Grado en Ingeniería Informática
Grado en Matemáticas e Informática

Asignatura: PROGRAMACIÓN II

Clases y Objetos

Manuel Collado,
Ángel Lucas González Martínez,
Jaime Ramírez
Guillermo Román

DLSIIS - E.T.S. de Ingenieros Informáticos
Universidad Politécnica de Madrid

Septiembre 2013

¿Qué es un objeto?

Un **objeto** representa una entidad, ya sea física o conceptual, relevante para comprender el dominio del problema o para formular la solución.

- Entidad física: *coche, persona*,..
- Entidad conceptual: *buffer, árbol binario*,…

Los objetos son abstracciones

- Toda abstracción es una *simplificación* que representa ciertas características relevantes y olvida otras.

- Dependiendo del observador de un coche:
  - **un vendedor**: modelo, precio, color . . .
  - **un mecánico**: tipo de motor, transmisión, suspensión . . .
  - **un ingeniero**: . . .
  - . . . etc. . . .

Características de los objetos

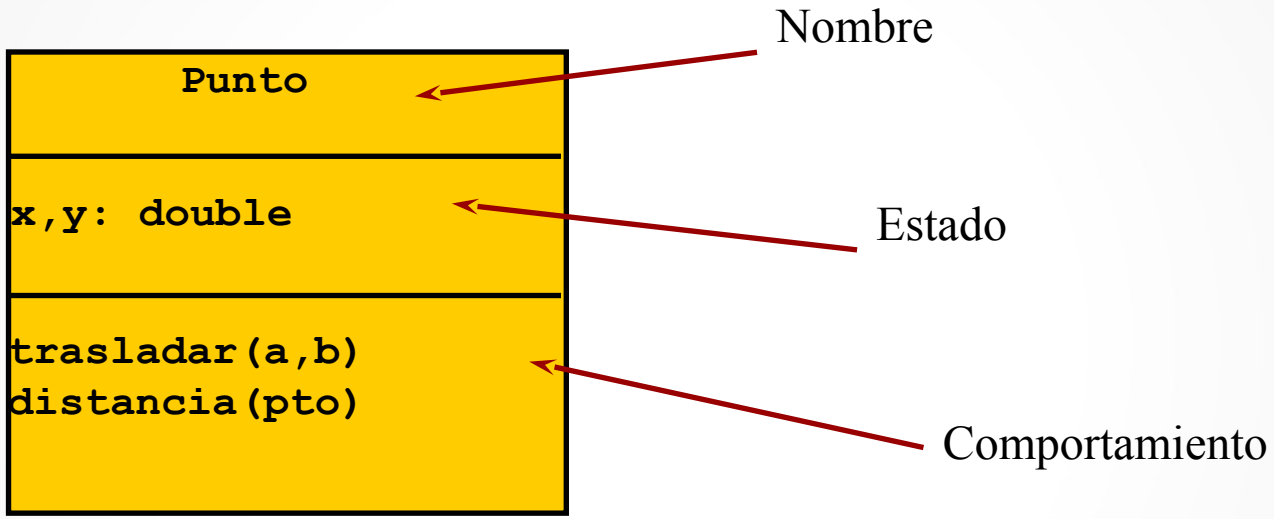
- **Estado**:
  - Viene determinado por los valores de sus atributos.
- **Comportamiento**:
  - Define cómo reacciona un objeto a peticiones de otros objetos.
  - Viene dado como una serie de servicios que el objeto proporciona a otros.

¿Qué es una clase?

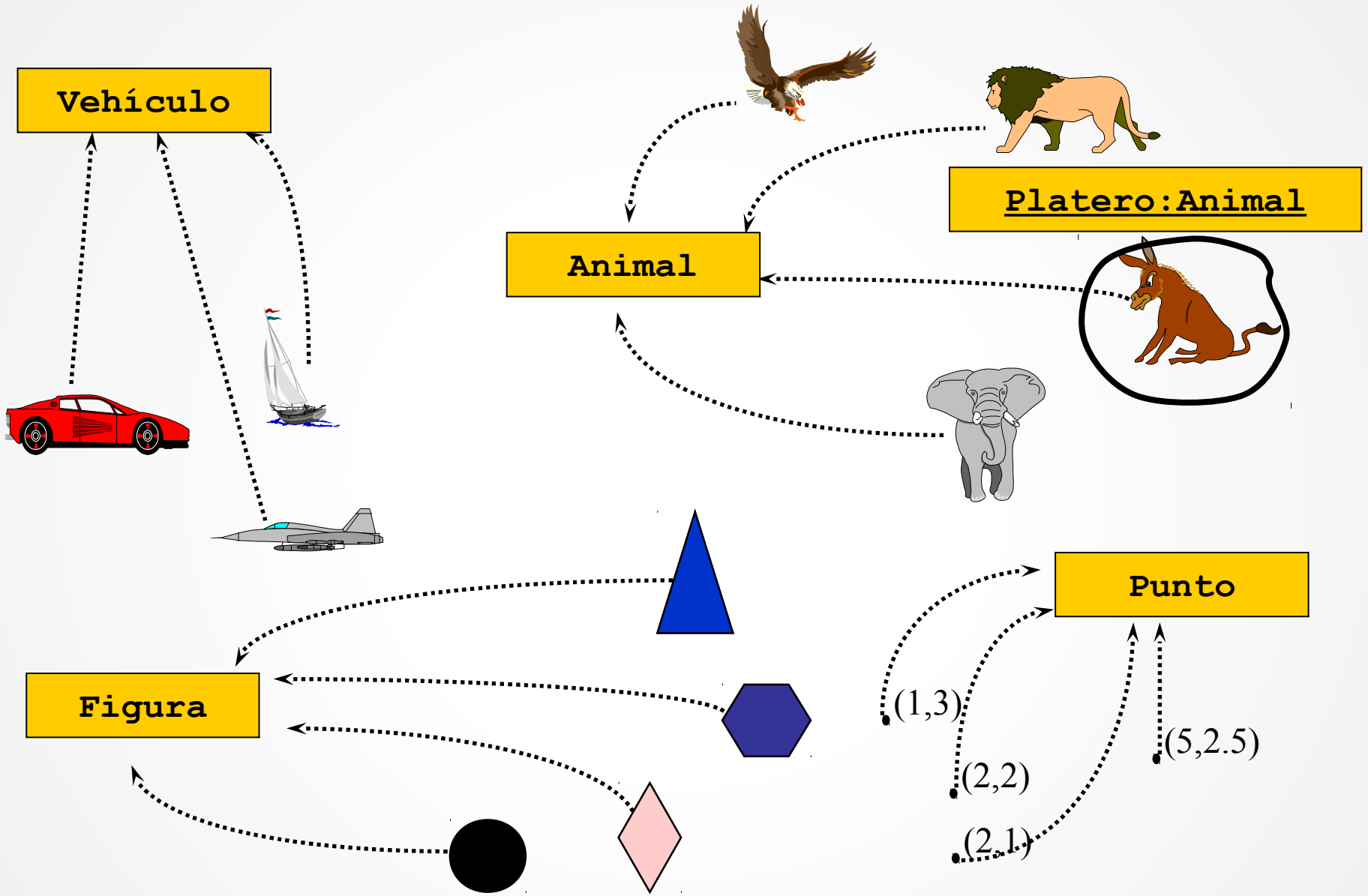
- Una **clase** es una plantilla para la creación de objetos que comparten características comunes:
  - Su estado está determinado por los mismos *atributos*.
  - Saben realizar las mismas operaciones (*también llamadas métodos*).
- Un objeto es una instancia de una clase
- Las clases deberán incluir al menos un *método constructor (casi todos los lenguajes incorporan uno por defecto*).

Representación gráfica

- Diagramas UML (*Unified Modelling Language*)



Las clases y objetos están en todas partes



¿Qué ocurre cuando se ejecuta un programa?

- Se crean objetos que luego residen en memoria.
- Los objetos colaboran entre sí para llevar a cabo las distintas tareas que el usuario solicita al programa.
  - Una tarea requiere cierta cantidad de operaciones.
  - Cada objeto sólo sabe realizar unas ciertas operaciones (*sus propios métodos*).
  - Si necesita realizar una operación que no sabe realizar, le pide a otro objeto que la realice (*llama a un método de otro objeto*).

Ocultación de Información

- Es un principio de diseño que se debe respetar para facilitar los futuros cambios en el programa.
- Un objeto debe ocultar sus atributos al resto de los objetos del programa ⇒**los atributos deben ser privados**
  - Esto se especifica en la definición de la clase
- Un objeto no tiene por qué ofrecer todas las operaciones que sabe realizar a los otros objetos del programa ⇒**habrá operaciones públicas y privadas**
  - Esto se especifica en la definición de la clase
- Los lenguajes de programación OO proporcionan modificadores para especificar la visibilidad de los atributos y métodos.

Para qué sirven los métodos

- **Constructores**: permiten dar contenido inicial a los objetos al crearlos
- **Observadores**: permiten obtener información del objeto, sin modificarlo
  - En particular los "*getters*" sirven para obtener el valor de un atributo determinado
- **Modificadores**: permiten cambiar el contenido del objeto
  - En particular los "setters" sirven para dar valor a un atributo determinado

Las Clases en Java

- Las clases son el elemento fundamental de los programas en Java.

- Todo programa consta de al menos una clase.
  - Esta clase contiene el `main()`

Punto de entrada

Definición de una clase en Java

[modificadoresClase] **class** *NombreClase* {

[modificadores] *Tipo nombreAtributo*

.....

[modificadores] *TipoResultado nombreMétodo(tipo1 arg1,..., tipoN argN)* {

"implementación del método o cuerpo" }

}

modificadorClase = **abstract | public | final**

modificador = **private | public | protected | final | static**

¿Cómo decidir qué modificador usar?

- Modificadores para métodos:
  - **private**: si se utiliza como función auxiliar
  - **public**: si va a ser utilizada fuera de la clase
- Los atributos deben ser privados
  - Si es necesario acceder al atributo se usarán *getters* y/o *setters*
    - Método getAtributo ()
    - Método setAtributo (valor)
- El resto de los modificadores se estudiarán más adelante en el curso.

Ejemplo de clase

```
public class Empleado {
   private String nombre;
   private double sueldo;
 public String getNombre() {
   return nombre;
 }
 public double getSueldo() {
   return sueldo;
 }
 public void setNombre(String nombre1) {
   nombre = nombre1;
 }
 public void setSueldo(double sueldo1) {
   sueldo = sueldo1;
 }
}
```

Método de instancia observador (*getter*)

Método de instancia asignador (*setter*)

Ejemplo de clase

```
public class TestEjemplo {

 public static void main(String[] args) {
   Empleado emp1;
   emp1 = new Empleado();
   emp1.setNombre("Pepe");
   emp1.setSueldo(1000);
   System.out.print("El sueldo de " +
                    emp1.getNombre() + " es " +
                    emp1.getSueldo());
  }
}
```

Creación de una instancia usando el constructor por defecto

Ciclo de vida de un objeto

- Supuesta la definición de la clase *NombreClase*

1. Declaración de una variable objeto (vacía)

```
NombreClase obj; // declaración
```

2. Creación de una instancia

```
obj = new NombreClase ();//creación e inicialización.
                         //obj es una referencia o puntero
```

3. Utilización de la instancia

```
obj.función(....); // invocación de un método
a = obj.atributo; // acceso a un atributo
//función y atributo tienen que ser accesibles por quien lo utiliza*.
```

Ciclo de vida de un objeto

4. Destrucción del objeto o instancia

- Automática
- El objeto ya no es referenciado por nadie:
  - A null todas las referencias del objeto
  - Se sale del ámbito

```
NombreClase obj= new NombreClase ();
{//Se entra en un ámbito
 NombreClase obj2= obj; // declaración y creación de una referencia
.....
} //Eliminación de la referencia obj2
obj= null; // Eliminación de la última referencia obj
```

- Entra en juego el **Garbage Collector** (GC)

Constructor

- Método especial
- Permite inicializar un objeto al crearlo
- Mismo nombre que la clase
- No puede devolver valor o dato
- Si no se define, java proporciona uno por defecto, que
  - Asigna a cada atributo de instancia su valor por defecto:

    0, false, 0l, 0.0f, null,'\0'
  - **No se inicializan las variables locales de un método**

Constructor

- El constructor se ejecuta al crear el objeto con el **new**
- Al llamar a **new** se asigna memoria al objeto
- El constructor puede tener parámetros
- Los parámetros se utilizan para inicializar los atributos
- Los parámetros se pasan al invocar a **new**
- Para notificar problemas al inicializar se utilizan excepciones (se verán más adelante)

Ejemplo 1 de constructores

```
public class Punto {
  private double x, y;

  public Punto(double xo, double yo) {
    x=xo; y=yo;
  }
  public String toString() {
    return "(" + x + "," + y + ")" ;
  }
}
```

Ejemplo 1 de constructores

```
public class TestPunto {
  public static void main(String[] args) {
    Punto punto1, punto2;
    punto1 = new Punto(-4, 11.3);
    punto2 = new Punto(3.1, 2);
    System.out.println("El primer punto es " + punto1);
    System.out.println("El segundo punto es " + punto2);
  }
}
```

Ejemplo 2 de constructores

```
public class Fecha {
  private int day, month, year;
  public Fecha(String date) {
    String [] partes = date.split("/");
    day = Integer.parseInt(partes[0]);
    month = Integer.parseInt(partes[1]);
    year = Integer.parseInt(partes[2]);
  }
  public Fecha(int dia, int mes, int anio) {
    day=dia; month=mes; year=anio;
  }
  public String toString() {
    return day + "/" + month + "/" + year;
  }
}
```

Sobrecarga del constructor

Ejemplo 2 de constructores

```
public class TestFecha {
public static void main(String[] args) {
    Fecha ob1, ob2;
    ob1 = new Fecha(4, 11, 1996);
    ob2 = new Fecha("22/10/2001");
    System.out.println("La primera fecha es " + ob1);
    System.out.print("La segunda fecha es " + ob2);
  }
}
```

Tipos básicos en Java

- Java maneja algunos datos que no son objetos
- Tipos básicos:
  - `int`: valores numéricos enteros
  - `float, double`: valores numéricos con parte fraccionaria
  - `boolean`: valores lógicos (`true, false`)
  - `char`: valores de tipo carácter de un texto

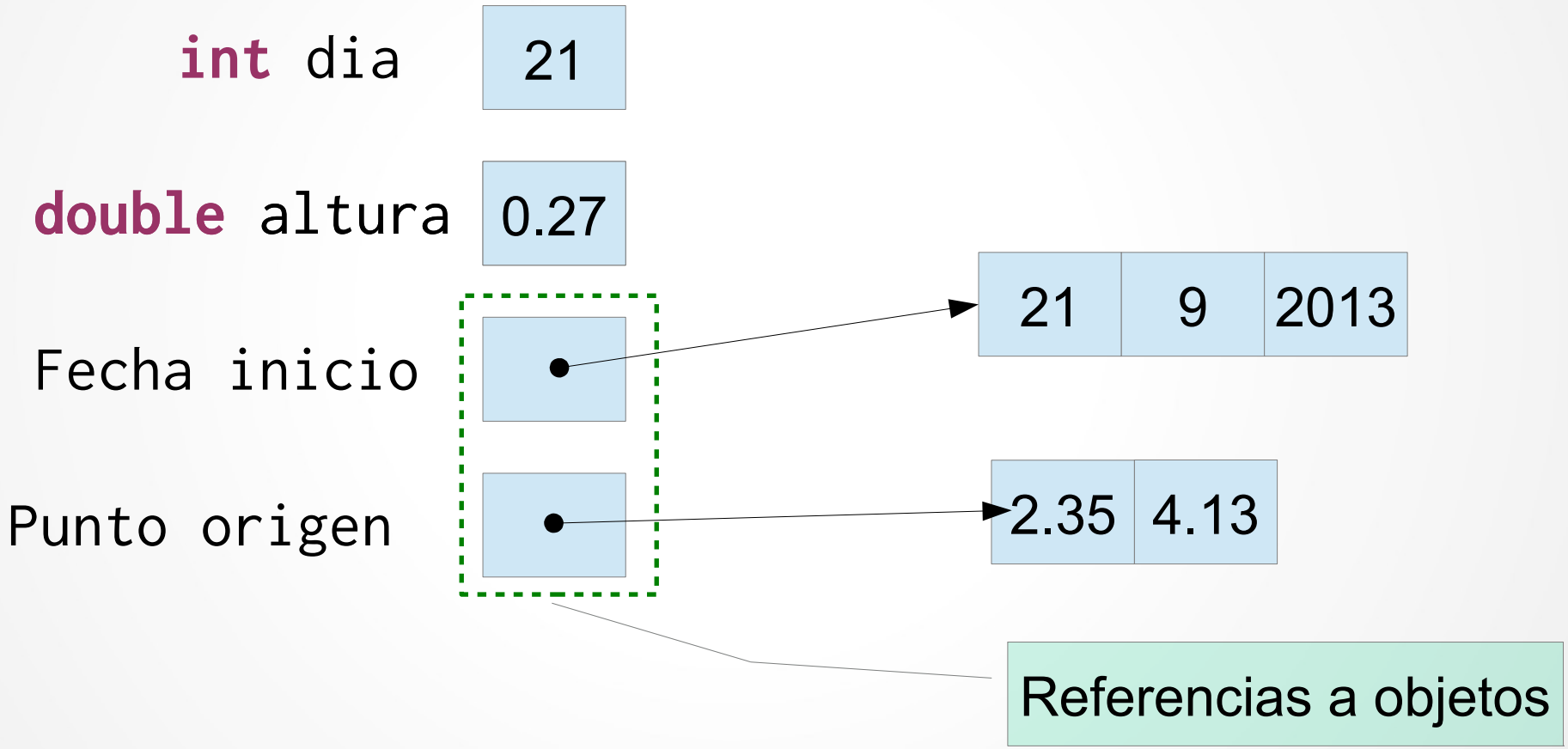
Variables en Java

- Una variable es un contenedor que puede almacenar información
- Una variable tiene siempre asociado un tipo o clase de valores
- Una variable puede tener nombre
- Una variable se declara indicando su tipo, nombre y (opcionalmente) el valor inicial

```
int año;
double longitud = 0.0;
Fecha inauguracion;
Fecha inicioCurso = new Fecha( 3, 9, 2013);
```

Valores y referencias (punteros)

- Una variable de un **tipo básico** contiene un **valor**
- Una **variable objeto** contiene una **referencia al objeto**



Uso de valores y referencias

- El hecho de que unas variables contengan valores y otras contengan referencias se traduce en comportamientos diferentes en ciertos casos, tales como:
- Asignación: operador **=**
- Comparación por igualdad: operador **==**
- Paso de argumentos a métodos: `operacion(arg1, arg2)`

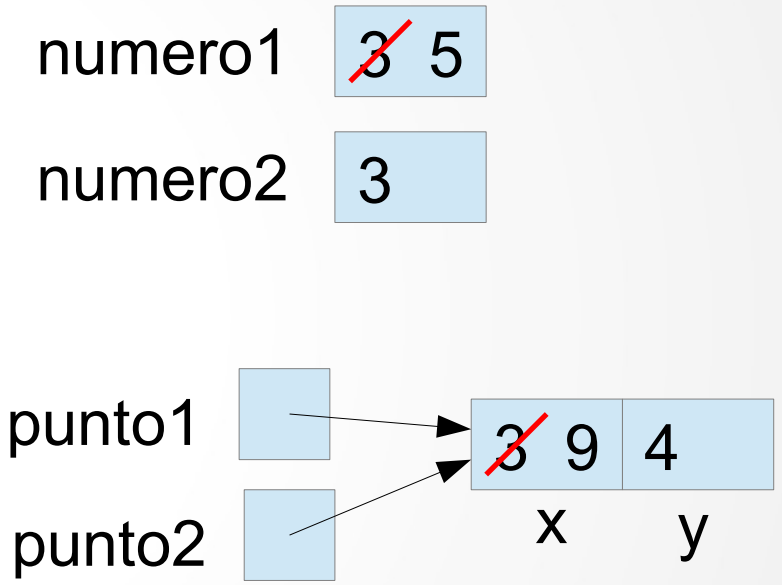
Asignación, ¿copia valor o referencia?

```
public static void main(String[] args) {
  int numero1, numero2;
  Punto punto1, punto2;

  numero1 = 3;
  numero2 = numero1;  // copia valor
  System.out.println("numero1=" + numero1 + "; numero2=" + numero2);
  numero1 = 5;
  System.out.println("numero1=" + numero1 + "; numero2=" + numero2);

  punto1 = new Punto(3, 4);
  punto2 = punto1;    // copia referencia
  System.out.println("punto1=" + punto1 + "; punto2=" + punto2);
  punto1.setX(9);
  System.out.println("punto1=" + punto1 + "; punto2=" + punto2);
}
```

```
numero1=3; numero2=3
numero1=5; numero2=3
punto1=(3.0,4.0); punto2=(3.0,4.0)
punto1=(9.0,4.0); punto2=(9.0,4.0)
```

Asignación, ¿copia valor o referencia?

```
int numero1, numero2;
Punto punto1, punto2;

numero1 = 3;
numero2 = numero1;  // copia valor

numero1 = 5;


punto1 = new Punto(3, 4);
punto2 = punto1;    // copia referencia

punto1.setX(9);
```



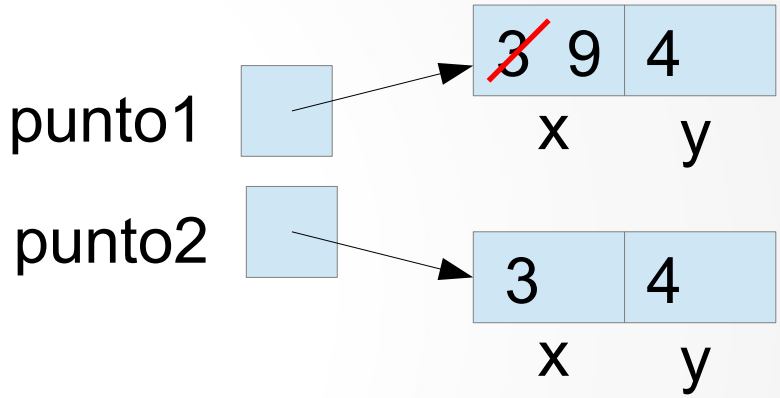
Asignar valor de un objeto

- Para copiar objetos (su valor) hay que crear un duplicado
- Para ello se puede usar un constructor de copia

```
public Punto(Punto p) {    // constructor de copia
    x = p.x;
    y = p.y;
}

....
    punto1 = new Punto(3, 4);
    punto2 = new Punto(punto1);    // copia valor (duplicado)
    System.out.println("punto1=" + punto1 + "; punto2=" + punto2);
    punto1.setX(9);
    System.out.println("punto1=" + punto1 + "; punto2=" + punto2);
....
```

Asignar valor de un objeto

```
punto1 = new Punto(3, 4);
punto2 = new Punto(punto1);

punto1.setX(9);
```



Igualdad

- El operador **==** compara el contenido directo de las variables
- Por tanto, si las variables son objetos, compara referencias

```
Punto punto1, punto2;

punto1 = new Punto(3, 4);
punto2 = punto1;     // copia referencia
System.out.println("punto1==punto2: " + (punto1==punto2)); // cierto

punto2 = new Punto(punto1); // copia valor (duplicado)
System.out.println("punto1==punto2: " + (punto1==punto2)); // ¡falso!
```

- En Java el operador **==** representa la identidad de objetos

Igualdad

- Para evaluar si dos objetos contienen los mismos valores hay que definir un método de comparación. En Java se suele llamar equals().

```
public class Punto {
  private double x, y;
....
  // Puntos iguales si tienen las mismas coordenadas
  public boolean equals(Punto p) {
    return x == p.x && y == p.y;
  }
....
}
```

Igualdad

- La función equals() representa la equivalencia entre puntos

```
Punto punto1, punto2;

punto1 = new Punto(3, 4);
punto2 = punto1;    // copia referencia
System.out.println(
  "punto1.equals(punto2): " + punto1.equals(punto2)); // cierto

punto2 = new Punto(punto1); // copia valor (duplicado)
System.out.println(
  "punto1.equals(punto2): " + punto1.equals(punto2)); // cierto
```

Comparar objetos que contienen objetos

- Si un atributo es un objeto, hay que compararlo también con la correspondiente función equals() o similar

```
public class Linea {
  private Punto inicio, final;
....
  // Líneas iguales si tienen los mismos extremos
  public boolean equals(Linea lin) {
    return inicio.equals(lin.inicio) && final.equals(lin.final);
  }
....
}
```

Uso riguroso de equals() en Java

Nota informativa:

- En Java todas las clases (incluso las definidas por el usuario) disponen de un método predefinido para comparar por igualdad

```
public boolean equals( Object obj )
```

- El código predefinido de este método funciona como el operador `==`, pero puede ser redefinido. De hecho las clases de la librería estándar de Java tienen este método redefinido de manera adecuada
- ¡Ojo! El argumento es un `Object` (puede ser un objeto cualquiera) y no un objeto de la misma clase que el que se compara. Por lo tanto, para acceder a los atributos del objeto argumento habrá que hacer un *casting* o conversión de tipo
- Esto quiere decir también que cuando definimos `equals(`*`MismaClase`*`)` estamos definiendo un método diferente al estándar de Java
- Más adelante se verá cómo redefinir equals(Object) para que funcione con objetos de la misma clase

Paso de parámetros

- Algunos lenguajes de programación (C++, ADA, MODULA-2, etc.) pueden utilizar dos formas alternativas de pasar los parámetros al llamar a un subprograma:
  - **Paso por valor** (para los datos de entrada que recibe el subprograma)
    - Los datos de entrada no deben ser modificados
    - Se saca una copia del dato original que se pasa como argumento
    - Se tienen 2 valores duplicados e independientes, por tanto los cambios en uno no afectan al otro
  - **Paso por referencia** (para los datos de entrada/salida o salida)
    - Los datos van a ser inicializados o modificados dentro del subprograma
    - Se pasa una referencia al dato original
    - Cualquier acción implica cambio en los 2 valores (el que está fuera y el que está dentro del subprograma)

Paso de parámetros en Java

- El paso de parámetros en Java sólo es por valor
  - Se hace una copia del argumento en el contexto del método.
- **Pero, OJO:**
  - En Java las **variables de tipo objeto** realmente **contienen referencias a objetos**
  - Si pasamos como argumento un objeto, pasamos una copia de una referencia a él.
  - Los cambios en los objetos afectan **a todas sus referencias**
  - El objeto que se pasa como argumento se puede modificar dentro del método (dato de entrada/salida o salida)

Ejemplo: intercambiar dos variables

- No se puede implementar un método que intercambie dos argumentos de los tipos básicos

```
static void intercambiar(int p, int q) {
  int aux = p;
  p = q;
  q = aux;
}

public static void main(String[] args) {
  int a=3; int b=7;
  System.out.println(a + " " + b);  // --> 3 7

  int aux = a;
  a = b;
  b = aux;
  System.out.println(a + " " + b);  // --> 7 3

  intercambiar(a,b);  // no intercambia
  System.out.println(a + " " + b);  // --> 7 3
}
```

Ejemplo: intercambiar dos variables

- El esquema de código anterior tampoco funciona con argumentos que sean objetos (referencias)

```
static void intercambiar(Punto p, Punto q) {
  Punto aux = p;
  p = q;
  q = aux;
}

public static void main(String[] args) {
  Punto a = new Punto(3,7);
  Punto b = new Punto(4,8);
  System.out.println(a + " " + b);  // --> (3,7) (4,8)
  Punto aux = a;
  a = b;
  b = aux;
  System.out.println(a + " " + b);  // --> (4,8) (3,7)
  intercambiar(a,b);  // no intercambia
  System.out.println(a + " " + b);  // --> (4,8) (3,7)
}
```

Ejemplo: intercambiar dos variables

- Para intercambiar objetos hay que intercambiar sus contenidos

```
static void intercambiar(Punto p, Punto q) {
  double xx = p.x; p.x = q.x; q.x = xx;
  double yy = p.y; p.y = q.y; q.y = yy;
}

public static void main(String[] args) {
  Punto a = new Punto(3,7);
  Punto b = new Punto(4,8);
  System.out.println(a + " " + b);  // --> (3,7) (4,8)
  Punto aux = a;
  a = b;
  b = aux;
  System.out.println(a + " " + b);  // --> (4,8) (3,7)
  intercambiar(a,b);  // ahora sí intercambia
  System.out.println(a + " " + b);  // --> (3,7) (4,8)
}
```

Puntero this

- Todo **método de instancia** lleva un parámetro implícito
- Este parámetro es una referencia al objeto sobre el que se hace la llamada
- Esta referencia se llama **this**
- Se puede usar para evitar colisiones de identificadores dentro de un método

```
public class Fecha{
  .....
  public Fecha(int dia, int mes, int año) {
    this.dia=dia; this.mes=mes; this.año=año;
  }
}
```

Puntero this

- También se puede usar para pasar como parámetro el objeto actual, como en el siguiente ejemplo ....

```
class Avion{
  private Aeropuerto destino;
  private Aeropuerto origen;

  public void setDestino (Aeropuerto destino) {
    this.destino = destino;
  }
  public void aterriza () {
    Pista pista;
    pista = this.destino.darPista(this);
    ……
  }
}
```

Puntero this

- …. ejemplo (continuación) ....

```
class Aeropuerto {
    private Pista pista;
    public Pista darPista(Avion avion) {
        if (puedeAterrizar(avion)) {
            return pista;
        } else {
            return null;
        }
    }
}
```

Puntero this

- …. ejemplo (continuación) ....

```
public static void main(String[] args){
  Aeropuerto aeropuerto;
  Avion avion;
  avion = new Avion();
  aeropuerto = new Aeropuerto("barajas");
  avion.setDestino(aeropuerto); // this == avion
  avion.aterriza(); // this == avion
}
```

Puntero this

- Otro ejemplo

```
class Ventana {
   private Ventana ventanaPadre;
   public Ventana (Ventana ventanaPadre) {
       this.ventanaPadre = ventanaPadre;
   }
   public void solicitarInformacion () {
       Ventana ventanaInfo = new Ventana (this);
       ventanaInfo.hacerVisible ();
   }
}
```

Algunas clases especiales

- En Java todos los datos que se manejan, a excepción de los de los tipos básicos, son objetos.
- Para facilitar la redacción de los programas, ciertos elementos de información, aun siendo objetos, se declaran y/o manejan de manera diferente de las clases normales. Esto incluye:
  - Las clases “envoltorio” (*wrappers*)
  - La clase `String`
  - Los tipos enumerados
  - Las estructuras “*array*”
  - .... etc. ....

Clases Asociadas a los Tipos Básicos

- Java proporciona **clases envoltorio (*wrappers*)**, que permiten operar con valores de los tipos básicos como si fueran objetos
- Tienen interés para aplicarlas al usar genéricos (se verán más adelante)
- De momento se usarán para convertir valores:

  Integer.valueOf(*string*) → *integer*

Clase *wrapper*	Tipo básico
`Boolean`	`boolean`
`Byte`	`byte`
`Character`	`char`
`Double`	`double`
`Float`	`float`
`Integer`	`int`
`Long`	`long`
`Short`	`short`

La clase String

- Es algo similar a un *array* de caracteres, pero con más facilidades de uso.
- Literales entre comillas dobles (`"ejemplo"`)
- Operador de concatenación (`"ejemplo" + "2"`)
- Fuerza conversión al concatenar (`"ejemplo" + 2`)
- Consulta del número de caracteres: *`string`*`.length()`
- Puede usarse como selector en `switch` (desde Java 7)
- **Una vez creado no se puede cambiar el valor de su contenido**
- `StringBuffer` define `Strings` que pueden ser modificados

Algunos métodos para uso de Strings

- `char charAt(int pos)`
- `int compareTo(String otro)`:
  - Resultado cero si son iguales
  - Resultado negativo si "otro" es mayor
  - Resultado positivo si "otro" en menor
- `int compareToIgnoreCase(String otro)`
- `boolean equals(Object otro)`
- `boolean equalsIgnoreCase(String otro)`
- `int length()`
- `static String valueOf(<tipo básico> valor)`
- `String substring(int posInicio, int posFin)`

"hola".compareTo(nombre)

String pruebas=String.*valueOf(15);*

Tipos definidos por enumeración

- Es posible definir nuevos tipos de datos, simples, a base de enumerar todos los valores posibles para los datos de ese tipo
- Los valores se designan mediante identificadores, y son abstractos, es decir, no tienen ningún significado en sí mismos, sólo el significado que se les dé en el programa que los usa
- Ejemplo: tipos de piezas del ajedrez
  PEON, CABALLO, ALFIL, TORRE, DAMA, REY
- Se definen como una clase, usando la palabra **`enum`** en lugar de **`class`**. La definición puede contener simplemente la lista de valores posibles. Cada valor será un objeto de la clase

```
enum Palo {OROS, COPAS, ESPADAS, BASTOS}
```

Uso de tipos definidos por enumeración

- Los valores se nombran anteponiendo el nombre de la clase: `Palo.OROS, PiezaAjedrez.CABALLO`
- Los tipos enumerados no necesitan tener atributos ni métodos, aunque se pueden definir si se desea
- Los tipos enumerados se pueden usar como selector en sentencias **`switch`**
- Los valores del tipo enumerado se pueden convertir en números correlativos (0, 1, 2, …) y viceversa

```
Palo.OROS.ordinal() → 0
Palo.values()[1] → Palo.COPAS
```

Atributos de Clase

- Se definen anteponiendo la palabra clave **`static`**.
- Existen sin necesidad de que ya haya sido creada una instancia u objeto de la clase
- Todos los objetos comparten el mismo valor en ese atributo.
- No existe una copia de ese atributo para cada instancia de la clase u objeto.
- USO: cuando todos los objetos de una misma clase comparten unos mismos datos.
  - Se pueden utilizar para definir constantes (**`final`**) (véanse las constantes `PI` y `E` de la clase `Math`)
  - Para datos comunes para todas las instancias de una clase

Métodos de Clase

- Se definen anteponiendo la palabra clave **`static`**.
- Se usan sin necesidad de que ya haya sido creada una instancia u objeto de la clase:

```
NombreClase.nombreMetodo(...)
```

- Sólo pueden acceder a atributos de clase y a métodos de clase.
- USO:
  - Inicialización, modificación y consulta de atributos de clase.
  - Métodos que no necesitan atributos del objeto actual: `parseInt()`, `parseDouble()`, etc.

Atributos y Métodos de Clase

```
public class CuentaCorriente {
/**
 * Los intereses son comunes para todas las cuentas
 * abiertas en el banco
 */
      private static double interes = 0.5;
      private double saldo;

      public CuentaCorriente (double saldoInicial){
            saldo=saldoInicial;
      }
      public void setSaldo (double saldo){
            this.saldo=saldo;
      }
      public static void setInteres (double interes){
            CuentaCorriente.interes=interes;
      }
      public String toString(){
            return interes + " " + saldo;
      }
}
```

Permite resolver la ambigüedad

Atributos y Métodos de Clase

```
public class TestCuentaCorriente {

  public static void main(String[] args) {
    CuentaCorriente cuenta1, cuenta2;
    cuenta1 = new CuentaCorriente(23);
    cuenta2 = new CuentaCorriente(40);
    System.out.println(cuenta1);
  // La forma correcta es CuentaCorriente.setInteres(1.5)
    cuenta2.setInteres(1.5);
  // El interés de cuenta1 es el mismo de cuenta2 y
  // cuando se cambia en uno se cambia en los dos
    System.out.println(cuenta2);
    System.out.println(cuenta1);
  //Cambiamos el interés a todas las cuentas usando el
  // nombre de la clase
    CuentaCorriente.setInteres(0.75);
    System.out.println(cuenta2);
    System.out.println(cuenta1);
  }
}
```

Arrays de objetos

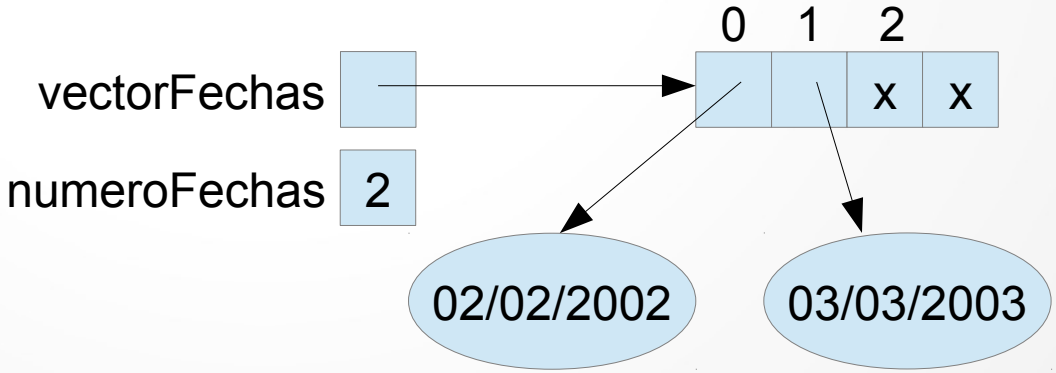
```
public static void main(String[] args) {

   Fecha [] fechas; // declaración del array

   fechas = new Fecha[3];

   fechas[0]=new Fecha(1,1,06);
   fechas[1]=new Fecha(2,1,06);
   fechas[2]=new Fecha(3,1,06);

   for (int i=0; i<fechas.length; i++)
       System.out.println(fechas[i]);
}
```

Se crea espacio para tres referencias a Fecha

Se crea la primera instancia y es referenciada por fechas[0]

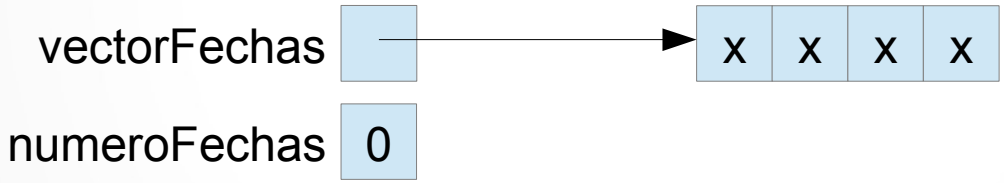
Arrays parcialmente llenos

- **Una vez creado un vector no cambia de tamaño**
- Para cambiar el número de elementos se puede definir el vector con capacidad para el máximo número de elementos, pero sólo se ocupan los primeros
  - Se puede usar una variable que lleve la cuenta de los elementos que contiene el vector
  - Sólo se procesan los elementos ocupados, no se necesita recorrer el vector entero
  - Si tenemos un vector de objetos Fecha con capacidad de 4, pero solo hemos guardado 2 Fechas, debería ser:



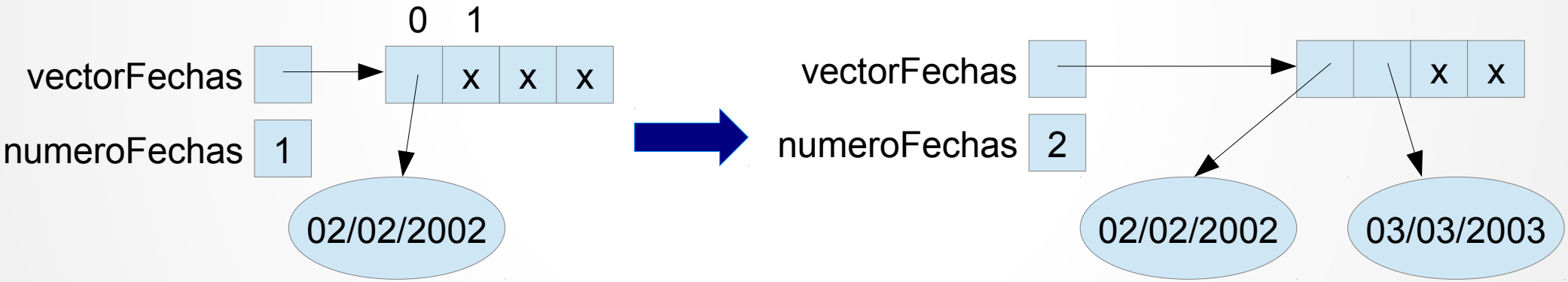
Creación del *array*

- Hay que prever el número **máximo** de elementos que pueden almacenarse en el vector
  - Se declarará un *array* vacío, con la capacidad dada
  - Inicialmente el número efectivo de elementos en el *array* será 0



Añadir un elemento

- Si se quiere **insertar** un nuevo elemento, dicho elemento debe meterse al **final** de los existentes
  - Si el *array* actualmente contiene **1** elemento y se quiere introducir otro, el nuevo se meterá en la posición ... **1**
  - Se actualiza el número de elementos en el *array*

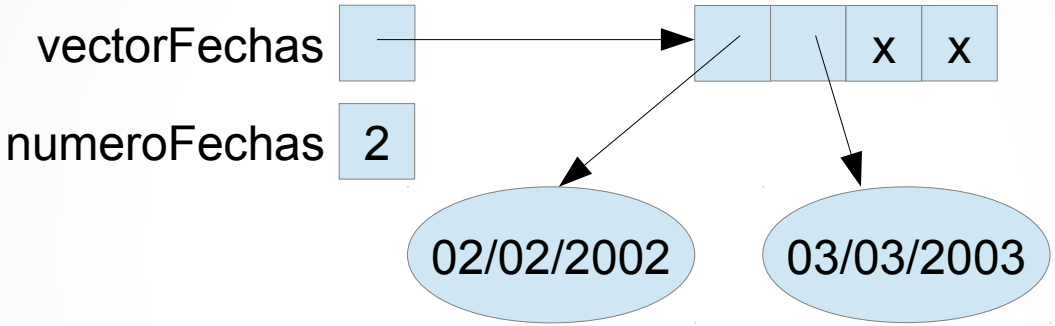


Añadir un elemento

- Si se quiere **insertar** un nuevo elemento, dicho elemento debe meterse al **final** de los existentes
  - Si el *array* actualmente contiene **n** elementos y se quiere introducir otro, el nuevo se meterá en la posición ... **n**.
  - Se actualiza el número de elementos en el *array*

Recorrido del *array*

- Si se va a **recorrer** el vector
  - Desde la posición 0 hasta la posición *numElementos*-1.
  - Ejemplo: Si tenemos el vector

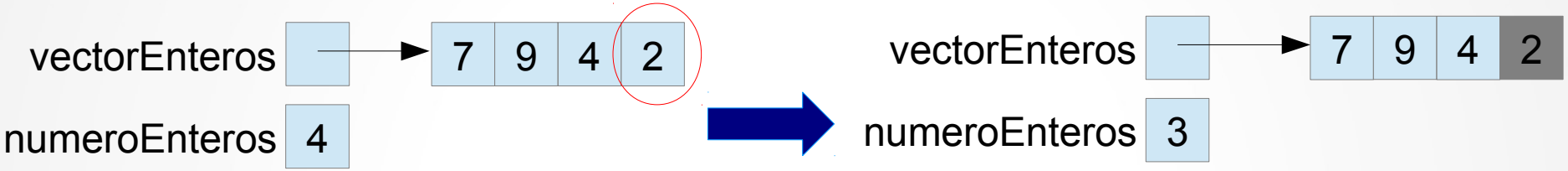


Se recorrerá desde la posición 0 hasta la 2-1 (1)

```
for (int i=0; i<numeroFechas; i++) {
    System.out.println(vectorFechas[i]);
}
```

Eliminar un elemento

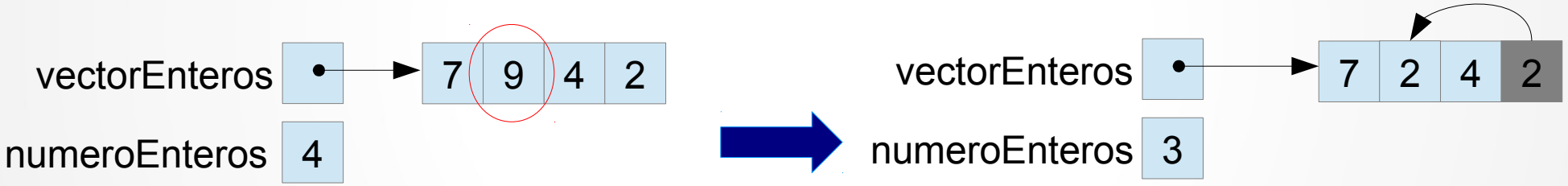
- Caso particular: se quiere **borrar** el **último** elemento del vector
  - Basta con decrementar el número de elementos



  - Cuando lo recorramos pararemos uno antes, y no se accederá al cuarto elemento

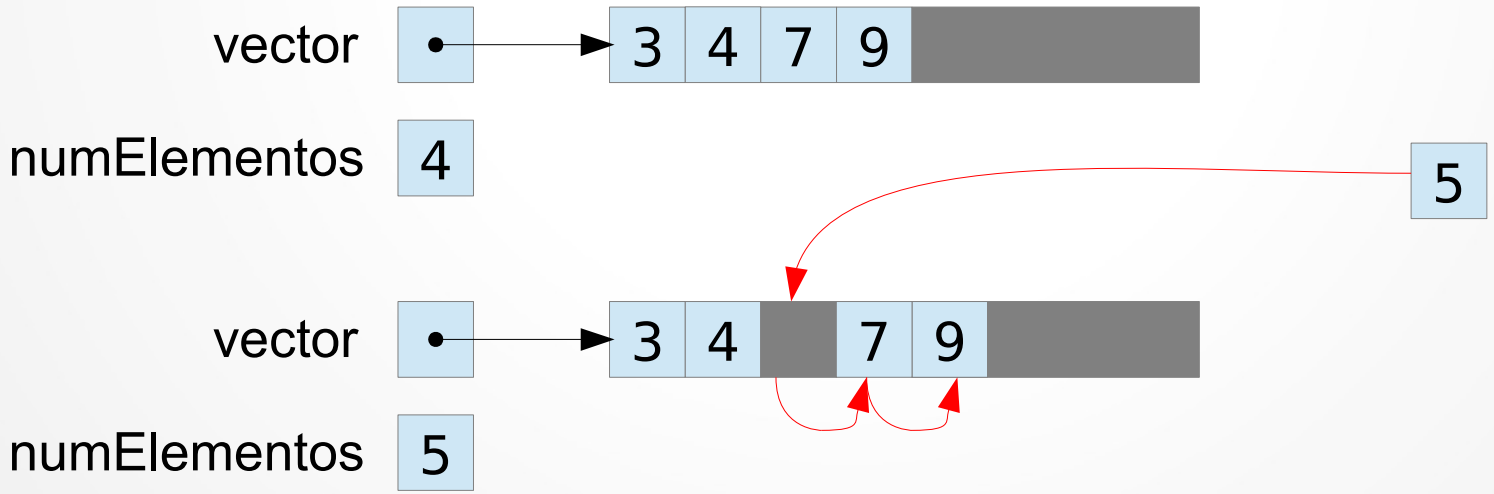
Eliminar un elemento

- Caso general: **borrar** un elemento que **no es el último**; debe rellenarse un hueco dentro del *array*:
  - Solución simple: **desplazar** el elemento situado en la **última** posición **a la posición que se quiere borrar**.
  - Se decrementa en uno el número de elementos en el *array*



Ordenación por inserción (construcción)

- Para aprovechar las ventajas de la búsqueda binaria los elementos del vector deben estar en orden.
  - Se pueden ir manteniendo en orden mientras se añaden elementos al vector.
  - Para mantener el orden hay que mover una posición hacia el final cada uno de los elementos mayores que el que se inserta.



Insertar un elemento

```
int[] vector = ....;
int numElementos = ....;
int nuevo;

int pos = numElementos;
while (pos > 0 && vector[pos-1] > nuevo) {
    vector[pos] = vector[pos-1];
    pos--;
}
vector[pos] = nuevo;
numElementos++;
```

Retirar un elemento

- Hay que mover una posición hacia el principio cada elemento detrás del que se elimina, para rellenar el hueco.

```
int[] vector = ....;
int numElementos = ....;

int pos = ....; // posición del elemento a retirar

for (int k = pos + 1; k < numElementos; k++) {
    vector[k-1] = vector[k];
}
numElementos--;
```

Ordenación por intercambios (burbuja)

- Si el vector no se ha construido ya ordenado, entonces hay que ordenarlo cuando haga falta.
- El método de ordenación más sencillo de programar es el de la **burbuja** (*bubble sort*).
  - Consiste en ir comparando cada pareja de elementos consecutivos, e intercambiarlos si no están en orden.
  - Se realizan pasadas sucesivas sobre el vector hasta que ya esté todo ordenado.
  - En cada pasada el mayor elemento que no esté en su sitio avanza hasta colocarse en su posición.

Ordenación por intercambios (burbuja)

```
int[] vector = ....;
int numElementos = ....;

int final = numElementos - 1;
int aux;
for (int k = final; k > 0; k--) {
  for (int j = 0; j < k; j++) {
    if (vector[j] < vector[j+1]) {
      aux = vector[j];
      vector[j] = vector[j+1];
      vector[j+1] = aux;
    }
  }
}
```