FPRG

Apuntes de la asignatura (2009)

tomados de

J.C. Dueñas

(Libro:) Java 2 (2ª ed.)
Jesús Sánchez Allende
McGraw Hill

FPRG

Juan C. Dueñas
www.lab.dit.upm.es/~fprg
jcduenas@dit.upm.es

Solicitud de cuenta, en lab.dit.upm.es
imprimir y pegar foto
llevar a A-127-2

moodle + laborat. de progs.

25-sep-2009

Tema 1. Introducción a la informática
- 1.1. Definiciones
- 1.2. Lenguajes de programación
- 1.3. Programas

1.1. **INFORMÁTICA:** ciencia del tratamiento automático de la información.

Tratamiento de la info: comunicar, transformar, almacenar, adquirir, presentar.

ORDENADOR: máquina de propósito general programable, ejecuta las operaciones bajo el control de un programa almacenado en ella.

ALGORITMOS: descripción de la solución de un problema secuencia ordenada y finita de instrucciones elementales que partiendo de los datos de entrada produce resultados

DATOS: conjunto de valores que representan a la realidad con propiedades y operaciones

PROGRAMA: representación de un algoritmo con sus datos de entrada capaz de ser interpretado, realizado o ejecutado por un ordenador.

1.2 **LENGUAJES** permiten expresar algoritmos de forma que puedan ser procesados por el ordenador.

1 lenguaje ⇒ 5 condiciones
- simples
- concisos
- precisos
- abstractos
- capaces de resolver probl.

Legibilidad
Corrección
Eficiencia

(Libro pág 8)

3 características necesarias a la hora de hacer un progr.
2º - dar los resultados esperados (correctos)
3º - eficientes: uso correcto de los recursos
1º - legibles: por una persona (entendible)

⇨

Varias familias de lenguajes de progr.
Java es compilado.

- **compilación** (compilador) → lee el progr. en Java, comprueba y escribe en máquina.
- **interpretación** (intérprete) → lee línea a línea, la pasa a lenguaje máquina y ejecuta la instrucción

1.3. PROGRAMAS (libro pág 6

- Análisis de requisitos
- Diseño de solución
- Programación = Implementación = Codificación
- Hacer pruebas

a) Análisis requisitos: hacerse una idea y escribir el problema que queremos que resuelva el programa

b) Diseño de solución: cómo resolveremos el problema

c) Programación: escribir la solución en Java

d) Pruebas: comprobar que funciona correctamente

→ - escribir el texto / código del programa
- compilar → código máquina [o objeto
- ejecutar el progr. (intérprete)

Hello world = Hola a todos

fichero: HolaATodos.java ← no .doc, .txt...

```
class HolaATodos {
  public static void main (String[] args) {
    System.out.println ("Hola");
  }
}
```

en ventana "comandos ejecutar" del msz = Terminal en MacOS

```
>javac HolaATodos.java   Enter
>
```

↳ si está bien...

... se crea un fichero en la misma carpeta que se llama HolaATodos.class

```
> java HolaATodos
```

Errores comunes:
- sintácticos: fallo en el lenguaje (ej. falta el ; final) (detectado por el compilador)
- semánticos: 2 peras + 3 manzanas (sólo se detecta con pruebas)

30-sep-2009

Libro: programación en Java 3ª ed.
Jesús Sánchez Allende ✓
McGraw Hill

Descargar e instalar jdk (num. versión) → java development kit → contiene compilador e intérprete

Tema 2: Introducción a la programación orientada a objetos.

Tema 2.1. Clase: "tipo"
- representación de una entidad / concepto del mundo real o de la solución a un problema. Ej: puntos en el espacio 3D, $\mathbb{Z}$, $\mathbb{N}$, $\mathbb{R}$...
- definición de un conjunto de elementos con sus posibles valores y sus operaciones
  Ej: conjunto de ptos en el espacio 2D; $(x \in \mathbb{R})_x (y \in \mathbb{R})$
  atributos → Al dar la definición de los elementos: $(x, y \in \mathbb{R})$
  métodos → y sus operaciones: +, x, mover, dist.(0,0), dist 2ptos.
- plantilla que define por una parte los datos o campos (atributos) y las operaciones, funciones o métodos que van a tener los elementos de ese conjunto por otra parte.

```
class Punto {                                   // cabecera de la clase
    private double x;                           // atributos (cuerpo)
    private double y;
    public double getX() {                      // métodos
        return this.x;
    }
    public void setX(double x) {
        this.x = x;
    }
    public double distancia() {
        return Math.sqrt((this.x * this.x) + (this.y * this.y));
    }
}
```

double = $\mathbb{R}$
Math.sqrt = $\sqrt{\ }$
* = multiplicar

Punto	← cabecera
double x double y	← atributos
double getX() void setX(double x) double distancia()	← métodos

2-oct-2009

Creación de nuevas clases

* Composición - agregación (componer clases que ya existen y crear una nueva)
* Concepto problema/solución (crear clase por un concepto nuevo, que se encuentra en la solución al problema que tengo)
* Por si acaso (hay un fallo en algún sitio, y como no sé en dónde, creo de nuevo la clase por si acaso)

ejemplo incompleto

```
class PolinomioGrado2 {
    double a;
    double b;
    double c;
    double valor (double x) {

    }
}
```

no escribir "2PolinomioGrado" nunca el número al principio

estrategia 1) el método main se escribe dentro de una clase ya hecha

```
class ..... {
    .....;
    public static void main (String [] args) {

    }
}
```

estrategia 2) clase nueva sólo con el método main

```
class MiPrograma {
    public static void main (String [] args) {

    }
}
```

public static void main (String [] args) siempre tiene que estar dentro de una clase

Tema 2.2 Objetos:

representación en la memoria del ordenador (ocupando memoria) de un elemento de una clase

- los objetos son dinámicos: se crean y se destruyen durante la ejecución del programa
- tienen sus propios valores de los atributos y sus propias copias de los métodos
- tienen identidad: no hay dos iguales
- ciclo de vida de los objetos:
  - + se crea (new Punto ()) un objeto
    se crea una referencia

```
Punto p = new Punto ();
```

  - + llamar a sus métodos y operaciones

```
p.setX (-2.1);
```

la coma es un punto
(-2.1) = número decimal
no par de coord.

  - + matar al objeto = descolgarlo

```
p = null;
```

new = crear
= = asignación

Tema 2.3 Referencias:

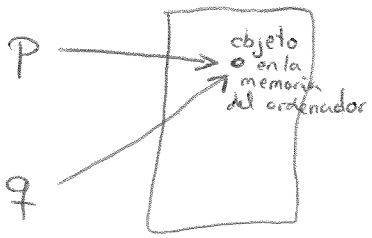
elemento/nombre que nos permite manejar un objeto

operaciones con referencias

+ creación

```
Punto p = new Punto ();
```

(crear referencia; asignarlos; crear objeto)



+ asignación
+ anularla
+ crear alias: asignar dos referencias a un mismo objeto

```
Punto q = p;
```

(q: refer.; p: refer.)

+ (comprobación) si p y q ~~eleve~~ apuntan al mismo objeto es correcto

7-oct-2009

Tema 2.4: Clases predefinidas

Bibliotecas de clases = conjunto de clases predef. (Library)

Bibliotecas del lenguaje = clases proporcionadas junto con el compilador
se pueden usar de forma directa
se cargan automáticamente al arrancar el programa

ejemplos clases:

① → String: representa una cadena de letras para escribir textos (palabras, líneas, Quijote...)

crear un objeto de clase String:

```
String s = new String ("hola");
String s = "hola";
```
lo mismo

```
String nombre = new String ("Jaribi Aguer")
```

concatenación: (= suma)

```
s = s + "pepito";
```
"holapepito"

si añades un espacio antes de pepito: " pepito" ⇒ "hola pepito"

`s.length()` = devuelve el número de letras

`s.equals("adiós")` = mira si "holapepito" es igual a "adiós" (dará falso)

poner algo

poner "texto" (con comillas) ó una referencia (sin comillas)

② → Math:

```
double Math.sqrt (__)
"      Math.cos  (__)   (en rad)
!      Math.sin  (__)   (seno)
!      Math.atan (__)   (arc tg)
!      Math.pow  (base, exp)  (potencia)
!      Math.PI
```

rad

③ → System.out

```
System.out.print (__);
System.out.println (__);   (imprime y hace salto línea)
```

9-oct-2009

Tema 3: Tipos y operaciones básicas

1. Datos y tipos
2. T. Lógico
3. T. Enumerados
4. T. caracter
5. T. Enteros
6. T. Reales
7. Conversiones
8. Arrays
9. Variables, atributos

moodle www.dit.upm.es
- entrar con usuario y contraseña propios
- matricularse con ⇒ contraseña: KKØt4a (minúsculas)

Tema 3.1. Datos y tipos

Dato: valor numérico o simbólico que representa cierta información del mundo real

Tipo: definición de un conjunto de datos expresando su rango o dominio y representando las operaciones.

Lenguajes tipados: (ej: Jara) cada expresión u operación es de un cierto tipo

Valor: representación de un elemento dentro de un tipo

Los datos ocupan memoria

Variable: contenedor que contiene un valor de un tipo. También ocupa memoria.

int = num. entero

```
int numeroAlumnos = 0;
```
nombre tipo — variable — valor

variables siempre en minúscula inicial

Los tipos | simples / básicos / primitivos | están predefinidos, vienen con el compilador y no se pueden modificar.

nombre clases → inicial Mayúscula
nombre variable → inicial minúscula

declarar una variable R en Jara con valor inicial 0.0 y nombre Temperatura

```
double temperatura = 0.0;
```

tipos primitivos incluidos en Java:
→ $\mathbb{R}$, $\mathbb{Z}$, letras o caracteres y valores lógicos (V o F)

Operaciones básicas aptas en todos los tipos primit.

1→ Operación de asignación:

```
numeroAlumnos = 60;
```
cambia el valor inicial 0 por 60

```
numeroAlumnos = numeroAlumnos + 60;
```
lo mismo

2→ Operación de comprobación de igualdad

```
(numeroAlumnos == 61)
```
↑ comparación

el anterior = 0

3→ Operación de comprobación de diferencia

```
(numeroAlumnos != 61)
```
¿c'es distinto?

desigualdad

!= ≡ ≠

Tema 3.2. Tipo lógico

true — false

```
boolean esTarde = true;
```

&& = and
|| = or

Operaciones lógicas

```
boolean resultado = false;
resultado = true && resultado;
```
← and
" false

&& tiene mayor preferencia que ||

true && true = true

T	&&	F
F	&&	T
F	&&	F

→ F

```
resultado = resultado || true;
```
← or
" true

false || false = false

T	||	F
F	||	T
T	||	T

→ true

```
resultado = !resultado;
```
" falso

! = no

Propiedad del cortocircuito

la ppdad and y or ⇒ si el primer término ya le hace obtener una conclusión, no sigue

ej: false && true
no lee true
porque sabe que si empieza por false será false

Tema 3.3 Tipos enumerados

Un tipo enumerado representa un conjunto de pocos valores, definidos por le programador, y escritos como una lista de valores en MAYÚSCULAS

```
enum ColorSemaforo {
    ROJO,  ← 0
    AMBAR, ← 1
    VERDE  ← 2
}
```

```
ColorSemaforo miSemaforo = ColorSemaforo.ROJO;
miSemaforo = ColorSemaforo.VERDE;
System.out.println(miSemaforo.toString()); ← imprime el color actual del semaf = Verde
System.out.println(miSemaforo.ordinal()); ← número 2
```

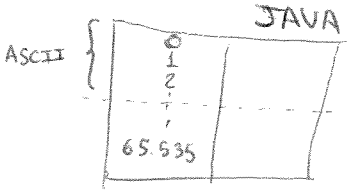
Tema 3.4 Tipo carácter

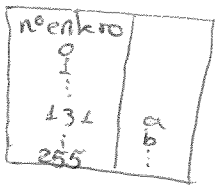
Compuesto por letras (May, min), números, signos de puntuación ...

Unicode ← ASCII

16 bits = 2 bytes — JAVA

8 bits = 1 byte — tabla de caracteres





'a' < 'b' < 'c' ... < 'z'

'A' < 'B' < 'C' ... < 'Z'

'0' < '1' < '2' ... < '9'

```
char letraNIF = 'B';
letraNIF = '\n';
char letra = 'A';
```

\n = enter
\t = tabulador
\\ = \
\' = '

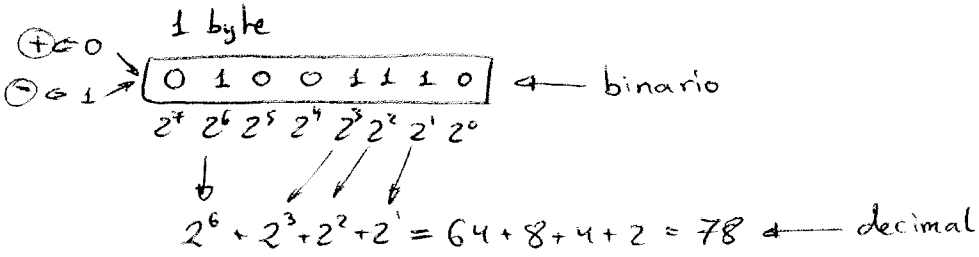
```
false = Char.isLowerCase(letra);
true  = Char.isUpperCase(letra);
'a'   = Char.toLowerCase(letra);
'A'   = Char.toUpperCase(letra);
```

Lower = minúscula
Upper = Mayúscula

14-oct-2009

Tema 3.5 Tipos enteros

Todos los enteros $(-\infty, +\infty)$ no caben en el ordenador
Si el resultado de una operación es mayor que el número máximo admitido, o menor que el mínimo da un ~~error de desbordamiento~~ = overflow



en Java...

long = 8 bytes	byte < short < int < long
int = 4 bytes	→ usar siempre int para números enteros salvo indicación contraria
short = 2 bytes	→ excepto
byte = 1 byte	- si son valores muy grandes (long)
	- si tenemos poca memoria (short/byte)

Definir una variable de tipo byte, short, int, long

```
byte numeritoChico = 0;
short numeroCorto = 0;
int numero = 0;
long numerazo = 0;
```

/ = división si y sólo si entero/entero

```
int columna = 2;
int fila = 3;
```

```
System.out.println(fila + columna); = 5
System.out.println(fila - columna); = 1
System.out.println(fila * columna); = 6
                  (fila / columna); = 1 (sólo el cociente)
                  (fila % columna); = 1 (sólo el resto)
                      (-fila);      = -3
(Math.abs(fila));                   = 3 (valor absoluto)
(Math.pow(fila, 2));                = 3^2 = 9
(++fila);
(fila++); } fila = fila + 1;
(--fila);
(fila--); } fila = fila - 1;
```

21-oct-2009

Operaciones-asignaciones compactas

operan y cambian el valor de la variable

```
++a → pre incremento  } a=a+1;
a++ → post incremento }
--a → pre decremento  } a=a-1;
a-- → post decremento }
```

```
int a=0;
System.out.println(++a);  ← imprime 1 y guarda a=1
System.out.println(a++);  ← imprime 1 y guarda a=2
System.out.println(--a);  ← imprime 1 y guarda a=1
System.out.println(a--);  ← imprime 1 y guarda a=0
System.out.println(a);    ← imprime 0 y guarda a=0
```

Tema 3.6 Tipo ℝ Reales

→ float < double
32 bits 64 bits
4 bytes 8 bytes

error de representación = desprecio de decimales

```
double a=-5.0;
double a=23e8;   // Número Entero!!
double a=-23e-8; // = -23·10^-8
```

23e8 = 23 · 10^8 → e ≡ E

operaciones con ℝ jamás dan problemas/errores
pueden dar como resultado ±∞ ó ±NaN
NaN Number = Indeterm

```
a+6.2;
a-1e6;
a*3.1;
a/0.6;
```

23-oct-2009

Tema 3.7 Conversiones

Java es fuertemente tipado ⇒ ⇒ todas las expresiones tienen un tipo

```
3.2 * 6
double  int
```

multiplicación de ℝ*ℝ ó ℤ*ℤ

→ la disciplina estricta de tipos NO permite esta operación

a) Conversiones implícitas o automáticas

byte < short < int < long < float < double

```
3.2 * 6
double int
```

→ el compilador convierte automáticamente → `3.2 * 6.0` (double double) ← ahora sí se realiza

las conversiones automát. sólo se aplican a la derecha de la asignación

```
int a = 3.2*6; ← double
  ↑ int
```

no puede guardar un double en un int → error de compilac.

b) Conversiones explícitas, forzadas o casting

byte < short < int < long < float < double

char

```
int a = (int)6.8;   a = 6
```

guarda un double en un int perdiendo la información decimal
NO redondea

```
(char)((int)'a' + 10)
```

convierte un char en un int para devolverlo a letra

Clases envoltorio

boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

```
int a = -10;
Integer A = new Integer(-10);
```

A = -10

permite usarlo como un objeto, es decir, usarlo con referencia

```
Integer A = -10;
int a = A + 6;
```

boxing (encajonar) + unboxing = autoboxing

Tema 3.8 Atributos y valores

```
class Punto {
    private double x;
    private double y;
    private final double PI = 3,1415926;
    public double getX() {
        return this.x;
    }
    public void setX(double x) {
        this.x = x;
    }
    public double distanciaAlO() {
        double a = ((this.x * this.x) + (this.y * this.y));
        return Math.sqrt(a);
    }
}
```

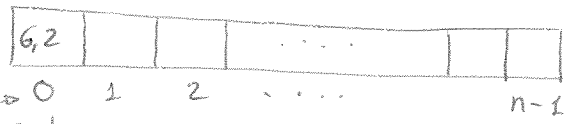
final: que no se puede cambiar nunca

una variable dentro de un método se destruye tras el mét. sólo su mét. puede acceder a él.

No dar valores inicialmente a los atributos sí a las variables de dentro de un método

Tema 3.9 Arrays (más importante de lo que parece)

Array: agrupación de variables del mismo tipo; conjunto homogéneo e indexado de variables de un tipo. ↳ (con índice)



índices: enteros que comienzan en 0 y acaban en n-1

vector → array de 1D
matriz → array de 2D
espacio/rejilla → array de 3D

los arrays son tipos referenciados

```
double [] notas = new double [100];
```
tipo de la base, 1D, referencia, tipo base, nº casillas

```
int [][] imagen = new int [100][100];
```
tipo base, 2D, ref, tipo base, filas, columnas

`notas[0] = 10.0;` la casilla 0 vale 10.0

`imagen[0][0] = 10;` la casilla (0,0) vale 10

`notas.length` devuelve 100

Tema 4: Métodos

1. Concepto
2. Definicion
3. Métodos básicos
4. Uso

Tema 4.1 Concepto

Método:
- función u operación que se aplica a los objetos de una clase
- parte de un algoritmo
- conjunto de operaciones (o sentencias) (incluyendo declaraciones de variables) este conjunto de sentencias está escrito dentro de una clase y cuando se llama/invoca o pide que se ejecute, realmente se realizan o ejecutan las operaciones en el orden en el que se han escrito, y devuelven un resultado.

Razones para no escribir todo el algoritmo en el main:
1. un método muy largo es poco entendible
2. si ponemos todas las operaciones dentro del main es poco reutilizable en el mismo/otro programa
3. se ha descubierto experimentalmente que si ponemos todo el algoritmo dentro del main se acaba repitiendo código.

→ Un método se define una sóla vez y sólo puede estar dentro del cuerpo de la clase.
→ No podemos definir métodos dentro de otros mét. ni fuera de las clases.
→ El orden de los métodos no es relevante.
→ Sólo podemos escribir operaciones o sentencias dentro del cuerpo de los métodos.
→ Cuando defino un método (escribirlo) (dentro de una clase) se puede usar todo lo que tiene la clase: atrib. y mét. (independientemente de public o private)

Para ejecutar un método: (al método se le puede llamar varias veces)
→ El orden de ejecución de las operaciones dentro del cuerpo de un método coincide con el orden en el que están escritas.
→ Para conocer el orden en el que se ejecutarán hay que empezar a leer desde el main: flujo de control.
→ El orden de ejecución de las operaciones NO tiene nada que ver con el orden en el que están los métodos

Práctica 1 10

```
class PolinomioGrado2 {

	private double a;
	private double b;
	private double c;

	public PolinomioGrado2 (double a, double b, double c) {
		this.a=a;
		this.b=b;
		this.c=c;
	}
	public double getA () {
		return this.a;
	}
	public double getB () {
		return this.b;
	}
	public double getC () {
		return this.c;
	}
	public void setA (double a) {
		this.a = a;
	}
	public void setB (double b) {
		this.b = b;
	}
	public void setC (double c) {
		this.c = c;
	}

	public double discriminante () {
		return (this.b*this.b - 4.0*this.a*this.c);
	}
	public double solucion1 () throws Exception {
		if (discriminante () < 0) {
			throw new Exception ("Error: discriminante negativo");
		}
		return (-this.b+Math.sqrt(this.discriminante()))/(2.0*this.a); // solucion 1: valor positivo de la raiz
	}
	public double solucion2 () throws Exception {
		if (discriminante () < 0) {
			throw new Exception ("Error: discriminante negativo");
		}
		return (-this.b-Math.sqrt(this.discriminante()))/(2.0*this.a); // solucion 2: valor negativo de la raiz
	}
	public double valor (double x) {
		return this.a*x*x + this.b*x + this.c ;
	}
	public Punto vertice () {
		Punto p = new Punto (0.0, 0.0);
		p.setX (-this.b / (2.0*this.a));
		p.setY (this.a*(-this.b/(2.0*this.a))*(-this.b/(2.0*this.a)) + this.b*(-this.b/(2.0*this.a)) + this.c);
		return p;
	}
	public boolean isConcava () {
		if (this.a > 0) {
			return true;
		}
		return false;
	  //devuelve true si el polinomio corresponde a una función cóncava
	}

}
```

Javier Rodriguez

```
class Punto {
  public double distancia (Punto p)
      throws Exception {
    if (p==null){
      throw new Exception ("No punto");
    }
    double d = Math.sqrt (((this.x - p.x)*(this.x-p.x))+
                          ((this.y-p.y)*(this.y-p.y)));
    return d;
  }
}
```

d tiene que ser Real

distancia entre P y otro punto ya definido

Tema 4.2 Definición

la cabecera del método indica cómo se usa o cómo se llama un método.

cabecera

a) → en la cabecera aparecen varias secciones:

↳ en la cabecera tiene que haber () = lista de parámetros

```
double distancia (Punto p) {
```

- double → tipo de retorno → indica lo que se espera recibir
- distancia → nombre del método → antes de ()

void → no esperamos ningún resultado

lista de los parámetros del método (. . . .)

parámetro: dato necesario para que se pueda ejecutar el método

si no hay parámetros se escribirán vacíos: ()

parámetros separados por comas

```
PolinomioGrado2 (double a, double b, double c) {
```

indicación de errores o excepciones (opcional)

```
double distancia (Punto p) throws Exception {
```

sobrecarga [dentro de una clase puede haber dos o más métodos con el mismo método.

para distinguirlos, el compilador usa la signatura del método: Punto (clase) distancia (nombre) Punto (parámetro)

cuerpo

b) El cuerpo de un método es un bloque de sentencias encerradas entre llaves.

Podemos: * crear variables (automáticas)

automáticas = la variable se crea y se destruye al finalizar automáticamnte al acabar el método.

→ No pueden ser usadas en otros métodos

* llamar a otros métodos
* se pueden usar los parámetros del método como si fueran variables automáticas

Los métodos se escriben en el orden en el que están escritos excepto:

+ ocurra un error o se lance una excepción, entonces, se parará la ejecución del método
+ si tenemos un método void se ejecutará desde la primera sentencia hasta la última
+ se ejecutará hasta que se encuentre "return"

Al llegar al final, se eliminarán las referencias.

Tema 4.3 : Métodos básicos

Deben de estar siempre en todas las clases

→ Constructores
→ Accesores
→ Modificadores

a) **Constructores**

+ se llaman = que la clase
+ sólo puede llamarse al constructor después de "new"
+ Si no se escribe ningún constructor, el compilador pondrá por omisión uno igual que "constr. 1" con valores iniciales 0

b) **Accesores**

permiten conocer el valor de un atributo desde fuera del objeto

c) **Modificadores**

llevan "void", no devuelven nada

```
class Punto {
  private double x;
  private double y;
  public Punto () {
    this.x = 0.0;
    this.y = 0.0;
  }                                   } constructor 1
  public Punto (double x, double y) {
    this.x = x;
    this.y = y;
  }                                   } const. 2

  public double getX () {             } accesor
    return this.x;                      (nombre del atributo con la primera letra Mayúsc.)
  }
  public double getY () {
    return this.y;
  }
  public void setX (double x) {       } modif.
    this.x = x;                         (inicial mayúscula)
  }
  public void setY (double y) {
    this.y = y;
  }
}
```

o
o
o

Método "básico" toString

```
System.out.println(p);
```

el compilador lo reconoce como: p.toString()

sólo puede haber uno.

```
class Punto {
    private double x;
    private double y;
    public String toString() {
        String m = "("+this.x+","+this.y+")";
        return m;
    }
    public boolean esOrigen() {
        return((this.x == 0.0) && (this.y == 0.0));
    }
```

método toString

boolean

6-nov-2009

```
class Punto {
    private double x;
    private double y;
    public Punto(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public double getX() {
        return this.x;
    }
    public void setX(double x) {
        this.x = x;
    }
    public double distancia(Punto p) throws Exception {
        if (p == null) {
            throw new Exception("Error");
        }
        double d = Math.sqrt(((this.x - p.x) * (this.x - p.x)) +
                             ((this.y - p.y) * (this.y - p.y)));
        return d;
    }
}
```

dos parám. formales

deben coincidir en orden:
3.2 = double x
6.1 = double y
- tipo
- número
2 parám. form
2 parám. reales
2 y 2

parámetro formal

← como es un double necesita devolver un double

* si hay void no requiere un return

Tema 4.4: Uso = llamar / invocar / ejecutar / usar un método

dos parám. reales

```
Punto p = new Punto(3.2, 6.1);
p.setX(-0.9);
```

parámetros reales

indicar el objeto (su referencia) seguida de un punto.

método al que se llama

valores de los parámetros/argumentos: (2 tipos):

1. parámetro formal: parámetro que aparece en la cabecera del método
2. parámetro real

los parámetros formales y reales tienen que coincidir en

- orden: el 1er parám. real se asocia al 1er parám. formal (a, b) => a → double x, b → double y
- tipo: si es double es double
- número: si la cabecera del método al que se llama requiere parámetros

```
Punto q = new punto (0.0, 0.0);
System.out.println (p.distancia (q));
```

parám. real

nombre método

objeto en el que se ejecuta

```
public double distancia (Punto p) throws Exception {
    double d = Math.sqrt ((this.x - p.x) * ...
}
```

Punto p = q

```
System.out.println (p.distancia(q));
```

```
public void setX (double x) {
    this.x = x;
}
double h = -0,9;
p.setX (h);
```

double x = h = -0,9

```
Punto p = new Punto (3.2, 6.1);
Punto q = new Punto (0.0, 0.0);
System.out.println (p.distancia(q));
```

this

parámetro

p → (3.2, 6.1) ← this

q → (0.0, 0.0)

ejercicio:

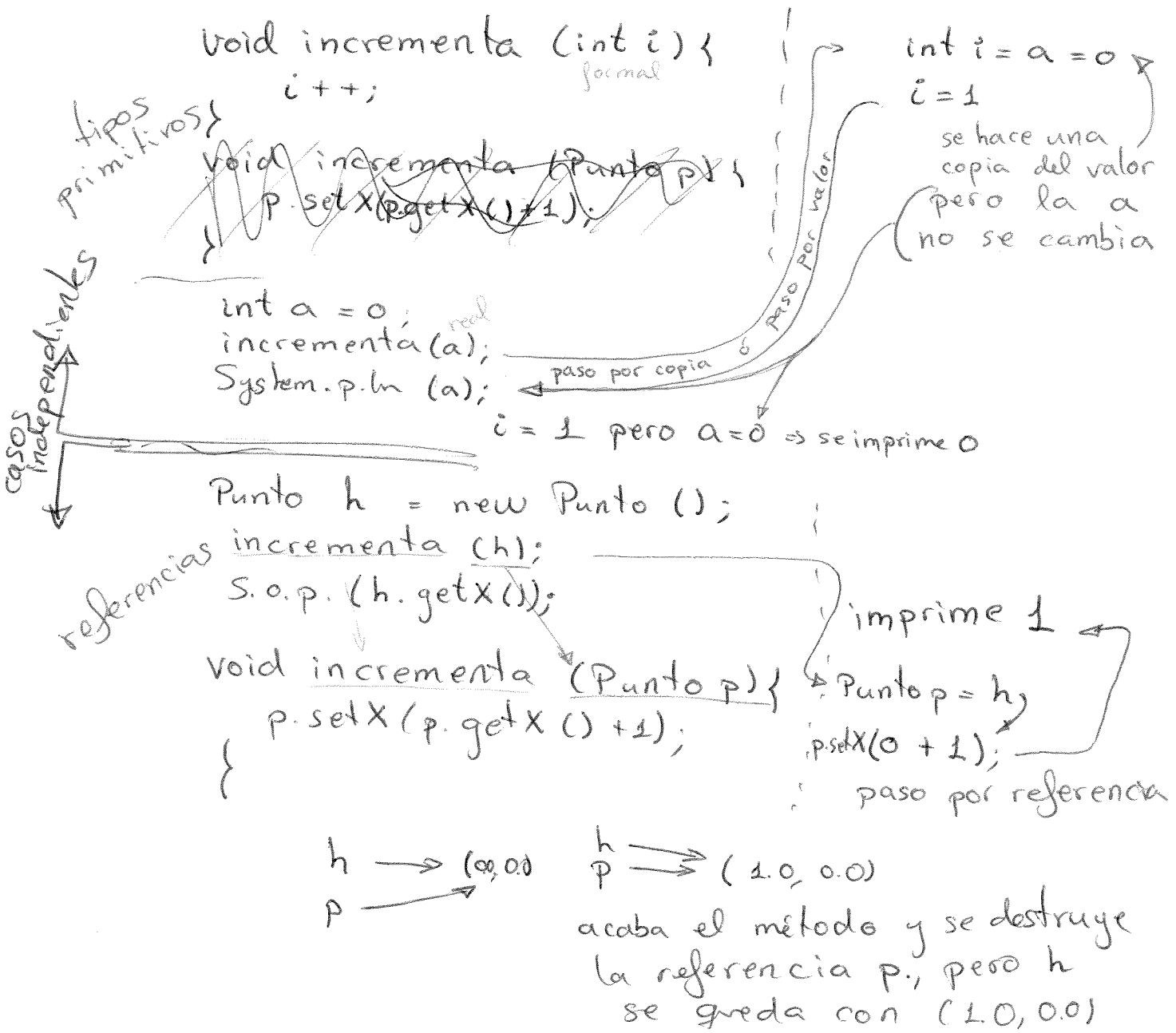
```
public double distancia (Punto p
    p'.setX (100.0);
    double d = Math.sqrt (this.x - p'.x)
{
```

```
Punto p = new Punto (3.2, 6.1)
Punto q = new Punto (0.0, 0.0)
System.out.println (p.distancia(q))
```

p → (3.2, 6.1) ← this

q ↔ (0.0, 0.0) ← p' → (100.0, 0.0)

cuando acaba la ejecución del método, se desvuelve la referencia (p' en este caso) No obstante, ahora q vale (100.0, 0.0)


void incrementa (int i) {
i++;
formal
tipos primitivos
void incrementa (Punto p) {
p.setX(p.getX()+1);
}
casos independientes
int a = 0;
incrementa(a);
real
System.p.ln (a);
paso por copia
paso por valor
int i = a = 0
i = 1
se hace una copia del valor pero la a no se cambia
i = 1 pero a = 0 ⇒ se imprime 0
Punto h = new Punto ();
referencias
incrementa (h);
S.o.p. (h.getX());
void incrementa (Punto p) {
p.setX(p.getX() + 1);
}
imprime 1
Punto p = h;
p.setX(0 + 1);
paso por referencia
h → (0,0)
p →
h
p → (1.0, 0.0)
acaba el método y se destruye la referencia p., pero h se queda con (1.0, 0.0)

11-nov-2009

Tema 5: Sentencias de control

1. Sentencias
2. Condicionales
3. Bucles
   - while
   - do while
   - for
   - for arrays
4. Saltos
5. Excepciones
6. Recursividad

Tema 5.1: Sentencias

Sentencia = operación (crear una variable, operar...)
las sentencias sólo pueden estar dentro de un método
flujo de control / flujo de ejecución: indicación del orden en el que se realizan (ejecutan) las sentencias
Las sentencias pueden ser:

* simples: (acaban con ";")
  - ej: declaración de una variable
    `double suma = 0.0;`
  - ej: asignación
    `suma = 100.0;`
  - ej: creación de un objeto con referencia
    `Punto p = new Punto();` → `Punto p; p=new Punto();`
  - ej: nada (sentencia nula)
    `;`
  - ej: llamada a un método
    `System.out.println("Jeo");`
* compuestas / bloque: (secuencia de sentencias encerradas entre llaves)
  - ej: declaración de variables
    ```
    double a = 0.0;
    ```

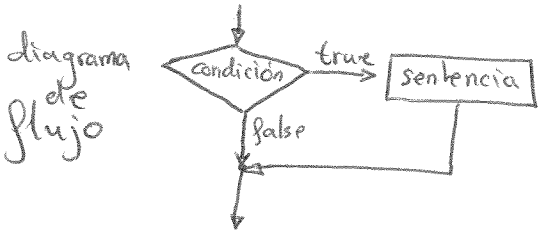
Tema 5.2: Sentencias condicionales { if, if else, switch }

Las sentencias condicionales permiten la ejecución de otras sentencias dependiendo de una condición

1. (if) permite la ejecución de la sentencia si y sólo si la condición es verdadera

```
if (condición) {
  sentencia;
}
```

diagrama de flujo

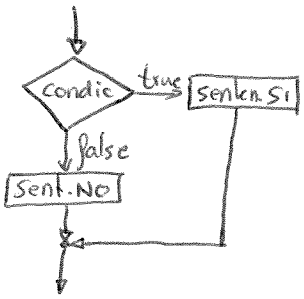


ej:
```
if (numeroAlumnos == 0) {
    System.out.println ("No hay clase");
}
```

2. if else

→ Si la condición es true se ejecuta la sentenciaSI y NO se ejecuta la sent.NO

→ Si la condición es falsa se ejecuta la sentenciaNO y NO se ejecuta la sent.SI



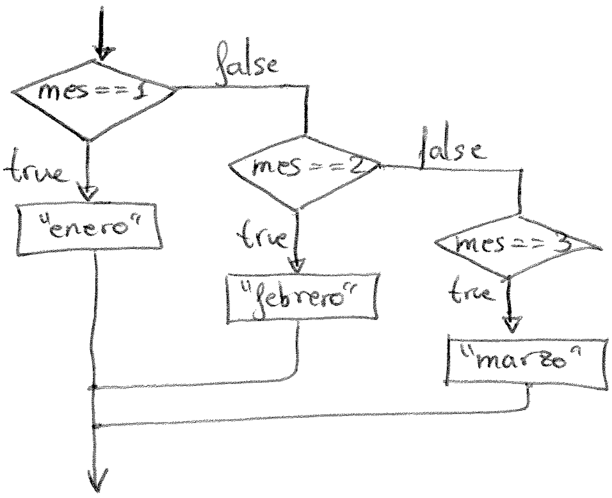

```
if (condición) {
    sentenciaSI;
}
else {
    sentenciaNO;
}
```

ej:
```
if (numeroAlumnos == 0){
    System.out.println ("No hay clase");
{
else{
    System.out.println ("Si hay clase");
}
```

ej:
```
if (numeroAlumos == 0) {
    System.out.println ("No hay clase");
}
else { if (numeroAlumnos > 0) {
    System.out.println ("Si hay clase");
    }
    else {
        System.out.println ("Error: nº de alumos negativo")
    }
}
```

13-nov-2009

ámbito estático / ámbito de la variable automática → todo el espacio en el que se puede usar: desde que se crea hasta que finaliza el método.




```
if (mes == 1) {
  System.out.println ("enero");
}
else {
  if (mes==2) {
    s.o.p. ("febrero");
  }
  else {
    if (mes ==3) {
      s.o.p. ("marzo");
    }
      :
  }
}
```

para poder simplificarse se han de cumplir:

- * las condiciones sean excluyentes (no puedan ser ciertas más de una)
- * en la rama else sólo haya una sentencia if

```
if (mes == 1) {
  sop. ("enero");
}
else if (mes==2) {
  s.o.p ("febrero");
}
else if (mes == 3) {
  s.o.p ("marzo");
}
else if (mes == 4) {...
```

3. switch ó multicondicional

```
switch (mes) {           // selector
   case 1 : System.out.println ("enero"); break;
   case 2 :    "     "    ("febrero"); break;
     :                    // etiquetas
   case 12 : S...       ("diciembre"); break;
   default : Systen...  ("error");    // opcional
}
```

si mes No coincide con ninguna etiqueta se ejecuta default

si no hubiese ningún break y mes fuese 2 se imprimiría: febreromarzoabrilmayo junio... diciembreerror

El selector sólo puede ser: boolean, int, char
Las etiquetas tienen que ser del mismo tipo que el selector

case = empezar a ejecutar desde ahí hasta el final
pero si añadimos break al final de cada case, cuando ejecuta un caso ya finaliza (olvidando las demás)

ej: escribir mediante un switch los días de cada mes.

```
int mes;
int dias = 0;
switch (mes){
   case 1:  dias = 31;
            break;
   case 2:  dias = 28;
            break;
   case 3:  dias = 31;
            break;
     :
   case 12: dias = 31;
}
```

```
int mes;
int dias = 0;
switch (mes){
   case 1:
   case 3:
   case 5:
   case 7:
   case 8:
   case 10:
   case 12: dias = 31; break;
   case 4:
   case 6:
   case 11: dias = 30; break;
   case 2:  dias = 28; break;
   default: ;
}
```

(1, 3, 5, 7, 8, 10, 12: nada) si mes = 3 ejecuta

← si no se pone default hace lo mismo.

Tema 5.3: Bucles

usado 30% while
usado 10% do while
usado 60% veces { for, for arrays

Bucle: sentencia de control iterativa
permite repetir la ejecución de las sentencias del bucle
Normalmente, el número de veltas depende de una condición lógica.

Clasificación de bucles:

a) se conoce cuántas vueltas se van a dar (antes del bucle)
   ej: sumatorio de los 100 primeros nos (sabemos que son 100) entonces elegiremos "for"
b) si el mímero de veltas mínimo es 1
   elegiremos "do while"
   si el número mínimo de veltas es 0
   elegiremos "while"

1. while entre 0 y n veces cuando n = desconocido

```
int n = (int) (100 * Math.random());
while (n>50){
    n = (int) (100 * Math.random());
}
```

Math.random da un valor real entre 0 y 1 aleatorio

2. do while entre 1 y n veces cuando n = desconocido

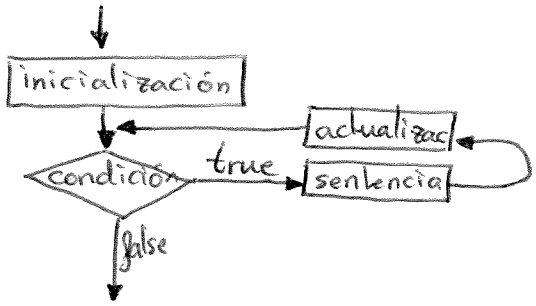
```
do {
  System.out.println("+");
} while (Math.random()>0.5);
```

Sheriff old West
primero dispara (ejecuta) luego pregunta

3. for

número mínimo de vueltas = 0
número máximo de vueltas conocido

```
for (inicialización; condición; actualización){
    sentencias;
}
```



obligatorio dar valor inicial

```
for (int i=0; i<100; i++){
  System.out.println (i);
}
```

se imprimirá:
0
1
2
.
.
99

```
for (int i=99; i>=0; i--){
  System.out.println (i);
}
```

```
for (int i=0; i<100; i++){
  System.out.println (99-i);
  {
```

ambos dos imprimen 99, 98, 97 ... 1, 0

```
for (int i=0; i<=49; i++){
  System.out.println (i*2);
}
```

```
for (int i=0; i<100; i=i+2){
  System.out.println (i);
}
```

```
for (int=0; i<100; i++){
   if (i%2==0){
     System.out.println(i);
   }
}
```

si (i%2==0) = false
no se hace nada
se actualiza y
vuelve a comprobar
la condición y la sentencia

$\sum_{i=0}^{99} i$ →

```
int suma=0;
for (int i=0; i<=99; i++){
   (suma+=i;) == (suma = suma+i)
}
System.out.println (suma);
```

n! →

```
int factorial (int n) {
   int factorial = 1;
   for (int i=n; i>=1; i--){
      (factorial *= i;)==(factorial = factorial*i)
   }
   return factorial;
}
```

20-nov-2009

mmiguel@dif.upm.es

Tema 5.4: Saltos

break; → termina el buble
continue; → termina el ciclo

```
for (i=1; i<20; i = i+1)
  if (tabla[i] == numero) break;   ← acaba el bucle
```
si la condición no se cumple nunca...
se acabará cuando acabe la condición

```
for (int i=1; i<20; i = i+1) {
  if (esPrimo(i)) continue;   ← termina el cuerpo del bucle y vuelve a evaluar la condición
  imprime(i);
```
imprime sólo los primos

Tema 5.5: Excepciones

```
throw new Exception ("lo tienes mal");
```

```
try {                    llaves SIEMPRE
} catch (tipo excep.) {
} catch (tipo excep) {
} finally {
```
opcional

```
public static void main(String[] args){
  try {
    double d = Double.parseDouble(args[0]);
    if (d<0) throw new Exception
    (args[0]+"no puede ser negativo");
    S.O.pln(Math.sqrt(d));
  } catch (NumberFormatException e){
    System.err.println(args[0]+"no es num.real");
  } catch (Exception e) {
      System.err.println(e);
  }
}
```

Tema 5.6: Recursividad

- directa: un método es recursivo cuando se llama a sí mismo
- indirecta: otro método al que éste llama incluye llamadas al método llamador

Arrays 1D

```
int [] a = null;
a = new int [100];   se crea el array de 100 casillas de enteros
a[0] = 10;   a la casilla 0 vale 10
a[3*4] = 16;   la casilla 12 vale 16
a[a.length - 1] = 19;   última casilla vale 19
```

```
Punto [] trayectoria = null;
trayectoria = new Punto [100];   no apunta a 100 objetos de clase Punto apunta a 100 referencias
trayectoria[0] = new Punto ();
```

```
Void m (Punto [] t) throws Exception{
    if (t == null) throw new Exception (" ");
    int n = t.length;
}
```

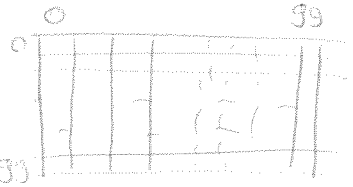
```
int [] b = { 1, 3, 8, 9, 16, 23 };
```
casilla 1, casilla 2, ..., casilla 6

Arrays 2D

```
double [][] mapa = new double [100][100];
```

```
mapa [0][0] = 3.0;
mapa.length   nº filas
mapa[0].length   nº columnas
```

mapa →




```
void ponVabo (int[] a, int n) {
    for (int i = 0; i < a.length; i++){
        a[i] = n;
    }
}
```

método en el que todas las casillas del array valdrán n

```
void imprimir (int[] a) {
    for (int i = 0; i < a.length; i++){
        System.out.println (a[i]);
    }
}
```

método que imprime cada casilla

Práctica 2

10

```
class Localizador {

// atributos

        private Punto p1; // p1 es el centro de la circunferencia 1: la antena 1
        private Punto p2; // p2 es el centro de la circunferencia 2: la antena 2

// constructor con dos parametros que lance una excepcion si algun punto no existe o tienen los mismos valores

        public Localizador (Punto p1, Punto p2) throws Exception {
                if (p1 == null || p2 == null) throw new Exception ("ERROR: no existe algun punto");
                if (p1.getX()==p2.getX() && p1.getY()==p2.getY()) throw new Exception ("ERROR: puntos iguales");
                this.p1 = p1;
                this.p2 = p2;
        }

// metodo privado que calcula un punto de corte de las circunferencias con centro en p1 y p2, y radios d1 y d2
// dependiendo de si el parametro signo es +1.0 o -1.0 se calculara uno de los puntos de corte o el otro
// si no hay solucion (si las circunferencias no son tangentes ni secantes entre ellas) se lanzara una excepcion

        private Punto calculaCorte (double d1, double d2, double signo) throws Exception {

                if (p1.distancia(p2)>d1+d2) throw new Exception ("ERROR: no existe Punto de corte");

                Punto c = new Punto (0.0, 0.0);
                double discriminante;

        // se calcula el punto de corte en el caso de que (x1 = x2)
                if (p1.getX() == p2.getX()) {

                  discriminante = d1*d1-(c.getY()-p1.getY())*(c.getY()-p1.getY());
                  if (discriminante < 0) throw new Exception ("ERROR: raiz de un negativo");

                        c.setY ((d1*d1-d2*d2+p2.getY()*p2.getY()-p1.getY()*p1.getY()) / (2*(p2.getY()-p1.getY())));
                        c.setX (p1.getX() + signo*Math.sqrt(discriminante));
                        return c;
                }

        // se calcula el punto de corte en el caso de que (y1 = y2)
                if (p1.getY() == p2.getY()) {

                  discriminante = d1*d1-(c.getX()-p1.getX())*(c.getX()-p1.getX());
                  if (discriminante < 0) throw new Exception ("ERROR: raiz de un negativo");

                        c.setX ((d1*d1-d2*d2+p2.getX()*p2.getX()-p1.getX()*p1.getX()) / (2*(p2.getX()-p1.getX())));
                        c.setY (p1.getY() + signo*Math.sqrt(discriminante));
                        return c;
                }

        // se calcula el punto de corte en el Caso General
                else {
                    PolinomioGrado2 pg = new PolinomioGrado2 (0.0, 0.0, 0.0);

                        double a = (d1*d1-d2*d2+p2.getX()*p2.getX()-p1.getX()*p1.getX()+p2.getY()*p2.getY()-p1.getY
()*p1.getY()) / (2*(p2.getX()-p1.getX()));
                        double b = (p1.getY()-p2.getY()) / (p2.getX()-p1.getX());

                    if (signo == +1.0) {
                        pg.setA (b*b + 1);
                        pg.setB (2*(b*a - b*p1.getX() - p1.getY()));
                        pg.setC ((a - p1.getX())*(a - p1.getX()) + p1.getY()*p1.getY() - d1*d1);

                        c.setY (pg.solucion1());
                        c.setX (a + b*c.getY());
                        return c;
                    }

                    if (signo == -1.0); {
                        pg.setA (b*b + 1);
                        pg.setB (2*(b*a - b*p1.getX() - p1.getY()));
                        pg.setC ((a - p1.getX())*(a - p1.getX()) + p1.getY()*p1.getY() - d1*d1);

                        c.setY (pg.solucion2());
                        c.setX (a + b*c.getY());
                        return c;
                    }
                }
        }
```

```
// metodo publico que devuelve el punto solucion si existe
// este metodo llama al metodo calculaCorte y calcula el punto medio

        public Punto calculaPosicion (double d1, double d2) throws Exception {
                if (calculaCorte(d1, d2, +1.0)==null || calculaCorte(d1, d2, -1.0)==null) throw new Exception
("ERROR: no existe Punto solucion");
                if (d1<0 || d2<0) throw new Exception ("ERROR: radio/s negativo/s");
                Punto c1 = calculaCorte(d1, d2, +1.0);
                Punto c2 = calculaCorte(d1, d2, -1.0);
                Punto medio = new Punto ((c1.getX()+c2.getX()) / 2, (c1.getY()+c2.getY()) / 2);
                return medio;
        }

}
```

Javier Rguez

```
boolean buscar (int[] a, int buscado){
    for (int i=0; i<a.length ; i++) {
        if ( a[i] == buscado)
            return true;
    }
    return false;
}
int buscar (int[] a, int buscado){
    for(int i=0 ; i<a.length ; i++){
        if (a[i] == buscado)
            return i;
    }
    return -1; //valor ilógico
}
int max (int [] a) {
    int max = a[0];
    for (int i=0; i < a.length; i++){
        if (a[i] > max)
            max = a [i];
    }
    return max;
}
```

27-nov-2009

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} \cdots$$

$$\left.\begin{array}{l} a_0 = 1 \\ a_n = -a_{n-1} \dfrac{x^2}{2n(2n-1)} \end{array}\right\} \cos(x) = \sum_{n=0}^{\infty} a_n = \left(\sum_{n=1}^{\infty} a_n\right) + 1$$

ej: hacer cos(x) con un bucle do-while

```
double cos (double x, double p){      p = precisión (error)
    double res = 1.0;
    int n = 1;
    double an = 1.0;
    do {
        an = -an * x*x / (2*n*(2*n-1));
        res += an; // res = res + an;
        n++; // n = n+1;
    } while (Math.abs(an) > p);
    return res;
}
```

```
int [] a = new int [100]          ≡   int [] a = new int [100]
for (int n : a) {                     for (int i=0; i<a.length; i++) {
}                                       int n = a[i];
                                      }
```

for (más simple) con arrays.

→ sólo con dos condiciones:

1* no importa el índice de la casilla

2* no se cambie el valor de ninguna casilla con el for.

```
int [] a = new int [100];
for ( int n : a) {
    System.out.println (n);
}
```

} for (int i=0; i<a.length; i++) { int n = a[i]; System.out.println(n); }

for con Arrays 2D

```
int [][] a = new int [100][100];
bucle: for (int i=0; i<a.length ; i++) {
    for (int j=0 ; j<a[i].length; j++) {
        System.out.println (a[i][j]);
        { break bucle;
    }
}
```

"nombre de la etiqueta" + ":"

cuando llega a break se acaba el bucle etiquetado como "bucle" (el más externo).

2-dic-2009

Excepciones

1. lanzamiento (throw)

```
throw new Exception ("mensaje");

Exception e = new Exception (" ");
throw e;
```

2. indicación de que un método puede lanzarla (en la cabecera) throws Exception {

3. tratamiento de excepciones

```
try {
    d = p.distancia (q);
    System.out.println (d);
} catch (Exception e) {
    d = 0.0;
    System.out.println ("Error");
    (e.getMessage());
}
```

} zona de cosas que tienen peligro si aquí salta

← aquí se atrapa

opcional → finally { ~~~ }

se ejecuta siempre

4-dic-2009

Tema 6: Clases

1. Clases
2. Derechos acceso
3. Elementos estáticos
4. Relaciones entre clases

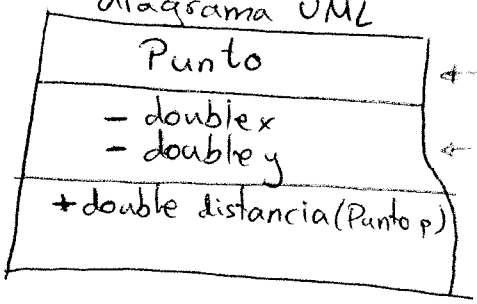
Tema 6.1: Clases

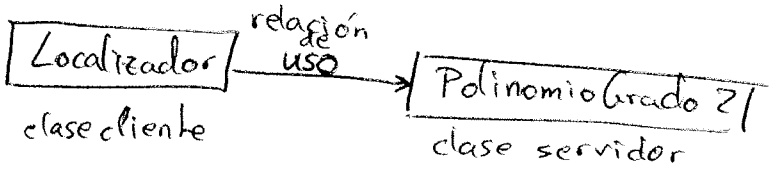
ENCAPSULACIÓN: propiedad de los lenguajes de programación por la cual se manejan de forma conjunta e indivisible algunos datos junto con sus operaciones.

1 objeto = 1 unidad de encapsulamiento

UML: Unified Modeling Language: regla que siguen los diagramas que muestran una clase.

diagrama UML





Tema 6.2: Derechos de acceso: muestran qué partes del programa pueden acceder (usar)

```
class Punto{
  private double x;
  private double y;
public Punto(){}
{
public void setX(double x){}
{
}
```

private: sólo se puede usar dentro de esa clase no se puede modificar ni leer!!

public: se puede usar en cualquier sitio

→ Interfaz de la clase = parte pública de la clase (cabeceras y cosas public)

→ Implementación de la clase = parte privada de la clase (cuerpos y atrib privados)

ADT = Tipo abstracto de datos => {atributos private y métodos public con constr. acces. y modif.

→ Ocultación de información => técnica que se logra marcando los atrib privados y los acc. y modif. públicos

Tema 6.3: Elementos estáticos =(Hacer chapuzas)

```
class Punto {
    private static int numPuntos = 0;
    public static int getNumPuntos () {
        return numPuntos;
    }
    public Punto () {
        this.x = 0.0;
        this.y = 0.0;
        numPuntos ++;
    }
```

si no estuviese el static al final se crearía una variable numPuntos para cada punto

Por cada creación de punto se crearía una nueva variable numPuntos con valor 1.

static: se pueden marcar tanto los atributos como los métodos como estáticos

static ⇒ { compartido con todos los objetos de la clase
existe una sóla vez
arranca con el programa y muere donde acaba aunque no se creen elementos o no se usen.

```
Punto.getNumPuntos()
```
← fuera de la clase accede al método de la clase Punto

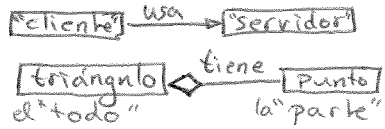
→ a los estáticos hay que darles valor inicial
son los únicos atributos que se inicializan

Static: Mosqueteros

uno para todos

Tema 6.4: Relaciones entre clases

a) Relación de uso
b) Relación de composición




```
class Triangulo {
    private Punto p1; private Punto p2;
    private Punto p3;
public Triangulo (Punto p1, Punto p2, Punto p3) {
    this.p1 = p1; this.p2 = p2...
}
```

por referencia

```
public Triangulo (Punto p1, Punto...
this.p1 = new Punto(p1.getX(), p1.get...
this.p2 = ...
```

por copia

9-nov-2009

Tema 7: Relaciones - clases

1. Polimorfismo
2. Implementación de interfaces
3. Uso de interfaces
4. Jerarquías de interfaces
5. Extensión ó Herencia
6. Jerarquías

Tema 7.1: Polimorfismo (Varios aspectos/formas de una cosa)

en java => 1 aspecto/forma de varias cosas

```
class Antena {
  private Punto posicion;
  private double distancia;
  public Punto getPosicion () {
    return posicion;
  }
```

```
class TlfMovil {
  private String numero;
  private String IMEI;
  private Punto posicion;
  public Punto getPosicion(){
  }
```

Condiciones entre clases para hacer un polimorfismo:
* sean clases distintas
* tener al menos una cabecera de método público igual

clase interfaz

```
interface + nombreclase {
```
(en vez de) class

dentro de una interfaz puede haber:
- constantes (final)
- cabeceras de mét. públicos

```
interface Ubicado {
  public Punto getPosicion();
}
```
← sin cuerpo!! sólo cabecera de método público

cada interfaz tiene un fichero. Nunca dentro de otra clase

prohibido dentro de una interfaz:
- atributos
- cuerpos de métodos
- cabeceras métodos privados
- constructor

qué puedo hacer:
- crear referencias
  Ubicado u = null;
- llamar a sus métodos
Punto p = u.getPosicion();

qué no se puede hacer:
- crear objetos
Ubicado u2 = new Ubicado(); MAL

Tema 7.2: Implementación de interfaces

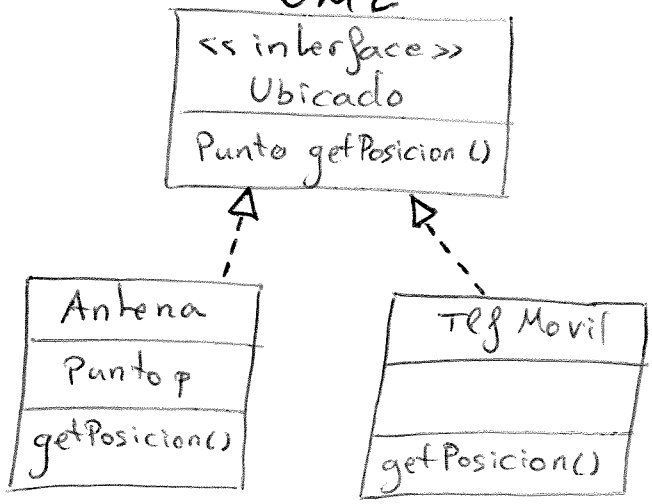
```
interface Ubicado {
    public Punto getPosicion();
}
```

```
class Antena implements Ubicado{
    private Punto posicion;
    public Punto getPosicion(){
        return posicion;
    }
...
}
```

```
class TlfMovil implements Ubicado{
    public Punto getPosicion() {
    ....
        return p;
    }
}
```

para poder implementar
- tiene que aparecer en la cabecera implements + nombre
- para cada cabecera de la interfaz tiene que haber cabecera y cuerpo en la/s clases.

UML

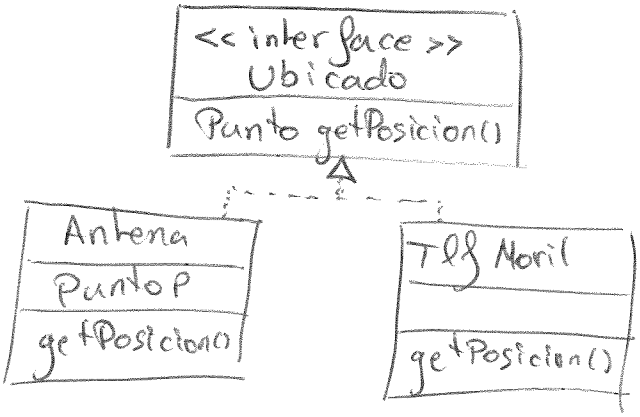


11-nov-2009

Tema 7.3: Uso de interfaces

Una referencia de tipo interfaz sólo puede apuntar a objetos de las clases que implementan a la interfaz

```
Ubicado u = new Antena();
u = new TlfMovil();
```
cambiamos la referencia



Up-casting: manejar un objeto de una clase mediante una referencia de tipo interfaz

a) Up-Casting (seguro)

ej: Cabeceras de métodos necesarios para calcular la distancia entre pares de elementos:
antena-antena, antena-movil, movil-movil

```
public double distancia (Antena a1, Antena a2) {
public double distancia (TlfMovil t1, TlfMovil t2)
public double distancia (TlfMovil t1, Antena a1)
public double distancia (Antena a1, TlfMovil t1) {
    return a1.getPosicion().distancia(t1.getPosicion());
}
```

o lo que es lo mismo:

```
public double distancia (Ubicado u1, Ubicado u2) {
    Punto p = u1.getPosicion();
    Punto q = u2.getPosicion();
    return p.distancia(q);
}
```

Ahora, si ponemos un camión, no necesitamos hacer la distancia entre ternas de elementos: camion-movil-antena si no que ya podemos usar el ubicado

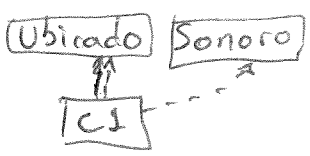
ligadura tardía o dinámica: identificar el método real al que se va a llamar durante la ejecución del programa

b) Down-Casting (peligrosillo) → poco recomendable

```
Ubicado u = . . . ;
Punto p = u.getPosicion();
((TlfMovil) u).sendSMS("ssss", "Loteria40pplos");
```

Si la referencia (u) no es de la clase TlfMovil salta un error incapturable con try-catch.
ERROR: classCastException

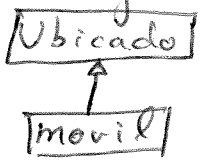
Tema 7.4: Jerarquías de Interfaces
una clase puede implementar varias interfaces




```
class C1 implements Ubicado, Sonoro {
    public Punto getPosicion() {
    }
{ - - - -
```

una interfaz puede agrupar a otras interfaces

```
interface Movil extends Ubicado {
    public void mover(Punto p);
{

class CL implements Movil {
    public void mover(Punto p) {
    {
    public Punto getPosicion() {
    {
{
```

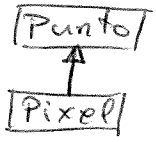


Tema 7.5: Extensión o Herencia

Herencia: relación entre clases que nos permite crear una clase nueva tomando como punto de partida una clase que ya está hecha y añadiéndole métodos o atributos.

Debe existir alguna relación entre la clase base (original) y la derivada (la nueva). Esta relación se llama **es-un**

ej: Pixel es-un Punto



Una clase derivada sólo tiene un padre

Clase Object = clase más alta de la jerarquía de clases

```
class Punto { ≡ class Punto extends Object {
```

este paso lo hace el compilador

```
class Pixel extends Punto {
    private String color;
    public String getColor() {
        return color;
    {
{
```

en la clase Pixel (o fuera) se pueden usar todos sus métodos y todo lo público de la clase de la que deriva. No se puede acceder a la parte privada de la clase base

```
Pixel px = new Pixel();
px.setColor("GREEN");
px.setX(10.6);
px.distancia();
```

Herencia de la Interfaz
No Interface
Sí Parte Pública de una Clase

Herencia de la Implementación
Parte Privada de una Clase

```
class AntenaOmnidireccional {

        private String nombre;
        private Punto posicion;
        private double alcance;

        public AntenaOmnidireccional (String nombre, Punto posicion, double alcance) {
                this.nombre = nombre;
                this.posicion = posicion;
                this.alcance = alcance;
        }

        public double getAlcance() {
                return this.alcance;
        }

        public String getNombre() {
                return this.nombre;
        }

        public Punto getPosicion() {
                return this.posicion;
        }

        public boolean cobertura (Punto p) throws Exception {
                if (p == null) throw new Exception ("ERROR: punto para calcular distancia null");
                return this.posicion.distancia (p) <= this.alcance;
        }
}
```

Javier Rguez

```
class Region {

// La clase Region representa una zona del espacio cuyas dimensiones se indican en el constructor (usando una unidad
// de longitud expresada mediante un número entero). De una región se puede obtener un mapa, que es un array o matriz
// bidimensional de caracteres en donde las coordenadas de cada punto son las de cada casilla del array; el mapa es
// uno de los atributos de la region.

        private String nombre;
        private char [] [] mapa;
        private AntenaOmnidireccional [] antenas;

        public Region (String nombre, int ancho, int alto) {
                this.nombre = nombre;
                this.mapa = new char [ancho] [alto];
                antenas = new AntenaOmnidireccional [9];
        }

// La region tambien incluye un array de 9 referencias a AntenaOmnidireccional, inicialmente vacio, y un metodo para
// añadir una AntenaOmnidireccional a este array (si queda sitio, porque si no cabe, no hace nada).

        public void addAntena (AntenaOmnidireccional a) {
                for (int i = 0; i < antenas.length; i++) {
                        if (antenas [i] == null) {
                                antenas [i] = a;
                                break;
                        }
                }
        }

// Indica para un punto de la region cuantas antenas tienen cobertura sobre el.

        public int alcanzadoPor (Punto p) {
                int numAntenas = 0;
                for (int i = 0; i < antenas.length; i++) {
                    try {
                        if (antenas[i].cobertura(p)) numAntenas++;
                    }
                    catch (Exception e) { }
                }
                return numAntenas;
        }

// Indica para un punto de la region si coincide con la posicion de alguna antena.

        public boolean hayAntena (Punto p) {
                for (int i = 0; i < antenas.length; i++) {
                    try {
                        if (antenas[i].getPosicion().distancia(p) == 0.0) return true;
                    }
                    catch (Exception e) { }
                }
                return false;
        }

// Cuando se ejecuta este metodo, se calcula y rellena el mapa de la clase Region.
// Para ello se recorre el mapa y se pone en cada casilla:
// un caracter 'x' si hay una antena, da igual que haya o no cobertura.
// un caracter ' ' (espacio en blanco) si no hay ninguna antena que de cobertura a ese punto.
// en el resto de los casos un caracter correspondiente al digito del numero de antenas que dan cobertura.

        public void calculo () {
                int w = (int) '0';
                for (int i = 0; i < mapa.length; i++) {
                    for (int j = 0; j < mapa[i].length; j++) {
                        if (alcanzadoPor(new Punto(i,j))==0) mapa[i][j] = ' ';
                        else mapa[i][j] = (char) (w + alcanzadoPor(new Punto (i, j)));
                        if (hayAntena(new Punto(i, j))) mapa [i][j] = 'x';
                    }
                }
        }

// Imprime los caracteres de la rejilla o mapa en la pantalla.

        public void imprime () {
                for (int y = mapa[0].length - 1; y >= 0; y --) {
                    for (int x = 0; x < mapa.length; x ++) {
                        System.out.print (mapa [x][y]);
                    }
                    System.out.println ();
```

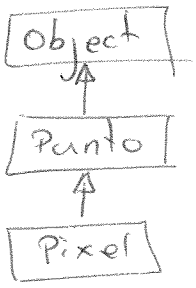
Javier Rguez

```
                }
        }

// Devuelve el caracter que tiene la casilla del mapa cuyas coordenadas son x,y
// Si estas coordenadas estan fuera del mapa devuelve un espacio en blanco ' '.

        public char cobertura (int x, int y) {
                if (x >= mapa.length || y >= mapa[x].length) return ' ';
                return mapa[x][y];
        }

}
```

Javier Rguez

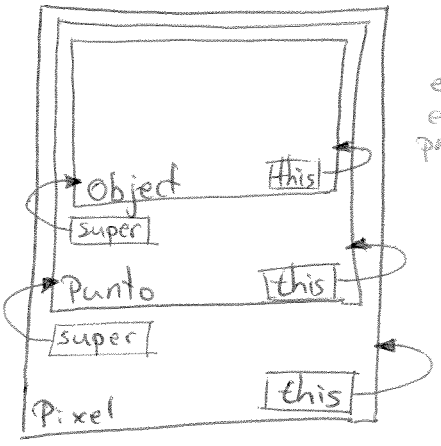
si en vez de marcar los atributos como privale los marcamos como protected esto implica que para las clases que hereda será público y para las demás clases privado.

15-dic-2009



class Punto {
compilador → extends Object

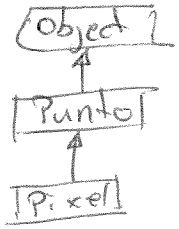
class Pixel extends Punto {



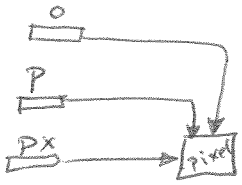

```
class Punto {
  Punto () {                    // el this debe estar primero
    this (0.0, 3.0);            // el constructor llama a otro constructor
  }//NO se pone new
  Punto (double x, double y) {
  }
}
```

```
class Pixel extends Punto {
  private String color;
  public Pixel (doubl x, double y,
                String color) {
    super (x, y);               // un super dentro de un constructor debe estar el primero
    this.color = color;
  }
```

- Up-casting (Abstracción)




```
Pixel px = new Pixel (—);
Punto p = px;
Object o = p;
```



- Down-casting (Detallado)

```
((Pixel) o).setColor ("RED");
```
No es seguro!!

cuando en una jerarquía de herencia hay varias clases con métodos con la misma cabecera, el último sombrea a los superiores.

```
class Object {
    public string toString() {
}
class Punto {
    public string toString() {
        return "(" + this.x + ", " + this.y + ")";
    }
}
class Pixel extends Punto {
    public string toString() {
        return this.color;
    }
}
```

```
Punto p = new Punto (32.6, -6.2);
System.out.println (p.toString());
```

Sombreado: misma cabecera de método público en dos métodos de dos clases que heredan.

```
Pixel px = new Pixel (3.2, -6.2, "RED");
System.out.println (px.toString()); → imprime RED
System.out.println (((Punto)px).toString()); → imprime RED
```

Sombreado: siempre se quedará con el toString del Pixel, pues está más abajo.

todo igual

```
class Object
class Punto
class Pixel extends Punto {
    public string toString() {
        return super.toString() + " " + this.color;
    }
}
```

```
System.out.println (px.toString); → imprime (3.2, -6.2) RED
```

```
class Acceso {
   private String nombre;
   private String contraseña;
   public boolean acceder (String n, String pwd) {
      return this.nombre.equals(n) && this.contraseña.
   {          equals(pwd);
}
```

con String NO se puede usar == para comparar, pues son referencias y sólo sería true si las dos refes. apuntan al mismo objeto

```
class Hacker extends Acceso {
   public boolean acceder (String n, String pwd) {
      return true;
   {
}
```

si ponemos { "final boolean acceder(—){}" ó "final class Acceso {}" } impide la herencia y ninguna clase puede sombrear ningún método ⇒ herencia prohibida

Obligación de herencia: clase abstracta que se ha escrito con el objetivo de extenderla posteriormente.

© Herencia forzada

```
abstract class Movil {
   private Punto p;
   abstract public void mover(Punto p);
   public Movil (Punto p) {
      this.p = p;
   {
```

← marcarlo en la cabecera

ponerla aquí → obliga ponerla

cabecera sin cuerpo

en una clase abstracta no se pueden crear objetos directamente ~~new Movil(p);~~ pero sí referencias.

Tema 8: Colecciones

1. Concepto
2. Lista
3. Conjunto
4. Clases auxiliares
5. Repaso arrays

lo tiene el compiler

```
interface List<E>{                    ← lista genérica
    public boolean add(E elemento);   ← añadir un elemento al final
    public void add(int posicion, E elemento);
    public void clear();
        boolean contains(E elemento);
        E get(int posicion);
        int indexOf(E elemento);
        boolean isEmpty();
        E remove(int posicion);
        boolean remove(E elemento);
        E set(int posicion, E elemento);
        int size();
}
```

Arrays: no persistencia, capacidad limitada, alta velocidad y contenidos homogéneos.

Listas: tipo de colecciones con capacidad ilimitada, velocidad media de acceso, no persistencia y contenidos homogéneos.

```
List<Punto> lp = new ArrayList<Punto>();
lp.add(new Punto(6,3));
lp.add(new Punto(0,0));
   :
for(Punto p : lp){
   System.out.println(p);
}
```

Tema 8.2: Lista

Para usar una lista es necesario importar la clase predefinida.

`import java.util.*;` antes de class {...}

```
List<Mensaje> l = new List<Mensaje>();
```

Erróneo: List es una interfaz y NO se pueden crear elementos de una interfaz

Correcto:
```
List<Mensaje> l = new ArrayList<Mensaje>();
l.add(new Mensaje());
```

```
for (Mensaje m : l) {
    System.out.println(m.toString());
}
```
← Para cada mensaje m de la lista l hacer (imprimir).

Estrategias de uso:

→ FIFO / cola
```
add(Mensaje)
if (!isEmpty()){
    remove(0);
{
```

→ LIFO / pila
```
add(0,Mensaje)
if (!isEmpty()){
    remove(0);
{
```

Tema 8.3: Conjunto

(también pertenece a java.util.*)

```
interface Set<E> {
    public boolean add(E e);
    public void clear();
    public boolean contains(E e);
    public boolean isEmpty();
    public boolean remove(E e);
    public int size();
}
```

```
Set<Punto> s = new HashSet<Punto>();
```

Tema 8.4: Clases auxiliares

```
class Arrays {   ← también está en java.util
    public static int binarySearch (Object [] a,
                                    Object k);
    public static void sort (Object [] a, int first, int to);
}
```

para obtener el array del ser parámetro en las clases List y Set existen:

```
public Object[] toArray();
```

Tema 8.5: Repaso Arrays

(omitido por su "complejidad")

```
import java.util.*;
import java.io.*;

public class Correo {
    private Buzon entrada;
    private Buzon salida;
    private Buzon spam;
    private Buzon valido;
    private List<Regla> reglas;

    public Correo () {
      entrada = new Buzon("entrada");
      salida = new Buzon ("salida");
      spam = new Buzon ("spam");
      valido = new Buzon ("valido");
      reglas = new ArrayList<Regla>();
    }

    public void iniciarReglas () {
        // spam
        reglas.add (new Regla (new ComprobacionRemitente("remite2"), new Almacenar (spam)));
        // sinclasificar
        reglas.add (new Regla (new ComprobacionRemitente("remite0"), new Almacenar (valido)));
        // salida
        reglas.add (new Regla (new ComprobacionRemitente("remite3"), new Almacenar (salida)));
        // duplicacion
        AccionCompuesta a = new AccionCompuesta();
        a.addAccion (new Almacenar (salida));
        a.addAccion (new Almacenar (valido));
        reglas.add (new Regla (new ComprobacionRemitente("remite8"), a));
        // sinclasificar
        reglas.add (new Regla (new ValeSiempre(), new Almacenar (valido)));
    }
    public void iniciarEntrada () {
// FIRST OPTION:
// crear un archivo msg.txt con varios mensajes y
// ejecutar el siguiente try-catch y NO ejecutar el for siguiente (que aparece comentado)
        try {
            BufferedReader f = new BufferedReader(new FileReader(System.getProperty("user.dir") + File.separatorChar + "msg.txt"));
            System.out.println (System.getProperty("user.dir"));
            String s = f.readLine();
            while (s != null) {
                StringTokenizer st = new StringTokenizer (s, ":");
                String rem = st.nextToken();
                String dest = st.nextToken();
                String head = st.nextToken();
                String body = st.nextToken();
                String doc = st.nextToken();
                String urg = st.nextToken();
                boolean urgente = false;
                if (urg.equals ("true"))
                    urgente = true;
                entrada.addMensaje(new Mensaje (rem, dest, head, body, doc, urgente));
                s = f.readLine();
            }
        }
        catch (Exception e) {
            System.out.println ("Error en la lectura de fichero");
        }

// SECOND OPTION:
// no crear ningún archivo
// comentar el try-catch superior y quitar el comentario del siguiente bucle for

                /*
        for (int i = 0; i < 10; i ++) {
           entrada.addMensaje(new Mensaje( "remite" + i, "dest" + i,
                                                                    "head" + i,     "body" + i,
                                                                    "doc" + i, false));
        }
                */
    }

    public void procesar () {
        Mensaje m = entrada.removePrimero();
        while (m != null) {
            for (Regla r: reglas) {
                if (r.cumple(m)) {
                    r.accion (m);
                    break;
                }
            }
            m = entrada.removePrimero();
        }
    }

    public static void main(String[] args) {
        Correo c = new Correo ();
        c.iniciarReglas();
        c.iniciarEntrada();
        System.out.println ("Entrada" + c.entrada);
        c.procesar();
        System.out.println ("Entrada" + c.entrada);
        System.out.println ("Salida" + c.salida);
        System.out.println ("Spam" + c.spam);
        System.out.println ("Valido" + c.valido);
    }
}
```

```
public class Mensaje {
    private String remitente;
    private String destinatario;
    private String cabecera;
    private String cuerpo;
    private String documento;
    private boolean urgente;


    public Mensaje (String remitente, String destinatario, String cabecera,
                    String cuerpo, String documento, boolean urgente) {
        this.remitente = remitente;
        this.destinatario = destinatario;
        this.cabecera = cabecera;
        this.cuerpo = cuerpo;
        this.documento = documento;
        this.urgente = urgente;
    }

    public String getCabecera() {
        return cabecera;
    }

    public String getCuerpo() {
        return cuerpo;
    }

    public String getDestinatario() {
        return destinatario;
    }

    public String getDocumento() {
        return documento;
    }

    public String getRemitente() {
        return remitente;
    }

    public boolean isUrgente() {
        return urgente;
    }

    public Mensaje duplica () {
        return new Mensaje (remitente, destinatario, cabecera,
                    cuerpo, documento, urgente);
    }

    public String toString () {
        return "" + remitente + ":"
                          + destinatario+ ":"
                          + cabecera + ":"
                          + cuerpo + ":"
                          + documento + ":"
                          + urgente + ":";
    }
}
```

```
public class Regla {
    private Comprobacion c;
    private Accion a;

    public Regla (Comprobacion c, Accion a) {
        this.c = c;
        this.a = a;
    }
    public boolean cumple (Mensaje m) {
        return c.cumple(m);
    }
    public void accion (Mensaje m){
        a.accion(m);
    }
}
```

```
// Fichero msg.txt

remite2:para los altos:mensaje de prueba 1:Este deberia ir a Spam:doc.exe:true
remite3:para los bajos:mensaje de prueba 2:Este deberia ir a salida:doc.exe:false
remite0:para los gordos:mensaje de prueba 3:Este deberia ir a valido:doc.exe:false
remite8:para los delgados:mensaje de prueba 4:Y este va a valido y salida:doc.exe:false
deDios:para mi:a valido de cabeza:pues eso, a valido:nada.doc:false
```

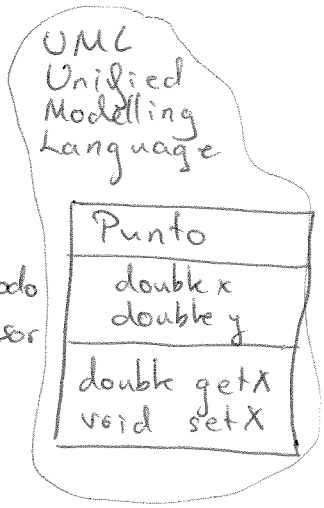
```
import java.util.*;

public class Buzon {

    private List<Mensaje> mensajes;
    private String nombre;

    public Buzon (String nombre) {
      this.nombre = nombre;
      mensajes = new ArrayList<Mensaje> ();
    }

    public String getNombre () {
        return nombre;
    }
    public void addMensaje (Mensaje m) {
      mensajes.add(m);
    }

    public Mensaje getPrimero () {
      return mensajes.get(0);
    }

    public Mensaje removePrimero () {
        if (mensajes.size() == 0)
            return null;
      Mensaje m = mensajes.get(0);
      mensajes.remove(0);
      return m;
    }

    public String toString () {
        String res = "";
        for (Mensaje m: mensajes)
            res += "\n" + m;
        return res;
    }
}
```

```
interface Accion {

// solucion de unas 3 lineas aprox
// contiene solo una cabecera de metodo publico que devuelve void
// el metodo se llama accion y toma como parametro un mensaje

        public void accion (Mensaje m);

}

------------
------------

import java.util.*;

class AccionCompuesta implements Accion {
// solucion de unas 15 lineas aprox

// tiene un atributo que es una lista de acciones (List<Accion>)
        private List<Accion> acciones;

// que debe inicializarse en el constructor como ArrayList<Accion>

        public AccionCompuesta () {
                acciones = new ArrayList<Accion>();
        }

// metodo para añadir acciones a la lista de acciones

        public void addAccion (Accion a) {
                acciones.add(a);
        }

// metodo accion con un mensaje como parametro
// este metodo recorre la lista de acciones y para cada una de ellas
// llama a su metodo accion pasandole el mensaje como parametro

        public void accion (Mensaje m) {
                for (int i = 0; i < acciones.size(); i++) {
                        acciones.get(i).accion(m);
                }
        }
}

----------
----------

class Almacenar implements Accion {
// solucion de unas 12 lineas aprox

// tiene un atributo que es referencia a un buzon

private Buzon bz;

// esta referencia se le pasa como parametro en el constructor

        public Almacenar (Buzon buz) {
                this.bz = buz;
        }

// la clase almacena el mensaje que se le pasa como parametro en su buzon

        public void accion (Mensaje m) {
                bz.addMensaje(m);
        }

}
```

```
interface Comprobacion {

// solucion de unas 3 lineas aprox
// tiene solo una cabecera de metodo publico (devuleve boolean)
// el metodo se llama cumple y toma como parametro un mensaje

        public boolean cumple (Mensaje m);

}

---------
---------

class ValeSiempre implements Comprobacion {

// solucion de unas 5 lineas aprox
// el metodo cumple de esta clase vale siempre true

        public boolean cumple (Mensaje m) {
                return true;
        }

}

---------
---------

class ComprobacionRemitente implements Comprobacion {

// solucion de unas 9 lineas aprox
// atributo de tipo String (cuyo valor se le pasa en el constructor)
// que es el nombre del remitente que buscamos
// El metodo cumple mira a ver si este atributo de tipo String
// es el mismo que el remitente del mensaje que le pasan como parametro
// La comprobacion la hace pasando a minusculas (con el metodo
// toLowerCase de String) las dos cadenas de caracteres.

private String s;

        public ComprobacionRemitente (String rmtte) {
                this.s = rmtte;
        }

        public boolean cumple (Mensaje m) {
                return (this.s.toLowerCase()).equals(m.getRemitente().toLowerCase());
        }

}
```

```
class Punto {

    private double x;
    private double y;
    public double getX () {
        return this.x;
    }
    public void setX (double x) {
        this.x = x;
    }
}
```

método accesor



28-oct-2009

ejercicio 14. (web)
Definir clase para representar el estado de una bombilla (encendida o apagado). A su vez, defina dos métodos para cambiar el estado.

```
class Bombilla {
  //atributos cuantos menos mejor
private boolean encendida;
  // métodos
public Bombilla (boolean a) {
    this.encendida = a;
}
public void on () {
    this.encendida = true;
}
public void off () {
    this.encendida = false;
}
public boolean encendida () {
    return this.encendida;
}
}
```

que sean mínimos

private: sólo se puede usar en esta clase

método constructor → crea y da valor inicial

a sólo la puedes usar dentro de este método

el atributo encendida toma el valor a

void = no devuelve nada

como no necesita información para encenderse (está vacío)

this = el objeto en el que estamos

como estamos fuera de Bombilla solo podemos usar el constructor y metodos Públicos

```
class PruebaBombilla {
    public static void main (String[] args){
        Bombilla b = new Bombilla (false);
        b.on();
        System.out.println (b.encendida());
    }
}
```

crear bombilla apagada →

ej: Dada una variable entera año escribir una sentencia lógica que nos diga si ese año es bisiesto

```
int año =      ;
boolean esBisiesto = ((año % 4)==0);
```

un año es bisiesto si es divisible por 4 excepto si es divisible por 100 que también debe ser por 400

```
int año =      ;
boolean esBisiesto = (((año % 4) == 0) || (((año % 100)==0)
                      && ((año%400) == 0)));
```