

## MODULARIDAD

Además de la abstracción y el encapsulamiento, uno de los principios fundamentales del diseño orientado a objetos es la **modularidad**. 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. Cada 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. Un ejemplo más esclarecedor, se puede considerar que una casa o un departamento consisten en varias unidades que interactúan: las instalaciones eléctricas, de calefacción y enfriamiento, el servicio sanitario y la estructura. Más que considerar que esos sistemas son un enredo gigantesco de alambres, ventilaciones, tubos y tableros, el arquitecto organizado que diseña una casa o departamento los considera como módulos separados que interactúan en formas bien definidas. Al hacer esa consideración se aplica la modularidad para aclarar las ideas, en una forma natural de organizar las funciones en unidades distintas y manejables. Otra forma que nuestro arquitecto "organizado" puede usar para ordenar su rompecabezas es el de la estructura jerárquica, donde cada enlace que va hacia arriba se puede leer como "**es un(a)**", como en "una cabaña es una casa es una construcción". En forma parecida, el uso de la modularidad en un sistema de programación también puede proporcionar una poderosa estructura organizativa que aporte claridad a una implementación.

Y volviendo a 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**": Un atributo del objeto **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
" <b>tiene un</b> "	El objeto de nuestra clase tiene atributos que son objetos de otras clases
" <b>usa un</b> "	Métodos pertenecientes al objeto de nuestra clase requieren del comportamiento de otras clases
" <b>es un</b> "	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.

## REFERENCIAS EN JAVA

Las referencias en Java son identificadores de instancias de las clases Java. Una referencia dirige la atención a un objeto de un tipo específico. No tenemos por qué saber cómo lo hace ni necesitamos saber qué hace ni, por supuesto, su implementación.

A continuación vamos a utilizar un ejemplo para demostrar el uso y la utilización que podemos hacer de las referencias en Java:

Pensemos en una referencia como si se tratase de la llave electrónica de la habitación de un hotel. Primero crearemos la clase **Habitacion**, implementada en el fichero Habitacion.java; Luego definiremos una clase Hotel, y lo crearemos usando instancias de Habitación.

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

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

    public Habitacion( int numHab,int numCam) { // (2) Sobrecarga del constructor
        numHabitacion = numHab;
        numCamas = numCam;
    }
}
```

```

    }

    public Habitación(Habitación hab) {           // (3)    mas sobrecarga del constructor
        numHabitacion = hab.numHabitacion;
        numCamas      = hab.numCamas;
    }

    public int getNumHab() {                       // (4)    Metodo público
        return numHabitacion;
    }

    public int getNumCam() {                       // (5)    Método público
        return numCamas;
    }

    public void setNumHab (int numHab){           // (6)    Método público
        numHabitacion = numHab;
    }

    public void setNumCam (int numCam){           // (7)    Método público
        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; tal como muestra el código siguiente, Hotel1.java:

```

import Habitación;
public class Hotel1 {
    public static void main( String args[] ) {
        Habitación hab1, hab2;           // Definimos referencias a habitaciones
        hab1 = new Habitación(1,2);      // Instanciamos usando constructor (1)
        hab2 = new Habitación(2,3);      // Instanciamos usando constructor (2)
        Habitación hab3, hab4;           // dos nuevas habitaciones estan listas
        hab3 = new Habitación(hab2);      // Tiene la capacidad de la hab2
        hab4 = new Habitación(hab2);      // Tiene la capacidad de la hab2
        hab3.setNumHab(3);                // Su número correcto es 3
        hab4.setNumHab(4);                // Su número correcto es 4
        Habitación hab5, hab6;           // queremos ampliar el hotel,
        hab5 = new Habitación();          // pronto le incorporaremos
        hab6 = new Habitación();          // dos habitaciones mas ...
    }
}

```

Nótese que la clase **Hotel1** no tiene objetos propios, 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 quisiéramos tener un método que nos permita obtener un listado de las habitaciones con la cantidad de camas que tienen, tendríamos dificultades. Ocurre que las habitaciones que estamos declarando y usando son locales a `main()`, no miembros de la clase **Hotel1**. Solucionaremos esto.

## SUCESIONES, PROGRESIONES, SERIES, SECUENCIAS.

A modo de ejemplo, comenzaremos tratando problemas relativamente sencillos que utilizan sucesiones, progresiones, series o secuencias de algo( caracteres, números).

Una primera duda: Es lo mismo sucesión, progresión, serie, secuencia o son conceptos diferentes?.

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

**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.

## COMPOSICION USANDO UNA SUCESION DE NUMEROS

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.

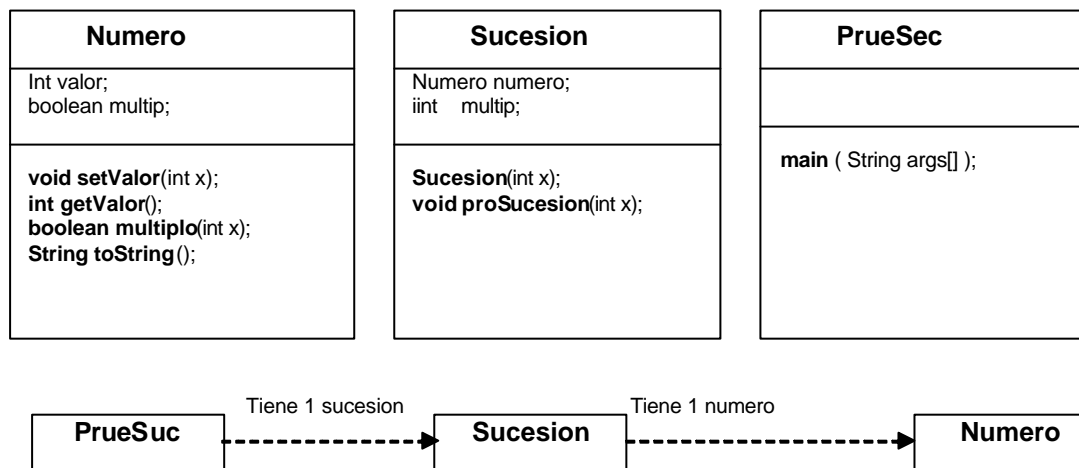
O que sea tendremos dos clases, y cual será la relación entre los objetos de ellas?

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”, una instancia **Número**.



Veamos esto en Java.

```

class Numero{
    private int valor;          // valor del numero
    private boolean multip;     // es o no multiplo?
    public void setValor(int x) // Genera el valor del numero aleatoriamente
    {
        valor=(int)(100*Math.random());
        multip = multiplo(x);
    }
    public int getValor(){ // retorna el valor del numero
        return valor;
    }

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

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

En la clase **Numero** tenemos dos atributos.

valor, el propio valor del numero entero.

Multip, un booleano que dice si es o no múltiplo (del parámetro pasado a setValor())

Obsérvese la elaboración de toString(), que devuelve la hilera con información del objeto

// Sucesion de números enteros

```
import Numero;
class Sucesion{
    Numero numero;           // Referencia a Numero
    int multip;              // valor para chequear multiplicidad

    public Sucesion(int x){ // Constructor
        numero = new Numero(); // el constructor de numeros
        multip= x;
    }
    public void proSucesion(int x){
        do{
            numero.setValor(multip);
            System.out.println(numero.toString());
        }while(numero.multiplo(multip));
    }
}
```

```
Deseo procesar una sucesion
de numeros divisibles por 3
Numero divisible 9
Numero divisible 84
Numero divisible 6
Numero no divis. 61
Process Exit...
```

En la clase **Sucesion** también usamos dos atributos  
numero, objeto de la clase Numero.

multip, valor para chequear multiplicidad

Obsérvese como proSucesion utiliza toString() de Numero para exhibir su propio objeto.

```
import In;
import Sucesion;
public class PrueSuc{
    public static void main(String args[]){
        System.out.println("\nDeseo procesar una sucesion");
        System.out.print("de numeros divisibles por ");
        int aux = In.readInt();
        Sucesion suc = new Sucesion(aux); // Instanciamos la sucesion
        suc.proSucesion(aux);              // y la procesamos ...
    }
} // PrueSuc
```

## COMPOSICION USANDO UNA SUCESION DE CARACTERES

**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 Carácter** ...

```
import java.io.IOException;
public class Carácter{
    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 minúscula ?
    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 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{ // Controlar secuencia
        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;
    }
};

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
    }
};

```

```

Introduzca una sucesi#n de
caracteres, finalizando con #

Aquí me pongo a cantar ...#
Totales proceso
Vocales      9
Consonantes  9
Digitos dec. 0
Terminado !!!

```

## COMPOSICION USANDO UNA PROGRESION DE CARACTERES

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

Porque hemos cambiado de **Sucesi#n para Progresi#n**?

Porque, para detectar alternancia, necesitamos de dos objetos Car#cter, no uno. El anterior y el actual. De acuerdo a nuestra convenci#n, **Progresi#n** 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 así
        || (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());
```

Reconocemos que a primera vista es bastante criptica, entonces la aclaramos:

El objeto **carAnt** invoca el método **setCar(...)**, quien recibe como 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 valuando 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 podría funcionar si 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;
}
```

Output 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\kawa\classes.zip;e:\jdk1.3\lib  
 File Compiled...

----- Compiler Output -----  
 Progresion.java:17: car has private access in Character  
 carAnt.car = carAct.car;  
 Progresion.java:17: car has private access in Character  
 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. Podríamos “suavizar” el encapsulado y retirarle a car el nivel private. Pasaría a ser friendly, veamos que pasa. Recompilamos Carácter, Progresion ...

Compilación OK. La ejecución también. 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. Además se “ralentiza”, más lento, hay más 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

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

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

## MÁS UNA COMPOSICION USANDO UNA PROGRESION DE CARACTERES

### Tratamiento de frases

(Cuántas palabras, cuántos caracteres, longitud promedio de la palabra)

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

(Cuántas palabras, cuántos caracteres, longitud promedio de la palabra)

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 Carácter 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 profe, ... 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)..

## CLASE FRASE

La clase **Frase**, usando comportamiento de la clase **Character**:

- Contabilizará palabras y caracteres.
- Calculará el promedio.
- Mostrará resultados

```
import Character;
```

```
public class Frase{
```

```
    private Character carAct, carAnt;
```

```
    private int contCar, contPal;
```

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

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

```
    public boolean actLetDig(){ // El actual es letra o digito ?
        boolean letDig = false;
        if (carAct.esLetra() || carAct.esDigDec()) // Pertenece a la palabra
            letDig = true;
        return letDig;
    }
```

```
    public boolean antLetDig(){ // El anterior es letra o digito ?
        boolean letDig = false;
        if (carAnt.esLetra() || carAnt.esDigDec()) // Pertenece a la palabra
            letDig = true;
        return letDig;
    }
```

```
    public boolean finPal(){           // Es fin palabra?
        boolean fPal = false;
```

```

        if (antLetDig() && !actLetDig())
            fPal = true;
        return fPal;
    }

    public void proFrase() throws java.io.IOException{           // Nuestro ciclo de lectura de caracteres
        do {
            carAct.lecCar();
            if (actLetDig()) contCar++;
            if (finPal()) contPal++;
            cicloCar();
        }while(carAct.getCar() != '#');           // Mientras la lectura no sea de un caracter '#'
        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;
    }
};

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

```

```

Introduzca caracteres, fin: #
to be or not to be, thas is the ...#

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

```

## REFERENCIAS Y ARRAYS (*Vectores de objetos*)

Java dispone de arrays de tipos primitivos o de clases. Vamos a hacer que la clase **Hotel2** tenga un objeto que sea un arreglo (array) de referencias a la clase **Habitación** y **que** tenga un método capaz de informarnos sobre sus comodidades.

```

import Habitacion;
public class Hotel2 {
    private int cantHab;           // Cantidad de habitaciones de nuestro hotel
    private Habitacion habitacion[]; // Una referencia al array de habitaciones

    public Hotel2(int cant){       // Un constructor del hotel
        cantHab = cant;
        habitacion = new Habitacion[cantHab]; // Instanciamos la referencia al array ...
        int canCam;                // Una variable de trabajo
        for(int i = 0; i < cantHab; i++){
            if(i < 3) canCam = 2;    // Las tres primeras habitaciones tienen 2 camas,
            else canCam = 3;         // el resto 3 camas
            habitacion[i] = new Habitacion(i+1,canCam); // Instanciamos cada referencia
            // pasamos a tener un array de objetos Habitación
        }
    }

    public void Comodidades(){
        System.out.println("Nuestro hotel dispone de las siguientes habitaciones");
        for(int i = 0; i < cantHab; i++)
            System.out.println("Hab. N " + habitacion[i].getNumHab() +

```

```

        }
    }
    " con " + habitacion[i].getNumCam() + " camas");
}

```

Como prometimos, ahora la clase Hotel2, mucho mas potente que Hotel1, puede contener en su objeto toda la información de sus comodidades. Y de mostrarlas.

Y ahora precisamos de una clase que se ocupe de ello. Va:

```

import Hotel2;
public class DemoHotel2 {
    public static void main( String args[] ) {
        Hotel2 miHotel = new Hotel2(6); // Instanciamos un objeto miHotel con 6 habitaciones
        miHotel.Comodidades();          // Mostramos sus comodidades
        System.out.println("Nada mas tenemos para ofrecerle ...");
    }
}

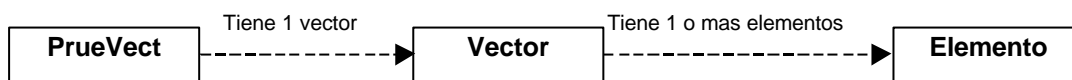
```

```

Nuestro hotel dispone de las ...
Hab. N 1 con 2 camas
Hab. N 2 con 2 camas
Hab. N 3 con 2 camas
Hab. N 4 con 3 camas
Hab. N 5 con 3 camas
Hab. N 6 con 3 camas

```

## COMPOSICION USANDO UN VECTOR DE ELEMENTOS



Encontrar el menor de los elementos numéricos del vector tiene mucho 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 compararse.

```

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 tamaño){
        element = new Elemento[tamaño]; // crea un vector de Elemento del tamaño informado
        for(int i=0; i<element.length; i++){
            element[i] = new Elemento(i); // Construye cada uno de los objetos Elemento
        }
        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(){    // El orden del menor elemento
    return orden;
}
}

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());
    }
}

```

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

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.**

## COMPOSICION USANDO UNA SECUENCIA DE NUMEROS

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 sí 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 sí 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

#### Atributos:

Valor

#### Responsabilidades:

Numero()  
InformarValor()  
CargarValor()  
Multiplo(x)

### SECUENCIA

#### Atributos:

valores  
tamaño

#### Responsabilidades:

Secuencia(x)  
CargarValores()  
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, podría 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;
```

```
// clase numero
```

```
class numero
```

```
{
    private int valor; // atributo de la clase
    public numero() // constructor. Inicializa en cero el valor cuando se crea un numero
    {
        valor = 0;
    }
    public void CargarValor() // carga desde el teclado el valor de un numero
    {
        valor=In.readInt();
    }
    public int InformarValor() // retorna al mundo exterior el valor del numero
    {
        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.
```

```
// clase secuencia
```

```
class secuencia
```

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

    public secuencia(int x) // crea un vector 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("\nTípe 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;
    }
}
```

#### Una ejecucion

Típe 5 numeros, separados por espacios

10 20 30 40 50

La cantidad de números multiplos de 4 es  
2

```
} // clase secuencia
```

Esto se ejecuta mediante un main() contenido en la siguiente clase

```
// clase programa
```

```
public class programasecuencia
```

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

#### Otra

Típee 5 numeros, separados por espacios

123 345 567 789 444

La cantidad de números multiplos de 4 es

1

Process Exit...

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.

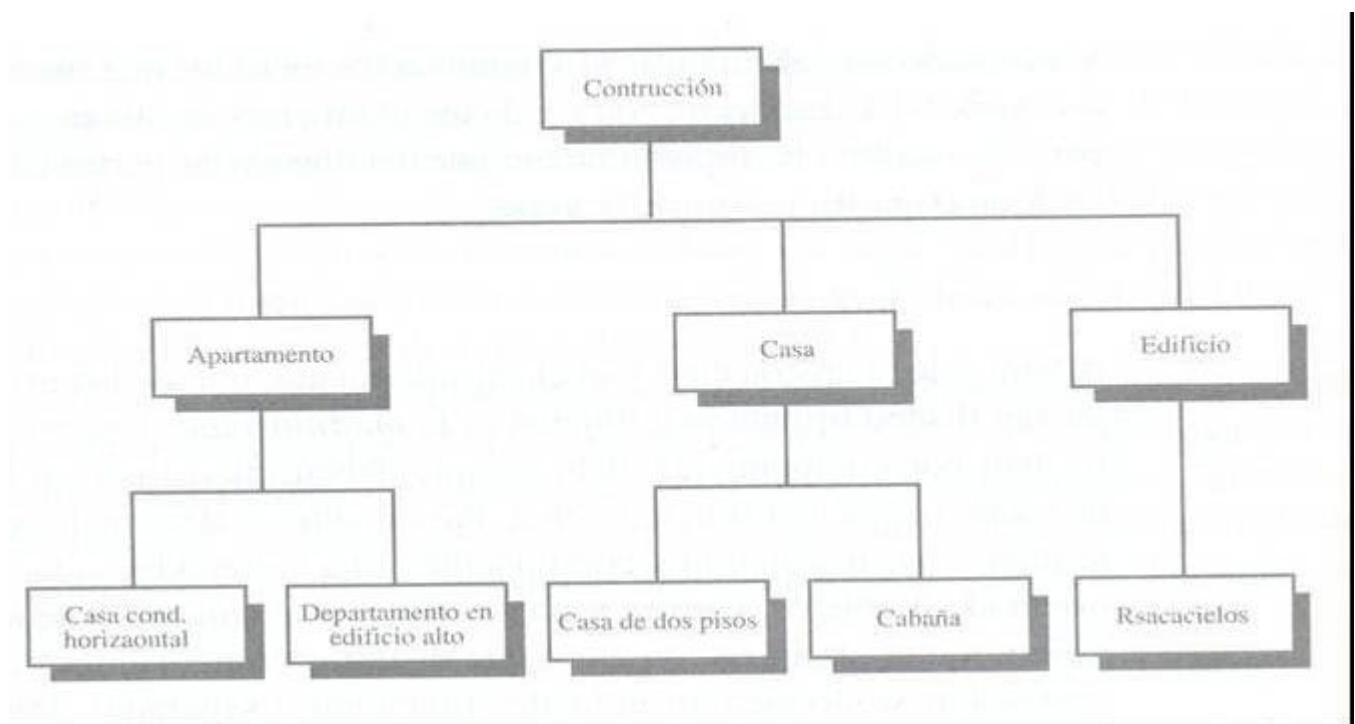
Podemos asegurar que se escribe bastante mas código para resolver un problema si lo comparamos con el paradigma estructurado, pero esto es muy notable solo en problemas simples y pequeños como este, de alguna manera, la infraestructura que rodea a la solución es un poco pesada para problemas tan sencillos. 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.

## HERENCIA

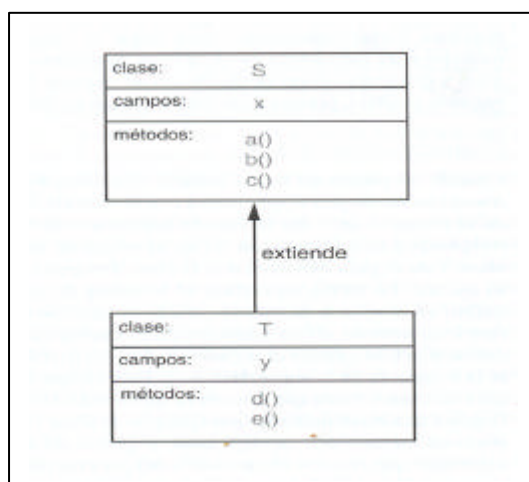
La organización impuesta por la modularidad torna que los módulos constituyentes de un programa sean **reutilizables**. Si los módulos del programa se escriben en forma abstracta, lo mas genérica posible, para resolver problemas generales, esos módulos se podrán usar de nuevo, cuando en otros contextos surjan instancias de esos mismos problemas. Puede haber muchos tipos de muro (Ladrillos, concreto, madera) pero una vez definido podremos usar esas especificaciones en muchos proyectos. Al reutilizar esa definición habrá partes que requieran redefinición; por ejemplo, un muro en una construcción domiciliar puede ser terminado con algún tipo de revoque, o puede ser ladrillo visto. En este segundo caso, la instalación eléctrica debe ser diseñada de forma distinta. El arquitecto, en consecuencia, podrá organizar los diversos componentes estructurales, como los eléctricos y estructurales, en forma *jerárquica*, agrupando definiciones abstractas semejantes en niveles que vayan desde los específicos hasta los más generales cuando se asciende por la jerarquía. Una aplicación común de las jerarquías se encuentra en un organigrama, donde cada enlace que va hacia arriba se puede leer como "**es un(a)**", como en "una cabaña es una casa es una construcción" (figura abajo). De igual forma, esta clase de jerarquías es útil en el diseño de programas porque agrupa la funcionalidad común en el nivel más general, y considera que el comportamiento especializado es una extensión de la regla general.



Para evitar programación redundante, el paradigma orientado a objetos permite tener una estructura organizativa modular y jerárquica para reusar el código, mediante una técnica llamada *herencia*. Esa técnica permite diseñar las clases genéricas que se especializan en clases más particulares, para que éstas reusen el código o programa de la clase genérica. Por ejemplo, la clase Number de Java se especializa en las clases Float, Integer y Long. La clase general, que también se llama *clase base* o *superclase*, puede definir variables de instancia y métodos "genéricos" que se apliquen en una multitud de situaciones. Una clase que *especializa*, *extiende* o *hereda* de una superclase no necesita dar más implementaciones para los métodos generales porque los hereda. Sólo debe definir los métodos que sean especializados para esta *subclase* particular (que también se llama *clase derivada*).

**Ejemplo** Se tiene una clase S que define objetos con un campo, x, y tres métodos, a(), b() y c() (Para este ejemplo, el tipo particular de x y la funcionalidad de a(), b() y c() no interesan). Supóngase que se debe definir una clase T que extienda a S e incluya un campo adicional y, y dos métodos, d() y e(). Esto implicaría que los objetos de la clase T tengan dos campos, x y y, y cinco métodos, a(), b(), c(), d() y e(). La clase T hereda la variable de instancia x y los métodos a(), b() y c() de S.

A continuación, se ilustran las relaciones entre la clase S y la clase T en un diagrama de herencia de clase, en la figura siguiente. Cada cuadro del diagrama representa una clase, y su nombre, campos (o variables modelo) y sus métodos se incluyen como subrectángulos. Una flecha de un cuadro T a otro cuadro S indica que la clase T **extiende a (o hereda de)** la clase S.



### Creación y referenciación de objetos

Cuando se crea un objeto **obj**, se asigna memoria a sus campos de datos y los mismos campos se inicializan con valores específicos iniciales. En el caso normal, se asocia el nuevo objeto **obj** con una variable, lo cual sirve como "enlace" al objeto **obj**, y se dice que es *referencia* o se *refiere* a **obj**. Cuando se desea dar acceso al objeto **obj** (con objeto de obtener uno de sus campos o ejecutar sus métodos), se puede pedir la ejecución de uno de los métodos *o* (definido por la clase a la que pertenece **obj**), o también se puede buscar uno de los campos de **obj**. En realidad, la forma principal en que un objeto **p** interacciona con otro objeto **obj** es que **p** mande un

"mensaje" a **obj** que invoque a uno de los métodos de **obj**; por ejemplo, para que **obj** imprima una descripción de sí mismo, para que **obj** se convierta en una string, o para que **obj** calcule el valor de uno de sus campos de datos. La forma secundaria en que *p* puede interactuar con **obj** es que *p* entre en forma directa a uno de los campos de **obj**, pero sólo si **obj** ha dado permiso de hacerlo a otros objetos, como *p*. Por ejemplo, una instancia de la clase Integer de Java guarda un entero como variable de instancia, y proporciona varias operaciones para llegar o "acceder" a estos datos, incluyendo métodos para convertirla en otros tipos de números, convertirla en una string de dígitos y convertir strings de dígitos en un número. No permite el acceso directo a su variable modelo, porque esos detalles están ocultos.

## Despacho dinámico

Cuando un programa desea invocar determinado método *a()* de cierto objeto *obj*, manda un mensaje a **obj**, que indica con la sintaxis de punto-operador, en la forma "**obj.a()**". En la versión compilada de este programa, el código que corresponde a esta invocación dirige al ambiente de ejecución para que examine la clase *T* de **obj**, para determinar si la clase *T* soporta un método *a()*, y si lo hace, que lo ejecute. Si *T* no define a un método *a()*, el ambiente de ejecución examina la superclase *S* de *T*. Si *S* define a *a()*, se ejecuta el método. Si *S* no lo define, el ambiente de ejecución repite la búsqueda en la superclase de *S*. Esta búsqueda continúa ascendiendo la jerarquía de clases, hasta que encuentra un método *a()*, que se ejecuta, o bien hasta que llegue a una clase máxima (por ejemplo, la clase Object en Java) y no haya encontrado un método *a()*, en cuyo caso se genera un error en tiempo de ejecución. El algoritmo que procesa el mensaje **obj.a()** para encontrar el método específico por invocar, se llama de **despacho dinámico** (o **enlace dinámico**). Este algoritmo de despacho dinámico, para encontrar un método *a()* con qué procesar el mensaje "**obj.a()**", es un mecanismo efectivo para localizar los programas que se reúsan. También proporciona una poderosa técnica a la programación orientada a objetos: el *polimorfismo*.

## Polimorfismo

En forma literal, "polimorfismo" significa "muchas formas". En el contexto del diseño orientado a objetos indica la capacidad que tiene una variable objeto para tomar distintas formas. Los lenguajes orientados a objetos, Java, manejan por ejemplo los objetos usando variables de referencia. La variable de referencia **obj** debe definir qué clase de objetos puede manejar o referirse, en términos de alguna clase *S*. Pero eso implica que **obj** también se puede referir a cualquier objeto que pertenezca a una clase *T* que extienda a *S*. Ahora bien, considérese qué sucede si *S* define un método *a()* y *T* también define a un método *a()*. El algoritmo de despacho dinámico para invocar los métodos siempre comienza su búsqueda desde la clase más restrictiva que se aplique. Cuando **obj** se refiere a un objeto de la clase *T*, entonces usará el método *a()* de *T* cuando se pida **obj.a()**, y no el de *S*. En este caso, se dice que *T* se *sobrepone* al método *a()* de *S*. También, cuando **obj** se refiere a un objeto de la clase *S* (que no sea también un objeto de *T*), ejecutará el método *a()* de *S* cuando se le pida **obj.a()**. Un polimorfismo como éste es útil porque quien llama a **obj.a()** no necesita conocer si el objeto *o* se refiere a una instancia de *T* o de *S*, para que el método *a()* se ejecute en forma correcta. Así, la variable objeto **obj** puede ser *polimórfica*, es decir, puede asumir muchas formas, dependiendo de la clase específica de objetos a la que se refiere. Esta clase de funcionalidad permite que una clase especializada *T* extienda una clase *S*, heredando los métodos "genéricos" de *S*, y que redefina a otros métodos de *S* para tener en cuenta las propiedades específicas de los objetos de *T*.

Algunos lenguajes orientados a objetos, como Java, también permiten un polimorfismo "desaguado", lo que con más propiedad se llama *sobrecarga* de método. La sobrecarga se presenta cuando una sola clase *T* tiene varios métodos con el mismo nombre, siempre que cada uno tenga una *firma* distinta. La firma de un método es una combinación de su nombre, la clase y la cantidad de argumentos que pasan a ella. Así, aun cuando en una clase pueda haber varios métodos con el mismo nombre, un compilador los puede distinguir si tienen firmas distintas, esto es, si en realidad son distintos. En lenguajes que tienen un método para sobrecarga, el ambiente de ejecución determina qué método se debe invocar en cierta llamada de método, buscando en ascenso por la jerarquía de clases hasta encontrar el primer método que tenga una firma que coincida con el método que se está invocando. Por ejemplo, supóngase que una clase *T*, que define a un



método `a()`, extiende a una clase `U` que define a un método `a(x, y)`. Si un objeto `o` de la clase `T` recibe el mensaje "`obj.a(x, y)`", entonces la que se invoca es la versión en `U` del método `a` (con los dos parámetros, `x` y `y`). De este modo, el polimorfismo verdadero sólo se aplica a métodos que tienen la misma firma, pero que se definen en clases distintas.

La herencia, el polimorfismo y la sobrecarga de métodos permiten el **desarrollo de programas reusables**. Se pueden definir clases que hereden las variables modelo y los métodos genéricos, para entonces definir variables de instancia y métodos más específicos que manejen aspectos especiales de la nueva clase.

## Uso de la herencia en Java

Hay dos formas principales de usar las clases de herencia en Java: la **especialización y la extensión**. Al usar la primera se especializa una clase general en subclases particulares. Estas subclases poseen una relación "es un" con su superclase. Entonces, las subclases heredan todos los métodos de la superclase. Para cada método heredado, si ese método funciona bien, independientemente de si funciona para una especialización, no se necesita trabajar más. Por otro lado, si un método general de la superclase no trabaja bien en la subclase, se debe sobreponer el método para tener la funcionalidad correcta de la subclase. Por ejemplo, se podría tener una clase general, `Perro`, que tenga un método `beber` y un método `oler`. Es probable que para especializar esta clase a una clase `PerrodeCaza` no se requiera sobreponer el método `beber`, porque todos los perros beben casi en la misma forma. Pero, podría necesitarse sobreponer el método `oler`, porque un perro de caza tiene un sentido del olfato mucho más sensible que un perro "genérico". Es así como la clase `PerrodeCaza` especializa a los métodos de su superclase `Perro`.

Por otra parte, al usar la extensión, se aprovecha la herencia para reusar el código escrito para métodos de la superclase pero a continuación se agregan métodos nuevos que no hay en la superclase, para extender la funcionalidad de ésta. Por ejemplo, regresando a la clase `Perro`, se podría crear una subclase `PerroOvejero` que herede todos los métodos genéricos de la clase `Perro`, pero que agregue un método nuevo, `rebaño`, porque los perros ovejeros tienen un instinto de manejo de rebaño que no tienen los perros genéricos. Al agregar el nuevo método se está extendiendo la funcionalidad de un perro genérico.

### Herencia de clase en Java

En Java, cada clase puede extender exactamente a una clase. Aun si en una definición de clase no se usa en forma explícita la cláusula **`extends`**, hereda de exactamente otra clase, que en este caso es `java.lang.Object`. Debido a esta propiedad, se dice que Java sólo permite *herencia simple* entre las clases.

### Tipos de sobreposición de un método

Dentro de la declaración de una nueva clase, se usan en Java dos métodos de sobreposición: *refinamiento* y *reemplazo*. En la sobreposición por reemplazo, un método reemplaza en forma total al método de la superclase a la que se sobrepone (como en el método `oler` de `PerrodeCaza` mencionado arriba). En Java, todos los métodos regulares de una clase usan este comportamiento de sobreposición.

Sin embargo, en el método de refinamiento, un método no se sobrepone al método de la superclase correspondiente, sino que agrega un código a la superclase. En Java, todos los métodos constructores usan la sobreposición por refinamiento, y a este esquema se le llama *encadenamiento de constructor*. Es decir, un constructor comienza a ejecutarse llamando a un constructor de la superclase. Esto se puede hacer en forma explícita o implícita. Para llamar en forma explícita a un constructor de la superclase se usará la palabra clave `super`, para indicar la superclase. (Por ejemplo, `superO` llama al constructor de la superclase sin argumentos.) Sin embargo, si no se hace llamada explícita en el cuerpo de un constructor, el compilador inserta en forma automática, como primer renglón del constructor, una llamada al método `superO`. (Nótese que hay una excepción a esta regla general, que se describirá en la siguiente sección.) En resumen, los constructores en Java usan el método de refinamiento por sobreposición, mientras que los

métodos regulares usan reemplazo.

## La palabra clave this

A veces, en una clase Java conviene designar el ejemplo actual de esa clase en la que está operando determinado método. Java proporciona esa referencia, que se llama `this`. Es útil usar esta construcción, por ejemplo, para cuando se desee designar un campo dentro del objeto actual, que tenga un conflicto de nombre con una variable definida en el bloque actual, como se muestra en el siguiente programa:

```
public class ThisTester {
    public int dog = 2;    // variable de instancia
    ThisTesterO { /* constructor nulo */}           // constructor

    public void clobberO {
        double dog=5.0;
        this.dog = (int) dog;                       // ¡dos perros distintos!
    }

    public static void main(String args[ ]) {
        ThisTester t = new ThisTesterO;
        System.out.println(" El campo dog = " + t. perro);
        t.clobberO;
        System.out.println("Después de castigarlo, dog = " + t.perro);
    }
}
```

Cuando se ejecuta este programa, imprime lo siguiente:

```
El campo perro = 2
Después de castigarlo, perro = 5
```

## Ejemplos de la herencia en Java

Para concretar más algunas de las nociones anteriores sobre la herencia y el polimorfismo, se describirán algunos ejemplos sencillos en Java.

En particular, se examinarán ejemplos de varias clases para recorrer e imprimir series numéricas. Una serie numérica es una sucesión de números en donde el valor de cada uno depende de uno o más de los valores anteriores. Por ejemplo, una serie aritmética determina el número siguiente sumando; así, la serie geométrica determina el número siguiente multiplicando. En cualquier caso, una serie requiere una forma de definir su primer valor y también una forma de identificar el valor actual.

Se comenzará definiendo una clase, *Progression*, que se ve en mas adelante, que define los campos "genéricos" y los métodos de una serie numérica. En especial, define los siguientes campos de dos enteros largos:

```
first: el primer valor de la serie;
cur: el valor actual de la serie;
```

y los tres métodos siguientes:

getFirstValue(): Restablecer la serie al primer valor, y mostrar ese valor.

getNextValueO: Mover la serie al siguiente valor, y mostrar ese valor.

printProgression(n): Reiniciar la serie e imprimir sus primeros  $n$  valores.

Se dice que el método printProgression no tiene salida, o resultado, porque no retorna valor alguno, mientras que los métodos getFirstValue y getNextValue si lo hacen.

La clase Progression también incluye un método ProgressionO, que es un método *constructor*. Recuérdese que los métodos de constructor establecen todas las variables de instancia en el momento en que se crea un objeto de esta clase. La clase Progression pretende ser superclase genérica, de la cual se heredan clases especializadas, por lo que este constructor es el programa que se incluirá en los constructores para cada caso que extienda la clase Progression.

```
public class Progression {
    protected long first; // Primer valor de la serie
    protected long cur; // Valor actual de la serie.
    Progression() { // Constructor predeterminado
        cur = first = 0;
    }
    protected long getFirstValue() { // reinicializar y regresar el primer valor
        cur = first;
        return cur;
    }
    protected long getNextValue() { // Retorna corriente previamente avanzado
        return ++cur;
    }
    public void printProgression(int n) { // Imprime n primeros valores
        System.out.print(getFirstValue());
        for (int i = 2; i <= n; i++){
            System.out.print(" " + getNextValue());
        }
    }
}
```

## Una clase de serie aritmética

A continuación se describirá la clase ArithProgression.

Esta clase define a una serie en la que cada valor se determina sumando inc, un incremento fijo, al valor anterior. Esto es, ArithProgression define a una serie aritmética.

La clase ArithProgression hereda los campos first y cur, y los métodos getFirstValueO e printProgression(n) de la clase Progression. Se agrega un campo nuevo, inc para guardar el incremento, y dos constructores para establecer el incremento. Por último, sobrepone el método GetNextValuerO para ajustarse a la forma de obtención de un siguiente valor para la serie aritmética.

En este caso lo que funciona es el **polimorfismo**. Cuando una referencia Progression apunta a un objeto ArithProgression, los objetos usan los métodos getFirstValueO y getNextValueO de ArithProgression. Este polimorfismo también es válido dentro de la versión heredada de printProgression(n) porque aquí las llamadas de los métodos GetFirstValueO y getNextValueO son implícitamente para el objeto "actual" (que en Java se llama this) y en este caso serán de la clase ArithProgression.

En la definición de la clase ArithProgression se han agregado dos métodos constructores, un método predeterminado (por omisión) que no tiene parámetros, y un método paramétrico, que toma un parámetro entero como incremento para la progresión. El constructor predeterminado en realidad llama al paramétrico, al usar la palabra clave **this** y pasar a 1 como el valor del parámetro de incremento. Estos dos constructores ilustran la sobrecarga del método, en la que un nombre de método puede tener varias versiones dentro de la misma clase porque en realidad un método se especifica por su nombre, la clase de objeto que llama y los tipos de argumentos que se le pasan a él: su firma. En este caso, la sobrecarga es para métodos constructores (un constructor predeterminado y uno paramétrico).

La llamada this( 1) al constructor paramétrico como primera declaración del constructor predeterminado activa una excepción a la regla general de encadenamiento de constructor. Es decir, siempre que la primera

declaración de un constructor C' llama a otro constructor C" de la misma clase usando la referencia **this**, el constructor de superclase no es llamado implícitamente para C'. Sin embargo, nótese que a la larga un constructor de superclase será llamado a lo largo de la cadena, sea en forma explícita o implícita. En particular, para la clase ArithProgression, el constructor predeterminado de la superclase (Progression) es llamado en forma implícita como primera declaración del constructor paramétrico de ArithProgression.

```
import Progression;
class ArithProgression extends Progression {
    protected long inc;    // incremento
    ArithProgression() { // Constructor predeterminado, llama al parametrico
        this(1);
    }
    ArithProgression(long increment) { //Constructor paramétrico que proporciona el incremento.
        super();
        inc = increment;
    }
    protected long getNextValue(){    // metodo sobrescrito
        cur += inc;    // Avanza la serie sumando el incremento al valor actual
        return cur;    // regresar siguiente valor de la serie

    // Hereda getFirsValue();
    // Hereda printProgression(..)
}
```

### Una clase de serie geométrica

Se definirá la clase GeomProgression que se muestra a continuación, que implementa una serie geométrica, determinada multiplicando el valor anterior por una base b. Una serie geométrica es como una serie genérica, excepto la forma en que se determina el siguiente valor. En consecuencia, se declara a GeomProgression como subclase de la clase Progression. Como en la clase ArithProgression, la clase GeomProgression hereda de la clase Progression los campos first y cur, y los métodos GetFirstValue y printProgression.

```
import Progresión;
class GeomProgression extends Progression {
    // Hereda las variables first y cur
    GeomProgression(){    // Constructor prdeterminado, llama al parametrico
        this(2);
    }
    GeomProgression(long base){    // Constructor parametrico,
        super();
        first = base;    // Define el primer termino en funcion de base
        cur = first;    // y el corriente de igual forma
    }
    protected long getNextValue(){
        cur *= first;    // Avanza la serie multiplicando la base por el valor actual
        return cur;    // regresar el siguiente valor de la serie
    }
    // Hereda getFirstValue()
    // Hereda printProgression(int).
}
```

### Clase de la serie de Fibonacci

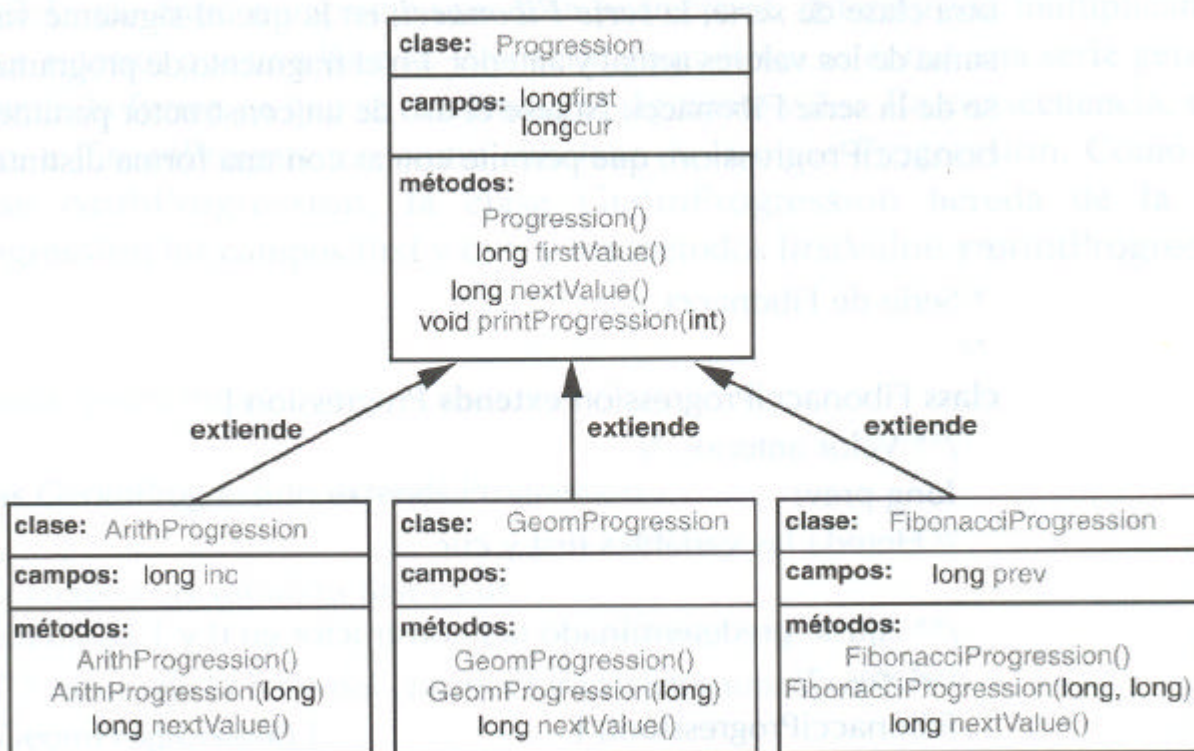
Como un ejemplo más se definirá una clase FibonacciProgression que representa otra clase de serie, la *serie Fibonacci*, en la que el siguiente valor se define como la suma de los valores actual y anterior. Nótese el uso de un constructor parametrizado en la clase FibonacciProgression, que permite contar con una forma distinta de iniciar la serie.

```
import Progression;
class FibonacciProgression extends Progression{
    long prev;    // Valor anterior
```

```

// Hereda las variables first y cur
FibonacciProgression() { // Constructor predeterminado
    this(0, 1); // llama al paramétrico
}
FibonacciProgression(long value1, long value2) { // Constructor paramétrico
    super();
    first = value1; // Definimos primer valor
    prev = value2 - value1; // valor ficticio que antecede al primero
}
protected long getNextValue() { // Avanza la serie
    long temp = prev;
    prev = cur;
    cur += temp; // sumando el valor anterior al valor actual
    return cur; // retorna el valor actual
}
// Hereda getFirstValue()
// Hereda printProgression(int).
}

```



Para completar el ejemplo, se definirá una clase `Tester`, que ejecuta una prueba sencilla de cada una de las tres clases. En esta clase, la variable `prog` es polimórfica durante la ejecución del método `main`, porque se refiere a objetos de la clase `ArithProgression`, `GeomProgression` y `FibonacciProgression` en turno.

El ejemplo presentado en esta sección es pequeño, pero proporciona una ilustración sencilla de la herencia en Java. Sin embargo, la clase `Progression`, sus subclases y el programa probador tienen varias limitaciones que no son aparentes de inmediato. Uno de los problemas es que las series geométrica y de Fibonacci crecen con rapidez, y no está previsto el manejo del desbordamiento inevitable de los enteros largos que se manejan. Por ejemplo, como  $3^{40} > 2^{63}$ , ( $**$  significa *potenciación*, lea *3 elevado al exponente 40*) una serie geométrica con la base  $b = 3$  tendrá desbordamiento de entero largo después de 40 iteraciones. En forma parecida, el 94avo término de la serie de Fibonacci es mayor que  $2^{63}$ , por lo que esta serie desbordará después de 94 iteraciones. Otro problema es que podrían no permitirse valores iniciales arbitrarios para una serie de Fibonacci. Por ejemplo, ¿se permite una serie Fibonacci que inicie con 0 y -1? Para manejar errores de entrada o condiciones de error que se presenten durante la ejecución de un programa de Java se requiere tener algún mecanismo que lo haga. Este tema se llama Tratamiento de Excepciones y lo veremos más adelante.

```

/** Programa de prueba para las clases de series */

```

```

import ArithProgression;
import GeomProgression;
import FibonacciProgression;
class Tester{
    public static void main(String[] args){ Progression prog;
        System.out.println("\n\nSerie aritmética con incremento predeterminado: ");
        prog = new ArithProgression();
        prog.printProgression(10);
        System.out.println("\n\nSerie aritmética con incremento 5: ");
        prog = new ArithProgression(5);
        prog.printProgression(10);
        System.out.println("\n\nSerie geométrica con la base predeterminada: ");
        prog = new GeomProgression();
        prog.printProgression(10);
        System.out.println("\n\nSerie geométrica con base 3: \n");
        prog = new GeomProgression(3);
        prog.printProgression(10);
        System.out.println("\n\nSerie de Fibonacci con valores iniciales predeterminados: ");
        prog = new FibonacciProgression();
        prog.printProgression(10);
        System.out.println("\n\nSerie Fibonacci con valores iniciales 4 y 6: ");
        prog = new FibonacciProgression(4,6);
        prog.printProgression(10);
    }
}

```

```

Serie aritmética con incremento predeterminado:
0 1 2 3 4 5 6 7 8 9

Serie aritmética con incremento 5:
0 5 10 15 20 25 30 35 40 45

Serie geométrica con la base predeterminada:
2 4 8 16 32 64 128 256 512 1024

Serie geométrica con base 3:
3 9 27 81 243 729 2187 6561 19683 59049

Serie de Fibonacci con valores iniciales
predeterminados:
0 1 1 2 3 5 8 13 21 34

```

## 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 subdirectorío 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 subdirectorío. 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 asterisco (\*) 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 implícitamente 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

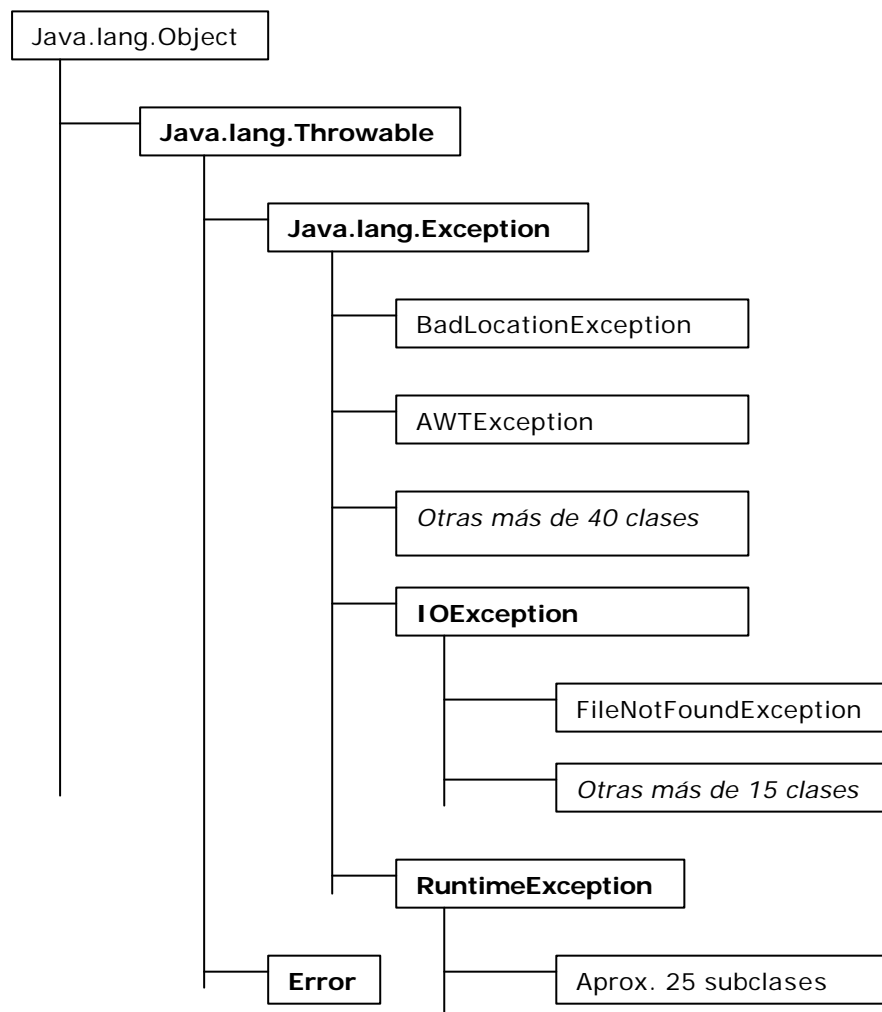
- Las clases pueden organizarse en **paquetes**
- Un paquete se identifica con la cláusula **package algún paquete**, al principio de un archivo fuente.
- Solo una declaración **package** por archivo.
- Varios archivos pueden pertenecer al mismo **paquete**
- Un archivo sin declaración **package** pertenece 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
- Al compilar o ejecutar, las clases de biblioteca proporcionadas por Java (Java, Javac, JDK) se cargan automáticamente, otras deben especificarse en CLASS PATH.  
`SET CLASSPATH= C:\JAVA;` //Hasta el directorio padre de paquetes
- Para incorporarlas en mi programa, puedo  
`import mis Paquetes.*` // Todas las clases que allí 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

## 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 describir 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 cuarto 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, 0,10,12};

    public static void excep() {
        int i;
        float coc;
        System.out.println("\nExcepciones, ej 01");
        for(i = 0; i < deno.length; i++){
            coc = (float)nume[i]/deno[i];
            System.out.println("Cociente "+i+" vale "+coc);
        }
    }

    public static void main(String args[]){
        excep();
    }
}
```

```
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"
~      ~      ~
```

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 límites del vector). Como nosotros no capturamos ni tratamos esta excepción, ella llega a la Máquina 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 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 que deje esto bien explicitado.

```
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 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{
    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");
    }
}
```

Ejecución →

```
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
```

```

    public static void main(String args[]){
        excep();
    }
}

```

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.

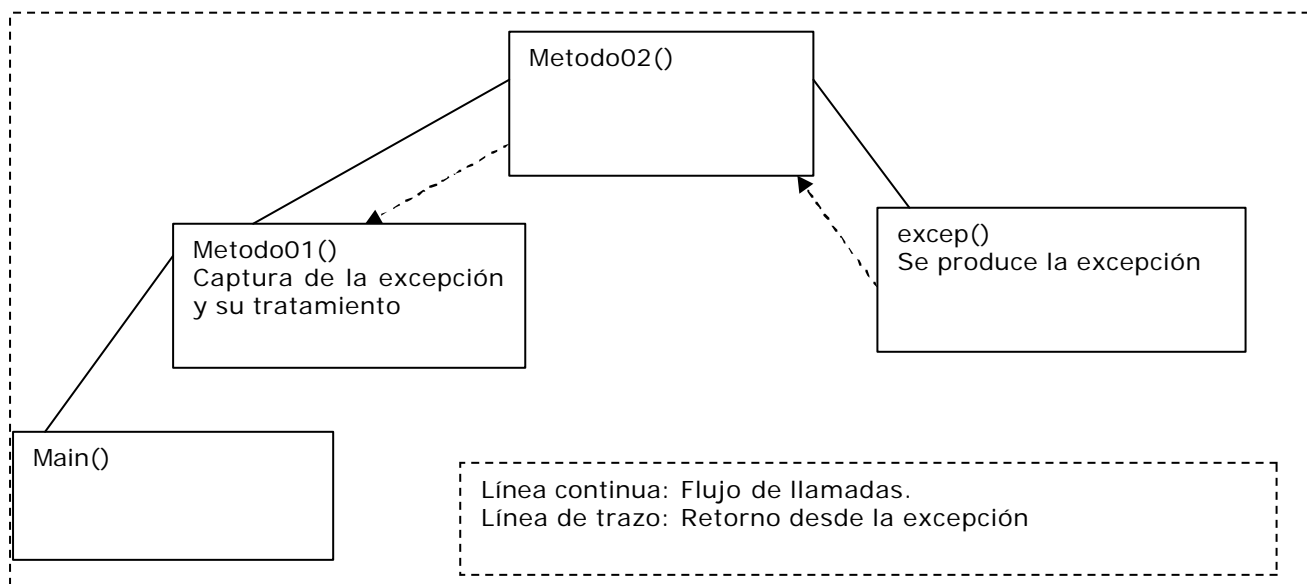
El ambiente Java comienza ejecutando el **try{bloque\_de\_instrucciones\_}**. Si esa ejecución **no genera excepciones**, el flujo del control continúa con el primer paso de programa después del último renglón de todo el bloque **try-catch**, a menos que ese bloque incluya un bloque finally opcional. 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 llamada a
            Metodo02();
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Capturada en Metodo01()!!!");
        }
    }
}

```

```

Entrada a Metodo01()
Entrada a Metodo02()
Entrada a excep()
Cociente 0 vale 5.0
Cociente 1 vale 5.0

```

```

    }
    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();
}
}

```

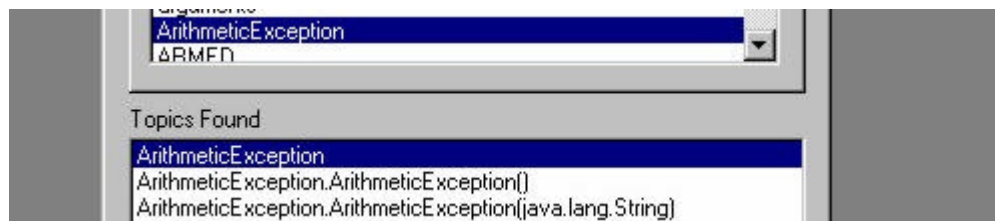
La excepción se produce, como antes, en el método `excep()`, pero ahora no se la captura, solo declaramos que se dispara, mediante la **cláusula throws**. El tratamiento de excepciones se realiza en `metodo01()`. Allí tenemos el bloque `try` y la cláusula de captura. En el seguimiento de la ejecución 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 mayoría 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 muestra el esquema hereditario al cual la clase `ArithmeticException` está subordinada. Perteneció 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 nuestra
            }
            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 !!!

excepción

Observamos que:

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

## BÚSQUEDA EN 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 items conteniendo 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 el obj. invocante que el 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
        item.valor = this.valor; // objeto parametro
        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 talla; // 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
        talla=tam; // inicializamos talla (tamaño)
        item = new Item[talla]; // Generamos el array
        for(int i=0;i<talla;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':{
        auxCod = talla - i;
        break;
    }
    case 'R':{
        auxCod = (int)(talla*Math.random());
    }
}
item[i] = new Item();
item[i].setCodigo(auxCod);
item[i].setValor((float)(talla*Math.random()));
}
System.out.print(this);
}

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

```

## BUSQUEDA 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 probable que el elemento a buscar pueda ser el primero el último o cualquier otro. De promedio, al menos el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del array. Este método de búsqueda, cuyo algoritmo es sencillo, es adecuado con arrays pequeños o no ordenados.

En la implementación que sigue heredamos de **class ArrItems**. Hacemos esto porque el array en el que buscaremos algunas claves **"es un"** array de items, y entonces nos conviene implementar esta relación mediante herencia. Haciendo esto el constructor de Busqueda() invoca al constructor de la clase base, quien hace todo el trabajo. También aprovechamos el método toString().

```

import ArrItems;
class Busqueda extends ArrItems{ // Busqueda secuencial en un array de items
    int posic; // Posicion del ultimo encuentro
    int clav;

    public Busqueda(int cant, char tipo){ // Un constructor
        super(cant,tipo); // que invoca otro
        posic = talla+1; // posición fuera del array
        clav = 0;
    }

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

    protected int getClav(){
        return clav;
    }

    protected int leerClav(){
        System.out.print("Que clave? (999: fin) ");
        clav = In.readInt();
        return clav;
    }
}

```



```

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

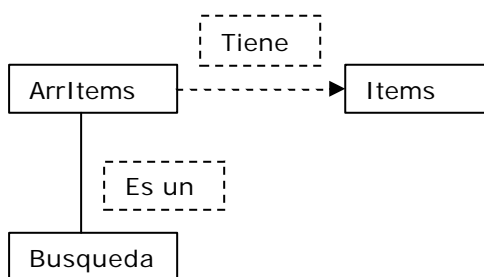
protected void demoBusqueda(){
    int cant;
    while (leerClav() != 999){
        cant =cuantas(clav);
        if(existe(clav))
            System.out.println("Código " + clav + ", existe " + cant+" veces");
        else
            System.out.println("Codigo " + clav + " inexistente");
    }
    System.out.println("Terminamos !!!\n");
}

};

import Busqueda;
class pruBusSeq{
    public static void main(String args[]){
        Busqueda busq = new Busqueda(100,'R');
        // Generamos un array de 100 objetos item random
        busq.demoBusqueda();
    }
}

```

El esquema de clases que estamos aplicando es:



## BUSQUEDA 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.

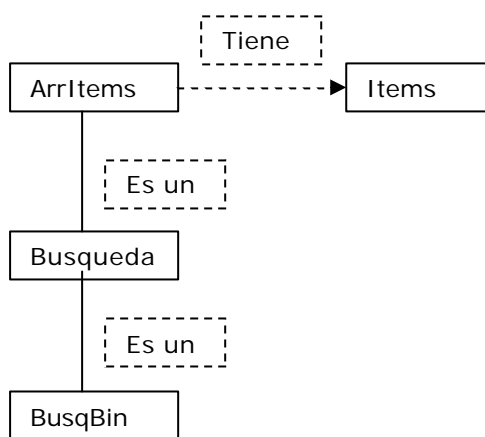
y este razonamiento se aplicará las veces necesarias hasta encontrar igual o definir que la clave no existe.

Como interrelacionamos la búsqueda binaria en el esquema de clases ya existente? Es bastante razonable modelizar pensando que una búsqueda binaria **es un** otro tipo de búsqueda que la secuencial que ya tenemos. Eso haremos. Una modelización posiblemente mejor conceptualmente sería partir de una clase búsqueda (Genérica) y de allí extender la secuencial y la binaria. Bueno, siempre existen soluciones “superadoras” .

```
import Busqueda;
class BusqBin extends Busqueda{ // Busqueda Binaria en un array de items
    public BusqBin(int cant, char tipo){ // Un constructor
        super(cant,tipo); // que invoca otro
    }
    protected boolean existe(int clave){ // Existe la clave parámetro ?
        boolean exist = false;
        int alt=talle-1,baj=0;
        int indCent, valCent;
        while (baj <= alt){
            indCent = (baj + alt)/2; // índice de elemento central
            valCent = item[indCent].getCodigo(); // valor del 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; // elemento 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<talle;i++){ // para arriba
                if (clave==item[i].getCodigo())
                    igual++;
                else break;
            }
            for(int i=posic-1;i>0;i--){ // tambien para abajo
                if (clave==item[i].getCodigo())
                    igual++;
                else break;
            }
            // if (exist= existe(clave))
            return igual;
        }
    };
};

import BusqBin;
class pruBusBin{
    public static void main(String args[]){
        BusqBin busq = new BusqBin(100,'A');
        // Generamos un array de 100 objetos item ascendente
        busq.demoBusqueda();
    }
};
```

```
Primeros 10 de 100
elementos Item
0 - 94.08064
1 - 70.68094
2 - 40.423695
3 - 76.080284
4 - 84.71842
5 - 33.304916
6 - 9.811888
7 - 62.708015
8 - 66.173584
9 - 46.92478
Que clave? (999: fin) 11
Código 11, existe 1 veces
Que clave? (999: fin) 25
Código 25, existe 1 veces
Que clave? (999: fin)
Código 0, existe 1 veces
Que clave? (999: fin) 101
Codigo 101 inexistente
```



## Comparación de la búsqueda binaria y secuencial

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 del **orden de  $n$**  y se expresa  $O(n)$

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. La función matemática que nos da el número máximo de comparaciones (peor caso) resultante de esta mecánica de sucesivas particiones es  $\log_2 n$ . Entonces el tiempo de búsqueda es del **orden  $\log_2 n$**  y se expresa  $O(\log_2 n)$ .

Números de comparaciones considerando el peor caso

Tamaño array	Búsqueda binaria	Búsqueda secuencial
1	1	1
10	4	10
1.000	11	1.000
5.000	14	5.000
100.000	18	100.000
1.000.000	21	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 básicos y mejorados.

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 mas 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 burbujan (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.

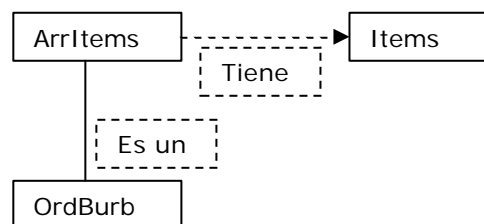
Como en lo visto para búsqueda, por las mismas razones, nos conviene definir **class OrdBurb** heredando de **class ArrItems**. Para trazar como va quedando el array tras sucesivas pasadas, vamos a incluir un método mostArr() que muestre en una línea los 10 primeros elementos del array. Y a este método lo usaremos opcionalmente dentro del **burbuja**, para ir trazando el avance del ordenamiento.

```
import ArrItems;
public class OrdBurb extends ArrItems{    // Ordenamiento de un array de items
                                           // mediante el método Burbuja

    public OrdBurb(int cant, char tipo){    // Un constructor
        super(cant,tipo);
        System.out.println("\nPasadas de ordenamiento");
    }

    public String toString(){              // Mostrando los valores del array
        String aux = "";
        int cuant=(talle < 10 ? talle : 10); // Lo que sea menor
        for (int i=0;i<cuant;i++)
            aux += item[i].getCodigo()+" ";
        return aux;
    }

    void ordenar(boolean trace){ // El método de ordenamiento
        int i,j;
        for (i = 1; i<talle; i++){ // Indicando talle - 1 pasadas
            for (j = talle-1; j>=i; j--) // realizando la pasada
                if (item[j].esMayor(item[j-1])) // Si invocante es mayor que parámetro
                    item[j].intercambio(item[j-1]); // intercambio
            if (trace)
                System.out.println(this); // Mostramos el array despues de la pasada
        }
    }
}
```



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

```
Ordenamiento metodo Burbuja
1, 0, 2, 8, 9, 1, 2, 7, 5,
8,
Pasadas de ordenamiento
9, 1, 0, 2, 8, 8, 1, 2, 7,
5,
9, 8, 1, 0, 2, 8, 7, 1, 2,
5,
9, 8, 8, 1, 0, 2, 7, 5, 1,
2.
```

```
Ordenamiento metodo Burbuja
2, 1, 7, 3, 1, 9, 5, 2, 0,
8,
Pasadas de ordenamiento
9, 2, 1, 7, 3, 1, 8, 5, 2,
0,
9, 8, 2, 1, 7, 3, 1, 5, 2,
0,
9, 8, 7, 2, 1, 5, 3, 1, 2,
0,
```

En la primera ejecución tenemos una pasada innecesaria. En la segunda, tres. Y en un array grande podrían ser muchas. Es necesario mejorar esto.

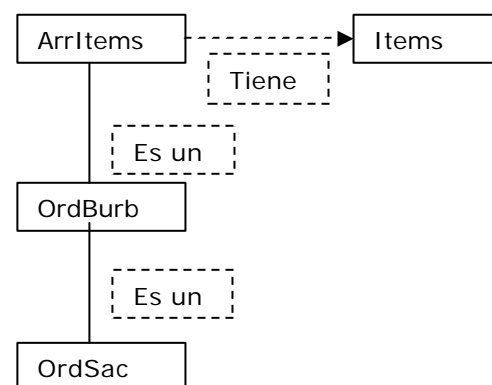
## Ordenamiento por Método Sacudida.

El Método "sacudida" que veremos ahora incorpora un par de ventajas:

- 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.

```
public class OrdSac extends OrdBurb{    // Ordenamiento de un array de items
                                        // mediante el método Sacudida
    public OrdSac(int cant, char tipo){ // Un constructor
        super(cant,tipo);
    }
    void ordenar(boolean trace){
        int j,k = talle-1, iz = 1, de = talle-1; Item aux;
        do {                            // Ciclo de control de pasadas
            for (j = de; j >= iz; j--) // Pasada descendente
                if (item[j].esMayor(item[j-1])){
                    item[j].intercambio(item[j-1]);
                    k = j; // Guardamos el lugar del último intercambio
                }
            iz = k+1;
            if (trace)
                System.out.println(this); // Mostramos el array despues de la pasada
            for (j = iz; j <= de; j++) // Pasada ascendente
                if (item[j].esMayor(item[j-1])){
                    item[j].intercambio(item[j-1]);
                    k = j; // Guardamos el lugar del último intercambio
                }
            de = k-1;
            if (trace)
                System.out.println(this); // Mostramos el array despues de la pasada
        } while (iz <= de);
    } // void ordenar
} // public class OrdSac
```

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



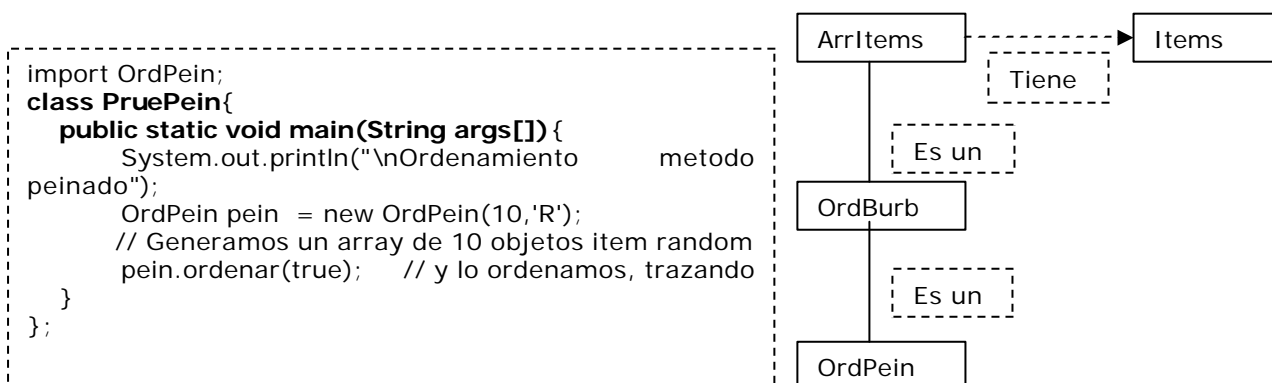
Ordenamiento metodo Sacudida  
 5, 5, 2, 9, 2, 3, 8, 9, 1, 1,  
 Pasadas de ordenamiento  
 9, 5, 5, 2, 9, 2, 3, 8, 1, 1,  
 9, 5, 5, 9, 2, 3, 8, 2, 1, 1,  
 9, 9, 5, 5, 8, 2, 3, 2, 1, 1,  
 9, 9, 5, 8, 5, 3, 2, 2, 1, 1,  
 9, 9, 8, 5, 5, 3, 2, 2, 1, 1,  
 9 9 8 5 5 3 2 2 1 1

Ordenamiento metodo Sacudida  
 5, 6, 9, 9, 3, 0, 2, 8, 1, 1,  
 Pasadas de ordenamiento  
 9, 5, 6, 9, 8, 3, 0, 2, 1, 1,  
 9, 6, 9, 8, 5, 3, 2, 1, 1, 0,  
 9, 9, 6, 8, 5, 3, 2, 1, 1, 0,  
 9, 9, 8, 6, 5, 3, 2, 1, 1, 0,  
 Process Exit...

## 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. Codificación y demo a continuación.

```
import OrdBurb;
public class OrdPein extends OrdBurb{    // Ordenamiento de un array de items
                                         // mediante el método Peinado
    public OrdPein(int cant, char tipo){    // Un constructor
        super(cant,tipo);
    }
    void ordenar(boolean trace){          //programa de ordenamiento por peinado
        int i,paso = talla;
        boolean cambio;
        do {
            paso = (int)((float) paso/1.3);
            paso = paso>1 ? paso : 1;
            cambio = false;
            for (i = 0; i<talla-paso; i++)
                if (item[i].esMayor(item[i+paso])){
                    item[i].intercambio(item[i+paso]);
                    cambio = true;
                }
            if (trace)
                System.out.println(this);
        } while (cambio || paso>1);
    }
    // void ordenar
    // public class OrdPein
}
```



```
Ordenamiento metodo Peinado
4, 5, 9, 1, 3, 7, 7, 3, 0, 6,
Pasadas de ordenamiento
3, 0, 6, 1, 3, 7, 7, 4, 5, 9,
3, 0, 4, 1, 3, 7, 7, 6, 5, 9,
1, 0, 4, 3, 3, 5, 7, 6, 7, 9,
1, 0, 3, 3, 4, 5, 7, 6, 7, 9,
0, 1, 3, 3, 4, 5, 6, 7, 7, 9,
0, 1, 3, 3, 4, 5, 6, 7, 7, 9,
Process Exit...
```

```
Ordenamiento metodo Peinado
4, 1, 4, 5, 5, 7, 8, 2, 3, 3,
Pasadas de ordenamiento
2, 1, 3, 5, 5, 7, 8, 4, 3, 4,
2, 1, 3, 3, 4, 7, 8, 4, 5, 5,
2, 1, 3, 3, 4, 5, 5, 4, 7, 8,
2, 1, 3, 3, 4, 4, 5, 5, 7, 8,
1, 2, 3, 3, 4, 4, 5, 5, 7, 8,
1, 2, 3, 3, 4, 4, 5, 5, 7, 8,
Process Exit...
```

## ANALISIS DE TIEMPOS

Disponemos de dos metodos de ordenamiento basicos y uno mejorado. Es interesante comparar su desempeño comparando sus tiempos. Para ello, los usaremos ordenando de 1000, 5000, 10000, 20000, 50000 elementos. Como no sabemos que que nos provee Java para trabajar con el reloj del sistema, vamos al help y tras unas pocas búsquedas conseguimos elementos para codificar el siguiente main()

```
import java.util.Calendar;
import java.util.GregorianCalendar;
import OrdBurb;
class PrueGrecal{
    static int minut, segun, milis;
    static long horIni, horFin, tiempo;
    public static void main(String args[]){
        Calendar cal1 = new GregorianCalendar();
        minut = cal1.get(Calendar.MINUTE);
        segun = cal1.get(Calendar.SECOND);
        milis = cal1.get(Calendar.MILLISECOND);
        horIni= minut*60*1000 + segun*1000 + milis;
        System.out.println("\nOrdenamiento metodo Burbuja");
        System.out.println("Inicio "+minut+": "+segun+"."+milis);
        OrdBurb ord = new OrdBurb(1000,'R');
        ord.ordenar(false);          // y lo ordenamos, sin trazado
        System.out.println(ord);
        Calendar cal2 = new GregorianCalendar();
        minut = cal2.get(Calendar.MINUTE);
        segun = cal2.get(Calendar.SECOND);
        milis = cal2.get(Calendar.MILLISECOND);
        horFin= minut*60*1000 + segun*1000 + milis;
        System.out.println("Fin  "+minut+": "+segun+"."+milis);
        tiempo = horFin - horIni;
        System.out.println("Tiempo: " + tiempo + " milisegundos");
    }
};
```

Si lo ejecutamos obtenemos la siguiente salida:

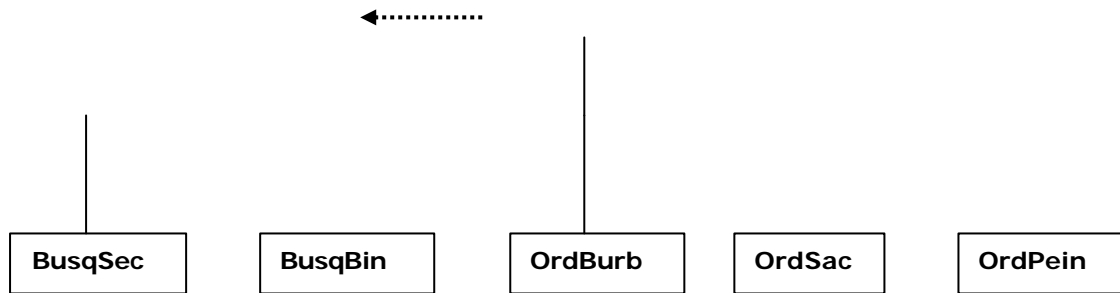
```
Ordenamiento metodo Burbuja
Inicio 49:55.450
999, 698, 846, 885, 335, 474, 838, 738, 244, 565,
Pasadas de ordenamiento
999, 999, 999, 997, 997, 997, 994, 993, 992, 991,
Fin 49:55.560
Tiempo: 110 milisegundos
Process Exit...
```

O sea que para ordenar un array de 1000 elementos, esta humilde cpu (Pentium III, 750 Mhs, 64 MB RAM), sin correr ningun otro proceso del usuario en forma simultanea requiere de 110 milisegundos. Presentamos en una tabla el comparativo de tiempos

Metodo	1000	5000	10000	20000	50000	100000
Burbuja	110	2190	10490	47570	345970	
Sacudida	110	1930	8840	38830	270010	
Peinado	50	60	110	280	600	1270

Una primera sorpresa: el metodo "sacudida", bastante mas elaborado que el burbuja, no mejora mucho sus tiempos. La segunda, es la relación de eficiencia del "peinado" respecto cualquiera de los dos anteriores. Un array de 50000 elementos "peinado" lo ordena **450 veces** mas rápido que "sacudida".

*Bueno, recapitulemos. Todo lo de búsqueda y ordenamiento en arrays lo hemos visto usando un conjunto de clases inter relacionadas, que trabajan "en equipo". Primero definimos la clase Item, que vincula un código con un valor. Esta clase la incluimos en la clase ArrItems (Relación "tiene un"). Heredando de ArrItems (relación "es un") implementamos las dos clases de búsqueda y las tres de ordenamiento.*



## 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 trabajar 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 cuestion ? ");
    mat01[f][c] = System.in.read();
}

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

```

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
    }
}

```

```

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

```

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**; podríamos 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. Usamos 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 valor? ");
                c = In.readInt();
                System.out.print("Cual es el valor en cuestion ? ");
                mat01[f][c] = In.readInt();
                break;          // Lectura exitosa, salimos del ciclo
            }catch(java.lang.ArrayIndexOutOfBoundsException e){
                System.out.println("\nCuidado, desborde de indice...")
                continue;      // Seguimos intentando la lectura
            }
        } // 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

```

```

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 randomicamente (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].getValor();
            }
        }
        prom = (double)(acuVal/contCod);
        return prom;
    }
}

import MatItems;
class pruMatItems{

```

```

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

```

```

    public static void main(String args[]){
        MatItems mtlIt = new MatItems(20,20,'R');
        // Generamos una matriz de 20 x 20 objetos Item,
        // codigos y valores al azar
        System.out.println(mtlIt);
        System.out.println("En la fila 10 el codigo 10 ocurre "+
                           mtlIt.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 "+mtlIt.promValCodEnCol(10,10));
    }
};

```

## EJERCITACION PRÁCTICA SUGERIDA

### Temas quinta semana

#### COMPOSICION USANDO UNA SUCESSION 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 SUCESSION 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 DígitoDecimal Letra, en ese orden, tenemos?
- Cuantas alternancias DígitoDecimal 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?

## Temas sexta semana

### 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.

## Temas septima semana

### 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.

## Temas octava semana

### 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(pg 44) 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.