
- **Errores de Datos**: Aquellos errores que el programador puede evitar, o simplemente no puede hacerlo, ya que son problemas en la interacción programa-usuario.

Es claro que los primeros son muy graves, pues al utilizarlos no deberían existir. Sin embargo estos errores no son los más comunes.

Java provee una forma de atrapar los errores de datos y poder controlar así los programas evitando que el usuario utilice mal el programa o simplemente no pasen imprevistos como falta el archivo, no tiene permisos para escribir o que se haya cerrado el puerto.

¿Cómo se hace?

Ya hemos utilizado una forma de control de esto en archivos. Veamos el siguiente ejemplo:

```
public class Archivo {
      static public void main (String[] args)  throws Exception {
            BufferedReader br = new BufferedReader(
                  new FileReader ("archivo.txt"));

            // ... continúa el programa

            br.close();
      }
}
```

Este sencillo código nos muestra como utilizamos un archivo de texto (visto en clases anteriores). A diferencia de los programas tradicionales, este método posee una sentencia adicional en su firma que dice **throws Exception**, esto quiere decir, que maneje (handle) las excepciones que ocurran. Si no pusiéramos este handler, el compilador nos da el error:

```
Archivo.java:6:  Exception  java.io.FileNotFoundException  must  be
caught, or it must be declared in the throws clause of this method.
                        new FileReader ("archivo.txt"));
                        ^
Archivo.java:10: Exception java.io.IOException must be caught, or it
must be declared in the throws clause of this method.
                br.close();
                        ^
2 errors
```

De esa forma (y solo en este caso) el compilador nos indica cuáles son las excepciones que debe manejar el método. También, es por eso que podríamos poner:

```
public class Archivo {
      static public void main (String[] args)
                  throws IOException, FileNotFoundException  {
            BufferedReader br = new BufferedReader(
                  new FileReader ("archivo.txt"));

            // ... continúa el programa

            br.close();
      }
}
```

y también está correcto. Pero solo en el caso especial de los archivos se obliga poner un **throws** en la firma del método.

Veamos otro ejemplo, en el cuál no es necesaria una cláusula **throws** pero si se pueden controlar excepciones de otro tipo:

```
class Datos {
      private BufferedReader lector;
      public Datos () {
            lector = new BufferedReader(
                  new InputStreamReader(System.in));
      }

      public int leeEntero() throws NumberFormatException {
            return Integer.parseInt(lector.readLine());
      }
}

public class Lector {
      static public void main (String[] args) throws Exception {
            Datos l = new Datos();

            System.out.print("Ingrese un Entero?");
            int e = l.leeEntero();

            System.out.print("Ingrese un Real?");
            int r = l.leeEntero();
      }
}
```

En este caso (y como se puede ver) se piden 3 valores distintos, pero se leen los 3 como enteros. En este caso y si el usuario ingresa lo que le piden, enviará un **NumberFormatException** al ingresar el real.

Aquí ocurre algo muy interesante, pues en el método **leeEntero()**, se está indicando que captura la excepción **NumberFormatException**, pero en el método padre (llamador) **main(String[] args)** se indica que es **Exception** la que debe manejar. ¿Por qué?. En realidad la razón fue solamente por comodidad, pues **Exception** es una superclase de todas las excepciones[19]:

- **class java.lang.Exception**
  - class java.awt.AWTException
  - class java.security.acl.AclNotFoundException
  - class java.rmi.AlreadyBoundException
  - class java.lang.ClassNotFoundException
  - class java.lang.CloneNotSupportedException
    - class java.rmi.server.ServerCloneException
  - class java.util.zip.DataFormatException
  - class java.security.DigestException
  - class java.io.IOException
    - class java.io.CharConversionException
    - class java.io.EOFException
    - class java.io.FileNotFoundException

[19] Se muestra una lista completa de excepciones. Aquellas que se encuentran marcadas son las que comunmente les podría ocurrir. Las demás son para que conozcan todas las excepciones que existen

class java.io.InterruptedIOException
class java.net.MalformedURLException
class java.io.ObjectStreamException
class java.io.InvalidClassException
class java.io.InvalidObjectException
class java.io.NotActiveException
class java.io.NotSerializableException
class java.io.OptionalDataException
class java.io.StreamCorruptedException
class java.io.WriteAbortedException
class java.net.ProtocolException
class java.rmi.RemoteException
class java.rmi.AccessException
class java.rmi.ConnectException
class java.rmi.ConnectIOException
class java.rmi.server.ExportException
class java.rmi.server.SocketSecurityException
class java.rmi.MarshalException
class java.rmi.NoSuchObjectException
class java.rmi.ServerError
class java.rmi.ServerException
class java.rmi.ServerRuntimeException
class java.rmi.server.SkeletonMismatchException
class java.rmi.server.SkeletonNotFoundException
class java.rmi.StubNotFoundException
class java.rmi.UnexpectedException
class java.rmi.UnknownHostException
class java.rmi.UnmarshalException
class java.net.SocketException
class java.net.BindException
class java.net.ConnectException
class java.net.NoRouteToHostException
class java.io.SyncFailedException
class java.io.UTFDataFormatException
class java.net.UnknownHostException
class java.net.UnknownServiceException
class java.io.UnsupportedEncodingException
class java.util.zip.ZipException
class java.lang.IllegalAccessException
class java.lang.InstantiationException
class java.lang.InterruptedException
class java.beans.IntrospectionException
class java.lang.reflect.InvocationTargetException
class java.security.KeyException
class java.security.InvalidKeyException
class java.security.KeyManagementException

class java.security.acl.LastOwnerException
class java.security.NoSuchAlgorithmException
class java.lang.NoSuchFieldException
class java.lang.NoSuchMethodException
class java.security.NoSuchProviderException
class java.rmi.NotBoundException
class java.security.acl.NotOwnerException
class java.text.ParseException
class java.beans.PropertyVetoException
class java.lang.RuntimeException
class java.lang.ArithmeticException
class java.lang.ArrayStoreException
class java.lang.ClassCastException
class java.util.EmptyStackException
class java.lang.IllegalArgumentException
class java.lang.IllegalThreadStateException
class java.security.InvalidParameterException
class java.lang.NumberFormatException
class java.lang.IllegalMonitorStateException
class java.lang.IllegalStateException
class java.awt.IllegalComponentStateException
class java.lang.IndexOutOfBoundsException
class java.lang.ArrayIndexOutOfBoundsException
class java.lang.StringIndexOutOfBoundsException
class java.util.MissingResourceException
class java.lang.NegativeArraySizeException
class java.util.NoSuchElementException
class java.lang.NullPointerException
class java.security.ProviderException
class java.lang.SecurityException
class java.rmi.RMISecurityException
class java.sql.SQLException
class java.sql.SQLWarning
class java.sql.DataTruncation
class java.rmi.server.ServerNotActiveException
class java.security.SignatureException
class java.util.TooManyListenersException
class java.awt.datatransfer.UnsupportedFlavorException

Entonces, uno puede atrapar todas las excepciones utilizando la clase **Exception**. El otro punto importante es que, al capturar una excepción dentro de un método, todos los llamadores deben traspasar el control de ella hasta el método main (o al principal). Por eso es requerido que el main tuviera el **throws** en su firma. Sin embargo, no es estrictamente esa sentencia la que se debe usar (veremos otra).

Atrapar y Controlar Excepciones

Lo siguiente que veremos es la forma en que se puede atrapar una excepción y evitar que el programa se caiga por ello (lo que siempre hemos deseado).

> *Una sección critica es línea o trozo de código que pueden "caerse" por causa de una excepción.*

Una sección crítica se delimita con la sentencia **try... catch**. Su sintaxis es:

```
try {
      // Código que se quiere evitar una excepción
}
catch (Exception1 <var1>) {
      // Código que reemplaza la ejecución cuando ocurre Exception1
}
catch (Exception2 <var2>) {
      // Código que reemplaza la ejecución cuando ocurre Exception2
}
...
catch (ExceptionN <varN>) {
      // Código que reemplaza la ejecución cuando ocurre ExceptionN
}
```

Cuando existe una sección crítica, los catchs indican las excepciones que pueden ocurrir en cualquier nivel de éste. Las *Exception1 … ExceptionN* son los nombres de las excepciones que ocurren. Un ejemplo es:

```
public class Archivo {
      static public void main (String[] args) {
            try {
                  BufferedReader br = new BufferedReader(
                        new FileReader ("archivo.txt"));

                  // ... continúa el programa

                  br.close();
            }
            catch(Exception e) {
                  System.out.println("Ha ocurrido un error");
            }
      }
}
```

Esta es la versión más simple. Si ejecutan este código, el resultado si **archivo.txt** no existe es "Ha ocurrido un error", y no el molesto stack de excepción indicando qué ocurrió. Pero se puede dividir también en distintas excepciones:

```
public class Archivo {
      static public void main (String[] args) {
            try {
                  BufferedReader br = new BufferedReader(
                        new FileReader ("archivo.txt"));

                  // ... continúa el programa
```



```
                  br.close();
            }
            catch(FileNotFoundException e) {
                  System.out.println("No existe el archivo");
            }
            catch(IOException e) {
                  System.out.println("Error al leer el archivo");
            }
      }
}
```

En este segundo caso, el "handler" cambia, pues por razones de distintos errores pueden ocurrir las 2 excepciones, y dependiendo del tipo de error que ocurra, se despliega el mensaje adecuado.

Veamos otro ejemplo:

```
class Datos {
      private BufferedReader lector;
      public Datos () {
            try {
                  lector = new BufferedReader(
                        new InputStreamReader(System.in));
            }
            catch (Exception e) {
                  System.out.println ("Imposible abrir entrada");
                  System.exit(0);
            }
      }

      public int leeEntero() throws NumberFormatException {
            return Integer.parseInt(lector.readLine());
      }
}

public class Lector {
      static public void main (String[] args) {
            Datos l = new Datos();
            boolean entero = false;

            while (!entero) {
                  System.out.print("Ingrese un Entero?");
                  try {
                        int e = l.leeEntero();
                        entero = true;
                  }
                  catch (NumberFormatException e) {
                        System.out.println ("No ingresó entero");
                  }
            }

            System.out.println("Ahora si fue un entero");
      }
}
```

Este nuevo código permitiría que el usuario nunca ingresara un valor distinto a un entero (es decir letras, reales, otras cosas) sin que el programa se caiga, por supuesto.

Ahora, ¿para qué sirve la variable **e** que posee el *catch*?.

La variable que va definida en el catch a un costado del tipo de excepción es una referencia a un objeto del tipo de esa excepción. Sirve para obtener mayor información de ella:

- **String getMessage()**: Obtiene un corto mensaje descriptivo del error que ocurrió.
- **void printStackTrace()**: Imprime en la salida de error (estándar) el stack de excepción de la que ocurrió.

Por ejemplo, si miramos el main anterior (último ejemplo):

```
static public void main (String[] args) {
      Datos l = new Datos();
      boolean entero = false;

      while (!entero) {
            System.out.print("Ingrese un Entero?");
            try {
                  int e = l.leeEntero();
                  entero = true;
            }
            catch (Exception e) {
                  System.out.print ("Error: ");
                  System.out.println (e.getMessage());
                  e.printStackTrace();
            }
      }
      System.out.println("Ahora si fue un entero");
}
```

En este caso, si ocurre el error, se puede saber qué ocurrió. Un ejemplo de ejecución:

```
Ingrese un Entero?A
Error: A
java.lang.NumberFormatException: A
        at java.lang.Integer.parseInt(Integer.java:409)
        at java.lang.Integer.<init>(Integer.java:544)
        at Datos.leeEntero(Lector.java:11)
        at Lector.main(Lector.java, Compiled Code)

Ingrese un Entero?8.5
Error: 8.5
java.lang.NumberFormatException: 8.5
        at java.lang.Integer.parseInt(Integer.java:418)
        at java.lang.Integer.<init>(Integer.java:544)
        at Datos.leeEntero(Lector.java:11)
        at Lector.main(Lector.java, Compiled Code)

Ingrese un Entero?6
Ahora si fue un entero
```

Si no pusiéramos el método **printStackTrace()** el programa queda:

```
Ingrese un Entero?A
Error: A

Ingrese un Entero?8.5
```



```
Error: 8.5

Ingrese un Entero?6
Ahora si fue un entero
```

Algo mucho más limpio y ordenado.

Lanzar Excepciones

Java provee además la posibilidad de lanzar excepciones propias. Esto se hace a través de la sentencia **throw** (si, parecido a un **throws** pero sin la *s* final).

Por ejemplo:

```
class Datos {
      private BufferedReader lector;
      public Datos () {
            try {
                  lector = new BufferedReader(
                        new InputStreamReader(System.in));
            }
            catch (Exception e) {
                  System.out.println ("Imposible abrir entrada");
                  System.exit(0);
            }
      }

      public int leeEntero()  throws NumberFormatException  {
            String linea = "";
            try {
                  linea = lector.readLine();
            }
            catch (IOException e) {
            }
            if (linea.length() <= 0)
                  throw new NumberFormatException();
            return Integer.parseInt(linea);
      }
}

public class Lector {
      static public void main (String[] args) {
            Datos l = new Datos();
            boolean entero = false;

            while (!entero) {
                  System.out.print("Ingrese un Entero?");
                  try {
                        int e = l.leeEntero();
                        entero = true;
                  }
                  catch (NumberFormatException e) {
                        System.out.println ("No ingresó entero");
                  }
            }
            System.out.println("Ahora si fue un entero");
      }
}
```

Este programa obliga al sistema a enviar una excepción de tipo **NumberFormatException** cuando es ingresado por el teclado un string vacío. Nótese que es muy importante poner la sentencia **throws** para que la excepción lanzada sea capturada por el método llamador.

Es muy interesante este punto de vista, ya que, mezclado con la sentencia **try**... **catch** puede ser una herramienta útil al momento de controlar errores que antes no podíamos. Veamos otro ejemplo:

```
public class Selector {
	static public BufferedReader b = new BufferedReader (
		new InputStreamReader(System.in));

	static public String leeOpcion()  throws Exception {
		String op;
		try {
			op = b.readLine();
		}
		catch (IOException e) {
			op = "";
		}

		if (op.length() <= 0)
			throw new Exception("Debe ingresar una opción");
		if (!op.equals("A") &&
		    !op.equals("B") &&
		    !op.equals("C"))
			throw new Exception("Las opciones son A, B, C");

		return op;
	}

	static public void main (String[] args) {
		while (true) {
			System.out.print("Ingrese A, B o C?");
			String op = "";
			try {
				op = leeOpcion();
				break;
			}
			catch (Exception e) {
				System.out.println(e.getMessage());
			}
		}
	}
}
```

La salida de este programa sería:

```
Ingrese A, B o C?
Debe ingresar una opcion
Ingrese A, B o C? 34
Las opciones son A, B, C
Ingrese A, B o C? sdf
Las opciones son A, B, C
Ingrese A, B o C? A
```

Como podemos notar, ahora solo permitimos lanzar excepciones de tipo **Exception** pero con mensaje personalizado.

Crear Excepciones

El problema que hay con utilizar **Exception** directamente, es que a veces ocurren otras excepciones y se pierde saber qué ocurrió. Es por eso que podemos crear propias excepciones que se ajusten al nivel del programa.

Veamos el mismo ejemplo, pero algo modificado:

```
public class Selector {
	static public BufferedReader b = new BufferedReader (
		new InputStreamReader(System.in));

	static public String leeOpcion()  throws OptionException {
		String op;
		try {
			op = b.readLine();
		}
		catch (IOException e) {
			op = "";
		}

		if (op.length() <= 0)
			throw new OptionException();
		if (!op.equals("A") &&
		    !op.equals("B") &&
		    !op.equals("C"))
			throw new OptionException();

		return op;
	}

	static public void main (String[] args) {
		while (true) {
			System.out.print("Ingrese A, B o C?");
			String op = "";
			try {
				op = leeOpcion();
				break;
			}
			catch (OptionException e) {
				System.out.println(e.getMessage());
			}
		}
	}
}
```

Cambiamos la excepción por una llamada **OptionException**. ¿De dónde salió?. Pues la idea es que implementemos esa clase para que se pueda utilizar como excepción igual que las otras:

```
public class OptionException extends Exception  {
	public OptionException {
		super("Opción inválida. Las opciones son A, B o C");
	}
}
```

Ahora, el programa impide que se ingrese algo distinto de A, B o C lanzando una excepción **OptionException** y todo queda solucionado.

Problemas

Se quiere crear un visor de archivos de texto. Para ello se le pide:

(a) Construya una clase que manipule los archivos de texto encapsulando la clase **BufferedReader** y controlando las **IOException** que puedan ocurrir (No existe el archivo, Fin de Archivo inesperado, etc.)

Solución

Versión sin crear una Excepción:

```
public class Visor {
	private String nombre;

	public Visor (String nombre) {
		this.nombre = nombre;
	}

	public String leeArchivo () throws IOException {
		BufferedReader lector;
		lector = new BufferedReader (
			new FileReader(this.nombre));
		String texto = "";
		String linea = "";
		while ( (linea = lector.readLine()) != null)
			texto = texto + "\n" + linea;
		lector.close();
		return texto;
	}
}
```

Versión usando **Exception**:

```
public class Visor {
	private String nombre;

	public Visor (String nombre) {
		this.nombre = nombre;
	}

	public String leeArchivo () throws Exception {
		BufferedReader lector;
		try {
			lector = new BufferedReader (
				new FileReader(this.nombre));
		}
		catch (FileNotFoundException e) {
			throw Exception("El archivo no existe");
		}
		String texto = "";
		try {
			while (true)
				texto = texto + "\n" + lector.readLine();
		}
		catch (EOFException e) {
		}
		lector.close();
		return texto;
```



```
	}
}
```

Nótese que en esta segunda versión ya no se usa el NULL como fin de archivo.

(b) Escriba un programa (main) que simule el siguiente diálogo:

```
Bienvenido al Visor 1.0

Ingrese el Nombre del Archivo: tarea.java
No existe el Archivo

Ingrese el Nombre del Archivo: tarea4.java
[INICIO DEL ARCHIVO]
...    // Aquí aparece el archivo de texto línea a línea
[FIN DEL ARCHIVO]
Posee 57 líneas y 453 caracteres

Ingrese el Nombre del Archivo:

Se utilizó el programa 2 veces
1 archivos leídos
1 archivos no existían
```

Solución

Utilizando la versión sin crear una Excepción:

```
static public void main (String[] args) throws IOException {
	BufferedReader in = new BufferedReader (
		new InputStreamReader (System.in));
	System.out.println("Bienvenido al Visor 1.0");

	// Variables que se leen al final
	int siL = 0;
	int noL = 0;

	// Ciclo de Lectura
	while (true) {
		System.out.print("Ingrese el Nombre del Archivo: ");
		String nombre = in.readLine();
		if (nombre.length() <= 0) break;

		// Leemos el arhivo y controlamos la excepci ón
		Visor v = new Visor(nombre);
		try {
			String texto = v.leeArchivo();

			// Si ocurrió una excepción, no continúa
			System.out.println("[INICIO DEL ARCHIVO]");
			System.out.println(texto);
			System.out.println("[FIN DEL ARCHIVO]");

			// Contamos las líneas y los caracteres
			int ls = contarLineas(texto);
			int cs = texto.length();

			System.out.println("Posee " + ls +
				" línea y " + cs + " caracteres");
			siL++;
```

```
            }
            catch (FileNotFoundException e) {
                  System.out.println ("No existe el archivo");
                  noL++;
            }
      }

      // Fin del programa
      System.out.println("Se utilizó el programa " +
            (siL + noL) + " veces");
      System.out.println(siL + " archivos leídos");
      System.out.println(noL + " archivos no existían");
}
```

Utilizando la versión con **Exception**:

```
static public void main (String[] args) throws IOException {
      BufferedReader in = new BufferedReader (
            new InputStreamReader (System.in));
      System.out.println("Bienvenido al Visor 1.0");

      // Variables que se leen al final
      int siL = 0;
      int noL = 0;

      // Ciclo de Lectura
      while (true) {
            System.out.print("Ingrese el Nombre del Archivo: ");
            String nombre = in.readLine();
            if (nombre.length() <= 0) break;

            // Leemos el arhivo y controlamos la excepci ón
            Visor v = new Visor(nombre);
            try {
                  String texto = v.leeArchivo();

                  // Si ocurrió una excepción, no continúa
                  System.out.println("[INICIO DEL ARCHIVO]");
                  System.out.println(texto);
                  System.out.println("[FIN DEL ARCHIVO]");

                  // Contamos las líneas y los caracteres
                  int ls = contarLineas(texto);
                  int cs = texto.length();

                  System.out.println("Posee " + ls +
                        " línea y " + cs + " caracteres");
                  siL++;
            }
            catch (Exception e) {
                  System.out.println (e.getMe ssage());
                  noL++;
            }
      }

      // Fin del programa
      System.out.println("Se utilizó el programa " +
            (siL + noL) + " veces");
      System.out.println(siL + " archivos leídos");
      System.out.println(noL + " archivos no existían");
}
```


Y ahora el que cuenta líneas

```
static public int contarLineas (String texto) {
      int total = 1;
      for (int i=0;
           texto.indexOf(",", i) > 0;
           i = texto.indexOf(",", i)  + 1)
            total++;
}
```

(c) **Propuesto**. Construya un applet que utilice el Visor construído en (a) para que funcione gráficamente.

Capítulo XVIII: Tipos y Estructuras de Datos

Motivación

Con todo lo que ya hemos visto hasta ahora podemos resolver un sin número de problemas computaciones aplicando los conceptos, esquemas, patrones de programación e instrucciones que utiliza Java.

Veamos un problema:

Los auxiliares de CC10A corrigen las preguntas de un grupo de mechones de todas las secciones por separado. Se les ha pedido que cada vez que corrijan construyan un archivo llamado "**pX-auxNN**" en donde X indica el número de la pregunta y NN el número del auxiliar (numerado entre 01 y 18). Lo más simpático es la correspondencia de corrección, es decir siempre ocurre que:

- Auxiliar 1 corrige Pregunta 1
- Auxiliar 2 corrige Pregunta 2
- Auxiliar 3 corrige Pregunta 3
- Auxiliar 4 corrige Pregunta 1
- Auxiliar 5 corrige Pregunta 2
- ...

Se pide construir un programa que permita leer TODOS LOS ARCHIVOS de las 3 preguntas del control 2, almacenándolas en memoria, para luego que un alumno ingrese su número interno muestre las notas de las 3 preguntas, el promedio del control y el código del auxiliar que le corrigió. Suponga que hay un máximo de 1000 alumnos (de 000 a 999) y que la estructura de los archivos es:

- Código interno del alumno (3 caracteres)
- Nota del alumno en la pregunta (3 caracteres con . en medio)

¿Por qué no podríamos resolver este problema? Es bastante sencillo pues tenemos toda la información posible. Veamos como se resolvería con lo que sabemos:

```
public class NotasControl {
	// Lector de la Entrada Estándar
	BufferedReader in = new BufferedReader(
		new InputStreamReader(System.in));

	static public void main (String args[]) {
		// Arreglos que almacenarán las notas y
		// el código del auxiliar que corrigió
		double[][] notas = double[1000][3];
		int[][] aux = int [1000][3];

		// Se inician notas en 1 y auxiliaries en 0
		for (int x=0; x<1000; x++) {
			for (int y=0; y<3; y++) {
```



```
				notas[x][y] = 1.0;
				aux[x][y] = 0;
			}
		}

		// Ciclo lector de las preguntas
		int nn = 1;
		for (int x=0; x<3; x++) {
			String nombre = "p";
			nombre += x;
			nombre += "-aux";
			if (nn < 10)
				nombre += "0";
			nombre += nn;

			BufferedReader bf = new BufferedReader(
				New FileReader(nombre));
			String linea;
			while( (linea=bf.readLine()) != null ) {
				int cod = Integer.parseInt(
					linea.substring(0, 3));
				double nota = new Double(
					linea.substring(3, 3)
					).doubleValue();

				notas[cod][x] = nota;
				aux[cod][x] = nn;
			}
			bf.close();

			nn++;
		}

		// Ya tenemos leido todos los datos.
		// Veamos el ciclo de consulta de notas.
		while(true) {
			System.out.print("Código de alumno?");
			int codigo = Integer.parseInt(in.readLine());

			System.out.println("Tus notas son: ");
			double suma = 0;
			for (int p=0; p<3; p++) {
				System.out.println("Pregunta " + (p+1) +
					" = " + notas[codigo][p] +
					" (Aux: " + aux[codigo][p] + ")");
				suma += notas[codigo][p];
			}
			double prom = suma / 3;
			System.out.println("Promedio = " + prom);
		}
	}
}
```

Podemos ver que la solución no es para nada difícil de realizar, sin embargo se están utilizando 2 arreglos para almacenar los valores. ¿Podría haber sido solo uno?.

Veremos que las Estructuras de Datos nos permiten realizar o almacenar objetos y cualquier tipo de elemento que se nos ocurra utilizando una clase la cual almacena la información que nosotros queremos manipular.

Concepto

Tipo de Dato Abstractos

> *Se denominará Tipo de Dato a una clase que será construida para almacenar un elemento especial. Nos permitirá modelar estructuras combinados con otros tipos u objetos de clases de Java.*

Esta definición nos abre la mente para trabajar en forma más genérica de lo que hemos estado haciéndolo hasta ahora. Por ejemplo nos gustaría modelar el ranking de la ATP sabiendo que:

- Todo tenista posee un nombre
- También una nacionalidad
- Y un cantidad de puntos

Unos dirían: "esto es una matriz". Pero no es tan sencillo, ya que la cantidad de puntos es un valor entero y los otros 2 campos son Strings. ¿Cómo mezclamos esto? Sencillo. Definamos el tipo de dato **Tensita**:

```
public class Tenista {
	public String nombre;
	public String pais;
	public int puntos;

	public Tenista (String nombre, String pais) {
		this.nombre = nombre;
		this.pais = pais;
		this.puntos = 0;
	}
}
```

Una sencilla clase que define por completo al tenista. Si observamos un poco el constructor que se ha declarado es la idea de un nuevo tenista que recién entra al ranking. Además, podemos definir todas aquellas funcionalidades que pueden ser útiles como **ganarPuntos(int puntos)** agrega la cantidad de puntos al tenista.

Pero ¿para qué sirve?. Bueno, si pensamos un poco, ahora bastaría crea solo 1 arreglo de tipo **Tenista** y cargar todo allí de la siguiente forma:

```
Tenista ranking[];
...
ranking[37] = new Tenista("Marcelo Ríos", "Chile");
ranking[37].ganarPuntos(1000);
```

y tenemos un nuevo tenista en la ATP con 1.000 puntos.

Ahora, ¿cuál sería la utilidad al momento de querer ordenar el arreglo? Analicemos este no tan sencillo problema y démosle una solución.

```
void bubbleSort (Tenista[] a, int nMin, int nMax) {
```



```
	// Versión Recursiva de BUBBLESORT
	if (nMax <= nMin)
		return;

	for (int j=nMin; j<nMax; j++) {
		if (a[j] > a[j+1]) {
			Tenista auxiliar = a[j];
			a[j] = a[j+1];
			a[j+1] = auxiliar;
		}
	}

	bubbleSort (a, nMin, nMax-1);
}
```

Todo está bien EXCEPTO por la línea destacada en la cual se comparan los elementos, ya que claramente se trata de comparar 2 objetos de tipo **Tenista** en donde no está definido el comparador **>** (mayor que) directamente.

Para solucionar esto, usaremos lo que en la última clase quedó como problema, utilizar una clase que permitiera que los objetos se compararan: **EsComparable**. Sin embargo, y gracias a los creadores de Java, se han adelantado a esto y la interface **Comparable** ya existe es:

```
public interface Comparable {
	public int compareTo (Object obj);
}
```

Con esto hacemos ahora que nuestra clase **Tenista** sea **Comparable**:

```
public class Tenista implements Comparable {
	public String nombre;
	public String pais;
	public int puntos;

	public Tenista (String nombre, String pais) {
		this.nombre = nombre;
		this.pais = pais;
		this.puntos = 0;
	}

	public int compareTo(Object obj) {
			return this.puntos - ((Tensita) obj).puntos;
	}
}
```

Entonces, podemos modificar nuestro ordenamiento y ahora permitir utilizar esta comparación:

```
void bubbleSort (Tenista[] a, int nMin, int nMax) {
	// Versión Recursiva de BUBBLESORT
	if (nMax <= nMin)
		return;

	for (int j=nMin; j<nMax; j++) {
		if (a[j].compareTo(a[j+1]) > 0) {
			Tenista auxiliar = a[j];
			a[j] = a[j+1];
			a[j+1] = auxiliar;
```

```
                }
        }

        bubbleSort (a, nMin, nMax-1);
}
```

¡Hemos creado un ordenador de objetos genéricos!

Si, ya que si miramos bien bastaría cambiar el método **compareTo** de la clase **Tensita** para cambiar la forma de ordenamiento y NO el método de ordenamiento.

Además, si ahora generalizamos un poco más la firma del método:

```
void bubbleSort (Comparable[] a, int nMin, int nMax) {
        ...
        // Cambiar Tensita por Comparable
        ...
}
```

Tendremos un método que puede ordenar CUALQUIER clase de Tipos de Dato que sean **Comparable**[20].

Interesante.

Estructura de Datos

> *Una Estructura de Datos es una combinación de elementos de un mismo Tipo de Dato tradicional o abstracto, que puede ser referenciado con variables y que posee algunas características que la hacen especial y diferente a otras estructuras de datos conocidas.*

Esta definición de lo que son las Estructuras de Datos nos permite ampliar el espectro de los que son los Arreglos de Datos a otras estructuras más comeplejas.

Por la definición, un arreglo es una Estructura de Datos, ya que nos permite almacenar un montón de elementos del mismo tipo en una misma variable. Veamos otras estructuras de datos que son útiles y que se implementan usando tipos de datos conocidos. Luego veremos tipos nuevos con los cuales se podrían implementar las estructuras:

Sintaxis

Pilas y Colas

Estas estructuras de datos tienen algunas características especiales y pueden ser representadas a través de arreglos de valores. Sin embargo ambos son encapsulados para convertirlos en estructuras.

[20] El Tipo de Dato **String** es comparable por definición. Solo en el caso de los tipos numéricos nativos no funcionaría este método, pero ellos tienen comparadores >, <, >=, <= y ==.

Pila (LIFO): Es una estructura lineal que se asemeja a las torres de cosas (de platos, de papeles, de tarros de salsa o de objetos). Su característica principal está dada por la sigla con la cuál se les llama LIFO ("*Last In First Out*") que significa que el último que llega a una pila es el primero que sale.

Por ejemplo: Si ponemos un CD sobre otro CD y así sucesivamente hasta llegar a 20 CD's apilados, solo podremos VER y/o SACAR el de arriba sin afectar a los demás, ya que este fue el último que puse en la pila de CD's.

Su implementación puede ser hecha a través de un arreglo[21] (con un margen máximo) encapsulada dentro de una clase **Pila** (la haremos de objetos que son comparables. En el caso del tipo de dato String funciona perfectamente, ya que es comparable):

```
class Pila {
        protected Comparable[] pila;
        protected int n;

        public Pila (int n) {
                this.pila = new Comparable[n];
                this.n = 0;
        }

        public void push(Comparable obj) {
                if (this.n == this.pila.length)
                        return;   // PILA LLENA
                this.pila[this.n] = obj;
                this.n++;
        }

        public Comparable pop() {
                if (this.n == 0)
                        return null;   // PILA VACIA
                this.n--;
                return this.pila[this.n];
        }
}
```

La definición estándar de pila indica que debe tener 2 métodos: uno para poner al tope de la pila (**push**) y otro para sacar del tope de la pila (**pop**). El constructor solo crea la pila con un máximo número de elementos y luego pone el tope en 0 para indicar que el siguiente elemento que puede entrar, entrará en esa posición de la pila.

Un ejemplo de utilización de la pila puede ser:

```
Pila p = new Pila (100);
Tenista t1 = new Tenista("Marcelo Ríos", "Chile");
Tenista t2 = new Tenista("Boris Becker", "Alemania");
p.push(t2);
p.push(t1);
```

[21] No es la única forma de representación. Más adelante se ven otras formas de implementar una Pila y una Cola.

Este ejemplo pone a Marcelo Ríos en el tope de la pila dejando a Boris Becker para salir después de Marcelo.

Cola (FIFO): Es una estructura lineal que se asemeja a las filas o colas en la vida real (la cola de un banco, la fila del colegio o fila de fichas de dominó). Su característica principal está dada por la sigla con la cuál se les llama FIFO ("*First In First Out*") que significa que el primero que llega a una cola es el primero que sale (que se atiende).

El ejemplo más clásico es el de los bancos, ya que el que llega primero a la fila siempre sale primero del banco. Aunque aquí ocurren algunas singularidades, por ejemplo, paso a embarazadas y ancianos, etc, en el fondo es una cola FIFO.

Al igual que en el caso de las pilas, las colas tienen una representación en Java a través de un arreglo circular que se encapsula en una clase llamada **Cola**:

```
class Cola {
      protected Comparable[] cola;
      protected int beg, end;
      protected boolean full;

      public Cola (int n) {
            this.cola = new Comparable[n];
            this.beg = 0;
            this.end = 0;
            this.full = false;
      }

      public void put(Comparable obj) {
            if (full)
                  return;   // COLA LLENA
            this.cola[this.end] = obj;
            this.end++;
            full = (this.beg == this.end);
      }

      public Comparable get() {
            if (this.beg == this.end)
                  return null;   // COLA VACIA
            this.beg++;
            return this.cola[this.beg-1];
      }
}
```

La definición estándar de pila cola que debe tener 2 métodos: uno para poner al final de la cola (**put**) y otro para sacar del inicio de la cola (**get**). El constructor solo crea la cola con un máximo número de elementos y luego pone el final en 0 para indicar que el siguiente elemento que puede entrar, el inicio en 0 porque la cola está vacía y un indicador booleano que dirá cuándo estará llena la cola.

Un ejemplo de utilización de la cola puede ser:

```
Cola p = new Cola (100);
Tenista t1 = new Tenista("Marcelo Ríos", "Chile");
```



```
Tenista t2 = new Tenista("Boris Becker", "Alemania");
p.put(t2);
p.get(t1);
```

Este ejemplo, a diferencia del de Pilas (aunque parezcan iguales) pone a Marcelo Ríos en el final de la cola dejando a Boris Becker para salir antes que Marcelo de la fila.

Pero la mejor forma de entender estas cosas es practicando, así que, manos a la obra.

Listas Enlazadas

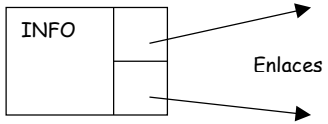
Nodo: Tipo de Dato Abstracto que permite almacenar elementos con cierta estructura y que se pueden enlazar con uno o más elementos del mismo tipo.

Esta sencilla definición es para declarar tipos de datos dinámicos que nos permitas ir combinando las propiedades de los arreglos, pero en forma más dinámica y crecida.

Un Nodo se compone genéricamente de:

- **Campo de Información**: Almacena la información asociada al Tipo de Dato que define el Nodo.
- **Punteros**: Enlace a otro (s) elemento(s) del mismo tipo de dato.

Gráficamente se vería como:

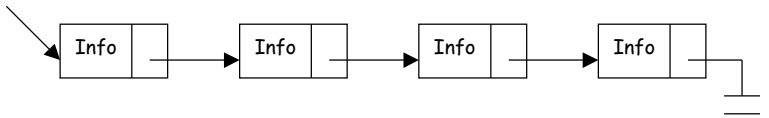



```
public class Nodo {
   Object info;
   Nodo sgte;
   public Nodo(Object o) {
      this.info = o;
      this.sgte = null;
   }
}
```

Lista Enlazada (1): Conjunto de Nodos que permiten simular un arreglo dinámico, es decir, que puede crecer con el tiempo.

Lista Enlazada (2): Estructura de Datos que permite almacenar un número variable de Nodos y que puede estar restringido SOLO por la cantidad de memoria del procesador que está almacenando la lista.

En forma rápida y sencilla, una Lista Enlazada es la estructura que es formada por un conjunto de Nodos enlazados entre sí. Una lista con enlace simple se vería como:

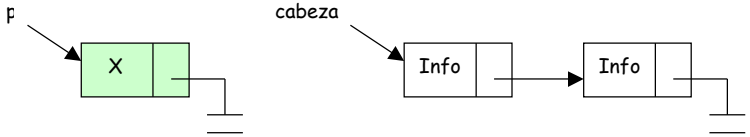


Para poder referencias esta estructura de datos se requiere de una **Cabeza Lectora** que apunta al inicio de la lista enlazada, y el resto se recorre siguiente los punteros al siguiente elemento de la lista (es como caminar de Nodo en Nodo).

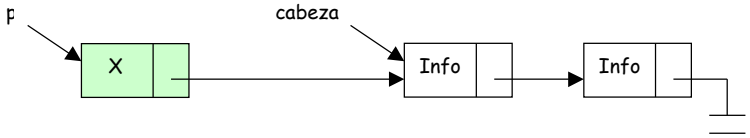
Pero veámoslo de verdad cómo funcionan.

Inserción al Inicio de la Lista: Veamos gráficamente cómo funciona este caso:

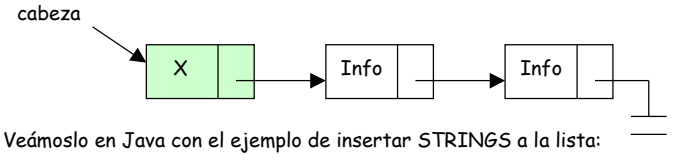
Paso 1: Creamos un nuevo NODO y lo apuntamos por **p**.



Paso 2: Apuntamos el siguiente de **p** a lo que apunta la **cabeza**.



Paso 3: Permitimos que la **cabeza** ahora referencie como primer elemento de la lista a **p**.



Veámoslo en Java con el ejemplo de insertar STRINGS a la lista:

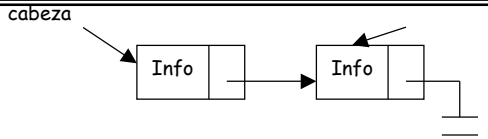
```
// La lista está vacía
Nodo cabeza = null;

// Ingreso de nombres:
while(true) {
       System.out.print ("Nombre? ");

       // Creamos el nodo con su nombre
       Nodo p = new Nodo(in.readLine);

       // Luego lo ponemos en la lista
       p.sgte = cabeza;
       cabeza = p;
}
```

Pero esta solo es una forma de insertar un nuevo Nodo a una lista enlazada. En general existen 3 casos de cómo insertar un elemento: Al inicio de la lista (lo visto anteriormente), en medio y al final. Veamos gráficamente cómo se insertan en los siguientes 2 casos:

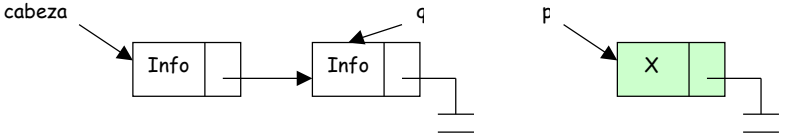
Inserción Al Final de la Lista:

Paso 1: Recorremos la lista hasta el elemento último con un iterador **q**.

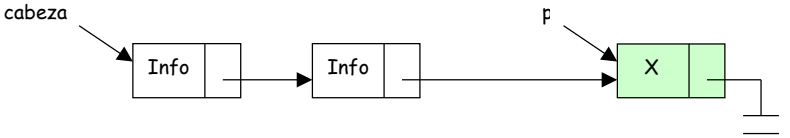
q



Paso 2: Creamos un nuevo NODO y lo apuntamos por **p**.



Paso 3: Apuntamos el siguiente de **q** a lo que apunta el **p**.



Veamos como se implementan estas líneas en lenguaje Java con el mismo ejemplo anterior:

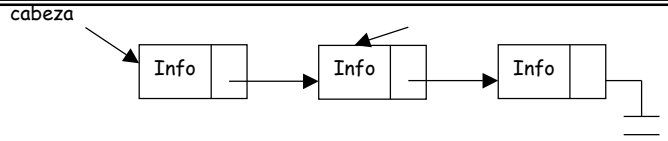
```
// La lista está vacía
Nodo cabeza = null;

// Ingreso de nombres:
while(true) {
       System.out.print ("Nombre? ");

       // Iteramos hasta que llegamos al final
       Nodo q = cabeza
       while (q != null && q.sgte != null)
              q = q.sgte;

       // Creamos el nodo con su nombre
       Nodo p = new Nodo(in.readLine);

       // Luego lo ponemos en la lista
       if (q != null) {
              q.sgte = p;
       }
       else { // Lista vacía
              cabeza = p;
       }
}
```

Ummm... Se puede ver que este caso es un poco más complejo que el de insertar al inicio de la lista, pero se asemeja más al caso general que viene en seguida:

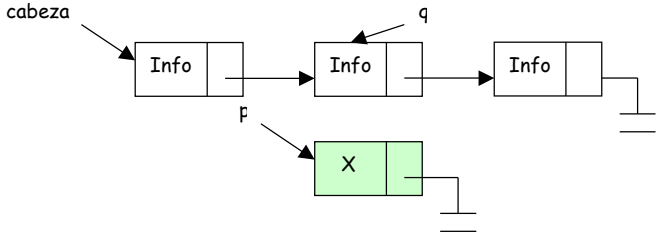
Inserción En Medio de la Lista:

Paso 1: Recorremos la lista hasta antes de la posición donde debe ir con un iterador **q**.

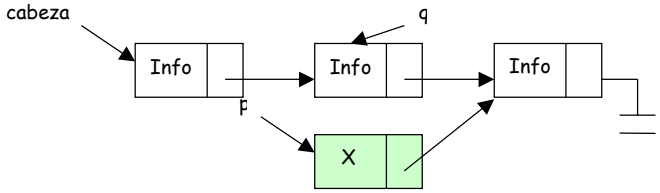
q



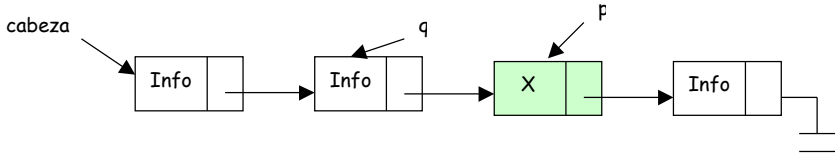
Paso 2: Creamos un nuevo NODO y lo apuntamos por **p**.



Paso 3: Apuntamos el siguiente de **p** a lo que apunta el siguiente de **q**.



Paso 4: Apuntamos el siguiente de **q** a lo que apunta **p**.



Veamos como se implementan estas líneas en lenguaje Java con el mismo ejemplo anterior:

```
// La lista está vacía
Nodo cabeza = null;

// Ingreso de nombres:
while(true) {
      System.out.print ("Nombre? ");

      // Iteramos hasta que llegamos al punto de inserción
      Nodo q = cabeza
      while (q != null && <condición de inserción>)
            q = q.sgte;

      // Creamos el nodo con su nombre
      Nodo p = new Nodo(in.readLine);
```

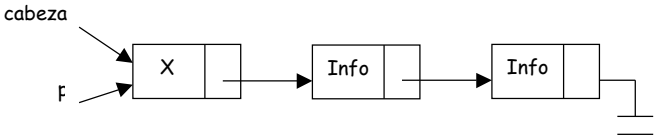


```
      // Luego lo ponemos en la lista
      if (q != null) {
            p.sgte = q.sgte;
            q.sgte = p;
      }
      else { // Lista vacía
            cabeza = p;
      }
}
```

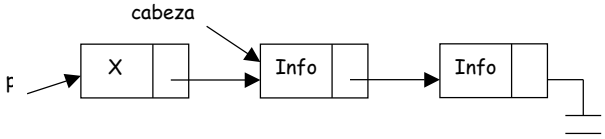
Nos falta ver como eliminar de una lista enlazada en sus 3 casos también.

Eliminación Al Inicio de la Lista:

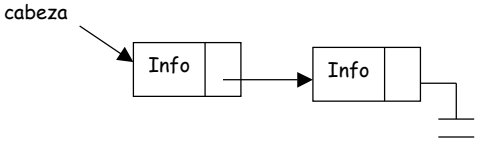
Paso 1: Apuntamos **p** a la cabeza de la lista.



Paso 2: Apuntamos la cabeza de la lista al siguiente de **p**.



Paso 3: Se destruye lo apuntado por **p**.



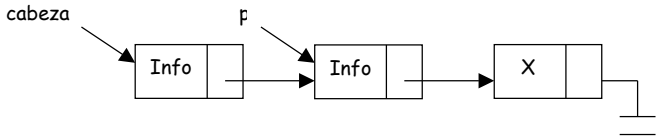
Veamos como se implementan estas líneas en lenguaje Java con el mismo ejemplo anterior:

```
Nodo p = cabeza;
cabeza = p.sigte;
p = null;
```

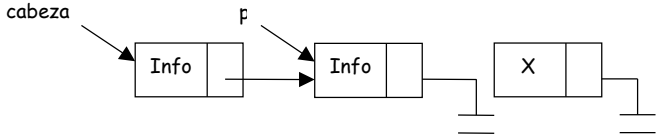
En general se puede o no hacer el null para liberar la variable ya que Java posee un Garbage Collector que limpia cada cierto tiempo la memoria de variables que apuntan a nada.

Eliminación Al Final de la Lista:

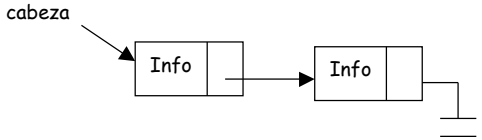
Paso 1: Apuntamos **p** al penúltimo elemento de la lista.



Paso 2: Apuntamos la cabeza de la lista al siguiente de **p**.



Paso 3: Se destruye solito el nodo (por el GC[22]).



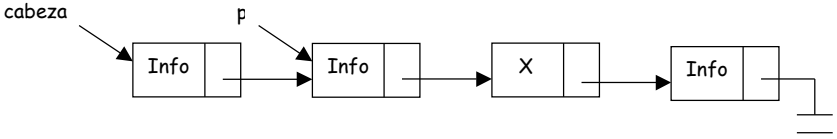
Veamos como se implementan estas líneas en lenguaje Java con el mismo ejemplo anterior:

```
Nodo p = cabeza;

while(p != null && p.sgte != null && p.sgte.sgte != null) {
      p = p.sgte;
}

if (p != null && p.sgte != null) {
      p.sgte = null;
}
else if (p != null) {
      // Es el único caso especial
      cabeza = null;
}
```

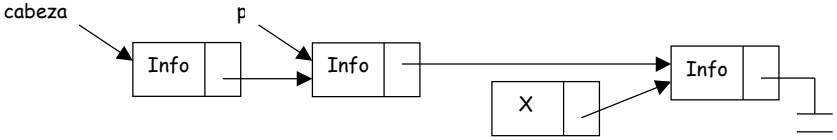
[22] Garbage Collector: Se preocupa de limpiar la memoria inútil del computador convirtiéndola en memoria útil.

Eliminación En Medio de la Lista:

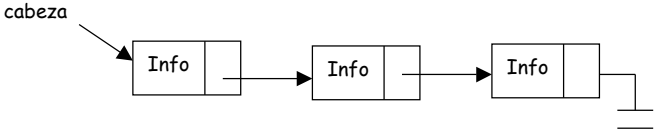
Paso 1: Apuntamos **p** al anterior del elemento de la lista a eliminar.



Paso 2: Apuntamos el siguiente de **p** al siguiente del siguiente de **p**.



Paso 3: Se destruye solito el nodo (por el GC).



Veamos como se implementan estas líneas en lenguaje Java con el mismo ejemplo anterior. Hay que tener cuidado con los casos de borde (primero y último elemento):

```
Nodo p = cabeza;

while(p != null && p.sgte != null && CONDICIÓN PARA ENCONTRARLO) {
      p = p.sgte;
}

if (p != null && p.sgte != null) {
      p.sgte = p.sgte.sgte;
}
else if (p != null) {
      // Es el primero o el último
      if (p == cabeza) {
            cabeza = p.sgte;
      }
      else {
            p.sgte = null;
      }
}
```

Árboles

Arbol: Estructura de datos que permite almacenar información y organizarla de tal forma que tengas sucesores o elementos siguientes como hijos en forma de ramas de un árbol.

Esta definición bastante rara es una forma genérica con la cual se definen los árboles.

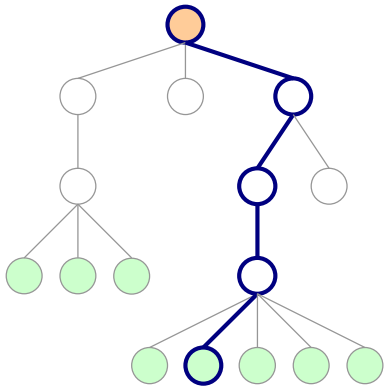
Algunas definiciones útiles para comprender la nomenclatura es:

RAIZ es el primer nodo del árbol y es por donde se accede a él, es decir, la "cabeza" del árbol siempre será la raíz.

HOJAS del árbol son todos aquellos nodos que estén dentro del árbol pero que NO tengan ningún hijo.

NODO INTERNO son todos aquellos nodos que no son ni raíz ni hojas.

ALTURA es la cantidad de nodos que hay que recorrer para llegar desde la raíz del árbol hasta la hoja más alejada de la raíz. También se define como la máxima distancia que hay entra la raíz y todas las hojas del árbol.



Existen varios tipos de árboles, de entre los cuales mencionaremos los más comunes.

Árboles Binarios: Son árboles que poseen 2 nodos hijos, uno izquierdo y otro derecho. En general se utilizan para indicar orden dentro de los elementos del árbol, es decir, los hijos izquierdo son siempre MENORES que el nodo, y los hijos derecho son siempre MAYORES que el nodo que los contiene.

Un ejemplo de árbol puede ser aquél que va almacenando número enteros dependiendo del orden de ingreso de los valores (no requiere se ordenado como los arreglos o las listas):

```
class Nodo {
	public int info;
	public Nodo izq, der;
	public Nodo(int o) {
		this.info = o;
		this.izq = null;
		this.der = null;
	}
}
```

Como podemos ver, la representación del Nodo es igual a la representación de una lista doblemente enlazada. La diferencia está en el momento de realizar el enlace, pues este "enlace" se hace a nodos que no son "correlativos". Implementemos el problema de insertar números ordenadamente:

```
class Arbol {
	protected Nodo raiz;

	public Arbol() {
		this.raiz = null;
	}
```

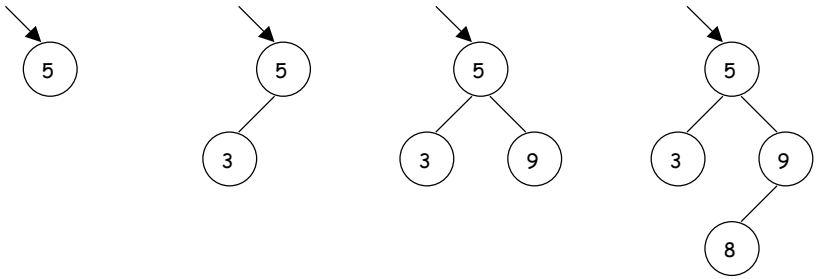


```
	// Versión iterativa del insertar
	public void insertar(int x) {
		Nodo p = new Nodo(x);
		Nodo q = this.raiz;
		Nodo f = null;
		while (q != null) {
			f = q;
			if (q.info > x) q = q.izq;
			else q = q.der;
		}
		if (f == null)
			this.raiz = p;
		else {
			if (f.info > x) f.izq = p;
			else f.der = p;
		}
	}
}
```

Si analizamos el insertar, lo que va haciendo en su forma general es:

1. Preguntar si donde estoy va.
2. Si va, se inserta.
3. Si no va, se busca por el lado en que debería ir

El insertar visto aquí es solo un ejemplo para entender cómo se va llenando el árbol. Pues bien, veamos gráficamente como quedaría un árbol si vamos leyendo los datos en el siguiente orden: 5, 3, 9 y 8.



Vemos rápidamente que TODO lo que está a la derecha es mayor a 5 y lo que está a la izquierda es menor a 5. Intenta completar esto metiendo los número 2, 7, 6 y 4.

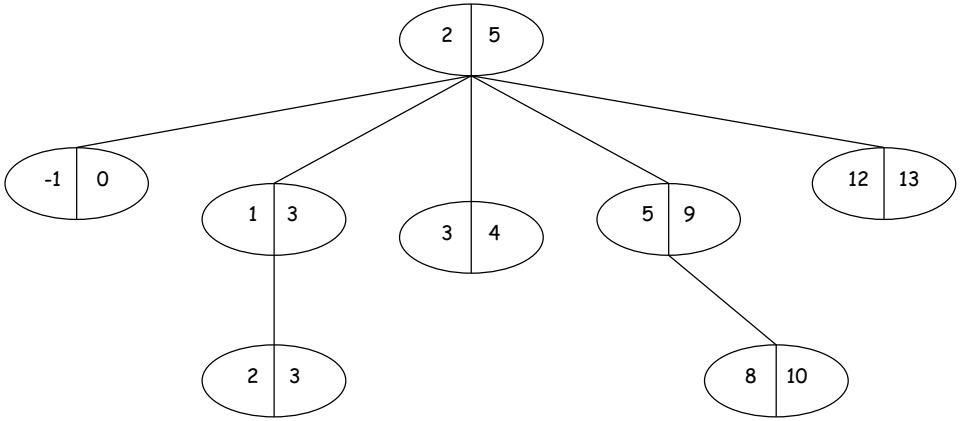
Árboles Genéricos: Son árboles que pueden tener un número de hijos no definidos, es decir, con 2, 3, 4... n hijos. El árbol binario es una clase especial de árboles genéricos que solo posee 2 hijos, pero no es limitante.

Por ejemplo, usemos árboles con número que cumplan la siguiente propiedad:

- Cada nodo almacenada la información de un intervalo de enteros.
- Los hijos izquierdos son los nodos menores al intervalo.

- Los hijos izquierdo-centrales son nodos que intersectan el intervalo por la izquierda.
- Los hijos centrales son los nodos que están dentro del intervalo.
- Los hijos derecho-centrales son nodos que intersectan el intervalo por la derecha,
- Los hijos derechos son nodos mayores al intervalo.

Entonces, un árbol como este quedaría así:



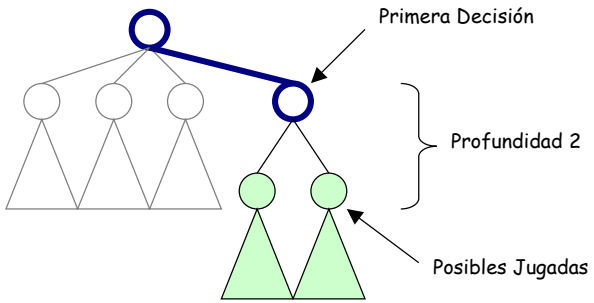
Como podemos ver en el ejemplo, el número de hijos es variable, pero entre 0 y 5.

Propuesto: Implemente el tipo Nodo que permita crear este tipo de árboles.

Y ¿para qué sirven estos árboles? En el fondo, lo que hace un computador con un juego de ajedrez es analizar una serie de movidas prefabricadas dependiendo de la posición en la que se encuentre el tablero. Estas posiciones pueden ser MILLONES y sería bastante difícil realizar una búsqueda en un arreglo o listas de millones de datos para encontrar la mejor movida.

Así que los árboles permiten ir seleccionando un **Nodo** (un estado) que indica la posición del tablero, y sus **Hijos** (que pueden ser muchos) que representan las posibles movidas que tiene el computador en ese momento. Además, las hojas del *árbol de decisión* son los posibles términos del juego. ¡Hey!, ¿pero eso significaría que el computador siempre sabe cómo ganar? Por supuesto que no, ya que si analizara el camino más corto para llegar a una hoja, esto le tomaría a la computadora más rápida unos días, y es mucho en un juego de ajedrez. Es por eso que utilizan profundidad y evalúan la mejor jugada según un mini-árbol de altura **n** con el cual puede jugar.

Es decir:



¿Interesante no?. Pero esto árboles ya son difíciles de trabajar, por lo que nos centraremos en los árboles binarios como nuestra nueva estructura de datos.

Definiremos entonces un **Nodo** para un árbol binario de la siguiente forma:

```
class Nodo {
      public Object info;    // Recuerda que el tipo puede cambiar
      public Nodo izq, der;  // Los nodos hijos

      public Nodo(int o) {   // Constructor para crear un nodo
            this.info = o;
            this.izq = null;
            this.der = null;
      }
}
```

Esta forma básica cumple con lo mínimo que es necesario para tener un nodo de árbol, por lo que se le puede agregar cosas como que sean comparables, o tal vez un criterio de decisión de cuál va a la derecha o izquierda de otro.

Entonces, para insertar en el árbol, en su forma genérica, es:

```
// Creación del nuevo nodo
Nodo p = new Nodo(x);

// Inicio del recorrido del árbol hasta el lugar (f) donde va.
Nodo q = this.raiz;
Nodo f = null;
while (q != null) {
      f = q;
      if (q.info > p.info)// Criterio para que vaya por la izquierda
            q = q.izq;
      else                // Criterio para que vaya por la derecha
            q = q.der;
}

// Inserción
if (f == null)            // Si es la raíz del árbol (árbol vacío)
      this.raiz = p;
else {                    // Si en f tenemos una hoja
      if (f.info > p.info)  // Si va a la izquierda de la hoja (f)
```

```
            f.izq = p;
      else                    // Si va a la derecha de la hoja (f)
            f.der = p;
}
```

Esta versión **iterativa** de la inserción se basa en que los nodos son valores comparables como los números. Pero es fácil transformarlo para que funcione con otros tipos de datos.

Ahora bien, es bastante sencillo insertar, sin embargo, el problema viene cuando hay que eliminar, porque si eliminamos un nodo interno ¿qué hacemos con sus hijos?. Veamos todos los casos:

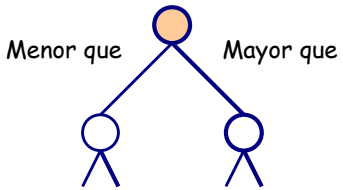
```
// Recorrido para encontrar el valor x
Nodo q = raiz;
Nodo f = null;   // Guardaremos el padre del que vamos a eliminar.
while(q != null && q.info <> x) {
      f = q;
      if (q.info > p.info)// Criterio para que vaya por la izquierda
            q = q.izq;
      else                // Criterio para que vaya por la derecha
            q = q.der;
}

// Eliminamos
if (q != null)  {    // Si fuera null, significa que no está.
      // Se guardan los hijos del que vamos a eliminar
      Nodo qi = q.izq;
      Nodo qd = q.der;

      // Se elige cuál de los dos va a colgar del padre de q
      // En este caso elegiremos el  árbol derecho pero
      // Podría haber sido al revés.
      f = qd;

      // Luego se pone el lado derecho, en la hoja más a
      // la izquierda de qd.
      Nodo r = qd;
      Nodo hi = null;
      while(r != null) {
            hi = r;
            r = r.izq;
      }

      if (hi != null)   // Si existe la rama izquierda
            hi.izq = qi;
      else
            f = qi;
}
```

¡OJO QUE QUEDA PENDIENTE BORRAR LA RAIZ! (**Propuesto**).

Árboles de Búsqueda Binaria: Los árboles de búsqueda binaria (ABB) son aquellos que sus hijos cumplen una condición de orden con sus padres, que no permiten duplicados y que sirven para realizar búsquedas en un tiempo O(Log n) en un conjunto de datos "ordenados".

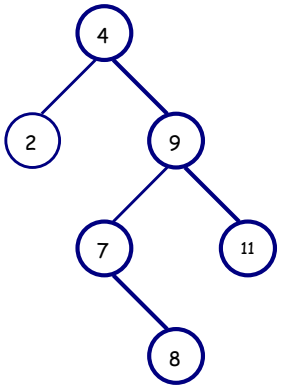
Gran parte de la utilidad de los árboles binarios y en especial de los ABB es que los nodos que se van insertando en el árbol, siempre van quedando en forma "ordenada".

Pero ¿a qué se refiere con "ordenado"?. La condición de que un árbol sea ABB es que el sub-árbol hijo izquierdo es siempre MENOR que el nodo y el sub-árbol hijo derecho es siempre MAYOR que el nodo.



Esto permite que las búsquedas se reduzcan bastante, ya que en el caso promedio NO se debe recorrer el árbol completo para encontrar el valor buscado.

Veamos un ejemplo: Dado el siguiente árbol:



Buscar el número 7 costaría solo pasar por 3 nodos:

1. Comparar el 7 con la raíz del árbol (4). Como es mayor, nos iremos por la derecha.
2. Comparar el 7 con el nodo de la derecha de la raíz (9). Como es menor, nos vamos por la izquierda.
3. Comparar el 7 con el nodo de la izquierda de (9). Como es el 7... ¡BINGO!

Si lo hubiésemos dispuesto en una lista enlazada, ésta quedaría ordenada de la siguiente forma: 2-4-7-8-9-11. Casualmente es la misma cantidad de nodos. Pero si vamos al caso del 11, en el árbol nos cuesta 3 nodos, y en la lista enlazada 6 nodos, es decir, el tamaño de la lista. Malo, malo, malo.

Veamos como quedaría el algoritmo de búsqueda de un nodo en un árbol ABB:

```
// Suponemos x como valor para buscar

// Para recorrer
Nodo q = raiz

while (q != null && q.info <> x) {
      if (q.info > x)
            q = q.izq;
      else
            q = q.der;
}

if (q == null)
```

```
        // NO LO ENCONTRAMOS
else
        // LO ENCONTRAMOS y está en q
```

El patrón es bastante simple, ya que lo único que hace es ir decidiendo que rama coger para continuar la búsqueda.

Con este último patrón de programación hemos visto los 3 básicos que son utilizados en los árboles binarios y ABB: **Inserción, Eliminación** y **Búsqueda**. Ahora veamos prácticamente cómo lo vemos esto dentro de un programa en Java:

Declaremos primero una clase que nos permita modelar el **Nodo** considerando números reales:

```
public class Nodo {
        public double info;
        public Nodo izq, der;

        public Nodo (double x) {
                this.info = x;
                this.izq = null;
                this.der = null;
        }
}
```

Ya tenemos lo que es un nodo. Ahora veamos una representación de árbol con las 3 operaciones básicas: Insertar un elemento, Eliminar un elemento y Buscar un elemento:

```
public class ABB {
        public Nodo raiz;

        // Inicializa el árbol de búsqueda binaria.
        public ABB() {
                this.raiz = null;
        }

        // Para insertar el valor x.
        public void insertar (double x) {
                // Creación del nuevo nodo
                Nodo p = new Nodo(x);

                // Se busca donde va el nuevo nodo
                Nodo q = this.raiz;
                Nodo f = null;
                while (q != null) {
                        f = q;
                        if (q.info > p.info)
                                q = q.izq;
                        else
                                q = q.der;
                }

                // Inserción del nuevo nodo
                if (f == null)
                        this.raiz = p;
                else {
                        if (f.info > p.info)
                                f.izq = p;
                        else
```



```
                                f.der = p;
                        }
        }

        // Para eliminar un valor x del árbol.
        // Retorna el nodo eliminado.
        // En caso de no encontrarlo, retorna null.
        public Nodo eliminar (double x) {
                // PENDIENTE
        }

        // Para buscar un valor x en el árbol
        // Será semi-recursivo
        public Nodo buscar (double x) {
                return buscar(x, this.raiz);
        }

        private Nodo buscar (double x, Nodo p) {
                // Casos base
                if (p == null) return null;      // No está
                if (p.info == x) return p;       // Lo encontramos

                // Paso recursivo
                if (p.info > x)
                        return buscar(x, p.izq);
                else
                        return buscar(x, p.der);
        }
}
```

Solo falta ver una aplicación práctica de estas componentes. Se desea organizar un grupo de valores tomados en el laboratorio. Para ello debe imitar el diálogo:

```
Ingrese valor? 3.7                      Buscar Lectura? 5.7
Ingrese valor? -1                       No está la lectura!!
Ingrese valor? 8.2
...                                     Buscar Lectura? -3.5
                                        Lectura encontrada!!
Ingrese valor?
       (SOLO ENTER)
                                        ...
Listas las lecturas!.
```

La solución sería bastante sencilla (main):

```
public class Lecturas {
        public static BufferedReader bf = new BufferedReader(
                new InputStreamReader(System.in));

        public static void main (String args[]) {
                // Creamos el árbol
                ABB arbol = new ABB();

                // Ingresamos los valores
                System.out.print("Ingrese Lectura?");
                while ( (String l = bf.readLine()).length() > 0) {
                        double x = new Double(l).doubleValue();
                        arbol.insertar(x);
                        System.out.print("Ingrese Lectura?");
                }
```

```
            System.out.println("Listas las lecturas!");

            // Ahora consultamos
            while (true) {
                  System.out.print("Buscar Lectura?");
                  String l = bf.readLine();
                  double x = new Double(l).doubleValue();

                  Nodo p = ABB.buscar(x);
                  if (p == null)
                        System.out.println("No está la lectura!");
                  else
                        System.out.println("Lectura encontrada!");
            }
      }
}
```

Y ahí está un problema con su ciclo completo, desde definir la estructura básica hasta un problema aplicado.

Solución a la Motivación

(a) Los auxiliares de CC10A corrigen las preguntas de un grupo de mechones de todas las secciones por separado. Se les ha pedido que cada vez que corrijan construyan un archivo llamado "**pX-auxNN**" en donde X indica el número de la pregunta y NN el número del auxiliar (numerado entre 01 y 18). Lo más simpático es la correspondencia de corrección, es decir siempre ocurre que:

- Auxiliar 1 corrige Pregunta 1
- Auxiliar 2 corrige Pregunta 2
- Auxiliar 3 corrige Pregunta 3
- Auxiliar 4 corrige Pregunta 1
- Auxiliar 5 corrige Pregunta 2
- ...

Se pide construir un programa que permita leer TODOS LOS ARCHIVOS de las 3 preguntas del control 2, almacenándolas en memoria, para luego que un alumno ingrese su número interno muestre las notas de las 3 preguntas, el promedio del control y el código del auxiliar que le corrigió. Suponga que hay un máximo de 1000 alumnos (de 000 a 999) y que la estructura de los archivos es:

- Código interno del alumno (3 caracteres)
- Nota del alumno en la pregunta (3 caracteres con . en medio)

Solución

```
class Alumno implements Comparable {
      public int codigo;
      public double[] notas;
      public int[] auxs;

      public Alumno(int codigo) {
            this.codigo = codigo;
            this.notas = new double[3];
```



```
            this.auxs = new int[3];
      }

      public void ponerNota(int p, double nota, int aux) {
            this.notas[p] = nota;
            this.auxs.[p] = aux;
      }

      public int compareTo(Object obj) {
            return this.codigo - ((Alumno) obj).codigo;
      }
}

class NotasControl {
      // Lector de la Entrada Estándar
      BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));

      static public void main (String args[]) {
            // Arreglo que almacenará las notas y
            // el código del auxiliar que corrigió
            Alumno[] als = new Alumno[1000];

            // Se inician notas en 1 y auxiliaries en 0
            for (int x=0; x<1000; x++) {
                  als[x] = new Alumno(x);
                  for (int y=0; y<3; y++) {
                        als[x].ponerNota(y, 1.0, 0);
                  }
            }

            // Ciclo lector de las preguntas
            int nn = 1;
            for (int x=0; x<3; x++) {
                  String nombre = "p";
                  nombre += x;
                  nombre += "-aux";
                  if (nn < 10)
                        nombre += "0";
                  nombre += nn;

                  BufferedReader bf = new BufferedReader(
                        New FileReader(nombre));
                  String linea;
                  while( (linea=bf.readLine()) != null ) {
                        int cod = Integer.parseInt(
                              linea.substring(0, 3));
                        double nota = new Double(
                              linea.substring(3, 3)
                              ).doubleValue();

                        als[cod].ponerNota(x, nota, nn);
                  }
                  bf.close();

                  nn++;
            }

            // Ya tenemos leido todos los datos.
            // Veamos el ciclo de consulta de notas.
            while(true) {
                  System.out.print("Código de alumno?");
                  int codigo = Integer.parseInt(in.readLine());
```

```
                    System.out.println("Tus notas son: ");
                    double suma = 0;
                    for (int p=0; p<3; p++) {
                          System.out.println("Pregunta " + (p+1) +
                                " = " + als[codigo].notas[p] +
                                " (Aux: " + als[codigo].auxs[p] +
                                ")");
                          suma += als[codigo].notas[p];
                    }
                    double prom = suma / 3;
                    System.out.println("Promedio = " + prom);
              }
        }
}
```

Queda más elegante que la versión anterior, porque además si quisiéramos ordenarlos de alguna forma bastaría llamar a uno de los métodos de ordenación adecuados para "comparables" y listo.

Solo recuerda:

Estructura \ TDA	Arreglo	Lista	Arbol
Pilas			
Colas			
Diccionarios			

Según esta tabla, las pilas y las colas se pueden implementar (es muy común) con Arreglos y Listas, en cambio existen otras estructuras como los diccionarios que se implementan con árboles.

Problemas

(a) Implemente un Tipo de Dato comparable que modele un platillo volador venusino. Los platillo voladores en Venus posee las siguientes características:

- Patente intergaláctica (alfanumérico)
- Nombre del piloto (alfanumérico)
- Número de pasajeros (entero)
- Velocidad expresada con respecto a la velocidad luz (real entro 0 y 1)

Además, permita que la comparación sea flexible, es decir, que posea un switch que permita comparar entre la Patente, Número de Pasajeros y la Velocidad del platillo.

Solución

```
class Platillo extends Comparable {
      public String patente;
      public String piloto;
      public int pasajeros;
      public double velocidad;
      public int compareRule;
```



```
      public Platillo (String patente, String piloto,
                  int pasajeros, double velocidad)  {
            this.patente = patente;
            this.piloto = piloto;
            this.pasajeros = pasajeros;
            this.velocidad = velocidad;
            this.compareRule = 1;            // Patente
      }

      public int compareTo(Object obj) {
            switch (compareRule) {
                  case 1:             // Patente
                        return this.patente.compareTo(
                              ((Platillo) obj).patente);
                  case 1:             // Piloto
                        return this.piloto.compareTo(
                              ((Platillo) obj).piloto);
                  case 1:             // Pasajeros
                        return this.pasajeros -
                              ((Platillo) obj).pasajeros;
                  case 1:             // Velocidad
                        return Math.round(this.velocidad -
                              ((Platillo) obj).velocidad);
                  default:     // Otros Casos
                        return -1;
            }
      }
}
```

(b) Implemente una estructura de dato basada en Pilas y Colas que controle la entrada y salida de platillos venusinos desde su espaciopuerto ubicado detrás de la Luna. Siga las siguientes características:

- Existen platillos que pueden entrar al final de una lista (número de pasajeros > 10) y otros que entran al principio de la lista (los de pocos pasajeros).
- Todos salen en orden, es decir, siempre sale el primero que está en la lista (que no siempre coincide con el primero que entró.
- El máximo de naves permitidas en el espaciopuerto son de 10.

Solución

```
class EspacioPuerto {
      protected String nombre;:
      protected Comparable[] esclusas;
      protected int beg, end;
      protected boolean full;

      public EspacioPuerto (String nombre, int n) {
            this.nombre = nombre;
            this.esclusas = new Comparable[n];
            this.beg = 0;
            this.end = 0;
            this.full = false;
      }

      // Entrada de Prioridad
      protected void push(Comparable obj) {
            if (full)
                  return;
```

```
            this.esclusas[this.beg] = obj;
            this.beg--;
            full = (this.beg == this.end);
      }

      // Entrada Normal
      protected void put(Comparable obj) {
            if (full)
                  return;
            this.esclusas[this.end] = obj;
            this.end++;
            full = (this.beg == this.end);
      }

      // Salida
      protected Comparable get() {
            if (this.beg == this.end)
                  return null;
            this.beg++;
            return this.esclusas[this.beg-1];
      }

      // Despegue encapsulado
      public Platillo despegue() {
            return (Platillo) this.get();
      }

      // Aterrizaje
      // Se indica el número de pasajeros y retorna donde entró
      // 0. Espaciopuerto Lleno
      // 1. Prioridad
      // 2. Normal
      public int aterrizaje(Platillo obj, int nPasajeros) {
            if ( obj.pasajeros > nPasajeros ) {
                  this.push(obj);
                  return 1;
            }
            else {
                  this.put(obj);
                  return 2;
            }
      }
}
```

(c) Simule el siguiente diálogo que ocurre en la torre de mando del espaciopuerto venusino.

```
Bienvenido al Espaciopuerto Lunia

Identifique el tipo de movimiento de naves? Salida
No hay naves para una salida

Identifique el tipo de movimiento de naves? Llegada
Indique la patente de la nave? XKL390
Nombre del piloto? KILTARO
Número de pasajeros? 16
Velocidad de acercamiento? 0.5
Que el vuelo XKL390 del sr KILTARO reduzca su velocidad en 0.2c
antes de aterrizar por la entrada ALFA del espaciopuerto.

Identifique el tipo de movimiento de naves? Llegada
Indique la patente de la nave? FWO442
Nombre del piloto? GURTINYU
```



```
Número de pasajeros? 3
Velocidad de acercamiento? 0.7
Que el vuelo FWO442 del sr GURTINYU reduzca su velocidad en 0.6c
antes de aterrizar por la entrada BETA del espaciopuerto.

Identifique el tipo de movimiento de naves? Salida
El vuelo FWO442 tiene el espacio libre para su despliegue por la
salida GAMMA del espaciopuerto. Sr. GURTINYU se le agradece su
visita a Lunia.
```

Nota: Todos los platillos deben entrar a 0.3c por la entrada BETA y a 0.1c por la entrada ALFA. Los comandos válidos son: Salida (para salida de una nave), Llegada (para ingresar una nave), Fin (para terminar). Si no hay espacio en el espaciopuerto, el sistema debe retornar "Que el vuelo espere en el cuadrante X58 mientras se desocupa una de las esclusas de aterrizaje".

Solución

```
BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
System.out.println("Bienvenido al Espaciopuerto Lunia");
EspacioPuerto ep = new EspacioPuerto("Lunia", 10);

while(true) {
      System.out.print("Identifique el tipo de movimiento "+
            de naves?");
      String acc = in.readLine();

      switch (acc.toLowerCase()) {
            case "salida":
                  Platillo p = ep.despegue();
                  if (p == null)
                        System.out.println("No hay naves para "+
                              "una salida");
                  else
                        System.out.println("El vuelo "+ p.patente+
                              " tiene el espacio libre para "+
                              "su despliegue por la salida "+
                              "GAMMA del espaciopuerto. Sr. " +
                              p.piloto + " se le agradece su "+
                              "visita a Lunia.");
            case "llegada":
                  Platillo p = new Platillo("", "", 0, 0.0);
                  System.out.print("Indique patente de la nave?");
                  p.patente = in.readLine();
                  System.out.print("Nombre del piloto?");
                  p.piloto = in.readLine();
                  System.out.print("Número de pasajeros?");
                  p.pasajeros = Integer.parseInt(in.readLine());
                  System.out.print("Velocidad de acercamiento?");
                  p.velocidad = new Double(
                        in.readLine()).doubleValue();
                  int entrada = ep.aterrizaje(p, 10);
                  if (entrada == 0)
                        System.out.println("Que el vuelo espere "+
                              "en el cuadrante X58 mientras "+
                              "se desocupa una de las esclusas "+
                              "de aterrizaje");
                  else {
                        if (entrada == 1)
                              String puerta = "ALFA";
```

```
                        else
                              String puerta = "BETA";
                        System.out.println("Que el vuelo " +
                              p.patente + " del sr " + p.piloto +
                              " reduzca su velocidad en " +
                              (p.velocidad - 0.3) +
                              "c antes de aterrizar por " +
                              "la entrada " + puerta +
                              " del espaciopuerto.");
            case "fin":
                  break;
            default:
                  System.out.println("Opción no válida");
      }
}
in.close();
```

(d) Se desea implementar un codificador **Morse** de textos utilizando un ABB en donde:

- Por la derecha de un Nodo corresponde a un punto (.)
- Por la izquierda de un Nodo corresponde a una raya (-)

Además, la cantidad de puntos y rayas dependen a qué profundidad dentro del árbol se encuentra la letra ubicada.

1) Defina la clase nodo que almacena la letra.

```
public class NodoMorse {
      public String letra;
      public NodoMorse punto, raya;

      public NodoMorse(String letra) {
            this.letra   = letra;
            this.punto   = null;
            this.raya    = null;
      }
}
```

2) Implemente el método aleatorio que construye el árbol Morse con las letras del alfabeto. Suponga que existe un método estático en la clase **Morse** llamado **String codigoMorse(String letra)** que dada una letra del alfabeto, retorna la cadena de puntos y rayas que representan esa letra en código morse.

Nota: Recuerde que la raíz del árbol debe ser pasada por parámetro para hacerlo recursivo.

```
private void insertarLetra (NodoMorse nodo,
                            String letra,
                            String cadenaMorse) {
      // No se puede insertar
      if (cadenaMorse.length <= 0)
            return;

      // Si se puede insertar
      if (cadenaMorse.length() == 1) {
```



```
            if (cadenaMorse.equals("."))
                  nodo.izq = new NodoMorse(letra);
            else
                  nodo.der = new NodoMorse(letra);
            return;
      }

      // Si no, buscar donde
      if (cadenaMorse.charAt(0).equals("."))
            insertarLetra (nodo.izq, letra,
                           cadenaMorse.substring(1));
      else
            insertarLetra (nodo.der, letra,
                           cadenaMorse.substring(1));
}

public void llenarArbol (NodoMorse nodoRaiz) {
      String alfabeto = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

      // La raíz del árbol no tiene letra
      nodoRaiz = new NodoMorse("");

      // Insertar cada una
      for (int i=0; i<alfabeto.length(); i++)
            insertarLetra(nodoRaiz,
                          alfabeto.charAt(i),
                          Morse.codigoMorse(alfabeto.charAt(i));
}
```

3) Desarrolle el método que le permita pasar escribir desde una codificación morse a texto normal utilizando el patrón de búsqueda en árboles binarios.

Nota: Suponga que la raíz del Árbol está en **this.raiz**. Recuerde que cada letra de la palabra se separa por 1 espacio y las palabras por 2 espacios.

```
private String buscarLetra(NodoMorse p, String cadena) {
      // No hay que transformar
      if (p == null)
            return "";

      // Saca letra
      String caracter = cadena.charAt(0);
      if (cadena.length() == 1) {
            if (caracter.equals("."))
                  return p.izq.letra;
            else
                  return p.der.letra;
      }

      // Buscamos recursivamente
      cadena = cadena.substring(1);
      if (caracter.equals("."))
            return buscarLetra(p.izq, cadena);
      else
            return buscarLetra(p.der, cadena);
}

public String convierteTexto (String textoMorse) {
      NodoMorse p = this.raiz;
      String morse = textoMorse;
```

```
        String textoFinal = "";
        int esp = morse.indexOf(" ");
        while (esp >= 0) {
                String cadena = morse.substring(0, esp);
                morse = morse.substring(esp + 1);

                String letra = buscarLetra(raiz, morse);
                if (letra.length() <= 0)   // No hay letra, era espacio
                        textoFinal = textoFinal + " ";
                else
                        textoFinal = textoFinal + letra;

                esp = morse.indexOf(" ");
        }

        return textoFinal;
}
```

4) Escriba la clase **Morse** que contiene los método anteriormente pedidos en (b) y (c) y cree un constructor que permita llenar automáticamente el árbol. Solo enuncie el método estático **codigoMorse** mediante su firma. Además agregue un método que transforme (utilizando codigoMorse) un texto en codificación morse.

```
public class Morse {
        private NodoMorse raiz;

        // Constructor
        public Morse() {
                llenarArbol(this.raiz);
        }

        // Método Estático
        public static String codigoMorse(String letra) {...}

        // Métodos antes vistos
        public void llenarArbol (NodoMorse nodoRaiz) {...}
        private void insertarLetra (NodoMorse nodo,
                                    String letra,
                                    String cadenaMorse) {...}

        private String buscarLetra(NodoMorse p, String cadena) {...}
        public String convierteTexto (String textoMorse) {...}

        // Transforma a morse
        public String convierteMorse (String texto) {
                String textoMorse = "";
                for (int i=0; i<texto.length(); i++) {
                        if (texto.charAt(i).equals(" ");
                                textoMorse = textoMorse + " ";
                        else
                                textoMorse = textoMorse +
                                        Morse.codigoMorse(
                                                texto.charAt(i));
                }
                return textoMorse;
        }
}
```

5) **Propuesto:** Aplique todo lo anterior al siguiente diálogo.


```
Codificador Morse
Creando Diccionario... Ok!

Ingrese una frase: Hola cómo estás
En morse se escribe: [TEXTO EN MORSE]

Ingrese un texto en morse: ... --- ...
El texto que escribió: SOS
```

(e) Implemente la Estructura de Datos **Lista** que permita manipular una lista enlazada con los siguientes métodos:

Método	Descripción
Lista()	Crea una lista vacía
void poner(Nodo o, int i)	Pone en la posición i-ésima (como en un arreglo)
Nodo sacar(int i)	Saca el i-ésimo elemento (como en un arreglo)

Solución

```
class Lista {
        private Nodo cabeza;
        public int largo;      // Solo si es necesario

        public Lista() {
                this.cabeza = null;
                this.largo = 0;
        }

        public void poner(Nodo o, int i) {
                if (i > this.largo)
                        return;          // No se pone más allá del largo

                Nodo q = this.cabeza
                for (int n=0; n<i; n++)
                        q = q.sgte;

                o.sgte = q.sgte;
                q.sgte = o

                this.n++;
        }

        public Nodo sacar(int i) {
                if (i >= this.largo)
                        return null;     // No existe más allá del largo

                Nodo q = this.cabeza
                for (int n=0; n<i; n++)
                        q = q.sgte;

                Nodo p = q.sgte;
                q.sgte = p.sgte;
                this.n--;

                return p;
        }
}
```

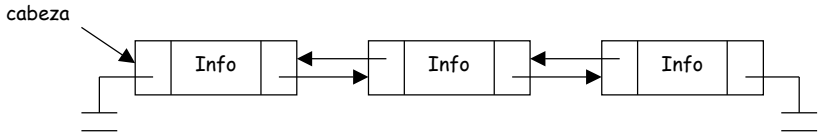
(f) Utilice la Lista anterior para implementar una Pila de elementos. Recuerde la implementación que se hizo para los arreglos y podrá hacer el símil con las listas enlazadas.

Solución

```
class Pila {
	private Lista pila;

	public Pila() {		// Ya no es necesario ponerle tope.
		pila = new Lista();
	}

	public void poner(Object o) {
		Nodo p = new Nodo(o);
		pila.poner(Nodo, pila.largo);
	}

	public Object sacar() {
		Nodo p = pila.sacar(pila.largo-1);
		return p.info;
	}
}
```

(g) **Propuesto**. Haga lo mismo que en (b) para una Cola.

(h) En el caso de las listas doblemente enlazadas, se pueden diferenciar 2 nodos apuntados: el siguiente y el anterior, sin embargo, el anterior tiene como siguiente el nodo que lo apunta (es decir, se puede recorrer tanto de ida como de vuelta).



Como podemos ver en la gráfica, estas listas quedan definidas por un campo adicional en la estructura. A modo de ejercicio propuesto será interesante que implementen estas listas creando una estructura **Nodo** apropiada y una estructura **Lista** que lo maneje con la siguiente estructura:

Método	Descripción
Lista()	Crea una lista vacía
void poner(Nodo o, int i)	Pone en la posición i-ésima (como en un arreglo)
Nodo sacar(int i)	Saca el i-ésimo elemento (como en un arreglo)
Nodo anterior (Nodo x)	Devuelve un puntero al anterior del nodo x
Nodo siguiente (Nodo x)	Devuelve un puntero al siguiente del nodo x

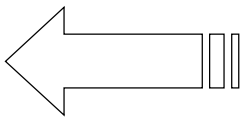
En los casos de **anterior(Nodo x)** y **siguiente(Nodo x)** debe pensar que:

- Debe buscar el nodo x.
- Si no encuentra el nodo x, debe retornar NULL.

- Si encuentra el nodo x, debe retornar su siguiente o anterior (según corresponda).

(i) El problema básico es insertar los nodos en el árbol para así ir construyendo esta estructura. La regla es una regla matemática que nos permitirá diferenciar cuáles son los dígitos de la representación binaria del número (base 2):

```
 0 = 0
 1 = 1
 2 = 10
 3 = 11
 4 = 100
 5 = 101
 6 = 110
 7 = 111
 8 = 1000
 9 = 1001
10 = 1010
11 = 1011
12 = 1100
13 = 1101
14 = 1110
15 = 1111
16 = 10000
etc.
```



Se hace en base a potencias de 2:

El n-ésimo término se aplica si el número se puede descomponer como 2^n + otros términos.

Para ello, sabemos que, para convertir a binario un número **N**, debemos descomponerlo en una sumatoria:

$$N = a_n * 2^n + \ldots + a_2 * 2^2 + a_1 * 2 + a_0 = \pi\ a_i * 2^i$$

$$\text{Con } a_i = 1,\ 0$$

Ahora, ¿cómo se construiría el número binario?

La representación binaria es la serie de coeficientes de la sumatoria uno tras otro (por sus valores que son 1 y 0, es decir, valores **binarios**).

Analizando un poco llegaremos a que la conversión se transforma al siguiente algoritmo:

```
int NUM = Número que buscamos descomponer;
String binario;

// Buscamos el mayor índice que puede servir
int n = 0;
while (NUM > Math.pow(2, n)) n++;

// Ponemos el primer valor
binario = "1";
NUM = NUM - Math.pow(2, n);

// Iteramos desde n hasta 0 para ir encontrando los valores de
N
for (int i = n - 1; i >= 0; i--) {
```

```
        if (NUM >= Math.pow(2, i)) {
                binario = binario + "1";
                NUM = NUM - Math.pow(2, i);
        }
        else {
                binario = binario + "0";
        }
}
```

Ahora, deberemos hacer el procedimiento que inserta valores en un árbol que cumple con esta características: Los de la izquierda poseen 1 y los de la derecha 0. Suponemos que existe la **raiz** del árbol en la variable de instancia (this.raiz):

```
private String insertar (int numero) {
        // Retorna la representación binaria
        // Para recorrer el árbol
        String q = this.raiz;

        // Construimos su representación binaria
        int num = numero;
        String binario;

        int n = 0;
        while (num > Math.pow(2, n)) n++;

        // Ponemos el primer valor
        binario = "1";
        num = num - Math.pow(2, n);

        // Iteramos para ir encontrando los valores de N
        for (int i = n - 1; i >= 0; i--) {
                if (NUM >= Math.pow(2, i)) {
                        binario = binario + "1";
                        NUM = NUM - Math.pow(2, i);
                }
                else {
                        binario = binario + "0";
                }
        }

        // En orden inverso, comenzamos a recorrer el árbol
        // hasta encontrar la hoja en donde insertaremos el
nodo.
        Nodo p = new Nodo(numero);

        for (int j = binario.length(); j > 0; j--) {
                if (binario.charAt(j) == "1")
                        q = q.izq;
                else
                        q = q.der;
        }
        if (binario.charAt(0) == "1")
                q.izq = p;
        else
                q.der = p;

        return binario;
}
```


Como caso base, el nodo RAIZ no debe tener valor (suponemos –1). Pero ¿para qué sirve este método?. Este método es útil al momento de llenar el árbol con número enteros:

```
public void llenar(int maximo) {
        this.raiz = new Nodo(-1);

        for (int n = 0; n <= maximo; n++)
                String x = insertar(n);
}
```

Con esto, llenamos un árbol binario con la representación binaria de un número **maximo** de números enteros. Hagamos un método que consulte ahora la representación binaria de un número utilizando el árbol. Este método **consultar(n)** será semi-recursivo:

```
public String consultar(int n) {
        String binario = null;

        // Buscamos a ambos lados y solo guardamos
        // cuando lo encontremos
        binario = "1" + consultar(n, this.raiz.izq);
        if (binario == null)
        binario = "0" + consultar(n, this.raiz.der);

        // Borramos los 0 a la izquierda, porque no valen nada
        while (binario.charAt(0) == 0)
                binario = binario.substring(1);

        return binario;
}

// El método que si es recursivo busca el valor n a
// partir desde el nodo p.
private String consultar(int n, Nodo p) {
        // Si no está
        if (p == null) return null

        // Si lo encontramos
        if (p.info = n) return "";

        // Buscamos al lado correcto
        String pre, bin;
        if (n > p.info) {
                pre = "1";
                bin = consultar(n, p.izq);
        }
        else {
                pre = "0";
                bin = consultar(n, p.der);
        }
        if (bin = null) return null;

        // Vamos componiendo paso a paso
        return pre + bin;
}
```

Listo. Es un poquito dificil de ver, pero solo importa analizar la utilización del árbol binario para almacenar este tipo de estructuras.

(j) **Propuesto.** Construir una clase **ArbolDeBinarios** que cumpla con las funcionalidades:

- **void llenar(int n)** el árbol.
- **String buscar(int n)** una representación dentro del árbol.
- **ArbolDeBinarios()** construye un árbol vacío (constructor)

También debe escribir la implementación de **Nodo** que soporte esta estructura. Sea riguroso.

Capítulo XIX: Archivos de Acceso Aleatorio

Motivación

Hasta ahora hemos visto que existen formas para acceder archivos en Java. Estos archivos por lo general son archivos de textos que podemos escribir en el bloc de notas o en otro editor de textos cualquiera.

La desventaja de estos archivos además que solo son archivos de strings sin ningún formato, es que por lo general ocurre es la secuencialidad de la lectura de ellos, es decir, al momento de leer una línea no se puede volver atrás.

Para ello existen unos tipos de archivos "binarios" que poseen las características:

- Guardar texto con formato
- Guardar datos binarios como imágenes, música, etc.
- Avanzar y retroceder dentro del archivo a discreción.

En esta clase veremos lo útil que puede ser aplicar estos archivos en algunos problemas [23].

Concepto

Byte

Un byte es un número entero entre 0 y 255 con el cuál se representan distintos tipos de información.

Un **byte** por lo general se asocia mucho a los siguiente tipos de elementos:

- Un número entre 0 y 255 (aunque parezca obvio, no lo es tanto)
- Un carácter (letra o símbolo de la tabla ASCII)
- Una tonalidad de gris
- Un número en el rango [-128, 127] (es desplazar el intervalo 0 y 255 en –128)

o cualquier pieza de información que posea a lo más 256 estados diferentes.

Es interesante notar que mientras más bytes se ocupen para representar cierta información, más detalle se puede obtener de ésta. Por ejemplo, las tarjetas de video monocromática usa 1 byte para representar 8 puntos en pantalla; la tarjeta EGA usaba en algunos caso 1 byte para representar 2 puntos en 16 colores; la tarjeta VGA usaba 1 byte por punto, pues podía tener

[23] Desde que las Bases de Datos se han convertido accesibles, los Archivos de Acceso Aleatorio han perdido parte de su aplicabilidad. Pero nunca está demás ver brevemente como utilizarlos en problemas más sencillos y de poca envergadura de datos.

hasta 256 colores; en cambio las tarjetas de video actuales usan más de un byte para el millones de colores que comparte cada punto.

El byte es en el fondo la componente principal en computación. Los tipos primitivos de Java que hemos usado también se representan en bytes:

- **bolean** usa 1 byte
- **int** usa 4 bytes
- **double** usa 8 bytes
- etc...

Archivo de Acceso Aleatorio

Estructura de Dato predefinida (modelada en una clase Java) similar a un arreglo que se almacena en disco. Los elementos de este arreglo son bytes.

En este caso, quedaría definido entonces como un arreglo de bytes o letras escritas en disco. Pero ¿qué diferencia hay entre un archivo de texto plano (BufferedReader) con este tipo de elementos?

Bueno, la diferencia en que un Archivo de Acceso Aleatorio posee algunas características propias como:

- No solo almacenar letras
- Solo es necesario abrir una vez el archivo para recorrerlo completamente
- No es necesario leerlo en forma secuencial
- También se puede utilizar un mismo archivo para leer, escribir y actualizar datos.

RandomAccessFile

Es la clase Java que define un Archivo de Acceso Aleatorio.

RandomAccessFile (RAF) es la clase Java que representa uno de nuestros archivos. Pero ¿qué tiene de especial? ¿cómo se usa?. Fijemos nuestros esfuerzos en responder estas preguntas.

Sintaxis

Para crear un RAF se debe usar la siguiente sintaxis:

```
RandomAccessFile r = new RandomAccessFile("<archivo>", "<ops>");
```

En donde:

- <archivo>: nombre del archivo que se desea utilizar.
- <ops>: opciones del archivo como son lectura ("r") y/o escritura ("w");

Ejemplos:

```
// Abrir archivo datos.txt para lectura
RandomAccessFile r = RandomAccessFile("datos.txt", "r");

// Abrir archivo datos.txt para escritura
RandomAccessFile r = RandomAccessFile("datos.txt", "w");

// Abrir archivo datos.txt para lectura/escritura
RandomAccessFile r = RandomAccessFile("datos.txt", "rw");
```

Otro punto importante es saber el largo de un RAF o lo mismo es saber ¿dónde termina el archivo?. Pues para eso se puede obtener con **r.length()**, método que retornará la cantidad de bytes que posee el RAF en cuestión.

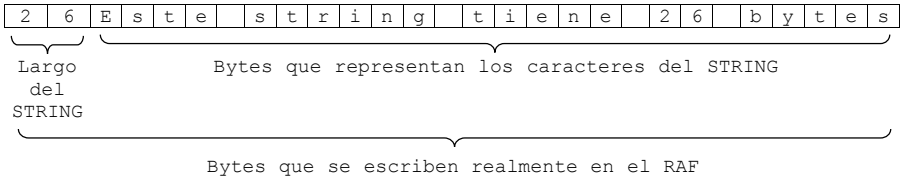
Veamos algunos patrones de programación aplicados a RAF.

Escritura de un RAF

Para escribir en un RAF se utilizan los siguientes métodos:

- **writeInt(x)** escribe en el RAF el entero x (4 bytes).
- **writeDouble(x)** escribe en el RAF el double x (8 bytes).
- **writeUTF(x)** escribe en el RAF el string x (n+2 bytes).

Como podemos ver, se indica la cantidad de bytes que utiliza escribir estos valores en un RAF. En el caso de los Strings, la cantidad de bytes es siempre igual al largo del string (1 byte por carácter) más 2 bytes al inicio de la cadena que indica la cantidad de bytes que corresponden al String (nuevamente el largo). Algo como:



Por lo tanto calcular la cantidad de caracteres que se deben leer debe ser cuidadosamente considerado con los 2 bytes adicionales de cada String.

Lectura de un RAF

Para leer en un RAF se utilizan los siguientes métodos:

- **readInt()** lee un entero desde el RAF (4 bytes).
- **readDouble(x)** lee un double desde el RAF (8 bytes).
- **readUTF(x)** lee un string desde el RAF (n+2 bytes).

Moverse dentro de un RAF

Una de las ventajas que poseen los archivos binarios es su capacidad de mover el cabezal de lectura/escritura hacia delante o atrás a gusto.

El método que permite esto es **seek(n)** en donde se le especifica como parámetro el número de byte al cual se desea desplazar, es decir, como el índice dentro de este gran arreglo de bytes.

Como los RAF puedes ser muy grandes, un entero no basta para indicar la posición, por lo que la firma del método seek es:

```
public void seek(long n);
```

Es por eso que antes de hacer un seek, se debe realizar un cast explícito para indicar que estamos en presencia de un long como:

```
r.seek( (long) (3*4) );   // 3 enteros del comienzo (12 bytes)
```

Aunque hacer un seek es una ventaja sobre los archivos de texto, su utilización debe ser con "mucho cuidado" pues es demasiado costoso para el procesador. Como comparación, un seek cuesta lo mismo que leer 20.000 caracteres desde un archivo de texto plano.

También, es útil saber el lugar en el que se encuentra detenido un RAF. Para ello se usa el método **getFilePointer()** que entrega el # de byte en donde está parada la cabeza lectora.

Ejemplo Práctico

Pero ya basta de cháchara y veamos como usamos esto en la práctica. Supongamos que queremos almacenar las notas de 100 alumnos en un archivo de texto plano:

```
BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
PrintWriter out = new PrintWriter(
      new FileWriter("notas.txt"));
for (int n=0; n<100; n++) {
      System.out.print("Nombre? ");
      String nombre = in.readLine();
      System.out.print("Nota? ");
      double nota = new Double(
            in.readLine()).doubleValue();
      out.println(n + " " + nombre + " " + nota);
}
out.close();
in.close();
```

y si quisiéramos buscar el alumno 53 deberíamos hacer lo siguiente:

```
BufferedReader in = new BufferedReader(
      new FileReader("notas.txt"));
for (int n=0; n<53; n++) {
      String alumno = in.readLine();
```



```
}
in.close();
System.out.println(alumno);
```

Claramente esto se convierte en algo costoso, pero que con RAF es algo más sencillo. Primero que todo si queremos almacenar todos esos datos, debemos saber cuántos bytes usaremos. En general si los nombres TODOS miden 30 caracteres, suponemos que el registro completo mide 4 bytes (entero) + 8 bytes (real) + 30 bytes (String) + 2 bytes (largo del String) = 44 bytes.

```
BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
RandomAccessFile out = new RandomAccessFile("notas.txt", "w");
for (int n=0; n<100; n++) {
      out.writeInt(n);
      System.out.print("Nombre? ");
      out.writeUTF(in.readLine());
      System.out.print("Nota? ");
      out.writeDouble(new Double(in.readLine()).doubleValue());
}
out.close();
in.close();
```

Pero además la búsqueda del alumno 53 sería:

```
RandomAccessFile arch = new RandomAccessFile("notas.txt", "rw");
arch.seek(44 * 53);
String alumno = arch.readInt() + " " +
                arch.readUTF() + " " +
                arch.readDouble();
System.out.println(alumno);
arch.close();
```

Es una manera distinta de ver el mundo, ¿no?

Lectura y Escritura de Strings

Uno de los problemas más graves es que los strings no son todos iguales, lo que nos impide saber la cantidad de bytes reales que tiene el archivo.

Los programadores astutamente por supuesto se dieron cuenta que podría ser un problema minimizado si ellos cargaran con el peso de crear estos UTF de manera que para un archivo siempre los strings tuvieran el mismo largo. Es por eso que antes de poder escribir un RAF es necesario escribir un método que uniforme nuestros strings de un largo fijo. ¿Cómo?. Poniéndole espacios por supuesto, es decir:

```
public String convertUTF (String txt, int largo) {
      String s = txt;

      // Trunca el string si es más grande
      if (s.length() > largo)
            return s.substring(0, largo);

      // Rellena con espacios si le faltan
      for (int n=s.length(); n<largo; n++)
            s = s + " ";
```

```
        return s;
}
```

Con este sencillo método convertimos los strings a un largo definido. Por lo tanto, antes de escribir un string, haremos:

```
arch.writeUTF(convertUTF(s, 30)); // Uniforma s a 30 bytes
```

Problema

Veamos un problema sencillo primero. Se desea modelar un sistema de acceso con claves y permisos de usuario a través de archivos binarios (por seguridad). Para ello se sabe que cada usuario posee los siguientes campos (variables de instancia):

- Nombre de Usuario (8 caracteres)
- Nombre Completo (40 caracteres)
- Clave de Acceso (11 caracteres)
- Permiso de Lectura (1 = SI / 2 = NO)
- Permiso de Escritura (1 = SI / 2 = NO)

Para ello se le pide que implemente el tipo de dato que permita modelar esto. La idea que debe declarar:

(a) Variables de Instancia
(b) Constructor que permita el ingreso de los datos (TODOS)
(c) Método que permita escribir en un RAF el registro (debe recibir el nombre del archivo y escribirlo al final de éste).

Solución

```
public class Usuario {
      // Largo del registro
      private final int largoRegistro = (8 + 2) + (40 + 2) +
                                        (11 + 2) + 4 + 4;

      // Variables de Instancia
      public String username, password;
      public String nombre;
      public int lectura, escritura;

      // Constructor
      public Usuario (String p1, String p2, String p3,
                      int p4, int p5) {
            this.username = p1;
            this.password = p3;
            this.nombre = p2;
            this.lectura = p4;
            this.escritura = p5;
      }

      // UTF para String
      private String convertUTF (String txt, int largo) {
            String s = txt;
            // Trunca el string si es más grande
            if (s.length() > largo)
                  return s.substring(0, largo);
            // Rellena con espacios si le faltan
            for (int n=s.length(); n<largo; n++)
                  s = s + " ";
            return s;
      }

      // Guarda en RAF
      public void writeToRAF (String filename) {
            // Abrir el archivo
            RandomAccessFile a = new RandomAccessFile(
                  filename, "rw");

            // Ir al final del mismo
            a.seek(a.length());

            // Escribir registro
            a.writeUTF(convertUTF(this.username, 8));
            a.writeUTF(convertUTF(this.nombre, 40));
            a.writeUTF(convertUTF(this.password, 11));
            a.writeInt(this.lectura);
            a.writeInt(this.escritura);

            // Cerrar el archivo
            a.close();
      }
}
```


Problema Propuesto

Este es realmente un reto. Suponga que tiene un RAF con todos los RUT de los alumnos del curso con el formato de un DOUBLE, es decir el número sin puntos y sin el dígito verificador. Por ejemplo:

```
13252311
4993023
...
```

El RAF se llama "RUTS.BIN" y se quiere buscar un RUT en particular.

(a) Modifique un algoritmo de ordenamiento para que funcione ordenar en un RAF. Para ello recuerde que si se para en una posición y hace un **write**, éste sobrescribe el valor que ha en esa posición (en bytes), por ejemplo:

```
arch.seek(48);
arch.writeInt(57);   // Sobrescribe en el byte 48 el entero 57.
```

(b) Utilice búsqueda binaria (suponga que ya está ordenado el archivo) aplicada a un RAF para verificar la existencia de un RUT dentro de la lista, es decir, simule el siguiente diálogo:

```
Ingrese RUT: 12091128
No está ese RUT
Ingrese RUT: 13252311
Encontré el RUT en el byte x
```

Capítulo XX: Bases de Datos

Motivación con RAF's

Desde algún tiempo atrás hasta el día de hoy, el procesamiento de información en problemas reales ha sido una gran preocupación de los programadores.

Inicialmente, utilizaban estructuras de datos en memoria para almacenar la información: pero la memoria es poca y la información era mucha. Luego se decidieron utilizar archivos para guardar mayores volúmenes de datos: también fue insuficiente, ya que el crecimiento de los archivos de texto era demasiado y las búsquedas se hacían cada vez más lentas.

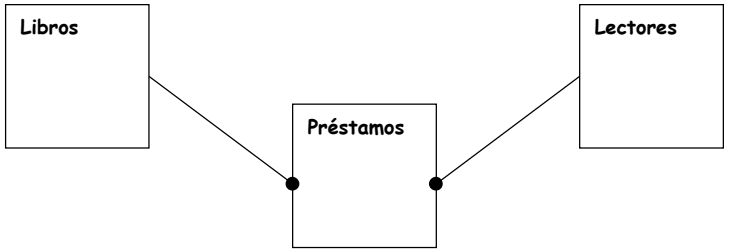
Es así como la tendencia se ha ido hacia crear medios para almacenar grandes volúmenes de datos y optimizar las consultas. A estos medios se les llamó **Bases de Datos**.

Conceptos

Base de Datos

> *Conjunto de tablas o archivos relacionados que permiten el almacenamiento estructurado y ordenado de grande volúmenes de información.*

Como se entiende en esta definición, las Bases de Datos son medios que permiten almacenar información. Esta información queda en estructuras mas simples que son llamadas **Tablas** o **Archivos**. Como ejemplo, veamos las tablas para una Base de Datos de préstamos de libros en una biblioteca:



Tablas (Archivos)

> *Secuencia de registros (líneas) que almacenan información del mismo tipo organizados en filas o tuplas de datos.*

Las tablas son las estructuras básicas para el almacenamiento de la información. Cada registro posee un grupo de características (comunes dentro de la misma tabla):

Libros

Código	Título	Autor	Editorial	Fecha
. . .				

Lectores

RUN	Nombre	Dirección	Fono	E-mail
. . .				

Préstamos

Código Libro	*RUN Lector*	Fecha Devolucíon
. . .		

Registros, Filas, Línea o Tuplas

> *Secuencia de campos o columnas que representan las características de cada elemento almacenable en la tabla.*

Podemos ver que cada registro almacena las características de un elemento de la tabla. Todos los elementos de una misma tabla poseen los mismos tipos de campos. Por ejemplo:

Un libro posee Código, Título, Autor, Editorial y Fecha.

Sintaxis

Comenzaremos definiendo una interface para los registros la cual utilizará una RAF para almacenar la tabla. Todos los registros poseerán un set de funcionalidades propias y son:

- Largo de un registro.
- Escribir registro en la tabla.
- Leer el registro desde la tabla.

Con estas 3 definiciones, escribiremos la interface **Registro** como sigue:

```
import java.io.*;

public interface Registro {
	public long largo();
	public void leer(RandomAccessFile raf) throws Exception ;
	public void escribir(RandomAccessFile raf) throws Exception;
}
```

Nótese que utiliza un RandomAccessFile para escribir y leer. Esto se debe realizar, porque solo es el registro quién conoce la definición de los tipos de datos que guarda. La tabla solo sabe que posee registros.

Por ejemplo si queremos hacer un registro de alumnos con la tupla (RUT, Nombre, y Correo), deberemos implementar la interface Registro según la siguiente definición:

```
import java.io.*;

public class Alumno implements Registro {
	// Definimos la estructura del registro
	public long rut;                // 8 bytes
	public String nombre;           // 100 bytes (+2)
	public String correo;           // 50 bytes  (+2)

	// La constante que nos indica de qué tamaño es el registro
	private final int largo = 162;

	public Alumno() { /* Constructor vacío */ }

	public long largo() {
		return this.largo;
	}

	public void leer(RandomAccessFile raf) throws Exception {
		this.rut = raf.readLong();
		this.nombre = raf.readUTF();
		this.correo = raf.readUTF();
	}

	public void escribir(RandomAccessFile raf) throws Exception {
		raf.writeLong(this.rut);
		raf.writeUTF(this.uniformar(this.nombre, 100));
		raf.writeUTF(this.uniformar(this.correo, 50));
	}

	// Recordemos que los String se guardan uniformados
	private String uniformar(String s, int l) {
		String u = "";
		for (int i=0; i<l; i++) {
			if (i < s.length()) { u += s.charAt(i); }
			else { u += " "; }
		}
		return u;
	}
}
```


Con la definición del registro (en su forma genérica) podríamos definir nuestra tabla de datos, a la cual llamaremos **TablaRAF** como un archivo de registros. Como la idea es almacenarlos en un RAF, deberemos manejar las excepeciones de los archivos. Aquí se utiliza el **try ... catch** para manejar las excepciones.

```
import java.io.*;

public class TablaRAF {
	private RandomAccessFile raf;

	public TablaRAF() {
	}

	public TablaRAF(String nombre) {
		this.abrir(nombre);
	}

	public void abrir(String nombre) {
		try {
			raf = new RandomAccessFile(nombre, "rw");
			raf.seek(0);
		}
		catch (Exception e) {
			System.err.println("ERROR: al abrir!");
		}
	}

	public void insertar(Registro r) {
		try {
			raf.seek(raf.length());
			r.escribir(raf);
		}
		catch (Exception e) {
			System.err.println("ERROR: al insertar!");
		}
	}

	public void actualizar(int n, Registro r) {
		try {
			raf.seek((n-1) * r.largo());
			r.escribir(raf);
		}
		catch (Exception e) {
			System.err.println("ERROR: al actualizar ");
		}
	}

	public void eliminar(int n) {
		// Ooops... no sabemos eliminar aún
		System.err.println("ERROR: al eliminar!");
	}

	public void ver(int n, Registro r) {
		try {
			raf.seek((n-1) * r.largo());
			r.leer(raf);
		}
		catch (Exception e) {
			System.err.println("ERROR: al ver!");
		}
	}
```

```
	public boolean finDatos() {
		boolean r = false;
		try {
			r = (raf.getFilePointer() >= raf.length());
		}
		catch (Exception e) {
			System.err.println("ERROR: en la tabla!");
		}
		return r;
	}

	public void cerrar() {
		try {
			raf.close();
		}
		catch (Exception e) {
			System.err.println("ERROR: al cerrar!");
		}
	}
}
```

(**Ojo:** que la mayor parte de los métodos son de una o dos líneas, pero utilizan **try…catch** para evitar las excepciones[24]).

Por ejemplo, si tenemos un registro de **Curso**, nos gustaría obtener su columna código:

```
...
TablaRAF t = new TablaRAF("cursos");
int i=0;
while (!t.finDatos()) {
	// Creamos un curso vacío
	Curso c = new Curso();

	// Le metemos los datos desde la tabla
	t.ver(i, c);

	// Mostramos en pantalla el código
	System.out.println(c.codigo);
}
t.close()
...
```

Veamos ahora las operaciones que se pueden hacer con esta representación.

Existen 3 tipos de operaciones que se pueden realizar entre las tablas de una base de datos: **Selección**, **Proyección** y **Pareo/Cruce**.

Selección

Recupera las filas de una tabla según algún criterio de búsqueda que utilice los campos de la tabla.

[24] Existe un capítulo no obligatorio con este material de excepciones. Revísalo.

Para realizar una selección, es necesario primero saber bajo qué criterio. Por ejemplo, de la tabla **Libros** vista anteriormente, nos gustaría encontrar aquellos libros que sean de Java (que contengan en su nombre en alguna parte esa palabra)[25]:

```
TablaRAF libros = new TablaRAF();
libros.abrir("libros");
Libro[] resultados = new Libro[MAXDATA];
int n = 0;
for (int i=0; !libros.finDatos(); i++) {
	Libro r = new Libro();
	libros.ver(i, r);
	if (libros.titulo.indexOf("Java") >= 0) {
		libro[n] = r;
		n++;
	}
}
libros.cerrar();
```

Con este código, en el arreglo **resultados** tenemos todos los registros que contienen la palabra "Java" en el campo de **Título**.

Como se puede observar en el ejemplo, las tablas también se pueden representar como arreglos de registros en la memoria del computador. Esto hace que cada uno de los resultados obtenidos en una selección también sea una tabla. Entonces si continuamos el programa anterior:

```
// ... continuación
TablaRAF librosJava = new TablaRAF();
librosJava.abrir("LibrosJAVA");
for (int i=0; i<n; i++) {
	librosJava.insertar(libro[i]);
}
librosJava.cerrar();
```

Hemos creado una tabla **LibrosJava** con todos aquellos libros que en su título poseen la palabra Java.

Pero la selección no es condicionada solo por un campo. También se puede realizar selección según 2 o más criterios. Por ejemplo, si los libros son de **Isaac Asimov** y además la editorial se **Universitaria**:

```
TablaRAF libros = new TablaRAF();
libros.abrir("Libros");
Libro[] resultados = new Libro[MAXDATA];
int n = 0;
for (int i=0; !libros.finDatos(); i++) {
	Libro reg = new Libro();
	libros.leerRegistro(i, reg);
	if (reg.autor.equals("Isaac Asimov") &&
	    reg.editorial.equals("Universitaria")) {
		resultados[n] = reg[i];
		n++;
```

[25] Suponemos que MAXDATA es un número suficientemente grande.

```
        }
}
libros.cerrar();
```

Como ven, solo nos basta cambiar las condiciones de comparación y está listo.

Proyección

Recupera solo algunas columnas de una tabla, es decir, solo selecciona algunos datos a desplegar.

La diferencia que hay entre una **Selección** y una **Proyección** es que en el segundo caso se seleccionan columnas específicas de la tabla, cosa que no es así en la primera.

Queremos seleccionar solo los RUN's de los lectores que vivan en la comuna de Santiago:

```
TablaRAF t = new TablaRAF();
t.abrir("Lectores");
Lector[] resultados = new Lector[MAXDATA];
int n = 0;
for (int i=0; !t.finDatos(); i++) {
       Lector reg = new Lector();
       t.leerRegistro(i, reg);
       if (reg.direccion.indexOf("Santiago") >= 0) {
              resultados[n].run = reg.run;
              n++;
       }
}
t.cerrar();
```

En este caso, solo estamos guardando los RUN. Veamos si queremos también los nombres:

```
TablaRAF t = new TablaRAF();
t.abrir("Lectores");
Lector[] resultados = new Lector[MAXDATA];
int n = 0;
for (int i=0; !t.finDatos(); i++) {
       Lector reg = new Lector();
       t.leerRegistro(i, reg);
       if (reg.direccion.indexOf("Santiago") >= 0) {
              resultados[n].run = reg.run;
              resultados[n].nombre = reg.nombre;
              n++;
       }
}
t.cerrar();
```

Y si quisiermos los nombres y los correos electrónicos:

```
TablaRAF t = new TablaRAF();
t.abrir("Lectores");
Lector[] resultados = new Lector[MAXDATA];
int n = 0;
for (int i=0; !t.finDatos(); i++) {
       Lector reg = new Lector();
       t.leerRegistro(i, reg);
```



```
       if (reg.direccion.indexOf("Santiago") >= 0) {
              resultados[n].nombre = reg.nombre;
              resultados[n].correo = reg.correo;
              n++;
       }
}
t.cerrar();
```

Como pueden ver, la diferencia entre la selección y proyección es mínima, y se puede llegar a que una selección es un una proyección de todos los campos de la tabla.

Pareo/Cruce (Join)

El Join se utiliza para recuperar información de varias tablas, a través de una función entre ellas.

El join es similar a realizar un **merge**: se van mezclando los datos de una tabla con los de la otra por un criterio definido. En el caso del mergesort se realizaba según un orden (ordenamiento). Sin embargo, en el caso de las bases de datos, este pareo utiliza un criterio de comparación entre las columnas de una tabla y otra.

Por ejemplo: ¿Cómo saber los nombres de los lectores que poseen libros prestados? Para ello se utilizan las tablas **Lectores** y **Préstamos**, una clase llamada **Deudor** para almacenar los resultados, que solo posee un String y suponiendo que ambas tablas están ordenadas por RUN del lector:

```
TablaRAF t1 = new TablaRAF();
t1.abrir("Lectores");
TablaRAF t2 = new TablaRAF();
t2.abrir("Prestamos");
Deudor[] resultados = new Deudor[MAXDATA]
int n = 0;
for (int i=0; !t2.finDatos(); i++) {
       Prestamo regPrestamo = new Prestamo();
       t2.leerRegistro(i, regPrestamo);
       for (int j=0; !t1.finDatos(); j++) {
              Lector regLector = new Lector();
              regLector = t2.leerRegistro(j);
              if (regLector.RUN.equals(regPrestamo.RUN)) {
                     resultados[n].nombre = regLector.nombre;
                     n++;
                     break;
              }
       }
}
t.cerrar();
```

En **resultados** se encuentran todos los nombres que poseen libros prestados, pero repetidos en caso de que tenga más de un libros prestado.

Problemas

Una concesionaria de automóviles quiere hacer un sistema para almacenar su información de venta y así pagar las comisiones a sus empleados. Para ello diseñaron 2 tablas relacionadas que modelan el sistema de almacenamiento de datos:

Vendedor
- RUT
- Nombre
- Sueldo (Base)

Automovil
- Código
- Marca
- Modelo
- Precio
- Comisión (fracción de comisión)
- Fecha (de Venta)
- RUN (vendedor)

(a) Implemente un programa que permita obtener el sueldo del vendedor Juan Pérez para el mes de septiembre (considere que la fecha de venta viene en formato dd/mm/yyyy como String).

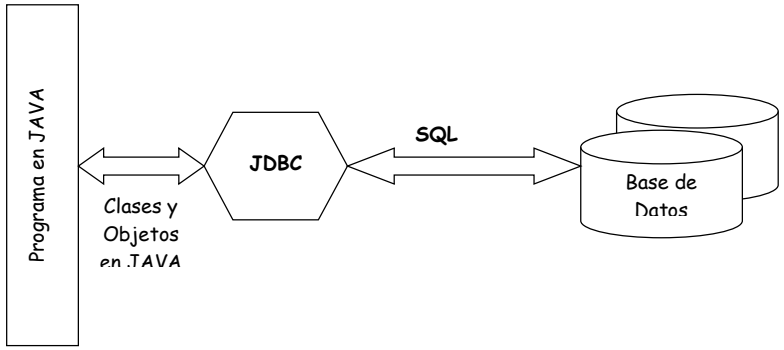
```
// Abrimos las tablas para el cruce
TablaRAF vend = new TablaRAF("Vendedor");
TablaRAF auto = new TablaRAF("Automovil");

// Declaramos el resultado. Como es solo 1...
Vendedor vendedor = new Vendedor();

// Ciclo de lectura de vendedores
for (int i=0; !vend.finDatos(); i++) {
       // Buscamos al vendedor
       vend.ver(i, vendedor);
       if (vendedor.nombre.equals("Juan Pérez")) {
              // Ciclo de lectura de automoviles
              for (int j=0; !auto.finDatos(); j++) {
                     // Calculamos el sueldo
                     Automovil vendido = new Automovil();
                     auto.ver(j, vendido);
                     if (vendido.RUN == vendedor.RUN &&
                         vendido.fecha.indexOf("09") == 3) {
                            // Aumentamos la comisión al sueldo
                            vendedor.sueldo += vendido.precio *
                                               vendido.comision;
                     }
              }
       }
}
auto.cerrar();
vend.cerrar();
```


(b) Modifique su programa anterior para que almacene en una tabla llamada **Sueldo** el RUT, el nombre y el sueldo (incluyendo comisión) para el mes de septiembre.

```
// Abrimos las tablas para el cruce
TablaRAF vend   = new TablaRAF("Vendedor");
TablaRAF auto   = new TablaRAF("Automovil");
TablaRAF sueldo = new TablaRAF("Sueldo");

// Declaramos el resultado. Ahora no es 1
Vendedor[] vendedor = new Vendedor[MAXDATA];

// Ciclo de lectura de vendedores
for (int i=0; !vend.finDatos(); i++) {
       vendedor[i] = new Vendedor();
       vend.ver(i, vendedor);

       // Ciclo de lectura de automoviles
       for (int j=0; !auto.finDatos(); j++) {
              // Calculamos el sueldo del vendedor i
              Automovil vendido = new Automovil();
              auto.ver(j, vendido);
              if (vendido.RUN == vendedor[i].RUN &&
                  vendido.fecha.indexOf("09") == 3) {
                     // Aumentamos la comisión al sueldo
                     vendedor[i].sueldo += vendido.precio *
                                           vendido.comision;
              }
       }

       // Lo dejamos en la tabla Sueldo
       sueldo.insertar(vendedor[i]);
}
sueldo.cerrar();
auto.cerrar();
vend.cerrar();
```

Motivación con SQL

Para realizar una comunicación entre Java y Bases de Datos de verdad (SQL*Server, mySQL, Oracle, Sybase, DB2, etc), se necesita entender previamente lo que es llamado **Standard Query Language (SQL)**.



Conceptos

Base de Datos Relacional

Una Base de Datos relacional es un espacio de almacenamiento de grandes volúmenes de datos, organizados en tablas, y que permite especificar relaciones entre aquellas tablas.

Motor de Base de Datos Relacional[26]

El Motor de una Base de Datos Relacional (RDBMS o Sistema Administrador de Bases de Datos Relacionales) es un software que permite administrar un conjunto de Bases de Datos Relacionales.

Existen un sin número de Bases de Datos relaciones. Por ejemplo, **Oracle** es una empresa que desarrolló un gran software de bases de datos relacionales. En la actualidad, Oracle, tiene en el mercado la versión 8 de su motor de datos (del mismo nombre) y está a punto de sacar al mundo la versión 9 mejorada.

SQL

Lenguaje estandarizado que utilizan los RDBMS para interpretar los requerimientos que los usuarios hacen a sus modelos de datos.

[26] Si te atreves, un motor liviano e instalable en Windows (y en Linux también) de un RDBMS puedes encontrarlo en http://www.mysql.com/downloads/mysql-3.23.html (**MySQL**).

Standard Query Language es un lenguaje que permite a los programadores realizar consultas a las Bases de Datos que desean ocupar. Su sintaxis se basa en un set de pocos comandos o instrucciones y en combinaciones de ellas.

Las consultas son enviadas al RDBMS a través de un texto, el cual es interpretado por él y ejecutado sobre el modelo en el cual se está conectado. Una vez que esa consulta es procesada, el RDBMS retorna los datos como si fuesen una nueva tabla.

Los distintos RDBMS tienen interfaces de consulta que permiten a los usuarios escribir directamente sus consultas en SQL y ver los resultados en pantalla. Por ejemplo, Oracle posee una aplicación llamada SQL Plus (en todas sus versionas ha mantenido este programa) y que posee un diálogo tipo:

```
> SELECT * FROM usuarios
NOMBRE               CLAVE
-------------------- ----------------------
averuzzi             kifd99.
djacintos            a99sjjs
tchan                karina1000

3 rows selected
> _
```

Este sencillo ejemplo, lo que hace es una selección desde la tabla **usuarios**, la cual solo posee 3 datos.

JDBC

JDBC (JAVA Database Connection) es un set de herramientas que permiten conectarse a un RDBMS y conversar con él para que las consultas salgan de instrucciones JAVA y los resultados procesarlos en una clase JAVA.

Para realizar una conexión con JAVA y un RDBMS se utiliza un **driver** el cual permite hacer una conexión con ese tipo de RDBMS. Por ejemplo, si queremos conectarnos con SQL*Server, utilizaremos una clase que se baja de la página de Microsoft y se llama com.ms.JDBCDriver.

Sin embargo, antes de comenzar a conectarnos entre una Base de Datos y JAVA es conveniente saber cómo funciona SQL y su sintaxis.

Sintaxis[27]

SQL posee un conjunto de instrucciones categorizado en los siguientes tipos:

- **Consulta de Datos**
- **Actualización de Datos**
- **Administración de Base de Datos**

[27] Si desean mayor información, visiten http://www.desarrolloweb.com/manuales/9/

Consulta de Datos

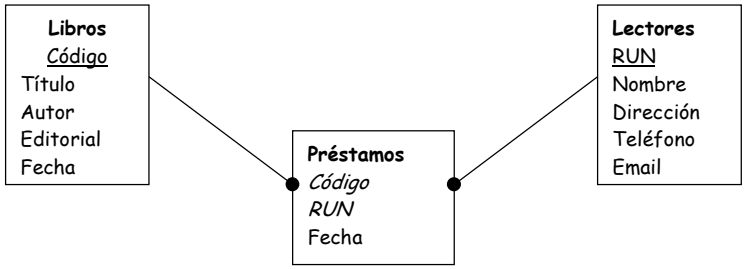
Para realizar una consulta en SQL se necesita saber la sintaxis básica de un comando de este tipo. Para ello existe solo una instrucción que es llamada **SELECT**:

```
SELECT columna1, columna2, ..., columnaN
  FROM tabla1, tabla2, ..., tablaM
 WHERE condicion1
AND/OR condicion2
AND/OR ...
AND/OR condicionO
ORDER BY criterio1, criterio2, ..., criterioP
```

En donde (Las líneas en negritas son obligatorias):

columna1, ... columnaN: Son las columnas a desplegar como resultado
tabla1, ..., tablaM: Son los nombres de las tablas que se desea consultar (selección/proyección).
condicion1, ..., condicionO: Son condiciones que se deben cumplir (join).
criterio1, ..., criterioP: Son las columnas por los cuales será ordenado el resultado.

Por ejemplo, recordando la estructura de la Base de Datos de unas clases atrás, tenemos:



En este modelo podemos ver que la tabla **Libros** posee un identificador único[28] que es **Código** y que es relacionado con otro identificador único de la tabla **Lectores** que es **RUN**, ambos en la tabla **Préstamos**.

Veamos un set de consultas y su traducción en SQL:

```
SELECT *
  FROM lectores
```

Obtiene el detalle de todos los lectores que existen.

```
SELECT *
  FROM libros
 WHERE editorial = 'Universitaria'
```

Obtiene todos los libros que sean de la editorial Universitaria.

```
SELECT run, nombre
  FROM lectores
```

Obtiene todos los nombres y runs de todos los lectores del sistema.

```
SELECT lectores.nombre,
       libros.titulo
  FROM lectores, prestamos, libros
 WHERE libros.codigo = prestamos.codigo
   AND prestamos.run = lectores.run
```

Obtiene los nombres de los lectores y los títulos de los libros que tienen en su poder.

```
SELECT *
  FROM libros, prestamos
 WHERE libros.codigo = prestamos.codigo
   AND libros.nombre = 'JAVA'
 ORDER BY prestamos.fecha
```

Obtiene todos los detalles de los libros de JAVA que se encuentren prestados ordenados por fecha de devolución.

El primer ejemplo, en **mySQL**, da por resultado:

```
mysql> select * from lectores;
+----------+----------------+-----------+----------+-------------------------+
| run      | nombre         | direccion | telefono | email                   |
+----------+----------------+-----------+----------+-------------------------+
| 13252311 | Andres Munoz O |           |          | andmunoz@entelchile.net |
+----------+----------------+-----------+----------+-------------------------+
1 row in set (0.00 sec)

mysql>
```

(Considerando que solo existe un lector en la Base de Datos).

Como se puede apreciar, el formato de salida es igual a una tabla con las columnas especificadas. En el caso de poner un ***** (asterisco) como nombre de columna, éste nos indica que se seleccionan todas las columnas de la tabla.

Otro ejemplo en **mySQL** es con proyección:

```
mysql> select run, nombre
    -> from lectores;
+----------+----------------+
| run      | nombre         |
+----------+----------------+
| 13252311 | Andres Munoz O |
+----------+----------------+
1 row in set (0.00 sec)

mysql>
```

[28] En Base de Datos, ese identificador es llamado **Llave Primaria**.

Nos podemos dar cuenta que, al especificar las columnas, nos entrega exactamente esos datos no más.

Un último ejemplo es con un JOIN:

```
mysql> select libros.titulo, lectores.nombre
    -> from libros, prestamos, lectores
    -> where libros.codigo = prestamos.codigo
    -> and prestamos.run = lectores.run
    -> order by libros.codigo;
+--------------------+----------------+
| titulo             | nombre         |
+--------------------+----------------+
| Introducción a Java | Andres Munoz O |
+--------------------+----------------+
1 row in set (0.06 sec)

mysql>
```

Acá podemos ver las columnas de distintas tablas completamente mezcladas sin saber de qué se trata cada una.

Actualización de Datos

SQL permite además realizar actualización on-line de datos a través de 3 sentencias básicas para la administración: Inserción, Actualización y Eliminación.

INSERT: Esta cláusula permite insertar valores a una tabla específica. Su sintaxis es:

```
INSERT INTO tabla
            (columna1, columna2, ..., columnaN)
     VALUES (valor1, valor2, ..., valorN)
```

Como se ve en la sintaxis, solo es posible insertar valores SOLO a 1 tabla. Además, uno puede especificar el orden y qué columnas llenar. En caso de no especificar una columna de la tabla, esta se inserta con NULL (solo si no está definida como NOT NULL). Por último, si no se especifican las columnas (ninguna), se supone que se insertarán TODAS las columnas.

Ejemplos:

```
INSERT INTO lectores
     VALUES ('12688049-9',
             'Juan Perez González',
             'Alameda 1020',
             '223 9023',
             'jperez@hotmail.com')
```

Inserta un lector a la tabla de lectores (todas las columnas).

```
INSERT INTO libros
            (codigo, nombre, autor)
     VALUES ('333443-23',
             'La Fundación',
             'Isaac Asimov')
```

Solo inserta las columnas código, nombre y autor a la tabla libros (en ese orden).

UPDATE: Esta cláusula permite actualizar valores a una tabla específica y dependiendo de condiciones. Su sintaxis es:

```
UPDATE tabla
   SET columna1 = valor1,
       columna2 = valor2, ...
       columnaN = valorN
 WHERE condición1 AND/OR condición2 ... AND/OR condiciónM
```

La sentencia UPDATE especifica el nombre de la tabla, las columnas con los valores a actualizar y las condiciones para seleccionar solo aquellos registros que se desean actualizar. Más claro lo podemos ver en los siguientes ejemplos:

```
UPDATE lectores
   SET nombre = 'Alberto Fujimori',
          run = '9324423-8'
 WHERE email = 'chino@yahoo.es'
```

Cambia solo un registro (o todos aquellos que coincidan con el e-mail).

```
UPDATE libros
   SET editorial = 'Universitaria'
```

Cambia todos los registros de la tabla libros y les pone la editorial = UNIVERSITARIA

```
UPDATE libros
   SET editorial = 'Universitaria'
 WHERE codigo = '3849232'
```

Cambia todos los registros de la tabla libros y les pone la editorial = UNIVERSITARIA solo si el codigo es 3849232.

DELETE: Esta cláusula permite borrar registros de una tabla específica y dependiendo de condiciones. Su sintaxis es:

```
DELETE FROM tabla
      WHERE condición1 AND/OR condición2 ... AND/OR condiciónM
```

La sentencia más sencilla es esta, ya que solo se debe especificar la tabla y las condiciones para eliminar las columnas (que calcen con la cláusula WHERE). Ejemplos:

```
DELETE FROM lectores
```

Elimina todos los lectores de la Base de Datos.

```
DELETE FROM libros
      WHERE editorial = 'Alcántara'
```

Elimina todos los libros de la editorial Alcántara.

El cuidado al eliminar se debe tener en las relaciones de la Base de Datos. Por ejemplo, al eliminar con la sentencia DELETE FROM LECTORES, se eliminarán todos los lectores SOLO SI no existen registros de lectores (run's) en la tabla PRESTAMOS.

Administración de Tablas

Para administrar las tablas de una Base de Datos existen 2 comandos sencillos (pero no son los únicos) que sirven para esta labor:

CREATE TABLE: Esta sentencia permite crear tablas en una base de datos. Su sintaxis es:

```
CREATE TABLE  nombre
              (columna1 tipo1 NOT NULL,
               columna2 tipo2 NOT NULL,
               ...
               columnaN tipoN NOT NULL)
```

Esta versión bastante simplificada de la sintaxis indica que se crea una tabla con su nombre y se indican las columnas con su tipo y se indican si son o no son NOT NULL.

Veamos algunos ejemplos:

```
CREATE TABLE revistas
            (codigo varchar NOT NULL,
             nombre varchar NOT NULL,
             numero number,
             valor number)
```

Crea la tabla revistas en la Base de Datos.

Algunos de los tipos más utilizados son:

- VARCHAR y VARCHAR2: En vez de String.
- NUMBER: Para guardar valores numéricos

Nota: En general utilizaremos VARCHAR para no complicarnos la vida :-).

DROP TABLE: La sentencia permite eliminar una tabla que esté vacía (sin registros).

```
DROP TABLE  tabla
```

Es bastante sencilla la sintaxis y no tiene más ciencia que esa.

Como se ha visto hasta ahora, solo sabemos utilizar ambos lados de la moneda. Sin embargo nos falta la capa intermedia y es como y donde se unen los programas Java con la sintaxis SQL de los RDBMS.

Ahora veremos paso a paso como utilizamos JDBC para realizar una consulta SQL y conectar un programa Java con una Base de Datos real (con RAF, o un RDBMS).

Realizar la Conexión

El primer paso que se debe realizar es abrir una conexión con el DBMS que se desea utilizar. Para ello usaremos 2 puntos importantes:

Cargar el Driver: Antes de cualquier cosa, Java debe saber contra qué se está enfrentando para abrir una conexión con una base de datos. Para ello utiliza unas clases especiales que son llamadas Drivers y que se bajan desde internet rápidamente.

Existen Drivers para un sin número de DBMS[29] como son MS Access, SQL Server, Oracle, DB2, etc. Para cargar el driver se utiliza un método estático de la clase **Class** llamado **forName**.

```
Class.forName("<nombre del driver>");
```

En general, se utiliza esta línea indicando el nombre del driver, que por lo general son de la forma **jdbc.DriverX**. Esta especificación siempre viene en la documentación del driver. Por ejemplo, para conectar a través del driver más básico (ODBC) se utiliza:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

y listo.

Abrir la Conexión: Para abrir la conexión, luego de haber cargado el driver, se debe crear un objeto de la clase **Connection** (del package java.sql.*) de la siguiente forma:

```
Connection c =
      DriverManager.getConnection ("<bd>", "<login>", "<password>");
```

quedando así en la variable c una conexión a la base de datos de nombre (url) **bd** y que el usuario utilizado está identificado por **login** y **password** como clave.

Con estos dos casos, por ejemplo, para conectar a una base de datos Access llamada **alumnos** y que está registrada como ODBC en el computador tenemos que realizar los dos pasos siguientes:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Connection c =
      DriverManager.getConnection ("jdbc:odbc:alumnos", "", "");

...

c.close();
```

[29] Ver la **Java Developer Connection** del sitio http://java.sun.com

Casualmente, las Bases de Datos Access no utilizan ni nombre de usuario ni clave para la conexión, así que esos parámetros se dejan vacíos.

Creando Sentencias JDBC

Con la conexión hecha (considere **c** como la variable de conexión), es necesario crear una sentencia para realizar una acción sobre la base de datos. Las sentencias son transportadas al DBMS a través de un objeto de tipo **Statement**:

```
Statement stmt = c.createStatement();
```

Con esta sentencia (que es la única forma posible), se crea un objeto **stmt** que se conectará con la conexión **c** para enviar un comando SQL al DBMS.

Existen 2 tipos de comandos en SQL que enviar:

Consulta: Un comando de consulta es aquél que retorna resultados (SELECT). Para ello se utiliza un método de la clase **Statement** llamado **executeQuery**:

```
ResultSet rs = stmt.executeQuery("<SQL de SELECT>");
```

Los resultados son guardados dentro de un objeto **ResultSet** para luego obtenerlos a través de un iterador:

```
while (rs.next()) {
      // Se obtienen los valores por cada registro
      ...
}
```

Con este ciclo, que lee en cada iteración 1 registro del **ResultSet**, se pueden ir obteniendo los valores para cada columna de los registros. Para obtener los valores se usan:

rs.getString("<colname o numcol>");	Obtiene una columna VARCHAR o CHAR
rs.getFloat("<colname o numcol>");	Obtiene una columna FLOAT o NUMERIC
rs.getInt("<colname o numcol>");	Obtiene una columna INTEGER o NUMERIC
rs.getObject("<colname o numcol>");	Obtiene cualquier tipo de dato.

Por ejemplo:

```
Statement stmt = c.createStatement();
Resultset rs = stmt.executeQuery("SELECT * FROM LECTORES" +
                                 "WHERE RUN = '13252311-8'");
while(rs.next()) {
      System.out.print ("  Nombre = " + rs.getString("NOMBRE"));
      System.out.print ("   Email = " + rs.getString("EMAIL"));
      System.out.print ("Teléfono = " + rs.getString("FONO"));
}
```


Otra cosa que es interesante es que se puede obtener el nombre de las columnas y también la cantidad de columnas de un ResultSet. Esto se puede usar utilizando otra clase llamada **ResultMetaData**:

```
ResultSetMetaData rsmd = rs.getMetaData();
int n = rsmd.getColumnCount();
for (int i=1; i<=n; i++) {
      System.out.println(rsmd.getColumnLabel(i));
}
```

Este pequeño código imprime en cada línea los nombres de las columnas de la tabla de resultados. Esto es muy útil al momento de desplegar la información en pantalla. Por ejemplo:

```
Statement stmt = c.createStatement();
Resultset rs = stmt.executeQuery("SELECT * FROM LECTORES" +
                                 "WHERE RUN = '13252311-8'");
ResultMetaData rsmd = rs.getMetaData();
int n = rsmd.getColumnCount();
while(rs.next()) {
      for (int i=1; i<=n; i++) {
            System.out.print(rsmd.getColumnLabel(i) + " = ");
            System.out.println(rs.getObject(i));
      }
      System.out.println();
}
```

Imprimiendo todos los datos de resultados sin saber los nombres de los campos de cada registro. Nótese que los índices parten desde 1 y no de 0 como es originalmente en todo en Java.

Actualización: Las sentencias de actualización son todos esos comandos que permiten realizar algún cambio en la estructura o en los datos de la Base de Datos (CREATE, DROP, INSERT, UPDATE o DELETE). Para ello se utiliza un método **executeUpdate** (que no retorna nada [void]) con el comando como parámetro:

```
stmt.executeUpdate("<SQL de Create/Drop o Insert/Update/Delete>");
```

Por ejemplo:

```
stmt.executeUpdate("DELETE * FROM LIBROS");
```

Utilizando Transacciones

Las transacciones son trozos de ejecución de comandos SQL que no son reflejados directamente en la Base de Datos. Es como si se armara un conjunto de comandos en un paquete y luego se enviara todo es set completo para ser ejecutado por el DBMS.

Antes de hacer una transacción, es importante saber cómo funciona en SQL. Las transacciones son limitadas por un COMMIT o un ROLLBACK. Todo el SQL que se va ejecutando, no se hace

efectivo sin un COMMIT al final. Si ocurre un error o simplemente para volver atrás uno utiliza el comando ROLLBACK.

En Java, la idea es la misma. Para realizar una transacción, primero se desactiva el auto commit (fin de transacción automático) que tiene por defecto la conexión. Luego van las instrucciones que en general son de actualización y se realiza un commit explícito (igual que en SQL). Al final se activa nuevamente el auto commit:

```
c.setAutoCommit(false);
// Instrucciones dentro de la Transacción
c.commit();
c.setAutoCommit(true);
```

Manejando Excepciones y Warnings

El problema que hay con las Bases de Datos es muy similar al ocurrido con los archivos: hay excepciones que se deben manejar para que funcionen bien.

La excepción más importante es **SQLException** que es la que guarda y obtiene todos los errores que vienen del DBMS y los pone como una excepción en Java. En general estos errores vienen con la codificación usada por el mismo DBMS para identificarlos. Por ejemplo con el driver entre corchetes y/o como ORA-xxxx para indicar el código del error de Oracle.

Es recomendable insertar el código de conexión y/o ejecución dentro de áreas críticas para atrapar la excepción (**try... catch**) o simplemente pasarla a un método mayor (**throws**).

También, y solo en el caso de poner **Class.forName(...)**, puede ocurrir la excepción **ClassNotFoundException**. Esta excepción ocurre cuando no se encuentra el driver que se pone entre paréntesis.

Por último, las conexiones con Bases de Datos pueden enviar algunas advertencias o warnings en caso de existir. Para ello se utiliza el método **getWarnings()** de la clase **Statement** y de la clase **ResultSet**.

Para obtener los warnings (en ambos casos) se utiliza, por ejemplo:

```
SQLWarning warn = stmt.getWarnings();
while (warn != null) {
      System.err.println("   Mensaje = " + warn.getMessage());
      System.err.println("  SQLState = " + warn.getSQLState());
      System.err.println("    Código = " + warn.getVendorCode());
      warn = warn.getNextWarning();
}
```

Ejemplo Práctico

El siguiente ejemplo permite abrir una conexión con una Base de Datos Access, enviar un INSERT y consultar datos mostrándolos en la salida estándar. La base de datos se llama **cc10a** y está registrada apuntando al archivo **cc10a.mdb** a través de ODBC.

Versión 1: En esta versión las excepciones son lanzadas fueras del método main para que la JVM tome el control de ellas

```
// Debemos importar algunos package
import java.sql.*;
import java.lang.*;

public class SQLConsulta {
      public void main (String[] args)
            throws SQLException, ClassNotFoundException {
            Console w = new Console();

            // Cargar Driver
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

            // Abrir conexión
            Connection c =
                  DriverManager.getConnection("jdbc:odbc:cc10a",
                        "scott", "tiger");

            // Crear el paquete
            Statement stmt = c.createStatement();

            String sql = w.readLine();
            if (sql.toUpperCase().indexOf("SELECT") == 0) {
                  // Consulta SQL
                  ResultSet rs = stmt.executeQuery(sql);

                  // Trabajar con los resultados
                  while(rs.next()) {
                        w.println("Valor = " + rs.getObject());
                  }
            }
            else {
                  // Ejecuta SQL
                  stmt.executeUpdate(sql);
            }

            // Cierra conexión
            c.close();
      }
}
```

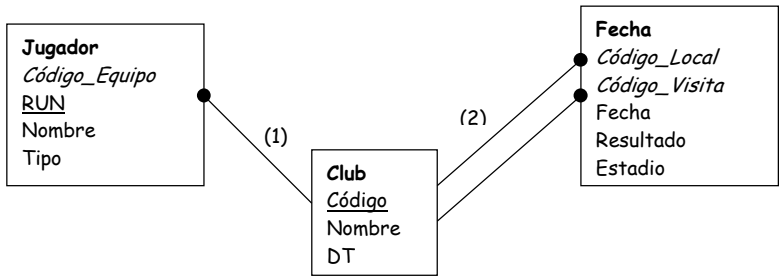
Versión 2: En esta versión captura las excepciones dentro del método main controlando sus resultados

```
// Debemos importar algunos package
import java.sql.*;
import java.lang.*;

public class SQLConsulta {
      public void main (String[] args) {
            Console w = new Console();

            try {
                  // Cargar Driver
                  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            }
            catch (ClassNotFoundException e) {
                  w.println(e.getMessage());
```

```
                System.exit(0);
            }

            try {
                // Abrir conexión
                Connection c =
                    DriverManager.getConnection(
                        "jdbc:odbc:cc10a",
                        "scott", "tiger");

                // Crear el paquete
                Statement stmt = c.createStatement();

                String sql = w.readLine();
                if (sql.toUpperCase().indexOf("SELECT") == 0) {
                    // Consulta SQL
                    ResultSet rs = stmt.executeQuery(sql);

                    // Trabajar con los resultados
                    while(rs.next()) {
                        w.println("Valor = " +
                                  rs.getObject());
                    }
                }
                else {
                    // Ejecuta SQL
                    stmt.executeUpdate(sql);
                }

                // Cierra conexión
                c.close();
            }
            catch (SQLException e) {
                w.println(e.getMessage());
                System.exit(0);
            }
        }
}
```

Problemas

Supongamos el siguiente modelo de datos:



En él podemos identificar que un club está compuesto por muchos jugadores y un jugador solo pertenece a un club (1), y que los clubes pueden estar participando en más de una fecha, y que en una fecha participan 2 equipos (2).

Escriba en SQL las siguientes consultas:

(a) Lista de jugadores (todos sus detalles) que pertenecen al club de la "*Universidad de Chile*".

```
SELECT jugador.run, jugador.nombre, jugador.tipo
  FROM jugador, club
 WHERE jugador.codigo_equipo = club.codigo
   AND club.nombre = 'Universidad de Chile'
```

(b) Lista de los partidos y su resultado que se jugaron el "*10/08/2001*".

```
SELECT local.nombre, visita.nombre, fecha.resultado, fecha.estadio
  FROM club local, club visita, fecha
 WHERE local.codigo = fecha.codigo_local
   AND visita.codigo = fecha.codigo_visita
   AND fecha.fecha = '10/08/2001'
```

En este caso estamos incluyendo un nuevo concepto de **sinónimos** para tablas, en las cuales ocurre que, en la sentencia FROM indicamos un nombre alternativo para una tabla (en el caso de tablas repetidas es muy útil). Es decir:

```
SELECT ...
  FROM tabla sinon, ...
```

Y todas las columnas son llamadas como **sinon.nombre_columna**.

(c) Nombre de los jugadores que participaron en el encuentro en que la "*Universidad Católica*" jugó de local el "*23/06/2001*".

```
SELECT jugador.nombre
  FROM jugador, club, fecha
 WHERE jugador.codigo_equipo = club.codigo
   AND club.nombre = 'Universidad CAtólica'
   AND club.codigo = fecha.codigo_local
   AND fecha.fecha = '23/06/2001'
```

(d) Listar todos los partidos en donde haya jugado "*Colo-Colo*" y sus resultados.

```
SELECT local.nombre, visita.nombre,
       fecha.fecha, fecha.resultado, fecha.estadio
  FROM club local, club visita, fecha
 WHERE local.codigo = fecha.codigo_local
   AND visita.codigo = fecha.codigo_visita
   AND ( local.nombre = 'Colo-Colo'
         OR visita.nombre = 'Colo-Colo' )
```

Lo nuevo de este caso es que ingresamos una sentencia OR dentro del WHERE de la consulta.

Capítulo XXI: Concurrencia

Motivación

Hasta ahora hemos visto que los programas corren y ejecutan en forma lineal algo. Por ejemplo, si estás ejecutando un problema matemático de integración, la solución es bastante pautada:

- Dibujar los rectángulos bajo la cubra
- Calcular el área de cada uno de los rectángulos
- Sumar las áreas
- Devolver como aproximación la suma

¿Qué significa esto? Pues que antes de poder continuar con el siguiente paso debes completar el paso actual. Una verdadera receta.

Pensemos un momento en el problema de un juego de damas.

- Cada pieza en el tablero posee un árbol de decisión (a veces 2 movidas, en caso de las piezas normales y más de 2 movidas en caso de las damas).
- El jugador posee entonces un árbol de decisión con al menos N subárboles, uno para cada pieza que tenga el tablero.

¿Creen que en forma lineal esto se resolvería rápidamente?

La respuesta a esta pregunta es que por supuesto que no. Si tenemos 10 piezas y solo analizamos un nivel, debemos considerar tantas movidas que ni siquiera podríamos mentalmente visualizarlas todas para elegir la mejor. Complicado sería para el computador hacerlo en tiempos razonablemente cortos.

Pensemos ahora en otro tipo de juegos. Imagínense que ustedes son los defensores de la tierra y vienen unos marcianos a atacarla con naves (¿Space Invaders?).

¿Cómo el computador va a manejar los movimientos de tu nave y los movimientos de los malos al mismo tiempo sin quedarse pegado?

Imagínate que se tarda 0.5 segundos en recoger un movimiento y realizarlo en pantalla. Además tenemos 25 naves en la pantalla dibujadas y que deben tener movimiento propio. Por lo tanto de manera secuencial, dibujar la pantalla costaría una espera de 0.5 x 25 segundos, es decir, 12.5 segundos. Ahora llevamos nuestro juego al procesador más rápido del mundo que tarda solo 0.1 segundos en recoger y dibujar. Tomaría exactamente 2.5 segundos en toda la pantalla. Todo esto sin considerar que también se deben mover los proyectiles disparados de cada nave… ¡Ufff!... Me aburrí muy rápido de jugar.

Pero en realidad no hay tanto problema, porque la solución existe y se llama **Concurrencia**.

Conceptos

> *La **Concurrencia** es la capacidad de ejecutar distintos procesos de manera síncrona y de forma independiente.*

Hasta ahora nuestros programas no han tenido concurrencia, por supuesto. Con este concepto se acabaron los procesos batch y las largas esperas de procesamiento mientras se ejecuta el programa. Pero sin embargo, la concurrencia necesita de algo que se llama **Proceso**.

> *Un **Proceso** es un trozo de programa que se ejecuta en un espacio separado de memoria, aislado del programa principal.*

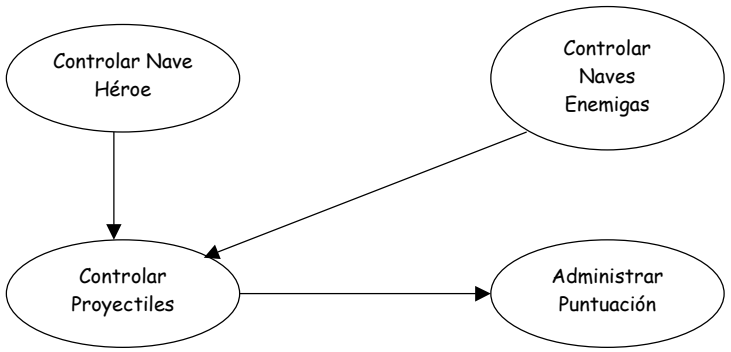
Básicamente un proceso es un programa en ejecución, pero que no depende de nadie. Este tan independiente programa ejecuta sus funciones y al terminar solo desaparece. El ejemplo más claro de estas cosas es el **MSN Messenger**. Si tienes tu Messenger ejecutándose como un **Proceso**, te permitirá navegar en internet, jugar Quake, hacer tu tarea de computa, pero sin interrumpir tu trabajo el continúa su ejecución.

¿Para qué sirven?

Imagínate nuevamente el Space Invaders que debe realizar muchas cosas:

- Mover naves enemigas
- Recibir movimientos de la nave héroe
- Mover proyectiles
- Calcular puntuación y manejar vidas

Si todo esto lo realiza en forma lineal, sería muy lento. Veamos cómo sería en forma paralela:



Las burbujas del diagrama representan los proceso que realiza y cómo se comunican (a grandes rasgos). Entonces, ¿cómo leer el diagrama?: "Mientras se controla a la nave héroe y a las naves enemigas, se verifica que los disparos lleguen a sus objetivos y si es así hay que sumar puntos o restar vidas.

Suena sencillo, pero en la práctica ¿cómo se hace?, con objetos especiales que veremos ahora:

Sintaxis

En concurrencia se utiliza una clase especial llamada **Thread**. Esta clase está construida de la siguiente forma:

```
public class java.lang.Thread
       extends java.lang.Object
       implements java.lang.Runnable {
    java.util.Map threadLocals;
    java.util.Map inheritableThreadLocals;
    public static final int MIN_PRIORITY;
    public static final int NORM_PRIORITY;
    public static final int MAX_PRIORITY;
    public static native java.lang.Thread currentThread();
    public static native void yield();
    public static native void sleep(long)
      throws java.lang.InterruptedException;
    public static void sleep(long, int)
      throws java.lang.InterruptedException;
    public java.lang.Thread();
    public java.lang.Thread(java.lang.Runnable);
    public java.lang.Thread(java.lang.ThreadGroup,
                            java.lang.Runnable);
    public java.lang.Thread(java.lang.String);
    public java.lang.Thread(java.lang.ThreadGroup,java.lang.String);
    public java.lang.Thread(java.lang.Runnable,java.lang.String);
    public java.lang.Thread(java.lang.ThreadGroup,
                            java.lang.Runnable,java.lang.String);
    public native synchronized void start();
    public void run();
    public final void stop();
    public final synchronized void stop(java.lang.Throwable);
    public void interrupt();
    public static boolean interrupted();
    public boolean isInterrupted();
    public void destroy();
    public final native boolean isAlive();
    public final void suspend();
    public final void resume();
    public final void setPriority(int);
    public final int getPriority();
    public final void setName(java.lang.String);
    public final java.lang.String getName();
    public final java.lang.ThreadGroup getThreadGroup();
    public static int activeCount();
    public static int enumerate(java.lang.Thread[]);
    public native int countStackFrames();
    public final synchronized void join(long)
      throws java.lang.InterruptedException;
    public final synchronized void join(long, int)
      throws java.lang.InterruptedException;
    public final void join() throws java.lang.InterruptedException;
```



```
    public static void dumpStack();
    public final void setDaemon(boolean);
    public final boolean isDaemon();
    public final void checkAccess();
    public java.lang.String toString();
    public java.lang.ClassLoader getContextClassLoader();
    public void setContextClassLoader(java.lang.ClassLoader);
    static {};
}
```

¿Qué significa esto? Bueno, en realidad son muchos métodos que se pueden utilizar con threads, pero que solo utilizaremos una pequeña parte en las clases que heredaremos de ella.

Para crear un proceso independiente, se debe crear una clase que extienda de Thread:

```
public class MiThread extends Thread {
       public void run() {
              for (int i=1; i<=10; i++) {
                     System.out.println(i);
              }
              System.out.println("FIN DEL THREAD");
       }
}
```

Este thread como pueden verlo posee solo un método propio (todos los demás los trae la clase Thread) y ese se llama **run()**. Este método es OBLIGATORIO (al igual que el main en el caso de los programas normales) y es el encargado de ejecutar el thread, es decir, allí se programa lo que el thread hace. En este caso, solo imprime los números del 1 al 10 en pantalla, uno por línea.

Si nosotros compilamos esta clase y la ejecutamos no pasaría nada, de hecho echaría de menos el main el compilador y reclamaría. ¿Por qué?, bueno, porque no se puede ejecutar un thread como si fuese un programa principal, si no que se utiliza lanzándolo desde otro programa:

```
public class programa {
       static public void main (String[] args) {
              Thread t = new MiThread();
              t.start();
              System.out.println("FIN DEL PROGRAMA PRINCIPAL");
       }
}
```

Notemos claramente que este programa crea un thread del tipo personalizado que cuenta de 1 a 10 en pantalla (clase MiThread), lo echa a correr utilizando el método **start()** nativo de la clase Thread y luego pone en pantalla el texto "FIN DEL PROGRAMA PRINCIPAL". Con lo que sabemos ahora, esperaríamos la siguiente salida:

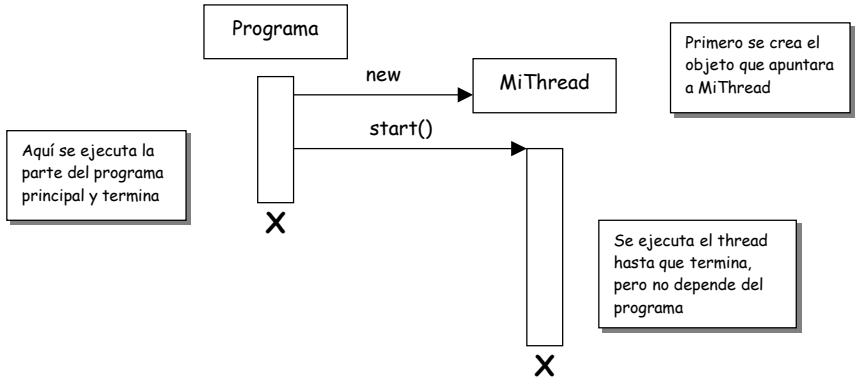
```
1
2
...
9
10
FIN DEL THREAD
FIN DEL PROGRAMA PRINCIPAL
```

Sin embargo esto no es realmente lo que pasa, y la salida real es la siguiente:

```
FIN DEL PROGRAMA PRINCIPAL
1
2
...
9
10
FIN DEL THREAD
```

¿Que diablos?

Bueno, este tema es sencillo. Veamos con un diagrama de proceso lo que ocurre (ojo que se parece a un diagrama de interacción de UML):



Si se dan cuenta, el objeto Programa termina mucho antes que el objeto MiThread, es decir el programa puede terminar sin que el thread haya terminado su labor. Línea a línea pasaría lo siguiente:

```
public class programa {
static public void main (String[] args) {

        Thread t = new MiThread();

        t.start();

        System.out.println("...");

}
}
```

```
public class MiThread extends Thread {

public void run() {

        System.out.println("1");

        System.out.println("2");
        ...
        System.out.println("9");
        System.out.println("10");

        System.out.println("...");

}
}
```

Entonces, el thread comienza justo al hacer el start().

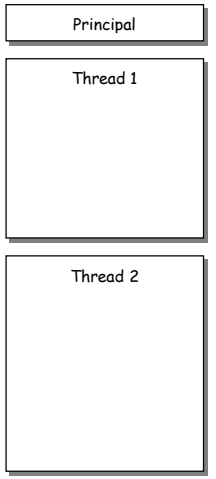
Compliquemos el ejemplo. Supongamos que nuestro programa principal sea:

```
public class programa {
       static public void main (String[] args) {
              Thread t1 = new MiThread();
              Thread t2 = new MiThread();
              t1.start();
              t2.start();
              System.out.println("FIN DEL PROGRAMA PRINCIPAL");
       }
}
```

¿Qué debería salir?

En este caso no es facil adivinarlo, porque la ejecución de 2 thread se torna compleja. Así que la salida al ejecutar el programa es:

```
FIN DEL PROGRAMA PRINCIPAL
1
2
3
4
5
6
7
8
9
10
FIN DEL THREAD
1
2
3
4
5
6
7
8
9
10
FIN DEL THREAD
```



¡Hey, eso sí que es extraño!

Si lo pensamos de cómo funciona Java, pues no es tan extraño. Recuerda que el thread posee el método **start()** definido como synchronized, es decir, se ejecuta un método run() a la vez. Pero entonces, ¿dónde está el paralelismo? ¡esto es secuencial!. Veamos si cambiamos algo en MiThread:

```
public class MiThread extends Thread {
       public void run() {
              for (int i=1; i<=10; i++) {
                     System.out.println(i);
                     try {
                            super.sleep(10);
                     }
                     catch (InterruptedException e)  {
```

```
                }
            }
            System.out.println("FIN DEL THREAD");
        }
}
```

Y la nueva salida sería:

```
FIN DEL PROGRAMA PRINCIPAL
1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
9
9
10
10
FIN DEL THREAD
FIN DEL THREAD
```

¡Ahora si que quedamos mal!....

Si tampoco es tan complicado el tema si explicamos primero cuál fue la modificación que le hicimos al thread para que hiciera esto. **sleep(long)** es un método de la clase Thread que permite hacer "dormir" el proceso durante una cantidad de milisegundos (¡si!, una pequeña siesta) que se le pasan por parámetros, es decir, que se queda "esperando" durante un momento antes de continuar con la ejecución.

Como en este caso estamos esperan 10 milisengundos, podemos decir que se intercala la ejecución de cada thread, mostrando que quedan prácticamente en paralelo (los dos al mismo tiempo).

Además, si sumamos el hecho que el start() estaba sincronizado, significa que cada proceso hará un ciclo por vez en el procesador, y cada vez que el proceso haga sleep() estaremos avisando para que otro tome el control del procesador por un ciclo más o hasta que éste haga otro sleep. Cuando el sleep se acaba, el proceso que despierta toma el control y comienza de nuevo a ejecutar desde donde se durmió.

Manejando las Excepciones del Thread

Como lo dice su definición, algunos métodos deben manipular excepciones. ¿Cuáles son esas excepciones?, bueno, eso depende del método.

En el caso del método **run()** estamos obligados a capturar la excepción **InterruptedException** que nos permite controlar cuando el thread es interrumpido en su ejecución. Esta interrupción puede pasar por distintas razones: falta de memoria, llamada del proceso, etc.

Ejemplo

Veamos ahora un programa "real". Queremos hacer la gráfica de un reloj, pero que se mueva segundo a segundo un indicador. Sería sencillo hacer primero la gráfica:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Reloj {
  private Frame programa;
  private Canvas area;

  public Reloj() {
    programa = new Frame("JReloj");
    programa.setLayout(new FlowLayout());

    area = new Canvas();
    area.setSize(600, 600);
    programa.add(area);

    programa.pack();
    programa.show();

    Graphics pincel = area.getGraphics();
    pincel.setColor(Color.blue);
    pincel.drawOval(10, 10, 590, 590);

    Thread seg = new Timer(pincel, 300, 200);
    seg.start();
  }

  public static void main(String[] args) {
    Reloj r = new Reloj();
  }
}
```

Fijémonos que la lógica de la interfaz es nula y que solo estamos pasándole el control del reloj a un proceso Timer llamado **seg**. La lógica entonces estaría acá:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

class Timer extends Thread {
      Graphics segundero;
      int pos_x, pos_y;
      long seg = 180;
      public Timer (Graphics g, int x, int y) {
            this.segundero = g;
            this.pos_x = x;
            this.pos_y = y;
      }
```

```
	public void run() {
		int ang = 6, x, y;
		while(true) {
			x = (int) Math.round(250 *
				Math.sin(Math.toRadians(seg)));
			y = (int) Math.round(250 *
				Math.cos(Math.toRadians(seg)));

			segundero.setColor(Color.blue);
			segundero.drawLine(pos_x, pox_y,
				pos_x + x, pos_y + y);

			try {
				super.sleep(1000);
			}
			catch(Exception e) {
			}

			segundero.setColor(Color.white);
			segundero.drawLine(pos_x, pox_y,
				pos_x + x, pos_y + y);

			seg -= ang;
			if (seg == 360) {
				seg = 0;
			}
		}
	}
}
```

En el constructor hacemos almacenar la posición inicial del segundero y además el pincel que nos permitirá dibujar sobre el canvas de la interfaz creada antes. El método run() en este caso tiene la misma estructura que cualquiera y se duerme el proceso cada cierto tiempo por un lapso de 1 segundo. Mientras el proceso se despierta, borra el segundero desde donde estaba, le aumenta el ángulo al segundero (en este caso el ángulo es 6, porque si dividimos 360 / 60 partes nos queda que cada 6° debería correr el segundero) y luego calculamos las nuevos valores para la proyección sobre el eje x e y del segundero y así dibujarlo. Luego de esto, a dormir de nuevo.

La lógica es super simple, pero el programa ya se ve más complicado. Sin embargo la parte de paralelismo o concurrencia es siempre igual.

Sin embargo, la gracia del ejemplo radica en la sencilla premisa de que el procesador se mantiene más tiempo desocupado que si lo hubiésemos hecho dentro de la clase de la interfaz. Esto implica:

- Ahorro de memoria
- Ahorro de tiempo de espera
- Aprovechamiento del tiempo ocioso del procesador

Capítulo XXII: Comunicación de Datos

(en construcción)

Capítulo XXIII: Paquetes de Clases

Motivación

¿Alguna vez te preguntas qué era esos java.awt.* o java.io.* que el profe ponía en los imports de las clases? Pues siempre se decía que los import lo que hacían era importar bibliotecas de clases que pertenecían a Java propiamente tal.

Descubriremos en velo de la duda ahora indicando cómo se crean esas bibliotecas y cómo se usan.

Conceptos

Bilbioteca o Paquete

> *Conjunto de clases agrupadas por una temática definida y que pueden ser reutilizadas en cualquier programa.*

Esta definición tan sencilla puede ser una herramienta potentísima si conocemos cuál es la estructura que permite crear bilbiotecas o paquetes. Vamos a la parte sintáctica rápidamente para que podamos aprender esto de una vez por todas.

Sintaxis

Un paquete de clases en java es llamado **Package**. Estas estructuras son simples directorios, a partir del CLASSPATH, que contienen las clases.

Por ejemplo, si tenemos las clases matemáticas en el directorio **C:\Bibl\cc10a\matematica** y nuestro CLASSPATH está definido como **C:\Bibl**, entonces, las clases matemáticas que estamos utilizando se encuentran en el package **cc10a.matematica**.

A pesar de esto, no es tan sencillo llegar y utilizar esto como un pretexto de orden, sino que se debe indicar dentro de la clase a cuál biblioteca corresponde.

Veamos ahora cada uno de los temas relevantes de los packages.

Definir un Package

Antes de comenzar a utilizar un package, es necesario definirlo, es decir, especificar donde va a estar ese package almacenado. Tenemos varias alternativas:

(a) Definir el package en el directorio **lib** de mi JDK, es decir, por ejemplo en el directorio **C:\jdk1.2\lib**.

La ventaja de esta opción es que no tenemos que incluir ningún directorio adicional en el CLASSPATH, ni tampoco tener definido un directorio home para ejecutar mis clases.

Además de esta forma, dejamos la biblioteca disponible para cualquier desarrollo que se nos ocurra (más genérico).

Por ejemplo, definamos la biblioteca matemática:

```
C:\> cd \jdk1.2\lib
C:\jdk1.2\lib\> cd cc10a               //   HOME   de   bibliotecas
CC10A[30]
C:\jdk1.2\lib\cc10a\> mkdir matematica  // Biblioteca matemática
C:\jdk1.2\lib\cc10a\> cd matematica
C:\jdk1.2\lib\cc10a\matematica\> _
```

(b) Definir el package en un directorio que si esté en el CLASSPATH o que agreguemos al CLASSPATH como home de mis bibliotecas, por ejemplo, **C:\javaBibl**.

Al igual que en el anterior caso, la ventaja que tiene de separarlas es que cuando queremos reutilizarlas, las podemos tener accesibles siempre, sin fijar nuestro directorio de ejecución de clases, sin embargo, posee la facultad de que si cambiamos o reinstalamos el JDK, no es necesario respaldar la bilbioteca.

Por ejemplo, definamos las bibliotecas en nuestro PC:

```
C:\> cd javaBibl                 // HOME de bibliotecas CC10A
C:\javaBibl\> mkdir matematica   // Biblioteca matemática
C:\javaBibl\> cd matematica
C:\javaBibl\matematica\> _
```

¡Se parece a lo anterior! Pero como decíamos, esto no depende del jdk como lo anterior. Esto nos llevaría a definir adicionalmente algo en nuestro CLASSPATH. Veamos:

```
C:\> set CLASSPATH        // Verificamos qué tiene el CLASSPATH
CLASSPATH=.;C:\jdk1.2\lib
C:\> set CLASSPATH=%CLASSPATH%;C:\javaBibl
```

Como nos podemos dar cuenta, lo que estamos haciendo es simplemente agregar al CLASSPATH que ya existe, la ruta base de la biblioteca. Así usaremos toooooda la bilbioteca entera y no es necesario hacerlo por cada una de las bilbiotecas que existan.

(c) Definir el package desde el directorio home en el cual comienzo a ejecutar mis clases, por ejemplo, **C:\CC10A**.

A modo de orden, personalmente pienso que es la mejor opción. De esta forma tenemos un directorio de desarrollo y desde allí montamos todos los programas que queramos. De esta forma trabajamos sobre un directorio no más sin necesidad de setar el CLASSPATH ni preocuparnos del JDK que estemos utilizando.

[30] Recuerda que si no existe la bilbioteca cc10a, es necesario crearla antes poniendo:

```
C:\jdk1.2\lib\> mkdir cc10a
```

Crear una Clase en un Package

Una vez definido dónde guardaremos la bilbioteca, es necesario empezar a desarrollar clases dentro de ella. Para ello se usa una sentencia que va al principio de la clase (antes que cualquier cosa) y que indica a qué biblioteca pertenece:

package <nombre_bilbioteca>

En donde el nombre de la bilbioteca es igual a los directorios desde la raíz de la bilbioteca en donde se encontrará la clase, separados por puntos (".").

Supongamos como ejemplo la biblioteca matemática, y definamos la clase **Complejo** de la clase matemática.

```
package matematica;
public class Complejo {
	public double parte_real;
	public double parte_img;
	public class Complejo (double r, double i) {
		this.parte_real = r;
		this.parte_img = i;
	}
	...
}
```

Fíjate que la única diferencia en la que incurrimos es que pusimos el package al principio. Si quieres guardar esta clase, debes hacerlo en un archivo complejo.java dentro del directorio matematica en el home de bibliotecas.

Supongamos que queremos guarda la biblioteca ahora como cc10a\mat (es decir biblioteca matemática de cc10a). Escribamos la clase:

```
package cc10a.mat;
public class Complejo {
	public double parte_real;
	public double parte_img;
	public class Complejo (double r, double i) {
		this.parte_real = r;
		this.parte_img = i;
	}
	...
}
```

Y lo guardamos en el directorio cc10a\mat del home de bibliotecas. ¿Sencillo?

Usar un Package

Para utilizar un package o biblioteca existe algo que hemos visto más de una vez y es el comando import:

import <nombre_package>.*;

Esta es la forma genérica y significa que importa TODAS las clases que están definidas como públicas en el package, es decir, tenemos acceso a todas las clases que están definidas dentro de la biblioteca. Otro ejemplo de import es:

import <nombre_package>.<nombre_clase>;

En donde importa solo una clase específica del package, es decir, solo tendremos acceso a ESA clase de la biblioteca.

Hagamos un programa que sume 2 complejos:

```
import matematica.*;			// importamos la biblioteca donde
					// está la clase complejo
public class Programa {
	public static void main(String[] args) {
		Complejo a = new Complejo (1, 2);
		Complejo b = new Complejo (2, 3);
		Complejo ab = new Comlejo (a.parte_real + b.parte_real,
			a.parte_img + b.parte_img);
	}
}
```

En este caso supusimos que la clase Complejo la guardamos en la biblioteca matematica. Supongamos ahora el segundo ejemplo, en la biblioteca cc10a.mat:

```
import cc10a.mat.*;		// importamos la biblioteca donde
				// está la clase complejo
public class Programa {
	public static void main(String[] args) {
		Complejo a = new Complejo (1, 2);
		Complejo b = new Complejo (2, 3);
		Complejo ab = new Comlejo (a.parte_real + b.parte_real,
			a.parte_img + b.parte_img);
	}
}
```

Ambos casos son iguales, la diferencia radica en la biblioteca. Pero ¿qué pasa si la biblioteca (cc10a.mat o matematica, depende del caso) posee más clases que no utilizamos? Podemos usar la segunda versión de import para que importemos solo la clase útil:

```
import cc10a.mat.Complejo;		// importamos la clase COMPLEJO
public class Programa {
	public static void main(String[] args) {
		Complejo a = new Complejo (1, 2);
		Complejo b = new Complejo (2, 3);
		Complejo ab = new Comlejo (a.parte_real + b.parte_real,
			a.parte_img + b.parte_img);
	}
}
```

Por supuesto existe una tercera forma de usar la clase Complejo sin utilizar import y es referenciarla directamente desde su package. ¿Qué diferencia hay? usas muchas veces el mismo texto para referenciar la clase:

```
public class Programa {
	public static void main(String[] args) {
		cc10a.mat.Complejo a = new cc10a.mat.Complejo (1, 2);
		cc10a.mat.Complejo b = new cc10a.mat.Complejo (2, 3);
		cc10a.mat.Complejo ab = new cc10a.mat.Comlejo
			(a.parte_real + b.parte_real,
			 a.parte_img + b.parte_img);
	}
}
```

¿Peludo?

Compilar y Ejecutar una clase de un Package

Obviamente no todo iba a ser sencillo, porque no es tan facil ejecutar una clase. Comencemos con la compilación.

Si la clase que nosotros programamos para el package la guardamos en cc10a\mat entonces deberemos meternos en ese directorio para compilarlo, igual como si compiláramos cualquier otra.

El problema viene al momento de ejecutar. Para hacer ésto, debemos realizarlo desde la raíz de las bibliotecas, es decir, desde el C:\Bibl por ejemplo:

```
C:\Bibl\> java cc10a.mat.Complejo
```

Como podemos ver, para llamar una clase de un package, debemos anteponerle el nombre del package obviamente. Pues ese realmente es el nombre.

En general no se ejecutan esas clases, sino que uno crea sus propias clases que utilizan los packages, o una clase principal que controla todo el modelo de paquetes.

Problema

Pongamos en práctica lo que has aprendido. Tenemos definida la siguiente clase:

```
import java.io.*;
public class ArchivoLectura {
	private BufferedReader fd;
	public ArchivoLectura (String nombre) {
		try {
			this.fd = new BufferedReader(
					new FileReader(nombre));
		}
		catch (Exception e) {
		}
	}
	public String leerLinea() {
		try {
			return this.fd.readLine();
		}
		catch (Exception e) {
			return null;
		}
```



```
	}
	public void cerrar() {
		try {
			this.fd.close();
		}
		catch (Exception e) {
		}
	}
}
```

(a) A partir del directorio C:\Java\CC10A crear la biblioteca io.archivo.texto que permite manipular archivos de texto.

R: Suponemos que estamos parados en el directorio Java\CC10A indicado y creamos la carpeta io:

```
C:\Java\CC10A\> mkdir io
C:\Java\CC10A\> cd io
C:\Java\CC10A\io\> _
```

Creamos la carpeta archivo:

```
C:\Java\CC10A\io\> mkdir archivo
C:\Java\CC10A\io\> cd archivo
C:\Java\CC10A\io\archivo\> _
```

Y creamos la carpeta texto:

```
C:\Java\CC10A\io\archivo\> mkdir texto
C:\Java\CC10A\io\archivo\> cd texto
C:\Java\CC10A\io\archivo\texto\> _
```

Luego verificamos si la carpeta Java\CC10A está en el CLASSPATH:

```
C:\Java\CC10A\io\archivo\texto\> set CLASSPATH
```

Si ese directorio no está, lo agregamos al CLASSPATH:

```
C:\Java\CC10A\io\archivo\texto\> set CLASSPATH=%CLASSPATH%;C:\Java\C
C10A
C:\Java\CC10A\io\archivo\texto\> _
```

(b) Indicar qué se debe hacer para que la clase pertenezca a ese paquete.

```
package io.archivo.texto;
import java.io.*;
public class ArchivoLectura {
	...
}
```

Después de que esté escrita la clase, se debe guardar en el directorio creado en la parte (a)

(c) Escribir un programa que permita imprimir en la salida estándar (System.out) todo el contenido de un archivo utilizando la clase ArchivoLectura.

```
import io.archivo.texto.*;
public class Programa {
	public static void main (String[] args) {
		ArchivoLectura al = new ArchivoLectura("xxx");
		while (true) {
			String linea = al.leerLinea();
			if (linea == null) break;
			System.out.println(linea);
		}
		al.cerrar();
	}
}
```


Capítulo XXIV: Diseño de Software UML[31]

Motivación

Hoy te levantas de tu cama y tienes un mail esperando en tu computadora. Vas y te das cuenta que el Presidente te encarga resolver un problema que te plantea de la siguiente forma:

> From: Ricardo Lagos
> Subject: Proyecto de Software
>
> Hola.
>
> He sabido que los alumnos de CC10A tienen mucha fama en desarrollar programas para cualquier tipo de problemas.
>
> Necesitamos un software que nos permita modelar el comportamiento de las leyes en el Congreso. Es decir, que vigile una ley desde que esta se escribe por algún profesional hasta que es aceptada por las Cámaras Alta y Baja y es promulgada tras mi firma.
>
> Espero que puedan ayudar al país. Recibirán una gratificación si realizan este programa.

Como ven, este extraño enunciado es muy abstracto y no nos hablan de clases, objetos, variables de instancia, métodos, ni nada por el estilo.

Veremos algunas prácticas que nos pueden ayudar a "traducir" el problema de un usuario a especificaciones de un software.

Concepto

Para el desarrollo de software no basta conocer un lenguaje de programación y saber programar, sino que también es necesario comprender las necesidades de un cliente o las especificaciones de un problema. Para ello existen metodologías de diseño de software:

Metodología

> *Están provistos de lenguajes de modelamiento y procesos de desarrollo que nos permiten modelar un problema y especificar el producto final.*

Existen muchas metodologías que son utilizadas por los profesionales en la actualidad. En este capítulo nos enfocaremos en la primera parte de la metodología que es el lenguaje.

UML

> *La sigla UML significa Unified Modeling Language (lenguaje unificado de modelamiento) y es la notación, principalmente gráfica, para expresar diseños de software Orientados al Objeto.*

[31] Toda esta materia fue sacada del libro UML Gota a Gota (UML Destilled) de Fowler y Scott.

UML es el sucesor de la oleada de métodos de análisis y diseño orientados a objeto (OOA&D) que surgió a fines de la década del 80 y principios de los 90. Principalmente unifica los métodos de Booch, Rumbaugh (OMT) y Jacobson[32], pero su alcance es mucho más amplio.

Para entender UML es muy importante entender Orientación al Objeto (OO), y que Java tiene como base en su programación. Se dice que un buen modelamiento hecho con UML puede ser "traducido" fácilmente en un software OO, en nuestro caso hecho en Java.

Las técnicas de UML son:

Tarjetas CRC (Clase – Responsabilidad – Colaboración)[33]

Oficialmente no pertenecen a UML, pero son muy valiosas para aprender OO. Originalmente las tarjetas CRC fueron diseñadas para trabajar con objetos. El término "tarjetas" fue usado por su característica física de ser tarjetas de 6x4 pulgadas de tamaño:

Nombre de la Clase	
Responsabilidades	Colaboración
Responsabilidad 1 Responsabilidad 2 ...	Clase 1 Clase 2 ...

Las **Clases** eran representadas por cada una de las tarjetas y son fieles elementos de modelo que interpretan una entidad en la vida real (dentro del problema).

Las **Reposabilidades** son la descripción de los propósitos de la clase (métodos). El espacio era reducido, por lo que las descripciones eran muy escuetas en unas cuántas frases (solo lo que cabía dentro de la tarjeta).

Las **Colaboraciones** eran clases con las que se trabajaba para cumplir el o los propósitos de la clase (uso de objetos).

Diagramas de Interacción

Son muy útiles para hacer explícita la estructura de los mensajes y, en consecuencia, tienen la ventaja de resaltar los diseños demasiado centralizados, en los que un objeto realiza todo el trabajo. Estos diagramas pueden ser de 2 tipos:

Los **Diagramas de Secuencia** son diagramas que especifican el orden de llamadas y procesamiento de la información a través de todos los objetos que pertenecen al sistema. Es

[32] Booch, Rumbaugh y Jacobson son conocidos en el ámbito de Software como "The Three Amigos" al igual que en esa película cómica ambientada en México.
[33] Ver http://www.c2.com/doc/oopsla89/paper.html. No hay muchas publicaciones al respecto y es una de las originales que aún existe.

bastante intuitivo pasar de un diagrama de secuencia a un programa, pues sale casi línea a línea que va realizando el mismo.

Los **Diagramas de Colaboración** son diagramas que muestran la comunicación entre las clases y los objetos que interactúan en el sistema.

Diagramas de Clases

Son usados para ilustrar modelos de clases. Los modelos de clases son muy similares a los modelos de datos (diagramas de tablas) que hemos visto en el capítulo anterior, por lo que resultan cómodos. El mayor problema de estos modelos es que se tiende a orientarlo a los datos y no al objeto (algo que hay que tener mucho cuidado).

Diagramas de Caso de Uso

Son diagramas que muestran la interacción de los actores (entidades externas) con distintos módulos o clases del sistema. Es muy gráfico e intuitivo, pues especifica con poca notación los procesos que el software debe seguir y son muy fáciles de interpretar por el cliente.

Concepto de Patrones

El empleo de patrones es vital para OO, pues hace centrarse en lograr buenos diseños y aprender en base a ejemplos. Una vez que hayamos logrado el dominio de algunas técnicas para modelar, tales como diagramas de clases o de interacción, será el momento de ver los patrones.

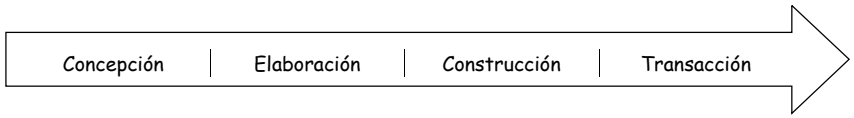
Desarrollo Iterativo Incremental

Algo no tan despreciable es utilizar la técnica o proceso de desarrollo adecuado. Con UML viene de la mano el desarrollo llamado **Iterativo-Incremental** que trata de partir de una base sencilla del problema completo y resolverlo de manera iterativa, es decir, con ciclos cortos dentro de los cuales se vayan solucionando los sub-problemas más importantes hasta tener al final el desarrollo completo del problema que el cliente plantea.

Es así como para realizar un software, no es tan sencillo como sentarse frente al computador y programar. Ahora, hagamos un alto y comencemos viendo cómo es el proceso de desarrollo paso a paso y la sintaxis que el lenguaje UML provee para el desarrollo de este proceso utilizado por los ingenieros en la actualidad.

Proceso de Desarrollo

La técnica del **Desarrollo Iterativo-Incremental** es la clave para explotar la OO.



Se divide en 4 etapas de desarrollo:

La **Concepción** es la primera etapa, la cual permite ver el problema en su punto de vista global. Es crucial que el equipo de desarrollo entienda la importancia de entender y comprender las necesidades del cliente o los puntos más relevantes del problema y aclarar la nebulosa que siempre tiene el cliente como idea de producto.

La **Elaboración** es la segunda etapa. En ella se realiza la mayor especificación de requerimientos del software que se desea construir. El contacto con el cliente es crucial en esta etapa, porque es quien realiza las observaciones necesarias para que se ajuste a sus necesidades. La utilización de diagramas y UML en la especificación de requerimientos es muy importante, pues los diagramas deja mucho más claros tanto a los clientes (Casos de Uso) como a los codificadores (Diagramas de Clases y Secuencia).

La **Construcción** es la tercera y más fuerte etapa. En ella se desarrollan la mayor parte de los requerimientos especificados y se complementan con las observaciones del cliente. La mayor parte del tiempo empleado en esta etapa se gasta en la codificación de los requerimientos, es por eso que, a través de los distintos ciclos que posee, es importante que la calidad del software sea clara, ya que como es un desarrollo incremental, toda pieza de software será utilizada de alguna manera en el producto final.

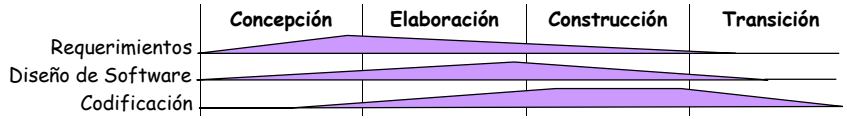
La **Transacción** es la última etapa y conforma toda la parte de implantación, seguimiento, afinamiento y marcha blanca del software. En general en esta etapa no se especifican nuevos requerimientos pero se realiza un afinamiento de aquellos que ya fueron desarrollados. Muchas veces en esta etapa queda la capacitación de los usuarios, que está muy de moda en la actualidad. Los paquetes de software ya no son entregados como cajas negras y las empresas requieren más seguido una capacitación de su personal para la utilización de ellos.

Cada una de estas etapas, posee un elemento que es llamado **iteración**. Cada una es un ciclo de desarrollo del sub-proyecto en donde posee ciertas características (en forma general [34]):

- Planificación de la Iteración
- Especificación de Requerimientos
- Diseño de Software
- Codificación
- Pruebas

[34] Cada iteración posee más sub-etapas, pero principalmente se pueden dividir en esas 5 áreas.

La carga de cada una de estas características a través de las fases del desarrollo se distribuyen más o menos de la siguiente forma:



La iteración posee un tiempo definido y abarca el espectro de un grupo de funcionalidades específicas (sub-proyecto). Al final de cada etapa la idea es tener un prototipo ya armado y funcionando, que es un software, más pequeño que el deseado, pero que está completamente operativo.

A través de cada etapa del desarrollo, las iteraciones se van distribuyendo según la cantidad de requerimientos. Pero en general, se puede tener un número fijo de iteraciones por etapa:

- Concepción: 1 iteración que básicamente solo es la definición del alcance del proyecto y la planificación del proyecto.
- Elaboración: 1 a 3 iteraciones en donde se reúnen los requerimientos más detallados, se hacen análisis y diseños de alto nivel para definir arquitectura base y se crea el plan de construcción.
- Construcción: Muchas iteraciones dividiendo la cantidad de requerimientos funcionales a través del tiempo para construir los requerimientos en productos incrementales.
- Transición: 1 iteración que básicamente es la implantación y marcha blanca del proyecto. Acá se incluye la capacitación y la afinación del desempeño, aunque esto puede ser otra iteración adicional.

También es muy importante mencionar que para el desarrollo con este proceso, los autores han procurado crear otros elementos importantes que definen cómo trabajar, a esto se le llama **Roles**.

Los roles dentro del proceso de desarrollo cumplen la función de indicar quién debe ser el responsable de cada una de los pasos en una iteración. Es así como hay Diseñadores de Software, Codificadores, Ingenieros de Requerimientos, etc. que no necesariamente son distintas personas, pero si son distintos "personajes" en ese momento.

Todo este proceso de desarrollo es llamado actualmente RUP (Rational Unified Process) [35] y es acompañado generalmente por documentación que explica paso a paso todo esto.

[35] **Rational** es una empresa creada por los autores de UML y su proceso posee un sin fin de herramientas para el apoyo al desarrollo de cada uno de los elementos que son necesarios y que veremos en este capítulo.

Casos de Uso

Un Caso de Uso es en esencia una interacción entre un usuario o personaje externo y un sistema computacional. Los casos de uso básicamente cumplen con 3 características:

- Capta alguna funcionalidad visible para el usuario
- Puede ser pequeño o grande
- Logra un objetivo puntual para el usuario

En su forma inicial (básica) el caso de uso es creado a partir de una entrevista o conversación con el cliente que solicita la funcionalidad del sistema o por el usuario que solicita el sistema: Se aborda cada requerimiento en forma discreta (puntual), se le da un nombre y un texto descriptivo breve que indique su objetivo.

Los Caso de Uso más generales aparecen en la etapa de elaboración, pero no se desesperen, ya que en las siguientes etapas se pueden ir complicando, digo, completando y dando más valor a cada caso de uso que realicen. :-)

Cada caso de uso nace a partir de **interacciones con el sistema** que realiza el usuario. Por ejemplo, veamos lo que hace la función de Formato de Texto que trae Microsoft Word como un sistema (conocido como Style Sheets), Primero definamos las interacciones que tiene:

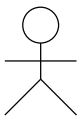
- Cambiar Tipografía
- Definir Estilo de Texto
- Usar Estilo de Texto
- etc.

En general, estas interacciones definen lo que el usuario hace con el sistema, pero no el objetivo que el usuario trata de conseguir usando la herramienta. Los verdaderos **Objetivos del Usuario** se describirán como "Dar Formato al Texto" y "Utilizar Formato en otros Párrafos".

Es así como ambos casos pueden representarse como Casos de Uso, pero en instancias diferentes de problemas. En el caso de interacciones, el caso de uso sirve para planificación. En los objetivos del usuario pueden verse distintas alternativas para solucionar los problemas de manera más amplia, que con el primer caso muchas veces se restringe.

Diagramas de Casos de Uso

Los diagramas de casos de uso son representaciones gráficas que nos permiten dar forma a los distintos casos de uso que participan en el sistema. Para la notación, los casos de uso utilizan 2 elementos básicos que procederemos a explicar:



Actores: Los actores, generalmente representados por personajes alargados (como dibujos infantiles), son utilizados para representar al usuario, cuando desempeña ese papel con respecto al sistema.

Un actor en general representa solo un ROL dentro de la especificación dentro del sistema, es decir, representa a un personaje general y no a una persona física. Muchos actores pueden representar a la misma persona física (por ejemplo el actor **Profesor Auxiliar** (que hace las clases auxiliares) y **Profesor Evaluador** (que corrige los controles) en general son personificados a veces por las mismas personas, es decir, el profesor auxiliar siempre tiene que tomar el papel de evaluador después de un control.

Siempre es más fácil identificar los Casos de Uso a través de los actores, pues es más sencillo inicialmente plantearse los objetivos para cada actor (que obviamente uno los puede conocer a priori cuando se enuncia el problema) y luego enumerar cada uno de los Casos de Uso que responden a esos objetivos por actor. Si tratáramos a priori de listar todos los casos de uso, es probable que lo primero que ocurra es que se nos quede uno en el tintero.

Una particularidad de un actor es que no es necesario que sean seres humanos, es decir, un actor perfectamente puede ser otro sistema informático relacionado.

Veamos un ejemplo sencillo: Representar el diagrama de casos de uso para un sistema PAC (Pago Automático de Cuentas) de una nueva empresa Internet llamada **Págate.cl**.

En la primera entrevista con ejecutivos de la empresa podemos obtener:

- Los clientes que utilizan el sistema son usuarios registrados.
- Si un cliente desea ser usuario debe registrarse en nuestro sistema ingresando sus datos personales.
- Para el pago de una cuenta, se consulta el saldo al sistema de consulta de Transbank.

En estas poquitas líneas (que en general no son más de 5 a 10 minutos de conversación) ya podemos identificar a los actores y algunos de los objetivos. Veamos:

- **Usuario**: Persona real
- **Cliente**: Persona real. Puede ser el mismo usuario (ES un usuario)
- **Transbank**: Sistema Informático de Consulta de Saldos (SICS)

Como ya podemos ver tenemos 3 actores distintos que se pueden representar por una persona única y un sistema computacional (y no 3 personas como quedan en el diagrama).

A modo de regla general, los actores que se muestran en el diagrama serán solo aquellos que necesiten de un caso de uso, y no necesariamente que interactúen con él. Por ejemplo, en el caso del SICS nombrado anteriomente, será considerado como Actor solo por el hecho que requiere una transacción a realizar de pago del sistema para cargar a la cuenta corriente o tarjeta de crédito, y no por la consulta de saldos que es la importante a realizar.

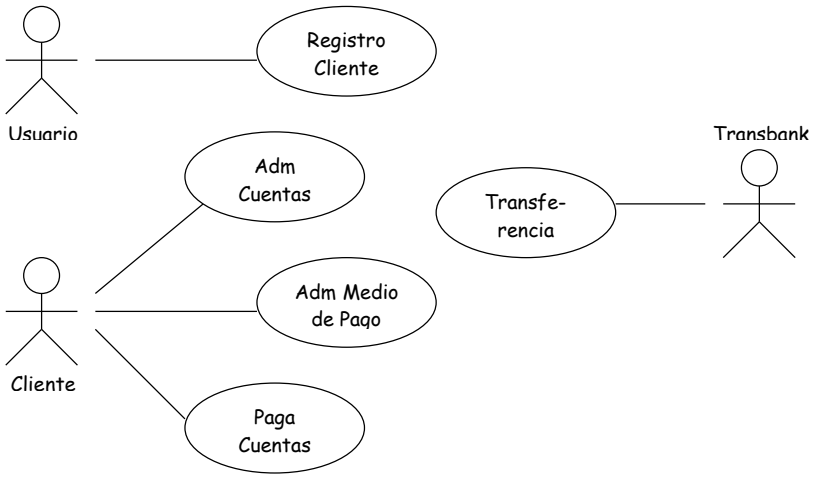
Casos de Uso: Son identificados por elipses con un texto dentro y representan un objetivo o una interacción del usuario necesaria con el sistema que se está desarrollando.

Todos los casos de uso tratan de funcionalidades requeridas externamente (no internamente). Si el SICS necesita de una transacción para procesar el cargo a la cuenta corriente o a la tarjeta de crédito, ese se convierte en un requerimiento que se debe satisfacer.

Veamos como identificar los casos de uso del ejemplo de **Págate.cl**. Tomando el mismo enunciado, listemos primero los objetivos de usuario por actor a satisfacer:

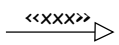
- **Usuario**:
  - Registra como Cliente
- **Cliente**:
  - Administra de Cuentas
  - Administra Medios de Pagos
  - Paga de Cuentas
- **Transbank**:
  - Realiza Transferencia

Ya conocemos los objetivos de los actores, dibujemos nuestro primer caso de uso:



Al ver este diagrama, es claro notar que, al mostrárselo a un no especialista informático se entienden las interacciones que tendrá el sistema con el exterior, aún cuando no posea ninguna especificación de software detrás.

Pero ¿satisface lo que el software realmente hará al final?. La respuesta es que tal vez no, porque es solo una comprensión de los objetivos que se deben cumplir para que el software sea exitoso. Tal vez existan muchos casos de uso que no hemos considerado, y que cumplirán un rol importante, pero eso lo iremos refinando a través del proceso de desarrollo.



Uses y Extends: Los uses (o usa) y extends (o extiende) se representan con una flecha de punta blanca y un texto (use o extend). Indican las relaciones que existen entre 2 casos de uso del sistema.

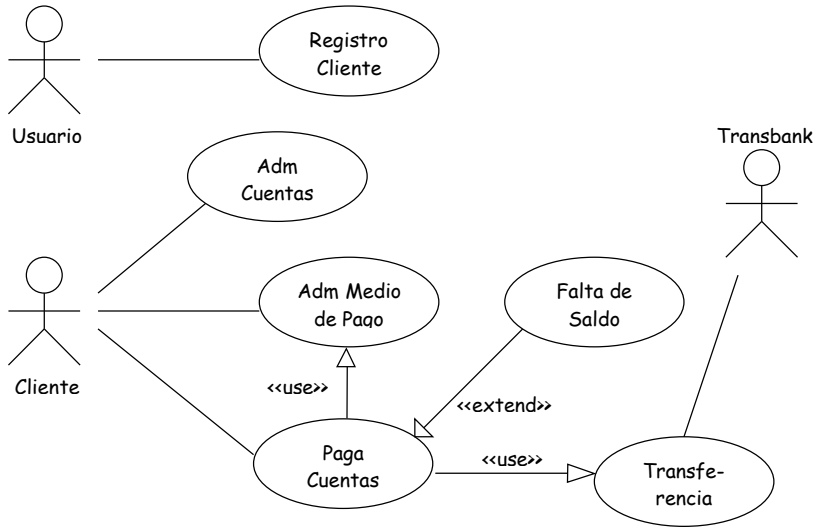
Con mucha frecuencia se confunden ambos conceptos, pero es lo normal y detallamos a continuación cada uno:

- **Extend**: se usa cuando se tiene un caso de uso que es similar a otro, pero que hace un poco más que él. Lo más comparable con esto es cuando se extiende una clase (herencia) en Java, ya que, en este caso, el caso de uso "hereda" el comportamiento de su padre.

- **Use**: se usa cuando se tiene una porción del comportamiento de un caso de uso en otro y no se quiere duplicar esa conducta. Al igual que el anterior, podemos parearlo con la utilización de método que llaman a otros que poseen esa cualidad, ya que no se desea copiar dicha información en los métodos, porque es siempre el mismo proceso.

En nuestro ejemplo, podemos identificar estas relaciones.

En el caso de un **extend** podemos ver la situación cuando Transbank rechaza la transacción por falta de fondos. En tanto, la relación de **use** aparece cuando para realizar un pago, debe ingresarse un medio de pago y además realizar la transacción con Transbank:



A modo de resúmen:

- Usa relaciones **extends** cuando escriba una variación de una conducta normal (casos especiales, condiciones de borde, otras utilizaciones).

- Usa relaciones **uses** para evitar repeticiones de casos de uso ya realizados (conjuntos de conductas más pequeñas o generalizaciones).

Escenario: Un término que se asocia con los casos de uso, y que muchas veces se confunde con ellos, es el término de escenario. Un escenario corresponde a una sola ruta de proceso en un caso de uso (por ejemplo, el proceso de Pago de Cuenta). Es por eso que decimos que un caso de uso se compone de muchos escenarios.

Realización: En la jerga de UML también utilizamos este concepto para nombrar a las distintas formas de presentar un caso de uso. Muchos diseñadores de software hacen varias realizaciones de un mismo caso de uso para ponerlo en discusión y tomar la mejor alternativa.

Es muy común que los diseñadores hagan los casos de uso con los usuarios, ya que son ellos los que identifican sus propias necesidades más fácilmente. Otros profesionales que pueden hacer un caso de uso rápidamente son los psicólogos y comunicólogos, quienes pueden identificar las necesidades de los usuarios en una entrevista, haciendo las preguntas exactas y escuchando entre frases (o entre líneas).

Problema

Modele el Caso de Uso que responde a esta situación:

> Un sistema de administración de notas para primer año se puede realizar en Java de muchas formas. Este proyecto es muy importante, ya que permitiría una publicación automática y consulta por los alumnos de manera sencilla.
>
> Considere las siguientes características:
>
> - Los evaluadores ingresan las notas de cada alumno en las preguntas del control.
> - Los alumnos pueden revisar sus notas y publicar un reclamo para una re-revisión por los evaluadores.
> - Si posee un reclamo, el evaluador tiene por misión re-corregir. Un aviso vía correo electrónico le llega con una lista de los usuarios que reclamaron.
> - La secretaria docente ingresa al sistema para obtener un listado con las notas de los alumnos para publicarlas en el fichero.
> - Los profesores de cátedra pueden ver estadísticas de notas y los promedios de cada alumno. Ellos también pueden corregir notas en caso de que el reclamo se realice directamente a ellos.

Diagramas de Interacción

Definición

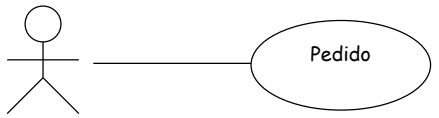
> *Modelos que describen la manera en que colaboran grupos de objetos para cierto comportamiento, captando el comportamiento de un Caso de Uso.*

El diagrama se compone de distintos objetos y mensajes que actúan en un proceso de un Caso de Uso. Muestra cierto número de ejemplos de objetos y mensajes que se pasan entre estos objetos dentro del caso de uso.

Ilustremos con la siguiente especificación de caso de uso:

1. El usuario inicia un nuevo pedido.
2. La Ventana Entrada de Pedido envía un mensaje "prepara" a Pedido.
3. El Pedido envía entonces un mensaje "prepara" a cada Línea de Pedido dentro del Pedido
4. Cada Línea de Pedido revisa el Artículo de Inventario correspondiente:
   - 4.1. Si esta revisión devuelve "verdadero", la Línea de Pedido descuenta la cantidad apropiada de Artículo de Inventario del almacén. Si la cantidad restante es más baja que el stock de reorden, genera un nuevo Artículo de Reorden para aumentar el stock del Articulo de Inventario.
   - 4.2. De lo contrario, no se realiza el pedido.

En donde el Caso de Uso quedaría como:

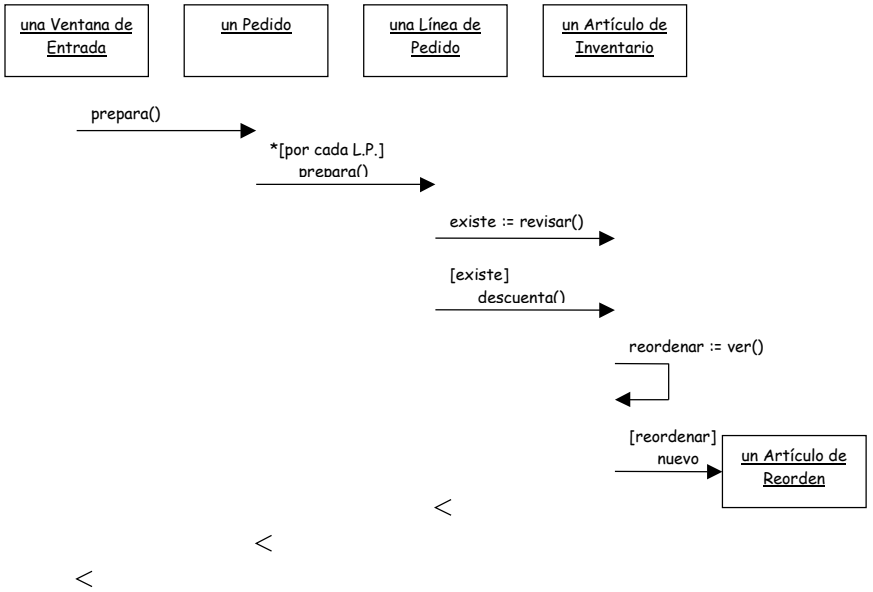


Los diagramas posibles de interacción pueden ser de 2 tipos, que analizaremos cada uno por separado:

Diagrama de Secuencia

Los diagramas de secuencia muestran cómo los objetos interactúan entre sí con el envío de mensajes y condiciones. En si da una secuencia de cómo se irá ejecutando (paso a paso) el sistema para que los objetos cumplan su finalidad dentro del caso de uso.

Analicemos el diagrama se secuencia del caso de uso que dimos de ejemplo:



Este diagrama de secuencia representa básicamente lo que está especificado como caso de uso.

Cada línea vertical se llama **Linea de Vida** del objeto y representa la vida del objeto durante la interacción.

Cada **Mensaje** se representa por una flecha entre las líneas de vida de dos objetos. El orden en el que se dan estos mensajes transcurre de arriba hacia abajo y cada uno es etiquetado por lo menos con el nombre del mensaje. Existen distintos tipos de mensajes:

(a) Mensajes en donde se especifica el nombre son como las llamadas a métodos. Estos se pueden especificar también los argumentos para que quede más claro, pero no es estrictamente necesario.

(b) Iteraciones que se indican con un * antes de la condición de iteración, por ejemplo, "**[por cada L.P.] prepara()*"significa que por cada Línea de Pedido en un pedido llamara a prepara de esa Línea de Pedido.

(c) Asignaciones a atributos como por ejemplo "*existe := revisar()*" permite guardar en existe un valor de verdad si es que existe o no inventario de ese artículo.

(d) Condiciones que se especifican con las condiciones entre corchetes y la acción bajo ellas, como por ejemplo "*[existe] descuenta()*" que significaba que si el atributo existe es verdadero, envía el mensaje descuenta().

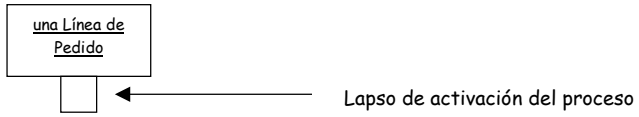
(e) Autodelegación que es sencillamente cualquier mensaje que pueda hacerse hacia el mismo objeto por ejemplo, "*reordenar := ver()*".

(f) Creación que se especifican con la palabra **nuevo** en caso de crear otro objeto que no exista hasta ahora en la interacción.

(g) Regreso que se indica con una flecha punteada hacia atrás indicando que la última llamada retorna. Hay algunos regresos que no se especifican, porque su llamada es puntual, es decir, llama y retorna en seguida.

Existen también unos diagramas de secuencia especiales para la sincronización de procesos concurrentes o en paralelo, que se parecen mucho a los anteriores, pero ingresan nuevos elementos. Por ejemplo:

Activación: Se especifica como un rectángulo a lo largo de la línea de vida en el momento en que el proceso está corriendo (está procesando o trabajando en algo). En los lugares en que no se está procesando, la línea de vida aparece punteada.



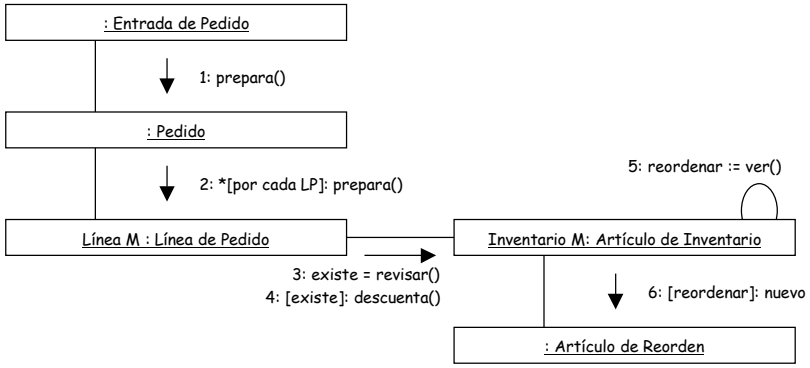
Mensaje Asíncrono: Indican a través de una flecha, con media cabeza abajo, aquellos mensajes que se envían a otros objetos o procesos que correrán en paralelo (sin bloquear al invocador).

Borrado: Mostrado como una X bajo la línea de vida de un proceso, indica cuando ese proceso u objeto deja de existir a través de un retorno o simplemente por finalizarse a si mismo.

Diagrama de Colaboración

Los diagramas de colaboración son otra forma de ver los diagramas de interacción, pero que significan casi lo mismo. La idea es ver los objetos en extenso (sin líneas de vida) pero en donde las interacciones nos muestran los flujos y los mensajes entre ellos.

Veamos el ejemplo del Pedido para que quede bien claro como va cambiando el mismo diagrama pero de distintos puntos de vista.



En este caso se observa como "colaboran" los objetos que participan en la interacción. Las flechas indican los mensajes y las líneas las interacciones. Además, cada uno de los mensajes está etiquetado debidamente numerado para indicar el orden en el cual son emitidos.

Referente a las etiquetas, una variante de ellas es utilizar para cada uno de los objetos un nivel de profundidad de un esquema numerado. Es así como los mensajes quedan individualizados por objeto y no por el conjunto completo del diagrama.

Un último detalle es que los nombres de los objetos están individualizados como *Nombre del Objeto : Nombre de la Clase*. Es posible omitir el nombre del objeto solo dejando el nombre de la clase y manteniendo los dos puntos (:) delante de ella, si es necesario.

La utilización de estos diagramas de interacción puede ser diversa, y dependiendo de la situación es importante utilizar uno de secuencia o de colaboración. En muchos casos el de secuencia es mucho más ordenado y riguroso en el orden de los mensajes, pero si se desea un esquema de interacción entre los objetos, se recomienda el de colaboración.

Pero la complejidad es un punto muy relevante. Siempre es importante que estos diagramas sean sencillos y no demasiado sobrecargados y considere siempre que es solo para representar un Diagrama de Casos de Uso y no un conjuntos de ellos.

Diagramas de Estados

Un siguiente paso en UML es analizar lo que son los Diagramas de Estado. Antes de esto definiremos primero lo que es un Estado bajo el contexto de UML:

Estado

> *Grupo de características definidas de un objeto que pueden cambiar solo a través de una acción (cambio de estado).*

Esta definición tan "de diccionario" nos ayuda a comprender y nivelar que entendemos por estado. Por ejemplo: Con la acción de encender una ampolleta (estado apagada) se pasa a un modo de iluminación (estado encendida) de la ampolleta.

El estado posee una serie de características o **Variables de Estado** que nos permiten controlar y/o saber que se involucra en ese estado. También posee una serie de **Actividades** que representan la misión que el objeto cumple en ese estado. Todo esto se ilustra con un rectángulo redondeado en las puntas y con las variables y actividades dentro.

Existen 3 tipos de actividades:

- **entry** - actividades que se realizan al entrar al estado
- **exit** - eventos que se ejecutan al abandonar el estado
- **do** - actividades realizadas mientras se esta en el estado

Diagrama de Estados

> *Es un diagrama que ilustra las transiciones entre los distintos estados que el objeto puede tomar.*

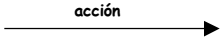
El diagrama de estado entonces se puede construir a partir de un conjunto de estados posibles que puede poseer un objeto en especial. Por ejemplo los distintos estados de una ampolleta (encendido/apagado/quemado) y cómo se pasa de un estado a otro.

La notación que se utiliza en los diagramas se basa en los estados (antes ya nombrados), las transiciones y los estados incial y final.

Estado Inicial: Es el comienzo del diagrama y se ve como un estado sin importancia, pero útil para el momento de saber cuál es el primer estado. Su representación es un círculo relleno.

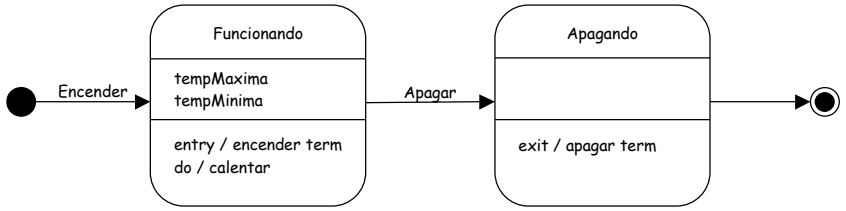
Estado Final: Al igual que el estado inicial, este estado solo se utiliza para saber cuándo termina el diagrama (o cuál es el último estado antes de finalizar el funcionamiento). Su representación es un "ojo de toro" (del inglés "bull eye").

Transición: Es simplemente el traspaso entre un estado a otro. Se representa con una flecha y una etiqueta opcional que le indica la acción manual que produjo ese cambio de estado.



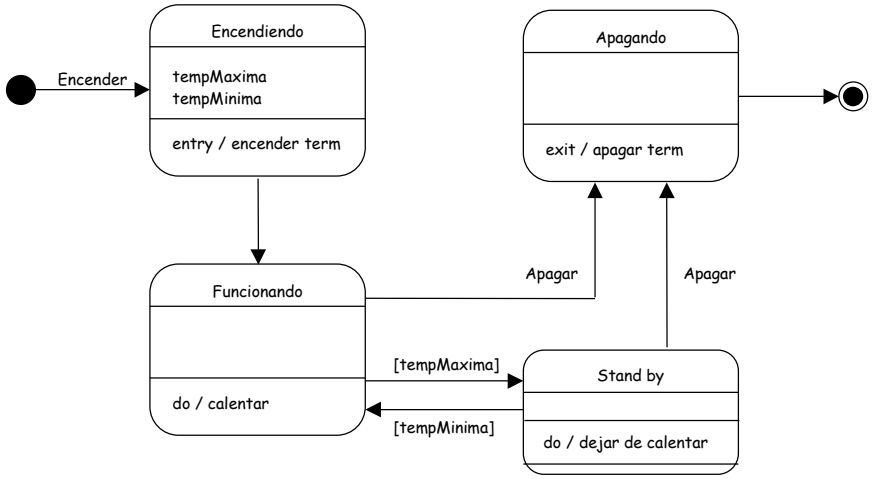
La transición puede ser expresada como un mensaje o como una condición (entre corchetes).

Por ejemplo, veamos el sencillo caso del funcionamiento de un calefactor eléctrico:



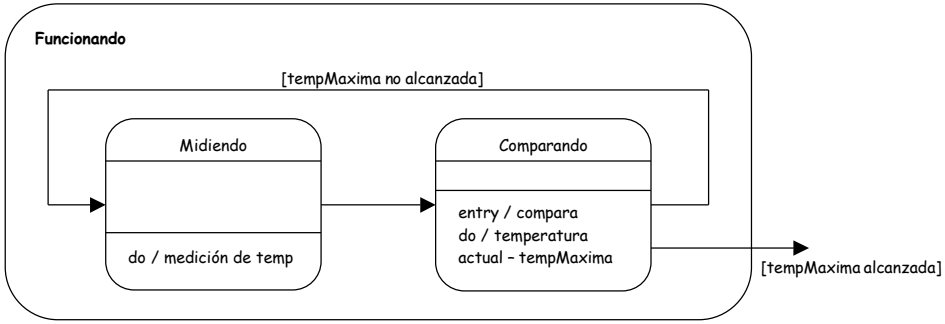
Como podemos ver, en el primer estado, el calefactor necesita de 2 variables de estado para su funcionamiento y que son tempMaxima y tempMinima. La idea que ese es el margen de temperaturas en el cual el calefactor va a mantener calefaccionado. Además, posee unas actividades en cada estado que son encendido del termostato y apagado del termostato, porque en ese estado, el calefactor debe cumplir esas actividades para funcionar mejor.

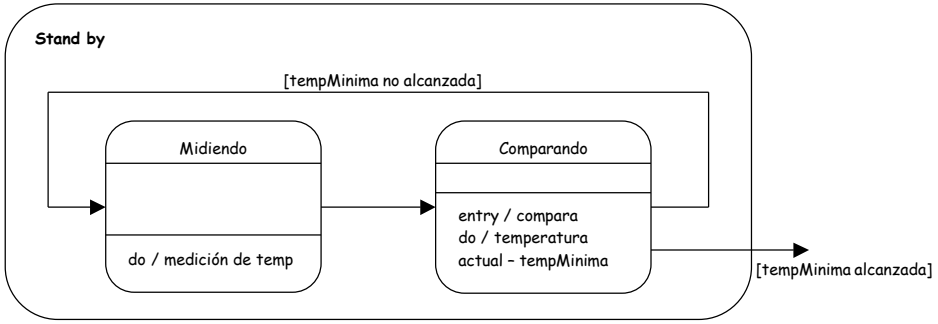
Ahora bien, este sencillo diagrama nos ilustra básicamente los estados que el calefactor puede tomar. Pero dentro del estado "Funcionando" se puden encontras más estados, ya que el calefactor se apaga (stand by) cuando está muy caluroso (tempMaxima) y se enciendo cuando la temperatura vuelve a bajar del mínimo (tempMinima). Veamos como podemos extender el diagrama para abarcar estas condiciones.



En este nuevo diagrama se ilustra más detalladamente cómo es el proceso de funcionamiento del calefactos eléctrico. Podemos ver que existe una validación en las transiciones que van desde Funcionando a Stand by, ya que esas transiciones se realizar si se ha alcanzado esa temperatura máxima (de Funcionando a Stand by) o se ha llegado a la temperatura mínima (de Stand by a Funcionando).

Con un análisis simple podemos darnos cuenta que aún quedan algunos estados "internos" que explicar y es como se validan y obtiene las temperaturas el termostato para encender o apagar el calefactor. Para ello utilicemos subestados dentro de Funcionando y Stand by:





Con estos diagramas ya se tiene bien claro el funcionamiento del objeto que se deseaba analizar. También, se ha dejado lo suficientemente explicitado cómo se hacen los diagramas de estado.

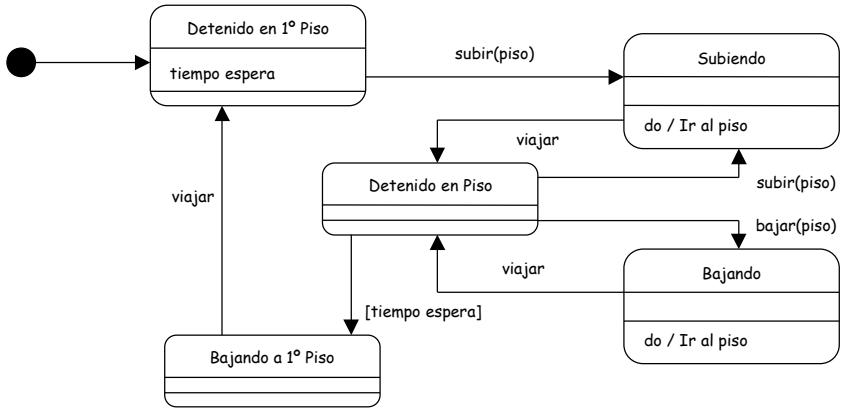
Utilización

Es necesario realizar diagramas de estado, porque ellos ayudan a los analistas, diseñadores y desarrolladores a entender el comportamiento de los objetos en un sistema. Los programadores en particular, deben suponer que hacen los objetos para poder implementarlos en el software. No es suficiente implementar los objetos solo con las especificaciones de lo que hace.

Problema

A través de diagramas de estado, modele el comportamiento de un ascensor. Suponga que el ascensor comienza siempre en el primer piso. Además, el ascensor debe volver siempre al primer piso pasado un cierto tiempo.

Solución



Diagramas de Clase

A partir de un "buen" diseño de casos de uso (diagramas) se pueden identificar las interacciones y los objetivos que, claramente el usuario quiere del sistema que se desea desarrollar. Entonces, ahora que entendemos al cliente, ¿cuál es el siguiente paso para entregar o realizar el desarrollo del software pedido?

> *Los **Diagramas de Clase** describen los tipos de objetos que hay en el sistema y las diversas clases de relaciones estáticas que existen entre ellos.*

En esta definición semi-formal podemos encontrar los fundamentos de lo que son los diagramas de clase, pero en realidad qué son y para qué sirven lo podremos ir vislumbrando a partir de un desarrollo más tangible que de la teoría computacional en Ingeniería de Software.

Clase

> *Una **Clase** es una representación de una unidad, perteneciente al sistema, y que realiza algunas interacciones o relaciones con el entorno (otras clase o actores externos al sistema).*

Como hemos visto a través del curso, es muy claro que algunos objetos que son tangibles en la vida real pueden ser "modelados" a través de lo llamado como **Clase**. Es bastante sencillo realizar el mismo paralelo con las clases que se utilizan en los diagramas de clases.

Nombre de la Clase
Atributos
Operaciones

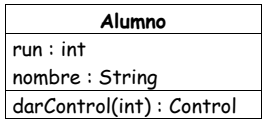
En el diagrama son representados por cajas de texto con 3 tipos de campos: el **Nombre de la Clase** que la identifica, los **Atributos** que son características que definen a la clase y las **Operaciones** que representan las funcionalidades a las que responde o que realiza la clase.

El Nombre de la Clase identifica rápidamente una clase directa en el lenguaje de programación. Por ejemplo en Java identifica inmediatamente la clase que se debe empezar a programar.

En el caso de los Atributos, también es claro que pasan a ser las variables de instancia de la clase. Muchas veces se especifica el tipo de los atributos indicando con 2 puntos después de su nombre. Por ejemplo: "**nombre : String**" indicaría que el atributo *nombre* es un *String*.

Las operaciones también son representadas en la codificación como las funciones o métodos de la clase. En este caso (y de hecho así es la notación) podemos darnos cuenta que las operaciones se anotan con un paréntesis vació después de su nombre si no posee parámetros. En el caso de poseer alguno, estos son especificados entre los paréntesis. También, y al igual que los atributos, las operaciones pueden tener un tipo de retorno, el cual se especifica con 2 puntos después de su "firma".

Un ejemplo del paralelo que hay entre el diagrama y Java es:



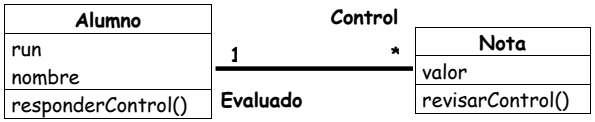

```
public class Alumno {
      public int run;
      public String nombre;

      public Control darControl(int x) { ... }
}
```

Relaciones Estáticas

> *Las **Relaciones Estáticas** son vínculos entre las clases del modelo y que indican la interacción y uso en el desarrollo del sistema.*

Existen 2 tipos de relaciones en un diagrama de clases:

Asociaciones: Representan relaciones entre instancias o conceptuales entre las clases. Son representadas por una recta que va desde una de las clases relacionadas hasta la otra clase relacionada.



Como se puede ver, una asociación posee varios elementos:

- Etiqueta: Es un texto que indica el tipo de asociación que existe entre las clases.
- Multiplicidad: Es un indicador que permite identificar la cantidad de objetos que participarán en la relación dada. Si la multiplicidad dice * (asterisco) indica que son muchos elementos.

Las multiplicidades más generales son:

A —1— B Un objeto de clase A se asocia siempre con un objeto de clase B

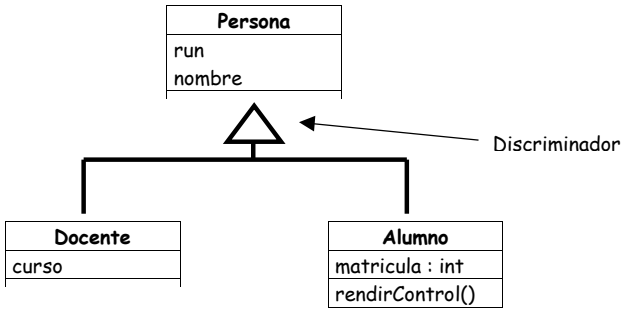
A —1..*— B Un objeto de clase A siempre se asocia con uno o más objetos de clase B

A —0..1— B Un objeto de clase A siempre se asocia con ninguno o con un objeto de clase B

A —*— B Un objeto de clase A siempre se asocia con ninguno, uno o más objetos de clase B

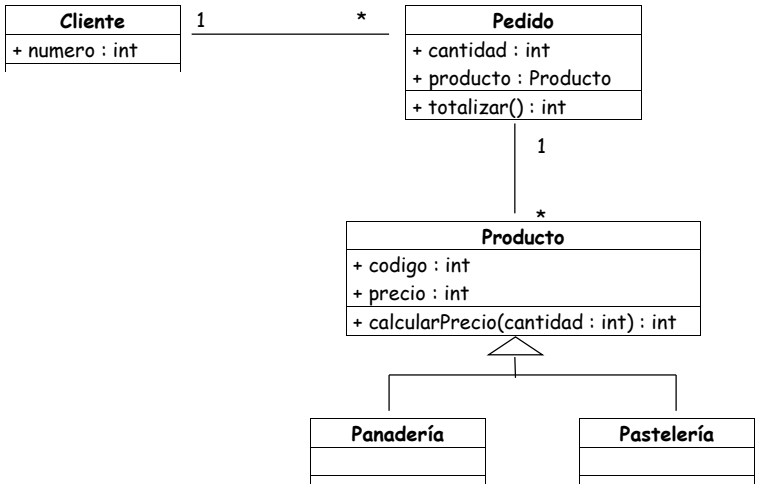
A —m..n— B Un objeto de clase A siempre se asociarán con un número entre m y n de objetos de clase B

Subtipos: Son relaciones que nos permiten indicar cuando una clase es subtipo de otra (caso especial de la clase padre). Se representan con una rectas que unen a la superclase (clase padre) y la subclase o subtipo (clase hijo) con un triángulo en la punta de la clase padre.



El discriminador indica que cada una de las instancias del supertipo que se esté utilizando corresponde a una y solo una instancia de las de los subtipos, aún cuando un supertipo puede tener muchos subtipos (cada uno desunido entre si, es decir, independientes).

Un ejemplo que utiliza todo lo anterior sería el modelar las clases para un software de procesamiento de pedidos en una panadería:



Sintaxis

Veamos algunas generalidades sobre la sintaxis para que quede más claro.

Existen varios elementos que contribuyen a la sintaxis de los atributos y operadores de las clases en el diagrama. Estas son:

Visibilidad: Indica que tan visible será el elemento. Esto puede ser + (público), # (protegido), - (privado). Es lo mismo que en Java llamábamos como privacidad.

Nombre: El nombre del elemento es una cadena de caracteres.

Tipo: El tipo corresponde al tipo del elemento en cuestión.

Valor por Omisión: Es el valor por defecto que debe tener ese elemento en caso de ser inicializado.

La sintaxis de los Atributos es:

<visibilidad> **<nombre>** : <tipo> = <valor por omisión>

siendo solo el nombre un parámetro obligatorio.

La sintaxis para una Operación es:

<visibilidad> **<nombre>** (**<parámetros>**) **<tipo de retorno>**

siendo el nombre, los parámetros y el tipo de retorno los obligatorios. En el caso de la sintaxis de los parámetros, estos se representan de la misma manera que los atributos de una clase.

Otros Conceptos

Navegabilidad

La navegabilidad se aplica en torno a lo que son las asociaciones. Las asociaciones pueden tener una "dirección" que nos indica qué objetos tienen la responsabilidad de decir a que clases corresponden y quienes no:

En este ejemplo indica que un Pedido debe de decir a qué Cliente corresponde. Esto es muy útil para la implementación, porque nos indicará en donde declararemos las instancias de la clase asociada. Sin embargo, no es necesario especificarlas, ya que eso indicaría una navegabilidad no definida (modelos conceptuales).

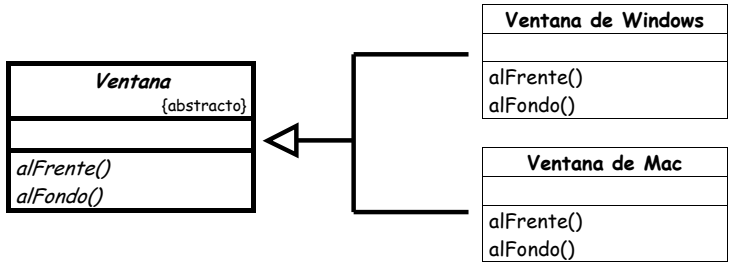
Reglas de Restricciones

Este elemento, cuya sintaxis se especifica como textos flotantes encerrados en llaves { ... } en torno a las clases, nos indican afirmaciones que deben cumplirse (algo como los if) dentro de esas clases.

Un ejemplo sería que "Si el pedido.cliente.estadoCrédito() es 'Pobre' entonces pedido.prepagado debe ser verdadero" en donde estadoCrédito() es un método de Cliente y prepagado es una variable de instancia.

Clases Abstractas

Las clases abstractas en UML se representan por dibujos rectangulares igual que en el caso de las clases, pero cuyo texto es en cursivas:

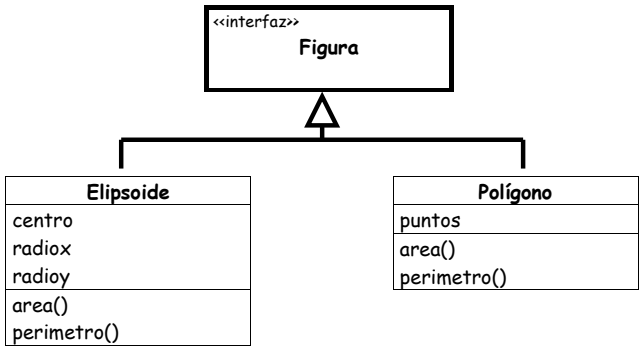


Se utiliza una clase abstracta al igual como se definen en Java: Una "generalización" de una o más clases que comparten algunos atributos o funcionalidades. Es por eso que la "herencia" en UML es conocida como "generalización".

Interfaces

Las interfaces en UML también representan directamente lo que significan en los lenguajes de programación como Java: Una caparazón en la cual se encapsulan las funcionalidades de un grupo o tipo de clase. También es conocido como tipo, ya que los objetos que se instancias son del tipo de la interfaz, pero su enlace dinámico es a una clase específica (que implementa la interfaz).

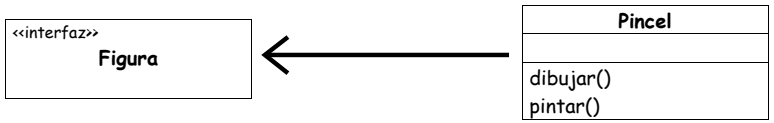
La "implementación" de las interfaces en UML se le llama **refinamiento** y se dibuja igual que la generalización, pero con líneas punteadas.



Dependencia

El concepto de dependencia es un tipo de asociación que se utiliza cuando existen interfaces o clases abstractas, y es una asociación que indica cuando otra clase utiliza de alguna forma una clase abstracta o una interfaz.

Su notación es una flecha punteada que va desde la clase que utiliza el elemento hasta el elemento del cual depende.



En este ejemplo, **Pincel** depende de los objetos dinámicos que fueron creados a partir de refinamientos de la interfaz **Figura**.

Perspectiva

Existen muchas formas de aproximarse a una representación de un sistema con un diagrama de clases y estas corresponden a 3 puntos de vista distintos que tal vez tienen puntos en común.

Perspectiva Conceptual

Es recomendable representar en el diagrama los conceptos del dominio que se está estudiando. Estos conceptos se relacionan de manera natural con las clases que los implementan, pero con frecuencia no hay una correlación directa. De hecho, los modelos conceptuales se deben dibujar sin importar el software con que se implementarán, por lo cual se pueden considerar como independientes del lenguaje.

En este tipo de perspectiva, la definición de atributos, operaciones y relaciones se hacen en forma bastante básica, solo para que se pueda comprender la idea y el plano general del modelo a desarrollar.

Perspectiva de Especificación

Viendo el software se puede llegar a esta perspectiva. Sin embargo, la implementación se ve a través de interfaces y no de las clases reales que se deben implementar, por lo tanto, en realidad vemos los tipos y no las clases. El desarrollo orientado a objetos pone un gran énfasis en la diferencia entre estos elementos, aunque se pase por alto con frecuencia.

Observar a través de esta perspectiva puede resultar en muchos casos algo complejo, pues tendremos que mirar "entre líneas" para llegar a la implementación.

Perspectiva de Implementación

Dentro de esta concepción, realmente tendremos clases y no interfaces ni conceptos filosóficos del sistema. Con este tipo de perspectiva, que es la más común dentro de los programadores y la más útil al momento de codificar el software, vemos directamente en bruto las especificaciones de las clases y la implementación directa.

El punto de vista global de esta perspectiva nos permite controlar mejor la implementación, pues vemos en ellos (en los diagramas) las interacciones y relaciones entre las instancias físicas de cada una de nuestras clases.

Es altamente recomendado un diseño como éste si es necesario controlar una gran cantidad de clases que se pueden convertir en inmanejables al momento de codificarlas.

Cuándo y Como usar Diagramas de Clases

Los diagramas de clases son la columna vertebral de casi todos los métodos OO. El problema radica en que éstos diagramas son tan ricos que pueden llegar a ser complicados y abrumadores. Pero, si hablamos de receta de cocina, tenemos algunos consejos prácticos para realizar buenos diagramas y en el momento y lugar adecuado:

- No trate de usar toda la notación a su disposición. Empiece siempre con la notación más sencilla definiendo las clases, asociaciones, atributos, funcionalidades y al final la generalización o herencia. Existe más notaciones que son útiles solo si se necesitan, pero no son recomendados para empezar.

- Ajuste la perspectiva desde la cual se dibujan los diagramas dependiendo la etapa del proyecto.

  Si es en análisis, dibuje modelos conceptuales.
  Cuando trabaje con software, céntrese en modelos de especificación.
  Dibuje modelos de implementación cuando este mostrando una técnica de implementación en particular.

- No dibuje modelos para todo, por el contrario, céntrese en las áreas clave del sistema. Es mejor usar y mantener al día unos cuantos diseños que tener muchos modelos olvidados y obsoletos.

El problema de los diagramas es que van empantanando los detalles de implementación. Para contrarrestar esto, siempre es útil usar las perspectivas conceptuales y de especificación para no "casarse" con la implementación directamente.

Problema

Se desea modelar usando UML un juego de Poker. El Poker se juega con cartas inglesas las cuales poseen cada una un número y una pinta, todas distintas entre sí. Cada Mano de poker posee 5 cartas (que son con las que juega cada jugador). Por supuesto que el juego no tiene sentido sin apuestas, por lo que se dispone de un monto de 10.000 por cada jugador para realizar apuestas. El juego, para simplificarlo, consiste en:

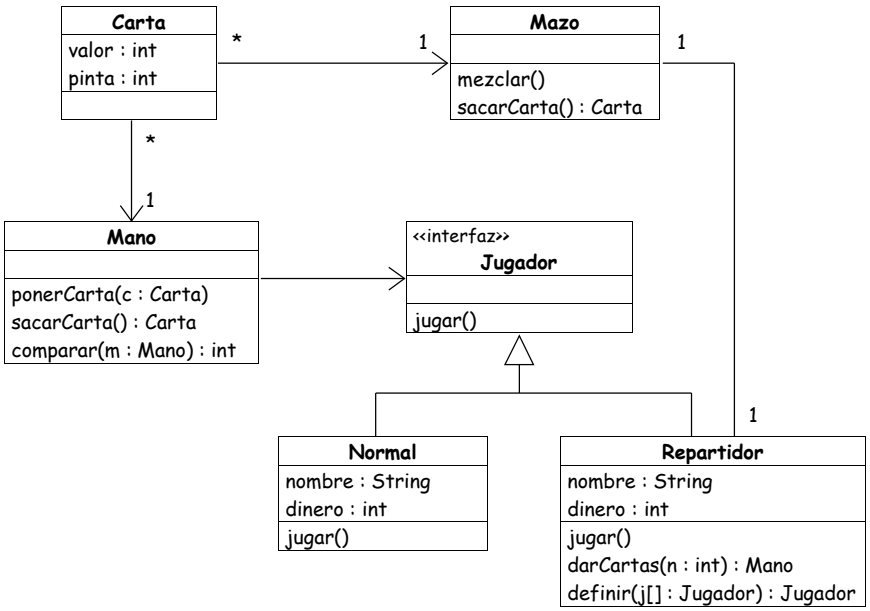
- 1 repartidor (posee el mazo con todas las cartas) y n jugadores.
- Todos dan una apuesta de 500 inicial.
- El repartidor da 5 cartas al azar a todos los jugadores (incluyéndose).
- Todos revisan sus cartas y descartan un número menor o igual a 5 cartas para reemplazarlas por otras, pagando 100 por hacer esto.

- Luego se muestran las cartas y gana el que posea mejor juego.

Dibuje el diagrama de clases de esta situación con la perspectiva conceptual.

Solución

Bueno, siempre en modelamiento existe más de una solución, pero es normal, pues depende del punto de vista que tenga el profesional que se encuentre modelando. Siempre es importante tener en cuenta las perspectivas que se utilizan y también que un diagrama de clases debe ilustrar el modelo que el software tendrá.



Referencias

En este documento se utilizó el siguiente material:

Libros

- Introducción a Java
- UML Gota a Gota (UML Destilled) de Fowler y Scott

Sitios Web

- Online API of Java 2 Platform SE v1.3.1 (http://java.sun.com/j2se/1.3/docs/api)
- Online API of Java 2 Platform SE v1.2.2 (http://java.sun.com/j2se/1.2/docs/api)
- Online API of Java 1.1.x (http://java.sun.com/products/jdk/1.1/docs/api/packages.html)
- Java homepage (http://java.sun.com)
- Tutorial de Java (http://sunsite.dcc.uchile.cl/SunSITE/java/docs/JavaTut/index.html)
- JBuilder (http://www.dcc.uchile.cl/~lmateu/JBuilder/index.html)
- Clases Console y ConsoleCanvas (http://www.holtsoft.com/java/hsa_package.html)
- RDBMS mySQL (http://www.mysql.com/downloads/mysql-3.23.html)
- Información sobre Tarjetas CRC (http://www.c2.com/doc/oopsla89/paper.html)
- Tutorial UML en 7 días (http://odl-skopje.etf.ukim.edu.mk/UML-Help/)
- Tutorial UML de Patricio Salinas (http://www.dcc.uchile.cl/~psalinas/uml)
- Más información sobre UML (http://www.cs.ualberta.ca/~pfiguero/soo/uml/)