Unidad III

Estrategias de Resolución

Año 2011

> "Pesemos la ganancia y la pérdida, considerando cara que Dios existe. Si ganáis, ganáis todo. Si perdéis, no perdéis nada. Apostad, pues, a que Dios existe, sin vacilar."
>
> Blas Pascal

Autor: Ing. Tymoschuk, Jorge

Unidad III – Estrategias de Resolución

Interacción de objetos

En la unidade anterior hemos examinado qué son los objetos y cómo se los implementa; en particular, cuando analizamos las definiciones de las clases, hablamos sobre atributos, constructores y métodos.

Ahora, iremos un paso más adelante. Para construir aplicaciones reales no es suficiente construir objetos que trabajan individualmente. En realidad, **los objetos deben estar combinados de tal manera que cooperen entre ellos al llevar a cabo una tarea en común. Esta es la estrategia que debemos estudiar, comprender y aplicar en la resolución de nuestros problemas.**

Abstracción

Cuando vimos tipo abstracto de datos en la Unidad II introducimos la idea de abstracción. Conceptualmente: La **abstracción** es la habilidad de ignorar los detalles de las partes para centrar la atención en un nivel más alto de un problema.

Problemas triviales como los vistos en esa unidad pueden ser resueltos como si fueran un único problema: se puede ver la tarea completa y divisar una solución usando una sola clase. En los problemas más complejos, esta es una visión demasiado simplista. Cuando un problema se agranda, se vuelve imposible mantener todos los detalles al mismo tiempo.

La solución que usamos para tratar el problema de la complejidad es la abstracción: dividimos el problema en subproblemas, luego en sub-subproblemas y así sucesivamente, hasta que los problemas resultan suficientemente fáciles de tratar. Una vez que resolvemos uno de los subproblemas no pensamos más sobre los detalles de esa parte, pero tratamos la solución hallada como un bloque de construcción para nuestro siguiente problema. Técnica conocida: divide y reinarás.

Imaginemos a los ingenieros de una fábrica de automóviles diseñando uno nuevo. Pueden pensar en partes tales como: su carrocería, tamaño y ubicación del motor, asientos, distancia entre ruedas, etc. Por otro lado, otro ingeniero (en realidad, este es un equipo de ingenieros pero lo simplificamos un poco en función del ejemplo), cuyo trabajo es diseñar el motor, piensa en las partes que componen un motor: los cilindros, el mecanismo de inyección, el carburador, la electrónica, etc. Piensa en el motor no como una única entidad sino como un conjunto compuesto por varias partes, una de esas partes podría ser una bujía, un pistón.

Por lo tanto, hay un ingeniero (quizás en una fábrica diferente) que diseña los pistones. Piensa en el pistón como un artefacto compuesto por varias partes. Puede haber hecho estudios complejos para determinar exactamente la clase de metal que debe usar dependiendo de la compresión, revoluciones, etc.

El mismo razonamiento es válido para muchas otras partes del automovil. Un diseñador del nivel más alto pensará una rueda como una única parte; otro ingeniero ubicado mucho más abajo en la cadena de diseño pasará sus días pensando sobre la composición química para producir el mejor material para construir los neumáticos. Para el ingeniero de los neumáticos, el neumático es algo complejo. La central automovilística comprará los neumáticos a otra fábrica (u varias) los verá como un componente y esto es la abstracción.

Por ejemplo, el ingeniero de la fábrica de automoviles hace abstracción de los detalles de la fabricación de los neumáticos para concentrarse en los detalles de la construcción de una rueda. El diseñador que se ocupa de la forma del coche se

abstrae de los detalles técnicos de las ruedas y del motor para concentrarse en el diseño del cuerpo del coche (le importará el tamaño del motor y de las ruedas).

El mismo argumento es cierto para cualquier otro componente. Mientras que algunas personas se ocupan de diseñar el espacio interior del coche, otros trabajan en desarrollar el tejido que usarán para cubrir los asientos. El punto es: si miramos detalladamente un automovil, está compuesto de tantas partes que es imposible que una sola persona conozca todos los detalles de todas las partes al mismo tiempo. Si esto fuera necesario, jamás se lo hubiera construido.

La razón de que se construyen exitosamente es que los ingenieros usan abstracción y **modularización:** dividen el coche en módulos independientes (rueda, motor, asiento, caja de cambios, etc.) y asignan grupos de gente para trabajar en cada módulo por separado. Cuando **construyen un módulo usan abstracción:** ven a ese módulo como un **componente** que se utiliza para **construir componentes más complejos.**

Presentamos el concepto de **modularización** en la unidad II. Ahora lo usaremos.

Concepto: La modularización es el proceso de dividir un todo en partes bien definidas que pueden ser construidas y examinadas separadamente, las que interactúan de maneras bien definidas.

La modularización y la abstracción se complementan mutuamente. La modularización es el proceso de dividir cosas grandes (problemas) en partes más pequeñas, mientras que la abstracción es la habilidad de ignorar los detalles para concentrarse en el cuadro más grande.

Abstracción en software

Los mismos principios de modularización y de abstracción discutidos en la sección anterior se aplican en el desarrollo de software. En el caso de programas complejos, para mantener una visión global del problema tratamos de identificar los componentes que podemos programar como entidades independientes, y luego intentamos utilizar esos componentes como si fueran partes simples sin tener en cuenta su complejidad interna.

En programación orientada a objetos, estos componentes y subcomponentes son objetos. Si estuviéramos tratando de construir un programa que modele un auto mediante un lenguaje orientado a objetos, intentaríamos hacer lo mismo que hacen los ingenieros: en lugar de implementar el coche en un único objeto monolítico, primeramente podríamos construir objetos independientes para un motor, una caja de cambios, una rueda, un asiento, pistones, etc., y luego ensamblar el objeto automovil partir de esos objetos más pequeños.

No siempre resulta fácil identificar qué clases de objetos debe tener un sistema de software que resuelve determinado problema. Pero no pongamos el carro delante de los caballos, comenzaremos con ejemplos simples.

Como un motor de automóvil, por ejemplo, los programas modernos se forman por componentes distintos que deben interactuar en forma correcta, para que todo el sistema funcione bien. Casa componente debe funcionar adecuadamente. En la POO, el concepto de *modularidad,* se refiere a una organización en la que distintos componentes de un sistema de programación se dividen en unidades funcionales separadas.

Graficando un poco nuestro ejemplo del motor, podremos tener una clase base Motor, de allí **extendemos** Motor Eléctrico, Combustión Interna, etc … Por otro lado, el motor de combustión interna **tiene** módulos (O componentes, u objetos) tales como Carburador, Bomba de Inyección, Radiador, etc, etc … que no pertenecen a la estructura jerárquica del motor. Casi seguramente pertenecen a otra. Decimos entonces que la clase Motor necesariamente **tiene objetos** Carburador, Radiador… a esta relación la podemos llamar **"tiene un":** El objeto Motor **tiene un** objeto de otra clase.

Finalmente, distinguiremos un tercer caso, cuando una clase necesita transitoriamente, por ejemplo dentro de alguno de sus métodos, del comportamiento de objetos de otra clase. En este caso, el objeto de nuestra clase no necesita del objeto de la otra al nivel de atributo, o sea permanente. Lo necesita transitoriamente. A esta relación la podemos llamar **"usa un"**

Relaciones entre objetos

En resumen, los tipos de relación que estudiaremos son los siguientes, en este orden:

Tipo de relación	Descripción
"tiene un"	El objeto de nuestra clase tiene atributos que son objetos de otras clases
"usa un"	Métodos pertenecientes al objeto de nuestra clase requieren del comportamiento de otras clases
"es un"	El objeto de nuestra clase es una extensión o especialización de la clase de la cual hereda. Esto lo vemos al final de esta unidad.

Comencemos con un ejemplo.

Primero crearemos la clase **Habitacion**, luego definiremos una clase **Hotel**, y lo haremos **usando objetos** Habitación.

```
public class Habitacion {
    private int numHabitacion;
    private int numCamas;

    public Habitación               //    (1) Constructor   sin argumentos
        numHabitacion   = 0;
            numCamas          = 0;
      }

    public Habitacion( int numHab,int numCam) { //    Sobrecarga del constructor
        numHabitacion   = numHab;
        numCamas        = numCam;
    }

    public Habitacion(Habitacion hab) {         //    mas       sobrecarga       del
constructor
        numHabitacion   = hab.numHabitacion;
        numCamas        = hab.numCamas;
    }

    public int getNumHab() {                    //    Siguen métodos get y set
        return numHabitacion;
    }

    public int getNumCam() {
              return numCamas;
    }

    public void setNumHab (int numHab){
        numHabitacion =  numHab;
    }

    public void setNumCam (int numCam){
```

```
        numCamas  = numCam;
    }
}
```

El código anterior sería el corazón de la aplicación. Vamos pues a construir nuestro **Hotel** creando Habitaciones y asignándole a cada una de ellas su número.

```
public class Hotel1 {
   public static void main( String args[] ) {
      Habitacion hab1, hab2;         // Definimos referencias a habitaciones
      hab1 = new Habitacion(1,2);    // Instanciamos
            hab2 = new Habitacion(2,3);
            Habitacion hab3, hab4;             // dos nuevas referencias
      hab3 = new Habitacion(hab2);  // Tiene la capacidad de la hab2
      hab4 = new Habitacion(hab2);  // lo mismo
      hab3.setNumHab(3);            //  Su número correcto es 3
      hab4.setNumHab(4);            //  Su número correcto es 4
      Habitacion hab5, hab6;        //  seguimos ampliando el hotel,
      hab5 = new Habitacion();      //  le incorporaremos
      hab6 = new Habitacion();      //  dos habitaciones mas ...
    }
}
```

Nótese que la clase **Hotel1** no tiene objetos propios, (No tiene ningún atributo); **usa** los de la clase **Habitación**. En realidad funciona como un demo del comportamiento de ella. Esto no deja de limitar bastante en lo que respecta a que se puede hacer con esta clase. Si quisiéremos tener un método que nos permita obtener un listado de las habitaciones con la cantidad de camas que tienen, tendremos dificultades. Ocurre que las habitaciones que estamos declarando y **usando** son locales a main(), no miembros de la clase **Hotel1**. Si quisieramos tener esta capacidad, la clase Hotel debe definir el tipo abstracto Habitación como atributo de la clase.

Veamos más ejemplos de problemas reueltos mediante interacción de objetos.
Para ello definamos algunos conceptos de sucesiones, progresiones, series, secuencias.

De la consulta de algunos autores (Goodrich/Tamassia, Estructuras de Datos y Algoritmos en Java) y diccionarios podemos ofrecer las siguientes definiciones:

Sucesión: Un término después de otro. No hay una ley de vinculación entre ellos. Tipicamente necesitamos del término actual.

Progresión, serie: Un término después de otro. Existe una ley de vinculación entre ellos. Tipicamente necesitamos del término actual y el anterior.

Secuencia: Un término después de otro. No hay una ley de vinculación predefinida entre ellos. En cualquier momento se debe disponer de todos o cualquiera de sus terminoos.

Comenzaremos tratando problemas relativamente sencillos que utilizan sucesiones, progresiones, series o secuencias de algún objeto(caracteres, números).

Composición usando una sucesión de objetos Número

Enunciado: Producir una sucesion de números aleatorios y listarlos. El listado debe concluir con el primer número divisible inclusive.

Si, teniendo presente la modularidad, analizamos "sucesion de números aleatorios" vemos que estamos en presencia de dos conceptos:

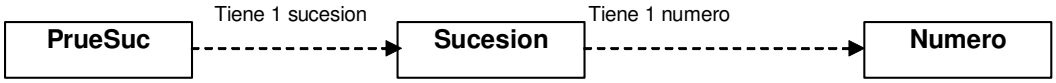
El concepto **Numero:** asociado a el tenemos el comportamiento de la generación aleatoria y la divisibilidad.

El concepto **Sucesion:** Es responsable de producir la sucesión de números y de detener esta producción cuando el número generado no es divisible.

O que sea tendremos dos clases, y cual será la relación entre los objetos de ellas?

Es inmediato que un numero **no** "es una" sucesion, ni viceversa.
Un número tiene una sucesion? **No**
Una sucesion tiene números? **Si**

Entonces la clase **Sucesion** "tiene un" objeto **Número.**




```
package secunum;
import java.io.*;
class Numero{
      private int valor;      // valor del numero
           private int divisor;     // valor del divisor
      public Numero(int divisor){  // Genera el valor del numero aleatoriamente
            this.divisor = divisor;
      }
      public void setValor(){  // Genera el valor del numero aleatoriamente
            valor=(int)(100*Math.random());
      }
           public int getValor(){  // retorna el valor del numero
            return valor;
      }

           public boolean multiplo(){
            boolean retorno = false;
            if (valor % divisor == 0) // es multiplo
                  retorno = true;
            return retorno;
      }

      public String toString(){           //   Retorna hilera con informaci¢n del
objeto
            String aux = "Numero ";
            if(multiplo()) aux += "divisible ";
            else         aux += "no divis. ";
            aux += valor;
            return aux;
      }
}  // fin clase numero.

package secunum;
class Sucesion{
      Numero numero;    // Referencia a Numero

      public Sucesion(int divisor){  // Constructor
            numero = new Numero(divisor);
      }


      public void proSucesion(){
            do{
```

```
run:
Deseo procesar una sucesion
de numeros divisibles por
2
Numero divisible 36
Numero divisible 60
Numero no divis. 75
BUILD SUCCESSFUL (total time: 5 seconds)
```

```
                numero.setValor();
                System.out.println(numero);
            }while(numero.multiplo());
      }

}
```

```
package secunum;
public class Main{
      public static void main(String args[]){
            System.out.println("Deseo procesar una sucesion ");
            System.out.println("de numeros divisibles por ");
            int divisor = In.readInt();
            Sucesion suc = new Sucesion(divisor);  // Instanciamos la sucesion
            suc.proSucesion();            //  y la procesamos ...
      }
} // Main
```

Composición usando una sucesión de objetos Caracter

Enunciado: Procesar una sucesión de caracteres. Informar cuantos de ellos son vocales, consonantes y dígitos decimales. El proceso concluye con la lectura del carácter '#' (numeral)

Si, teniendo presente la modularidad, analizamos "sucesión de caracteres" vemos que estamos en presencia de dos conceptos:

El concepto **caracter**: asociado a el tenemos el comportamiento de la deteccion de que caracteres son letras, dígitos, mayúsculas, etc, etc.

El concepto **sucesión**: Es responsable del ciclo de lectura y la detección de su fin. Y como es en el ciclo de lectura donde se detectaran quienes son vocales, consonantes, aquí los contabilizaremos.

Tenemos dos clases. La clase **Sucesión tiene un objeto Carácter**

Clase Caracter

El idioma Ingles considera letras a menos caracteres que el español. No tiene acentuadas, etc. Por eso, y para tener mas comportamiento "amistoso" con nuestras necesidades definimos una **clase Caracter** …

```
import java.io.IOException;
public class Caracter{
   private int car;              // Parte interna de la clase, nuestro caracter

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

   Caracter(int cara){car = cara;};    // Constructor, inicializa car mediante
     parámetro

   boolean esLetra(){            //  Retorna  true  si  verdadero,  false  caso
     contrario
     boolean letra = false;
     if(esLetMay()) letra=true;
     if(esLetMin()) letra=true;
     return letra;      //  Retornemos lo que haya sido
   } //  esLetra()

   boolean esLetMay(){           //  Es letra mayúscula ?
     boolean mayus = false;
     String mayu = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
    for(int i=0;i < mayu.length();i++)
        if(mayu.charAt(i)==car) mayus = true;  //  Es una letra mayúscula
    return mayus;
  }

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

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

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

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

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

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

  boolean pertPal(){  //  Caracter pertenece a la palabra?
    boolean pertenece = false;
    if(esLetra() || esDigDec())
       pertenece = true;
    return pertenece;
  }

  boolean lecCar() throws java.io.IOException{  //  Lectura de caracteres con
    detección de fin
    car = System.in.read(); // leemos caracter
    if(car=='#') return false;
    return true;
```

```
      }

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


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

import java.io.*;
import Caracter;
public class SuceCar{
      private     int vocal, conso, digDec; //  Contadores
      private Caracter car;
      public SuceCar(){        // Constructor
            vocal = 0;conso = 0; digDec = 0;
            car = new Caracter();
            System.out.println ("\nIntroduzca una sucesi¢n de ");
            System.out.println ("caracteres, finalizando con #\n");
      }

      public void proSuc() throws java.io.IOException{
            while (car.lecCar()){ // Mientras la lectura no sea de un caracter '#'
                  if(car.esVocal()) vocal++;
                  if(car.esConso()) conso++;
                  if(car.esDigDec())digDec++;
            }
            System.out.println (this);          // Exhibimos atributos del objeto

      }

      public String toString(){                 //  Metodo para exhibir el objeto
            String aux = " \n";                 //  Una l¡nea de separacion
            aux += " Totales proceso \n";       //  Un titulo
            aux += "Vocales      " + vocal + "\n";
            aux += "Consonantes  " + conso + "\n";
            aux += "Digitos dec. " + digDec+ "\n";
            aux += "Terminado !!!\n";
            return aux;
      }
};
```

```
Introduzca una sucesi¢n de
caracteres, finalizando con #

Aqui me pongo a cantar ...#
Totales proceso
Vocales      9
Consonantes  9
Digitos dec. 0
Terminado !!!
Process Exit...
```

```
import SuceCar;
import java.io.IOException;
public class PruebaSuc{
      public static void main(String args[]) throws IOException{
            SuceCar suc = new SuceCar();  // Construimos el objeto suc de la clase
Sucesion
            suc.proSuc();                 // Procesamos ese objeto
      }
};
```

Composición usando una progresión de caracteres

Enunciado: Procesar una progresión de caracteres. Necesitamos conocer cuantas alternancias consonante/vocal o viceversa ocurren. El proceso concluye con la lectura del carácter '#' (numeral).

Porque hemos cambiado de **Sucesión para Progresion**?
Porque, para detectar alternancia, necesitamos de dos objetos Carácter, no uno. El anterior y el actual. De acuerdo a nuestra convención, **Progresión** es el concepto que debemos modelar.

```
import Caracter;
class  Progresion{
      private int siAlt, noAlt;          //  Contadores de alternancias y no alt.
      private Caracter carAnt, carAct;
      public Progresion(){               //  Constructor,
            siAlt=0; noAlt=0;                  //  Contadores a cero
            carAnt =  new Caracter();
            carAct =  new Caracter();
            System.out.println("\nIntroduzca caracteres, fin: #");
      }

      public void cicloCar(){
            carAnt.setCar(carAct.getCar());     //  carAnt ← carAct
      }

      public boolean exiAlter(){               // Existe alternancia ?
            boolean alter = false;
            if((carAnt.esConso() && carAct.esVocal()) // Si la hubo asi
            || (carAnt.esVocal() && carAct.esConso())) // o de esta otra manera
                  alter = true;
            return alter;     // contabilizaremos que no
      }

      public void proAlter() throws java.io.IOException{
            do{
                  carAct.lecCar();
                  if(exiAlter()) siAlt++;       // Detectamos alternancia
                  else           noAlt++; // No la hubo
                  cicloCar();
            }while (carAct.getCar() != '#');    // Mientras no sea '#'
            System.out.println(this);
      }

      public String toString(){
            String aux = "\n Totales \n";
            aux += "Alternan " + siAlt + " \n";
            aux += "No alter " + noAlt + " \n";
            aux += "Terminado !!!\n";
            return aux;
      }
};
```

Documentando

En el método **public void cicloCar()** tenemos la expresión

carAnt.setCar(carAct.getCar());

A primera vista es bastante criptica, analicemos:

El objeto **carAnt** invoca el método **setCar(…)**, quien recibe com argumento el retorno de la la expresión encerrada entre paréntesis **carAct.getCar()**.

La expresión entre paréntesis se ejecuta primero, así lo ordena la jerarquía de prioridades de cualquier lenguaje. El objeto **carAct** invoca el método **getCar().** Si vemos la codificación entendemos que lo que se está haciendo es retornando el atributo **int car** del objeto invocante **carAct** . Este es el retorno que recibe como argumento el método **setCar(),** quien lo usa para valuar el atributo de su invocante, objeto **carAnt.**

O sea que todo lo que hacemos es valuar el atributo de un objeto con el de otro.
Como ese atributo es el único del objeto, podemos decir que estamos igualando los dos objetos. A esa operación la llamamos **cicloCar():** estamos valuendo el termino anterior de la progresión con el actual, para luego compararlo con el próximo.

– No sería mas sencillo, para los mismos efectos, hacer simplemente ***carAnt = carAct*** *y listo?*

Lamentablemente no lo es. Ocurre que estamos igualando dos referencias, o sea direcciones de memoria, a los objetos caract. Lo que la expresión hace es que ambos elementos apunten al objeto **carAnt**, y entonces el método exiAlter() jamas detectaría ninguna alternancia, estaría siempre comparando el anterior con si mismo.

– de acuerdo, pero podria funcionar se hacemos ***carAnt.car = carAct.car?***

Buena pregunta. Veamos que ocurre. Reemplazamos el método por la expresión que Ud propone, compilamos y

```
public void cicloCar(){
    carAnt.car = carAct.car;
}
```

```
utput | Build | Find in Files |
E:\jdk1.3\bin\javac.exe  Progresion.java
Working Directory - E:\Tymos\Catedras\AED2005\Unidad II
Class Path - .;E:\Kawa4.01\kawaclasses.zip;e:\jdk1.3\li
File Compiled...

---------------------------- Compiler Output -----------
Progresion.java:17: car has private access in Caracter
          carAnt.car = carAct.car;
                       ^
Progresion.java:17: car has private access in Caracter
          carAnt.car = carAct.car;
                               ^
2 errors
```

El compilador nos informa que **car es inaccesible**. Correcto, en la clase carácter lo hemos declarado privado. Estamos en otra clase, no lo podemos acceder. Podriamos "suavizar" el encapsulado y retirarle a car el nivel private. Pasaría a ser friendly, veamos que pasa. Recompilamos Carácter, Progresion …

Compilación OK. La ejecución tambien. El tema de encapsulamiento es una cuestión de criterio de diseño. Es como la seguridad. Cuantas más llaves, contraseñas, alarmas ponemos en una oficina, por ejemplo, todo se complica. Ademas se "ralentiza", mas lento, hay mas llamadas a métodos… Claro que si dejamos todo a nivel de acceso

public un día encontraremos nuestra oficina vacía … hay que pensar en una seguridad razonable

```
Introduzca caracteres, fin: #
Es la ultima copa ...#
Totales
Alternan 9
No alter 12
Terminado !!!
Process Exit...
```

```
import Progresion;
import java.io.IOException;
class PrueProg{
        public static void main(String args[]) throws IOExcept
                Progresion pro = new Progresion();
                pro.proAlter();
        }
}
```

Mas una composición usando progresión de objetos Caracter

Tratamiento de frases
(Cuantas palabras, cuantos caracteres, longitud promedio de la palabra)

Enunciado: Procesar una progresión de caracteres. Necesitamos conocer cuantas palabras contiene, cuantos caracteres en total y la longitud promedio de la palabra. El concluye con la lectura del carácter '#' (numeral).

Si consultamos el diccionario, veremos que dice de frase: Conjunto de palabras que tienen sentido. Nosotros no vamos a ser tan avanzados, solo vamos a considerar que las palabras, además de letras, pueden contener dígitos decimales. Por supuesto, no contendrán signos de puntuación. Los signos de puntuación, más los espacios en blanco separan las palabras.

O sea que tenemos frases constituidas por palabras y palabras por caracteres. El comportamiento de la clase Caracter está relacionado con todo lo relativo al carácter: esLetra, esVocal, etc, (Ya la usamos)

Que comportamiento debemos modelar para detectar palabras? Mínimamente debemos poder detectar si el carácter que hemos acabado de leer pertenece a la palabra que estamos procesando o si la palabra ha concluido. Consideremos la frase:

El prof**e**, ... dice el alumno, ... es un burro#

Supongamos que en este momento nuestro último carácter leído es la letra **'e'** (negrita). Es una letra, pertenece a la palabra. No sabemos cual será el próximo, leamos ... Usemos la variable **car** para la lectura.

El profe, ... dice el alumno, ... es un burro#

Ahora tenemos **car** conteniendo una ',' (coma). No es letra ni dígito, entonces no pertenece a la palabra, nos está indicando que la palabra ha terminado. Si nuestro propósito es contar palabras, es el momento de hacerlo. Sigamos leyendo.

El profe, ... dice el alumno, ... es un burro#

Ahora tenemos **car** conteniendo un '**.**' (punto). No es letra ni dígito, no pertenece a la palabra, pero tampoco podríamos decir que nos esté indicando que la palabra ha terminado. Ocurre que la palabra terminó en el carácter anterior. Descubrimos que no es suficiente considerar el contenido de **car**, una buena idea sería **definir una estrategia** basada en que **ocurre un fin de palabra cuando el carácter actual no pertenece a la palabra, pero el anterior si**(Es letra o dígito)**.**

```
package frase;
public class Frase{
     private Caracter carAct, carAnt;
     private int contCar, contPal;
     public Frase(){                //  Constructor,
```

```
            carAct = new Caracter(); //   instanciamos sus
            carAnt = new Caracter(); //   objetos atributos
            contCar = contPal = 0;
            System.out.println("\nIntroduzca caracteres, fin: #\n");
      }

      public void cicloCar(){
            carAnt.setCar(carAct.getCar()); //  Anterior <== Actual
      }

      public boolean finPal(){       // Es fin palabra?
            boolean fPal = false;
                  if  (carAnt.pertPal() && !carAct.pertPal())
                        fPal = true;
            return fPal;
      }



       public void proFrase() throws java.io.IOException{   //      lectura      de
caracteres
            do {
                  carAct.lecCar();
                  if (carAct.pertPal()) contCar++;
                  if (finPal())    contPal++;
                  cicloCar();
            }while(carAct.getCar() != '#');     // Mientras no sea un '#'
            System.out.println(this);
       }

       public String toString(){                // la exhibición de los resultados
            float       auxPro = (float)contCar/contPal;
            String aux = "\n Totales \n";
            aux += "Procesamos frase con " + contPal + " palabras,\n";
            aux += "constituidas por     " + contCar  + " caracteres, \n";
            aux += "su longitud promedio " + auxPro  + " caracteres \n";
            aux += "Terminado !!!\n";
            return aux;
       }
      }
```

```
package frase;
import java.io.IOException;
class Main{
       public static void main(String args[])
                        throws IOException{
            Frase frase = new Frase();
            frase.proFrase();
       }
}
```

```
run:

Introduzca caracteres, fin: #

to be or not to be, that is the ...#

 Totales
Procesamos frase con 9 palabras,
constituidas por     22 caracteres,
su longitud promedio 2.4444444 caracteres
Terminado !!!
```

Agrupar objetos

Cuando escribimos programas, frecuentemente necesitamos agrupar objetos del mismo tipo, para procesarlos. Buscar el menor, mayor, promedio, etc., etc. pueden ser situaciones de este tipo. Necesitamos de una **colección.**
Por ejemplo:

. Las agendas electrónicas guardan notas sobre citas, reuniones, fechas de cumpleaños, etc.

. Las bibliotecas registran detalles de los libros y revistas que poseen.

. Las universidades mantienen registros de la historia académica de los estudiantes.

En esta tarea de agrupar objetos Java nos permite organizarlos usando muy distintas herramientas y hay que estudiar bastante antes de decidir cual de ellas es la más adecuada a nuestra necesidad.

Una primera guía para esta definición es si el número de elementos almacenados en la colección varía a lo largo del tiempo. Veamos ejemplos

- en una agenda electrónica se agregan nuevas notas para registrar eventos futuros y se borran aquellas notas de eventos pasados en la medida en que ya no son más necesarios; en una biblioteca, el inventario cambia cuando se compran libros nuevos y cuando algunos libros viejos se archivan o se descartan. Aquí lo ideal es una colección apta para almacenar un **número variable de elementos** (Objetos)
- Las palabras reservadas en un lenguaje son fijas, por lo menos mientras no cambie su versión. El compilador de Java, en el momento de compilar el código que UD está digitando, necesita disponer de esa colección, una colección que almacene **un número fijo de elementos**.

Históricamente los lenguajes siempre han sido capaces de procesar colecciones de tamaño fijo, y comenzaremos por ellas. Java ha incorporado una cantidad de clases para manipular **número variable de elementos** y veremos esto mas adelante.

Colecciones de tamaño fijo – Arreglos

Las colecciones de tamaño flexible son muy potentes porque no necesitamos conocer anticipadamente la cantidad de elementos que se almacenarán y porque es posible variar el número de los elementos que contienen. Sin embargo, algunas aplicaciones son diferentes por el hecho de que conocemos anticipadamente cuántos elementos desea almacenar en la colección y este número permanece invariable durante la vida de la colección. En tales circunstancias, tenemos la opción de elegir usar una colección de objetos de tamaño fijo, especializada para almacenar los elementos eficientemente.

Una colección de tamaño fijo se denomina **array o arreglo**.
Concepto: Un arreglo es un tipo especial de colección que puede almacenar un número fijo de elementos.

A pesar del hecho de que los arreglos tengan un tamaño fijo puede ser una desventaja, se obtienen por lo menos dos ventajas en compensación, con respecto a las clases de colecciones de tamaño flexible:

- El acceso a los elementos de un arreglo es generalmente más eficiente que el acceso a los elementos de una colección de tamaño flexible.
- Los arreglos son capaces de almacenar **objetos o valores de tipos primitivos**. Las colecciones de tamaño flexible sólo pueden almacenar objetos.

Otra característica distintiva de los arreglos es que tienen una sintaxis especial en Java, el acceso a los arreglos utiliza una sintaxis diferente de las llamadas a los métodos habituales. La razón de esta característica es mayormente histórica: los arreglos son las estructuras de colección más antiguas en los lenguajes de programación y la sintaxis para tratarlos se ha desarrollado durante varias décadas. Java utiliza la misma sintaxis establecida en otros lenguajes de programación para mantener las cosas simples para los programadores que *todavía usan arreglos*, aunque no sea consistente con el resto de la sintaxis del lenguaje.

En las siguientes secciones mostraremos cómo se pueden usar los arreglos para mantener una colección de tamaño fijo. Luego, siguiendo el orden cronológico, veremos como se tratan las de tamaño flexible.

Declaración de variables arreglos.

```
private int[ ) contadoresPorHora;
```

La característica distintiva de la declaración de una variable de tipo arreglo es un par de corchetes que forman parte del nombre del tipo: int[ ]. Este detalle indica que la variable contadoresPorHora es de tipo arreglo de enteros. Decimos que int es el tipo base de este arreglo en particular. Es importante distinguir entre una declaración de una variable arreglo y una declaración simple ya que son bastante similares:

```
int hora;

int [ ] contadoresPorHora;
```

En este caso, la variable hora es capaz de almacenar un solo valor entero mientras que la variable contadoresPorHora se usará para **hacer referencia** a un objeto arreglo, una vez que dicho objeto se haya creado. La declaración de una variable arreglo **no crea en sí misma un objeto arreglo**, sólo reserva un espacio de memoria para que en un próximo paso, usando el operador new, se cree el arreglo tal como con los otros objetos.

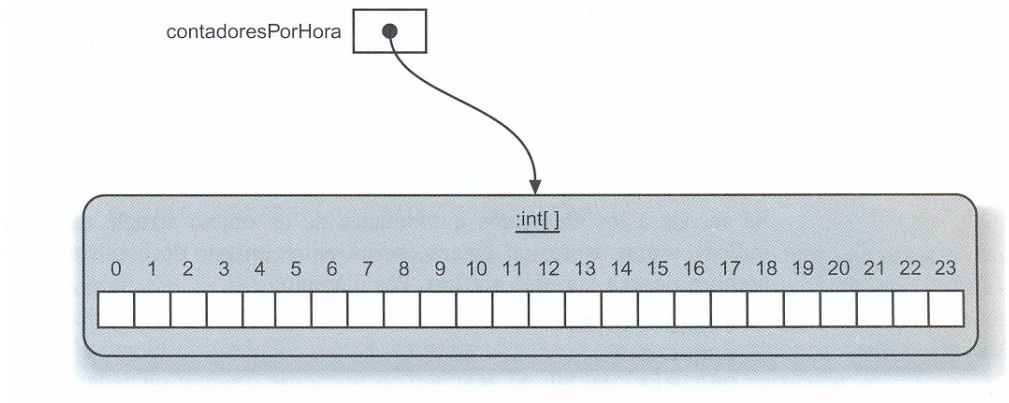
Merece la pena que miremos nuevamente la sintaxis de esta notación por un momento. Podría ser una sintaxis de aspecto más convencional tal como Array<int> pero, tal como lo mencionamos anteriormente, las razones de esta notación son más históricas que lógicas. Deberá acostumbrarse a leer los arreglos de la misma forma que las colecciones, como un «arreglo de enteros».

Creación de objetos arreglo

La próxima cuestión por ver es la manera en que se asocia una variable arreglo con un objeto arreglo.

```
contadoresPorHora = new int[24];
```

Esta sentencia crea un objeto arreglo que es capaz de almacenar 24 valores enteros y hace que la variable arreglo contadoresPorHora haga referencia a dicho objeto. La Figura muestra el resultado de esta asignación.



La forma general de la construcción de un objeto arreglo es:

```
new tipo[expresión-entera]
```

La elección del tipo especifica de qué tipo serán todos los elementos que se almacenarán en el arreglo. La expresión entera especifica el tamaño del arreglo, es decir, un número fijo de elementos a almacenar.

Cuando se asigna un objeto arreglo a una variable arreglo, el tipo del objeto arreglo debe coincidir con la declaración del tipo de la variable. La asignación a contadoresPorHora está permitida porque el objeto arreglo es un arreglo de enteros y la variable contadoresPorHora es una variable arreglo de enteros. La línea siguiente declara una variable arreglo de cadenas que hace referencia a un arreglo que tiene capacidad para 10 cadenas:

```
String [ ] nombres = new String[10];
```

Es importante observar que la creación del arreglo asignado a nombres no crea realmente 10 cadenas. En realidad, crea una colección de tamaño fijo que es capaz de almacenar 10 cadenas en ella. Probablemente, estas cadenas serán creadas en otra parte de la clase a la que pertenece nombres. Inmediatamente después de su creación, un objeto arreglo puede pensarse como vacío. En la próxima sección veremos la forma en que se almacenan los elementos en los arreglos y la forma de recuperar dichos elementos.

Usar objetos arreglo

Se accede a los elementos individuales de un objeto arreglo mediante un índice. Un índice es una expresión entera escrita entre un par de corchetes a continuación del nombre de una variable arreglo. Por ejemplo:

```
etiquetas[6];
maquinas[0];
gente[x + 10 - y];
```

Los valores válidos para una expresión que funciona como índice dependen de la longitud del arreglo en el que se usarán. Los índices de los arreglos siempre comienzan por cero y van hasta el valor uno menos que la longitud del arreglo. Por lo que los índices válidos para el arreglo contadoresPorHora son desde 0 hasta 23, inclusive.

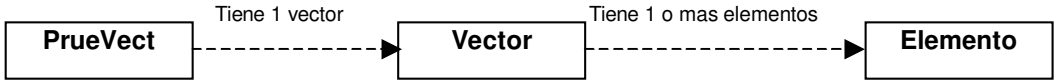
Cuidado: dos errores muy comunes al trabajar con arreglos: uno es pensar que los índices válidos de un arreglo comienzan en 1 y otro, usar el valor de la longitud del arreglo como un índice. Usar índices fuera de los límites de un arreglo trae aparejado un error en tiempo de ejecución denominado ArrayIndexOutOfBoundsException.

Las expresiones que seleccionan un elemento de un arreglo se pueden usar en cual quier lugar que requiera un valor del tipo base del arreglo. Esto quiere decir que podemos usarlas, por ejemplo, en ambos lados de una asignación. Aquí van algunos ejemplos de uso de expresiones con arreglos en diferentes lugares:

```
etiqueta[5] = "Salir";
double mitad = lecturas[0] / 2;
System.out.println(gente[3].getNombre());
maquinas [0] = new MaquinaDeBoletos (500);
```

El uso de un índice de un arreglo en el lado izquierdo de una asignación es equivalente a un método de modificación (o método set) del arreglo porque cambiará el contenido del mismo. Los restantes usos del índice son equivalentes a los métodos de acceso (o métodos get).

Composición usando un arreglo de objetos Elemento



Encontrar el menor de los elementos numéricos del vector tiene que ver con el vector en si. No es una propiedad de un Elemento aislado. Solo tiene sentido si existen otros contra los que puede comparase.

```
import In;
class Elemento{
      private int valor;         // atributos instanciables de la clase

      public Elemento(int ind){  // carga desde el teclado el valor de un numero
            System.out.print("Valor del elemento ["+ind+"]");
            valor = In.readInt();
      }

      public int getValor(){  // retorna al mundo exterior el valor del numero
            return valor;
      }
}  // fin clase Elemento.

import Elemento;
class Vector{
      Elemento element[];  // vector de objetos Elemento
      private int orden = 0, menor; // atributos no instanciables
      public Vector(int tamanio){
            element = new Elemento[tamanio]; // crea un vector de Elemento
            for(int i=0;i<element.length;i++)
                  element[i]  =  new  Elemento(i);  //  Construye  cada  uno  de  los
objetos
            menor = element[0].getValor();
      }

      public String toString(){
            int auxVal;
            String aux = "El arreglo contiene \n";
            aux       += "Ord. Val\n";
            for(int i =0;i<element.length;i++) {
                  auxVal =  element[i].getValor();    //  obtenemos el valor
                  aux += "["+i+"]  "+auxVal+"\n";     //    lo  incorporamos  a  la
hilera
                  detMenor(auxVal,i);                 //  vemos si es el menor
            }
            aux += "\ny su menor elemento es el orden " + getOrden() + "\n";
            return aux;
      }
      private void detMenor(int val, int ind){
                  if (val < menor){
                  menor = val;
                  orden = ind;
            }
      }
      private int getOrden(){
            return orden;
      }
}
```

```
Cuantos elementos tendra el vector?
5
Valor del elemento [0]10
Valor del elemento [1]20
Valor del elemento [2]30
Valor del elemento [3]9
Valor del elemento [4]33
El arreglo contiene
Ord. Val
[0]  10
[1]  20
[2]  30
[3]  9
[4]  33
y su menor elemento es el orden 3
Process Exit...
```

```
import  Vector;
public class PrueVect{
      public static void main(String args[]){
            System.out.print("\nCuantos elementos tendra el vector? ");
            int aux = In.readInt();
            Vector vect = new Vector(aux);      // Creamos el objeto VectorNum
            System.out.println(vect.toString());
      }
}
```

A continuación, un aporte de un profesor de la Cátedra, el tema de cuantos números divisibles por un dado valor, lo trata usando un vector, por eso lo intercalamos aquí.

// Aporte del Ing. Silvio Serra.

Composición usando una secuencia de números

Analizar una secuencia de 5 números retornando la cantidad de aquellos que sean múltiplos de 4

Lo primero es plantear los objetos que existen en este problema. En principio hay que identificar aquellos elementos que tienen algún comportamiento, o sea, que "harán algo" en este escenario.
Podemos identificar dos grandes actores, los **números** en si y la **secuencia**. Son dos elementos separados y no uno solo porque hacen cosas diferentes, en otras palabras, tienen "responsabilidades" diferentes.
Los números serán responsables de saber que número son y de poder informar, a quien le pregunte, características de si mismos, como por ejemplo, si es par o si es múltiplo de cuatro o no.
La secuencia es responsable de poder construir una sucesión de números y realizar cualquier procesamiento que sea necesario sobre ella, como por ejemplo, llenarla de datos, recorrerla, informar la acumulación de los componentes que sean múltiplo de cuatro y cualquier otra cosa que se pueda esperar de una secuencia.
Podemos expresar la composición de los números y la secuencia discriminando sus atributos y responsabilidades de la siguiente manera:

NUMERO	SECUENCIA
Atributos:	Atributos:
Valor	valores
	tamaño
Responsabilidades:	Responsabilidades:
Numero()	Secuencia(x)
InformarValor()	CargarValores()
CargarValor()	
Multiplo(x)	CuantosMultiplos(x)

Existe un gráfico denominado "diagrama de clases" que permite definir las clases gráficamente de una manera muy descriptiva y es fácil de usar, podía sernos de utilidad en el aula.
Si al describir un método caemos en alguna lógica compleja, puede usarse un típico diagrama de flujo para describir ese comportamiento.

¿Como podemos plantear la solución de este problema en Java?. Con la construcción de dos clases, una para representar los números y otra para representar la

secuencia, donde los atributos serán variables y las responsabilidades métodos (funciones). La secuencia tendrá (relación "tiene un") objetos numero.

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

class numero{
      private int valor;  // único atributo de la clase
      public numero(){  valor = 0;} // constructor
      public void CargarValor() {valor=In.readInt();}
      public int InformarValor() {return valor;}
      public boolean Multiplo(int x){  // retorna si el numero x es o no multiplo
            boolean retorno = false;
            if (valor % x == 0) // es multiplo
                  retorno = true;
            return retorno;
      }
}  // fin clase numero.


class secuencia{
      numero valores[];  // vector de numeros
      int tamaño;

      public secuencia(int x){  // crea un vecrtor de numeros de tamaño x
            valores = new numero[x];  // el vector de numeros
            for(int i=0;i<x;i++)
              valores[i] = new numero();
            tamaño = x;  // el tamaño del vector
      }

      public void CargarValores(){  // carga los valores de la secuencia.
            System.out.println("\nTipee 5 numeros, separados por espacios\n");
            for(int i=0;i<tamaño;i++){
                  valores[i].CargarValor();
            }
      }

      public int CuantosMultiplos(int x){
            int retorno = 0;
            for(int i=0;i<tamaño;i++){
                  if(valores[i].Multiplo(x)==true)   // el numero es multiplo
                        retorno++;
            }
            return retorno;
      }
}     // clase secuencia
```

Una ejecucion

```
Tipee 5 numeros, separados por
espacios

10 20 30 40 50
La cantidad de números multiplos
de 4 es 2
Process Exit...
```

Otra

```
Tipee 5 numeros, separados por espacios

123 345 567 789 444
La cantidad de números multiplos de 4 es 1
Process Exit...
```

Se ejecuta mediante un main() contenido en la siguiente clase

```
public class Main{
      public static void main(String args[]){
            secuencia UnaSecuencia = new secuencia(5);  // 5 objetos secuencia
            UnaSecuencia.CargarValores();  // Carga desde teclado toda la secuencia
            int cantidad;
            cantidad = UnaSecuencia.CuantosMultiplos(4); // cantidad de números
                                                   // multiplos de 4 de
UnaSecuencia.
            System.out.println("La cantidad de números multiplos de 4 es
      +cantidad);
```

```
        }
}
```

Obviamente esta no es la única forma de resolver este problema, probablemente existen otras formas de plantear un modelo que cumpla con la consigna inicial.
Podemos asegurar que se escribe bastante mas código para resolver un problema si lo comparamos con el paradigma estructurado, pero esto es muy notable solo en problemas simples y pequeños como este, de alguna manera, la infraestructura que rodea a la solución es un poco pesada para problemas tan sencillos. La programación Orientada a Objetos adquiere mas fuerza en problemas complejos.
No obstante lo dicho en el párrafo anterior, tenemos que tener en cuenta que aquí hemos hecho mucho mas que cargar una serie y decir cuantos son múltiplos de cuatro, lo que hicimos es generar dos objetos que pueden ser reutilizados en cuanto programa nos resulte necesario y que responderán a los mensajes que se les envíe independientemente del contexto en que se encuentren. En otras palabras, escribimos más código para resolver este problema simple, pero ahora tenemos una altísima posibilidad de reutilizar lo que hemos hecho. Más código ahora, se transforma en mucho menos código después.

// Fin Aporte del Ing. Silvio Serra.

Agrupar objetos en colecciones de tamaño flexible

A continuación usaremos el ejemplo de una agenda personal para ilustrar una de las maneras en que Java nos permite agrupar un número arbitrario de objetos en un único objeto contenedor.

Una agenda personal

Características básicas:

. Permite almacenar notas sin límite de cantidad.

. Mostrará las notas de manera individual.

. Nos informará sobre la cantidad de notas que tiene almacenadas.

Podemos implementar todas estas características muy fácilmente si tenemos una clase que sea capaz de almacenar un número arbitrario de objetos (las notas). Una clase como ésta ya está preparada y disponible en una de las bibliotecas que forman parte del entorno estándar de lava.

Una de las características de los lenguajes orientados a objetos que los hace muy potentes es que frecuentemente están acompañados de bibliotecas de clases. Estas bibliotecas contienen, comúnmente, varios cientos o miles de clases diferentes que han demostrado ser de gran ayuda para los desarrolladores en un amplio rango de proyectos diferentes. Java cuenta con varias de estas bibliotecas y seleccionaremos clases de varias de ellas a lo largo del libro. Java denomina a sus bibliotecas como paquetes (packages). Podemos usar las clases de las bibliotecas exactamente de la misma manera en que usamos nuestras propias clases: las instancias se construyen usando la palabra new y las clases tienen campos, constructores y métodos. En la clase **Agenda** haremos uso de la clase ArrayList que está definida en el paquete java.util;

ArrayList es un ejemplo de una clase colección.
Las colecciones pueden almacenar un número arbitrario de elementos en el que cada elemento es otro objeto.

```
import java.util.ArrayList;
/**
* Una clase para mantener una lista arbitrariamente larga de notas.
* Las notas se numeran para referencia externa.
* La numeración de las notas comienzan en 0.
*/
package agenda;
import java.util.ArrayList;
public class Agenda{
    private ArrayList<String> notas;
    public Agenda (){   //    Constructor
        notas = new ArrayList<String>();
    }
    public void guardarNota (String nota){notas.add(nota);}

    public void eliminarNota (int numeroDeNota){
        notas.remove(numeroDeNota);
    }
    public int numeroDeNotas (){return notas.size();}

    public String mostrarNota(int numeroNota){
        return notas.get(numeroNota);
    }
}
```

```
package agenda;
public class Main {
    public Main(){}
    public static void main(String[] args) {
        Agenda agenda = new Agenda();
        agenda.guardarNota("Cargar Celular");
        agenda.guardarNota("Tel. Pedro");
        agenda.guardarNota("Mañana continuamos");
        System.out.println("Mi agenda contiene "+agenda.numeroDeNotas()+"
                                                              notas");
        System.out.println("La segunda dice "+agenda.mostrarNota(1));
        System.out.println("Demo terminado");
    }
}
```

Analizamos algunas sentencias.

`private ArrayList<String> notas;`

Cuando usamos colecciones, debemos especificar dos tipos: el tipo propio de la colección (en este caso, ArrayList) y el tipo de los elementos que planeamos almacenar en la colección (en este caso, String). Podemos leer la definición completa del tipo como «ArrayList de String». Usamos esta definición de tipo como el tipo de nuestra variable notas.

En el constructor de la agenda, creamos un objeto de tipo ArrayList<String> y guardamos dentro de él nuestro campo notas. Observe que necesitamos especificar nuevamente el tipo completo con el tipo de elemento entre los símbolos de menor y de mayor, seguido de los paréntesis para la lista de parámetros (vacía):

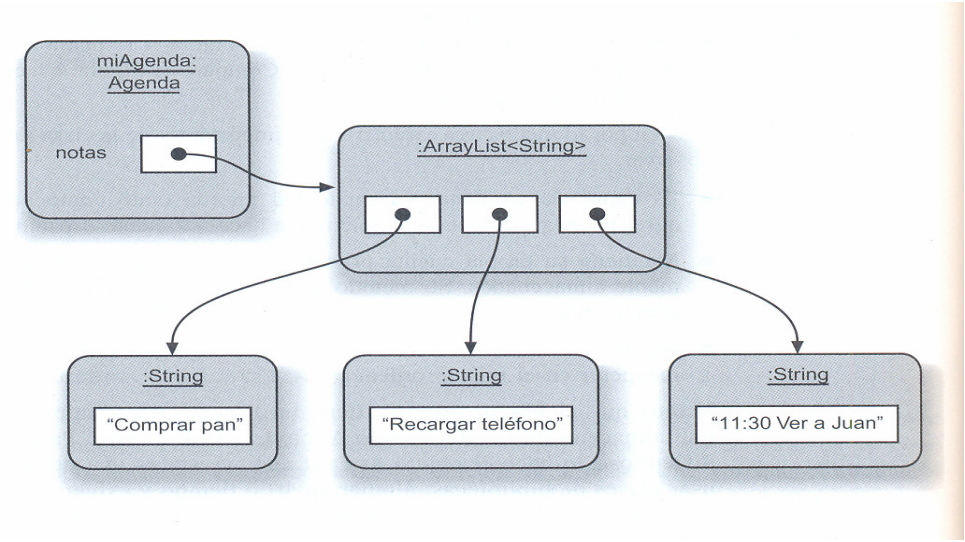
```
notas = new ArrayList<String>();
```

Las clases similares a ArrayList que se parametrizan con un segundo tipo se denominan clases genéricas.

La clase ArrayList declara muchos métodos. Por ahora sólo usaremos tres:

- **add (String nota)**, almacena una nota al final de la agenda,.
- **size()**, devuelve la cantidad de notas de la agenda.
- **get**(numeroDeNota), retorna el elemento especificado como parámetro.

Para comprender cómo opera una colección de objetos tal como ArrayList resulta útil examinar un diagrama de objetos. Una agenda con tres notas:



Características importantes de la clase ArrayList:

. Es capaz de aumentar su capacidad interna tanto como se necesite: cuando se agregan más elementos, simplemente hace suficiente espacio para ellos. Esto es transparente al programador.

. Mantiene su propia cuenta de la cantidad de elementos almacenados.

. Mantiene el orden de los elementos que se agregan; se podrá recuperarlos en el mismo orden.

Vemos que el objeto Agenda tiene un aspecto muy simple: tiene sólo un campo que almacena un objeto de tipo ArrayList<String>. Parece que todo el trabajo dificultoso lo hace el objeto ArrayList, y esta es una de las grandes ventajas de usar clases de bibliotecas: alguien invirtió tiempo y esfuerzo para implementar algo útil y nosotros tenemos acceso prácticamente libre a esta funcionalidad usando esa clase.

En esta etapa, no necesitamos preocupamos por cómo fue implementada la clase ArrayList para que tenga estas características; es suficiente con apreciar lo útil que resulta su capacidad. Esto significa que podemos utilizarla para escribir cualquier cantidad de clases diferentes que requieran almacenar un número arbitrario de objetos. Esto lo pondremos a prueba re-implementando nuestra clase ArrItems, la llamaremos ColItems, claro.

En la segunda característica, el objeto ArrayList mantiene su propia cuenta de la cantidad de objetos insertados, tiene consecuencias importantes en el modo en que implementamos la clase Agenda. A pesar de que la agenda tiene un método numeroDeNotas, no hemos definido realmente un campo específico para guardar esta información. En su lugar, la agenda delega la responsabilidad de mantener el número de elementos a su objeto ArrayList, quiere decir que la agenda no duplica

información que esté disponible desde cualquier otro objeto. Si un usuario solicita información a la agenda sobre el número de notas que tiene guardadas, la agenda pasará la pregunta al objeto notas y luego devolverá cualquier respuesta que obtenga de él.

Clases genéricas

La nueva notación que utiliza los símbolos de menor y de mayor que hemos visto con anterioridad merece un poco más de discusión. El tipo de nuestro campo notas fue declarado como:

```
ArrayList<String>
```

La clase que estamos usando aquí se denomina justamente ArrayList, pero requiere que se especifique un segundo tipo como parámetro cuando se usa para declarar campos u otras variables. Las clases que requieren este tipo de parámetro se denominan **clases genéricas**. Las clases genéricas, en contraste con las otras clases que hemos visto hasta ahora, no definen un tipo único en Java sino potencialmente muchos tipos. Por ejemplo, la clase ArrayList puede usarse para especificar un ArrayList de Strings, un ArrayList de Personas, un ArrayList de Rectángulos, o un ArrayList de cualquier otra clase que tengamos disponible. Cada ArrayList en particular es un tipo distinto que puede usarse en declaraciones de campos, parámetros y tipos de retorno. Podríamos, por ejemplo, definir los siguientes dos campos:

```
private ArrayList<Persona> alumnos;
private ArrayList<MaquinaDeBoletos> maquinitas;
```

Estas declaraciones establecen que alumnos contiene un ArrayList que puede almacenar objetos Persona, mientras que maquinitas puede contener un ArrayList que almacena objetos MaquinaDeBoletos. Tenga en cuenta que ArrayList<Persona> y ArrayList<MaquinaDeBoletos> son tipos diferentes. Los campos no pueden ser asignados uno a otro, aun cuando sus tipos deriven de la misma clase.

La numeración dentro de las colecciones sigue el mismo esquema ya visto para los arrays (Colecciones de tamaño fijo).

El método mostrarNota() ilustra la manera en que se usa un índice para obtener un elemento desde el ArrayList mediante su método get; este método no elimina un elemento de la colección.

Si Ud. intenta acceder a un elemento de una colección que está fuera de los índices válidos del ArrayList obtendrá un mensaje del error denominado desbordamiento. “IndexOutOfBoundsException”.

Eliminar un elemento de una colección

Sería muy útil tener la capacidad de eliminar las notas viejas de la Agenda cuando ya no nos interesen más. En principio, hacer esto es fácil porque la clase ArrayList tiene un método **remove** que toma como parámetro el índice de la nota que será eliminada. Cuando un usuario quiera eliminar una nota de la agenda, podemos lograrlo con sólo invocar al método remove del objeto notas.

Una complicación del proceso de eliminación es que se modifican los valores de los índices de las restantes notas que están almacenadas en la colección. Si se elimina una nota que tiene por índice un número muy bajo, la colección desplaza todos los siguientes elementos una posición a la izquierda para llenar el hueco; en consecuencia, sus índices disminuyen en 1.

Más adelante veremos que también es posible insertar elementos en un ArrayList en otros lugares distintos que el final de la colección. Esto significa que los elementos que ya están en la lista deben incrementar sus índices cuando se agrega

un nuevo elemento. **Los usuarios deben ser conscientes de estos cambios en los índices cuando agregan o eliminan notas.**

Estas opciones no existen en los arrays que implementan colecciones de tamaño fijo, como vimos anteriormente. No pueden incluir ni excluir elementos, ello implicaría modificar su tamaño, es como ir en contra de su definición. Esto es una rigidez importante, y habla muy a favor de agrupar objetos usando colecciones flexibles. Nos preocupa el tema de la eficiencia (Tiempos para igual tarea) y por esto, cuando analicemos métodos de ordenamiento, haremos un comparativo de tiempos usando colecciones de una y otra arquitectura.

Procesar una colección completa

Si agregar y eliminar notas significa que los índices pueden cambiar con el tiempo, seria de gran ayuda tener un método en la clase Agenda que pueda listar todas las notas con sus índices actuales. Podemos establecer de otra manera lo que podría hacer el método diciendo que queremos obtener cada número de índice válido y mostrar la nota que está almacenada en ese número.

El ciclo for-each

Un ciclo for-each es una forma de llevar a cabo repetidamente un conjunto de acciones, sin tener que escribir esas acciones más de una vez. Podemos resumir las acciones de un ciclo for-each en el siguiente pseudocódigo:

```
for (TipoDelElemento elemento : colección){
      cuerpo del ciclo
}
```

Ya dijimos que el lenguaje Java tiene dos variantes del ciclo for: uno es el ciclo for-each del que estamos hablando ahora, el otro se denomina simplemente ciclo for y ya lo vimos en la Unidad I.
Un ciclo for-each consta de dos partes: un encabezado de ciclo (la primera línea del ciclo) y un cuerpo a continuación del encabezado. El cuerpo contiene aquellas sentencias que deseamos llevar a cabo una y otra vez.

El ciclo for-each toma su nombre a partir de la manera en que podemos leerlo: si leemos la palabra clave For como «para cada» y los dos puntos en la cabecera del ciclo como las palabras «en la», entonces la estructura del código que mostramos anteriormente comenzaría a tener más sentido, tal como aparece en este pseudocódigo:

```
Para cada elemento en la colección hacer:{
      cuerpo del ciclo
}
```

Cuando compare esta versión con el pesudocódigo original de la primera versión, observará que elemento se escribió de manera similar a una declaración de variable: TipoDelElemento elemento. Esta sección declara una variable que luego se usa a su vez, para cada elemento de la colección. Un ejemplo en Java.

```
public void imprimirNotas (){ * Imprime todas las notas de la agenda
      for(String nota : notas){System.out.println(nota);}
}
```

En este ciclo for-each, el cuerpo del ciclo (que consiste en la única sentencia System.out.println) se ejecuta repetidamente, una vez para cada elemento del ArrayList notas.

En cada iteración, antes de que la sentencia se ejecute, la variable notas se configura para contener uno de los elementos de la lista: primero el del índice O,

luego el del índice 1, y así sucesivamente. Por lo tanto, cada elemento de la lista logra ser impreso.

Permítanos disecar el ciclo un poco más detalladamente. La palabra clave for introduce el ciclo. Está seguida por un par de paréntesis en los que se definen los detalles del ciclo. El primero de estos detalles es la declaración String nota, que define una nueva variable local nota que se usará para contener los elementos de la lista. Llamamos a ella **variable de ciclo**, puede tener cualquier nombre. Su tipo debe ser el mismo que el tipo del elemento declarado para la colección que estamos usando, en nuestro caso String.

A continuación aparecen dos puntos y la variable que contiene la colección que deseamos procesar. Cada elemento de esta colección será asignado en su turno a la variable de ciclo, y para cada una de estas asignaciones el cuerpo del ciclo se ejecutará una sola vez. Luego, podemos usar en el cuerpo del ciclo la variable de ciclo para hacer referencia a cada elemento.

Ahora, ya hemos visto cómo podemos usar el ciclo for-each para llevar a cabo algunas operaciones (el cuerpo del ciclo) sobre cada elemento de una colección. Este es un gran paso hacia adelante, pero no resuelve todos nuestros problemas. Algunas veces necesitamos un poco más de control y Java ofrece una construcción de ciclo diferente que nos permite hacerlo: el ciclo while, que ya vimos en la Unidad I.

Podemos escribir un ciclo while que imprima todas las notas de nuestra lista, tal como lo hemos hecho anteriormente mediante un ciclo for-each. La versión que usa un ciclo while:

```
int indice = O;
while (indice < notas. size()){
      System.out.println(notas.get(indice));
      indice ++:
}
```

Este ciclo while es equivalente al ciclo for-each que hemos discutido en la sección anterior. Observaciones:

. En este ejemplo, el ciclo while resulta un poco más complicado. Tenemos que declarar fuera del ciclo una variable para el índice e iniciarlo por nuestros propios medios en 0 para acceder al primer elemento de la lista.

. Los elementos de la lista no son extraídos automáticamente de la colección y asignados a una variable. En cambio, tenemos que hacer esto nosotros mismos usando el método get del ArrayList. También tenemos que llevar nuestra propia cuenta (índice) para recordar la posición en que estábamos.

. Debemos recordar incrementar la variable contadora (índice) por nuestros propios medios.

La última sentencia del cuerpo del ciclo while ilustra un operador especial para incrementar una variable numérica en 1:

Hasta ahora, el ciclo for-each es claramente la mejor opción para nuestro objetivo: fue menos complicado de escribir y es más seguro porque garantiza que siempre llegará a un final.

En nuestra versión del ciclo while es posible cometer errores que den por resultado un ciclo infinito. Si nos olvidamos de incrementar la variable índice (la última línea del cuerpo del ciclo) la condición del ciclo nunca podría ser evaluada como falsa y el ciclo se repetiría indefinidamente.

Por lo tanto, ¿cuáles son los beneficios de usar un ciclo while en lugar de un ciclo loreach?

Existen dos fundamentos: primeramente, el ciclo while no necesita estar relacionado con una colección (podemos reciclar cualquier condición que necesitemos); en segundo lugar, aun si usáramos el ciclo para procesar la colección, no necesitamos procesar cada uno de sus elementos, en cambio, podríamos querer frenar el recorrido tempranamente.

Como ejemplo, usaremos el ciclo while para escribir un ciclo que busque en nuestra colección un elemento específico y se detenga cuando lo encuentre. Para ser precisos, queremos un método de nombre buscar que tenga un parámetro String de nombre cadABuscar y luego imprima en pantalla la primer nota de la agenda que contenga la cadena de búsqueda. Se puede llevar a cabo esta tarea con la siguiente combinación del ciclo while con una sentencia condicional:

```
int indice = 0;
boolean encontrado = false;
String nota = “”;
while (indice < notas.size() && !encontrado){
      if (nota. contains (cadABuscar)) {
            nota = notas. get(indice);
            encontrado = true;
      }
      indice++;
}
System.out.println(nota);
```

Estudie este fragmento de código hasta que logre comprenderlo (es importante). Verá que la condición está escrita de tal manera que el ciclo se detiene bajo cualquiera de estas dos condiciones: si efectivamente se encuentra la cadena buscada, o cuando hemos controlado todos los elementos y no se encontró la cadena buscada.

Recorrer una colección

Antes de avanzar, discutiremos una tercer variante para recorrer una colección, que está entre medio de los ciclos while y for-each. Usa un ciclo while para llevar a cabo el recorrido y un **objeto iterador** en lugar de una variable entera como índice del ciclo para mantener el rastro de la posición en la lista.

> Concepto: Un **iterador** es un objeto que proporciona funcionalidad para recorrer todos los elementos de una colección.

Examinar cada elemento de una colección es una actividad tan común que un ArrayList proporciona una forma especial de recorrer o iterar su contenido. El método **iterator** de ArrayList devuelve un objeto Iterator. La clase Iterator también está definida en el paquete java.util de modo que debemos agregar una segunda sentencia import a la clase Agenda para poder usarla.

```
import java.util.ArrayList;
import java.util.Iterator;
```

Un Iterator provee dos métodos para recorrer una colección: hasNext y next. A continuación describimos en pseudocódigo la manera en que usamos generalmente un Iterator:

```
Iterator<TipoDelElemento> it = miColeccion.iterator();
while (it.hasNext()){
      Invocar it.next() para obtener el siguiente elemento;
      Hacer algo con dicho elemento;
}
```

En este fragmento de código usamos primero el método iterator de la clase ArrayList para obtener un objeto iterador. Observe que Iterator también es de tipo genérico y por lo tanto, lo parametrizamos con el tipo de los elementos de la colección. Luego usamos dicho iterador para controlar repetidamente si hay más elementos (it.hastNext() y para obtener el siguiente elemento (it.next().

Podemos escribir un método que usa un iterador para listar por pantalla todas las notas.

```
public void listarTodasLasNotas(){
      Iterator<String> it = notas.iterator();
      while (it.hasNext()){System.out.println(it.next());}
}
```

Preste atención en distinguir las diferentes capitalizaciones de las letras del **método iterator** y de la **clase Iterator.**

Tómese algún tiempo para comparar esta versión con las dos versiones del método listarTodasLasNotas que se mostraron anteriormente. Un punto para resaltar de la última versión es que usamos explícitamente un ciclo while, pero no necesitamos tomar precauciones respecto de la variable indice. Es así porque el Iterator mantiene el rastro de lo que atravesó de la colección por lo que sabe si quedan más elementos en la lista (hasNext) y cuál es el que debe retornar (next), si es que hay alguno.

Comparar acceso mediante índices e iteradores

Hemos visto que tenemos por lo menos tres maneras diferentes de recorrer un ArrayList. Podemos:

- usar un ciclo for-each
- el método get con un índice, dentro de un ciclo while
- un objeto Iterator, dentro de un ciclo while.

Por lo que sabemos hasta ahora, todos los abordajes parecen iguales en calidad. El primero es un poco más fácil de comprender.

El primer abordaje, usando el ciclo for-each, es la técnica estándar que se usa si deben procesarse todos los elementos de una colección porque es el más breve para este caso.

Las últimas dos versiones tienen el beneficio de que la iteración puede ser detenida más fácilmente en el medio de un proceso, de modo que son preferibles para cuando se procesa sólo parte de una colección.

Para un ArrayList, los dos últimos métodos (usando ciclos while) son buenos aunque no siempre es así. Java provee muchas otras clases de colecciones además de ArrayList. Y en algunas de ellas es imposible o muy ineficiente acceder a elementos individuales mediante un índice. Por lo que nuestra primera versión del ciclo while es una solución particular para la colección ArrayList y puede que no funcione para otros tipos de colecciones.

La segunda solución, usando un iterador, está disponible para todas las colecciones de las clases de las bibliotecas de lava y es un patrón importante.

Resumen del ejemplo agenda

En el ejemplo agenda hemos visto cómo podemos usar un objeto ArrayList, creado a partir de una clase extraída de una biblioteca de clases, para almacenar un número arbitrario de objetos en una colección. No tenemos que decidir anticipadamente cuántos objetos deseamos almacenar y el objeto ArrayList mantiene automáticamente el registro de la cantidad de elementos que contiene.

Hemos hablado sobre cómo podemos usar un ciclo para recorrer todos los elementos de una colección. Java tiene varias construcciones para ciclos; las dos que hemos usado en este lugar son el ciclo for-each y el ciclo while.

En un ArrayList podemos acceder a sus elementos por un índice o podemos recorrerla completamente usando un objeto Iterator.

Herencia

A continuación veremos recursos adicionales de programación orientadas a objetos que nos ayudan a mejorar la estructura general, la modularidad de nuestras aplicaciones. Estos conceptos son la herencia y el polimorfismo.

La herencia es una potente construcción que puede usarse para crear soluciones de problemas de diferente naturaleza. Esclarezcamos esto mediante un ejemplo:
DoME (Database of Multimedia Entretainment)

En esencia, DoME es una aplicación que nos permite almacenar información sobre discos compactos de música (en CD) y de películas (en DVD). La idea es crear un catálogo de CDs y DVDs.

La funcionalidad que pretendemos de DoME incluye como mínimo lo siguiente:

- permitimos ingresar información sobre los CD y los DVD.
- almacenar esta información de manera permanente.
- brindar una función de búsqueda que nos permita:
  - encontrar todos los CD de un cierto intérprete.
  - encontrar todos los DVD de determinado director.
- imprimir listados
  - Todos los DVD que hay en la base
  - Todos los CD de música.
- permitimos eliminar información.

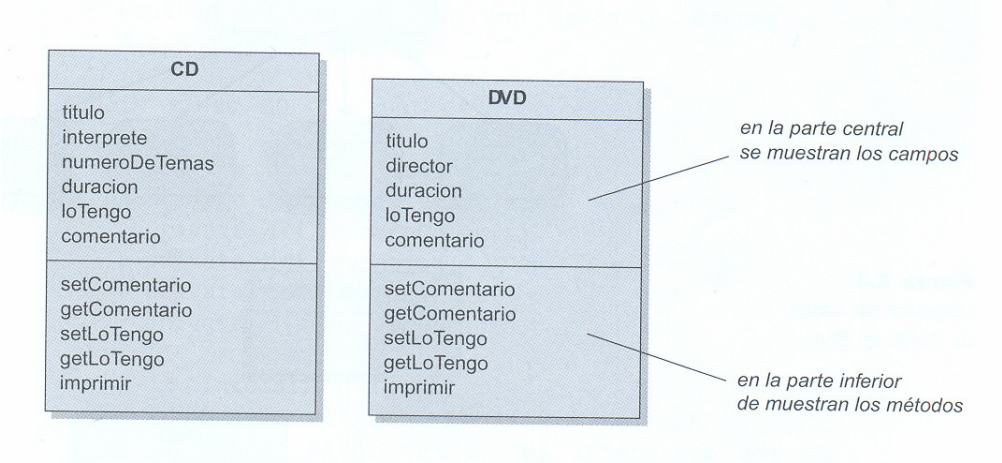
Los detalles que queremos almacenar de cada CD son:

. el título del álbum;
. el intérprete (el nombre de la banda o del cantante);
. el número de temas;
. el tiempo de duración;
. una bandera que indique si tenemos una copia de este CD y
. un comentario (un texto arbitrario).

Los detalles que queremos almacenar de cada DVD son:

. el título del DVD
. el nombre del director
. el tiempo de duración (película principal)
. una bandera que indique si tenemos una copia de este DVD y
. un comentario (un texto arbitrario).

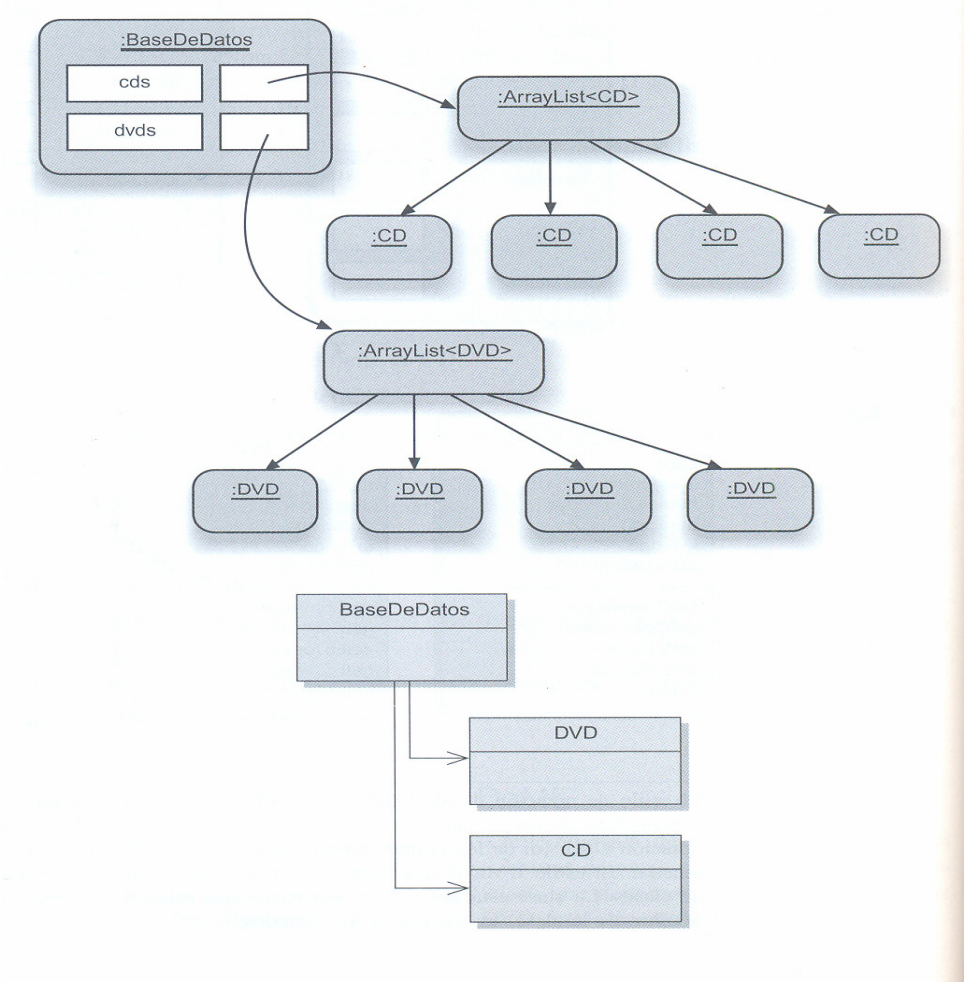
Para implementar esta aplicación, primero tenemos que decidir qué clases usaremos para modelar este problema. Claramente: clase CD y clase DVD; los objetos de estas clases deben encapsular datos y sus métodos de tratamiento.



Notación UML para atributos y métodos en un diagrama de clases

Una vez que hemos definido las clases CD y DVD podemos crear tantos objetos CD y tantos objetos DVD como necesitemos. Aparte de esto, necesitamos otro objeto: un objeto base de datos que pueda contener una colección de CD y una colección de DVD.

El objeto base de datos puede contener dos colecciones de objetos (por ejemplo, una de tipo ArrayList<CD> y otra de tipo ArrayList<DVD». Luego, una de estas colecciones puede contener todos los CD y la otra todos los DVD.



En este punto hemos llevado a cabo la parte más importante: hemos definido la estructura de datos que almacena la información esencial.

Hasta el momento, el diseño ha sido sumamente fácil y ahora podemos avanzar y diseñar el resto que aún falta, pero antes de hacerlo, discutiremos la calidad de la solución lograda.

Existen varios problemas fundamentales en nuestra solución actual; la más obvia es la **duplicación de código**.

Hemos observado que las clases CD y DVD son muy similares, y en consecuencia, la mayoría del código de ambas clases es idéntico, hay muy pocas diferencias. Es como que gran parte de la programación será copiar y pegar, y luego arreglar todas las diferencias.
Frecuentemente se presentan problemas asociados al mantenimiento del código duplicado.

Hay otro lugar en el que tenemos duplicación de código: en la clase BaseDeDatos. Podemos ver en ella que cada cosa se hace dos veces, una vez para los CD y otra para los DVD. La clase define dos variables para las listas, luego crea dos objetos lista, define dos métodos «agregar» y tiene dos bloques casi idénticos de código en el método listar para imprimir ambas listas.

Los problemas que traen aparejados esta duplicación de código se tornan mas evidentes si analizamos lo qué tendríamos que hacer para agregar otro tipo de elemento multimedial en este programa. Imagine que queremos almacenar información

no sólo sobre DVD y CD sino también sobre libros. Los libros se parecen bastante a los elementos antes mencionados, de modo que sería fácil modificar nuestra aplicación para incluir libros. Podríamos introducir otra clase, Libro, y escribir, esencialmente, una tercera versión del código que ya está en las clases CD y DVD. Luego tenemos que trabajar en la clase BaseDeDatos y agregar otra variable para la lista de libros, otro objeto lista, otro método «agregar» y otro ciclo en el método listar.

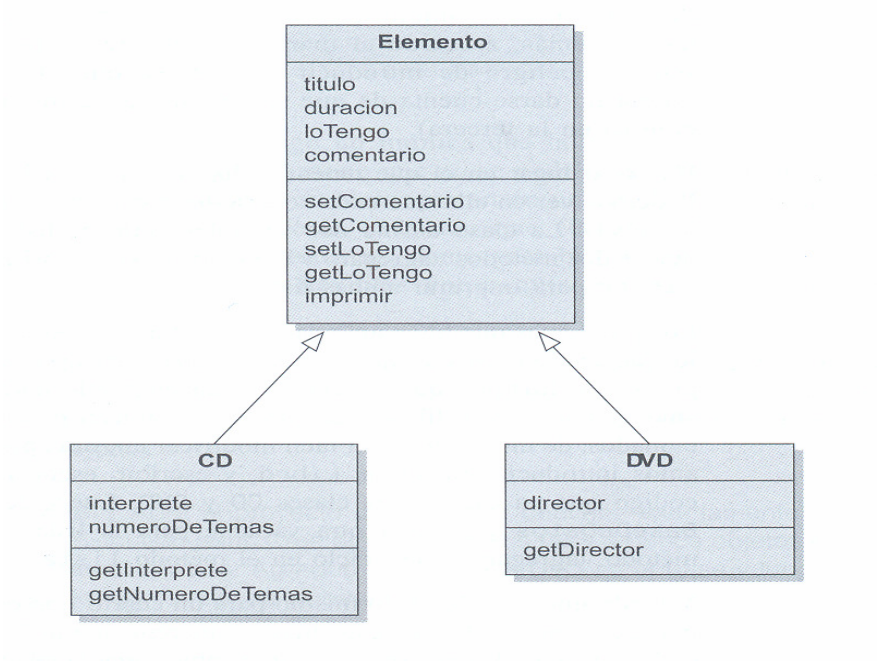
Y tendríamos que hacer lo mismo para un cuarto tipo de elemento multimedial.

Cuanto más repitamos este proceso, más se incrementarán los problemas de duplicación de código y más difícil será realizar cambios más adelante. Cuando nos sentimos incómodos con una situación como ésta, frecuentemente es un buen indicador de que hay una alternativa mejor de abordaje. Para este caso en particular, los lenguajes orientados a objetos proveen una característica distintiva que tiene un gran impacto en programas que involucran conjuntos de clases similares. En los títulos siguientes introduciremos esta característica que se denomina herencia.

Usar herencia

Concepto: la herencia nos permite definir una clase como extensión de otra.

Es un mecanismo que nos ofrece una solución a nuestro problema de duplicación de código. La idea es simple: en lugar de definir las clases CD y DVD completamente independientes, definimos primero una clase que contiene todas las cosas que tienen en común ambas clases. Podemos llamar a esta clase Elemento y luego declarar que un CD es un Elemento y que un DVD es un Elemento. Finalmente, podemos agregar en la clase CD aquellos detalles adicionales necesarios para un CD y los necesarios para un DVD en la clase DVD. La característica esencial de esta técnica es que necesitamos describir las características comunes sólo una vez.



El diagrama muestra la clase Elemento en la parte superior; esta clase define todos los campos y métodos que son comunes a todos los elementos (CD y DVD). Debajo de la clase Elemento, aparecen las clases **CD y DVD** que contienen sólo aquellos campos y métodos que son únicos para cada clase en particular. Aquí hemos agregado los métodos: getInterprete y getNumeroDeTemas en la clase CD y getDirector en la clase DVD, las clases CD y DVD pueden definir sus propios métodos.

Esta nueva característica de la programación orientada a objetos requiere algunos nuevos términos. En una situación tal como esta, decimos que la clase CD **deriva** de la clase Elemento. La clase DVD también **deriva** de Elemento. Cuando hablamos de programas en Java, también se usa la expresión «la clase CD **extiende** a la clase Elemento» pues Java utiliza la palabra clave «extends» para definir la relación de herencia.

La flecha en el diagrama de clases (dibujada generalmente con la punta sin rellenar) representa la relación de herencia.

La clase Elemento (la clase a partir de la que se derivan o heredan las otras) se denomina **clase padre**, **clase base o superclase**. Nos referimos a las clases heredadas (en este ejemplo, CD y DVD) como **clases derivadas**, **clases hijos o sub-clases**.
Concepto: una **superclase** es una clase extendida por otra clase

Algunas veces, la herencia también se denomina relación **«es un»**. La razón de esta nomenclatura radica en que la subclase es una **especialización** de la superclase. Podemos decir que «un CD **es un** elemento» y que «un DVD **es un** elemento».
Concepto: una **subclase** es aquella que extiende a otra clase. Hereda todos los campos y métodos de la clase que extiende.

El propósito de usar herencia ahora resulta bastante obvio. Las instancias de la clase CD tendrán todos los campos que están definidos en la clase CD y todos los de la clase Elemento. (CD hereda los campos de la clase Elemento.) Las instancias de DVD tendrán todos los campos definidos en las clases DVD y Elemento. Por lo tanto, logramos tener lo mismo que teníamos antes, con la diferencia de que ahora necesitamos definir los campos titulo, duracion, loTengo y comentario sólo una vez (pero podemos usarlos en dos lugares diferentes).

Lo mismo ocurre con los métodos: las instancias de las subclases tienen todos los métodos definidos en ambas, la superclase y la subclase. En general, podemos decir: dado que un CD es un elemento, un objeto CD tiene todas las cosas que tiene un elemento y otras más. Y dado que un DVD también es un elemento, tiene todas las cosas de un elemento y otras más.

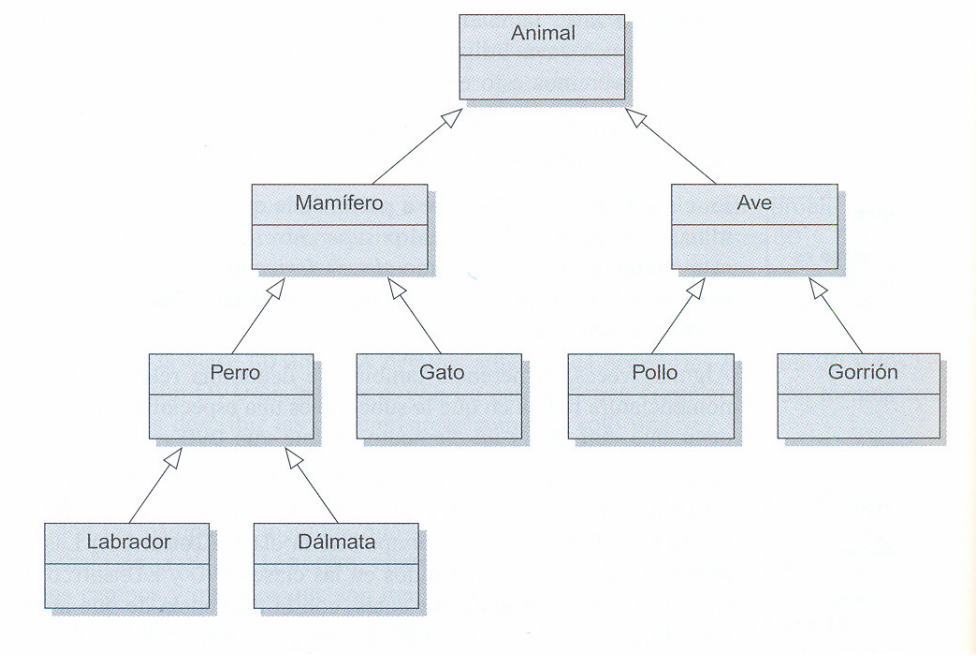
Por lo tanto, la herencia nos permite crear dos clases que son bastante similares evitando la necesidad de escribir dos veces la parte que es idéntica. La herencia tiene otras ventajas más que discutiremos a continuación, sin embargo, primero daremos otra mirada más general a las jerarquías de herencia.

Jerarquías de herencia

La herencia puede usarse en forma mucho más general que el ejemplo que mostramos anteriormente. Se pueden heredar más de dos subclases a partir de la misma superclase y una subclase puede convertirse en la superclase de otras subclases. En consecuencia, las clases forman una jerarquía de herencia.

Un ejemplo más conocido de una jerarquía de herencia es la clasificación de las especies que usan los biólogos. En la Figura 8.6 se muestra una pequeña parte de esta clasificación: podemos ver que un dálmata es un perro, que a su vez es un mamífero y que también es un animal.

El labrador es un ser vivo, ladra, come carne. Muchas más cosas sabremos porque además es un perro, un mamífero y un animal.




```
public class Elemento{
      private String titulo;
      private int duracion;
      private bolean loTengo;
      private String comentario;
      // omitimos constructores y métodos
}
```

Hasta ahora, esta superclase no tiene nada especial: comienza con una definición normal de clase y declara los campos de la manera habitual. A continuación, examinamos el código de la clase CD:

```
public class CD extends Elemento{
      private String interprete;
      private int numeroDeTemas;
      // se omitieron constructores y métodos
}
```

En este código hay dos puntos importantes para resaltar.

- la palabra clave **extends** define la relación de herencia.
- la clase CD define sólo aquellos campos que son únicos para los objetos CD (interprete y numeroDeTemas). Los campos de Elemento se heredan y no necesitan ser listados en este código. No obstante, los objetos de la clase CD tendrán los campos titulo, duracion y así sucesivamente.

```
public class DVD extends Elemento{
      private String director;
      // se omitieron constructores y métodos
}
```

Herencia y derechos de acceso entre una subclase y su superclase:

Las reglas de privacidad se aplican normalmente

- una subclase **no puede** acceder a los miembros privados de su superclase.
- una subclase **puede** usar cualquier miembro público de su superclase.
- una subclase **puede** usar cualquier miembro protected de su superclase.

Herencia e inicialización

Cuando creamos un objeto, el constructor de dicho objeto tiene el cuidado de inicializar todos los campos con algún estado razonable. Veamos cómo se hace esto en las clases que se heredan a partir de otras clases.

Cuando creamos un objeto CD, pasamos varios parámetros a su constructor: el título, el nombre del intérprete, el número de temas y el tiempo de duración. Algunos de estos parámetros contienen valores para los campos definidos en la clase Elemento y otros valores para los campos definidos en la clase CD. Todos estos campos deben ser correctamente inicial izados, el código siguente muestra la forma correcta de hacerlo.

```
public class Elemento{
      private String titulo;
      private int duracion;
      private boolean loTengo;
      private String comentario;

      public Elemento(String elTitulo, int tiempo){
            titulo = elTitulo;
            duracion = tiempo;
            loTengo = false;
            comentario = “”;
      }
      //    Omitimos métodos
}

public class CD extends Elemento{
      private String interprete;
      private int numeroDeTemas;
      public CD(String elTitulo, String elInterprete, int temas, int tiempo){
      super(elTitulo, tiempo);
      interprete = elInterprete;
      numeroDeTemas = temas;
      }
      //    Omitimos métodos
}
```

Observaciones.

- El constructor Elemento recibe los parámetros y código necesarios para inicializar los campos de su clase.
- El constructor CD recibe los parámetros necesarios para inicializar tanto los campos de Elemento como los de CD. La sentencia
  super (elTitulo, tiempo); llamada al constructor de la superclase.

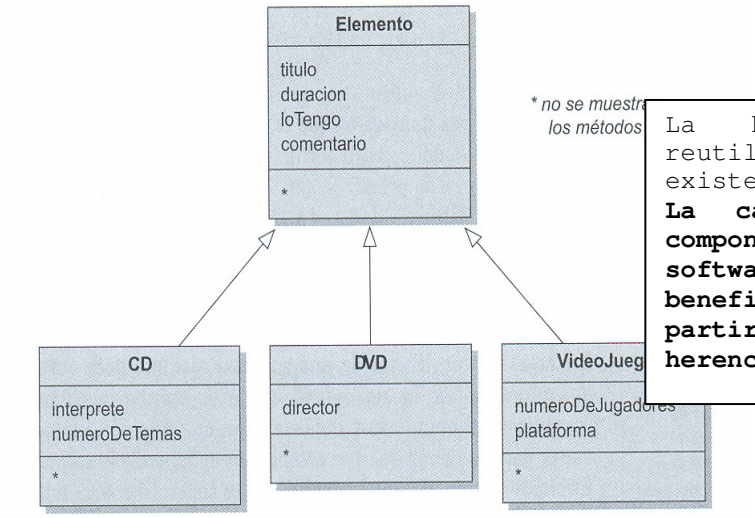
El efecto de esta llamada es que se ejecuta el constructor de Elemento, formando parte de la ejecución del constructor del CD.

En lava, un constructor de una subclase siempre debe invocar en su primer sentencia al constructor de la superclase. Si no se escribe una llamada al constructor de una superclase, el compilador de lava insertará automáticamente una llamada a la superclase, para asegurar que los campos de la superclase se inicialicen adecuadamente. La inserción automática de la llamada a la superclase sólo funciona si la superclase tiene un constructor sin parámetros (ya que el compilador no puede adivinar qué parámetros deben pasarse); en el caso contrario, lava informa un error.

En general, es buena técnica incluir siempre en los constructores llamadas explícitas a la superclase, aun cuando sea una llamada que el compilador puede generar automáticamente.

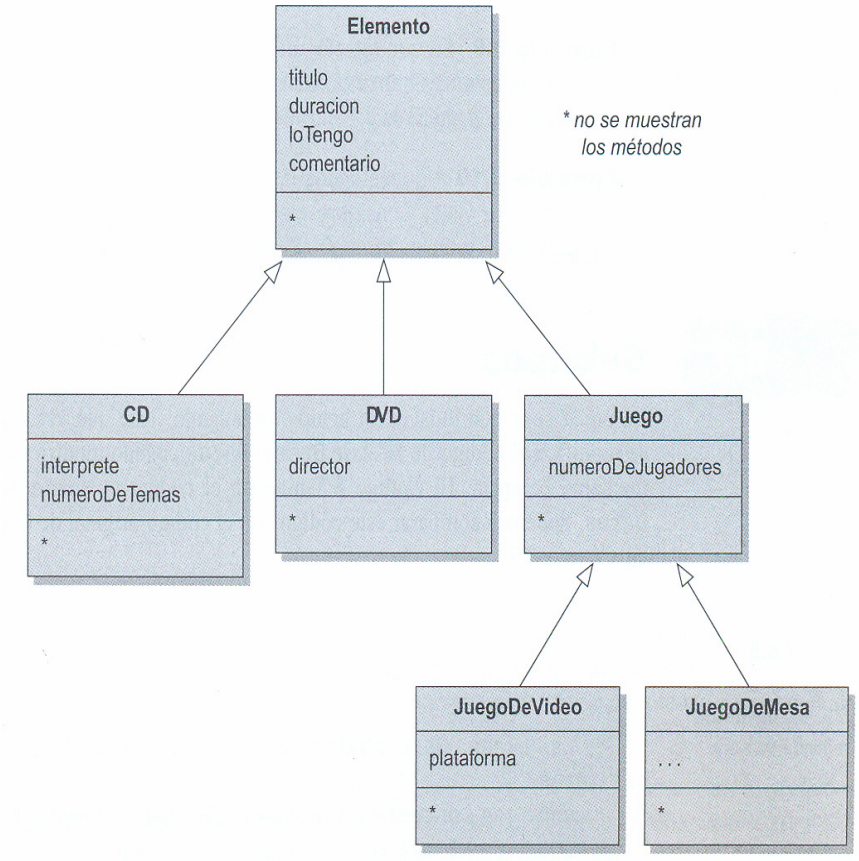
Agregando nuevos elementos a una jerarquía existente.

Queremos agregar en nuestra base de datos información sobre juegos de vídeo, podemos definir una nueva subclase de Elemento de nombre JuegoDeVideo. Como ella es una subclase de Elemento, automáticamente hereda todos los campos y métodos definidos en Elemento; luego agregamos atributos y métodos que son específicos de los juegos tales como número máximo de jugadores o la plataforma sobre la que



La herencia nos permite reutilizar el trabajo existente.
La capacidad de reutilizar componentes existentes de software es uno de los grandes beneficios que obtenemos a partir de la facilidad de la herencia.

corren.



Refactorización:
Supongamos Que necesitamos agregar otros juegos, y descubrimos que hay un atributo común: numeroDeJugadores. No queremos duplicar código, introducimos una nueva clase, Juegos y subordinamos a ella JuegosDeVideo y JuegosDeMesa. Esto es Refactorizar

Ventajas de la herencia

- Evita la duplicación de código.
- Se reutiliza código
- Facilita el mantenimiento
- Facilita la extendibilidad

Subtipos

El código de BaseDeDatos, ya usando herencia:

```
import java.util.ArrayList;
public class BaseDeDatos{
      private ArrayList<Elemento> elementos;
      public BaseDeDatos(){   //    constructor
            elementos = new ArrayList<Elemento>();

      }
      public void agregarElemento (Elemento elElemento){
            elementos.add(elElemento);

      }

      public void listar ( ){ //    lista todos los elementos contenidos
            for (Elemento elemento : elementos )
                  elemento.imprimir();
      }
}
```

Observe que tenemos un único método agregarElemento que sirve para agregar CDs y DVDs indistintamente. Lo mismo para imprimir.

```
public void agregarElemento (Elemento elElemento)
```

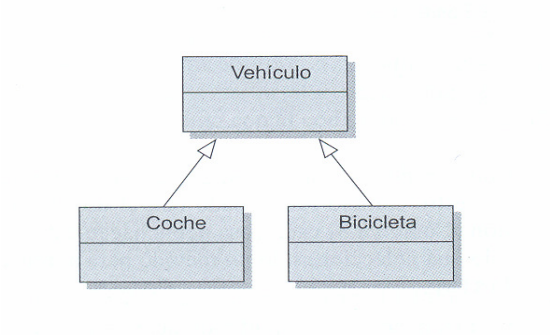
Los objetos de las subclases pueden usarse en cualquier lugar que se requiera el tipo de su superclase. Esto me posibilita declarar como parámetro formal Elemento y en la invocación mandarle objetos Elemento, CD, DVD indistintamente.

Subclases y subtipos

Las clases definen tipos. Un objeto que se crea a partir de la clase DVD es de tipo DVD. También hemos dicho que las clases pueden tener subclases, por lo tanto, los tipos definidos por las clases pueden tener subtipos.
En nuestro ejemplo, el tipo DVD es un subtipo del tipo Elemento.

Subtipos y asignación

Cuando queremos asignar un objeto a una variable, el tipo del objeto debe coincidir con el tipo de la variable. Por ejemplo:




```
coche miCoche = new Coche();
```

es una asignación válida porque se asigna un objeto de tipo coche a una variable declarada para contener objetos de tipo Coche. Ahora que conocemos la herencia debemos establecer la regla de tipos de manera más completa: una variable puede contener objetos del tipo declarado **o de cualquier subtipo del tipo declarado**.

```
Vehículo v1 = new Vehivulo();
Vehículo v2 = new Coche();
Vehículo v3 = new Bicicleta();
```
Todas las sentencias anteriores son perfectamente válidas.

Variables polimórficas

En Java, las variables que contienen objetos son variables polimórficas. El término «polimórfico» (literalmente: muchas formas) se refiere al hecho de que una misma variable puede contener objetos de diferentes tipos (del tipo declarado o de cualquier subtipo del tipo declarado). El polimorfismo aparece en los lenguajes orientados a objetos en numerosos contextos, las variables polimórficas constituyen justamente un primer ejemplo.

Observemos la manera en que el uso de una variable polimórfica nos ayuda a simplificar nuestro método listar. El cuerpo de este método es

```
for (Elemento elemento : elementos)
      elemento.imprimir();
```

En este método recorremos la lista de elementos (contenida en un ArrayList mediante la variable elementos), tomamos cada elemento de la lista y luego invocamos su método imprimir. Observe que los elementos que tomamos de la lista son de tipo CD o DVD pero no son de tipo Elemento. Sin embargo, podemos asignarlos a la variable elemento (declarada de tipo Elemento) porque son variables polimórficas. La variable elemento es capaz de contener tanto objetos CD como objetos DVD porque estos son subtipos de Elemento.

Por lo tanto, el uso de herencia en este ejemplo ha eliminado la necesidad de escribir dos ciclos en el método listar. La herencia evita la duplicación de código no sólo en las clases servidoras sino también en las clases clientes de aquellas.

Nota: Si UD. codifica lo que estamos detallando descubrirá que el método imprimir tiene un problema: no imprime todos los detalles. Esto es así porque estamos invocando el método imprimir() de la clase Elemento, que no imprime atributos de CD o DVD. Esto se solucionará cuando conozcamos un poco mas de polimorfismo. Un poquito de paciencia.

Enmascaramiento de tipos (Casting)

Algunas veces, la regla de que no puede asignarse un supertipo a un subtipo es más restrictiva de lo necesario. Si sabemos que la variable de un cierto supertipo contiene un objeto de un subtipo, podría realmente permitirse la asignación. Por ejemplo:

```
Vehiculo v;
Coche a = new Coche();
v = a;      //    Sin problemas
a = v;      //    Error, según el compilador
```

Obtendremos un error de compilación en **a = v**.
El compilador no acepta esta asignación porque como a (Coche) tiene mas atributos que v (Vehículo) partes del objeto a quedan sin asignación. El compilador no sabe que v ha sido anteriormente asignado por un coche.

Podemos resolver este problema diciendo explícitamente al sistema, que la variable v contiene un objeto Coche, y lo hacemos utilizando el operador de enmascaramiento de tipos, en una operación también conocida como casting.

```
a = (Coche)v;     // correcto
```

En tiempo de ejecución, el Java verificará si realmente v es un Coche. Si fuimos cuidadosos, todo estará bien; si el objeto almacenado en v es de otro tipo, el

sistema indicará un error en tiempo de ejecución (denominado **ClassCastException**) y el programa se detendrá.

El compilador no detecta (Naturalmente) errores de enmascaramiento en tiempo de compilación. Se detectan en ejecución y esto no es bueno.
El enmascaramiento debiera evitarse siempre que sea posible, porque puede llevar a errores en tiempo de ejecución y esto es algo que claramente no queremos. El compilador no puede ayudamos a asegurar la corrección de este caso.

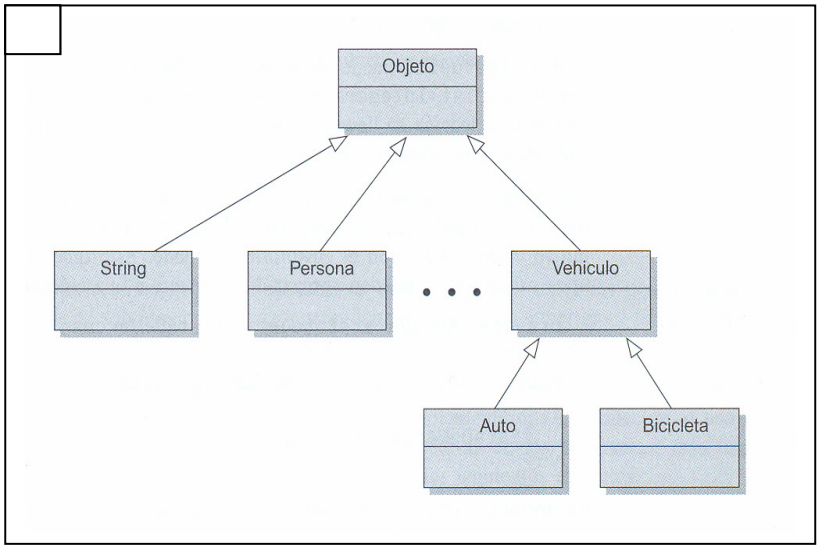
En la práctica, raramente se necesita del enmascaramiento en un programa orientado a objetos **bien estructurado**. En la mayoría de los casos, cuando se use un enmascaramiento en el código, debiera reestructurarse el código para evitar el enmascaramiento, y se terminará con un programa mejor diseñado. Generalmente, se resuelve el problema de la presencia de un enmascaramiento reemplazándolo por un **método polimórfico** (Un poquito de paciencia).

La clase Object

Todas las clases tienen una superclase. Hasta ahora, nos puede haber parecido que la mayoría de las clases con que hemos trabajado no tienen una superclase, excepto clases como DVD y CD que extienden otra clase. En realidad, mientras que podemos declarar una superclase explícita para una clase dada, todas las clases que no tienen una declaración explícita de superclase derivan implícitamente de una clase de nombre **Object.**

Object es una clase de la biblioteca estándar de Java que sirve como superclase para todos los objetos. Es la **única clase de Java sin superclase.** Escribir una declaración de clase como la siguiente

public **class Person**{} es equivalente a public **class Person extends Object**{}



Tener una superclase sirve a dos propósitos.
. Podemos declarar variables polimórficas de tipo Object que pueden contener cualquier objeto (esto no es importante)

. Podemos usar Polimorfismo (Ya lo vemos) y esto si es importante.

Autoboxing y clases «envoltorio»

Hemos visto que, con una parametrización adecuada, las colecciones pueden almacenar objetos de cualquier tipo; queda un problema, Java tiene algunos tipos que no son objetos.

Como sabemos, los tipos primitivos tales como int, boolean y char están separados de los tipos objeto. Sus valores no son instancias de clases y no derivan de la clase Object. Debido a esto, no son suptipos de Object y normalmente, no es posible ubicarlos dentro de una colección.

Este es un inconveniente pues existen situaciones en las que quisiéramos crear, por ejemplo, una lista de enteros (int) o un conjunto de caracteres (char). ¿Qué hacer?

La solución de Java para este problema son las **clases envoltorio**. En Java, cada tipo simple o primitivo tiene su correspondiente clase envoltorio que representa el mismo tipo pero que, en realidad, es un tipo objeto. Por ejemplo, la clase envoltorio para el tipo simple int es la clase de nombre Integer.

La siguiente sentencia envuelve explícitamente el valor de la variable ix de tipo primitivo int, en un objeto Integer:

```
Integer ienvuelto = new Integer(ix);
```

y ahora ienvuelto puede almacenarse fácilmente por ejemplo, en una colección de tipo ArrayList<Integer>. Sin embargo, el almacenamiento de valores primitivos en un objeto colección se lleva a cabo aún más fácilmente mediante una característica del compilador conocida como autoboxing.

En cualquier lugar en el que se use un valor de un tipo primitivo en un contexto que requiere un tipo objeto, el compilador automáticamente envuelve al valor de tipo primitivo en un objeto con el envoltorio adecuado. **Esto quiere decir que los valores de tipos primitivos se pueden agregar directamente a una colección**:

```
private ArrayList<Integer> listaDeMarcas;

public void almacenarMarcaEnLista (int marca){
      listaDeMarcas.agregar(marca);
}
```

La operación inversa, **unboxing**, también se lleva a cabo automáticamente, de modo que el acceso a un elemento de una colección podría ser:

```
int primerMarca = listaDeMarcas.remove(0);
```

El proceso de autoboxing se aplica en cualquier lugar en el que se pase como parámetro un tipo primitivo a un método que espera un tipo envoltorio, y cuando un valor primitivo se almacena en una variable de su correspondiente tipo envoltorio. De manera similar, el proceso de unboxing se aplica cuando un valor de tipo envoltorio se pasa como parámetro a un método que espera un valor de tipo primitivo, y cuando se almacena en una variable de tipo primitivo.

Tipo estático y tipo dinámico

Volvemos sobre un problema inconcluso: el método imprimir de DoME, no muestra todos los datos de los elementos.

El intento de resolver el problema de desarrollar un método imprimir completo y polimórfico nos conduce a la discusión sobre **tipos estáticos y tipos dinámicos y sobre despacho de métodos**. Comencemos desde el principio.

Necesitamos ver más de cerca los tipos. Consideremos la siguiente sentencia:

```
Elemento e1 = new CD();
```

¿Cuál es el tipo de e1?

Depende de qué queremos decir con «tipo de e1».

- El tipo de la variable **e1** es Elemento; (tipo estático)
- el tipo del objeto almacenado en e**1** es CD. (tipo dinámico)

Entonces el tipo estático de **e1** es Elemento y su tipo dinámico es CD.

En el momento de la llamada **e1.imprimir();** el **tipo estático** de la variable elemento es Elemento mientras que su **tipo dinámico** puede ser tanto CD como DVD. No sabemos

cuál es su tipo ya que asumimos que hemos ingresado tanto objetos CD como objetos DVD en nuestra base de datos.

Y en que clase debe estar codificado el método imprimir()?

- En tiempo de compilación necesitamos de la existencia de imprimir() en la clase Elemento, el compilador trabaja con tipo estático.
- En tiempo de ejecución necesitamos de la existencia de un método imprimir() adecuado a los datos del objeto CD o DVD.

En definitiva, necesitamos de imprimir() en las tres clases. Aunque no será lo mismo lo que se imprima en cada uno de ellos. Lo que debemos hacer entonces es

Sobrescribir el método

Veamos el método imprimir en cada una de las clases.

```
public class Elemento{
      …

      public void imprimir(){
            System.out.print(titulo + " (" + duracion + " minutos) " );
            if (loTengo){System.out.println("*");
            }
            else {System.out.println();}
            System.out.println("    " + comentario);
      }
}

public class CD extends Elemento{
public void imprimir(){
            System.out.println("    " + interprete);
            System.out.println("    temas: " + numeroDeTemas);
      }
}
public class DVD extends Elemento{
      public void imprimir(){
            System.out.println("    director: " + director);
      }
}
```

Este diseño funciona mejor: compila y puede ser ejecutado, aunque todavía no está perfecto. Proporcionamos una implementación de este diseño mediante el proyecto dome-v3.

La técnica que usamos acá se denomina **sobrescritura** (algunas veces también se hace referencia a esta técnica como **redefinición**). La sobrescritura es una situación en la que un método está definido en una superclase (en este ejemplo, el método imprimir de la clase Elemento) y un método, con exactamente la misma signatura, está definido en la subclase.

En esta situación, los objetos de la subclase tienen dos métodos con el mismo nombre y la misma signatura: uno heredado de la superclase y el otro propio de la subclase. ¿Cuál de estos dos se ejecutará cuando se invoque este método?

Búsqueda dinámica del método (Despacho dinámico)

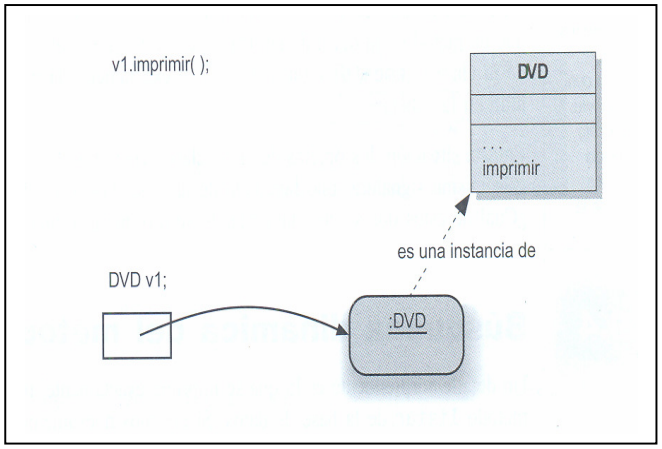
Si ejecutamos el método listar de la BaseDeDatos, podremos ver que se ejecutarán los métodos imprimir de CD y de DVD pero no el de Elemento, y entonces la mayor parte de la información, la común contenida en Elemento, no se imprime.

Que está pasando? Vimos que el compilador insistió en que el método **imprimir** esté en la clase Elemento, no le alcanzaba con que los métodos estuvieran en las subclases. Este experimento ahora nos muestra que el método de la clase Elemento no se ejecuta para nada, pero sí se ejecutan los métodos de las subclases.

Ocurre que el control de tipos que realiza el compilador es sobre el tipo estático, pero en **tiempo de ejecución los métodos que se ejecutan son los que corresponden al tipo dinámico**.

Saber esto es muy importante pero todavía insuficiente.
Para comprenderla mejor, veamos con más detalle cómo se invocan los métodos. Este procedimiento se conoce como búsqueda de método, ligadura de método o despacho de método. En este libro, nosotros usamos la terminología «búsqueda de método».

Comenzamos con un caso bien sencillo de búsqueda de método. Suponga que tenemos un objeto de clase DVD almacenado en una variable v1 declarada de tipo DVD (Figura 9.5). La clase DVD tiene un método imprimir y no tiene declarada ninguna superclase. Esta es una situación muy simple que no involucra herencia ni polimorfismo.
Ejecutamos **v1.imprimir{)**. Esto requiere de las siguientes acciones:



1. Se accede a la variable v1.

2. Se encuentra el objeto almacenado en esa variable (siguiendo la referencia).

3. Se encuentra la clase del objeto (siguiendo la referencia «es instancia de»).

4. Se encuentra la implementación del método imprimir en la clase y se ejecuta.

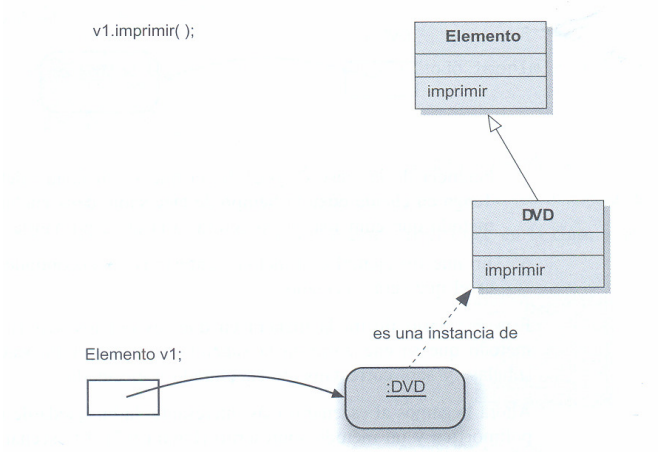
Hasta aquí, todo es muy simple.

A continuación, vemos la búsqueda de un método cuando hay herencia. El escenario es similar al anterior, pero esta vez la clase DVD tiene una superclase, Elemento, y el método imprimir está definido sólo en la superclase
Ejecutamos la misma sentencia. La invocación al método comienza de manera similar: se ejecutan nuevamente los pasos 1 al 3 del escenario anterior pero luego continúa de manera diferente:

4. No se encuentra ningún método imprimir en la clase DVD.

5. Se busca en la superclase un método que coincida. Y esto se hace hasta encontrarlo, subiendo en la jerarquía hasta Object si fuera necesario. Tenga en cuenta que, en tiempo de ejecución, debe encontrarse definitivamente un método que coincida, de lo contrario la clase no habría compilado.

6. En nuestro ejemplo, el método imprimir es encontrado en la clase Elemento y es el que será ejecutado



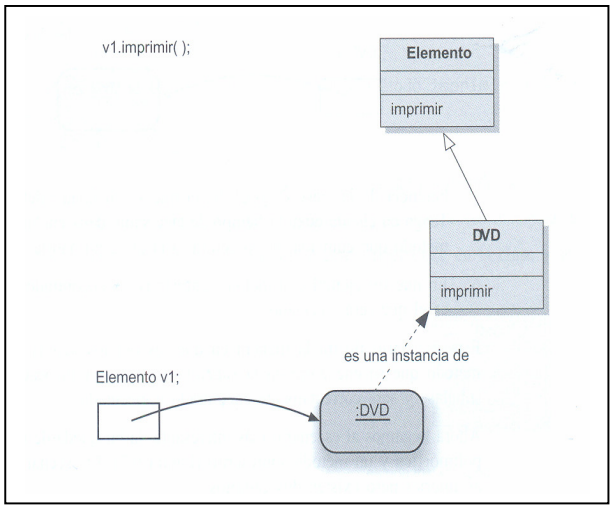
Este ejemplo ilustra la manera en que los objetos heredan los métodos. Cualquier método que se encuentre en la superclase puede ser invocado sobre un objeto de la subclase y será correctamente encontrado y ejecutado.

Ahora llegamos al escenario más interesante: la búsqueda de métodos con una variable polimórfica y un método sobrescrito. Los cambios:

. El tipo declarado de la variable v1 ahora es Elemento, no DVD.

. El método imprimir está definido en la clase Elemento y redefinido (o sobrescrito) en la clase DVD.

Este escenario es el más importante para comprender el comportamiento de nuestra aplicación DoME y para encontrar una solución a nuestro problema con el método imprimir.



Los pasos que se siguen para la ejecución del método son exactamente los mismos pasos 1 al 4, primer caso

Observaciones:

. No se usa ninguna regla especial para la búsqueda del método en los casos en los que el tipo dinámico no sea igual al tipo estático.

. El método que se encuentra primero y que se ejecuta está determinado por el tipo dinámico, no por el tipo estático. La instancia con la que estamos trabajando es de la clase DVD, y esto es todo lo que cuenta.

. Los métodos sobrescritos en las subclases tienen precedencia sobre los métodos de las superclases. La búsqueda de método comienza en la clase dinámica de la instancia, esta redefinición del método es la que se encuentra primero y la que se ejecuta.

Esto explica el comportamiento que observamos en nuestro proyecto DoME. Los métodos imprimir de las subclases (CD y DVD) sólo se ejecutan cuando se imprimen los elementos, produciendo listados incompletos. Como podemos solucionarlo?

Llamada a super en métodos

Ahora que conocemos detalladamente cómo se ejecutan los métodos sobrescritos podemos comprender la solución al problema de la impresión. Es fácil ver que lo que queremos lograr es que, para cada llamada al método imprimir de, digamos un objeto CD, se ejecuten para el mismo objeto **tanto el método imprimir de la clase Elemento como el de la clase CD**. De esta manera se imprimirán todos los detalles.

Cuando invoquemos al método imprimir sobre un objeto CD, inicialmente se invocará al método imprimir de la clase CD. En su primera sentencia, este método se convertirá en una invocación al método imprimir de la superclase que imprime la información general del elemento. Cuando el control regrese del método de la superclase, las restantes sentencias del método de la subclase imprimirán los campos distintivos de la clase CD.

```
public void imprimir(){ //    Método imprimir de la clase CD
      super.imprimir();
      System.out.println("    " + interprete);
      System.out.println("    temas: ") + numeroDeTemas);

}
```

Detalles sobre diferencias del super usado en constructores:

- el nombre del método de la superclase está explícitamente establecido. Una llamada a super en un método siempre tiene la forma

  super.nombre-del-método(parámetros);

- La llamada a super en los métodos puede ocurrir en cualquier lugar dentro de dicho método. No tiene por qué ser su primer sentencia.

- La llamada a super no se genera, es completamente opcional.

Método polimórfico

Lo que hemos discutido en las secciones anteriores, desde Tipo estático y tipo dinámico hasta ahora, es lo que se conoce como despacho de método polimórfico (o mas simplemente, Polimorfismo).

Recuerde que una variable polimórfica es aquella que puede almacenar objetos de diversos tipos (cada variable objeto en lava es potencialmente polimórfica). De manera similar, las llamadas a métodos en lava son polimórficas dado que ellas pueden invocar diferentes métodos en diferentes momentos. Por ejemplo, la sentencia **elemento.imprimir();** puede invocar al método imprimir de CD en un momento dado y al método imprimir de DVD en otro momento, dependiendo del tipo dinámico de la variable elemento.

Bueno, no hay mucho más por ver en herencia y polimorfismo. Claro que para consolidar esto necesitamos **verlo funcionando**.

Para hacer mas completo el demo de polimorfismo, vamos a incorporar un elemento más: **Libro**, que extiende directamente **Elemento**, sin incorporarle ningún atributo adicional.

```
import java.util.ArrayList;
public class BaseDeDatos{
    private ArrayList<Elemento> elementos;
    protected String auxStr;
    public BaseDeDatos(){     //    constructor
        elementos = new ArrayList<Elemento>();
    }
```

```
    public void agregarElemento (Elemento elElemento){
        elementos.add(elElemento);
    }

    public String toString(){ // Cadena con todos los elementos contenidos
        auxStr = "Contenidos BaseDeDatos\n";
        auxStr+=elementos.toString();
        return auxStr;
    }
}
```

```
package dome;
public class Elemento{
    private String titulo;
    private int duracion;
    private boolean loTengo;
    private String comentario;

    public Elemento(String elTitulo, int tiempo){
        titulo = elTitulo;
        duracion = tiempo;
        loTengo = false;
        comentario = "";
    }

    public String toString(){
        String aux = titulo + " (" + duracion + " minutos) ";
        if (loTengo)aux += "*";
        aux += "  " + comentario+"\n";
        return aux;
    }
}
```

```
package dome;
public class CD extends Elemento{
    private String interprete;
    private int numeroDeTemas;
    public CD(String elTitulo, String elInterprete, int temas, int tiempo){
        super(elTitulo, tiempo);
        interprete = elInterprete;
        numeroDeTemas = temas;
    }

    public String toString(){
        String aux = super.toString();
               aux+= "  interprete (CD): " + interprete+"\n";
               aux+= "  temas: " + numeroDeTemas+"\n";
        return aux;
    }
}
```

```
package dome;
public class DVD extends Elemento{
    private String director;
    public DVD(String elTitulo, String elDirector, int time){
        super(elTitulo, time);
```

```
        director = elDirector;
    }

    public String toString(){
        String aux = super.toString();
               aux+= "  director (DVD): " + director+"\n";
               return aux;
    }
}
```

```
package dome;
public class Libro extends Elemento{
    public Libro(String elTitulo, int time){
        super(elTitulo, time);
    }
}
```

```
package dome;
// @author Tymoschuk, Jorge
public class Main {
    private BaseDeDatos db;
    public void DemoBaseDedatos(){
        System.out.println("Demo inicia");
        db = new BaseDeDatos();
        Elemento elem;
        //  Incluyo 2 CDs
        elem = new CD("Pajaros en la Cabeza","Amaral",14,35);
        db.agregarElemento(elem);
        elem = new CD("One chance","Paul Pots",10,30);
        db.agregarElemento(elem);
        //  Incluyo 2 DVDs
        elem = new DVD("Soy Leyenda","Francis Lawrence",120);
        db.agregarElemento(elem);
        elem = new DVD("Nada es Para Siempre","Robert Redford",105);
        db.agregarElemento(elem);
        //  Incluyo dos libros
        elem = new Libro("El Señor de los Anillos",5000);
        db.agregarElemento(elem);
        elem = new Libro("El Don Apacible",10000);
        db.agregarElemento(elem);
        //  veamos que hemos hecho
        System.out.println(db.toString());
        System.out.println("Demo terminado");
    }
    public static void main(String[] args) {
        Main demo = new Main();
        demo.DemoBaseDedatos();
    }
}
```

```
run:
Demo inicia
Contenidos BaseDeDatos
[Pajaros en la Cabeza (35 minutos)
      interprete (CD): Amaral
      temas: 14
, One chance (30 minutos)
      interprete (CD): Paul Pots
      temas: 10
, Soy Leyenda (120 minutos)
      director (DVD): Francis Lawrence
, Nada es Para Siempre (105 minutos)
      director (DVD): Robert Redford
, El Señor de los Anillos (5000 minutos)
, El Don Apacible (10000 minutos)
]
Demo terminado
BUILD SUCCESSFUL (total time: 7 seconds)
```

La sentencia **System.out.println(db.toString())**, método public void **DemoBaseDedatos()** es la que se ejecuta inicialmente. Esta sentencia:

- Incorpora en la cadena el resultado de **elementos.toString.** Como **elementos** es una instancia de ArrayList, usa el toString() de esta clase (De ahí los corchetes de cierre y las comas separadoras).
- **elementos** contiene 6 instancias de la variable polimórfica Elemento:
- las dos primeras tienen tipo dinámico CD. Entonces, en la ejecución del toString() propio invocan super.toString() (el de Elemento) y luego completan con los datos específicos de CD.
- Las dos siguientes tienen tipo dinámico DVD. Proceden exactamente lo mismo que CD.
- Las dos últimas instancias tienen tipo dinámico Libro. Como no tienen toString() propio, el despacho dinámico encuentra el de Elemnto y este es el que se ejecuta.

Complicado o facil? En todo caso, la programación es muy sintética, nada de sobreescritura, cada parte del armado de la cadena que imprime **System.out.println(db.toString())** lo hace el método del objeto responsable de ello, como manda la POO.

.
.
.
.
.

PAQUETES (package)

Los paquetes son una forma de organizar grupos de clases. Un paquete contiene un conjunto de clases relacionadas bien por finalidad, por ámbito o por herencia.

Los paquetes resuelven el problema del conflicto entre los nombres de las clases. Al crecer el número de clases crece la probabilidad de designar con el mismo nombre a dos clases diferentes.

Las clases tienen ciertos privilegios de acceso a los miembros dato y a las funciones miembro de otras clases dentro de un mismo paquete.

En el Entorno Integrado de Desarrollo (IDE) JBuilder de Borland, un proyecto nuevo se crea en un subdirectorio que tiene el nombre del proyecto. A continuación, se crea la aplicación, un archivo .java que contiene el código de una clase cuyo nombre es el mismo que el del archivo. Se pueden agregar nuevas clases al proyecto, todas ellas contenidas en archivos .java situadas en el mismo subdirectorio. La primera sentencia que encontramos en el código fuente de las distintas clases que forman el proyecto es package o del nombre del paquete.

```
//archivo MiApp.java

package nombrePaquete;
public class MiApp{
      //miembros dato
      //funciones miembro
}
//archivo MiClase.java

package nombrePaquete;
public class MiClase{
      //miembros dato
      //funciones miembro
}
```

La palabra reservada import

Para importar clases de un paquete se usa el comando import. Se puede importar una clase individual

```
import java.awt.Font;
```

o bien, se puede importar las clases declaradas públicas de un paquete completo, utilizando un arterisco (*) para reemplazar los nombres de clase individuales.

```
import java.awt.*;
```

Para crear un objeto fuente de la clase Font podemos seguir dos alternativas

```
import java.awt.Font;
Font fuente=new Font("Monospaced", Font.BOLD, 36);
```

O bien, sin poner la sentencia import

```
java.awt.Font fuente=new java.awt.Font("Monospaced", Font.BOLD, 36);
```

Normalmente, usaremos la primera alternativa, ya que es la más económica en código, si tenemos que crear varias fuentes de texto.

Se pueden combinar ambas formas, por ejemplo, en la definición de la clase BarTexto

```
import java.awt.*;
public class BarTexto extends Panel implements java.io.Serializable{
//...
}
```

Panel es una clase que está en el paquete java.awt, y Serializable es un interface que está en el paquete java.io

Algunos paquetes estándar

Paquete	**Descripción**
java.applet	Contiene las clases necesarias para crear applets que se ejecutan en la ventana del navegador.
java.awt	Contiene clases para crear una aplicación GUI independiente de la plataforma.
java.io	Entrada/Salida. Clases que definen distintos flujos de datos.
java.lang	Contiene clases esenciales, se importa impícitamente sin necesidad de una sentencia import.
java.net	Se usa en combinación con las clases del paquete java.io para leer y escribir datos en la red.
java.util	Contiene otras clases útiles que ayudan al programador.

Hay muchos mas …

Resumen

- Las clases pueden organizarse en **paquetes**
- Un paquete se identifica con la cláusula **package nombre_paquete**, al principio de un archivo fuente.
- Solo una declaración **package** por archivo.
- Varios archivos pueden pertenecer al mismo **paquete**
- Un archivo sin declaración **package** pertenece al **paquete unnamed**
- Los nombres de los paquetes están relacionados con la organización de los archivos
- **Los directorios padre de los paquetes deben incluirse en CLASSPATH** (Variable de entorno del sistema operativo)

- Al compilar o ejecutar, las clases de biblioteca proporcionadas por Java (Java, Javac, JDK) se cargan automáticamente, otras deben especificarse en CLASSPATH.

```
SET CLASSPATH= C:\JAVA; //Hasta el directorio padre de paquetes
```

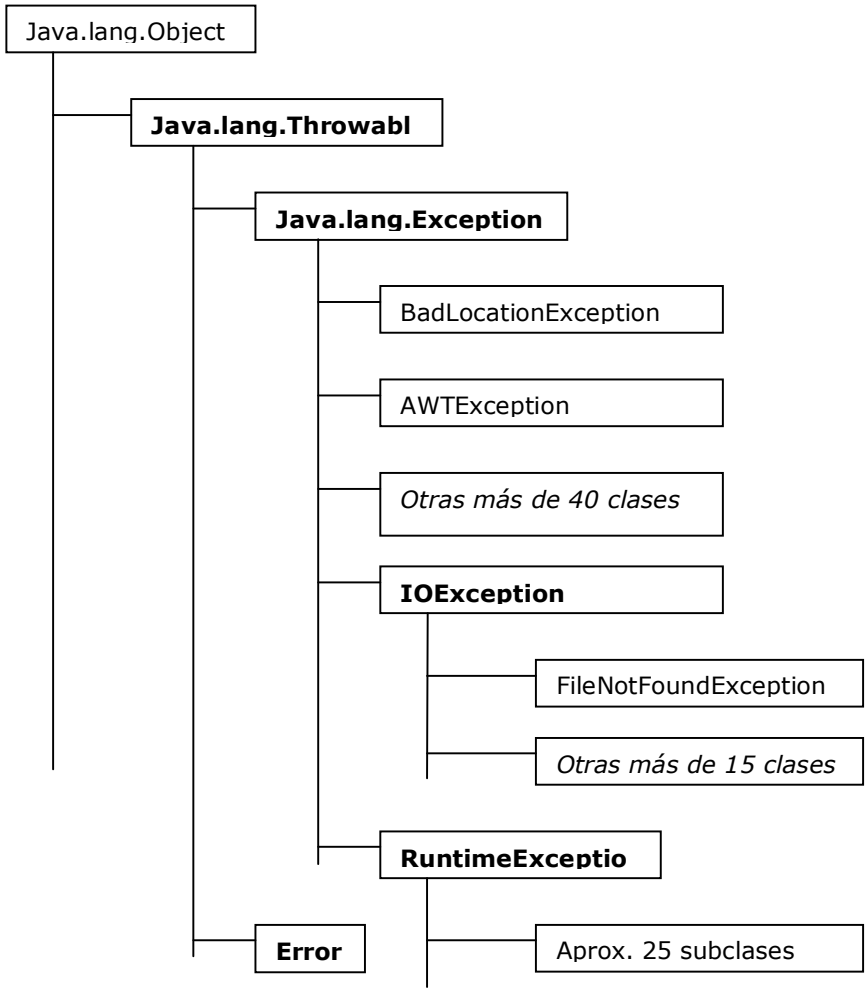
- Para incorporarlas en mi programa, puedo

```
import mis Paquetes.*          // Todas las clases que alli existan
import mis Paquetes.Madre      // Solo la clase madre
```

- Para nombres de paquetes, recomendable usar nombres de dominio de Internet
- La compilación de un paquete produce tantos archivos .class como clases tenga el paquete.
- Los archivos .class se graban en el mismo directorio del fuente .Java, a menos que la opción –d indique otro destino

Tratamiento de Excepciones

Las excepciones son eventos inesperados que suceden durante la ejecución de un programa. Una excepción se puede deber a una condición de error, o tan sólo a un dato no previsto. En cualquier caso, en un lenguaje orientado a objetos, como lo es Java, se puede considerar que las excepciones mismas son objetos de clases que extienden, directa o indirectamente, Java.lang.exception.



Como se puede ver en esta jerarquía de clases, Java define las clases Exception y Error como subclases de Throwable, que denota cualquier objeto que se puede lanzar y atrapar. También, define la clase RuntimeException como subclase de Exception.

La clase Error se usa para condiciones anormales que se presenten en el ambiente de ejecución, como cuando se acaba la memoria. Los errores se pueden atrapar, pero es probable que no, porque en general señalan problemas que no se pueden manejar con elegancia. Un mensaje de error, seguido de la terminación repentina del programa es lo máximo que podemos pretender.

La clase Exception es la raíz de la jerarquía de excepciones. Se deben definir las excepciones especializadas (por ejemplo, la BoundaryViolationException) subclasificadas por Exception o RuntimeException. Nótese que las excepciones que no sean subclases de RuntimeException se deben declarar en la cláusula **throws** de cualquier método que las pueda lanzar.

Dijimos que las excepciones son objetos. Todos los tipos de excepción (es decir, cualquier clase diseñada para objetos lanzables) debe extender la clase Throwable o una de sus subclases. La clase Throwable contiene una cadena de texto que se puede utilizar para describir la excepción. Por convenio, los nuevos tipos de excepción extienden a Exception, una subclase de Throwable.

Las excepciones son principalmente **excepciones comprobadas** (*también llamadas verificadas o síncronas)*, lo que significa que el compilador comprueba que nuestros métodos lanzan sólo las excepciones que ellos mismos han declarado que pueden lanzar. Las excepciones y errores estándar en tiempo de ejecución extienden una de las clases **RuntimeException** o **Error**, y se denominan **excepciones no comprobadas** *(También llamadas asíncronas).* Todas las excepciones que generemos deberían extender a **Exception**, siendo así excepciones comprobadas.

Las excepciones comprobadas representan condiciones que, aunque excepcionales, se puede esperar razonablemente que ocurran, y si ocurren deben ser consideradas de alguna forma. Al hacer que estas excepciones sean comprobadas se documenta la existencia de la excepción y se asegura que el que llama a un método tratará la excepción de alguna forma. Las excepciones no comprobadas representan condiciones que, hablando en términos generales, reflejan errores en la lógica de nuestro programa de los que no es posible recuperarse de forma razonable en ejecución. Por ejemplo, una excepción **IndexOutOfBoundsException** que se lanza al intentar acceder fuera de los límites de un array, nos indica que nuestro programa calcula los índices de forma incorrecta, o que falla al verificar un valor que se va a utilizar como índice. Estos errores deben corregirse en el código del programa. Como pueden producirse errores al escribir cualquier sentencia, sería totalmente imposible tener que declarar y capturar todas las excepciones que podrían surgir de estos errores, de ahí la existencia de las excepciones no comprobadas.

Algunas veces es útil tener más datos para descríbir la condición excepcional, además de la cadena de texto que proporciona Exception. En esos casos se puede extender a Exception creando una clase que contenga los datos añadidos (que generalmente se establecen en el constructor). Esto lo veremos mas adelante.

Lanzamiento de excepciones

En Java, las excepciones son objetos *"lanzados"* por un código que encuentra una condición inesperada. También pueden ser arrojadas por el ambiente de ejecución en Java, si llega a encontrar una condición inesperada, como por ejemplo si se acaba la memoria de objetos. Una excepción lanzada puede ser es *atrapada* por otro programa que la "maneja" de alguna manera, o bien el programa se termina en forma inesperada.
Las excepciones se originan cuando un fragmento de programa Java encuentra alguna clase de problemas durante la ejecución y *lanza* un objeto de excepción, al que se identifica con un nombre descriptivo.
Hagamos un primer ejemplo. Tomamos dos arreglos de elementos int de distinta longitud, y hacemos un ciclo donde nume[i] es dividido por deno[i]. El denominador tiene un 0 en el cuatro casillero, así que esperamos algún problema, sabido es que no se puede dividir por 0. También el hecho de ser los arreglos de distinta longitud, si comandamos el ciclo en función del más largo, debe dar problemas.

```
class Excep01{          //          Su ejecución
      static int nume[] = {10,20,30,40,50};
      static int deno[] =  {  2, 4,   6,  0,10

      public static void excep() {
            int i;
            float coc;
            System.out.println("\nExcepciones,
            for(i = 0; i < deno.length; i++){
```

```
Excepciones, ej 01
Cociente 0 vale 5.0
Cociente 1 vale 5.0
Cociente 2 vale 5.0
Cociente 3 vale Infinity
Cociente 4 vale 5.0
java.lang.ArrayIndexOutOfBoundsException
      at Excep01.excep(Excep01.java:12)
      at Excep01.main(Excep01.java:18)
Exception in thread "main"
Process Exit...
```

```
                coc = (float)nume[i]/deno[i];
                System.out.println("Cociente "+i+" vale "+coc);
            }
        }
        public static void main(String args[]){
            excep();
        }
}
```

Hay una sorpresa. La división por 0 no causa una excepción, como esperábamos. El println nos dice: **Cociente 3 vale Infinity**. El tema de ir mas allá del fin del vector nume causa la excepción esperada, *java.lang.**ArrayIndexOutOfBoundsException,*** (Excepción de índice fuera de los limites del vector). Como nosotros no capturamos ni tratamos esta excepción, ella llega a la Maquina Virtual Java que nos informa que ella ocurrió en la línea 12 del código fuente Java del método excep() de la clase Excep01. Este método excep(), fué a su vez invocado en la línea 18 del método main(..) de la misma clase.

Nos preocupa la sorpresa de la división por cero. Muchos libros de programación, incluso Java ejemplifican la división por cero como una **excepción típica**. Parece no ser tan así, o por lo menos aquí no ocurrió. También se encuentran ejemplos del método Carácter.toLowerCase((char) c), (Transforma mayúsculas a minúsculas). Se supone que si lo que tiene que transformar no es una letra habría problemas. Pues no los hay. Volviendo a nuestra división por cero, será que se puede seguir operando con una variable que vale **Infinity**? Hagamos un programita.

```
import java.io.IOException;
class Infinity{
        public static void infinito() {
            double a = 10.0, b = 0.0, c, d,e = 5.0, f,g;
            System.out.println("\nPreocupandonos por el Infinito\n");
            c = a+e;
            System.out.println("valor de c = "+c);
            d = a/b;    // Supuestamente Infinity
            System.out.println("valor de d = "+d);
            e+=d;       // Supuestamente Infinity
            System.out.println("valor de e = "+e);
            f = a - d;  // Supuestamente -Infinity
            System.out.println("valor de f = "+f);
            g = d/e;    // Que pasara ?
            System.out.println("valor de g = "+g);
        }
        public static void main(String args[]){
            infinito();
        }
}
```

```
Preocupandonos por el Infinito
 valor de c = 15.0
 valor de d = Infinity
 valor de e = Infinity
 valor de f = -Infinity
 valor de g = NaN
 Process Exit...
```

Ahora lo sabemos. En Java, (por lo menos para tipo de dato double) se puede dividir por cero y el valor resultante es infinito. Este valor resultante es operable, por lo menos en algunas operaciones. Volveremos sobre esto. Ahora retornemos a nuestras excepciones: vamos a capturarla y tratarla nosotros.

Atrapado de excepciones

Cuando se lanza una excepción ésta debe *atraparse* o de lo contrario el programa se terminará. En cualquier método particular puede atraparse la excepción en el generada, o bien se la puede "pasar" (Devolver, retroceder) hasta alguno de los métodos de la pila de llamadas, la última de las cuales activo nuestro método. Cuando se atrapa una excepción se puede analizar y manejar. La metodología general para manejar excepciones es *"tratar"* (try) de ejecutar algún fragmento de programa

que pueda lanzar una excepción. Si se lanza una excepción, esa excepción queda *atrapada* y hace que el flujo de control salte a un bloque catch predefinido. Dentro del bloque **catch** se podrá entonces manejar la circunstancia excepcional.

Atrapando excepciones en el propio método

```
import java.io.IOException;
class Excep02{                //          Ejecución ➔
      static int nume[] = {10,20,30,40,50};
      static int deno[] = { 2, 4, 6, 0,10,12};
      public static void excep() {
            int i;
            float coc;
            try{
                  System.out.println("\nExcepciones, ej 02");
                  for(i = 0; i < deno.length; i++){
                        coc = (float)nume[i]/deno[i];
                        System.out.println("Cociente "+i+" vale "+coc);
                  }
                  System.out.println("Saliendo del bloque de prueba ...");
            }catch(ArrayIndexOutOfBoundsException e){
                  System.out.println("Excepcion capturada !!!");
            }
            System.out.println("Exit excep(), class Excep02");
      }
      public static void main(String args[]){
            excep();
      }
}
```

```
Excepciones, ej 02
Cociente 0 vale 5.0
Cociente 1 vale 5.0
Cociente 2 vale 5.0
Cociente 3 vale Infinity
Cociente 4 vale 5.0
Excepcion capturada !!!
Exit    excep(),    class
Excep02
Process Exit...
```

Que pasó. Se produce la excepción tipo **ArrayIndexOutOfBoundsException** dentro del bloque de prueba (bloque **try**) y es capturada por la cláusula **catch(ArrayIndexOutOfBoundsException e)**. La captura es posible porque el catch captura eventos de excepción del tipo **ArrayIndexOut…** El tratamiento que le damos es solamente imprimir el mensaje ("Excepcion del tipo ArrayIndexOut... capturada !!!") y nada mas. Después de esta captura termina la ejecución.

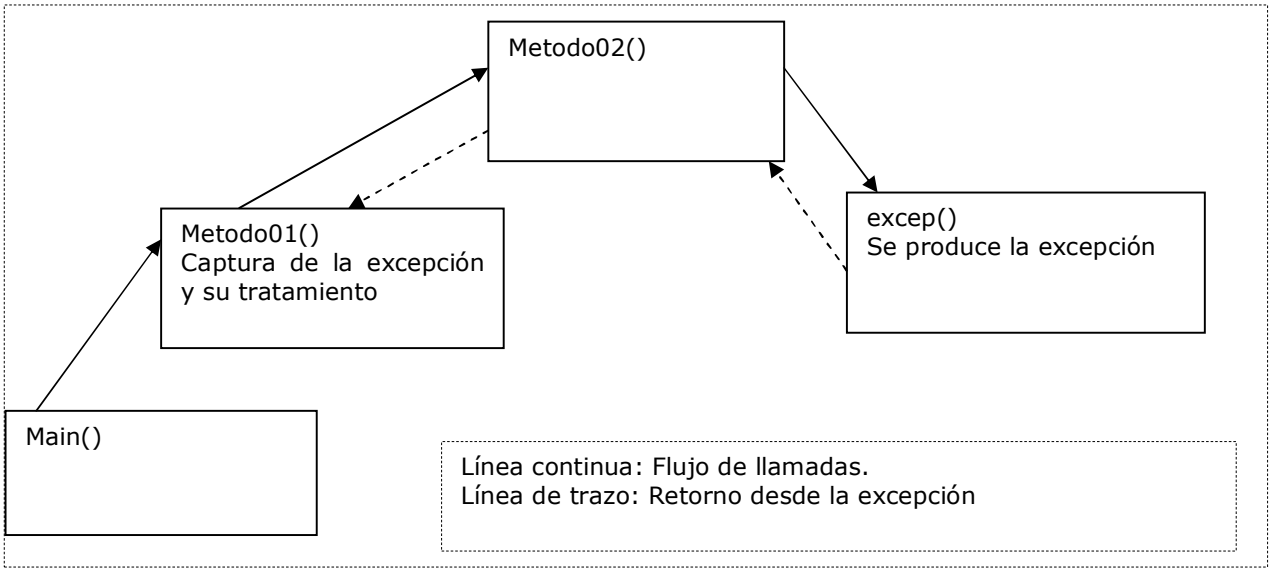
El ambiente Java comienza ejecutando el **try{bloque_de_instrucciones_}**. Si esa ejecución **no genera excepciones**, el flujo del control continúa con el primer paso de programa después del último renglón de todo el bloque **try-catch**, a menos que ese bloque incluya un bloque finally opcional. El bloque finally, de existir, se ejecuta independientemente de si las excepciones son lanzadas o atrapadas. Así, en este caso, si no se lanza alguna excepción, la ejecución avanza por el bloque try, salta al bloque finally y entonces continúa con el primer renglón que siga al último renglón del bloque try-catch.

Por otro lado, si **try{bloque_de_instrucciones_}** genera una excepción, la ejecución en el bloque try termina en ese punto, y salta al bloque catch **cuyo tipo coincida con el de la excepción**. Si la clase de la excepción generada es una subclase de la declarada en catch, la excepción **será también capturada**. Una vez que se completa la ejecución de ese bloque catch, el flujo pasa al bloque opcional finally, si existe, o bien de inmediato a la primera instrucción después del último renglón de todo el bloque try-catch, si no hay bloque finally. De otro modo, si no hay bloque catch que coincida con la excepción lanzada, el control pasa al bloque finally opcional si existe, y entonces la excepción se lanza de regreso al método llamador.

Hemos visto un ejemplo de una excepción atrapada en el bloque match del propio método que genera la excepción. Ahora veremos el segundo caso, el método no tiene catch para el tipo de excepción producida. En realidad, en el ejemplo, no tiene ningún catch

Atrapando excepciones "pasadas" desde métodos posteriores.

Vamos al mismo caso anterior, pero ejemplificando una excepción que no es capturado en el método donde se produce. La excepción se produce en el último método, excep(), es capturada en el primero, Metodo01(), allí es tratada y, en nuestro ejemplo, termina el proceso. Gráficamente.




```
import java.io.IOException;
class Excep03{
      static int nume[] = {10,20,30,40,50};
      static int deno[] = { 2, 4, 6, 0,10,12};
      public static void Metodo01(){
            System.out.println("\nEntrada a Metodo01()");
            try{  //  en el bloque de prueba esta la llama
                  Metodo02();
            }catch(ArrayIndexOutOfBoundsException e){
                  System.out.println("Capturada en Metodo0
            }
            System.out.println("Salida de Metodo01()");
      }
      private static void Metodo02(){
            System.out.println("Entrada a Metodo02()");
            excep();
            System.out.println("Salida de Metodo02()");
      }
      private static void excep() throws ArrayIndexOutOfBoundsException{
            int i;
            float coc;
            System.out.println("Entrada a excep()");
            for(i = 0; i < deno.length; i++){
                  coc = (float)nume[i]/deno[i];
                  System.out.println("Cociente "+i+" vale "+coc);
            }
            System.out.println("Salida de excep(), Excep03");
      }
      public static void main(String args[]){
            Metodo01();
      }
}
```

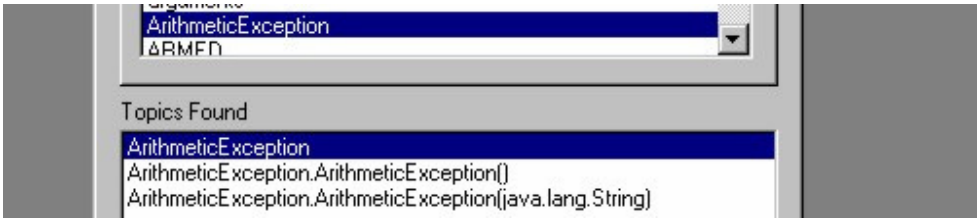
```
Entrada a Metodo01()
Entrada a Metodo02()
Entrada a excep()
Cociente 0 vale 5.0
Cociente 1 vale 5.0
Cociente 2 vale 5.0
Cociente     3     vale
Infinity
Cociente 4 vale 5.0
Capturada            en
Metodo01()!!!
Salida de Metodo01()
Process Exit...
```

La excepción se produce, como antes, en el método excep(), pero ahora no se la captura, solo declaramos que se dispara, mediante la **cláusula throws**. El

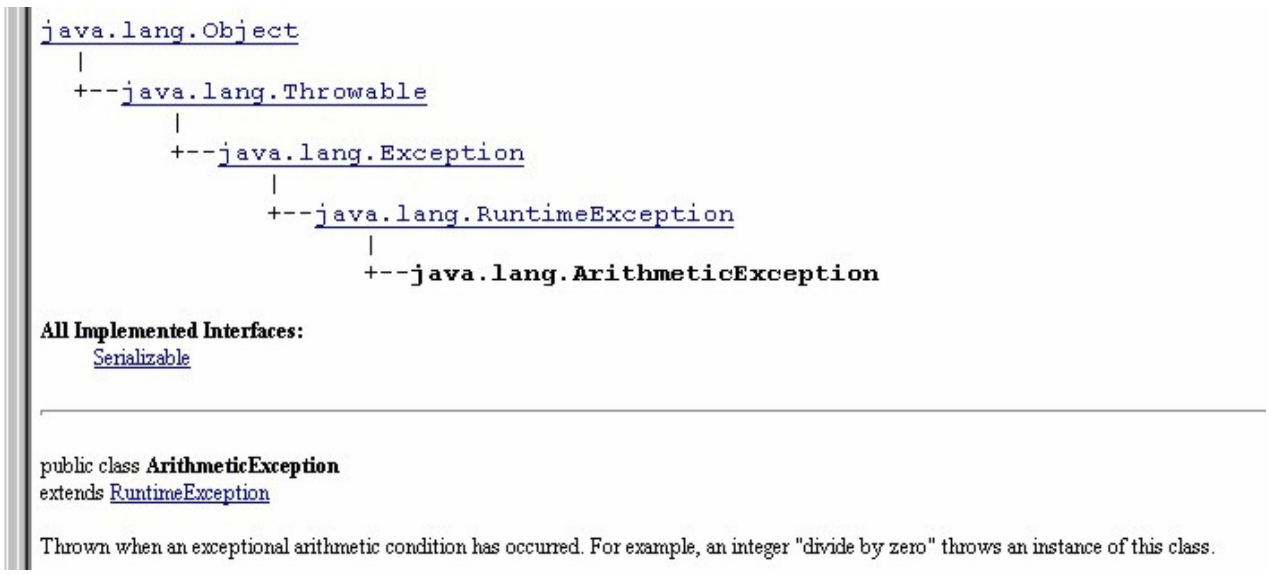
tratamiento de excepciones se realiza en metodo01(). Allí tenemos el bloque try y la cláusula de captura. En el seguimiento de la ejecución vemos que efectivamente hay un retroceso hasta el método llamador que contiene la captura.
Si consideramos que en general cada método contiene decisiones que definen sobre cual método se invoca a seguir, esto significa que en cada uno de ellos podemos continuar por distintos caminos. Pero en el punto de partida, o en un punto intermedio, o selectivamente, podremos tratar las excepciones generadas mas adelante. La captura es por tipo de objeto de excepción, esto significa que podemos capturar selectivamente. Esta captura será tanto de excepciones ya previstas en el lenguaje Java como propias nuestras, puesto que podemos definir nuestras propias excepciones, extendiendo clases existentes. Y ya que estamos, generemos una excepción para la división por cero.

Generando nuestras propias excepciones.

Hemos dicho que la captura es por tipo de excepción y que si el tipo tiene subclases, serán atrapadas allí. O sea que si declaramos una cláusula match con tipo Excepción, allí quedarían capturadas la mayoriía de las excepciones, cosa que no deseamos. Necesitamos ser más específicos.
Una excepción de división por cero es del tipo aritmético. Si entramos en el help de Java, por contenido y tipeamos Arithmetic <enter>, obteneos varios tópicos: la clase y dos constructores.



Seleccionamos el primero y <display >
La parte superior del capture abajo muestra el esquema hereditario al cual la clase ArithmeticException está subordinada. Pertenece al paquete java.lang y extiende RuntimeException. Si traducimos la última línea: *Disparada cuando ha ocurrido una condición aritmética excepcional. Por ejemplo, una división* ***"entero por cero"*** *dispara una instancia de esta clase.*



Ahora las cosas comienzan a aclararse. Solo la división por cero en números

enteros produce una excepción. Nosotros dividimos por cero números reales, allí no ocurre esta excepción. Si queremos que esto ocurra, podemos codificar una clase que extienda ArithmeticException y hacerlo allí.

```
import java.io.IOException;
class RealDivideExcep extends ArithmeticException{
      static int nume[] = {10,20,30,40,50};
      static int deno[] = { 2, 4, 6, 0,10,12};

      public RealDivideExcep(){
            super();
      }
      public void excep(){//  Ejecución ➔
            int i; float coc;
            try{
                  System.out.println("\nExcepciones, ej 04");
                  for(i = 0; i < deno.length; i++){
                        if (deno[i] == 0)       //  La situación que nos preocupa
                              throw new RealDivideExcep(); //           Disparamos
excepción
                        coc = (float)nume[i]/deno[i];
                        System.out.println("Cociente "+i+" vale "+coc);
                  }
                  System.out.println("Saliendo del bloque de prueba ...");
            }catch(ArrayIndexOutOfBoundsException e){
                  System.out.println("Excepcion, ArrayIndexOut... capturada !!!");
            }catch(ArithmeticException e){
                  System.out.println("Excepcion,    RealDivideExcep...    capturada
!!!");
            }
            System.out.println("Exit RealDivideExcep()");
      }
      public static void main(String args[]){
            RealDivideExcep rDivEx = new RealDivideExcep();
            rDivEx.excep();
      }
}
```

```
Excepciones, ej 04
Cociente 0 vale 5.0
Cociente 1 vale 5.0
Cociente 2 vale 5.0
Excepcion, RealDivideExcep... capturada
!!!
Exit RealDivideExcep()
```

Observamos que:
Declarando el catch con la clase **ArithmeticException** capturamos la **RealDivideExcep**.
La excepción **ArrayIndexOutOfBoundsException** no llega a producirse, salimos antes.

Algoritmos de búsqueda, recorrido y ordenamiento

Cuando vimos el tema Agrupar objetos, dijimos que Java nos permite organizarlos usando muy distintas herramientas y hay que estudiar bastante antes de decidir cual de ellas es la más adecuada a nuestra necesidad.

En efecto, y solo a modo introductorio, distingamos tres distintas situaciones.

- La colección contiene un **número fijo de elementos**. Tratar esto usando arrays, ya lo vimos en Agrupar objetos en colecciones de tamaño fijo.
- La colección contiene un **número variable de elementos** y ademas será tratada concurrentemente en varios hilos (flujos independientes) simultáneamente. Java provee la clase Vector.
- La colección contiene un **número variable de elementos** y será tratada por un único flujo. Nos conviene usar la clase **ArrayList**, similar a Vector, pero más rápida, por no necesitar preocuparse por la concurrencia.

En lo que sigue comenzaremos tratando en detalle la primera situación, arrays. Por ser esta implementación del agrupamiento la primera (Y única por decadas) existe una muy grande cantidad de algoritmos que la tratan. Pero que Java la trate no deja

de ser compatibilidad con el pasado. Entonces considerando la gran funcionalidad y rico comportamiento de la clase de colección **arrayList**, que tiene toda la capacidad de los arrays, y un importante comportamiento adicional, veremos también algunos algoritmos usando este agrupamiento de objetos.

Búsqueda en arreglos (arrays)

Con frecuencia es necesario determinar si un elemento de un array contiene un valor que coincide con un determinado *valor clave.* El proceso de determinar si este elemento existe se denomina búsqueda. Existen distintas técnicas de búsqueda, vamos a estudiar dos: secuencial y binaria.

Un ejemplo de la vida cotidiana. Estamos comprando en un supermercado. El operador de caja presenta el producto al lector de código de barras. Y el sistema le devuelve el valor que se imprime en el ticket.
Normalmente el valor no es lo que está codificado en las barras. El valor puede variar, el producto puede ponerse en oferta, o encarecerse. Lo que viene en el código de barras es el código del producto. (perdón por la redundancia).
El código de producto está asociado a su valor. Hay muchas maneras de almacenar esta asociación, pero como estamos estudiando arrays vamos a suponer tenemos un array de objetos items conteniendo asociaciones asociaciones código /valor. Entonces tenemos que ***buscar el código*** *en el array para obtener su valor asociado.*

Si vamos a tener un array de items código/valor, necesitamos, primero de una clase con lo relacionado con el comportamiento mínimo de los items. Inicializar, leer. Será nuestra **class Item**. *Y bueno, si esos items estarán almacenados en un array de items, tendremos que preocuparnos por la obtención de área, carga de items en el array, mostrarlos ... definiremos todo esto en la* ***class ArrItems***. *Una vez que tengamos todo esto funcionando comenzaremos a aprovecharlo en las aplicaciones clásicas de los arrays: buscar, ordenar, actualizar, etc, etc .....*

```
class Item {    // Una clase de claves asociadas a un valor ...
      protected int codigo;
      protected float valor;
      public Item(){};          //  Constructor sin argumentos
      public      Item(int cod, float val){ //  Constructor de la clase
             codigo=cod;
             valor=val;
      }
      public      String      toString(){
            String aux = "";
            aux+=codigo+" - "+valor;
            return aux;
      }          //      Exhibimos
      public      int   getCodigo(){return codigo;}
      public      float getValor(){return valor;}
      public      Item  getItem(){return this;}
      public      void  setCodigo(int cod){codigo=cod;}
      public      void  setValor(float val){valor=val;}
      public      boolean esMayor(Item item){// Es mayor invocante que parámetro?
            return(codigo > item.codigo?true:false);
            // Si clave del objeto invocante > clave objeto parámetro,
      }     // retorno true, caso contrario false

      public      boolean esMenor(Item item){
            return(codigo < item.codigo?true:false);
                  // Si clave del objeto invocante < clave objeto parámetro,
      }           // retorno true, caso contrario false;

      public      void  intercambio(Item item){
            Item burb= new Item(item.codigo,item.valor);
            // intanciamos burb con datos parametro
```

```
            item.codigo = this.codigo;
            // asignamos atributos del objeto invocante al objeto parametro
            item.valor  = this.valor;
            this.codigo = burb.codigo;    // asignamos atributos del objeto burb al
            this.valor  = burb.valor;     // objeto invocante
      }
}

import Item;
class ArrItems{                     //    Una clase de implementación de un
      protected Item[] item;       // array de items, comportamiento mínimo ...
      protected int talle;          //  Tamaño del array de objetos Item
      public ArrItems(int tam, char tipo) {// Constructor de un array de Item's
            int   auxCod = 0;         // Una variable auxiliar
            talle=tam;                              //     inicializamos     talle
(tamaño)
            item = new Item[talle];    // Generamos el array
            for(int i=0;i<talle;i++){     // la llenaremos de Item's, dependiendo
                  switch(tipo){          // del tipo de llenado requerido
                        case 'A':{        // los haremos en secuencia ascendente
                              auxCod = i; break;
                        }
                        case 'D':{        // o descendente ...
                              auxCod = talle - i; break;
                        }
                        case 'R':{        // o bien randomicamentente (Al azar)
                              auxCod = (int)(talle*Math.random());
                        }
                  }
                  item[i] = new Item();
                  item[i].setCodigo(auxCod);
                  item[i].setValor((float)(talle*Math.random()));
            }
            System.out.print(this);
      }

      public String toString(){
            int ctos = (talle < 10 ? talle : 10);
            String aux = " Primeros "+ctos+" de "+talle+"\n elementos Item\n";
            for(int i=0;i<ctos;i++)
                  aux+=item[i].toString()+"\n";
            return aux;
      }
}
```

Búsqueda secuencial

La búsqueda secuencial verifica la existencia de un valor denominado clave en un array. En una búsqueda secuencial los elementos de una lista o vector se exploran (se examinan) en secuencia, uno después de otro. Es el único método de búsqueda posible cuando el array no está ordenado.
El algoritmo de búsqueda secuencial compara cada elemento del array con la *clave* de búsqueda. Dado que el array no está en un orden prefijado, es igualmente posible que el elemento a buscar sea el primero, el último o cualquier otro. En promedio, se tendrá que comparar la clave de búsqueda con la mitad de los elementos del array. Este método de búsqueda, cuyo algoritmo es sencillo, es el único posible con arrays no ordenados.

```
package busqsec;
class BusqSec{          // Busqueda secuencial en un array de items
   int posic;     //  Posicion del ultimo encuentro
   int clav;       //  Clave buscada
   ArrItems arrItems;  //  El array donde buscaremos
```

```
   public  BusqSec(int cant, char tipo){  // Un constructor
      arrItems = new ArrItems(cant,tipo); // que invoca otro
      posic = arrItems.talle+1;     //  posici¢n fuera del array
      clav  = 0;
   }

   protected void setClav(int clave){
      clav = clave;
   }

   protected int getClav(){
      return clav;
   }

   protected int leerClav(){
      clav = In.readInt();
      return clav;
   }

   protected boolean existe(int clave){;  // Existe la clave parámetro ?
      boolean exist = false;
      for(int i=0;i<arrItems.talle;i++)
            if (clave==arrItems.item[i].getCodigo()){
               posic = i; exist = true;
            }
      return exist;
   }

   protected int cuantas(int clave){; // Cuantas veces existe la clave ?
      int igual=0;                        //  Cuantos iguales ...
      for(int i=0;i<arrItems.talle;i++)
            if(clave==arrItems.item[i].getCodigo())igual++;
      return igual;
   }

   protected void demoBusqSec(){
      int cant;
      System.out.println("Que clave? (999: fin) ");
      while (leerClav() != 999){
         if((cant = cuantas(clav)) > 0)
            System.out.println("Código " + clav + ", existe " + cant+" veces");
         else
            System.out.println("Codigo " + clav + " inexistente");
      }
      System.out.println("Demo finished!!!\n");
   }
};
```

```
package busqsec;
/**
 *
 * @author Tymoschuk, Jorge
*/
public class Main {
      public Main(){}
      public static void main(String[] args) {
            BusqSec busqSec = new BusqSec(100,'R');
            busqSec.demoBusqSec();
      }
}
```

```
run:
 Primeros 10 de 100
 elementos Item
93 - 71.66544
64 - 82.801315
18 - 95.81637
92 - 29.936878
46 - 97.975876
22 - 32.22439
62 - 43.16789
41 - 51.03334
47 - 95.86333
77 - 53.540943
Que clave? (999: fin)
12
Codigo 12 inexistente
Código 41, existe 2 veces
41
44
Código 44, existe 2 veces
999
Demo finished!!!
```

Búsqueda binaria

La búsqueda secuencial se aplica a cualquier array. Si está ordenado, la *búsqueda binaria* proporciona una técnica de búsqueda mejorada. Una búsqueda binaria típica es la búsqueda de un número en un directorio telefónico o de una palabra en un diccionario. Dado la palabra, se abre el libro cerca del principio, del centro o del final dependiendo de la primera letra de la palabra que se busca. Se puede tener suerte y acertar con la página correcta; pero, normalmente, no será así y el lector deberá probar en páginas anteriores o posteriores.
La misma idea se aplica en la búsqueda en un array ordenado. Nos posicionamos en el centro del array y se comprueba si nuestra clave coincide con la del elemento. Si no, tenemos dos situaciones:

La clave es mayor: debemos buscar en el tramo superior.
La clave es menor: la búsqueda será en el tramo inferior.

y este razonamiento se aplicará las veces necesarias hasta encontrar igual o definir que la clave no existe.

```
package busqbin;
class BusqBin{    //    Busqueda Binaria en un array de items
    int posic;    //  Posicion del ultimo encuentro
    int clav;       //  Clave buscada
    ArrItems arrItems;  //  El array donde buscaremos
    public  BusqBin(int cant, char tipo){       //    Un constructor
        arrItems = new ArrItems(cant,tipo); // que invoca otro
        posic = arrItems.talle+1;   //  posici¢n fuera del array
        clav  = 0;
    }

    protected int leerClav(){
      clav = In.readInt();
      return clav;
    }

    protected boolean existe(int clave){; // Existe la clave parámetro ?
      boolean exist = false;
      int alt=arrItems.talle-1,baj=0;
      int indCent, valCent;
      while (baj <= alt){
            indCent = (baj + alt)/2;      // índice de elemento central
            valCent = arrItems.item[indCent].getCodigo(); // elemento central
            if (clave == valCent){       // encontrado valor;
                posic = indCent; exist = true; break;
            }
            else if (clave < valCent)
            alt = indCent - 1;   // ir a sublista inferior
            else  baj = indCent + 1; // ir a sublista superior
        }
        return exist;                    // elenento no encontrado
    };

    protected int cuantas(int clave){;    // Cuantas veces existe la clave ?
      int igual=0; boolean exist;
      if (exist= existe(clave)){ // Si la clave existe, puede estar repetida
            for(int i=posic;i<arrItems.talle;i++)//  para arriba
            if (clave==arrItems.item[i].getCodigo())
                    igual++;
            else break;
            for(int i=posic-1;i>0;i--)    //    tambien para abajo
```

```
                if (clave==arrItems.item[i].getCodigo())
                    igual++;
                else break;
          }
        return igual;
    }

    protected void demoBusqBin(){
      int cant;
        System.out.println("Que clave? (999: fin) ");
      while (leerClav() != 999){
            if((cant = cuantas(clav)) > 0)
            System.out.println("Código " + clav + ", existe " + cant+" veces");
            else
            System.out.println("Codigo " + clav + " inexistente");
        }
      System.out.println("Demo finished!!!\n");
    }
}

package busqbin;
class  Main{
    public static void main(String args[]){
      // Generamos un array de 100 objetos item ascenden
      BusqBin busq = new BusqBin(100,'A');
      busq.demoBusqBin();
    }
}
```

```
run:
 Primeros 10 de 100
 elementos Item
0 - 3.7149723
1 - 64.90963
2 - 14.63609
3 - 36.667366
4 - 47.95137
5 - 14.102343
6 - 84.096565
7 - 88.0534
8 - 25.201313
9 - 80.84969
Que clave? (999: fin)
10
Código 10, existe 1 veces
Código 20, existe 1 veces
20
35
Código 35, existe 1 veces
999
Demo finished!!!
```

Es muy distinto el desempeño, la eficiencia de un método u otro, para poder evaluarlos necesitamos tener

Nociones de Complejidad Computacional

Operaciones Primitivas

Hacemos el análisis en forma directa en el programa de alto nivelo en el pseudocódigo. Definimos un conjunto de **operaciones primitivas** de alto nivel, en su mayor parte independientes del lenguaje de programación que se use y que se pueden identificar también en el pseudocódigo. Entre las operaciones primitivas se encuentran las siguientes:

- Asignación de un valor a una variable·
- Llamada de un método.
- Ejecución de una operación aritmética (por ejemplo, sumar dos números).
- Comparar dos números.
- Utilizar índices en un arreglo.
- Seguir una referencia de objeto.
- Retorno desde un método.

Una operación primitiva corresponde a una instrucción en bajo nivel, cuyo tiempo de ejecución depende del ambiente del hardware y software, y es constante. En lugar de

tratar de determinar el tiempo específico de ejecución de cada operación primitiva, tan sólo se contará **cuántas operaciones primitivas se ejecutan** y esta cantidad t se usará como estimado, en alto nivel, del tiempo de ejecución del algoritmo. En este método, la hipótesis implícita es que los tiempos de ejecución de distintas operaciones primitivas son bastante parecidos entre sí. Así, la cantidad t de operaciones primitivas que ejecuta un algoritmo será proporcional al tiempo real de ejecución de ese algoritmo.

Conteo de operaciones primitivas

A continuación mostramos cómo contar la cantidad de operaciones primitivas que ejecuta un algoritmo, usando como ejemplo el algoritmo arrayMax, cuya implementación en pseudocódigo y en Java se muestra en los fragmentos de programa 3.1 y 3.2, respectivamente, de la página 101. El siguiente análisis se puede hacer viendo el pseudocódigo o bien su implementación en Java.

Como ejemplo concreto analizaremos el método arrayMax, cuya misión es retornar el valor máximo del array A.

```
public class ArrayMaxProgram{
      // Determina el elemento máximo en un arreglo A de n enteros.
      static int arrayMax(int[] A, int n){
            int currentMax = A[O];    // se ejecuta una vez
            for (int i=l;i < n;i++)   // se ejecuta una vez, n veces, n-1 veces.
            if (currentMax < A[i])    // se ejecuta n-1 veces
                  current Max = A[i]; // se ejecuta cuando mucho n-l veces
            return current Max;        // se ejecuta una vez
      }
      // mas código …
}
```

- Inicializar la variable **currentMax a A[O]** corresponde a dos primitivas
  1) poner índices en un arreglo — Aporte
  2) asignar un valor a una variable) — Unidades
  (sólo se ejecuta una vez al principio del algoritmo. **2**
- Inicio del for, inicializa a 1 el contador i. **1**

Ciclo

- Condición de permanencia ciclo, i < n. **1, n veces.**
- Cuerpo del ciclo **se ejecuta n – 1 veces, en** cada iteración:
  A[i] se compara con currentMax. **2**
  se asigna A[currentMax] a currentMax. **2 (posibl.)**
  se incrementa el contador i. **2**

Fin ciclo

- Retomar el valor de la variable currentMax. **1**

Por consiguiente, en cada iteración del ciclo se ejecutan ya sea cuatro o seis operaciones primitivas, dependiendo de si A[i]<= currentMax, o de si A[i] > currentMax. Por consiguiente, el cuerpo del bucle contribuye al conteo con entre 4(n – 1) y 6(n – 1) unidades.
Resumiendo, la cantidad de operaciones primitivas t(n) ejecutadas por el algoritmo arrayMax es, cuando menos

2 + 1 + n + 4(n – 1) + 1 = **5n**

y cuando mucho

2 + 1 + n + 6(n – 1) + 1 = **7n – 2**

El caso óptimo (t(n) = 5n) se presenta cuando A[0] es el elemento máximo, por lo que nunca se reasigna la variable currentMax. **El caso peor** (t(n) = 7n – 2) se presenta cuando los elementos se encuentran en orden creciente, así que la variable currentMax se reasigna en cada iteración para el ciclo.

O sea que tenemos el mejor óptimo, el peor, el promedio, habrá que aplicar teoría de probabilidades para ver con que frecuencia ocurre cada uno. Simplifiquemos, seamos pesimistas y digamos que siempre ocurre el peor caso. O sea, en todas las comparaciones que hagamos, seró siempre el peor caso.

Se dirá que el algoritmo arrayMax ejecuta t(n) = 7n – 2 operaciones primitivas en el peor de los casos, para indicar que la cantidad máxima de operaciones primitivas que ejecuta el algoritmo, calculada con todas las entradas de tamaño n, es 7n – 2.

Notación asintótica

Es notorio que se ha entrado mucho en el detalle de la evaluación del tiempo de ejecución de un algoritmo tan sencillo como el arrayMax. Este análisis hace que surjan varias preguntas:

· ¿Realmente se necesita este grado de detalle?
· ¿Qué tan importante es calcular la cantidad exacta de operaciones primitivas que ejecuta un algoritmo?
· ¿Con cuánto cuidado se debe definir el conjunto de operaciones primitivas?

Por ejemplo, ¿cuántas operaciones primitivas se usan en la instrucción y = a*x + b? (Se podría decir que se ejecutan dos operaciones aritméticas y una asignación, pero se está omitiendo la asignación "oculta" adicional del resultado de a*x a una variable temporal, antes de ejecutar la suma.)

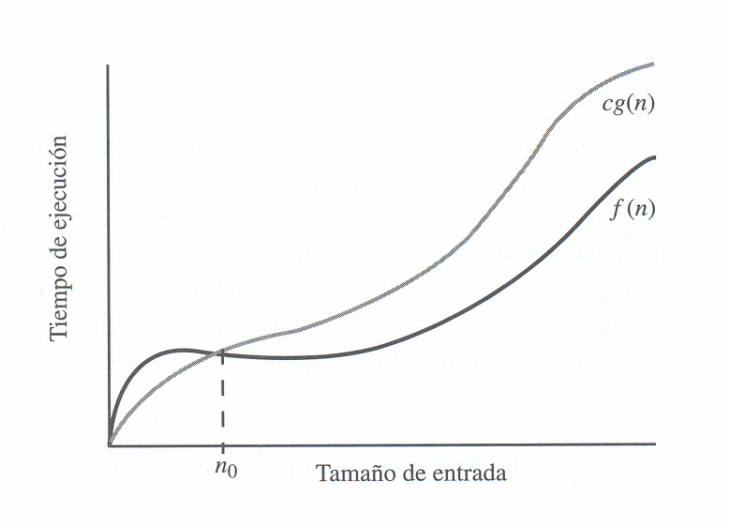
En general, cada paso de una descripción de pseudocódigo, y cada paso de programación en una implementación en alto nivel, corresponden a una pequeña cantidad de operaciones primitivas, que no dependen del tamaño de la entrada. Así, se puede hacer un análisis simplificado que estime la cantidad de operaciones primitivas ejecutadas hasta llegar a un factor constante, contando los pasos de las declaraciones en el pseudocódigo, o las instrucciones de programación que se usen en el lenguaje de alto nivel. Regresando al algoritmo arrayMax, el análisis simplificado indica que se ejecutan entre 5n y 7n – 2 pasos, con una entrada de tamaño n.

Simplificación ulterior del análisis

En un análisis de algoritmos es útil enfocarse en la tasa de crecimiento del tiempo de ejecución, en función del tamaño n de la entrada, adoptando un método de "panorámica", y no empantanarse con pequeños detalles. Con frecuencia basta sólo conocer que el tiempo de ejecución de un algoritmo como el arrayMax crece proporcionalmente a n, y que **el tiempo real de ejecución es n multiplicada por un pequeño factor constante** que depende de una entrada específica.
Se formaliza este método de analizar estructuras de datos y algoritmos usando una notación matemática para funciones, que no tiene en cuenta a los factores constantes. Es decir, se caracteriza el tiempo de ejecución y los requisitos de memoria de un algoritmo usando funciones que representan enteros a números reales, de tal forma que se enfoca la atención a los aspectos "panorámicos" primarios en una función de tiempo de ejecución, o de requisitos de espacio.

La notación "Oh"

Sean f(n) y g(n) funciones que representan enteros no negativos a números reales. Se dice que f(n) es O(g(n)) si hay una constante real c > 0 y un entero constante no >= 1 tal que f(n) <= cg(n) para todo entero n >= n0. Esta definición se llama notación "O", porque a veces se pronuncia “f(n) es Oh de g(n)". También se puede decir que “fl(n) es orden g(n)".
Esta definición se ilustra en la figura siguiente



Ejemplo de la notación Oh.
La función f(n) es O(g(n) para f(n) <= c.g(n) cuando n >= n0.

La notación Oh permite decir que una función de n es **"menor o igual a"** otra función, según la desigualdad "<=" en la definición, hasta un factor constante (por la constante c en la definición) y en el sentido asintótico al aumentar n hacia el infinito (por la afirmación "n >= no" en la definición).

La notación Oh se usa mucho para caracterizar tiempos de ejecución y límites de espacio, en función de algún parámetro n, que varía de un problema a otro, pero que por lo general se define como noción intuitiva del "tamaño" del problema.
Por ejemplo, si interesa determinar el mayor elemento en un arreglo de enteros (véase máxArreglo de los fragmentos de programa 3.1 y 3.2), lo más natural sería que n represente la cantidad de elementos del arreglo. La notación Oh permite pasar por alto los factores constantes y los términos de orden inferior, para enfocarse en los principales componentes de una función que afecten su crecimiento.

Proposición: El tiempo de ejecución del algoritmo arrayMax para calcular el elemento máximo en un arreglo de n enteros es O(n).

justificación: Como se concluyó anteriormente, la cantidad de operaciones primitivas que ejecuta el algoritmo arrayMax es, cuando mucho, 7n – 2. Por consiguiente, hay una constante positiva a que depende de la unidad de tiempo y del ambiente software y hardware donde se implemente, compile y ejecute el algoritmo arrayMax, tal que el tiempo de ejecución del mismo para un tamaño de entrada n es cuando mucho a(7n – 2). Se aplicará la definición Oh con c = 7a y n0 = 1, para llegar a la conclusión de que el tiempo de ejecución del algoritmo arrayMax es O(n).

Comparación de la búsqueda binaria y secuencial. Definición del **orden de complejidad** de cada una

La diferencia en tiempo entre los algoritmos de búsqueda secuencial y binaria se va haciendo espectacular a medida que crece el tamaño del array. Tengamos presente que:

En el caso de la búsqueda secuencial en el peor de los casos coincidirá el número de elementos examinados con el número de elementos del array. O sea que en este caso el tiempo de búsqueda es **orden de complejidad O(n)**

En el caso de la búsqueda binaria, realizamos comparaciones con el elemento medio del array y subsarrays resultantes de la partición expuesta en el punto anterior. El array es partido en dos antes de cada comparación. Muy rápidamente llegamos al

elemento buscado o a la conclusión de su inexistencia. El orden de complejidad correspondiente al número máximo de comparaciones (peor caso) resultante de esta mecánica de sucesivas particiones es **$O(\log_2 (n-1))$.**

Números de comparaciones considerando el peor caso

Tamaño array	Búsqueda binaria	Búsqueda secuencial
1	1	1
10	4	10
1.000	10	1.000
5.000	13	5.000
100.000	17	100.000
1.000.000	20	1.000.000

Ordenamiento – Introducción

Muchas actividades humanas requieren que a diferentes colecciones de elementos utilizados se pongan en un orden específico. Las oficinas de correo y las empresas de mensajería ordenan el correo y los paquetes por códigos postales con el objeto de conseguir una entrega eficiente; los anuarios o listines telefónicos se ordenan por orden alfabético de apellidos con el fin de encontrar fácilmente el número de teléfono deseado. Los estudiantes de una clase de la universidad se ordenan por sus apellidos o por los números de matrícula. Una de las tareas que realizan más frecuentemente las computadoras en el procesamiento de datos es la *ordenación*. (Después de todo, no olvidemos que a las computadoras se les llama también ordenadores).
El estudio de diferentes métodos de ordenación es una tarea muy interesante desde un punto de vista teórico y práctico. A continuación estudiaremos varios algoritmos de ordenamiento.

Algoritmos de ordenamiento básico y mejorado.

Es usual dividir los algoritmos de ordenamiento en dos grupos. Los básicos, de codificación relativamente simple, y los mejorados, de muy alta eficiencia en su tarea de ordenar.
De los básicos estudiaremos el Ordenamiento por Burbuja, el mas simple (e ineficiente) que existe, y el Ordenamiento por Sacudida, algo mas complejo, pero bastante mejor.
Los llamados Mejorados, no son algo mejores que los básicos, son 10 o mas veces mas rápidos, de ellos veremos el Algoritmo de Peinado, de invención bastante reciente, muy simple y eficiente. Anteriores a él hay varios, citemos al Shell, HeapSort y QuicSort. El Quick es el mas rápido de todos, (solo levemente que el de Peinado). Todos ellos son más complejos que el de Peinado.

Ordenamiento por Método Burbuja.

El método de ***Ordenación por Burbuja***, muy popular y conocido por los estudiantes de programación, es ***el menos eficiente*** de todos.
La técnica utilizada se denomina ordenación por hundimiento debido a que los valores mayores burbujean (suben hacia la cima) del array. Se comparan elementos de a pares, y si el array tiene n elementos, el método realiza n-1 pasadas. Como en cada pasada el mayor "burbujea" hasta la cima, cada pasada sucesiva es de un elemento menos. El método no tiene capacidad de detectar si el array ya esta ordenado. Esto significa que si el array hace n-1 pasadas siempre, aunque el array esté ordenado de partida.

Para trazar como va quedando el array tras sucesivas pasadas, vamos a incluir un método mostArr() que muestre en una línea los 10 primeros elementos del array. Y a este método lo usaremos opcionalmente dentro del ordBurb(), para ir trazando el avance del ordenamiento.

```
package ordburb;
class OrdBurb {                 //  Ordenamiento de un array de items
      protected ArrItems arrItems;// mediante el método Burbuja
      protected int pas;          //  pasada
      public OrdBurb(int cant, char tipo){      //   Un constructor
            arrItems =  new ArrItems(cant,tipo);
            System.out.println("\nPasadas de ordenamiento");
      }
      public String toString(){ // Mostrando los valores del array
            String aux  = "Pas "+pas+" ";
            int cuant=(arrItems.talle < 10 ? arrItems.talle : 10);
            // Lo que sea menor
            for (int i=0;i<cuant;i++)
                  aux += arrItems.item[i].getCodigo()+", ";
            return aux;
      }
      void ordenar(boolean trace){; // El método de ordenamiento
            int j;
            for (pas = 1; pas < arrItems.talle; pas++){
                  for (j = arrItems.talle-1; j>=pas; j--) // realizando la pasada
                        if (arrItems.item[j].esMayor(arrItems.item[j-1]))
                        // si invocante es mayor que parámetro
                              arrItems.item[j].intercambio(arrItems.item[j-1]);
            if (trace)
                        System.out.println(this);  // Mostramos el array
      }
      }
}

package ordburb;
class Main{
      public static void main(String args[]){
            System.out.println("\nOrdenamiento metodo Burbuja");
            OrdBurb burb   = new OrdBurb(10,'R'); // Generamos un array de 20
objetos item random
            burb.ordenar(true);     // y lo ordenamos, trazando
      }
}
```

```
Pasadas de ordenamiento
Pas 1 9, 8, 7, 7, 5, 2, 2, 1, 0, 1,
Pas 2 9, 8, 7, 7, 5, 2, 2, 1, 1, 0,
Pas 3 9, 8, 7, 7, 5, 2, 2, 1, 1, 0,
Pas 4 9, 8, 7, 7, 5, 2, 2, 1, 1, 0,
Pas 5 9, 8, 7, 7, 5, 2, 2, 1, 1, 0,
Pas 6 9, 8, 7, 7, 5, 2, 2, 1, 1, 0,
Pas 7 9, 8, 7, 7, 5, 2, 2, 1, 1, 0,
Pas 8 9, 8, 7, 7, 5, 2, 2, 1, 1, 0,
Pas 9 9, 8, 7, 7, 5, 2, 2, 1, 1, 0,
BUILD SUCCESSFUL (total time: 1
second)
```

```
Pasadas de ordenamiento
Pas 1 8, 4, 3, 1, 3, 7, 5, 7, 5, 2,
Pas 2 8, 7, 4, 3, 1, 3, 7, 5, 5, 2,
Pas 3 8, 7, 7, 4, 3, 1, 3, 5, 5, 2,
Pas 4 8, 7, 7, 5, 4, 3, 1, 3, 5, 2,
Pas 5 8, 7, 7, 5, 5, 4, 3, 1, 3, 2,
Pas 6 8, 7, 7, 5, 5, 4, 3, 3, 1, 2,
Pas 7 8, 7, 7, 5, 5, 4, 3, 3, 2, 1,
Pas 8 8, 7, 7, 5, 5, 4, 3, 3, 2, 1,
Pas 9 8, 7, 7, 5, 5, 4, 3, 3, 2, 1,
BUILD SUCCESSFUL (total time: 1
second)
```

9 pasadas, 7 innecesarias 9 pasadas, 2 innecesarias

Ordenamiento por Método Sacudida.

El Método **"sacudida"** incorpora un par de ventajas respecto Burbuja:

- Alterna pasadas de izquierda a derecha y viceversa, con lo que consigue que tanto menores fluyan a derecha como mayores a izquierda con igual velocidad
- Lleva un registro de donde fue la última inversión realizada en la pasada anterior, esto le posibilita no recorrer tramos de la pasada innecesariamente.

Este método es una variante mejorada del Burbuja, relacionamos con “es un”

```
package ordsac;
public class OrdSac extends OrdBurb{ //  Ordenamiento mediante el método Sacudida
    public OrdSac(int cant, char tipo){      //Un constructor
      super(cant,tipo);
    }
    void ordenar(boolean trace){
      int j,k = arrItems.talle-1,iz = 1,de = arrItems.talle-1; Item aux;
      do{                                  // Ciclo de control de pasadas
        for (j = de; j>=iz; j--) // Pasada descendente
            if (arrItems.item[j].esMayor(arrItems.item[j-1])){
                arrItems.item[j].intercambio(arrItems.item[j-1]);
                k = j;      // Guardamos el lugar del último intercambio
            }
        iz = k+1; pas++;
        if (trace)
            System.out.println(this);  // Mostramos el array despues de la pasada
        for (j = iz; j<=de; j++)   // Pasada ascendiente
            if (arrItems.item[j].esMayor(arrItems.item[j-1])){
                arrItems.item[j].intercambio(arrItems.item[j-1]);
                k = j;                // Guardamos el lugar del último intercambio
            }
        de = k-1; pas++;
        if (trace)
            System.out.println(this);  // Mostramos el array despues de la pasada
      }while (iz<=de);
    }
}
package ordsac;
class Main{
      public static void main(String args[]){
            System.out.println("\nOrdenamiento metodo Sacudida");
            OrdBurb sac  = new OrdBurb(10,'R');
            // Generamos un array de 10 objetos item random
            sac.ordenar(true);      // y lo ordenamos, trazando
      }
}
```

```
Ordenamiento metodo Sacudida
 Primeros 10 de 10 elementos Item
2 - 4.2255816
0 - 2.8287904 …
Pasadas de ordenamiento
Pas 1 9, 2, 0, 1, 3, 1, 0, 1, 7, 8,
Pas 2 9, 8, 2, 0, 1, 3, 1, 0, 1, 7,
Pas 3 9, 8, 7, 2, 0, 1, 3, 1, 0, 1,
Pas 4 9, 8, 7, 3, 2, 0, 1, 1, 1, 0,
Pas 5 9, 8, 7, 3, 2, 1, 0, 1, 1, 0,
Pas 6 9, 8, 7, 3, 2, 1, 1, 0, 1, 0,
Pas 7 9, 8, 7, 3, 2, 1, 1, 1, 0, 0,
Pas 8 9, 8, 7, 3, 2, 1, 1, 1, 0, 0,
Pas 9 9, 8, 7, 3, 2, 1, 1, 1, 0, 0,
BUILD SUCCESSFUL (total time:
```

```
Ordenamiento metodo Sacudida
 Primeros 10 de 10 elementos Item
7 - 5.4604025
3 - 8.100628 …
Pasadas de ordenamiento
Pas 1 9, 7, 3, 2, 7, 5, 4, 6, 2, 2,
Pas 2 9, 7, 7, 3, 2, 6, 5, 4, 2, 2,
Pas 3 9, 7, 7, 6, 3, 2, 5, 4, 2, 2,
Pas 4 9, 7, 7, 6, 5, 3, 2, 4, 2, 2,
Pas 5 9, 7, 7, 6, 5, 4, 3, 2, 2, 2,
Pas 6 9, 7, 7, 6, 5, 4, 3, 2, 2, 2,
Pas 7 9, 7, 7, 6, 5, 4, 3, 2, 2, 2,
Pas 8 9, 7, 7, 6, 5, 4, 3, 2, 2, 2,
Pas 9 9, 7, 7, 6, 5, 4, 3, 2, 2, 2,
BUILD SUCCESSFUL (total time:
```

9 pasadas, 2 innecesarias 9 pasadas, 4 innecesarias

Ordenamiento por Método "Peinado"

Es uno de los métodos mejorados. Es muy veloz, pero no consigue destronar al Quick Sort ó rápido. Su variante principal respecto a los métodos ya vistos es el concepto de paso: lo usa para comparar elementos no contiguos y lo va ajustando pasada tras pasada. Este metodo es totalmente nuevo, no es variante de anteriores.

```
package ordpein;
public class OrdPein{   //    Ordenamiento de un array de items
    private ArrItems arrItems;// mediante el método Burbuja
    private int pas 0;          //  pasada
    public OrdPein(int cant, char tipo){      //     Un constructor
        arrItems =  new ArrItems(cant,tipo);
        System.out.println("\nPasadas de ordenamiento");
    }
    void ordenar(boolean trace){    //programa de ordenamiento por peinado
      int i,paso = arrItems.talle; boolean cambio;
        do {paso = (int)((float) paso/1.3);
            paso = paso>1 ? paso : 1;
            cambio = false;
            for (i = 0; i<arrItems.talle-paso; i++)
                if (arrItems.item[i].esMayor(arrItems.item[i+paso])){
                    arrItems.item[i].intercambio(arrItems.item[i+paso]);
                    cambio = true;
            }
            pas++;
            if (trace) System.out.println(this);
      } while (cambio || paso > 1);
    } //    void ordenar
    public String toString(){ // Mostrando los valores del array
      String aux  = "Pas "+pas+" ";
      int cuant=(arrItems.talle < 10 ? arrItems.talle : 10);  // Lo que sea menor
      for (int i=0;i<cuant;i++)
            aux += arrItems.item[i].getCodigo()+", ";
      return aux;
    }
}           //    public class OrdPein
package ordpein;
class Main{
      public static void main(String args[]){
            System.out.println("\nOrdenamiento metodo Peinado");
            OrdPein pein  = new OrdPein(10,'R'); // 10 objetos item random
            pein.ordenar(true);     // y lo ordenamos, trazando
      }
};
```

```
Ordenamiento metodo Peinado
6 - 5.955661
2 - 8.241952
3 - 3.914549 …
Pasadas de ordenamiento
Pas 1 0, 2, 3, 2, 9, 0, 1, 6, 8,
3,
Pas 2 0, 1, 3, 2, 3, 0, 2, 6, 8,
9,
Pas 3 0, 1, 0, 2, 3, 3, 2, 6, 8,
9,
Pas 4 0, 1, 0, 2, 2, 3, 3, 6, 8,
9,
Pas 5 0, 0, 1, 2, 2, 3, 3, 6, 8,
```

6 pasadas, todas necesarias

```
Ordenamiento metodo Peinado
6 - 5.6722007
0 - 0.032508835
2 - 3.1329563 …

Pasadas de ordenamiento
Pas 1 2, 0, 2, 7, 5, 3, 8, 6, 1, 4,
Pas 2 2, 0, 2, 1, 4, 3, 8, 6, 7, 5,
Pas 3 1, 0, 2, 2, 4, 3, 5, 6, 7, 8,
Pas 4 1, 0, 2, 2, 4, 3, 5, 6, 7, 8,
Pas 5 0, 1, 2, 2, 3, 4, 5, 6, 7, 8,
Pas 6 0, 1, 2, 2, 3, 4, 5, 6, 7, 8,
BUILD SUCCESSFUL (total time: seconds)
```

6 pasadas, todas necesarias

Como sería el **Ordenamiento por Método "Peinado"**, pero trabajando sobre una colección Arraylist. Usando lo que ya vimos, bastante simple:

```
package ordpeincol;
import java.util.ArrayList;
public class OrdPeinCol{      //    Ordenamiento de un array de items
    private ArrayList<Item> colItems;
    private Item item;
    private int pas;           //  pasada
    public OrdPeinCol(int cant, char tipo){      //  Un constructor
      int   auxCod = 0; // Una variable auxiliar
        colItems = new ArrayList<Item>();
        for(int i=0; i<cant; i++){    // la llenaremos de Item's, dependiendo
            switch(tipo){            // del tipo de llenado requerido
            case 'A':{           // los haremos en secuencia ascendente
                    auxCod = i;
                  break;
            }
            case 'D':{       // o descendente ...
                    auxCod = cant - i;
                    break;
            }
            case 'R':{       // o bien randomicamentente (Al azar)
                    auxCod = (int)(cant*Math.random());
            }
            }
            item = new Item(auxCod,(float)(cant*Math.random()));
            colItems.add(item);
      }
    }

    void ordenar(boolean trace){    //programa de ordenamiento por peinado
        if(trace){
            System.out.println("Colección creada");
            System.out.print(colItems);
        }
      int i,paso = colItems.size();
        Item itAct, itSig;
      boolean cambio;
        do {
            paso = (int)((float) paso/1.3);
            paso = paso>1 ? paso : 1;
            cambio = false;
            if(trace) System.out.println("paso vale "+paso);
            for (i = 0; i<colItems.size() - paso; i++){
                itAct = (Item)colItems.get(i);
                itSig = (Item)colItems.get(i+paso);
                if (itAct.esMayor(itSig)){
                    colItems.set(i,itSig);
                    colItems.set(i+paso,itAct);
                    cambio = true;
            }
            }
            pas++;
      } while (cambio || paso > 1);
        if(trace){
            System.out.println("Colección ordenada");
            System.out.print(colItems);
        }
    } //    void ordenar
```

```
 }          //    public class OrdPeinCol
```

```
package ordpeincol;
class Main{
      public static void main(String args[]){
            System.out.println("\nOrdenamiento metodo Peinado");
            OrdPeinCol pein  = new OrdPeinCol(10,'R'); // 10 objetos item random
            pein.ordenar(true);     // y lo ordenamos, trazando
      }
};
```

```
run:

Ordenamiento metodo Peinado
Colección creada
[9 - 9.709201, 5 - 3.8829691, 2 - 9.417128, 5 - 1.6681215, 8 - 9.54292, 7
- 7.0527325, 4 - 4.7038474, 6 - 0.8901949, 6 - 5.2643456, 5 - 8.892109]
paso vale 7
paso vale 5
paso vale 3
paso vale 2
paso vale 1
paso vale 1
Colección ordenada
[2 - 9.417128, 4 - 4.7038474, 5 - 1.6681215, 5 - 3.8829691, 5 - 8.892109,
6 - 5.2643456, 6 - 0.8901949, 7 - 7.0527325, 8 - 9.54292, 9 - 9.709201]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Determinación del Orden de Complejidad. Análisis de tiempos

Vamos a medir determinar el orden de complejidad de estos cuatro algoritmos. Nos interesa comparar desempeños para un numero creciente de ítems. Que tiempos se necesitan para ordenar 1000, 10000, 10000, 100000, 200000 items?.

```
package analtiempos;
class Orden{
    void ordenar(boolean ord){
        //  Solo para que el compilador encuentre jerarquía hereditaria y compile
    }   //  Las clases OrdBurb, OrdPein, OrdPeinCol extienden Orden.
}

package analtiempos;
import java.util.*;
public class Tiempos{
    Calendar cale;
    int cuantos;
    Orden orden[];
    long tpoIni, tpoFin, tiempo;
    String metodo[] = {"OrdBurb    ","OrdSsc     ","OrdPein    ","OrdPeinCol "};
    String resultados = "Metodo, Cuantos, Inicio, Fin, Tiempo\n";
    public Tiempos(){
        System.out.println("Cuantos items ordenamos");
        cuantos = In.readInt();
        orden = new Orden[4];
        orden[0] = new OrdBurb(cuantos,'R');
        orden[1] = new OrdSac(cuantos,'R');
        orden[2] = new OrdPein(cuantos,'R');
```

```
        orden[3] = new OrdPeinCol(cuantos,'R');
    }

    public void proceso(){
        for(int i = 0; i<orden.length; i++){
            cale = new GregorianCalendar();
            tpoIni = cale.getTimeInMillis();
            orden[i].ordenar(false);
            cale = new GregorianCalendar();
            tpoFin = cale.getTimeInMillis();
            tiempo = tpoFin - tpoIni;
            resultados+=metodo[i]
                    + cuantos+", " + tpoIni +", "
                    + tpoFin +", " + tiempo +"\n";
        }
        System.out.println(resultados);
        System.out.println("Demo terminado!!!");
    }
}
public class Main {
    public static void main(String[] args) {
        Tiempos tiempos = new Tiempos();
        tiempos.proceso();
    }
}
```

```
Cuantos items ordenamos
1000
Pasadas de ordenamiento
Metodo,   Cuantos, Inicio,        Fin,       Tiempo
OrdBurb    1000, 1201866817562, 1201866817593, 31
OrdSsc     1000, 1201866817593, 1201866817625, 32
OrdPein    1000, 1201866817625, 1201866817625, 0
OrdPeinCol 1000, 1201866817625, 1201866817640, 15
Demo terminado!!!
```

```
Cuantos items ordenamos
10000
 Pasadas de ordenamiento
Metodo,   Cuantos,  Inicio,        Fin,       Tiempo
OrdBurb    10000, 1201867626875, 1201867629296, 2421
OrdSsc     10000, 1201867629296, 1201867631203, 1907
OrdPein    10000, 1201867631203, 1201867631218, 15
OrdPeinCol 10000, 1201867631218, 1201867631250, 32
Demo terminado!!!
```

```
Cuantos items ordenamos
25000
Pasadas de ordenamiento
Metodo,   Cuantos, Inicio,        Fin,         Tiempo
OrdBurb    25000, 1201868448625, 1201868472234, 23609
OrdSsc     25000, 1201868472234, 1201868486937, 14703
OrdPein    25000, 1201868486937, 1201868486984, 47
OrdPeinCol 25000, 1201868486984, 1201868487109, 125
Demo terminado!!!
```

```
Cuantos items ordenamos
50000
Pasadas de ordenamiento
Metodo,   Cuantos, Inicio,        Fin,          Tiempo
OrdBurb    50000, 1201867930765, 1201868033359, 102594
OrdSsc     50000, 1201868033359, 1201868097203, 63844
OrdPein    50000, 1201868097203, 1201868097312, 109
OrdPeinCol 50000, 1201868097312, 1201868097703, 391
Demo terminado!!!
```

Conclusiones:

- Los tiempos de los metodos básicos **crecen exponencialmente** según la cantidad de elementos a ordenar. (Terrible)
- El método OrdPeinCol, que usa ArrayList, demora un poco mas del doble que el OrdPein, que usa array. (Ambos son muy rápidos, esto no es significativo)
- Los tiempos de ambos **crecen linealmente** seg/cant. de elementos. (Excelente)
- Para 50000 elementos, ordPein es 586 veces mas rápido que OrdSac y 941 veces que OrdBurb

Arrays multidimensionales

Los arrays vistos anteriormente se conocen como arrays unidimensionales (una sola dimensión) y sus elementos se referencian mediante un único subíndice. Los arrays multidimensionales tienen más de una dimensión y, en consecuencia, de un índice. Son usuales arrays de dos dimensiones. Se les llama también como ***tablas*** o matrices. Es posible crear arrays de tantas dimensiones como se necesite, el límite puede ser nuestra capacidad de interpretación.

En memoria, un array bidimensional se almacena en un espacio lineal, fila tras fila. Si fuera tridimensional nos lo imaginamos constituido por planos, filas, columnas. Su almacenamiento es siempre lineal, en primer lugar tendríamos los elementos del primer plano, fila tras fila, luego el segundo plano y así sucesivamente.

La sintaxis de un array bi dimensional es:

<Tipo de dato> <Nombre Array> [<Numero de filas>] [<Numero de columnas>]

Algunos ejemplos de declaración de matrices.

```
char pantalla [25 ] [80];     // Para trabajr una pantalla en modo texto
int matCurs [10 ] [12];       // Cantidad de alumnos por materia/curso
```

Inicialización de arrays multidimensionales

Los arrays multidimensionales se pueden inicializar, al igual que los una dimensión, cuando se declaran. Esto lo hacemos asignándole a la matriz una lista de constantes separadas por comas y encerradas entre llaves, ejemplos.

```
public class PruMat01{
      protected int[] [] mat01 = {{51, 52, 53}, {54, 55, 56}};
      // Declaramos e inicializamos el objeto array bidimensional

      public PruMat01(){      // El constructor
            System.out.print(this);
      }

      public String toString(){ // Visualización del objeto
            String aux = "Matriz mat01 \n";
            int f,c;
            for(f=0;f<mat01.length;f++){  // Recorremos filas
                  for(c=0;c<mat01[0].length;c++)      // Recorremos columnas
                        aux+=mat01[f][c]+", ";
                  aux+="\n";                    // A la fila siguiente
            }
            return aux;
      }

      public static void main(String args[]){
            PruMat01 mat = new PruMat01();
      }
}
```

```
Matriz mat01
51, 52, 53,
54, 55, 56,
Process Exit...
```

Java (Al igual que otros lenguajes, C++ por ejemplo), considera un array

bidimensional como un array de arrays monodimensionales. Esto lo podemos aprovechar. Si usamos el atributo length de arrays, cuando expresamos mat01.length (ejemplo anterior) nos estamos refiriendo a la cantidad de filas que el objeto mat01 posee. Cuando expresamos mat01[0].length nos referimos a la cantidad de elementos que contiene la fila 0. Y este concepto puede generalizarse a arrays de n dimensiones.

Acceso a los elementos de arrays bidimensionales

Se puede acceder a los elementos de arrays bidimensionales de igual forma que a los elementos de un array unidimensional. La diferencia reside en que en los elementos bidimensionales deben especificarse los índices de la fila y columna.
El formato general para asignación directa de valores a los elementos:

```
// Asignación de valores (Normalmente usaremos métodos setNombre())
<nombre matriz >[índice fila] [índice columna] =valor elemento;


// Obtención de valores (Normalmente usaremos métodos getNombre())
<variable> = <nombre array> [índice fila]  [índice columna];
```

Acceso a elementos mediante bucles

Se puede recorrer elementos de arrays bidimensionales mediante bucles anidados. Ya lo vimos en el método toString del ejemplo anterior. Para ejemplificar un poco mas, extendamos la clase anterior, incluyendo un metodo seteos() que modifica los valores originales del objeto de PruMat01 y luego lo muestra.

```
public class PruMat02 extends PruMat01{
     public PruMat02(){ // El constructor
      super();
     }

     public void seteos(){ // Modificando valores del objeto
      int f,c,cont=0;
      for(f=0;f<mat01.length;f++){              // Recorremos filas
            for(c=0;c<mat01[0].length;c++)      // Recorremos columnas
                  mat01[f][c]=++cont;
      }
     }

     public static void main(String args[]){
      PruMat02 mat = new PruMat02();
      mat.seteos();
      System.out.print(mat);
     }
}
```

```
Matriz mat01
51, 52, 53,
54, 55, 56,
Matriz mat01
1, 2, 3,
4, 5, 6,
Process Exit...
```

Hasta ahora estamos trabajando “desde adentro”. Las clases estan dentro del mismo paquete, en una estructura hereditaria. Inclusive el main() es un método de la propia clase. Vamos a variar esto un poco. Incorporaremos los metodos setValor(), lectValor() y getValor() para poder asignar y obtener valores desde fuera. Para ello extendemos PruMat02() en PruMat03(). En la nueva clase no definimos ningún main(), que como es estático **no se hereda** desde PruMat02(). El main() lo pondremos en una clase independiente de esta estructura, la PruMat04.

```
import java.io.IOException;
public class PruMat03 extends PruMat02{
```

```
     public PruMat03(){ // El constructor
      super();
     }

     public void setValor(int f, int c, int val){
      mat01[f][c] = val;
     }

     public void lectValor() throws java.io.IOException{
      int f,c,val;
      System.out.print("A que fila pertenece el valor? ");
      f = System.in.read();
      System.out.print("A que col. pertenece el valor? ");
      c = System.in.read();
      System.out.print("Cual es el valor en cu
      mat01[f][c] = System.in.read();
     }

     public int getValor(int f, int c){
      return mat01[f][c];
     }
}
```

```
Matriz mat01
51, 52, 53,
54, 55, 56,

A que fila pertenece el valor?
0
A que col. pertenece el valor?
1
Cual es el valor en cuestion ?
33

aux = mat.getValor(0,1): 33

Matriz mat01
```

La clase **PruMat04** para usar objetos y métodos de **PruMat03**, a continuación

```
import PruMat03;
import java.io.IOException;
public class PruMat04{
     public static void main(String args[]) throws java.io.IOException{
      int aux;
      PruMat03 mat = new PruMat03();      // Instanciamos objeto
      mat.lectValor();              //  Leemos un valor
      mat.setValor(1,2,25);               //  asignamos otro
      aux = mat.getValor(0,1);            //  Obtenemos otro
      System.out.println("\naux = mat.getValor(0,1): "+aux);     //  lo imprimimos
      System.out.println(mat);            //  Imprimimos el objeto
     }
}
```

Si analizamos la salida, vemos:

- El objeto mat, impresos sus valores iniciales.
- lectValor() solicitando datos
- setValor(1,2,25), que inicializa el casillero [1][2] con el valor 25
- aux recibiendo el valor [0][1]
- System.out.println(…) mostrando aux;
- El objeto mat, impresos sus valores finales.

Una segunda ejecución:

```
Matriz mat01
51, 52, 53,
54, 55, 56,
A que fila pertenece el valor? 2
A que col. pertenece el valor? 2
Cual es el valor en cuestion ?
java.lang.ArrayIndexOutOfBoundsException
        at PruMat03.lectValor(PruMat03.java:20)
        at PruMat04.main(PruMat04.java:8)
Exception in thread "main"
Process Exit...
```

Hemos intentado leer un elemento fuera de la matriz. No existe fila 2, solo tenemos 0 y 1. Se ha producido una excepción del tipo remarcado, **ArrayIndexOutOfBoundsException;** podriamos capturarla y tratarla?. Lo que pretendemos es que:

- Informemos al operador el problema en cuestion: desborde de indices
- Pueda repetir el procedimiento para la lectura, **public void lectValor()**

Como todo el resto del comportamiento de la clase **PruMat03** me sirve, lo vamos a aprovechar. Definimos una clase PruMat05 que extiende Prumat03 y la probamos en Prumat06.
En el método **lectValor()** incorporamos un ciclo permanente while (true) para la lectura repetida. Usamps un bloque try, dentro de él leemos. Si ocurre la excepción de desborde de índice, la reportamos y la sentencia continue nos posiciona al inicio del ciclo nuevamente. Si no ocurre, break nos sacará del ciclo …

```
import java.io.IOException;
import In;
public class PruMat05 extends PruMat03{
   public PruMat05(){   // El constructor
      super();
  }
  public void lectValor() throws java.lang.ArrayIndexOutOfBoundsException{
     int f,c,val;
     while (true){
      try{
         System.out.print("\nA que fila pertenece el valor? ");
         f = In.readInt();
         System.out.print("A que col. pertenece el
         c = In.readInt();
         System.out.print("Cual es el valor en cue
         mat01[f][c] = In.readInt();
         break;   // Lectura exitosa, salimos del
           }catch(java.lang.ArrayIndexOutOfBoundsE
         System.out.println("\nCuidado, desborde d
         continue;      // Seguimos intentando la
           }
      }     //    while
   }  //    void lectValor()
}     //    class PruMat05
```

```
Matriz mat01
51, 52, 53,
54, 55, 56,

A que fila pertenece el valor? 2
A que col. pertenece el valor? 2
Cual es el valor en cuestion ?
Cuidado, desborde de indice...

A que fila pertenece el valor? 1
A que col. pertenece el valor? 2
Cual es el valor en cuestion ?
25
aux = mat.getValor(0,1): 52
Matriz mat01
51, 52, 25,
54, 55, 25,
Process Exit...
```

```
import PruMat05;
import java.io.IOException;
public class PruMat06{
      public static void main(String args[]) throws java.io.IOException{
            int aux;
            PruMat05 mat = new PruMat05();      // Instanciamos objeto
            mat.lectValor();              //  Leemos un valor
            mat.setValor(0,2,25);         //  asignamos otro
            aux = mat.getValor(0,1);      //  Obtenemos otro
            System.out.println("\naux = mat.getValor(0,1): "+aux);           //  lo
imprimimos
            System.out.println(mat);      //  Imprimimos el objeto
      }
}
```

Hemos visto que Java considera a un array de dos dimensiones com un array unidimensional cuyos elementos son arrays unidimensionales. Generalizando, podemos decir que Java considera un array de dimension n como un array unidimensional cuyos elementos son arrays de dimension n-1.

Antes ya hemos trabajado bastante con un array unidimensional de objetos Item. Podriamos trabajar una matriz bidimensional de objetos Item.

Matriz de objetos Item

Aprovechando lo que ya tenemos, podemos implementar de más de una forma nuestra matriz de objetos Item. Una primera, digamos clásica, bastante similar a como implementamos ArrItems, definiendo item como una referencia a un array bidimensional, y usando un constructor que instancia e inicializa todos sus elementos Item. El objeto Matriz tiene muchos Objetos Item

Que comportamiento le asignamos? No mucho.
protected int cantCodEnFila(int cod, int fila) // Cuantas veces ocurre el cod parámetro en la fila parámetro
protected double promValCodEnCol(int cod, int col){// retorna el promedio de los valores asociados al
// código parámetro en la columna parámetro.

El comportamiento que podemos imaginar es realmente muy extenso. Podemos pensar en operaciones involucrando dos o mas objetos Matriz: igualdad, conformables para producto, etc, etc. Demos a nuestros jefes de prácticos y alumnos oportunidad de lucirse.

```
import Item;
class MatItems{            // Matriz de objetos Items
      protected Item[][] item; //
      protected int filas, cols;//  Dimensiones de la matriz
      public MatItems(int fil, int col, char tipo) { // Constructor
            int f,c,auxCod = 0;        // Una variable auxiliar
            filas=fil;cols=col;                // inicializamos tamaño
            item = new Item[filas][cols]; // Generamos la matriz
            for(f=0;f<item.length;f++)         // Barremos filas
                  for(c=0;c<item[0].length;c++){ // y columnas
                        switch(tipo){   // segun tipo de llenado requerido
                              case 'A':{        //   los   haremos   en   secuencia ascendente
                                    auxCod = c;
                                    break;
                              }
                              case 'D':{        // o descendente ...
                                    auxCod = cols - c;
                                    break;
                              }
                              case 'R':{        //  o  bien  randomicamentente  (Al azar)
                                    auxCod = (int)(cols*Math.random());
                              }
                        }                              //  switch
                        item[f][c] = new Item();
                        item[f][c].setCodigo(auxCod);
                        item[f][c].setValor((float)(cols*Math.random()));
                  }     // for(c=0
      }     //    public MatItems

      public String toString(){
            int ctos = (cols < 10 ? cols : 10);
            String aux = " Primeros "+ctos+" elementos de la fila 0\n";
```

```
            for(int i=0;i<ctos;i++)
                  aux+=item[i][0].toString()+"\n";
            return aux;
      }

      protected int cantCodEnFila(int cod, int fila){
            // Cuantas veces ocurre el cod parámetro en la fila parámetro
            int cuantas = 0,i;
            for(i=0;i<item[fila].length;i++)
                  if(item[fila][i].getCodigo() == cod)cuantas++;
            return cuantas;
      }

      protected double promValCodEnCol(int cod, int col){
            //  retorna el promedio de los valores asociados
            // al código parámetro en la columna parámetro.
            int contCod = 0,i;
            double acuVal = 0.0, prom;
            for(i=0;i<item.length;i++)
              if (item[i][col].getCodigo() == cod){
                  contCod++; acuVal+=item[i][col].get
              }
            prom = (double)(acuVal/contCod);
            return prom;
      }
}


import MatItems;
class  pruMatItems{
      public static void main(String args[]){
            MatItems mtIt = new MatItems(20,20,'R');
            //  Generamos una matriz de 20 x 20 objet
            // codigos y valores al azar
            System.out.println(mtIt);
            System.out.println("En la fila 10 el codigo 10 ocurre "+
                                          mtIt.cantCodEnFila(10,10)+" veces");
            System.out.println("En la columna 10 el valor promedio\n"+
                                       "de los valores asociados al codigo 10\n"+
                                       "es de "+mtIt.promValCodEnCol(10,10));
      }
};
```

```
Primeros 10 elementos de la
fila 0
4 - 2.7692828
17 - 3.8633575
12 - 0.31003436
10 - 19.872202
17 - 8.419615
0 - 8.653201
19 - 19.992702
9 - 0.018264936
15 - 12.856614
17 - 14.902431
En la fila 10 el codigo 10
ocurre 2 veces
En la columna 10 el valor
promedio de los valores
asociados al codigo 10 es de
12.118072032928467
```

Ejercitación práctica sugerida

COMPOSICION USANDO UNA SUCESION DE NUMEROS

Usando las clases Numero(pg 3) y Sucesión(pg 4), modifique sus comportamientos para satisfacer el siguiente enunciado:

- Procesar una sucesión de números compuesta de N términos. Los términos no deben ser exhibidos. N es informado por teclado.
- Contabilizar los términos divisibles por M. M es informado por teclado.
- Informar que porcentaje de términos fueron divisibles.

COMPOSICION USANDO UNA SUCESION DE CARACTERES

Usando las clases Carácter(pg 5) y SuceCar(pg 6), modifique el comportamiento de esta última para satisfacer el siguiente enunciado:

- Contabilice letras, dígitos decimales y signos de puntuación.

- Por fin de secuencia exprese que total fue mayor, intermedio y menor.

Usando las clases Carácter(pg 5) y SuceCar(pg 6), modifique el comportamiento de esta última para satisfacer el siguiente enunciado:

- cantidad de mayúsculas.
- Cuantas letras son a su vez dígitos hexadecimales?

COMPOSICION USANDO UNA PROGRESION DE CARACTERES

Usando las clases Carácter(pg 5), Alternan(pg 7) y Progresión(pg 8) modifique el comportamiento que sea necesario de la dos últimas para satisfacer el siguiente enunciado:

- Cuantas alternancias DigitoDecimal Letra, en ese orden, tenemos?
- Cuantas alternancias DigitoDecimal Letra, en cualquier orden, tenemos?

Usando las clases Carácter(pg 5), Alternan(pg 7) y Progresión(pg 8) modifique el comportamiento que sea necesario de la dos últimas para satisfacer el siguiente enunciado:

- Cuantos caracteres de palabra(Letras, dígitos decimales) tenemos?
- Cuantos caracteres separadores (Inclusive blancos) tenemos?
- Cual es el porcentaje de cada uno de ellos?

MÁS UNA COMPOSICION USANDO UNA PROGRESION DE CARACTERES

Modifique el comportamiento que sea necesario de las clases Palabra(pg 10) y Frase(pg 11) para satisfacer el siguiente enunciado:

- Cuantas palabras contienen Dígitos Decimales?
- Cuantas palabras contienen mas dígitos decimales que letras?
- Que porcentaje constituye el segundo punto respecto al total de palabras?

Modifique el comportamiento que sea necesario de las clases Palabra(pg 10) y Frase(pg 11) para satisfacer el siguiente enunciado:

- Cuantas palabras inician en vocal y terminan en consonante?
- Cuantas palabras no tienen dígitos decimales?
- Cuantas palabras tienen letras mayúsculas internas?

Modifique el comportamiento que sea necesario de las clases Palabra(pg 10) y Frase(pg 11) para satisfacer el siguiente enunciado:

- Cuantas palabras tienen mayoría vocales?
- En cuantas palabras son iguales la cantidad de vocales y consonantes?
- Cuantas palabras inician y terminan en el mismo carácter?

COMPOSICION USANDO UN VECTOR DE ELEMENTOS

Modifique lo que sea necesario de las clases Hotel2(Pg 11) y Habitación(pg 1) para que sea posible informar, no las comodidades, sino sobre las disponibilidades del hotel.

COMPOSICION USANDO UN VECTOR DE ELEMENTOS

Modifique el comportamiento que sea necesario de las clases Elemento(pg 13) y Vector(pg 13) para satisfacer el siguiente enunciado:

- Informar también el mayor, posición y valor.
- La cantidad de elementos a procesar debe ser informada externamente.
- En que valor difiere el mayor del valor promedio?

Modifique el comportamiento que sea necesario de las clases Elemento(pg 13) y Vector(pg 13) para satisfacer el siguiente enunciado:

- Cuantos elementos superan al valor promedio?
- En que posiciones del vector se encuentran estos elementos?
- Son estos elementos mayoría? (Mitad mas uno)

COMPOSICION USANDO UNA SECUENCIA DE NUMEROS

Modifique el comportamiento que sea necesario de las clases Numero(pg 14) y Secuencia(pg 15) para satisfacer el siguiente enunciado:

- Cantidad de veces que la secuencia cambia de orden. Un cambio de orden significa que los números invierten el orden que hasta el momento llevaban.
  1, 2, 5, 3, 6 // 2 cambios de orden
  6, 6, 6, 7, 8, // 0 cambios de orden

Modifique el comportamiento que sea necesario de las clases Numero(pg 14) y Secuencia(pg 15) para satisfacer el siguiente enunciado:

- Cantidad de números que tiene la sub secuencia mas larga, en cualquier orden.
  1, 2, 3, 5, 5, 5, 4, // 4 numeros
  2, 2, 2, 3, 7, 6, 5, 3, // 4 numeros

COMPOSICIONES USANDO HERENCIA

Use las clases Progresión(pg 20) y ArithProgresion(pg 21). Extienda esta última (ArithProgres01) y modifique su comportamiento en todo lo necesario para que esta clase genere una serie de números, (Random o informados). Terminada la generación (Numero 999 o lo que UD. defina) informe la cantidad de números pertenecientes a la serie y cuantos no. Por definición de serie, considere el primer termino perteneciente a la serie.

Extienda su clase ArithProgres01 en ArithProgres02. Al comportamiento de la clase base agregue el cálculo de porcentajes de términos en serie y no en serie.

Extienda su clase ArithProgres01 en XProgres03. La única variante que debe ser introducida es que **la razón de esta XSerie se incrementa en 1** por cada término generado random o leído. Como la generación o lectura son independientes de esta incrementación, habrá términos que satisfacen la serie y que no.

Use ArithProgres01 como clase base y extiéndala en una nueva clase PoliProgress que detecta si los términos están en serie aritmética, geométrica o Fibonacci. Tenga en cuenta que algunos pueden pertenecer a mas de una. Totales de términos pertenecientes a cada serie y ninguna.

Extienda su clase PoliProgress en AnyProgress. Esta clase debe contabilizar cuantos términos pertenecen a alguna clase o ninguna.

COMPOSICIONES TRATANDO EXCEPCIONES

Extienda la clase Excep01 en MiExc01 modificando lo que sea necesario de manera que en ella no se produzca la excepción esperada. (Puede haber sorpresas …)

Use la clase Excep02(pg 26) modificándola si es necesario y de ella derive una nueva clase MyExc02 que debe informar el promedio de los cociente calculados, haya excepción o no.

Extienda la clase RealDivideExcep(pg 26) en una clase MyRealDivide que contabiliza la cantidad de excepciones de division por cero producidas.

COMPOSICIONES TRATANDO BÚSQUEDA EN ARRAYS

Extienda la clase Búsqueda(pg 33) en MyBusSec incorporando un método que nos informe cuantas veces existe cada codigo del array de ítems. Atención: Necesitamos que nos informe (el código y cuantas veces existe) 1 sola vez, no tantas veces como ese código esté repetido en el array.

Extienda la clase Búsqueda(pg 33) en MyPrueba. El tema es conocer cuantos códigos equidistantes del centro del array de ítems son iguales. (El primero al ultimo, el segundo al penúltimo, …)

Extienda la clase BúsqBin(pg 35) en MyBusqBin. Se necesita verificar previamente si el array en el cual pretendemos buscar está ordenado ascendente.

Extienda su clase MyBusqBin en MyBusqBin02. Se pide incorporar comportamiento contenga un ciclo que informe la cantidad de comparaciones realizada para una clave informada por teclado, utilizando búsquedas secuencial y binaria.

COMPOSICIONES TRATANDO ARRAYS MULTIDIMENSIONALES

Extienda la clase Prumat05(pg 44) en MyMat04 incorporándole el siguiente comportamiento:

- Un método toStringCols() que permite la visualización de la matriz recorriendo los elementos por columna, de izquierda a derecha.
- Un método getTotFila(int fila) que retorna el total de la fila indicada.
- Un método getTotCol (int col) que retorna el total de la columna indicada.
- Un método getMaxValFila(int fila) que retorna el mayor valor de la fila indicada
- Un método getMaxValor() que retorna el mayor valor de la matriz. Usa getMaxValFila(int fila).
- Un método getPosMaxValFila(int fila) que retorna la posición (columna) del mayor valor de la fila indicada

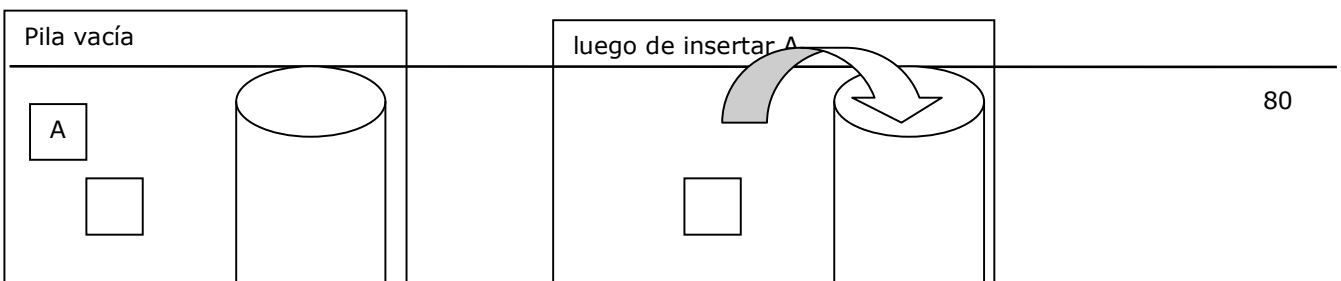
Extienda la clase Prumat05 en MyMat05 incorporándole comportamiento que permite trabajar con dos objetos matriz, uno el invocante (referenciado por this) y el segundo pasado por parámetro.

- obj1.esMayor(obj2) devuelve verdadero si obj1 es mayor el que obj2. Se considera que obj1 es mayor que obj2 si obj1 tiene más elementos mayores que los respectivos de obj2.
- obj1.cociente(obj2) retorna un objeto matriz cuyos elementos son cociente de los respectivos de obj1/obj2. Si en esta operación tenemos excepción por división, retorna una matriz vacía.

Estructuras Lineales: Pilas, Colas

CONCEPTO DE PILA: Una **pila (stack)** es una colección ordenada de elementos a los que solo se puede acceder por un único lugar o extremo. Los elementos de la pila se añaden o se quitan (borran) de la misma sólo por su parte superior (**cima).** Entonces, **Insertar y Excluir son las únicas operaciones lícitas (posibles) en una pila.** . Las pilas se conocen también como estructuras **LIFO** *(Last in, First-out),* último en entrar-primero en salir)

Al operarse únicamente por la parte superior de la pila, al excluir elementos retiramos el que está en la cima, el último que incluimos. Si repetimos esta operación retiraremos el penúltimo, el antepenúltimo, o sea en orden inverso al que los incluimos. Supongamos que tenemos las letras 'A', 'B' y 'C', las incluimos y luego las excluimos, esto visto “cuadro a cuadro”:



La operación de **Insertar** sitúa un elemento dado en la cima de la pila y **Excluir** lo elimina o quita

El lenguaje Java no tiene sentencias nativas para tratamiento de pilas, a diferencia de los arrays, que implementan colecciones de objetos de tamaño fijo. Entonces debemos "pedir prestado" recursos a otras clases y, a veces, programar lo que falte. Se pueden implementar pilas usando:

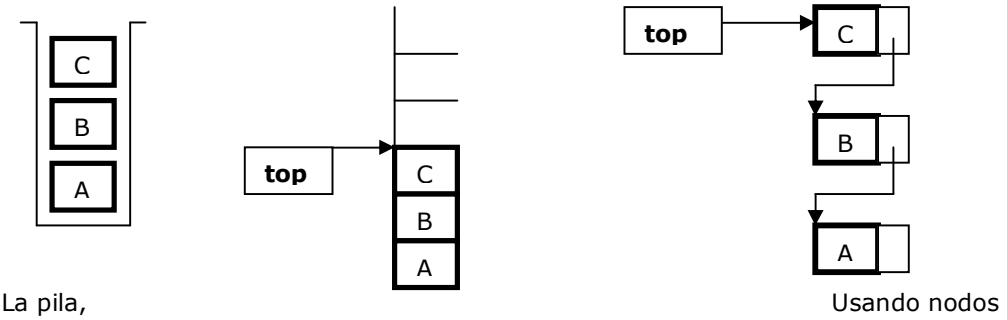
- **Arrays**. Es una implementación adecuada y de eficiente utilización. Tiene el serio inconveniente de que el array tiene tamaño fijo, lo que lo hace inadecuado si la pila, que es por definición de tamaño variable, lo puede estar subutilizando o sobreutilizando; respecto esta última situación, siempre deberemos verificar si exite disponibilidad de casilleros en el array antes de incluir un elemento. Ver figura abajo.

- **Nodos vinculados**. No tiene el inconveniente del tamaño de los arrays. Esta estructura contendrá tantos nodos como elementos apilados. Debemos programarla nosotros, pero es una programación muy simple. Ver figura abajo.

- **Usar la clase de colección Stack**, provista por Java. Esta clase, que extiende la colección Vector, provee 5 métodos con todo el comportamiento necesario para implemetar una pila. Por ser una extensión de Vector, sus métodos son sincronizados, aptos para programación concurrente. Esto tiene un costo adicional y puede ser un inconveniente, si no lo necesitamos. No debemos programar nada, solo implementar nuestra pila.

- **Usar la clase de colección LinkedList.** Es como trabajar con nodos vinculados, solo que ahora todos la programación está lista. Es la mejor solución, a menos que precisemos de sincronización; las soluciones basadas en array implementan una estructura de datos que posibilitan indización, cosa que aquí no necesitamos; en la pila trabajamos unicamnte en su extremo superior.

Gráficamente, si tenemos 'A', 'B', 'C' "apiladas".



La pila, Conceptualmente | Usando Array | Usando nodos vinculados

En la implementación con arrays, **top** es el índice del último casillero ocupado, (o del primero disponible). Es simple, ya dijimos que tiene el inconveniente de soportar una estructura típicamente dinámica (la pila) con una estática (el array): El array puede ser demasiado grande, o pequeño.

En la implementación con nodos vinculados **top** es un puntero al primer nodo de esta estructura lineal, el cual contiene siempre el último elemento insertado. Su implementación requiere un poco más de trabajo, pero al ser ambas estructuras del mismo tipo no existen los inconvenientes de la anterior. Por ello, esta es la que veremos en detalle.

La pila es una estructura importante en computación. Algunos de sus usos, transparentes para usuarios de un lenguaje de programación, es el retorno automático al punto en que una función fue invocada. Supongamos que main() invoca a func1(); dentro de func1() se invoca a func2(); dentro de func2() se invoca a func3(), allí procesamos y llegamos a la sentencia return. Una pila es la que

contiene la información para que el retorno sea a un punto del cuerpo de func2(), luego de dirigirá nuestro retorno a func1(), y finalmente al main().

Veamos la implementación con nodos vinculados.

```
package implpilas;
class Nodo{   // Nodo de una estructura enlazada simple
    private Item item; //  Nodo tiene un Item y
    private Nodo prox;  //  una referencia al próximo Nodo...

    public Nodo(Object item, Nodo sig) {// Constructor parametrico
        this.item = (Item)item;
        prox = sig;
    }

    public Item getItem(){return item;} // Retornamos item almacenado en Nodo

    public void setItem(Item it){item = it;};     // Inicializamos idem

    public Nodo getProx(){return prox;} // Retornamos puntero al proximo nodo

    public void setProx(Nodo ptr){prox = ptr;}// Inicilizamos puntero prox

    public String toString(){   //     Exhibimos datos de Nodo
       String aux = ""+item.toString()+"\n";   // Usamos toString() de Item
       return aux;
    }
}

package implpilas;
class Pila { //  Implementando Pila usando NodosEnl
    private Nodo top;  //  Referencia de entrada
    private boolean trace; // Trazamos ?

    public Pila(){top = null; trace = false;}  // Constructor

    public boolean vacia(){return top == null;} // No hay nodos

    public String toString(){
       String aux = "Pila contiene ";
       int cont = 0;
       Nodo auxPtr;
       for(auxPtr = top; auxPtr != null; auxPtr = auxPtr.getProx())cont++;
       aux+= cont + " elementos\n";
       return aux;
    }

    public void push(Object obj){
        top = new Nodo(obj,top);
        if(trace) System.out.print("incluido "+top);
    }

    public Nodo pop(){
       if(vacia()) return null; // Retornamos referencia nula
       Nodo ptr = top;    // Guardamos en aux la referencia del primer nodo
       top = top.getProx(); // Redirecionamos frente, saltando el primer nodo
       if (trace)System.out.print("excluido "+ptr);
       return ptr; //     Retornamos referencia nodo desvinculado
```

```
    }

    public void setTrace(boolean trazar){trace = trazar;}
}
```

```
package implpilas;
public class Main {
    public static void main(String[] args) {
        ArrItems arrItems = new ArrItems(5,'R');
        Pila pila = new Pila();
        pila.setTrace(true);
        for(int i = 0; i < arrItems.talle; i++)
            pila.push(arrItems.item[i]);
        for(int i = 0; i < 3; i++)
            pila.pop();
        System.out.println(pila);
    }
}
```

```
run:
incluido 0 - 2.5075107
incluido 3 - 1.7951043
incluido 0 - 3.8639455
incluido 4 - 0.630951
incluido 3 - 0.86312217
excluido 3 - 0.86312217
excluido 4 - 0.630951
excluido 0 - 3.8639455
Pila contiene 2 elementos
```

Bueno, ahora queremos comparar eficiencia de esta implementación con respecto a las otras implementaciones provistas por Java: Stack y LinkedList. Para ello:

```
package analpilas;
import java.util.*;
public class Tiempos{
    Calendar cal1, cal2, cal3, cal4, cal5, cal6;
    int push, pop;
    long tpoIni, tpoFin, tiempo;
    String resultados;
    public Tiempos(){
        System.out.println("Cuantos items apilamos?");
        push = In.readInt();
        System.out.println("Cuantos items desapilamos?");
        pop = In.readInt();
        resultados ="Apilaremos : "+push+", desapilaremos "+ pop+"\n";
        resultados+="Metodo, Inicio, Fin, Tiempo\n";
    }

    public void proceso(){
        ArrItems arrItems = new ArrItems(push,'R');
        System.out.println(arrItems);
        Pila pila = new Pila();
        //  pila.setTrace(true);
        cal1 = new GregorianCalendar();
        tpoIni = cal1.getTimeInMillis();
        for(int i = 0; i < push; i++)
            pila.push(arrItems.item[i]);
        for(int i = 0; i < pop; i++)
            pila.pop();
        cal2 = new GregorianCalendar();
        tpoFin = cal2.getTimeInMillis();
        tiempo = tpoFin - tpoIni;
        resultados+="Usando nodos vinculados "+tpoIni+", "+tpoFin+", "+tiempo+"\n";
        System.out.println(resultados);
        // Usando class Stack
        Stack stack = new Stack();
        cal3 = new GregorianCalendar();
```

```
        tpoIni = cal3.getTimeInMillis();
        for(int i = 0; i < push; i++)
            stack.push(arrItems.item[i]);
        for(int i = 0; i < pop; i++)
            stack.pop();
        cal4 = new GregorianCalendar();
        tpoFin = cal4.getTimeInMillis();
        tiempo = tpoFin - tpoIni;
        resultados ="Usando class Stack  "+tpoIni+", "+tpoFin+", "+tiempo+"\n";
        System.out.println(resultados);
        // Usando class LinkedList
        LinkedList linked = new LinkedList();
        cal5 = new GregorianCalendar();
        tpoIni = cal5.getTimeInMillis();
        for(int i = 0; i < push; i++)
            linked.push(arrItems.item[i]);
        for(int i = 0; i < pop; i++)
            linked.pop();
        cal6 = new GregorianCalendar();
        tpoFin = cal6.getTimeInMillis();
        tiempo = tpoFin - tpoIni;
        resultados ="Usando LinkedList   "+tpoIni+", "+tpoF
        System.out.println(resultados);
        System.out.println("Demo terminado!!!");
    }
}
```

```
run:
Cuantos items apilamos?
100000
Cuantos items desapilamos?
50000
 Primeros 10 de 100000
 elementos Item
47600 - 63635.363
35296 - 28377.375
35270 - 45458.695
89405 - 80822.83
60210 - 41252.98
89428 - 97108.41
55753 - 85464.64
5909 - 68967.55
9887 - 10521.339
10496 - 51154.19

Apilaremos      :      100000,
desapilaremos 50000

Metodo, Inicio, Fin, Tiempo
Usando    nodos    vinculados
1202228181175, 1202228181226,
51

Usando       class       Stack
1202228181236, 1202228181286,
50

Usando              LinkedList
1202228181286, 1202228181416,
130

Demo terminado!!!
```

```
package analpilas;
public class Main {
    public static void main(String[] args) {
        Tiempos tiempos = new Tiempos();
        tiempos.proceso();
    }
}
```

El proceso se corrió en un equipo Pentium® 4, CPU 1.60 GHz, 0.98 GB RAM, Windows XP prof., 2002, service pack 2

En 3 corridas con 100.000 items apilados, los tiempos fueron muy parecidos a los indicados: **51**, **50**, **120** mseg.

Se hicieron otras3 corridas apilando 20.000 items y desapilando 10.000. Aquí los tiempos promedio son de 33 para nodos vinculados, 17 para class Stack y 10 para Linked List.

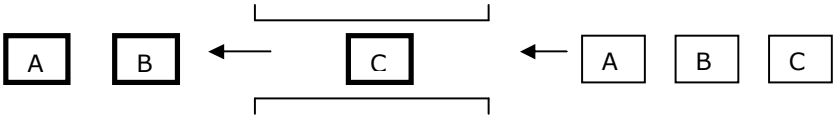
En conclusión:
Con 100.000 items gana Class Stack
Con 20.000 gana class LinkedList
Nuestra programación segunda siempre.

CONCEPTO DE COLA

Una **cola** es una estructura de datos que almacena elementos en una fila (uno a continuación de otro). Cada elemento se inserta en el fondo de la cola y se suprime o elimina por el frente (top). Las aplicaciones utilizan una cola para almacenar y liberar elementos en su orden de aparición o concurrencia. **Insertar y Excluir son las únicas operaciones lícitas (posibles) en una cola** y difieren del comportamiento

de la pila en que estas operaciones se realizan en extremos opuestos. **Insertar** por el fondo, **excluir** por el frente.

Entonces **Insertar y Excluir** se realizan en el mismo orden. Un buen ejemplo de cola es el de personas esperando utilizar el servicio público de transporte. (Si dijéramos un mal ejemplo también es verdad). La gestión de tareas de impresión en una única impresora de una red de computadoras es otro ejemplo. Supongamos que tenemos las letras 'A', 'B' y 'C', las incluimos y luego excluimos dos de ellas, necesariamente 'A' Y 'B'. Gráficamente:



Excluidas frente (top) fondo Antes de insertar

Como en el caso anterior, el lenguaje Java tampoco tiene sentencias específicas para tratamiento de colas. Para trabajar con colas debemos "pedir prestado" recursos a otras estructuras o clases. Podemos programar implementaciones de colas en arrays, usar nodos vinculados por punteros o usar alguna delas muchas clase que Java ofrece. Si consultamos el Help, tenemos:

Queue: All Known Implementing Classes:
AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, PriorityBlockingQueue, PriorityQueue, SynchronousQueue

Hay alternativas "para que tengan…"

Implementar una cola en un array tiene más inconvenientes que la pila, por lo que no decimos nada de eso.

En la implernentación con nodos vinculados **top** es un puntero al primer nodo de esta estructura lineal, el cual referencia siempre el elemento insertado mas antiguo y **bot** al mas reciente, último. Esta implernentación requiere un poco más de trabajo, pero al ser ambas estructuras del mismo tipo dinámico no existen los inconvenientes de usar arrays. Por ello, esta implementación es la que veremos en detalle y dejamos a los alumnos la experimentación de aplicar alguna de las clases arriba citadas.

En general, la cola es la estructura de gestión de recursos de alguna manera escasos. Impresoras que atienden a un grupo de estaciones de trabajo en una red, tablas de bases de datos que varios usuarios pretenden actualizar en forma exclusiva, puesto de peaje un domingo a las 21.00hs, etc, etc.

```
package implcolas;
public class Main {
    public static void main(String[] args) {
        ArrItems arrItems = new ArrItems(5,'R');
        Cola cola = new Cola();
        cola.setTrace(true);
        for(int i = 0; i < arrItems.talle; i++)cola.add(arrItems.item[i]);
        for(int i = 0; i < 3; i++)cola.remove();
        System.out.println(cola);
    }
}
```

```
package implcolas;
class Cola{
    protected Nodo top, bot;  //  Referencia de entrada
    protected boolean trace; // Trazamos ?
```

```
    public Cola(){            //    Un constructor sin argumentos
        top = bot = null;
        trace = false;
    }
    public void setTrace(boolean trazar){trace = trazar;}

    public boolean vacia(){return top == null;} // No hay nodos

    public void add(Object item){
      Nodo aux;
      if (top == null){ //    no teníamos nodos, este es el primero
            bot = top = aux = new Nodo(item,null);
      }else{            //  Ya teniamos, este es uno mas ...
            aux = new Nodo(item,null);
            bot.setProx(aux);
            bot = aux;
      }
      if(trace)
            System.out.print("incluido "+aux);
    }

    public Nodo remove(){
      if(vacia()) return null; // Retornamos referencia nula
      Nodo ptr = top;   // Guardamos en aux la referencia del primer nodo
      top = top.getProx(); // Redirecionamos top, saltando el primer nodo
      if(top == null)   //    No hay ya nodos ...
            bot = null;
      if (trace)
            System.out.print("excluido "+ptr);
      return ptr; //    referencia nodo liberado
    }


public String toString(){
      String aux = "Cola contiene ";
        int cont = 0;
        Nodo ptr;
        for(ptr = top; ptr != null; ptr = ptr.getProx())cont++;
      aux+= cont + " elementos\n";
      return aux;
    }
}
```

```
run:
incluido 3 - 1.807499
incluido 2 - 0.03642479
incluido 4 - 0.9543581
incluido 2 - 1.3906887
incluido 3 - 0.76472265
excluido 3 - 1.807499
excluido 2 - 0.03642479
excluido 4 - 0.9543581
Cola contiene 2 elementos
```

Recursividad

La **formulación recursiva** siempre consta de estos dos puntos.

a) El caso conocido, o por lo menos directamente deducible.
b) El planteo en términos de si mismo, siempre de un grado menor que el anterior.

El punto b) debe plantearse de forma que mas tarde o temprano llegue a ser el punto a.

Y no hay más que decir de la recursividad. Pero para que quede mas claro vamos a ilustrarlo con el ejemplo clásico, que sale en todos los libros: El **factorial de n**, siendo n un número entero positivo.

```
class Factorial{
        public Factorial(){;}
        public int factorial(int n){
                int fact;
                if (n == 0){
                        System.out.println("Llegamos al caso conocido: 0! = 1");
                        System.out.println("Retorno 1 a las invocaciones pendientes");
                        return 1;
                }else{
                        System.out.println("Invocando a factorial("+(n-1)+")");
                        fact = n*factorial(n - 1);
```

```
                System.out.println("Tengo n = "+n + ", fact = "+ fact);
                return fact;
        }
    }
    public static void main(String args[]){
        Factorial calculo = new Factorial();
        System.out.println("El factorial de 5 es:"+calculo.factorial(5));
    }
}
```

```
Invocando a factorial(4)
Invocando a factorial(3)
Invocando a factorial(2)
Invocando a factorial(1)
Invocando a factorial(0)
Llegamos al caso conocido: 0! = 1
Retorno 1 a las invocaciones pendientes
Tengo n = 1, fact = 1
Tengo n = 2, fact = 2
Tengo n = 3, fact = 6
Tengo n = 4, fact = 24
Tengo n = 5, fact = 120
El factorial de 5 es:120
Process Exit...
```

Introducción

Cuando dentro del cuerpo de un método invocamos a otro, el lenguaje (Java, C, todos los lenguajes imperativos) guarda en una pila la situación de todas sus variables locales y el punto de la instrucción desde la cual se hace la llamada. Gracias a esto es que la sentencia return nos lleva al punto exacto en el cuerpo del método llamador y allí retomamos los cálculos. Y así podemos ir retornando y retornando, haciendo el camino inverso al que recorrimos cuando fuimos invocando los métodos. Un detalle de esto vimos cuando estudiamos excepciones, que podían ser tratadas en la lista catch del método donde se produjo la excepción o en otra lista perteneciente a un método mas abajo en la pila de llamadas.

En la recursividad **no es diferente**. Al llamar apilamos los valores de las variables locales y el punto de llamada. Al retornar, desapilamos, los valores guardados en la pila reemplazan sus respectivos valores en las variables corrientes y el procesamiento continúa en el punto inmediato siguiente al de llamada. Una particularidad, no una diferencia, es que podemos retornar varias veces al mismo punto, tantas como desde allí llamamos.

Interpretemos calculo.factorial(5)

Invocaciones (Camino hacia delante).
"Invocando a factorial(4)": dentro de la intrucción **fact = n*factorial(n – 1)** hay un llamado a factorial pasando un argumento n – 1. El **producto no llega a concretarse**, tiene prioridad la resolución de llamada a método. Y esto ocurre cuatro veces mas, la última le pasamos argumento = 0. Aquí la lógica va por la otra rama. Estamos en el

Caso conocido
Factorial de 0 es 1. Retornamos 1, entonces. Y ese es el valor que aportará factorial() al primer producto pendiente de resolución. Esto ya ocurre en

Retornos (Camino de vuelta atras).
Se termina **de concretar el producto pendiente**. Podemos apreciar que el valor retornado es multiplicado por el valor de n "reposicionado" a partir del valor apilado, que es el que n tenía en el momento de la invocación. Esto es exhibido por **System.out.println("Tengo n = "+n + ", fact = "+ fact);** Llegamos al fin del cuerpo del método factorial(). La pila comanda el próximo retorno. Puede ser de

nuevo al producto pendiente, llegará un momento que retornaremos al metodo llamador, el main() en este caso.