Programación
Tema 3: Tipos y expresiones simples

Contenidos

- 1. Tipos primitivos: números enteros, números reales, booleanos, caracteres
- 2. Expresiones simples
- 3. Arrays
- 4. Campos (atributos)

Algoritmos y estructuras de datos

- Los lenguajes de programación permiten representar ***datos***
  - representación de la información
  - cómo los llamamos
  - cómo son (estructura)
  - qué se puede hacer con ellos
- Los lenguajes de programación permiten definir los ***algoritmos***
  - qué le hacemos a los datos
  - en qué orden (cuándo se lo hacemos)
  - cuántas veces

Tipos de datos primitivos

- Las clases nos permiten crear nuevos tipos de datos
- Podemos crear nuevas clases a partir de otras clases (composición-agregación)
  - Reutilizamos la biblioteca
- Pero … *¿Cuáles son los datos simples a partir de los que creamos las clases más sencillas?*
  - Los tipos primitivos de datos son tipos básicos con operaciones asociadas
  - Datos con tipo primitivo no son objetos

Tipos y valores primitivos

- Cada tipo primitivo tiene asociado un rango de valores: enteros, reales, {true, false}, caracteres
  - Y un conjunto de operadores (+,-,&,|, …)
- Podemos representar valores mediante literales (1, ‘a’, 2.0, true), variables o expresiones
- Las variables permiten almacenar información
  - Tienen asociado un tipo (primitivo o de clase)
  - Variables de campo (atributo), variables locales y parámetros
  - Podemos cambiar su valor tantas veces queramos
  - Solo podemos acceder (modificar o consultar) a ellas en su ámbito

Tipos primitivos en Java

- Java incluye tipos primitivos para representar
  - Caracteres
  - Enteros (valores enteros con signo)
  - Reales (racionales e irracionales con parte decimal)
  - Lógicos (verdadero, falso)
- Hay varios tipos primitivos para representar enteros y reales. *Porqué varios tipos para el mismo tipo de número?*
  - ocupan más o menos memoria
  - permiten representar mayor o menor rango de valores

Enteros

- Hay cuatro tipos primitivos para representar enteros: **int**, **long**, **short** y **byte**
- Los literales son los mismos para todos ellos: … -3, -2, -1, 0, 1, 2, 3 …

Entero	Bits	Mínimo	Máximo
int	32	`Integer.MIN_VALUE` -2^{31} (- 2147483658)	`Integer.MAX_VALUE` $2^{31}-1$ (2147483657)
short	16	`Short.MIN_VALUE` -2^{15} (-32768)	`Short.MAX_VALUE` $2^{15}-1$ (32767)
long	64	`Long.MIN_VALUE` -2^{63} (-9223372036854775808)	`Long.MAX_VALUE` $2^{63}-1$ (9223372036854775807)
byte	8	`Byte.MIN_VALUE` -2^{7} (-128)	`Byte.MAX_VALUE` $2^{7}-1$ (127)

Literales Enteros

- Literales de números enteros:
  - decimales: 53, 2300, -7, 0
  - hexadecimales: 0x35
  - octales: 065
  - Un literal terminado en L es un entero **long**: 7L (7 en 64 bits)
- no son enteros válidos:
  - 12,34
  - 45.67
  - 13E5

Operadores de Enteros

- Los operadores de los enteros son:
  - Unarios aritméticos (preceden a la expresión):
    - - invierte signo del entero
  - Binarios aritméticos (operan sobre sus operandos a izquierda y derecha):
    - +, - Suma y resta
    - * Multiplicación
    - / división que devuelve enteros 6 / 7 == 0, 15 / 4 == 3
    - % módulo (el resto cuando los números son positivos) x % y == x - (x / y) * y
  - ++, -- pre/post incremento/decremento
  - Binarios relacionales (comparan valores de las expresiones a izquierda y derecha):
    - >, >=, <, <= Menor, menor o igual, mayor, mayor o igual
    - ==, != igual, distinto

Reales

- Número reales en coma flotante (mantisa y exponente): **double**, **float**
- Número limitado de cifras significativas y de exponente

Real	**Bits**	**Mínimo** Menor Constante Representable	**Máximo** Mayor Constante Representable
float	32	`Float.MIN_VALUE`	`Float.MAX_VALUE`
double	64	`Double.MIN_VALUE`	`Double.MAX_VALUE`

Literales de Reales

- Entero, coma y decimales:
  - …-2.0, -1.0, 0.0, 1.0, 2.0, …
- Notación científica: mantisa y exponente:
  - 26.77e3
- Terminando en D o F decimos explícitamente si es *float* o *double*

Operadores de Reales

- Los operadores son los mismos que los enteros pero admiten/devuelven reales
  - Unarios aritméticos: -
  - Binarios aritméticos: +, -, *, /
  - Binarios relacionales: >, >=, <, <=, ==, !=
    - 0.0 == 0.0 es true, pero en general == y != nos puede dar resultados no esperados
    - Por ejemplo, prueba en scapbook que da false!:

```
float x = 1.1F; double y = 1.1D;
double z = 0.0D; boolean cero;
cero = (x - y) == z; // cero pasa a valer false
cero = 0.0 == 0.0; // cero pasa a valer true
```

Lógicos

- Tipo primitivo para valores lógicos: **boolean**
- Los literales: **true**, **false**
- Operadores sobre booleanos (son el resultado de operadores relacionales):
  - Unarios: ! negación
  - Binarios:
    - `x && y`: para que sea true, `x` e `y` deben ser true
      - Si `x` es false `y` no se evalúa
    - `x || y`: será true si cualquiera de las dos (`x` o `y`) es true
      - Si `x` es true `y` no se evalúa

Caracteres

- Tipo primitivo para valores carácter: **char**
  - Un único carácter en estándar Unicode 2.0 (16 bits)
  - ASCII
- Literales:
  - ‘a’ ‘j’ ‘\n’ ‘\t’ ‘\r’ ‘\\’
  - 0u0020
- Operadores:
  - Binarios relacionales: >, >=, <, <=, ==, !=
    - Los caracteres están ordenados según Unicode:
      - ‘0’, ‘1’, …, ‘9’, … ‘a’, ‘b’ … ‘z’, ‘A’, ‘B’, … ‘Z’
  - Muchas operaciones en clase (java.lang) **Character**

Caracteres

- **String** es una clase para representar cadenas de caracteres. Es una clase, no un primitivo
  - Pero tiene excepciones respecto otras clases
- Literales (una excepción; las clases no tienen literales)
  - "a", "Hola que tal“
  - Un literal puede ser tratado como una instancia de la clase **String.** (por ejemplo `"sss".compareTo("sss")`)
- Caracteres de escape
- Operadores: métodos de la clase **String** y **+**

Literales de Caracteres y Cadenas

- Caracteres:
  - ´a´ ´Ñ´ ´?´
- Cadenas de caracteres (strings):
  - ”Hola a todos” -> Hola a todos
  - ”esto es una cadena” -> esto es una cadena
  - ”dijo ´Adios´ y se fué” -> dijo ’Adios’ y se fué
  - ”qué \”raro\” es esto” -> qué ”raro” es esto

```
String miCadena = "hola";
```

Enumerados

- Algunas veces queremos un tipo de datos con un conjunto de valores limitado, concreto y del que describimos los literales
- Un enumerado (se declaran con "*enum*"), es un conjunto de literales que representan los posibles valores del tipo

```
enum Estaciones {INVIERNO,PRIMAVERA,VERANO,OTONO}
```

- Cuando queramos usar algunos de los literales de enumerado lo haremos:

```
Estaciones.INVIERNO
```

- **`enum`** define nuevos tipos de datos como lo hacemos con **`class`**

Tipos simples vs. Tipos compuestos

- **`int,short,long,byte,double,float, char,boolean`** son tipos primitivos
  - Las variables de esos tipo sólo almacenan un valor simple
  - Se representan con 8..64 bits
  - Se encuentran siempre contenidos dentro de un objeto, o están entre los parámetros y variables locales de un método activo
- ***String*** y ***Enumerados*** son objetos (compuestos) pero el lenguaje soporta algunos extras (p.e. literales, +)
- El resto de los tipos (compuestos) son clases:
  - Las variables de tipo compuesto referencian objetos
    - Para inicializarlas hay que usar **`new`**, o copiar otra referencia
  - Un objeto es una estructura compleja. Incluye bastante información (además de los datos): *referencia a la clase, identificador de objeto, monitor …*

Contenidos

- 1. Tipos primitivos: números enteros, números reales, booleanos, caracteres
- 2. Expresiones simples
- 3. Arrays
- 4. Campos (atributos)

Expresiones y precedencia operadores

- Una expresión permite realizar múltiples operaciones simples o complejas
  - (9.0/5.0) * *g* + 32.0
- Un operando puede ser:
  - Literal
  - Variable (campo, parámetro, variable local)
  - Resultado de una llamada a método que devuelve un resultado compatible con los operandos del operador
  - El resultado de una expresión
- Operadores de los tipos primitivos y de **String**

Expresiones y precedencia operadores

- Una expresión con varios operadores se ejecuta con el siguiente orden de precedencia:
  - Operadores unarios y llamadas a método, de izquierda a derecha
  - Operadores multiplicación y división de izquierda a derecha
  - Operadores de suma y resta de izquierda a derecha
  - Operadores de relación
  - Operadores de asignación
- Para fijar nuestro orden deseado (o para hacer expresiones más claras) usaremos paréntesis

```
-1 + 2 * (2 - 4 + 3) / 2
```

Conversiones

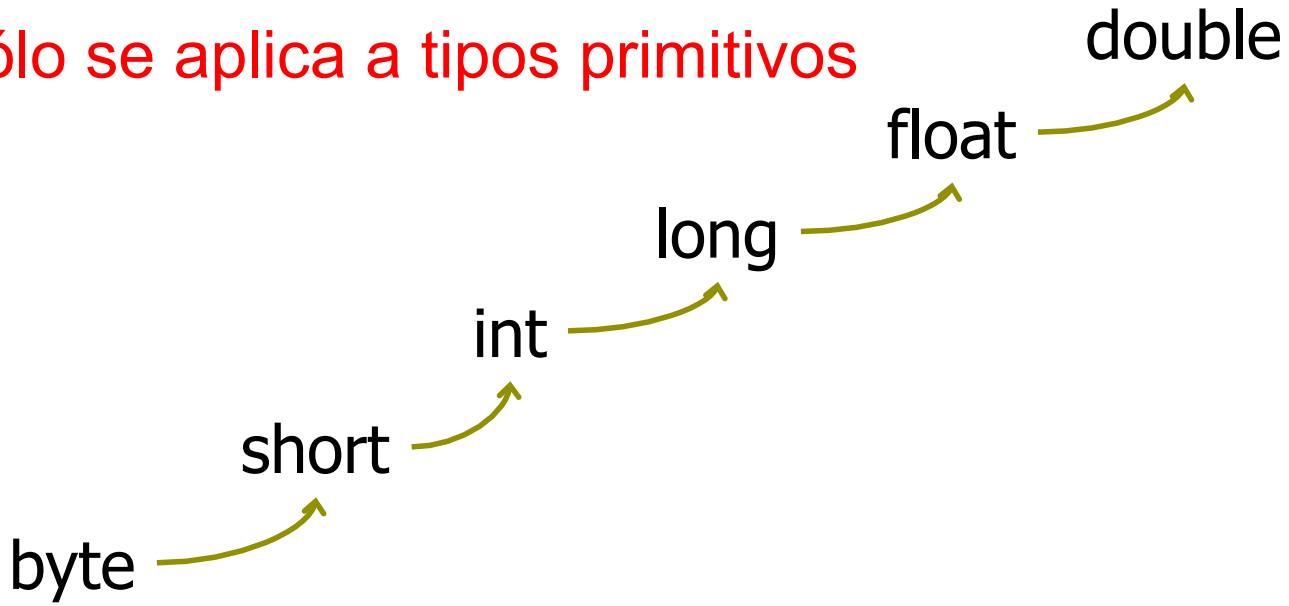
- Todos los operandos y parámetros tienen un tipo que debemos respetar
- Algunas veces queremos usar un valor que no es del tipo correcto (por ejemplo sumar enteros y reales)
- Conversión de tipos:
  - Implícita (definida en el lenguaje)
  - Explícita (casting o promoción), forzada por el programador

Conversiones implícitas

- 5.0 / 4 -> 1.25
- El 4 se convierte “automáticamente” en 4.0 para poder dividir
- La promoción tiene mayor prioridad
  - (float) 5 / 4 -> 1.25 (formato float)
  - El 5 se convierte por la promoción forzada a 5.0, el 4 se convierte “automáticamente”

Conversiones implícitas II

- Si se mezclan cantidades numéricas de diferentes tipos, se convierten automáticamente al tipo en el que seguro que se puede representar
- ¡Se puede perder información!
- Sólo se aplica a tipos primitivos



Conversiones implícitas III

- De tipos “pequeños” a “grandes”:
  - byte a short, int, long, float, o double
  - short a int, long, float o double
  - char a int, long, float, o double
  - int a long, float o double (¡OJO!)
  - long a float o double (¡OJO!)
  - float a double
- ¡Se puede perder información!
  - En el paso de long a float o double
  - En el paso de int a float

Conversiones explícitas

- Decimos de forma explícita cómo hacer la conversión:
  - **(nuevoTipo)** *expresion*
  - por ejemplo (char) 33 -> '!'
- Podemos perder información
- Podemos forzar operador que queremos usar:
  - ((double) 3) / 2
    - convierte explícitamente 3 a 3.0
    - usa un operador / de real
    - convierte automáticamente 2 a 2.0
- Hay muchas conversiones mediante métodos en `Integer,Byte,Short,Long,Double,Float,Char,String`

Convertir String en otros valores

- Leídos típicamente, desde línea de comandos

```
double base= Double.valueOf(args[0]).doubleValue();
int x= Integer.valueOf(args[1]).intValue();
```

- Otra forma:

```
int precio= Integer.parseInt(args[1]);
```

  - Si no se puede convertir, se lanzará una excepción: el programa termina abruptamente

Envoltorios

- `Integer, Long, Short, Byte, Float, Double, Boolean, Character` son clases de java.lang que representan primitivos con clases. Incluyen:
  - Operadores no predefinidos:
    compare(Double anotherDouble)
  - Operadores de conversión
    longValue()
  - Operadores para formatear impresiones
    toString()
  - Constantes típicas de los primitivos
    SIZE
- Java convierte automáticamente cuando hay que transformar de primitivo a objeto:
  public void m(Integer i) { … } m(7);

Contenidos

- 1. Tipos primitivos: números enteros, números reales, booleanos, caracteres
- 2. Expresiones simples
- 3. Arrays
- 4. Campos (atributos)

Vectores y arrays

- Vectores (álgebra)
  - un vector de N reales

    $v_0, v_1, v_2, ..., v_{n-1}$
  - acceso al término k-ésimo

    v_k
- Matrices (álgebra)
  - una matriz de MxN reales

    $v_{00}, v_{01}, ..., v_{0n-1}$

    $v_{10}, v_{11}, ..., v_{1n-1}$

    ...

    $v_{m-1,0}, v_{m-1,1}, ..., v_{m-1,n-1}$
  - acceso al término x, y

    v_{xy}
- arrays (java)
  - un array de N reales

    ```
    double[] v
        = new double[N];
    ```
  - acceso al término k-ésimo

    ```
    v[k]
    ```
- arrays (java)
  - una matriz de MxN reales

    ```
    double[][] v
        = new double[M][N]
    ```
  - acceso al término x, y

    ```
    v[x][y]
    ```

Arrays: Declaración

- Ejemplos de estructuras de datos representables con array:
  - Los puntos de una pantalla
  - El conjunto de números de teléfono de una agenda
- Los arrays nos permiten representar muchos valores primitivos u objetos de un mismo tipo
  - Nos referimos a todos ellos en conjunto, con el mismo nombre
  - Podemos referirnos a cada uno de ellos de forma individual

```
Tipo[] nombreVariable ;
Tipo nombreVariable[];
```

Arrays: Creación

- Los array son tipos de objetos, y antes de usarlos hay que crearlos:

  `nombreVariable = new Tipo[Expresion int];`

  - Al crear el array **fijamos** el número de elementos que queremos almacenar en él
  - Cada uno de los elementos del array se crea y se inicializa con su valor por defecto
    - Si el tipo base del array es un tipo primitivo, se inicializa con su valor por defecto (0 int, false boolean, \u0000 char, 0.0 reales)
    - Si el tipo base es una clase, todos los valores del array son null
  - Existen los literales de array: son un conjunto de expresiones del mismo tipo, separadas por comas y encerradas entre llaves

    ```
    int[] arrayDeInt = {2,4,6,8,10};
    int[][] matriz= {{ 1, 2, 3 }, { 4, 5, 6 }};
    ```

Arrays: Acceso

- El nombre de la variable de tipo array nos referencia todo el grupo en conjunto
- El array es un tipo de objeto (con algunas peculiaridades)
- Dos variables array pueden referenciar el mismo grupo

```
Tipo[] var1, var2;
...
var1=var2
```

  - var1 y var2 referencian el mismo conjunto (no son dos copias)

```
metodo(Tipo[] par) { ...}
...
obj.metodo(var1)
```

  - par nos hará referencia al conjunto que referencia var1
- El campo *length* de una variable array devuelve el número de elementos que tiene el array

Arrays: Acceso II

- Para referenciar un elemento en concreto debemos identificar su posición (índice), el primero es el 0

  `nombreVariable[Expresion int]`

  - Nos devuelve un valor del tipo base del array
  - Podemos usar esa posición como cualquier otra variable

  `nombreVariable[Expresion int]=Expresion del Tipo Base`

  - Por ejemplo:

```
variableArrayInt[i]=11;
variableArrayInt[i]=variableArrayInt[i]+(variableArrayInt[j]+1);
variableArrayObj[i].metodo();
```

- Cuando el índice que empleamos tiene un valor mayor o igual que `length`, o el número es menor de 0:
  - Se levanta una excepción: `ArrayIndexOutOfBoundException`

Arrays Multidimensionales

- En algunos casos nos interesa tener arrays con varios índices:
  - Los puntos de una pantalla los podemos representar mediante un array de dos dimensiones
- Por cada dimensión que queramos tener, incluimos un índice ([]), y en el constructor definimos el tamaño cada una de las dimensión

  ```
  Color pantalla[][]=new Color[1024][2048]
  ```

- Para acceder a un elemento base debemos incluir tantos índices como dimensiones tenga el array

  ```
  pantalla[i][j]
  ```

- Un array multidimensional es un array de arrays
- Con un solo índice (pantalla[i]) obtenemos referencia un array que representa todos los elementos de esa dimensión (una fila de la pantalla
  pantalla[i][0], pantalla[i][1], … pantalla[i][pantalla[i].length-1])

Arrays Multidimensionales II

- pantalla.length devuelve el número de elementos de la primera dimensión, pantalla[i].length devuelve el número de elementos de la dimensión i
- Podemos ir creando las diferentes dimensiones dinámicamente, pero una vez creadas, su tamaño no cambia

Contenidos

- 1. Tipos primitivos: números enteros, números reales, booleanos, caracteres
- 2. Expresiones simples
- 3. Arrays
- 4. Campos (atributos)

Campos

- Datos primitivos y referencias para los objetos de una clase
- Se declaran en cualquier punto dentro de una clase menos dentro de un método
- Sintaxis:

```
Tipo nombre;
Tipo nombre1,nombre2,... nombreN;
```

```
int i;
Punto origen, destino;
String nombre;
Persona propietario, conyugeDelPropietario;
```

Identificadores

- Cadena de letras Unicode, dígitos, $ y _ que no comienzan por dígito
  - Mayúsculas != Minúsculas
  - Número ilimitado de caracteres
  - No pueden incluir blancos
  - No se pueden usar palabras reservadas
- Un identificador determina el campo, parámetro, variable local, método, clase o paquete al que nos estamos refiriendo
- Ejemplos no validos:
  - día del año (blancos)
  - 1a
  - class
- Hacer esfuerzo por elegir nombres comprensibles

Campos: Inicialización

- Podemos fijar el valor inicial cuando hacemos la declaración
  - Tipo variable = expresión;
- Si no lo hacemos tomará el valor por defecto:
  - Enteros y reales 0 y 0.0
  - Booleanos false
  - Char 0u0000
  - referencias null
- El constructor es el lugar ideal para inicializar campos
- Debemos cuidar lo que escribimos en una expresión de inicialización. Los valores necesarios deben ser conocibles:

```
int i=metodoQueUsaj();
int j=20*i;
public int metodoQueUsaj() { return j;}
```

Campos: Constantes

- Algunas veces identificamos campos cuyo valor inicial es conocido y su valor será siempre el mismo. Ejemplos:
  - Valores que son siempre los mismos:
    - mayoría de edad
  - Valores iniciales por defecto:
    - tamaño por defecto de un array
  - Valores de uso típicos:
    - posición de origen
- Se declaran:
  - incluyen `final` antes del tipo
  - deben incluir inicialización
  - suelen ser `static` (lo veremos en el tema 6)
  - Convenio: usar mayúsculas
- Ejemplo: `final int MAYORIA_EDAD = 18;`

Ejemplo

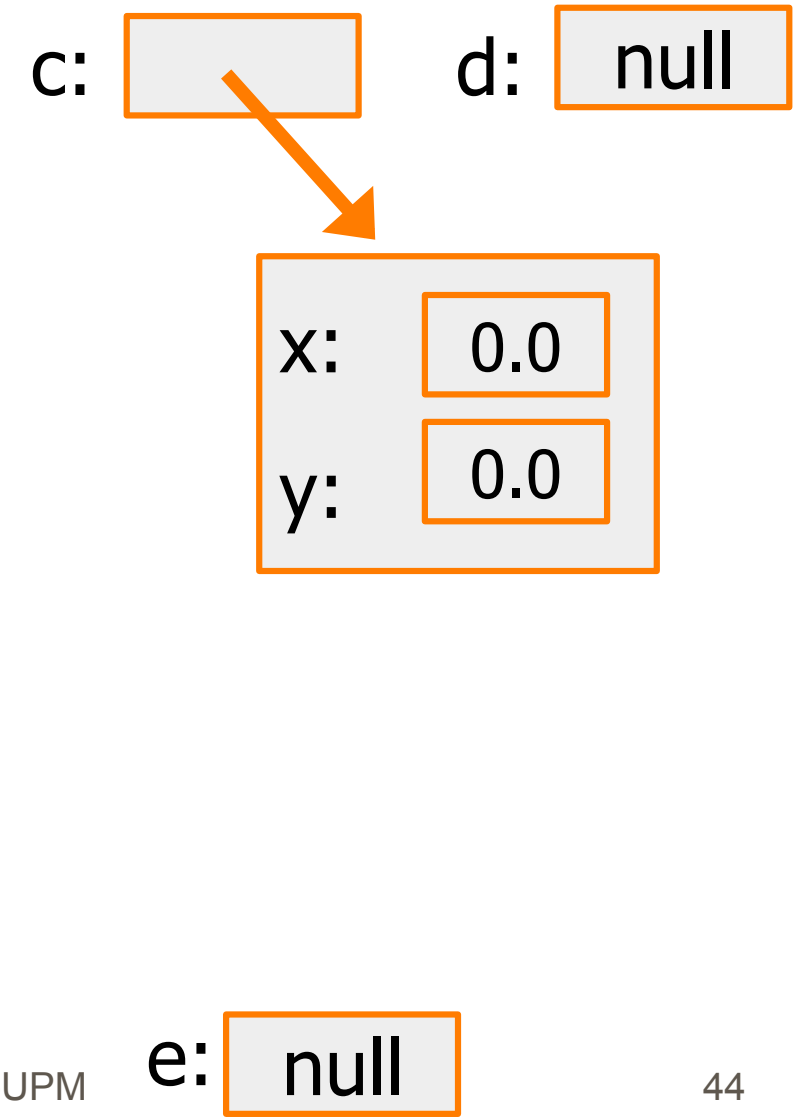
```
public class Punto{
        private double x;
        private double y;

        public Punto (double cx, double cy){
                x= cx;
                y= cy;
        }
}
```

Evolución en el tiempo

1. Punto c, d, e;

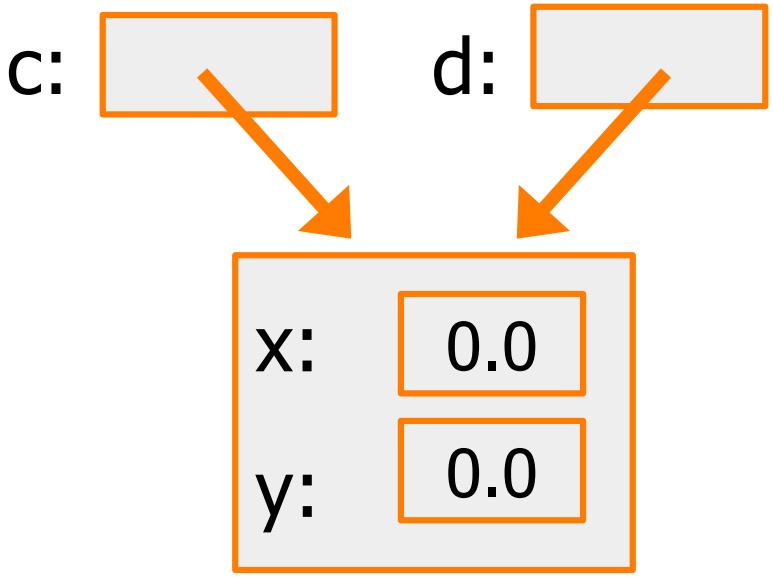
Evolución en el tiempo

1. Punto c, d, e;
2. c= new Punto(0.0,0.0);



Evolución en el tiempo

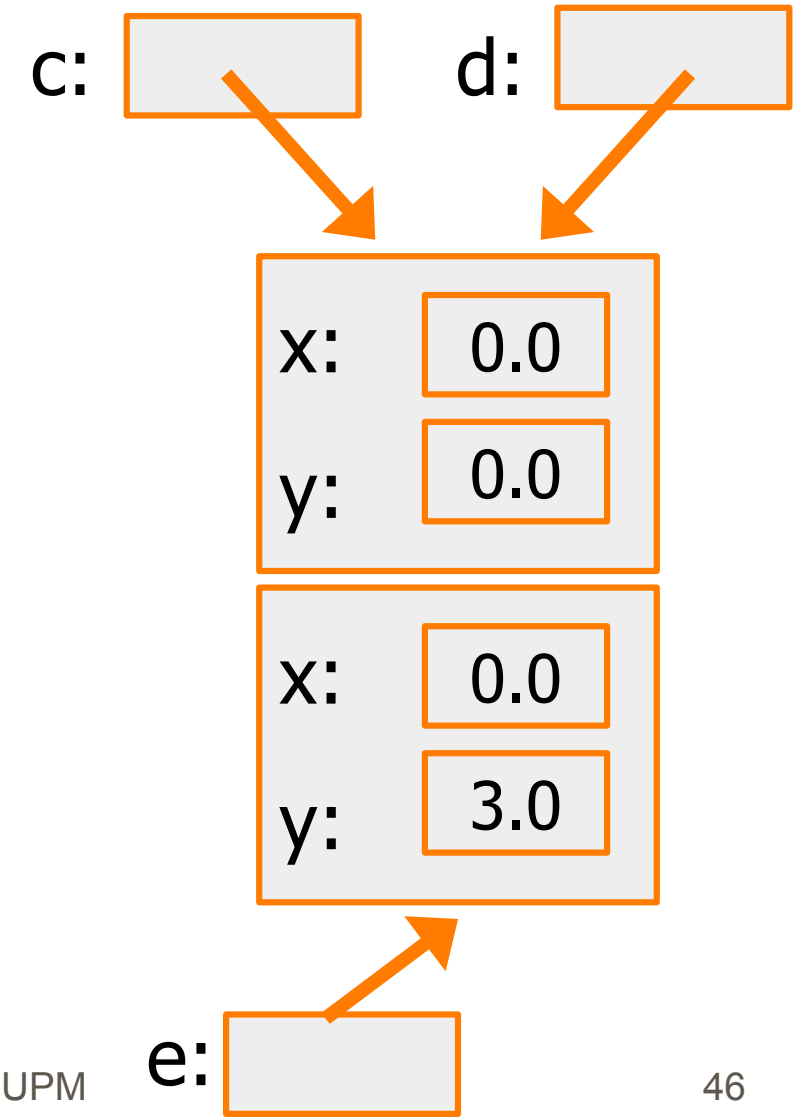
1. Punto c, d, e;
2. c= new Punto(0.0,0.0);
3. d=c;



e: null

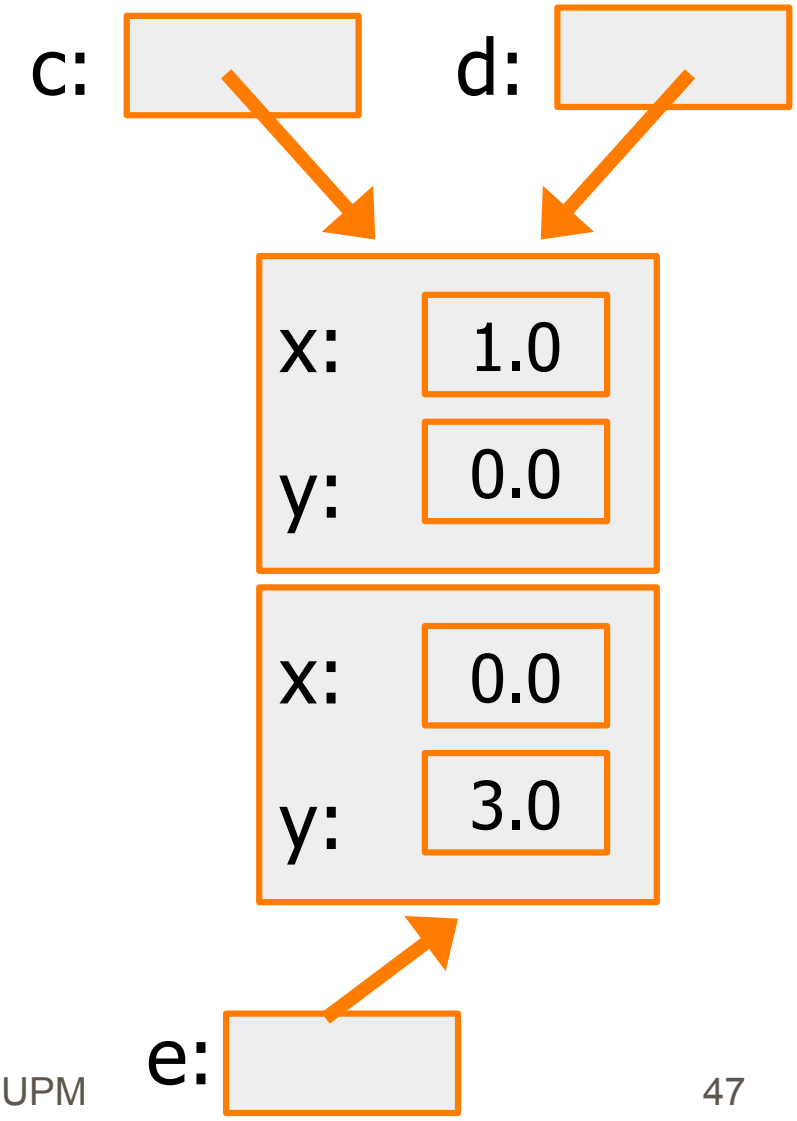
Evolución en el tiempo

1. Punto c, d, e;
2. c= new Punto(0.0,0.0);
3. d=c;
4. e= new Punto(0.0,3.0);



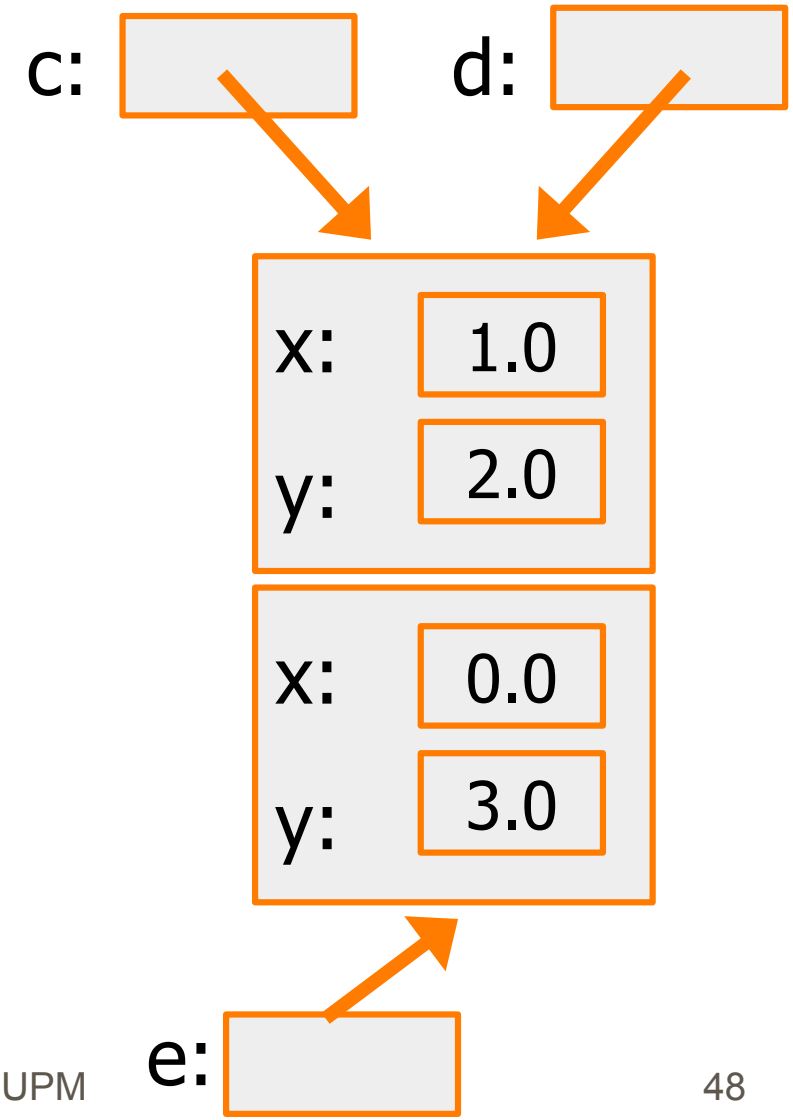
Evolución en el tiempo

1. Punto c, d, e;
2. c= new Punto(0.0,0.0);
3. d=c;
4. e= new Punto(0.0,3.0);
5. c.x=1.0;



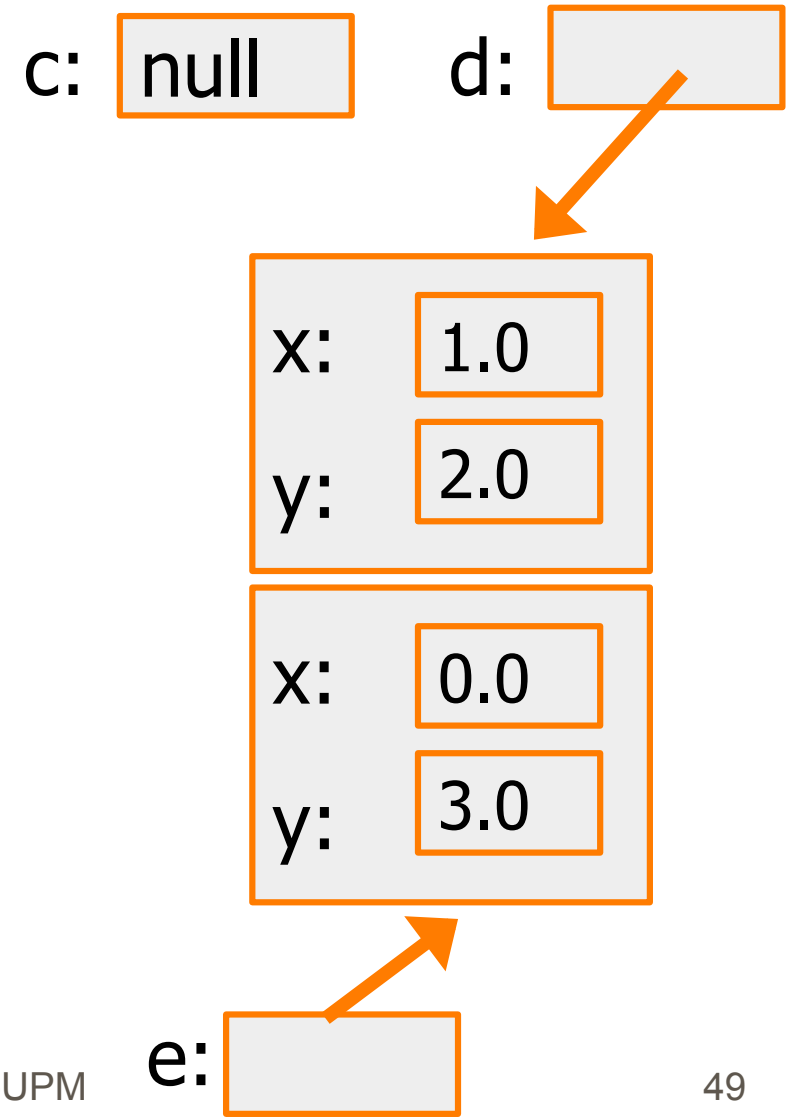
Evolución en el tiempo

1. Punto c, d, e;
2. c= new Punto(0.0,0.0);
3. d=c;
4. e= new Punto(0.0,3.0);
5. c.x=1.0;
6. d.y=2.0;



Evolución en el tiempo

1. Punto c, d, e;
2. c= new Punto(0.0,0.0);
3. d=c;
4. e= new Punto(0.0,3.0);
5. c.x=1.0;
6. d.y=2.0;
7. c= null;



Evolución en el tiempo

1. Punto c, d, e;
2. c= new Punto(0.0,0.0);
3. d=c;
4. e= new Punto(3.0,0.0);
5. c.x=1.0;
6. d.y=2.0;
7. c= null;
8. d= null;

- La memoria RAM
- se puede reciclar
- *Garbage Collection*