# Programación en Lenguaje Java

## Tema 6. Clases, referencias y objetos


**Michael González Harbour**

**Mario Aldea Rivas**

Departamento de Matemáticas,
Estadística y Computación

# Programación en Java

1. Introducción a los lenguajes de programación
2. Datos y expresiones
3. Estructuras algorítmicas
4. Datos Compuestos
5. Entrada/salida
6. *Clases, referencias y objetos*
   - Creación e inicialización de objetos. Tipos primitivos, referencias y objetos. Comparación de objetos. Recolector de basura. Métodos y campos de clase (o estáticos). Anidamiento de clases
7. Modularidad y abstracción
8. Herencia y polimorfismo
9. Tratamiento de errores
10 ..., 11 ...

# 6.1 Creación e inicialización de objetos

En el lenguaje Java los objetos se crean con el operador `new`:

```
UnaClase obj = new UnaClase();
```

Se invoca el CONSTRUCTOR

Los constructores son métodos especiales que

- se llaman igual que la clase
- no tienen tipo de retorno

Una clase puede tener varios constructores con diferentes parámetros

```
public UnaClase() { … }
public UnaClase(int i) { … }
public UnaClase(boolean b, int i) { … }
```

Si ***y solo si*** no definimos constructor, la clase tendrá el constructor por defecto (constructor sin parámetros que no hace nada)

```
public UnaClase() {}
```

# Creación implícita de objetos

En unos pocos casos la creación de objetos se realiza sin que el operador **new** aparezca explícitamente:

- Creación de strings utilizando un literal de string:

```
String str = "Hola";
```

Equivale[1] a:

```
String str = new String("Hola");
```

- Concatenación de strings:

```
String str1 = "Hola";
String str2 = "Adiós";
String str3 = str1 + str2;
```

Equivale a:

```
String str3 = new String("HolaAdiós");
```

1. No son equivalentes desde el punto de vista de la implementación de ambas instrucciones por parte de la máquina virtual (si se tiene curiosidad, buscar información sobre "String constant pool").

- Inicialización de array:

```
int[] números = {1, 3, 5};
```

Equivale a:

```
int[] números = new int[3];
números[0] = 1;
números[1] = 3;
números[2] = 5;
```

# Inicialización de los atributos

Cuando se crea un objeto sus atributos se inicializan en 3 etapas:

1. Se asigna a cada atributo su valor por defecto
   - tipos numéricos (`int`, `double`, …): 0 (o 0.0)
   - Caracteres (`char`): carácter nulo
   - Booleanos (`boolean`): `false`
   - Referencias a objetos (`String`, otras clases): `null`
2. Si la declaración del atributo contiene un inicializador, éste es evaluado y asignado

```
private int numRuedas = numCoches * 4;
```

3. Se ejecuta el constructor de la clase, que puede cambiar el valor de los atributos que desee

Recordar: sólo se inicializan los atributos, las variables locales de los métodos no son inicializadas por defecto

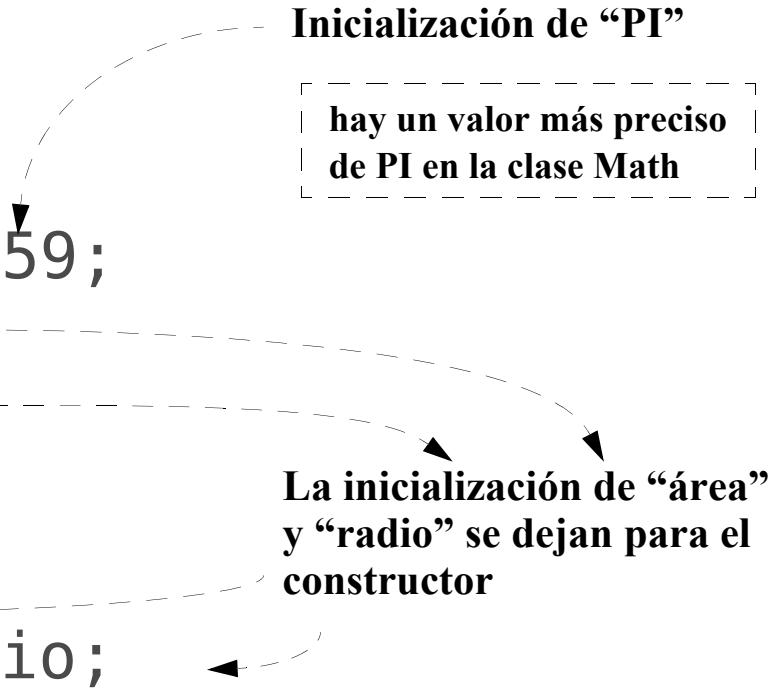
# Atributos “finales”

Su valor no se puede cambiar después de haber sido inicializados

- pueden inicializarse en la definición del atributo o en el constructor

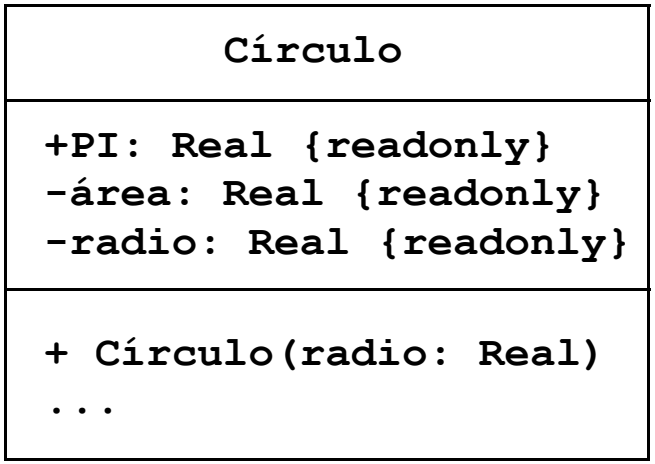
Ejemplo:

```
public class Círculo {
  public final double PI = 3.14159;
  private final double área;
  private final double radio;

  public Círculo(double radio) {
    this.radio = radio;
    this.área = PI * radio * radio;
  }
  … // los demás métodos de la clase no pueden
    // cambiar el valor de los atributos finales
```



# Atributos finales en UML

Se utiliza el modificador {readonly}




```
public class Círculo {

  public final double PI = 3.14159;
  private final double área;
  private final double radio;

  public Círculo(double radio) {
    this.radio = radio;
    this.área = PI * radio * radio;
  }

  …
```

# 6.2 Tipos primitivos, referencias y objetos

***Tipos primitivos***:

- `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` y `double`.
- Cuando se declarara una variable de un ***tipo primitivo*** el compilador reserva un área de memoria para ella

```
int i, j;
```

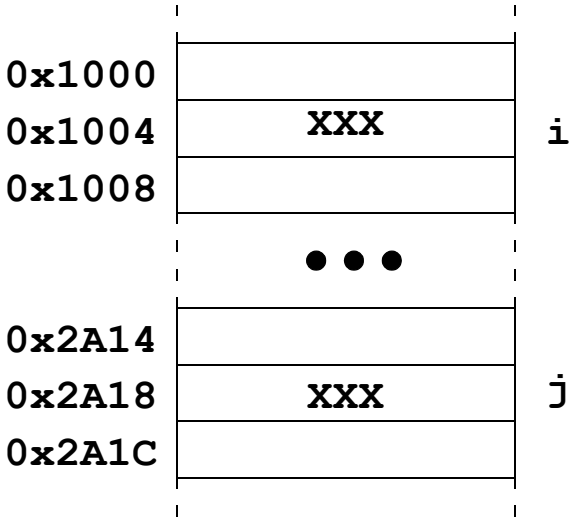

- Cuando se asigna un valor, éste es escrito en el área reservada

```
i=16;
j=3;
```


- La asignación entre variables significa copiar el contenido de una variable en la otra

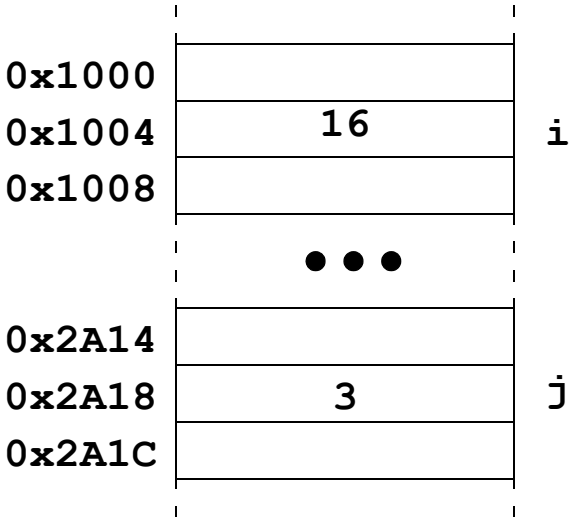
```
i=j;
```

# Tipos primitivos: representación en memoria

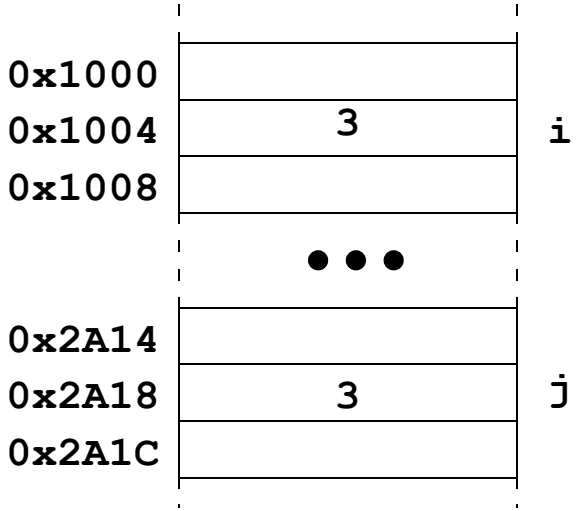



```
int i, j;
```

```
i=16;
j=3;
```




```
i=j;
```



# Referencias y objetos

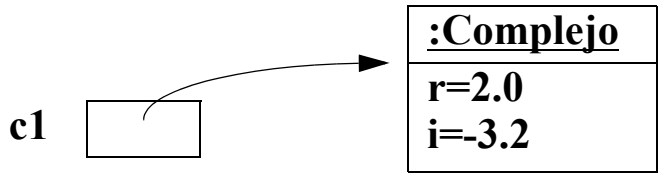
Cuando se **declara** una variable "objeto" la situación es diferente

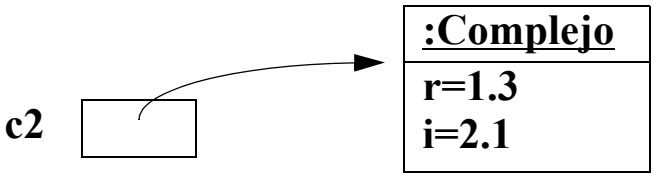
- Si no se usa el operador new sólo se crea una referencia a objeto

```
Complejo c1,c2;
```


- El espacio en memoria del objeto se reserva con el operador new

```
c1=new Complejo(2.0,-3.2);
c2=new Complejo(1.3,2.1);
```



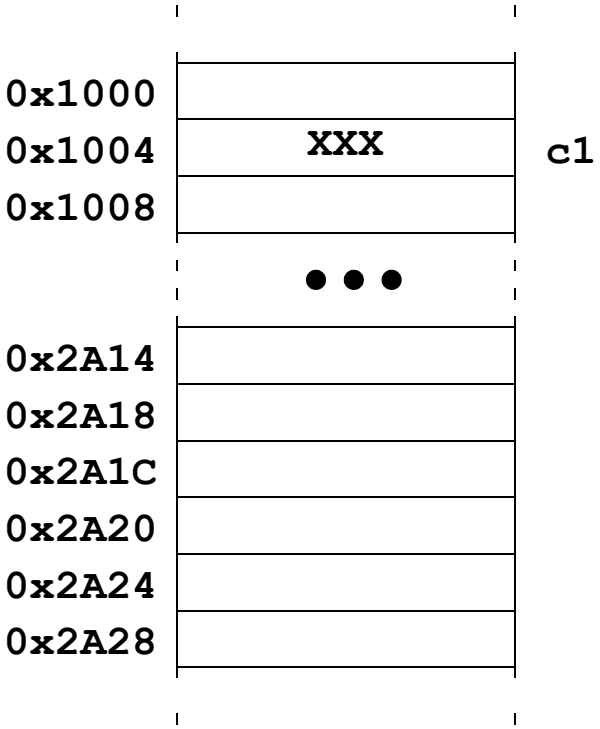



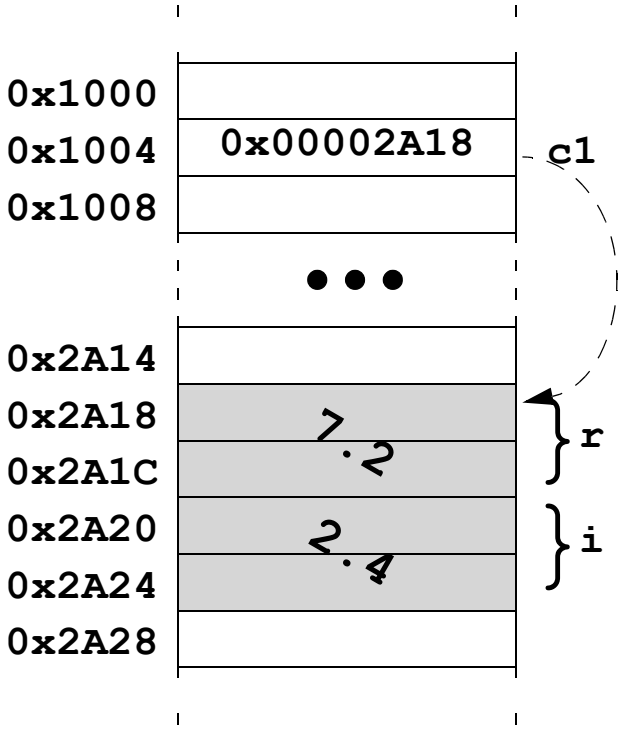

```
public class Complejo {
 private double r;
 private double i;
 …
}
```

# Representación en memoria:

```
Complejo c1;
```

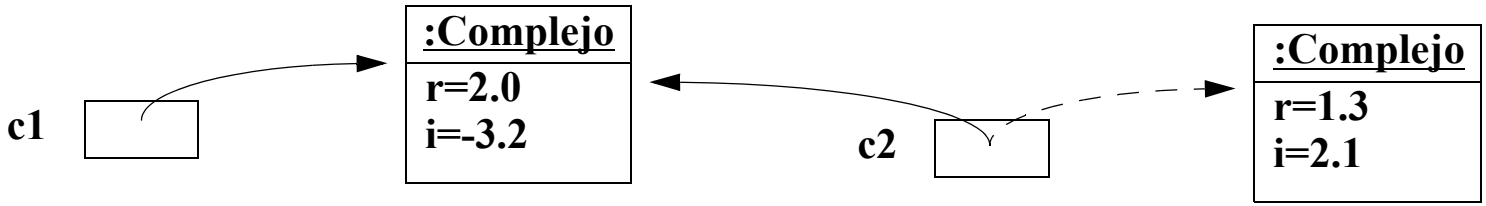
```
c1 = new Complejo(7.2, 2.4);
```



- La asignación entre referencias no copia los objetos
- después de la copia ambas referencias apuntan al mismo objeto (y los objetos no han sido modificados)

```
c2=c1;
```




- La comparación entre referencias no compara los contenidos de los objetos
- en la figura anterior `c1==c2` es `true`
- en la situación de debajo `c1==c2` es `false`


- el método `equals` permite comparar contenidos (Ver 6.3)

# Arrays de tipos primitivos

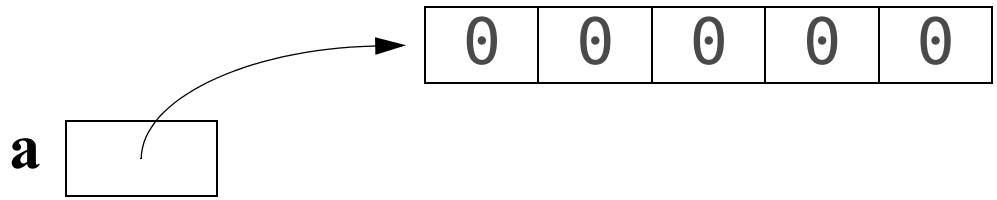
Java trata los arrays como objetos

- Crear una variable de tipo array sólo crea una referencia

```
int[] a;
```


- El espacio en memoria para el array se crea con new
  - cuando se trata de un array de un tipo primitivo se crean los elementos del array
  - y se inicializan a sus valores por defecto

```
a = new int[5];
```



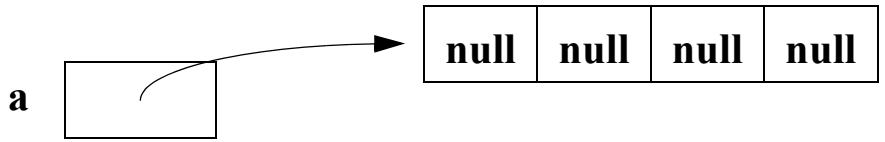
# Arrays de referencias a objetos

- Crear una variable de tipo array de objetos sólo crea la referencia

```
Complejo a[];
```

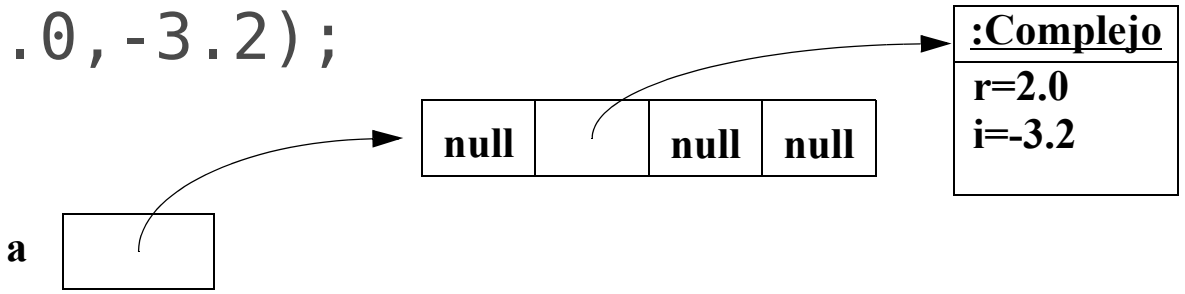

- Con `new` se reserva el espacio para el array de referencias
  - pero ***no los objetos*** a los que apuntarán las referencias
  - las referencias se inicializan a su valor por defecto (`null`)

```
a=new Complejo[4];
```




- Los objetos hay que crearlos posteriormente

```
a[1]=new Complejo(2.0,-3.2);
```

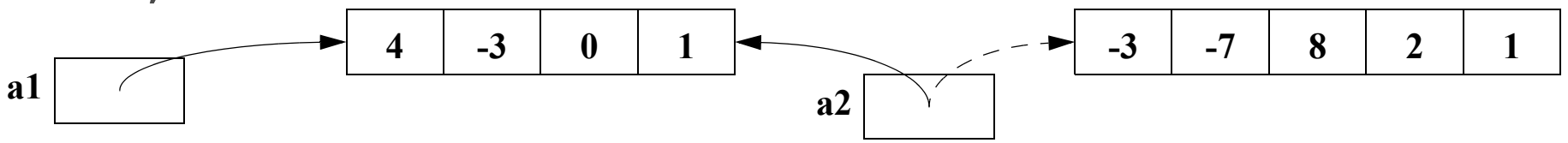


# Copia y comparación de arrays (y strings)

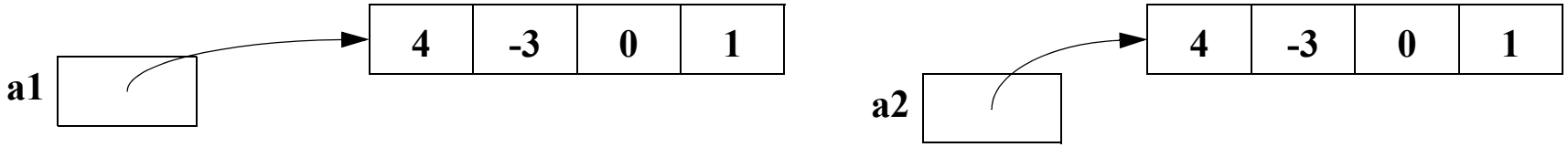
Los arrays y los strings son objetos, por consiguiente:

- La asignación sólo copia referencias

```
a2=a1;
```




- La comparación compara las referencias no los contenidos
  - en la figura anterior `a1==a2` es `true`
  - en la situación de debajo `a1==a2` es `false`

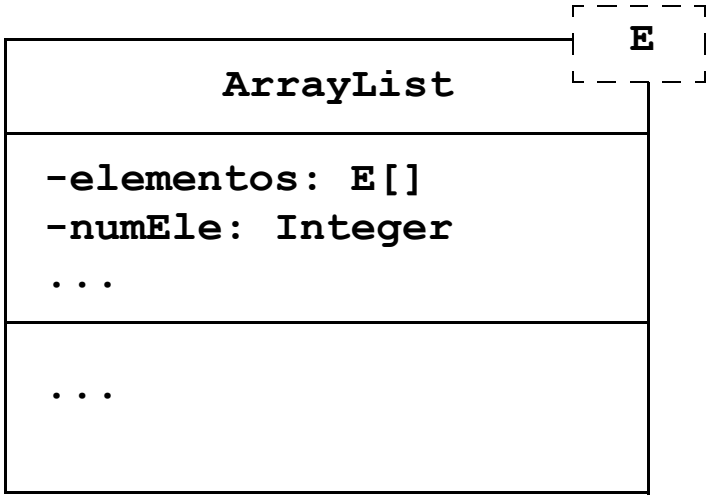



- Para comparar contenidos debe utilizarse el método `equals`

# Referencias y clase `ArrayList`

Un `ArrayList` almacena ***referencias*** a los objetos que "contiene"

- puesto que se implementa utilizando un array

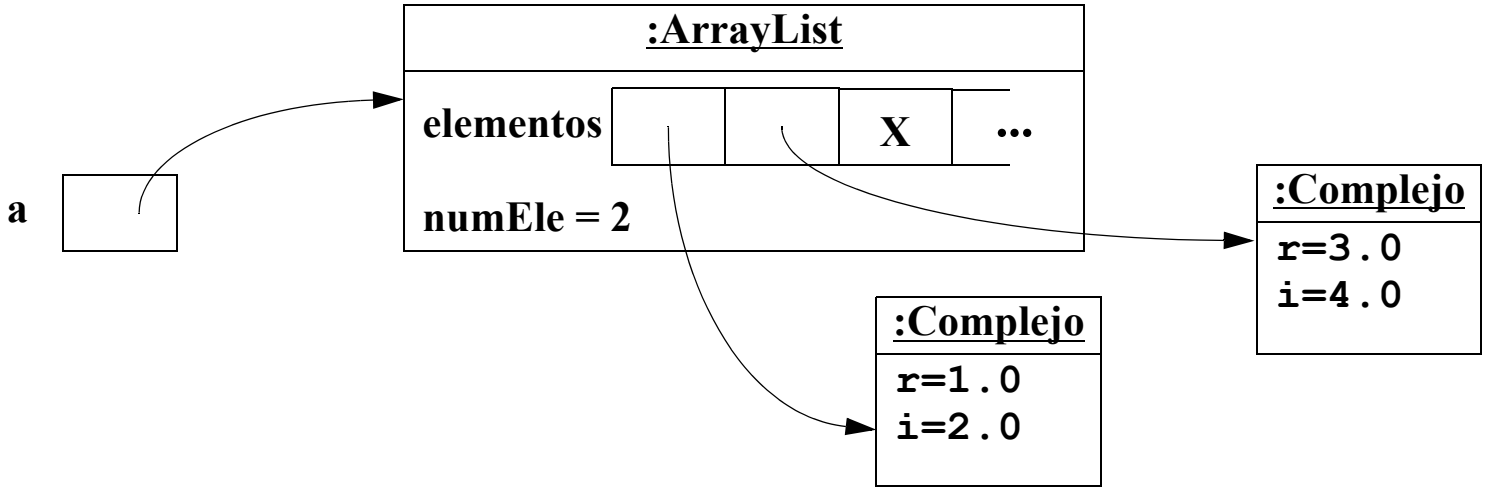



El método `equals` permite comparar dos `ArrayList`

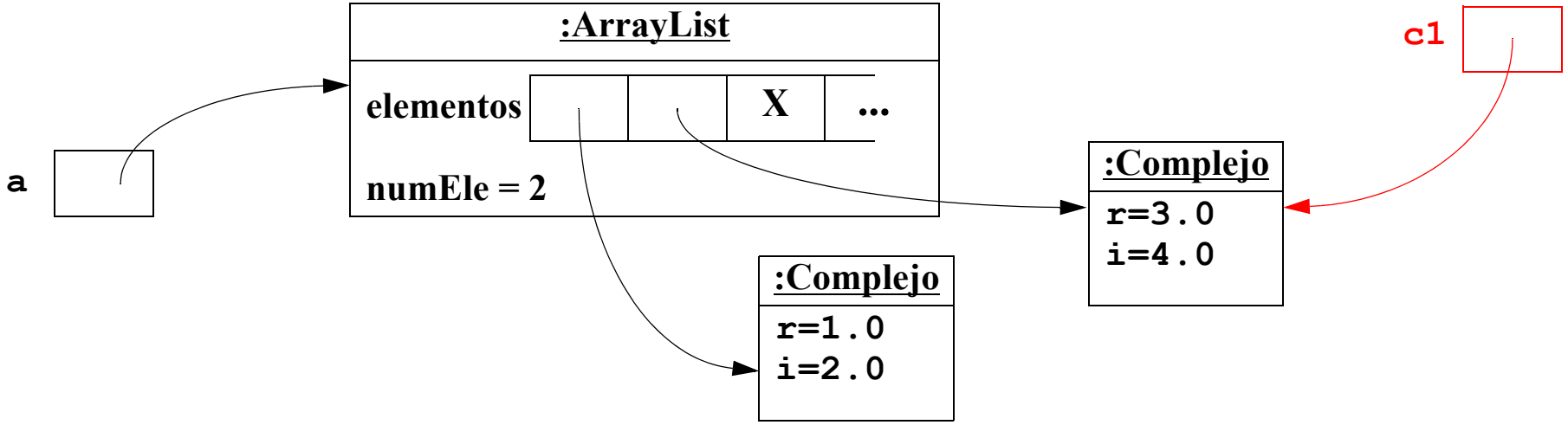
- Dos `ArrayList` son iguales si contienen los mismos elementos en el mismo orden

# Ejemplo ArrayList

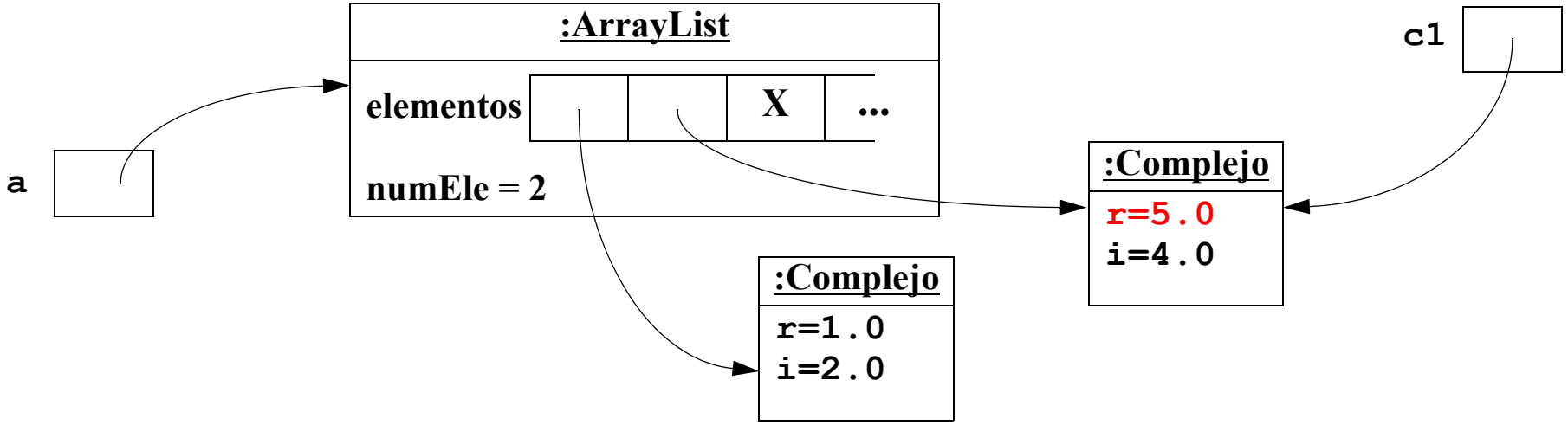
```
ArrayList<Complejo> a = new ArrayList<Complejo>();

a.add(new Complejo(1, 2));
a.add(new Complejo(3, 4));
```



```
Complejo c1 = a.get(1);
```




```
c1.r = 5;
```



# 6.3 Comparación de objetos

La comparación (`ref1==ref2`) compara referencias

- si queremos ***comparar los contenidos*** de los objetos debemos utilizar el método `equals`

Ejemplo: método `equals` para la clase `Coordenada`:

```
public class Coordenada {
    private int x; // coordenada en el eje x
    private int y; // coordenada en el eje y
…

    public boolean equals(Object obj) {
       Coordenada c = (Coordenada) obj;
       return c.x == x && c.y == y;
    }
}
```

**Veremos una definición más completa del método equals cuando veamos la herencia**

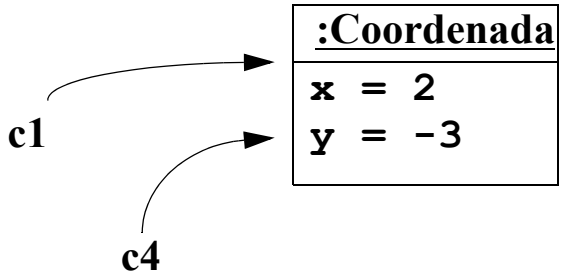
# Ejemplo de uso del método **equals**

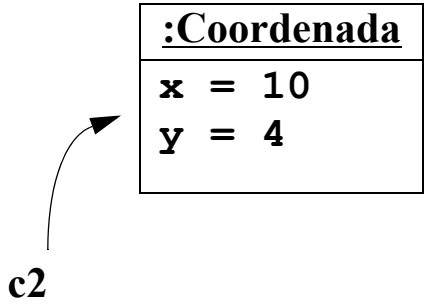
```
if (c1.equals(c1)) {
…
}

if (c1.equals(c4)) {
…
}

if (c1.equals(c2)) {
…
}

if (c1.equals(c3)) {
…
}
```

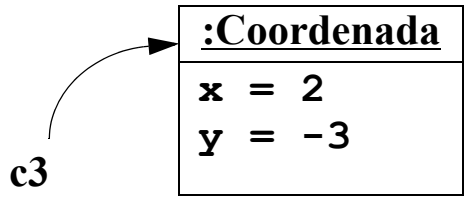
true

false

true



# Comparación de contenidos de arrays y strings

La clase `String` dispone del método `equals`:

```
String str1 = "hola", str2 = "hola";
if (str1.equals(str2)) {
   System.out.println("Los strings son iguales");
}
```

Para comparar arrays existe el método `equals` de la clase `Arrays`:

```
import java.util.Arrays;

int[] a1= {4, -3, 0, 1}, a2= {4, -3, 0, 1};
if (Arrays.equals(a1, a2)) {
   System.out.println("'a1' es igual a 'a2'");
}
```

# 6.4 Recolector de basura

La memoria ocupada por un objeto se reserva con el operador new

- en Java no existe ningún operador para liberar la memoria de un objeto que no se va a usar más

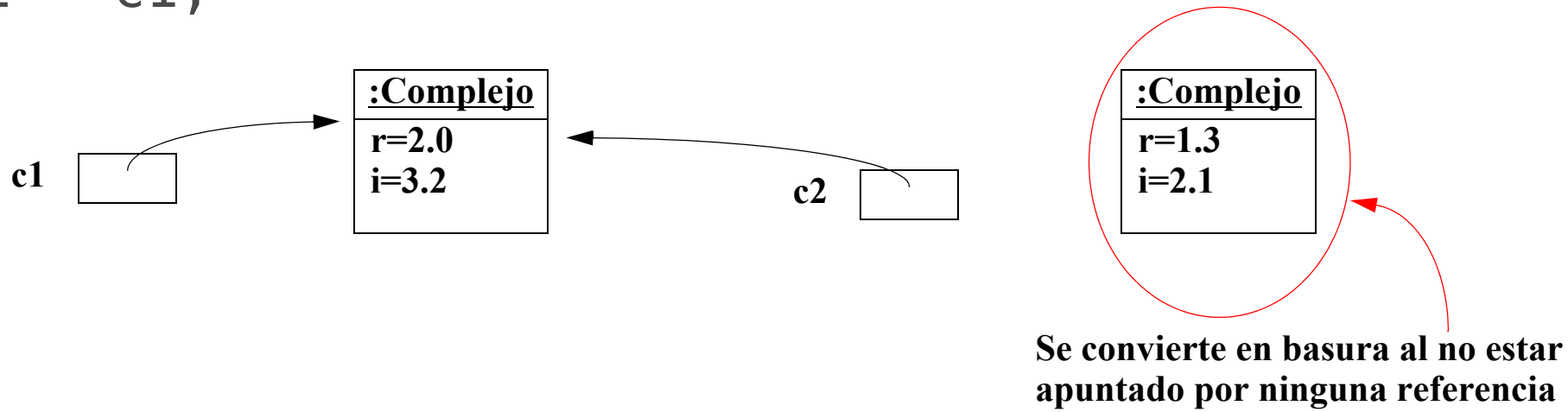
El ***Recolector de Basura*** realiza la ***liberación de memoria de forma automática***

- el recolector va contando el número de referencias que "apuntan" a cada objeto
- cuando un objeto no está apuntado por ninguna referencia, se convierte en "basura" (no va a poder ser utilizado nunca más)
- en los periodos de inactividad del sistema, el recolector reincorpora la "basura" a la memoria disponible

# Ejemplo de generación de basura

```
Complejo c1 = new Complejo(2.0, 3.2);
Complejo c2 = new Complejo(1.3, 2.1);
```


```
c2 = c1;
```

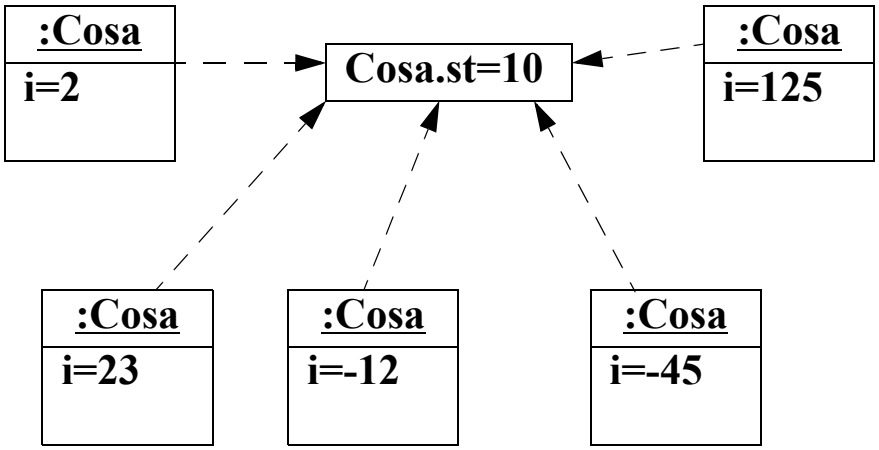


# 6.5 Métodos y campos de clase: estáticos

Una clase puede tener atributos y métodos estáticos:

- no pertenecen a los objetos de la clase, sino a la clase misma
- si son atributos, sólo existe uno por clase, no uno por objeto
  - la vida es la de la clase, y no la del objeto
- si son métodos, sólo pueden acceder a los objetos pasados como parámetros y a campos y métodos estáticos
- nos referimos a ellos como `NombreClase.miembroEstático`

```
public class Cosa {
   private static int st;
   private int i;
   …
}
```



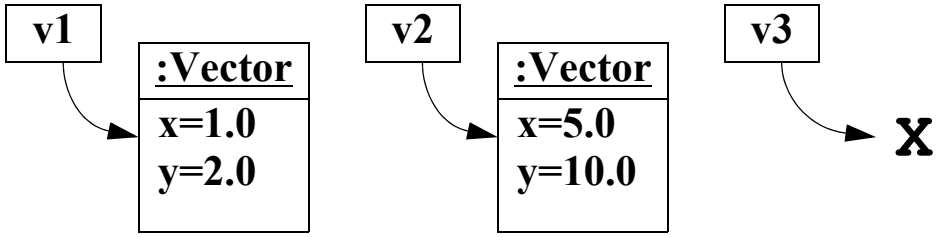
# Ejemplo de método de clase

```
public class Vector2D {


   // atributos
   private double x;
   private double y;


   /** Constructor */
   public Vector2D(double x, double y) {
      this.x = x;
      this.y = y;
   }
```

| Vector2D |
| --- |
| -x: double<br>-y: double |
| +Vector2D(x:double, y: double)<br>+suma(v: Vector2D): void<br>+<u>suma</u>(v1: Vector2D,<br>v2: Vector2D): Vector2D |

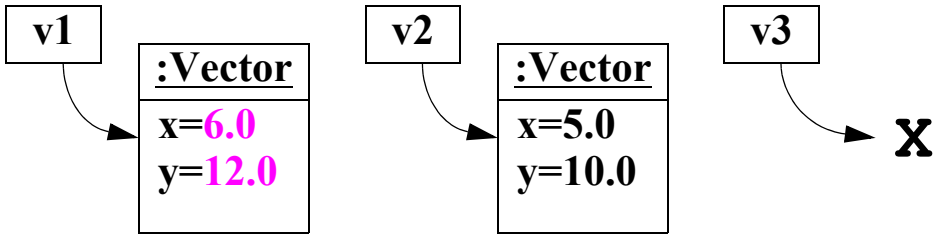
```
  /**
   * Método NO estático: Suma el vector v con
   * el vector actual
   */
  public void suma(Vector2D v){
    x = x + v.x;
    y = y + v.y;
  }

  /**
   * Método ESTÁTICO: Retorna el vector
   * resultado de v1+v2
   */
  public static Vector2D suma(Vector2D v1,
                              Vector2D v2) {
    return new Vector2D(v1.x+v2.x, v1.y+v2.y);
  }
}
```

```
Vector2D v1 = new Vector2D(1,2);
Vector2D v2 = new Vector2D(5,10);
Vector2D v3;
```



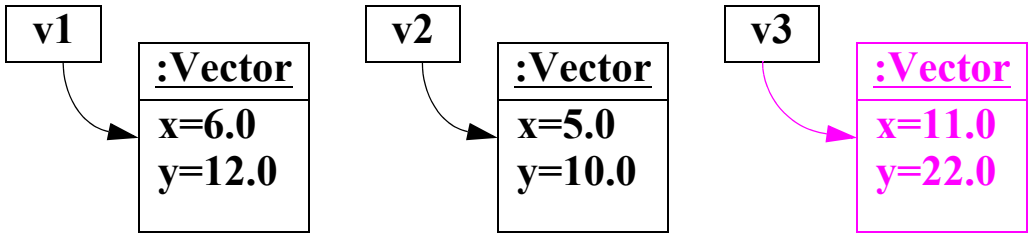

```
// suma no estática
v1.suma(v2);
```

refObjeto.nombreMétodo




```
// suma estática
v3=Vector2D.suma(v1,v2);
```

NombreClase.nombreMétodo



# 6.6 Anidamiento de clases

Una clase puede definirse dentro de otra

- las clases anidadas no estáticas se denominan clases internas y están asociadas con una instancia (objeto) de la clase
  - normalmente no las usaremos en esta asignatura
- las ***clases anidadas estáticas*** son como cualquier otra clase
  - salvo que su nombre es `ClaseContenedora.ClaseAnidada`
  - las usaremos principalmente con excepciones y enumerados

```
public class ClaseContenedora {
    public static class ClaseAnidada {
        …
    }
    …
}
```

# Ejemplo de clase anidada estática

```
public class Segmento {

   public static class Punto {
      double x, y;
   }

   // atributos
   private Punto pIni, pFin;

   /** constructor */
   public Segmento (Punto pIni, Punto pFin) {
      this.pIni = pIni;
      this.pFin = pFin;
   }


   …
}
```

```
public class PruebaSegmento {

   public static void main() {

      Segmento.Punto ptoIni = new Segmento.Punto();
      Segmento.Punto ptoFin = new Segmento.Punto();

      ptoIni.x=10.2;
      ptoIni.y=4.3;

      ptoFin.x=1.3;
      ptoFin.y=5.6;

      Segmento seg = new Segmento(ptoIni, ptoFin);
   }

}
```