

**Fundamentos de Programación**  
*Versión 0.4*  
*ETSI de Telecomunicación*  
*Apuntes Grupo 11.1*

*Luciano Rubio Romero*



### Reconocimiento-CompartirIgual 2.5 España

Usted es libre de:

- *copiar, distribuir y comunicar públicamente la obra*
- *hacer obras derivadas*

Bajo las condiciones siguientes:



**Reconocimiento.** *Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador.*



**Compartir bajo la misma licencia.** *Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.*



**No comercial.** *No puede utilizar esta obra para fines comerciales.*

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor

**Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.**

### Nota del autor:

El siguiente documento liberado con licencia **Creative Commons** corresponde a los apuntes del lenguaje *Java* que he tomado en la asignatura de *Fundamentos de Programación* impartida por *Juan Carlos Dueñas López* en la ETSI de Telecomunicación, Universidad Politécnica de Madrid (UPM). Al igual que con todos los ejercicios que he hecho o entregado, liberados bajo licencia *Gnu GPL* o similar, y manteniendo el deseo de que todos podamos acceder al conocimiento libre y gratuito, considérese esta breve documentación sobre la programación en *Java* material base o de apoyo de libre difusión y utilícese para cualquier otro uso, siempre y cuando no esté desacorde con la licencia citada con anterioridad. **Si vas a imprimir estos apuntes** no te olvides de esta hoja, pues es la que permite que más gente pueda darles uso.

Así pues siéntete **libre** de leer y compartir estos apuntes. Probablemente en un futuro se incluyan más ejemplos, el resto de temas, se irán añadiendo aclaraciones y se corregirán erratas. Agradeceré cualquier tipo de sugerencia, aviso de código con algún error o cualquier otra cosa que se te ocurra acerca del presente documento, incluso si te ha servido para algo, como por ejemplo aprobar la asignatura ;-)

Si algo está mal escrito, no compila, quieres enviar una sugerencia o ampliar estos apuntes, escribe a [lucianopre@gmail.com](mailto:lucianopre@gmail.com) . Si además quieres saber algo más de mí entra en <http://www.prepenumbra.tk>

# 1.Introducción básica

## 1.1.Definiciones:

*Informática:* Ciencia que lleva el tratamiento informático de la información, pudiéndose la:

- *Transmisión*
- *Almacenamiento*
- *Transformación*
- *Adquisición*
- *Presentación*

*Ordenador:* máquina programable, capaz de ejecutar programas almacenados.

*Algoritmo:* descripción de una solución a un problema. Serie ordenada y finita de instrucciones simples y elementales con el fin de producir unos resultados, trabajando sobre unos datos de entrada.

*Datos:* Conjunto de valores con propiedades y operaciones.

*Programa:* realización de un algoritmo en un ordenador.

## 1.2.Lenguaje de programación:

Los lenguajes de programación permiten escribir algoritmos para poder ser ejecutados en un ordenador. Han de ser simples, concisos y precisos.

**Lenguajes interpretados:** son aquellos que necesitan de un intérprete en cada ejecución del programa para convertir el lenguaje en otro comprensible para el ordenador.

**Lenguajes compilados:** son aquellos que necesitan un compilador, es decir, un programa intermedio que transforme dicho lenguaje en otro que comprenda el ordenador, y que una vez compilado el programa será ejecutable sin ningún tipo de intérprete.

El lenguaje de programación *Java* es una mezcla de ambos, puesto que los programas que escribamos han de compilarse (*javac*, se verá más adelante), y aún así necesitarán un intérprete (*java*).

## 1.3.Desarrollo:

Pasos para la creación del *programa*:

- *Definir* el problema
- *Diseñar* la solución
- *Implementar*
  - Editar
  - Compilar
  - Ejecutar.
- *Comprobar que soluciona el problema.*



# 2.Introducción a los objetos en Java

## 2.1.Clases:

*Clase:* representación de un concepto del mundo real o de la solución a un problema. Se trata de un conjunto de datos con propiedades y operaciones.

- *HolaATodos.java: programa básico que imprime un mensaje por pantalla.*

```
1  class HolaATodos {
2      public static void main(String[] args) {
3          System.out.println("Hola a todos");          //imprimimos por pantalla el mensaje
4      }
5  }
```

Proceso de compilación:

```
$ javac HolaATodos.java
$ java HolaATodos
```

- *Punto.java: programa para definir un punto del plano, sus dos coordenadas y definir una función que calcule la distancia de dicho punto al origen. Las funciones van seguidas de unos paréntesis (lo que las diferencia de las variables) y dentro de ella invocamos la operación raíz cuadrada que se encuentra dentro de la clase Math.*

```
1  class Punto {
2      double x;          //double = número real
3      double y;          //definimos x e y
4      double DistanciaAlCero(){
5          return Math.sqrt((x*x)+(y*y));          //creamos la función
6      }          //llamamos a la función raíz cuadrada
7  }
```

## 2.2.Objetos:

*Objeto:* representación en la memoria de un elemento de una clase.

- *PruebaPunto.java: el objetivo es crear objetos a partir de la clase Punto. Se crea un objeto “p”, asignamos valores a sus coordenadas “x” e “y” e imprimimos por pantalla el dato con la distancia al origen. Finalmente destruimos el objeto p.*

```
1  class PruebaPunto {
2      public static void main (String[] args) {
3          Punto p = new Punto();          //new = creación
4          p.x = 10.0;          //hacemos que la coordenada x sea 10
5          p.y = p.x;          //damos a y mismo valor que x
6          double d = p.distanciaAlCero();
7          System.out.println(d);          //imprimos por pantalla la función
8          p = null;          //destruimos el objeto p
9      }
10 }
```

## 2.3.Referencias:

*Referencia:* nombre a través del cual se puede manejar un objeto.

- *DosPuntos.java: creación de dos puntos y ejemplo de como orientarlos a distintos sitios.*

```
1  class DosPuntos {
2      public static void main (String[] args) {
3          Punto p = new Punto();
4          Punto q = new Punto();          //creamos un nuevo punto q
5          p.x = Math.sqrt(Math.PI);          //la x de p valdrá la raíz de pi
6          q.y = p.x*10.0;          //la y de q es diez veces la x de p
7          p = q;          //p ahora posee los mismos datos de q
8      }
9  }
```

9        }

## 2.4. Clases predefinidas:

Las clases predefinidas son aquellas que no hacen falta crear porque ya existen, si bien en las bibliotecas del *Java Development Kit* (JDK) pudiendo ser esenciales, caso de *Math* o *String*, o bien otras bibliotecas que podrán descargarse de internet.

2.4.1. La clase *Math*: es la más utilizada de todas las esenciales, ya que contiene constantes (como *pi* o el número *e*), funciones trigonométricas (*sin*, *cos*, *asin*, *acos*, *atan...*) además de sistemas de redondeo de números reales (*floor* redondea a la baja, *ceil* a la alta y *round* sigue un convenio más general) y funciones igualmente utilizadas (*log* la logarítmica, *exp* la exponencial, *sqr* la raíz cuadrada, *max* y *min* entre dos números, *pow* para potencias, y muchas más...).

➤ *Cadenas.java*: programa que realiza ejemplos con *Strings* (cadena de texto).

```

1      class Cadenas {
2          public static void main(String[] args) {
3              String s = new String("HolaATodos");
4                      // esta declaración abreviada será: String s = "HolaATodos";
5              System.out.println(s);
6              System.out.println(s + "amigos");
7                      //concatenación, el resultado será: HolaATodosamigos
8              s+="alumnos";           // s será ahora: HolaATodosalumnos
9          }
10     }
```

Algunas funciones utilizadas con cadenas aparte de la concatenación:

*s.length()*; obtiene como resultado el número de letras de la cadena.

*s.charAt(0)*; obtiene de la cadena la letra que ocupa la posición 0.

*s1.equals(s2)*; compara las cadenas *s1* y *s2* y devuelve verdadero si son exactamente iguales.

*s.toLowerCase()*; convierte la cadena de texto a minúsculas.

*s.toUpperCase()*; convierte la cadena de texto a mayúsculas.

➤ *PolinomioGrado2.java*: (este fichero se pide como ejercicio para entrega). Resuelve las raíces reales de polinomios de segundo grado.

```

1      class PolinomioGrado2 {
2          double A;
3          double B;
4          double C;
5          void setA(double a) {
6              A = a;
7          }
8          void setB(double b) {
9              B = b;
10         }
11         void setC(double c) {
12             C = c;
13         }
14         double getA() {
15             return A;
16         }
17         double getB() {
18             return B;
19         }
20         double getC() {
21             return C;
22         }
23         double discriminante() {
24             return (B*B - 4*A*C);
25         }
26         double raiz1() {
27             return (-B + Math.sqrt(discriminante())) / (2*A);
28         }
29         double raiz2() {
30             return (-B - Math.sqrt(discriminante())) / (2*A);
31         }
32     }
```

```
31     }
32 }
```

- *PruebaPolinomioGrado2.java*: este programa crea un objeto de la clase *PolinomioGrado2*, le asigna unos parámetros *a*, *b*, *c* y obtiene las soluciones a través de los métodos *raiz()* y *raiz2()*.

```
1  class PruebaPolinomioGrado2 {
2      public static void main(String[] args) {
3          PolinomioGrado2 pg2 = new PolinomioGrado2();
4          pg2.setA(1.0);
5          pg2.setB(2.0);
6          pg2.setC(1.0);
7          System.out.println("Polinomio "+pg2.getA()+" "+pg2.getB()+" "+pg2.getC());
8          System.out.println(pg2.discriminante());
9          System.out.println(pg2.raiz()+" "+pg2.raiz2());
10     }
11 }
```

- *Punto.java (v.2)*: clase *Punto* añadiéndole métodos accesoros y modificadores para coordenadas *x* e *y*.

```
1  class Punto {
2      double x;
3      double y;
4      void setX(double a) {
5          x = a;
6      }
7      void setY(double b) {
8          y = b;
9      }
10     double getX() {
11         return x;
12     }
13     double getY() {
14         return y;
15     }
16     double distanciaAlCero() {
17         return Math.sqrt((x*x) + (y*y));
18     }
19 }
```

- *PruebaPunto.java (v.2)*: clase *PruebaPunto* haciendo uso de los nuevos métodos de la clase *Punto*.

```
1  class PruebaPunto {
2      public static void main(String[] args) {
3          Punto p = new Punto();
4          p.setX(15.0);
5          p.setY(-4.2);
6          System.out.println("Punto "+p.getX()+" "+p.getY());
7          System.out.println(p.distanciaAlCero());
8      }
9  }
```





## 3. Datos simples y operaciones básicas

Los programas en general están organizados en dos partes: cómo escribir un algoritmo y cómo manejamos los datos, y luego el número de datos, sus nombres, valores, tipos...

### 3.1. Tipos simples o primitivos:

Se trata de los tipos más básicos, los que no hace falta inventar. Un tipo de dato además define un álgebra: un conjunto de datos con indicación de qué valores pertenecen al conjunto y las operaciones que puedan hacerse con éstos.

*Definición por extensión:* grupos concretos y definidos.

*Definición intensional:* alguna propiedad (ejemplo: los enteros no tienen parte decimal).

#### Características de los tipos primitivos:

Se pueden crear variables. Ej: `int numeroAlumnos;`  
 Asignación: se puede cambiar el valor a la variable. Ej: `numeroAlumnos = 110;`  
 Comprobación de igualdad: Ej: `numeroAlumnos == 126;`  
 Comprobación de diferencia: Ej: `numeroAlumnos != 115;`

### 3.2. Dato de tipo lógico o booleano:

Este tipo sólo puede albergar dos valores: true o false. Son el resultado de las comparaciones.

```
boolean estaLloviendo;  
estaLloviendo = (conductividad > 1832.0);
```

Esta línea asignará verdadero o falso a la variable *estaLloviendo* según el valor de la conductividad.

#### Operaciones con booleanos:

Operadores lógicos:

Binarios: `&&` (and), `||` (or), `^` (xor)

Unario: `!` (not)

Comparaciones: `==` (igualdad), `!=` (contrario), `>` (mayor), `>=` (mayor-igual), `<` (menor), `<=` (menor-igual)

```
boolean estaElProfesor;  
int numeroAlumnos;  
boolean hayClase;  
hayClase = ((estaElProfesor) && (numeroAlumnos > 5));
```

### 3.3. Dato de tipo enumerado:

Definición de un conjunto de pocos valores.

```
enum colorSemaforo {ROJO, AMARILLO, VERDE};  
colorSemaforo semaforoParaninfo = colorSemaforo.ROJO;  
semaforoParaninfo = colorSemaforo.VERDE;  
colorSemaforo.ROJO.toString();  
colorSemaforo.VERDE.ordinal(); //devuelve 2
```

### 3.3. Dato de tipo caracter:

Este tipo guarda una sola letra que bien puede ser del alfabeto (mayúsculas y minúsculas), dígitos, signos de puntuación... El orden impuesto, según tabla ASCII, es:  $A < B < C < \dots < Z < a < b < c < \dots < z < 1 < 2 \dots$

```
isLowerCase(char c)    //comprueba si el caracter está en minúscula  
isUpperCase(char c)    //comprueba si el caracter está en mayúscula  
toLowerCase(char c)    //transforma el caracter a minúscula  
toUpperCase(char c)    //transforma el carácter a mayúscula.
```

```
char letra = 'S'; // atención, se define la letra con una sola comilla.
letra = toUpperCase(letra);
java.lang.Character.toUpperCase(letra); // misma función que arriba.
```

### 3.4.Dato de tipo entero:

Hay cuatro subtipos de enteros según capacidad en memoria: byte (8 bits), short (16 bits), int (32 bits) y long (64 bits). Si tenemos un número de muchas cifras y le asignamos un entero escaso, se produce overflow.

```
int numeroAlumnos = 46; // forma decimal
int numeroAlumnos = 0xFF // forma hexadecimal
```

#### Operaciones:

Suma:  $3 + 4 = 7$   
 Resta:  $3 - 4 = -1$   
 Multiplicación:  $3 * 4 = 12$   
 División:  $3 / 4 = 0$ . Devuelve el entero inferior.  
 Módulo:  $3 \% 4 = 3$ . Resto de la división  
 Incremento:  $n++$ . Equivalente a  $n = n + 1$   
 Decremento:  $n--$ . Equivalente a  $n = n - 1$

```
System.out.println(6++); //da error, el incremento solo es para variables
int n = 6;
// Caso 1: postincremento
System.out.println(n++); //imprime 6, el incremento se hace a continuación
// Caso 2: preincremento
System.out.println(++n); //imprime 7, primero incrementa después imprime
```

### 3.4.Dato de tipo real:

Son los números con decimales, los que usamos cotidianamente. Existen dos subtipos: float (32 bits) y double (64 bits). También se puede producir el overflow.

#### Operaciones:

Suma:  $3.0 + 4.0 = 7.0$   
 Resta:  $3.0 - 4.0 = -1.0$   
 Multiplicación:  $3.0 * 4.0 = 12.0$   
 División:  $3.0 / 4.0 = 0.75$

➤ *esBisiesto.java*: programa que determina a través del parámetro año si éste es bisiesto o no.

```
1 class esBisiesto {
2     boolean esBisiesto(int año) {
3         return ((año%4==0)|| (año%100==0 && año%400==0));
4     }
5 }
```

### 3.5.Conversiones:

```
//Ejemplo introductorio
int a = 3.0 * 6; //devuelve 18.0, no se puede guardar en entero
```

#### Implícitas o transparentes:

El programador no se da cuenta, no es alertado por el compilador.  
 double d = 3; //válido

#### Explícitas o casting:

El programador convierte a voluntad de un tipo de variable a otro.  
 double d = 3.6; //double bien introducido. correcto  
 byte b = (byte)d; //conversión del double d a byte  
 boolean b = (boolean)0; //incorrecto, tipos incompatibles.

```
char c = (char)132;           //convierte el número 132 a carácter (usará la tabla ASCII)
int i = (int)'a';             //convierte el carácter 'a' en número (usará la tabla ASCII)
```

### Tipos primitivos y sus clases envoltorios:

Son las clases Byte, Short, Integer, Long, Double, Boolean y Char.

```
Integer.parseInt("345");      //devuelve el entero 345
Double.parseDouble("3A");     //se lanza una excepción, error.
Integer i = new Integer(10);  //forma alternativa para crear entero
Integer i = 10;               //forma válida a partir del jdk 1.5
int j = i;                    //boxing o encajonamiento
```

## 3.6.Constantes:

Actúan de la misma forma que un atributo en un método, pero es inalterable. Se denota con *final* y suelen tener nombres en mayúsculas (ej: PI, E...)

```
final double DISTANCIA_MINIMA = 1E-4;
```

- *Prestamo.java* (este fichero se pide como fichero para entregar. Puntúa): programa que calculará la cuota mensual que ha de pagarse al banco tras haber pedido un préstamo, determinando la cantidad, el interés anual que nos imponen y el número de cuotas en el que decidimos devolverlo.

```
1  class Prestamo {
2      double cantidad;
3      double interesAnual;
4      int numeroCuotas;
5
6      //métodos modificadores
7      void setCantidad (double x) {
8          cantidad = x;
9      }
10     void setInteresAnual (double y) {
11         interesAnual = y;
12     }
13     void setNumeroCuotas (int z) {
14         numeroCuotas = z;
15     }
16     double cuotaMensual() {
17         return (cantidad * (interesAnual / 1200)) / (1 - (1 / (Math.pow(1 +
18             (interesAnual / 1200), numeroCuotas))));
19     }
20 }
```

- *PruebaPrestamo.java*: programa que prueba la clase préstamo introduciendo los datos por línea de comandos.

```
1  import java.io.*;
2  class PruebaPrestamo {
3      public static void main (String[] args) throws Exception {
4          BufferedReader teclado = new BufferedReader (new InputStreamReader(System.in));
5          Prestamo p = new Prestamo();
6          System.out.println("Introduzca la cantidad del préstamo: ");
7          p.setCantidad(Double.parseDouble(teclado.readLine()));
8          System.out.println("Introduzca el interes del préstamo: ");
9          p.setInteresAnual(Double.parseDouble(teclado.readLine()));
10         System.out.println("Introduzca el numero de cuotas del préstamo: ");
11         p.setNumeroCuotas(Integer.parseInt(teclado.readLine()));
12         System.out.println("Cantidad a pagar en cada cuota: "+p.cuotaMensual());
13     }
14 }
```



# 4. Métodos

## 4.1. Concepto de método:

*Definición:* es una función en la cual se realizan operaciones y se devuelve un resultado. Son un conjunto de sentencias que serán ejecutadas cuando se invoque al método. Éstos pueden llamarse muchas veces, y pueden devolver resultado o no (ejemplos: métodos *set* y *get*, usados en la clase *Punto*).

## 4.2. Uso:

- Sólo se pueden escribir métodos dentro de una clase.
- El orden de los métodos es indiferente.
- Sólo se pueden escribir sentencias dentro del cuerpo del método

➤ *Punto.java (v.3): se añaden los métodos para calcular el ángulo que forma la recta origen-punto con el eje x, y otro para calcular la distancia entre dos puntos.*

```
1      class Punto {
2          double x;
3          double y;
4          void setX(double a) {          //método modificador de la x
5              x = a;
6          }
7          void setY(double b) {
8              y = b;
9          }
10         double getX() {                //método accesor de la x
11             return x;
12         }
13         double getY() {
14             return y;
15         }
16         double distanciaAlCero() {
17             return Math.sqrt((x*x) + (y*y));
18         }
19         double angulo() {
20             return Math.atan2(this.getY(),this.getX());
21             //Se incluye this para usar métodos o atributos de la misma clase
22             //La sentencia return es obligatoria para métodos no-void
23             //El tipo de devolución ha de ser el mismo que el de la cabecera del método.
24         }
25         double distancia(Punto p) throws Exception {
26             if (p == null)              //estructura condicional. Se dará en profundidad en el siguiente tema
27                 throw new Exception ("Falta punto");
28             double distancia = Math.sqrt(((this.getX() - p.getX())*(this.getX() - p.getX())) +
29                 ((this.getY() - p.getY())*(this.getY() - p.getY())));
30             return distancia;
31         }
32     }
```

➤ *PruebaPunto.java (v.3): programa con el que se prueba el nuevo método para calcular distancias entre puntos.*

```
1      class PruebaPunto {
2          public static void main(String[] args) {
3              Punto p1 = new Punto();
4              p1.setX(34.2);
5              p1.setY(81.2);
6              Punto p2 = new Punto();
7              p2.setX(-39.2);
8              p2.setY(-110.0);
9              System.out.println(p1.distancia(p2));
10         }
11     }
```

## 4.3. Clasificación:

**Constructores:** permiten dar un valor inicial a cada atributo cuando se crea un objeto. Han de llamarse igual que la clase en la que están, no tienen tipo de devolución y sólo se pueden llamar con *new*.

```
class Punto {  
    ...  
    Punto (double x, double y) {          //ejemplo 1 de constructor. Podemos asignar una x e y  
        this.x = x;  
        this.y = y;  
    }  
    Punto () {                            //ejemplo 2 de constructor. Actúa como si no hubiera constructor  
        this.x = 0.0;  
        this.y = 0.0;  
    }  
}
```

Y al crear el punto, invocamos de la siguiente manera:

```
Punto p1 = new Punto(3.2, 8.6);  
Punto p2 = new Punto();
```

**Accesores:** devuelven el valor de una variable.

```
double getX() {          //usaremos la nomenclatura getAtributo()  
    return this.x;  
}
```

**Modificadores:** asignan un nuevo valor a la variable

```
void setCantidad(double cantidad) {          //usaremos la nomenclatura setAtributo()  
    this.cantidad = cantidad;  
}
```

**Otros:**

```
//método toString. Transforma valores, atributos o números a cadena de texto.  
String toString() {  
    return "("+this.getX()+","+this.getY()+")";  
}  
  
//método equals. Compara puntos.  
boolean equals(Punto p) {  
    return ((this.getX() == p.getX()) && (this.getY() == p.getY()));  
}  
Punto p = new Punto(0, 1);  
Punto q = new Punto(0, 2);  
System.out.println(p.equals(q));          //imprime falso.
```

**Estáticos:** accesibles desde otras clases, sin necesidad de crear objeto.

```
static double distancia(Punto p, Punto q) {  
    return ...  
}
```

# 5. Estructuras de control

## 5.1. Sentencias

Cualquier tipo de línea en java. Sólo se pueden escribir sentencias en el cuerpo de métodos.

### **Sentencias simples:**

`;` nula, declara fin de línea.  
`int i;` declaración de variable.  
`i = 10;` asignación.  
`new Punto();` creación de objeto.  
`System.out.println("...");` llamada a un método.  
Sentencias de control

### **Sentencias compuestas (bloque):**

```
{  
sentencia1;           //se pueden crear variables, que sólo existirán en el bloque.  
sentencia2;  
...  
}
```

## 5.2. Condicionales

Ejecución de un bloque de código, que está condicionado al valor de una expresión booleana.

### **If-else:**

```
if (condición) {  
    sentencias;  
} else {  
    sentencias;  
}  
  
int mes;    //suponemos que se da valor numérico al mes  
  
void imprimeMes(int m) {  
    if ((mes < 1) || (mes > 12)) {  
        System.out.println("no válido");  
    }  
    else if (mes == 1) {  
        System.out.println("Enero");  
    }  
    else if (mes == 2) {  
        System.out.println("Febrero");  
    }  
    ...  
}
```

### **Switch-case:**

```
switch(mes) {  
    case 1: nombre = "enero";           //mes puede ser booleano, char o int.  
        break;                         //el case debe ser un valor, no una condición.  
    case 2: nombre = "febrero";  
        break;  
    ...  
    default: nombre = " ";             //La opción predeterminada, si existe, se escribe al final.  
}
```

## 5.3. Bucles

### **While:**

```
while (condición) {  
    operación;  
}
```

➤ *Factorial.java: programa para calcular el factorial de un número n.*

```

1      class Factorial {
2          int factorial(int n) {
3              int fact = 1;
4              while (n > 0) {
5                  fact = fact*n;
6                  n--;
7              }
8              return fact;
9          }
10     }

```

**Do-while:** se ejecuta al menos una vez

```

do {
    operación;
} while (condición);

```

**For:** el más usado, se sabe el número de vueltas

```

for (inicial; condición; actualización) {
    operaciones,
}

//Ejemplo: imprime los 100 primeros números
for (i = 0; i<100; i++) {
    System.out.println(i);
}

```

## 5.4.Saltos

**Definición:** sentencias que rompen el flujo normal de ejecución.

**Break:** Continúa después del bloque donde se encuentra.

```

for (int i = 0; i<=5, i++) {
    if (i == 5) {
        break;                //saldría fuera del bucle for
    }
    System.out.println(i);
}
...

switch(mes) {
    case 1: m = "enero";
        break;                //saldría fuera del switch-case
    case 2: ...
}

```

**Continue:** continúa el ritmo de ejecución del bucle.

```

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

```

**Break y continue etiquetados:** cierran el bucle al que apuntan.

```

//calcular valores i, j y z que satisfacen la ecuación i^2 + j^2 = z^2
bucle: for (int i = 1; i <=100; i++) {
    for (int j=1; j <= 100; j++) {
        for (int z=1; z <= 100; z++) {
            if ((i*i)+(j*j) == (z*z)) {
                System.out.println(i+" "+j+" "+z);
                break bucle;
            }
        }
    }
}
return;
System.exit(0);

```



## 5.5.Excepciones

*Definición:* ruptura del flujo normal de ejecución del problema, normalmente por error de ejecución.

- *Error aritmético:* por ejemplo, división por cero.
- *Lectura de datos:* tipo incorrecto introducido.
- *Conversiones incorrectas:* por ejemplo, mal convertida una String a primitivo.

**Lanzamiento de excepciones:**

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

**Indicación cabecera de método:**

```
int factorial(int n) throws Exception { //indicación de que el método puede lanzar excepciones
    if (n<0)
        throw new Exception("error"); //se lanza una excepción
    int fact = 1;
    for ( ; n >0; n--)
        fact *= n;
    return fact;
}
```

**Capturar excepciones:**

```
public static void main(String[] args) {
    int n = 44;
    try {
        int resultado = Factorial.factorial(n);
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

## 5.6.Recursividad

*Definición:* método que se llama a sí mismo.

```
int factorial (int n) throws Exception {
    if (n < 0) // Condición aplicabilidad
        throw new Exception("...");
    if ((n == 0) || (n == 1)) // Caso simple
        return 1;
    return n*factorial(n-1); // Caso general
}
```

*Ámbito dinámico:* conjunto de datos que maneja un método mientras se ejecuta. Existe solo en la ejecución.

➤ *Fibonacci.java:* programa que calcula la serie de Fibonacci

```
1    class Fibonacci {
2        int fibonacci(int n) throws Exception {
3            if (n < 0)
4                throw new Exception("...");
5            if ((n == 0) || (n == 1))
6                return 1;
7            return fibonacci(n-2)+fibonacci(n-1);
8        }
9    }
```

➤ *SerieAritmetica.java:*

```
1    class SerieAritmetica {
2        double primero;
3        double incremento;
4        SerieAritmetica (double primero, double incremento) {
5            this.primeros = primero;
```

```

6         this.incremento = incremento;
7     }
8     double suma(int n) {
9         double s=0.0;
10        for (int i=1; i <= n; i++) {
11            s+= this.enesimo(n);
12        }
13        return s;
14    }
15    double enesimo (int n) {
16        return this.primer0 + ((n-1)*this.incremento);
17    }
18 }

```

➤ *PiCuartos.java: programa que calcula una aproximación a pi cuartos mediante una serie matemática.*

```

1    class PiCuartos {
2        int signo = 1;
3        double s = 0.0;
4        for (int i=0; i<n; i++) {
5            s+=signo / ((2*i)+1);
6            signo = -signo;
7        }
8        return s;

```

### **Práctica grupo 11: Objetivos**

Esta práctica tiene por objetivo principal aprender a implementar algoritmos sencillos, y comprobar la utilidad del ordenador para realizar cálculos. También el uso de varias clases a la vez, para entender cómo se relacionan. Se cubren los siguientes elementos del lenguaje:

- definición de clases
- creación de objetos y referencias
- tipos primitivos
- sentencias de control para implementar algoritmos sencillos
- creación y uso de métodos
- operaciones matemáticas

### *El problema que se quiere resolver (especificación de requisitos)*

En esta práctica se implementa el algoritmo que permite a un teléfono móvil calcular su posición, si se sabe su distancia a dos puntos de coordenadas conocidas. Estos puntos serán las coordenadas de dos estaciones base (o antenas) que controlan su funcionamiento. El teléfono móvil enviará mensajes a estas estaciones base y, midiendo el tiempo que transcurre hasta que recibe las respuestas y usando un circuito electrónico específico, estimará un valor de la distancia a cada estación base. Estas operaciones dan valores aproximados, así que las probabilidades de error son altas.

En esta práctica sólo debemos realizar el programa capaz de hacer los cálculos para obtener la posición del teléfono móvil, a partir de las posiciones de las antenas, y de la distancia medida.

El problema es bastante complejo de resolver, así que simplificaremos el cálculo de la solución de acuerdo a lo siguiente:

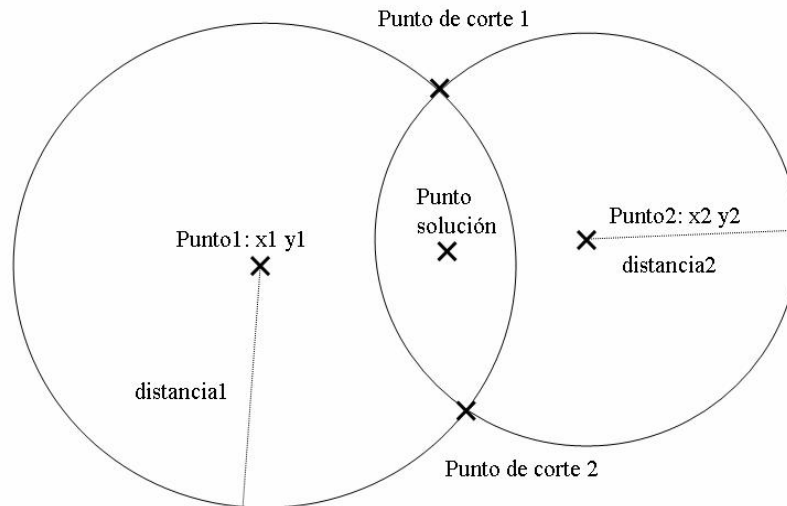
1. sólo se trabajará en el espacio bidimensional.
2. no se calculará exactamente el punto de intersección de las dos circunferencias, sino los puntos de corte de éstas. Puede haber cero puntos de corte (no hay solución al problema), un punto de corte si las circunferencias son tangentes, y dos puntos de corte si son secantes.
3. la solución al problema (la posición del móvil), si existe, es el punto de corte de las circunferencias tangentes, o el punto medio de los dos puntos de corte si son secantes.

Para todo ello, usaremos cuatro clases, aunque sólo hay que hacer cambios en una de ellas:

- la clase Punto, para representar puntos en el espacio bidimensional, y que se proporciona ya hecha.
- la clase PolinomioGrado2, para representar polinomios de segundo grado de los cuales podemos calcular el determinante y raíces reales; posteriormente se muestra cómo la solución al problema se puede formular en términos de ecuaciones de segundo grado; esta clase se proporciona ya hecha.
- la clase Localizador, que se encarga de realizar las comprobaciones y cálculos (usando objetos de clase Punto y PolinomioGrado2), y obtiene la solución. Es la que hay que completar.

- la clase PruebaLocalizador con algunas pruebas del funcionamiento del Localizador. Es recomendable hacer más pruebas cambiando este fichero, pero no hay que entregarlo. cambios en una de ellas las circunferencias tangentes, o el punto medio de los dos puntos de corte si son senc

### El diseño de la solución



La siguiente figura muestra un esquema de la solución que se utilizará en la práctica. En ella, aparecen dos circunferencias con centros y radios conocidos; estas circunferencias se solapan y el teléfono móvil que se pretende localizar se encontrará en el área de intersección de las dos circunferencias.

El algoritmo para localizar al teléfono móvil es el siguiente:

- comprobar que las distancias pueden dar una solución: la distancia entre los puntos 1 y 2 debe ser menor o igual a la suma de las distancias distancia1 más distancia2. En caso de que no se cumpla esta condición, no habrá solución, y habrá que lanzar una excepción,
- obtener el primer punto de corte,
- obtener el segundo punto de corte,
- obtener el punto medio entre los dos puntos de corte. Es el punto cuya coordenada x es la media de las x de los dos puntos de corte, y la coordenada y es la media de las y de los dos puntos de corte.

Este algoritmo funciona correctamente tanto si hay un punto de corte (el primer punto de corte y el segundo punto de corte son el mismo) como si hay dos.

### Cálculo de los puntos de corte de dos circunferencias

Para encontrar los puntos de corte de dos circunferencias con centro  $(x_1, y_1)$   $(x_2, y_2)$ , y con radios  $d_1, d_2$ :

- Si  $x_1 = x_2$ , entonces:

$$(1) y = (d_1^2 - d_2^2 + y_2^2 - y_1^2) / 2 (y_2 - y_1)$$

$$(2) x = x_1 \pm (d_1^2 - (y - y_1)^2)^{1/2}$$

- Por el contrario, si  $y_1 = y_2$ , entonces:

$$(3) x = (d_1^2 - d_2^2 + x_2^2 - x_1^2) / 2 (x_2 - x_1)$$

$$(4) y = y_1 \pm (d_1^2 - (x - x_1)^2)^{1/2}$$

- Y en el caso general:

$$(5) x = A + By$$

Donde:

$$(6) A = (d_1^2 - d_2^2 + x_2^2 - x_1^2 + y_2^2 - y_1^2) / 2 (x_2 - x_1)$$

$$(7) B = (y_1 - y_2) / (x_2 - x_1)$$

$$(8) (B^2 + 1) y^2 + 2 (BA - Bx_1 - y_1) y + (A - x_1)^2 + y_1^2 - d_1^2 = 0$$

Se trata de una ecuación de segundo grado:

- si tiene dos raíces reales, las circunferencias se cortan en dos puntos, son secantes;
- si tiene una raíz real, las circunferencias se cortan en un punto, son tangentes;
- si no tiene raíces reales, las circunferencias no se cortan.

Se llega a estas fórmulas a partir de las ecuaciones de dos circunferencias, restándolas.

### La clase Localizador

Cuando haya leído y comprendido las clases Punto y PolinomioGrado2 que se dan como parte del enunciado, pase a estudiar el esqueleto de la clase Localizador, que se comenta a continuación.

La clase Localizador debe contener obligatoriamente, aparte de los atributos que considere convenientes, los siguientes métodos, que hay que completar:

// constructor con dos parámetros

public Localizador (Punto p1, Punto p2)

// método 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 parámetro signo es +1.0 o -1.0 se calculará uno de los puntos de corte o el otro

// debe lanzarse excepcion si no hay solución

// el cuerpo de este método contiene los cálculos indicados en la sección anterior

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

// método público que devuelve el punto solución si existe

// este método llama al método calculaCorte y calcula el punto medio

public Punto calculaPosicion (double d1, double d2) throws Exception

### La programación o implementación

Estos son los ficheros que puede usar (ponga todos ellos en el directorio de su práctica):

- *Punto.java* contiene la clase Punto completa; no la cambie.

```

1  class Punto {
2      private double x;
3      private double y;
4      public Punto (double x, double y) {
5          this.x = x;
6          this.y = y;
7      }
8      public double getX() {
9          return this.x;
10     }
11     public double getY() {
12         return this.y;
13     }
14     public double distancia (Punto otro) {
15         return Math.sqrt (((this.getX() - otro.getX()) * (this.getX() - otro.getX()))
16                             + ((this.getY() - otro.getY()) * (this.getY() - otro.getY())));

```

```

16     }
17     public String toString () {
18         return "(" + x + "," + y + ")";
19     }
20 }

```

- *PolinomioGrado2.java* contiene la clase PolinomioGrado2; no puede cambiarla.

```

1  class PolinomioGrado2 {
2      private double a;
3      private double b;
4      private double c;
5      public PolinomioGrado2 (double a, double b, double c) {
6          this.a = a;
7          this.b = b;
8          this.c = c;
9      }
10     public double discriminante () {
11         return ((this.b * this.b) - (4.0 * this.a * this.c));
12     }
13     public double raiz1 () throws Exception {
14         if (this.discriminante () < 0)
15             throw new Exception ("PolinomioGrado2 no tiene solución real");
16         return (-this.b + Math.sqrt (this.discriminante())) / (2 * this.a);
17     }
18     public double raiz2 () throws Exception {
19         if (this.discriminante () < 0)
20             throw new Exception ("PolinomioGrado2 no tiene solución real");
21         return (-this.b - Math.sqrt (this.discriminante())) / (2 * this.a);
22     }
23     public double raiz (double signo) throws Exception {
24         if (this.discriminante () < 0)
25             throw new Exception ("PolinomioGrado2 no tiene solución real");
26         return (-this.b + (signo * Math.sqrt (this.discriminante()))) / (2 * this.a);
27     }
28     public String toString () {
29         return "" + a + "x2 +" + b + "x +" + c;
30     }
31 }

```

- *Localizador.java* es el esqueleto de la clase Localizador; es la que debe rellenar, pero no cambiar lo que ya hay escrito. Es el único fichero que debe entregar.

```

1  class Localizador {
2      //Este código se incluirá en la próxima re-edición de este documento.
3  }

```

- *PruebaLocalizador.java* un ejemplo de pruebas, que puede usar tal como está, aunque se recomienda que haga más pruebas.

```

1  class PruebaLocalizador {
2      public static void prueba (Punto p1, Punto p2,
3          double d1, double d2) throws Exception {
4          Localizador l = new Localizador (p1, p2);
5          System.out.println (l.calculaPosicion (d1, d2));
6      }
7      public static void main (String [] args) throws Exception {
8          prueba (new Punto (0, 0), new Punto (100, 0), 50, 50);
9          prueba (new Punto (0, 0), new Punto (100, 0), 60, 70);
10         prueba (new Punto (0, 0), new Punto (0, 100), 50, 50);
11         prueba (new Punto (0, 0), new Punto (0, 100), 1000, 1000);
12         prueba (new Punto (0, 0), new Punto (100, 100), 122, 122);
13         prueba (new Punto (0, 0), new Punto (100, 100), 70.8, 70.8);
14         prueba (new Punto (0, 0), new Punto (100, 100), 110, 32);
15         prueba (new Punto (0, 0), new Punto (100, 100), 100, 32);
16     }
17 }

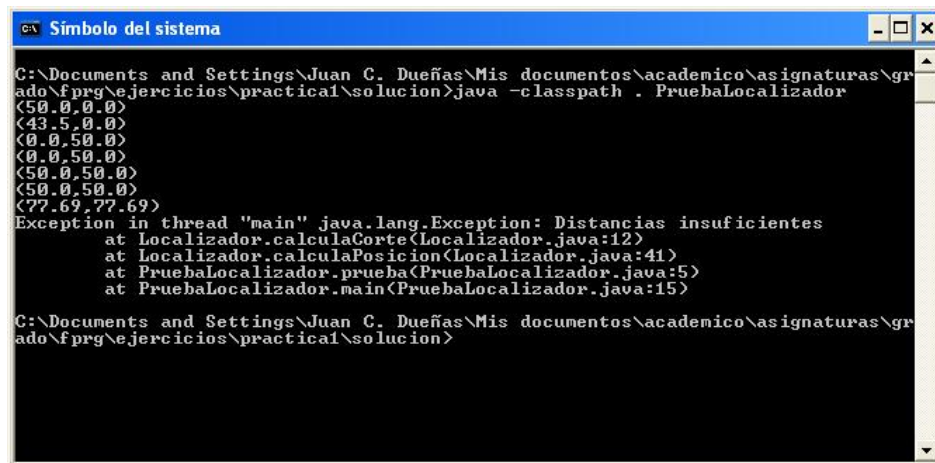
```

Ponga todos los ficheros en el directorio `fprg/practical` de su cuenta del laboratorio, y trabaje completando el fichero `Localizador.java`. Puede escribir o cambiar los programas de pruebas y ponerlos en el mismo directorio. Para programar la clase `Localizador.java` puede usar cualquiera de las herramientas o entornos de edición/compilación/ejecución del laboratorio.

## Las pruebas

Probar consiste en comprobar que el programa no tiene errores al ejecutarlo. Normalmente sí va a tener errores, así que una buena prueba es la que descubre algún error. En este caso, deberá hacer pruebas comprobando que el localizador funciona correctamente: si las distancias no son correctas debe lanzarse una excepción, y si todos los valores son correctos, se calcula el valor adecuado para la localización del teléfono móvil.

Si las cosas no funcionan como esperaba, trate de imaginar cómo se ejecutan las líneas de su programa (en qué orden) y añadir una sentencia `System.out.println(...)` en algún sitio para comprobar, por ejemplo, el valor de alguna de las variables. La clase `PruebaLocalizador` tiene varias de estas pruebas, cuyos resultados se imprimen por la pantalla. A continuación puede ver los resultados que se obtienen al ejecutar el método `main` de esta clase (después de haber rellenado correctamente `Localizador` y haber compilado todas las clases).



```
C:\Documents and Settings\Juan C. Dueñas\Mis documentos\academico\asignaturas\grado\fp\ejercicios\practical\solucion>java -classpath . PruebaLocalizador
(50.0,0.0)
(43.5,0.0)
(0.0,50.0)
(0.0,50.0)
(50.0,50.0)
(50.0,50.0)
(77.69,77.69)
Exception in thread "main" java.lang.Exception: Distancias insuficientes
    at Localizador.calculaCorte(Localizador.java:12)
    at Localizador.calculaPosicion(Localizador.java:41)
    at PruebaLocalizador.prueba(PruebaLocalizador.java:5)
    at PruebaLocalizador.main(PruebaLocalizador.java:15)
C:\Documents and Settings\Juan C. Dueñas\Mis documentos\academico\asignaturas\grado\fp\ejercicios\practical\solucion>
```

## Entrega: fecha y procedimiento

La práctica deberá entregarse antes de las 23:30 horas del día 21 de noviembre de 2001. Deberá entregar el fichero `Localizador.java`, puesto que el resto de ficheros son los proporcionados y no se pueden cambiar.

La entrega se realizará dejando el fichero `Localizador.java` en su cuenta del laboratorio, en el directorio `$HOME/fprg/practical`. Tenga especial cuidado con su contraseña para evitar malentendidos. Aparte de dejar este fichero en su cuenta de laboratorio, opcionalmente se puede hacer la entrega en el servidor Web del grupo 11.1.

## Evaluación

El día 22 de noviembre se recogerán el fichero `Localizador.java` en cada directorio `fprg/practical` de cada alumno.

La práctica se considerará "no presentada" si no se encuentra el fichero `Localizador.java`, o si no compila correctamente (esto es, sin errores), o si se detecta que ha sido copiado. Para las prácticas presentadas, la nota en esta práctica dependerá de que el `Localizador` sea capaz:

- de identificar el caso de que las distancias a las estaciones base sean incorrectas,
- generar un objeto de tipo `Punto` como solución para el caso de circunferencias con dos puntos de corte, y el resultado es aproximado al analítico (el obtenido haciendo las operaciones a mano),
- generar un objeto de tipo `Punto` como solución para el caso de circunferencias con un solo punto de corte, y el resultado es aproximado al analítico (el obtenido haciendo las operaciones a mano).

La nota subirá si, probando con diferentes combinaciones de estaciones base y distancias se obtiene un resultado aproximado al analítico (el obtenido haciendo las operaciones a mano). Se realizarán pruebas con circunferencias secantes y tangentes.

Otro consejo importante es que imprima su práctica en papel y la lleve el día del examen.

## Relación con otros temas y asignaturas

En esta práctica se han explorado, aunque muy brevemente, algunos campos de la ingeniería en las que los ordenadores juegan un papel importante, como es el del soporte a la telefonía en general, y a la telefonía móvil en particular. Las estaciones base, los terminales (los teléfonos móviles), los sistemas que controlan la red de comunicación, e incluso las antenas, todos llevan software especial. En algunos casos, el software es especial por el enorme número de funciones que realiza, en otros casos, lo más

importante es que funcione muy rápidamente y sin fallar; en otros, lo más importante es que sea fácil de usar. Durante sus estudios de ingeniero de telecomunicación, volverá a ver estos temas en asignaturas como "Sistemas de radiodifusión", "Sistemas de telefonía móvil", "Software de comunicaciones", "Laboratorio de Software de Comunicaciones" o "Conmutación", por ejemplo.

Las características que se han mencionado (que el software sea correcto, o que funcione muy rápidamente, o que sea fácil de utilizar) no son características puramente funcionales (no se refieren a lo que hace el programa, sino a cómo lo hace); a veces, a estos requisitos se les llama "características no funcionales", "requisitos no funcionales" o "características de calidad". Estudiará cómo conseguir que un programa las cumpla en las asignaturas "Arquitectura de ordenadores", "Laboratorio de programación de sistemas", "Ingeniería del software" o "Laboratorio de ingeniería del software".

Concretamente, los aspectos de rendimiento y eficiencia de un programa (cómo de rápido puede dar las respuestas), se volverán a tratar en temas siguientes de "Fundamentos de programación" y con más detalle en "Laboratorio de programación", cuando se explique qué es la complejidad de algoritmos.

Como habrá podido darse cuenta en el desarrollo de esta práctica, los ordenadores son mucho más rápidos que las personas cuando se trata de realizar cálculos matemáticos repetitivos. En algunas asignaturas de matemáticas de esta carrera, como "Cálculo numérico" o "Análisis computacional", estudiará las técnicas para implementar algoritmos matemáticos mediante programas de ordenador.

Por último, se habrá dado cuenta de que para hacer un programa que resuelve un problema hay que saber del problema: en este caso, unas nociones de geometría, que seguramente repasará (puesto que se suponen conocidas), en las asignaturas "Cálculo" y "Álgebra".

## *Preguntas y observaciones*

Las siguientes preguntas y comentarios tienen por objeto comprobar que se han entendido los conceptos manejados en esta práctica. Trate de contestarlos de forma personalizada, pero no se van a recoger ni evaluar. Podrían preguntarse en el examen.

1. ¿ Por qué deben lanzar excepciones los métodos de cálculo de las raíces en la clase PolinomioGrado2 ? . ¿ Habría otra forma de obtener las raíces reales e imaginarias ?
2. Escriba una prueba en la que se calcule la posición diez veces y mida cuánto tarda en ejecutarse. Ahora realice a mano el cálculo y compare el tiempo que tarda manualmente.
3. Intente imaginar otros métodos de resolución del problema: por ejemplo, ir recorriendo los puntos de alguna de las circunferencias hasta encontrar aquel cuya distancia a las otras dos estaciones base se parece más a la dada.
4. Como ejemplo de cómo se calcula la posible velocidad de ejecución de algunos programas, para un caso de prueba, indique cuáles instrucciones de Java se ejecutarán, en qué orden, y cuántas veces. Si ha hecho la pregunta anterior, compare el número de instrucciones con ambos métodos, y cual es el número mínimo, medio y máximo de instrucciones que se ejecutarán dependiendo del caso de prueba que se ponga.
5. Imagine cómo sería la solución al problema cuando el localizador puede conocer las coordenadas y las distancias a un número variable de estaciones base (siendo dos el mínimo).





# 6. Clases

## 6.1. Clases

*Definición:* plantilla con atributos y métodos. Dentro de los métodos no se pueden escribir otros, pero sí llamarlos. También se da un ciclo de los objetos: se crean, se usan y se destruyen.

Existe una relación entre clases: cuando *Localizador.java* crea un objeto de *PolinomioGrado2.java* y usa un método tal como *raiz()*, decimos que *Localizador* es clase cliente y *PolinomioGrado2* servidora.

## 6.2. Derechos de acceso

Se trata de definir qué partes de una clase se pueden usar en otras clases y cuales no.

```
class Punto {
    private double x;          //solo podrá accederse a los atributos en el cuerpo de la clase
    private double y;          //y en el cuerpo de los métodos.
    public Punto (...) { ... }  // Se podrán usar en cualquier parte
    public double distancia (...) { ... } // conviene que el constructor sea público.
```

**Encapsulación:** cada clase tiene sus propios atributos y métodos, de manera que es lo más autónoma posible.

*protected:* será público o privado.

Si no asignamos *public*, *private* o *protected*, las clases del mismo directorio (paquete) son “amigas” y pueden utilizar y compartir datos como si fueran públicos.

*Tipo abstracto de datos:* atributos privados, métodos públicos.

*Interfaz o parte pública de una clase:* aquello que las clases clientes deben conocer para utilizar ésta. Serán atributos *public* si existen, y la cabecera de los métodos públicos.

*Implementación o parte privada:* expresa las operaciones que se realizan, es todo lo *private*.

## 6.3. Estáticos

Se usa *static* en atributos o métodos: existe una sola vez para los elementos de la clase.

Private **static** double x; //todos los puntos tienen la misma x.

```
class Punto {
    private double x;
    private static int numPuntos = 0; //los estáticos han de inicializarse siempre
    //al crear varios puntos, numPuntos es global, solo existe 1 vez
    public Punto (double x, double y) {
        this.x = x; // el constructor no puede ser estático
        this.y = y;
        numPuntos++; // numPuntos aumenta cada vez que se crea un punto.
    }
    public static int getNumPuntos() {
        return numPuntos;
    }
    // alternativa
    public static int getNumPuntos() {
        x = 0; // ERROR: los estáticos no pueden manejar variables que
        y = 0; // no sean estáticas. x e y ni siquiera existen.
        return numPuntos;
    }
}
```

## 6.4. Relaciones entre clases

Uso: *cliente-servidora*.

- *Variable en método*
- *Parámetro de método*
- *Tipo devuelto de método*

```

class Localizador {
    ...
    Punto calculaCorte( ... ) {                // tipo devuelto de método
    ... }
    Punto calculaPosicion( ... ) {
        Punto p1 = calculaCorte( ... );        //variable en método
    }
}

```

**Composición:** un atributo es de otra clase.

```

class Localizador {
    private Punto p1;                          //localizador compuesto por puntos.
    private Punto p2;

    public Localizador (Punto p1, Punto p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    public Localizador (Punto p1, Punto p2) {    //constructor alternativo
        this.p1 = new Punto (p1.getX(), p1.getY());
        this.p2 = new Punto (p2.getX(), p2.getY());
    }
}

```

**[ejercicio nº 5] [dificultad 6]** Escriba un programa que calcule todos los números primos entre 1 y un número dado.

Para ello, cree el método "boolean EsPrimo(int n)" que indicará si n es primo o no. Este método comprobará si n es divisible por algún número entero entre 2 y la raíz cuadrada de n.

➤ *EscribePrimos.java:*

```

1      class EscribePrimos {
2          public static void main(String[] args) {
3              int n;
4              for (int i = 1; i <= n; i++) {
5                  if (EsPrimo(i)) {
6                      System.out.println(i);
7                  }
8              }
9          }
10         public static boolean EsPrimo(int n) {
11             for (int i = 2; i <= Math.sqrt(n); i++) {
12                 if ((n%i) == 0)
13                     return false;
14             }
15             return true;
16         }
17     }

```

**[ejercicio nº 9] [dificultad 6]** Escriba un método que calcule el valor de la función de Ackermann. Esta función se define para números enteros no negativos como:

$$\begin{aligned}
 &n + 1 \text{ si } m=0 \\
 &A(m,n) = A(m-1, 1) \text{ si } m>0 \text{ y } n=0 \\
 &A(m-1, A(m, n-1)) \text{ si } m>0 \text{ y } n>0
 \end{aligned}$$

➤ *ackermann.java:*

```

1      class ackermann {
2          int ackermann (int m, int n) throws Exception {
3              if ((m < 0) || (n < 0))
4                  throw new Exception("Uno de los números es negativo");
5              if (m == 0)
6                  return n+1;
7              if ((m > 0) && (n == 0))
8                  return ackermann(m-1, 1);
9              return ackermann(m-1, ackermann(m, n-1));
10         }
11     }

```

- *CodificacionDni.java: programa que imprime, dado un dni, la siguiente colocación de asteriscos. Además lanzará excepciones si falta el dni o es incorrecto, y añade un cero a la izquierda si es de 7 dígitos.:*

0123456789

```

      *
    *
  *
 *
*

```

```

1  class CodificacionDni {
2      public static void main(String[] args) throws Exception {
3          System.out.println("0123456789");
4          String dni = args[0];
5          if (dni == null)
6              throw new Exception("Introduce un dni");
7          int len = dni.length();
8          if ((len<7) || (len>8))
9              throw new Exception();
10         if (len==7) {
11             dni = "0"+dni;
12             len = 8;
13         }
14         for (int i =0; i < len; i++) {
15             int num = dni.charAt(i) - '0';
16             if (num > 9 || num < 0)
17                 throw new Exception();
18             while (num-- > 0)
19                 System.out.println(" ");
20             System.out.println("*");
21         }
22     }

```



# 7.Relaciones entre clases

## 7.1.Polimorfismo

*Definición:* El polimorfismo es la manera de manejar clases distintas, como si fueran una única. Se trata de la relación de “parecido” entre ellas, por ejemplo si tienen métodos con igual signatura.

*Ligadura estática:* cuando en un método de una clase A llamamos a otro de la clase B; al compilar A, la ligadura localiza el método de la clase correspondiente (B).

Ejemplo típico: creación de un objeto y uso de sus métodos:

```

1      class Punto {
..      public void setX(double x) {
..      ...
..      }
..      }

1      class PruebaPunto {
2      public static void main (String[] args) {
3          Punto p = new Punto();
4          p.setX(31.2);
5      }
6      }
```

*Ligadura dinámica:* hay casos donde el compilador no puede determinar el método exacto, al haber varios en distintas clases que tienen igual signatura. Es el intérprete quien durante la ejecución quien escoge el método exacto. La verdadera solución está en la implementación de interfaces.

```

1      class PruebaPunto {
2      public static void main (String[] args) {
3          Movil m;
..      m.mueve(new Punto(3.2));

1      class TelefonoMovil {
2      public void mueve (Punto p) {
..      ...
..      }

1      class Camion {
2      public void mueve (Punto p) {
..      ...
..      }
```

## 7.2.Implementación de Interfaces:

*Definición:* Una interfaz es una clase especial en Java. Sirve para realizar la ligadura dinámica, y está “hueca” (no tiene atributos, ni cuerpo sus métodos).

- No pueden llevar método constructor
- Todas sus cabeceras han de ser públicas
- No se denominan *class*, sino *interface*
- Se pueden incluir constantes en ellas.
- No se pueden crear objetos de *interfaces*
- Representan elementos comunes de otras clases

```

1      interface Ubicado {
2      public Punto getPosicion();
3      public static final double DISTANCIA_MINIMA = 10E-3;
4      }

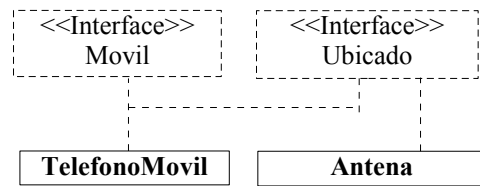
1      interface Movil {
2      public void mueve (Punto p);
3      }
```

```

1  class TelefonoMovil implements Ubicado, Movil {
2      public Punto getPosicion() {
3          ...
4      }
5      public void mueve (Punto p) {
6          ...
7      }
8  }

1  class Antena implements Ubicado {
2      public Punto getPosicion() {
3          ...
4      }
5  }

```



## 7.3. Uso de Interfaces

```

Movil m = null;
Ubicado n = null;
m = new TelefonoMovil();
n = new TelefonoMovil();
System.out.println(n.getPosicion());
n = new Antena();
System.out.println(n.getPosicion());
System.out.println(n.getPosicion().toString());
Ubicado n = new TelefonoMovil();           //upcasting
System.out.println(n.getPosicion());
...
n.marcar("6000000000");
((TelefonoMovil)n).marcar("6000000000");    //downcasting

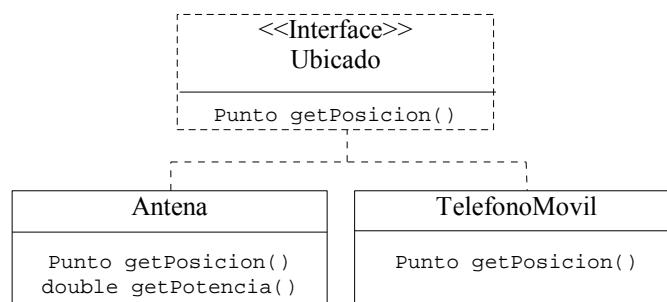
```

- *Upcasting*: generalizar, tratar a un objeto de una clase como si fuera de una clase más general.
- El upcasting nunca produce Excepción.
- *Downcasting*: es especializar. Usar una referencia como si fuera de una clase más detallada.
- El downcasting es más inseguro (*ClassCastException*)

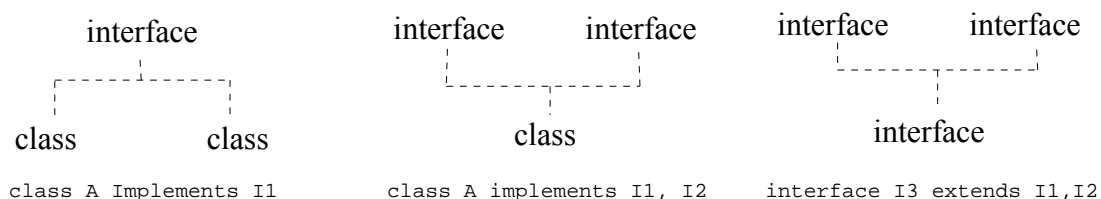
```

System.out.println(n.getPotencia());           //error de compilación
System.out.println((Antena)n).getPotencia();   //downcasting

```



## 7.4. Jerarquía de Interfaces



## 7.2. Herencia y extensión

clase existente (base) <--- clase nueva (derivada, extendida)

Punto (x, y) <--- Pixel (Punto + color)

```

1      class Pixel extends Punto {
2          private String color;
3          public String getColor() {
4              return this.color;
5          }
6          public String toString() {
7              return color;
8          }
9      }

```

- Extensión de la interfaz

```

Pixel px = new Pixel();
px.setColor("rojo");
px.setX(6.0);                                     //podemos llamar a métodos públicos de la clase base
px.setY(-6.2);
System.out.println(px.toString());                //imprime toString() de pixel

```

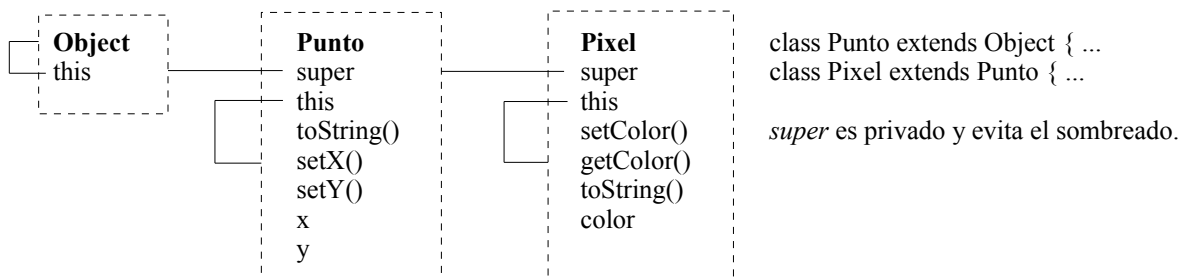
- Solución al método to String: cómo imprimir Punto y color

```

1      class Pixel extends Punto {
2          ...
3          public String toString() {
4              return super.toString()+color;
5          }
6      }

```

**Sombreado:** se produce al escribir dos métodos públicos de igual nombre en la clase base y en la derivada. Tiene preferencia el de derivada, al otro no se le puede llamar. Será utilizada la referencia *super*.



### Constructores:

```

public Pixel (double x, double y, String color) {
    this.x = x;                                     //ERROR: en la clase Pixel no están los atributos x e y.
    this.y = y;
    this.color = color;
}

```

```

public Pixel (double x, double y, String color) {
    super(x, y);                                     //super siempre va en la primera sentencia.
    this.color = color;
}

```

*protected*: tipo de atributo que es público para las clases derivadas y privadas para el resto.

```

class Punto {
    protected double x;
    protected double y;
    ...
}

```

```

class Pixel extends Punto {
    public Pixel (double x, double y, String color) {
        this.x = x;
        this.y = y;
        ...
    }
}

```

//código VÁLIDO.

*Upcasting y downcasting en herencias:*

```

Punto p = new Pixel();           //upcasting
((Pixel)p).setColor("verde");    //downcasting

```

¡¡El sombreado no se elimina con upcasting o downcasting!!

*Un poco más sobre la clase Object:*

- boolean equals(Object o);
- Object clone();
- String toString();

## 7.6 Herencia forzada y prohibida

*Forzada:*

```

abstract class Figura {
    private Punto centro;
    public Figura (Punto p) {
        this.centro = p;
    }
    public Punto getCentro() {
        return centro;
    }
    public void setCentro (Punto p) {
        this.centro = p;
    }
    public abstract double getPerimetro();
    public abstract double getArea();
}

```

Figura f = new Figura(p) //INCORRECTO. Figura es una clase abstracta, no permite objetos.

*Prohibida:*

```

final class Permiso {
    private String contraseña;
    private String usuario;
    public boolean valido(String user, String pswd) {
        return usuario.equals(user) && contraseña.equals(pswd);
    }
}

```

Ejemplo de hacking: si se creara esta clase y Permiso no fuera *final*, siempre se tendría acceso.

```

class Hacker extends Permiso {
    public boolean valido (String user, String pswd) {
        return true;
    }
}

```

*Creación de nuevas Excepciones (excepciones derivadas):*

```

class ValoresNoCorrectos extends Exception {
    public ValoresNoCorrectos (String m) {
        super(m);
    }
}

```

```

throw new ValoresNoCorrectos("Faltan datos");

```

//código CORRECTO.



```
try {  
    ...  
}  
catch (ValoresNoCorrectos e) {  
    ...                // errores derivados  
}  
catch (Exception e) {  
    ...                // errores base  
}  
finally {  
    ...                //opcional, SIEMPRE se ejecuta (funcione o no try)  
}
```



## 8. Estructuras Indexadas de Datos

### 8.1. Almacenes

*Definición:* Cualquier conjunto de datos capaz de manejarlos en cantidades masivas.

- Capacidad: puede ser limitada o ilimitada.
- Velocidad: los hay rápidos (escritos en memoria) o lentos (ficheros de disco duro o red)
- Persistencia: los datos pueden existir aunque acabe el programa.
- Tipo de contenido: será homogéneo si solo hay datos de un tipo o heterogéneo si no es así.

### 8.2. Arrays

*Definición:* Conjunto homogéneo e indexado de elementos. Están colocados de manera consecutiva, y pueden ser de una dimensión (vectores), de dos (matrices) o tridimensionales (rejillas). Cada uno de los elementos viene determinado por un índice.

- Se manejan mediante referencias.
- Tienen capacidad limitada.
- Se trata de almacenes rápidos y no persistentes.

*Partes de un array:*

- Tipo base: tipo de todos los elementos.
- Índice: número entero comprendido entre 0 y la longitud -1. Indica la posición de los elementos.
- Dimensión: una, dos o tres dimensiones. Se necesitan tantos índices como número de dimensiones haya.

### 8.3. Definición del array

*Estructura:* **tipo-base[dimensión] nombre** Al definirlo, se crea una referencia al array, pero todavía no existe. La definición alternativa e igualmente válida es: **tipo-base nombre[dimensión]**.

```
double[] notas;      //array unidimensional de notas de tipo real.
String[] args;       //array del método main que contiene los argumentos de línea de comandos.
int[][] imagen;      //array bidimensional de una imagen. Cada entero el color de un píxel.
Punto[] trayectoria;  //array que contiene todos los puntos de una trayectoria.
double[][][] espacio; //array tridimensional que define un espacio.
```

### 8.4. Creación

Una vez definido un array, para crearlo se repite el tipo base y definir el número de elementos por dimensión.

**tipo-base[dimensión] = new tipo-base[elementos dimensión]**

```
Punto[] trayectoria = new Punto[50];      //Array de una trayectoria definida por 50 puntos.
Int[][] imagen = new int[80][25];         //se crean 80x25 huecos de enteros en la memoria.
Double[] notas = new double [141];        //se crea un array para 141 notas.
Double[][][] espacio = new double[100][100][100]; //cubo de 100x100x100

int[][] cuadro = {{1,2},{3,4}}; //los array se pueden inicializar a mano, método engorroso.

Int[][] cuadro = {{1,2},{3,4,5}}; //es válido! Las matrices en java pueden tener huecos
//que apunten a null, ni siquiera el elemento 1,3 se rellena con cero.
//Funciona porque se trata de una matriz de matrices unidimensionales.
```

### 8.5. Acceso

*Estructura:* para utilizar un elemento del array, se usa como variable: **array[índice]**.

```
double[] notas;
notas = new double[141];
notas[0] = 10.0; //se rellena con 10.0 el primer elemento (elemento cero)
notas[notas.length-1] = 10.0; //rellena con 10.0 el último elemento (longitud -1)
notas[1] = notas[0]; //la nota del primer elemento se copia al segundo.
```

```
int[][] imagen = new int[80][24];
System.out.println(imagen[0][0]);
System.out.println(imagen[imagen.length-1][imagen[0].length-1]);
```

## 8.6. Operaciones

```
double[] notas;
notas = new double[141];
for (int i = 0; i < notas.length; i++) {
    notas[i] = 8.0; //asignaría 8.0 a todas las notas.
    Notas[i] = 10.0 * Math.random(); //asigna notas aleatorias.
}
double suma = 0.0;
for (int i = 0; i < notas.length; i++) {
    suma += notas[i];
}
System.out.println(suma/notas.length); //calcula media de todas las notas del array

void imprime(int[][] m) { //método que imprime los valores de una matriz
    for (int i = 0; i < m.length; i++) {
        for (int j = 0; j < m[i].length; j++) {
            System.out.println(m[i][j]+" ");
        }
        System.out.println();
    }
}

double[] notas = new double[50];
for (int i = 0; i < notas.length; i++) { //bucle genérico para recorrer arrays.
    notas[i] = ponerNota();
}
```

➤ *ImprimeArgs.java*: programa que al ejecutarlo con varios argumentos, éste los reimprime por pantalla.

```
1  class ImprimeArgs {
2      public static void main (String[] args) {
3          for (int i = 0; i < args.length; i++) {
4              System.out.println(args[i]);
5          }
6      }
7  }

double[] notas = new double[300];
double[] otroArray = new double[notas.length];
for (int i = 0; i < notas.length; i++) {
    otroArray[i] = notas[i]; //bucle que copia los valores de un array a otro
}

boolean iguales = (notas.length) == (otroArray.length);
if (iguales) {
    for (int i = 0; i < notas.length; i++) {
        if (notas[i] != otroArray[i]) { //compara arrays. El resultado se guarda
            iguales = false; //en el booleano 'iguales'.
            break;
        }
    }
}

boolean encontrado = false;
double elemento;
double[] notas;
for (int i = 0; i < notas.length; i++) {
    if (notas[i] == elemento) { //búsqueda lineal.
        encontrado = true;
        break;
    }
}
}
```

## 8.7. Vectores

Se usa la librería *java.util.Vector*. Los vectores son almacenes extensibles, compuestos por referencias a objetos y no por valores primitivos, como era el caso de los arrays. Para este apartado es necesario conocer el tipo *Object*, que

se refiere a los elementos del vector, que como hemos dicho que pueden ser referencias, *Object* se refiere a todas sean del tipo que sean.

**¡Atención! Dado que en la nueva versión de java (jdk1.5) se incluyen las Listas, es posible que los vectores y los Enumeration no entren en el EXAMEN FPRG 2005.**

### Métodos de la clase Vector:

- `void addElement (Object):` Añade una referencia al final del vector.
- `void setElementAt(Object, int) throws Exception:` Añade una referencia en una posición concreta.
- `void insertElementAt(Object, int) throws Exception`
- `boolean removeElement(Object):` Borra una referencia. Verdadero si la borra.
- `void removeAllElements():` Borra todas las referencias.
- `void removeElementsAt(int) throws Exception:` Borra la referencia de una determinada posición.
- `Object elementAt(int) throws Exception:` Devuelve la referencia que hay en una posición.
- `int indexOf(Object, int)`
- `int indexOf(Object)`
- `int size():` Devuelve el tamaño del vector.
- `boolean isEmpty():` True si está vacío.

```
//Ejemplo de creación de vector.
import java.util.*;
Vector v = new Vector();
v.addElement(new Punto(3,2));
```

➤ *Poligono.java:* clase que inserta los n puntos de un polígono en un vector.

```
1    class Poligono {
2        private Vector puntos;
3        public Poligono(int n) throws Exception {
4            if (n < 3) {
5                throw new Exception("No hay polígonos de menos de tres puntos");
6            }
7            puntos = new Vector(n);           //creación de vector de n puntos.
8        }
9        public void inserta(Punto p) throws Exception {
10           if (puntos.size() < puntos.capacity())           //uso del método capacity()
11               puntos.addElement(p);
12           else
13               throw new Exception("No hay capacidad para más puntos");
14       }
15   }

//Recorrido de un vector
for (int i = 0; i < puntos.size(); i++) {
    Object o = puntos.elementAt(i);
    Punto p = (Punto)o;           //downcasting. Se transforman todas las referencias a puntos.ç
```

## 8.8.Enumeration

En este apartado se verán otras formas más seguras de recorrer los elementos de un vector.

```
Enumeration e = puntos.elements();
while (e.hasMoreElements()) {
    Object o = e.nextElement();
    Punto p = (Punto)o;
}
```

**Métodos de Iterator:**

- `boolean hasNext();`
- `Object next();`
- `void remove();`

**Notación del jdk <=1.5 para evitar el Downcasting.**

```
Vector <Punto> puntos = new Vector<Punto>();
```

**Practica2 grupo 11: Objetivo**

Esta práctica tiene por objetivo principal ejercitarse en el uso de almacenes de datos, aplicación del concepto de composición, comprender el concepto de clase interfaz y ser capaz de realizar algunas clases que implementan una interfaz. Se cubren los siguientes elementos del lenguaje:

- definición de clases,
- creación y uso de métodos,
- clase compuesta (composición),
- uso e implementación de clases interfaz,
- uso de las clases predefinidas (almacenes o colecciones).

*El problema que se quiere resolver (especificación de requisitos)*

Se desea construir un sistema o programa de clasificación de los mensajes que llegan por correo electrónico. El programa constará de un buzón de correo electrónico al que llegan los mensajes, y su misión es repartir los mensajes en varios buzones: hay uno para los mensajes de correo basura, otro para los mensajes válidos y otro para los mensajes que hay que enviar de respuesta a otros usuarios. truir un sistema de clasificaguaje. Se usan varias clases ya hechas y hay que crear otras. El programa debe repartir los mensajes que llegan al buzón de entrada y colocarlos en el buzón adecuado (basura, válidos, para enviar) de forma automática, mediante unas reglas que se aplicarán a todos los mensajes del buzón de entrada. Una regla está compuesta por una condición o comprobación acerca de un mensaje, y por una acción que hay que realizar con el mensaje si se cumple la condición. En esta práctica se trabajará con buzones y mensajes simplificados, pero parecidos a los de un sistema de correo electrónico convencional.

*El diseño de la solución*

Se ha definido una clase Mensaje, que debe utilizarse de forma obligatoria; esta clase consta de varios atributos: remitente (quien envía el mensaje), destinatario (a quién se envía), cabecera (o tema), cuerpo del mensaje, documento (si va asociado un documento) y urgencia (si el mensaje es urgente o no). La clase consta de atributos, método constructor, métodos accesorios y método toString.

También se ha creado una clase Buzon que también debe usar en la solución. Un buzón es una lista de Mensajes (List<Mensaje> de Java 1.5), que además tiene un nombre. El buzón permite añadir un mensaje al final de la lista (método add), y sacar el primer mensaje (borrándolo o no). Un buzón funciona como una cola FIFO (los mensajes están ordenados según llegan).

Se proporciona la clase Correo que es la clase que arranca el programa (contiene un método main). Esta clase crea algunas reglas y las añade a una lista de reglas, también crea los buzones del sistema y rellena el buzón de entrada con algunos mensajes de prueba. El método procesar es el que se encarga de aplicar una por una todas las reglas a todos los mensajes del buzón de entrada. Dependiendo de las reglas, algunos mensajes irán a parar al buzón (o almacén) de basura, otros al buzón de mensajes válidos, y otros al buzón de mensajes para enviar como respuesta (para simplificar el problema, se envía el mensaje tal como está – en un sistema real habría que cambiar al menos el destinatario).

La clase Regla se proporciona ya hecha: tiene una referencia a una Accion y una referencia a una Comprobacion como atributos. Estos atributos se asignan en el constructor. La clase Regla tiene un método cumple que llama al método cumple de su Comprobacion, y un método accion que llama al método accion de su Accion.

Es necesario leer y entender completamente las clases que se dan hechas para poder implementar las clases que faltan. Estas clases que faltan son sencillas de implementar:

- una interfaz que se llama Accion y que debe contener únicamente una cabecera de método público que devuelve void, el método se llama accion y toma como parámetro un mensaje.
- una clase que implementa a la interfaz Accion y que se llama Almacenar. Esta clase tiene un atributo que es una referencia a un Buzon; esta referencia se le pasa como parámetro en el constructor. La acción que hace esta clase es la de almacenar el Mensaje que se le pasa como parámetro, en su Buzón.
- una clase que implementa a la interfaz Accion y que se llama AccionCompuesta. Esta clase debe contener un atributo que sea una lista de acciones (List<Accion>), debe inicializar esta lista en el constructor (como ArrayList<Accion>). Esta clase debe tener un método para añadir acciones a la lista de acciones. Como AccionCompuesta implementa a la interfaz Accion,

debe tener un método `accion` con un `Mensaje` como parámetro. Este método recorre la lista de acciones y para cada una de ellas, llama a su método `accion` pasándole el mensaje como parámetro.

- una interfaz que se llama `Comprobacion` y que debe contener únicamente una cabecera de método público que devuelve boolean, el método se llama `cumple` y toma como parámetro un mensaje.
- una clase que implementa a la interfaz `Comprobacion` y que se llama `ValeSiempre`. El método `cumple` de esta clase devuelve siempre `true`.
- una clase que implementa a la interfaz `Comprobacion` y que se llama `ComprobacionRemitente`. Esta clase debe contener un atributo de tipo `String` (cuyo valor se le pasa en el constructor) que es el nombre del remitente que buscamos. El método `cumple` mira a ver si este atributo de tipo `String` es el mismo que el remitente del mensaje que le pasan como parámetro. La comprobación la hace pasando a minúsculas (con el método `toLowerCase` de `String`) las dos cadenas de caracteres.

Con todo ello se consigue un sistema de clasificación muy flexible: por ejemplo, si se desea enviar algo al buzón de basura, se pone una regla con la comprobación correspondiente y cuya acción sea la de almacenar el mensaje en el buzón de basura. Si se desea que un mensaje se almacene en el buzón de mensajes válidos y además se envíe a otro sitio, se crea una regla con la condición que sea y la acción es una acción compuesta (almacenar en el buzón de salida y almacenar en el buzón de salida). El sistema se puede extender fácilmente con más comprobaciones y acciones.

## La programación o implementación

Estos son los ficheros que puede usar (ponga todos ellos en el directorio de su práctica):

- `Mensaje.java` contiene la clase `Mensaje` completa; no puede cambiarla.

```

1  public class Mensaje {
2      private String remitente;
3      private String destinatario;
4      private String cabecera;
5      private String cuerpo;
6      private String documento;
7      private boolean urgente;
8      public Mensaje (String remitente, String destinatario, String cabecera, String cuerpo,
9                      String documento, boolean urgente) {
10         this.remitente = remitente;
11         this.destinatario = destinatario;
12         this.cabecera = cabecera;
13         this.cuerpo = cuerpo;
14         this.documento = documento;
15         this.urgente = urgente;
16     }
17     public String getCabecera() {
18         return cabecera;
19     }
20     public String getCuerpo() {
21         return cuerpo;
22     }
23     public String getDestinatario() {
24         return destinatario;
25     }
26     public String getDocumento() {
27         return documento;
28     }
29     public String getRemitente() {
30         return remitente;
31     }
32     public boolean isUrgente() {
33         return urgente;
34     }
35     public Mensaje duplica () {
36         return new Mensaje (remitente, destinatario, cabecera,cuerpo, documento, urgente);
37     }
38     public String toString () {
39         return "" + remitente + ":"+ destinatario+ ":"+ cabecera + ":"+ cuerpo + ":"+ documento +
40             ":"+ urgente + ":";
41     }
42 }
```

- `Buzon.java` contiene la clase `Buzon`; no puede cambiarla.

```

1  import java.util.*;
2  public class Buzon {
3      private List<Mensaje> mensajes;
4      private String nombre;
```

```

5      public Buzon (String nombre) {
6          this.nombre = nombre;
7          mensajes = new ArrayList<Mensaje> ();
8      }
9      public String getNombre () {
10         return nombre;
11     }
12     public void addMensaje (Mensaje m) {
13         mensajes.add(m);
14     }
15     public Mensaje getPrimero () {
16         return mensajes.get(0);
17     }
18     public Mensaje removePrimero () {
19         if (mensajes.size() == 0)
20             return null;
21         Mensaje m = mensajes.get(0);
22         mensajes.remove(0);
23         return m;
24     }
25     public String toString () {
26         String res = "";
27         for (Mensaje m: mensajes)
28             res += "\n" + m;
29         return res;
30     }
31 }

```

- Correo.java contiene la clase Correo; no puede cambiarla. Esta clase contiene el método main que arranca el sistema.

```

1      import java.util.*;
2      import java.io.*;
3      public class Correo {
4          private Buzon entrada;
5          private Buzon salida;
6          private Buzon spam;
7          private Buzon valido;
8          private List<Regla> reglas;
9          public Correo () {
10             entrada = new Buzon("entrada");
11             salida = new Buzon ("salida");
12             spam = new Buzon ("spam");
13             valido = new Buzon ("valido");
14             reglas = new ArrayList<Regla>();
15         }
16         public void iniciarReglas () {
17             // spam
18             reglas.add (new Regla (new ComprobacionRemitente("remite2"), new Almacenar (spam)));
19             // sinclasificar
20             reglas.add (new Regla (new ComprobacionRemitente("remite0"), new Almacenar (valido)));
21             // salida
22             reglas.add (new Regla (new ComprobacionRemitente("remite3"), new Almacenar (salida)));
23             // duplicacion
24             AccionCompuesta a = new AccionCompuesta();
25             a.addAccion (new Almacenar (salida));
26             a.addAccion (new Almacenar (valido));
27             reglas.add (new Regla (new ComprobacionRemitente("remite8"), a));
28             // sinclasificar
29             reglas.add (new Regla (new ValeSiempre(), new Almacenar (valido)));
30         }
31         public void iniciarEntrada () {
32             for (int i = 0; i < 10; i++) {
33                 entrada.addMensaje(new Mensaje( "remite" + i, "dest" + i,"head" + i,"body" + i,
34                     "doc" + i, false));
35             }
36         }
37         public void procesar () {
38             Mensaje m = entrada.removePrimero();
39             while (m != null) {
40                 for (Regla r: reglas) {
41                     if (r.cumple(m)) {
42                         r.accion (m);
43                         break;
44                     }
45                 }
46             }
47         }
48     }

```



```

45     m = entrada.removePrimero();
46     }
47     }
48     public static void main(String[] args) {
49         Correo c = new Correo ();
50         c.iniciarReglas();
51         c.iniciarEntrada();
52         System.out.println ("Entrada" + c.entrada);
53         c.procesar();
54         System.out.println ("Entrada" + c.entrada);
55         System.out.println ("Salida" + c.salida);
56         System.out.println ("Spam" + c.spam);
57         System.out.println ("Valido" + c.valido);
58     }
59 }

```

- Regla.java contiene la clase Regla; no puede cambiarla.

```

1  public class Regla {
2      private Comprobacion c;
3      private Accion a;
4      public Regla (Comprobacion c, Accion a) {
5          this.c = c;
6          this.a = a;
7      }
8      public boolean cumple (Mensaje m) {
9          return c.cumple(m);
10     }
11     public void accion (Mensaje m){
12         a.accion(m);
13     }
14 }

```

- Comprobacion.java que debe escribir (una solución simple ocupa 3 líneas de código).

```

1  interface Comprobacion {
2      public boolean cumple (Mensaje m);
3  }

```

- ComprobacionRemitente.java que debe escribir (una solución simple ocupa 9 líneas de código).

```

1  class ComprobacionRemitente implements Comprobacion {
2      private String remitente;
3      public ComprobacionRemitente (String remitente) {
4          this.remitente = remitente;
5      }
6      public boolean cumple (Mensaje m) {
7          return ((this.remitente).toLowerCase()).equals((m.getRemitente()).toLowerCase());
8      }
9  }

```

- ValeSiempre.java que debe escribir (una solución simple ocupa 5 líneas de código).

```

1  class ValeSiempre implements Comprobacion {
2      public boolean cumple (Mensaje m) {
3          return true;
4      }
5  }

```

- Accion.java que debe escribir (una solución simple ocupa 3 líneas de código).

```

1  interface Accion {
2      public void accion (Mensaje m);
3  }

```

- AccionCompuesta.java que debe escribir (una solución simple ocupa 15 líneas de código).

```

1  import java.util.*;
2  class AccionCompuesta implements Accion {
3      private List<Accion> acciones;
4      public AccionCompuesta () {

```

```

5      acciones = new ArrayList<Accion> ();
6      }
7      public void accion (Mensaje m) {
8          for (Accion a: acciones) {
9              a.accion(m);
10         }
11     }
12     public void addAccion (Accion a) {
13         acciones.add(a);
14     }
15 }

```

- Almacenar.java que debe escribir (una solución simple ocupa 12 líneas de código).

```

1  class Almacenar implements Accion {
2      private Buzon b;
3      public Almacenar (Buzon b) {
4          this.b = b;
5      }
6      public void accion (Mensaje m) {
7          b.addMensaje(m);
8      }
9  }

```

Ponga todos los ficheros en el directorio fprg/practica2 de su cuenta del laboratorio. Para programar puede usar cualquiera de las herramientas o entornos de edición/compilación/ejecución del laboratorio.

Se recomienda que haga las clases poco a poco, y en este orden: Comprobacion, ValeSiempre, ComprobacionRemitente, Accion, Almacenar, AccionCompuesta.

### *Entrega: fecha y procedimiento*

La práctica deberá entregarse antes de las 23:30 horas del día 19 de enero de 2006. La entrega se realizará dejando los ficheros pedidos en su cuenta del laboratorio, en el directorio fprg/practica2 en su directorio personal (\$HOME). Tenga especial cuidado con su contraseña para evitar malentendidos. No se aceptarán entregas por ningún otro medio.

### *Evaluación*

El día 23 de enero se recogerán los ficheros de cada directorio fprg/practica2 de cada alumno. La práctica se considerará “no presentada” si no se encuentran los ficheros Accion.java ni Comprobacion.java, o si estos dan errores de compilación, o si se detecta que han sido copiados.

La corrección y puntuación de las prácticas presentadas dependerá de las siguientes pruebas, y en este orden (si se pasan correctamente irá subiendo la nota):

- Que esté bien definida la interfaz Comprobacion (necesario para aprobar la práctica).
- Que funcione correctamente la clase ValeSiempre (necesario para aprobar la práctica).
- Que funcione correctamente la clase ComprobacionRemitente.
- Que esté bien definida la interfaz Accion (necesario para aprobar la práctica).
- Que funcione correctamente la clase Almacenar.
- Que funcione correctamente la clase AccionCompuesta con pocas acciones y que todas sean de almacenar.
- Que las clases realizadas funcionen correctamente con la clase Correo.

Otro consejo importante es que imprima su práctica en papel y la lleve el día del examen.

## *Changelog* (control de cambios):

- 13.12.05.      Creación de este documento, temas 1 al 6. Aplicación de Licencia Creative Commons.
  
- 15.12.05.      Añadida la práctica 1 al final del tema 5.  
                    Incluido tema 8 (Estructuras indexadas de datos)  
                    Tres ejercicios corregidos al final del tema 6.
  
- 17.01.06.      Añadida la práctica 2 al final del tema 8.  
                    Corrección de erratas en la licencia.  
                    Añadido método *toLowerCase()* y *toUpperCase()* a teoría de Strings.
  
- 06.02.06      Añadido todo el código de la práctica 2.