

2. Programación Orientada a Objetos (POO)

2.4. Agrupación de objetos

Programación Modular y Orientada a Objetos

Felipe Ibañez y Juan Miguel Lopez

felipe.anfurrutia@ehu.es

juanmiguel.lopez@ehu.es

Dpto. de Lenguajes y Sistemas Informáticos

UPV/EHU

Contenido

- ❑ Colecciones
 - Tamaño fijo: `Array`
 - Tamaño indefinido: `ArrayList`
 - Iteración
- ❑ La importancia de la ocultación de la información
- ❑ Igualdad vs Identidad

La necesidad de agrupar objetos

- ❑ Muchas aplicaciones requieren agrupar objetos
 - Agendas
 - Catálogos de librerías o de productos
 - Sistema de matrícula de la Universidad
 - ...
- ❑ El número de elementos a agrupar varía
 - Adición de elementos
 - Eliminación de elementos
- ❑ Y hay que ser capaces de manipular los elementos de la colección
 - Búsqueda de elementos
 - Consulta y modificación de elementos

Colecciones de tamaño fijo: Arrays

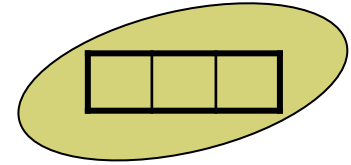
- A veces el tamaño máximo de una colección puede estar predeterminado
 - Para ello los lenguajes de programación suelen ofrecer un tipo de colección de tamaño fijo:

array

- Los **arrays** Java pueden guardar referencias a objetos o valores de tipos primitivos
 - Todas del mismo tipo
 - Aunque son objetos, su **declaración**, **creación** e **inicialización** es diferente que para otros objetos
 - El operador **indexación []** permite el acceso a cada elemento del array
 - El atributo **length** nos permite saber el nº de elementos que contiene el array

Arrays (Matrices) :

Declaración, Creación, Inicialización



- ❑ **Declaración**: Consiste en asignar un **identificador** al array y decir de qué **tipo** són los elementos que va a almacenar.

- Se puede hacer de 2 formas

```
tipo[] nombreArray;  
tipo nombreArray[];
```

- Después de la declaración aún no se ha asignado memoria para almacenar el array no podemos acceder a su contenido

- ❑ **Creación**: Consiste en reservar espacio en memoria para el array

- Es necesario utilizar **new** y especificar

tamaño del array `nombreArray = new tipo[numPosiciones];`

- Una vez creado el array sus elementos tienen los valores por defecto hasta que el array sea inicializado

Valores por defecto:

int, short, long = 0

float, double = 0.0

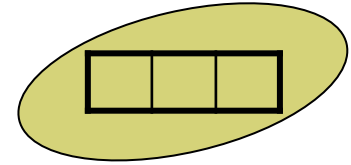
booleanos = false

String = null

MiClase = null

Arrays (Matrices) :

Declaración, Creación, Inicialización



□ **Inicialización**: Consiste en **dar valores** a los distintos elementos del array. Podemos hacerlo de varias formas:

- Elemento a elemento

```
nombreArray[0] = elemento0;  
nombreArray[1] = elemento1;  
...
```

- Mediante un bucle

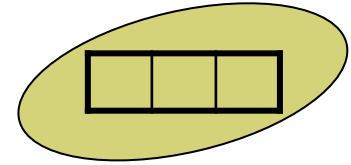
```
for(int i = 0; i < nombreArray.length; i++){  
    nombreArray[i] = elemento-i;  
}
```

- Por asignación directa

```
nombreArray = {elem1, elem2, elem3, ...};
```

Ejemplos: Arrays (Matrices)

Declaración, Creación, Inicialización



Arrays de tipos básicos

```
int a[];           //Declara
a = new int[3]     //Crea
a[0]=1;           //Inicializa
a[1]=2;
a[2]=3;
```

```
int a[] = new int[3] //Declara y Crea
a[0]=1;              //Inicialización
a[1]=2;
a[2]=3;
```

```
int a[] = new int[3] //Declara y crea
for(int i=0; i<a.length;i++){ //inicializa
    a[i]=i+1;
}
```

```
int a[] = {1, 2, 3}; //Declaración, creación e inicialización
```

Arrays de objetos (Tipos de referencia)

```
MiClase a[];      //Declara
a = new MiClase[3] //Crea
a[0]=new MiClase(param1);
a[1]=new MiClase(param2);
a[2]=new MiClase(param3);
```

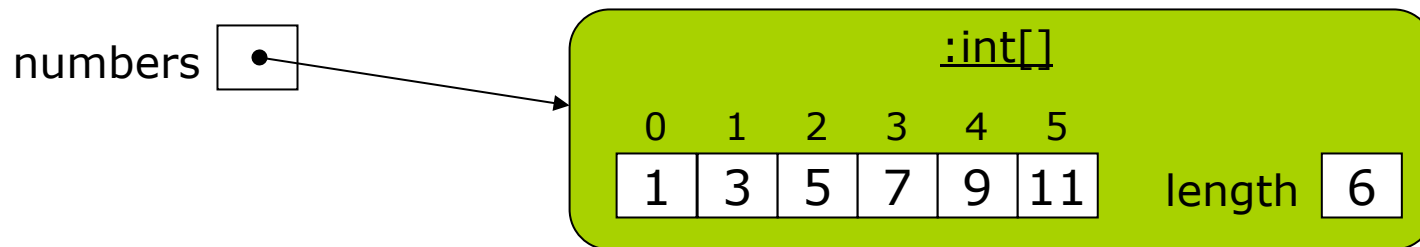
```
MiClase a[] = new MiClase[3]
//inicializa
a[0]=new MiClase(param1);
a[1]=new MiClase(param2);
a[2]=new MiClase(param3);
```

```
MiClase a[] = new MiClase[3]
//inicializa
for(int i=0; i<a.length;i++){
    a[i]=new MiClase(param-i);
}
```

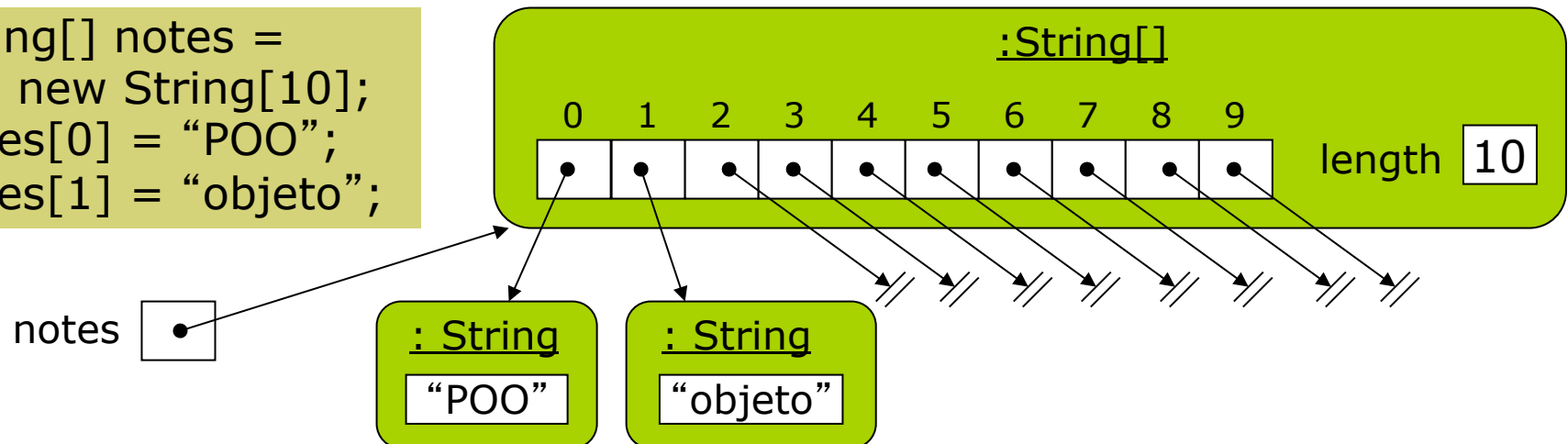
```
MiClase a[] = {new MiClase(param1), new MiClase(param2), new MiClase(param3)};
```

Arrays: tipo primitivo vs tipo referencia

```
int[] numbers = new int[6];  
for(int i = 0; i < numbers.length; i++)  
    numbers[i] = 2*i + 1;
```

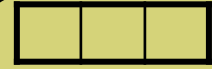


```
String[] notes =  
    new String[10];  
notes[0] = "POO";  
notes[1] = "objeto";
```



Arrays (Matrices): Errores frecuentes

Declaración, creación, inicialización



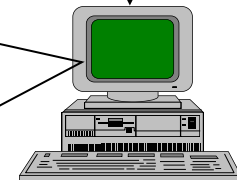
```
public class EjemplosMatrices{  
    public static void main(String args[]){  
        double miMatriz[];  
        System.out.println(miMatriz[0]);  
    }  
}
```

MAL

compilar

Falla la
compilación

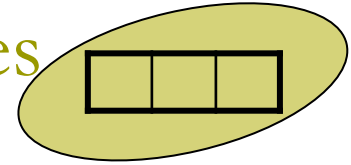
variable miMatriz may not have been initialized



Cuando la matriz sólo ha sido *declarada* no podemos acceder a sus elementos. El programa no compilaría y daría un *error*

Arrays (Matrices): Errores frecuentes

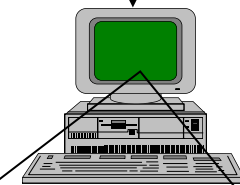
Declaración, creación, inicialización



```
public class EjemplosMatrices2{
    public static void main(String args[]){
        int miMatrizDeEnteros[] = new int[10];
        float miMatrizDeReales[] = new float[10];
        boolean miMatrizDeBooleanos[] = new boolean[10];
        char miMatrizDeCaracteres[] = new char[10];
        String miMatrizDeStrings[] = new String[10];
        MiClase miMatrizDeObjetos[] = new MiClase[10];
        System.out.println("Entero por defecto: " + miMatrizDeEnteros[0]);
        System.out.println("Real por defecto: " + miMatrizDeReales[0]);
        System.out.println("Booleano por defecto: " + miMatrizDeBooleanos[0]);
        System.out.println("Carácter por defecto: " + miMatrizDeCaracteres[0]);
        System.out.println("String por defecto: " + miMatrizDeStrings[0]);
        System.out.println("MiClase por defecto: " + miMatrizDeObjetos[0]);
    }
}
```

compilar

Ejecutar



Cuando la matriz sólo ha sido *declarada y creada* pero *no inicializada* podemos acceder a sus elementos pero estos tienen su *valor por defecto*

Entero por defecto: 0
Real por defecto: 0.0
Booleano por defecto: false
Carácter por defecto:
String por defecto: null
MiClase por defecto: null

Iteración

- ❑ Habitualmente queremos repetir varias veces un conjunto de acciones
 - Por ejemplo, imprimir las notas de la agenda una a una
- ❑ La mayoría de los lenguajes de programación incluyen sentencias de bucle para hacer esto:
 - `while`, `for`, `repeat`, ...
- ❑ Los bucles permiten controlar cuántas veces se repite un conjunto de acciones
 - En el caso de las colecciones es habitual repetir acciones para cada objeto de la colección particular

Iteración: el bucle for

- ❑ Hay dos variaciones del bucle for: *for* y *for-each*.
- ❑ El bucle *for* se utiliza normalmente para iterar un número fijo de veces
 - Generalmente se utiliza con una variable que se incrementa (o decrementa) una cantidad fija en cada iteración
 - Es parecido al *while*
- ❑ El bucle *for-each* repite el cuerpo del bucle para cada objeto de una colección
 - Más fácil de escribir
 - Más seguro: hay garantía de que acabará

Bucle for vs. while vs. for-each: ejemplo printNotes()

```
String[ ] notes;
```

□ for

```
for(int i = 0; i < notes.length; i++)  
    System.out.println(notes[i] + ",");
```

□ while

```
int i = 0;  
while(i < notes.length){  
    System.out.println(notes[i] + ",");  
    i++;  
}
```

□ for-each

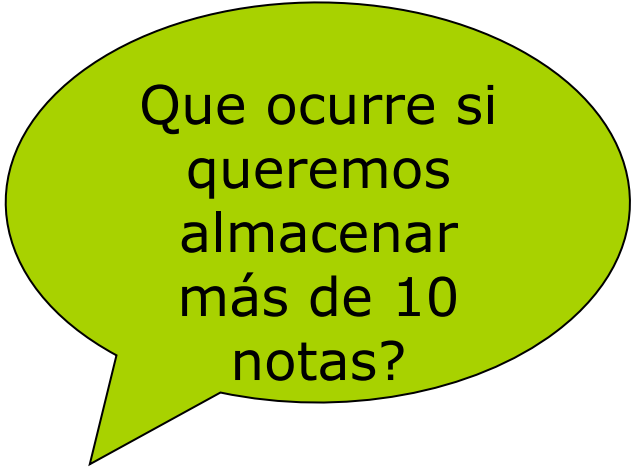
```
for(String note : notes)  
    System.out.println(note + ",");
```

Mientras el valor del índice i es menor que el tamaño, imprimir la siguiente nota, e incrementar el índice

Para cada nota en la colección notas, imprimir la nota

Ejemplo: NoteBook.java

```
public class NoteBook{
    private static final int SIZE_MAX = 10;
    private String[] notes;
    private int numberOfNotes;
    public NoteBook(){
        notas = new String[SIZE_MAX];
        numberOfNotes = 0;
    }
    public void storeNote (String note){
        if (numberOfNotes < SIZE_MAX){
            notes[numberOfNotes] = note;
            numberOfNotes++;
        }
        else System.out.println("Can`t store it");
    }
    public int getNumberOfNotes(){
        return numberOfNotes;
    }
    ...
}
```



Que ocurre si
queremos
almacenar
más de 10
notas?

Características de la colección indefinida

- ❑ Incrementa su capacidad como sea necesario
- ❑ Se puede saber el tamaño con el método `size()`
- ❑ Guarda los objetos en orden
- ❑ Los detalles de cómo ocurre todo esto están ocultos
 - ¿Realmente importan?
 - ¿El hecho de no conocerlos nos impide usar la colección?

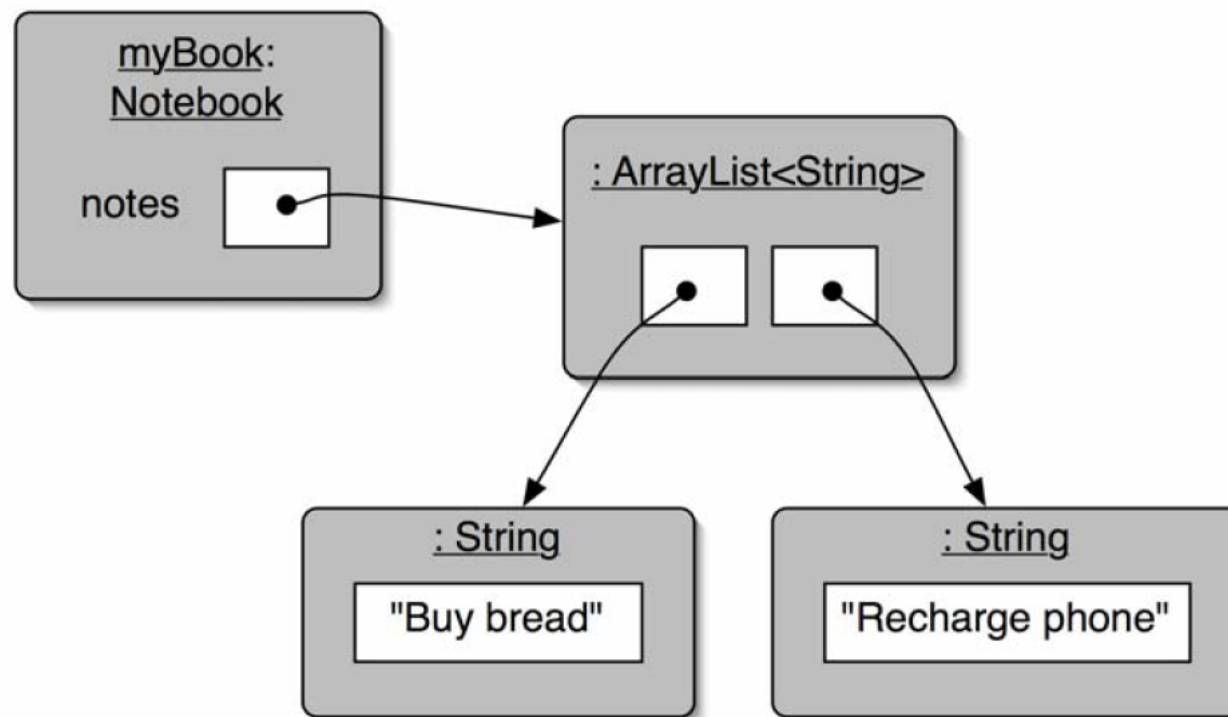
Librerías de clases

- ❑ No vamos a implementar las estructuras de datos para guardar las notas: usaremos librerías
 - Así no hay que hacerlo todo desde cero
- ❑ Una librería de clases es...
una colección de clases útiles
 - En Java una librería de clases es un *package*
- ❑ Como la agrupación de objetos es algo habitual, en Java hay una librería de clases para esto:
 - package **java.util**

Colecciones

- Se especifica
 - El tipo de la colección: `ArrayList`
 - El tipo de objetos que contendrá: `<String>`
- Esto se lee como: “*ArrayList de String*”

Estructuras de los objetos con colecciones



Ejemplo: Notebook.java

```
import java.util.ArrayList;

public class Notebook {
    // Storage for an arbitrary number of notes.
    private ArrayList<String> notes;

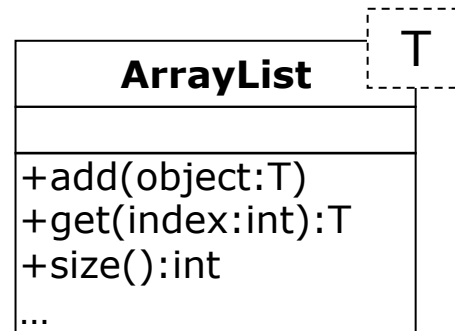
    /**
     * Perform any initialization required for the
     * notebook.
     */
    public Notebook(){
        notes = new ArrayList<String>();
    }
    ...
}
```

Indica que se va a utilizar la clase ArrayList del paquete java.util

Constructor de la clase genérica ArrayList

Clases genéricas

- Las colecciones están implementadas como tipos *genéricos* o *parametrizados*



T indica un tipo de dato, que es obligatorio definir para su uso

- El tipo de dato del parámetro indica de qué queremos que sea la lista:
 - **ArrayList<Person>**
 - **ArrayList<TicketMachine>**
 - etc.

Uso de la colección (delegación)

```
import java.util.ArrayList;

public class Notebook {
    // Storage for an arbitrary number of notes.
    private ArrayList<String> notes;
    ...
    public void storeNote(String note) {
        notes.add(note);
    }

    public int getNumberOfNotes () {
        return notes.size();
    }
    ...
}
```

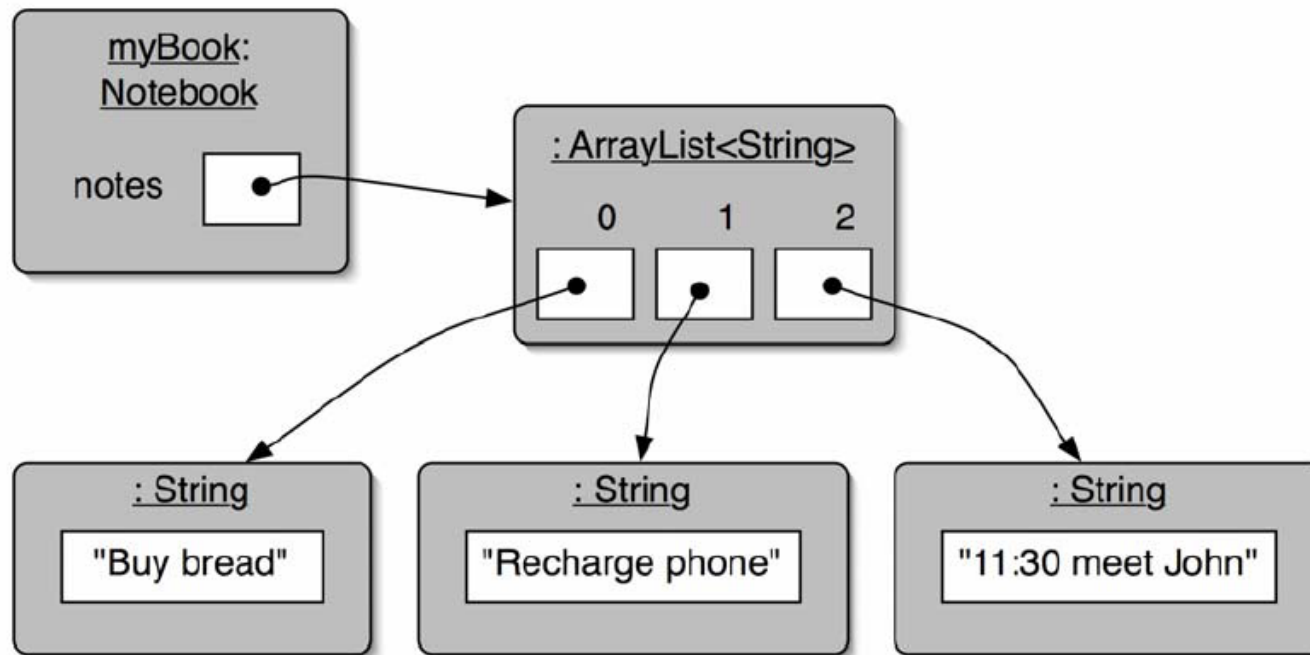
Añadir una nota

Devuelve el
número de notas

Estado después de añadir otra nota

```
myBook.storeNote("11:30 meet John");  
System.out.println(myBook.getNumberOfNotes());
```

Pantalla: 3



Recuperación de un objeto

```
public void showNote(int noteNumber) {  
    if(noteNumber < 0) {  
        // This is not a valid note number.  
    }  
    else if(noteNumber < getNumberOfNotes()) {  
        System.out.println(notes.get(noteNumber));  
    }  
    else {  
        // This is not a valid note number.  
    }  
}
```

Comprobación de la validez de los índices

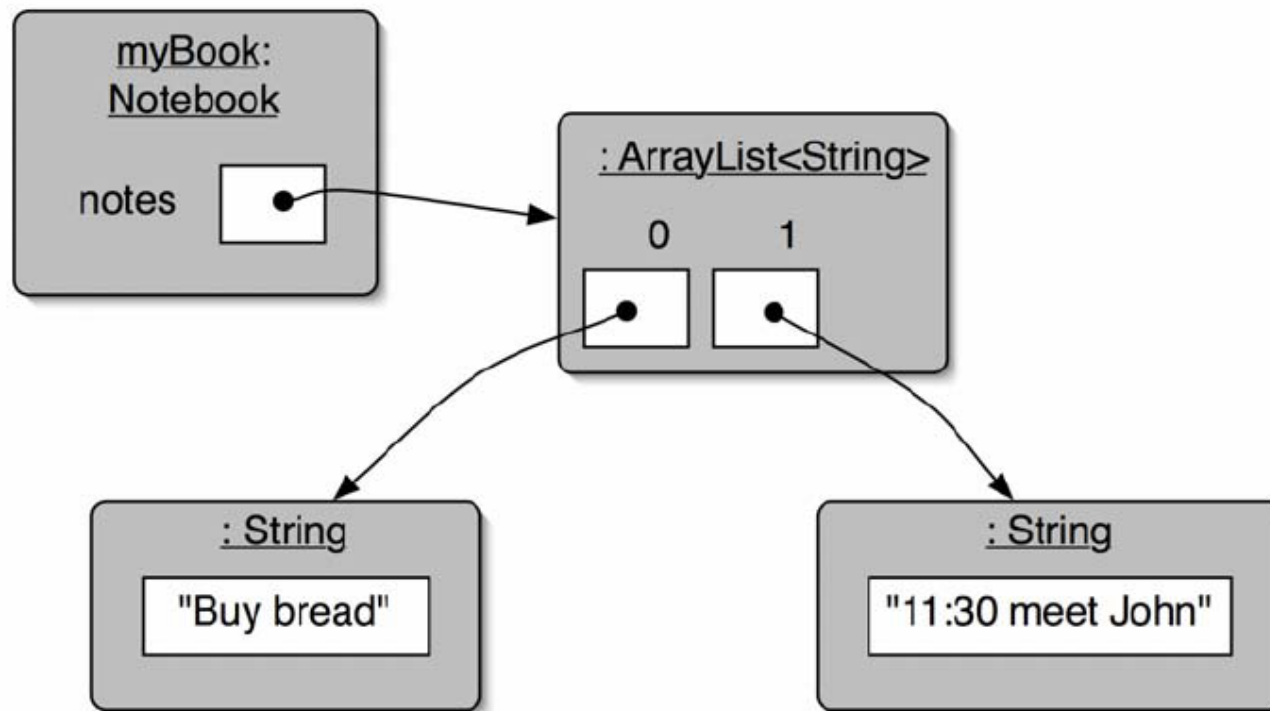
Recupera e imprime la nota

OJO: El añadir y eliminar objetos en el ArrayList afecta al uso del índice en la recuperación

Estado después de eliminar una nota

```
myBook.showNote(1);  
myBook.remove(1);  
myBook.showNote(1);
```

Pantalla: Recharge phone
11:30 meet John



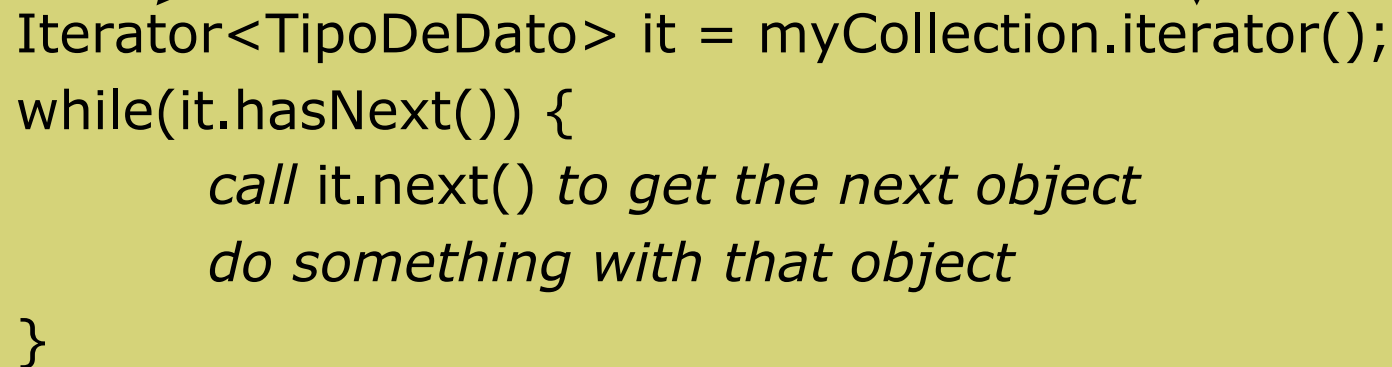
Buscando en una colección

```
int index = 0;
boolean found = false;
while(index < notes.size() && !found) {
    String note = notes.get(index);
    if(note.contains(searchString)) {
        // We don't need to keep looking.
        found = true;
    }
    else {
        index++;
    }
}
// Either we found it, or we searched the
// whole collection.
```

Usando un objeto Iterator (patrón iterador)

java.util.Iterator;

Devuelve un objeto
de tipo Iterator



```
Iterator<TipoDeDato> it = myCollection.iterator();  
while(it.hasNext()) {  
    call it.next() to get the next object  
    do something with that object  
}
```

```
public void printNotes() {  
    Iterator<String> it = notes.iterator();  
    while(it.hasNext())  
        System.out.println(it.next());  
}
```

Index vs. Iterator

- ❑ Varias formas de iterar en una colección:
 - Bucle for-each
 - ❑ Para procesar todos los elementos uno a uno
 - Bucle while
 - ❑ Si se quiere parar a mitad de camino
 - ❑ Para repeticiones que no se hacen sobre una colección
 - Objeto Iterator
 - ❑ Si se quiere parar a mitad de camino
 - ❑ A menudo en las colecciones donde el acceso indexado no es muy eficiente o imposible

- ❑ La iteración es un *patrón de diseño* muy importante

Resumen: iteración

- Las sentencias de bucle permiten repetir un bloque de instrucciones
 - El bucle for-each permite la iteración sobre una colección entera
 - El bucle while permite la repetición controlada por una expresión booleana
 - Todas las clases de colección suelen ofrecer objetos iteradores especiales que permiten el acceso secuencial a la colección

Resumen: Arrays

- ❑ Los arrays son apropiados cuando se trabaja con colecciones de tamaño fijo
 - Tienen una sintaxis especial y están soportados de manera nativa por el lenguaje de programación
- ❑ Los bucles for son una alternativa a los bucles while cuando el número de repeticiones es conocido

Resumen: colección *ArrayList*

- ❑ Las colecciones permiten guardar un número arbitrario de objetos
 - Que se pueden añadir y quitar
 - Cada elemento tiene un índice
 - Los valores de los índices pueden cambiar al eliminar o añadir elementos
 - Los principales métodos de *ArrayList* son *add*, *get*, *remove* y *size*
- ❑ Las librerías de clases suelen contener clases de colección bien probadas
- ❑ Las librerías de clases en Java se llaman *packages*
- ❑ La clase *ArrayList* del paquete *java.util* es un ejemplo de tipo genérico o parametrizado

La importancia de la ocultación de la información

```
public class Notebook{  
    private static final int SIZE_MAX = 10;  
    private String[] notes;  
    private int numberOfNotes;  
  
    public Notebook(){  
        notas = new String[SIZE_MAX];  
        numberOfNotes = 0;  
    }  
    public void storeNote (String note){  
        if (numberOfNotes < SIZE_MAX){  
            notes[numberOfNotes] = note;  
            numberOfNotes++;  
        }  
        else  
            System.out.println("Can`t store it");  
    }  
    public int getNumberOfNotes(){  
        return numberOfNotes;  
    }  
    ...  
}
```

```
import java.util.ArrayList;  
  
public class Notebook {  
    private ArrayList<String> notes;  
  
    public Notebook(){  
        notes = new ArrayList<String>();  
    }  
  
    public void storeNote(String note) {  
        notes.add(note);  
    }  
  
    public int getNumberOfNotes (){  
        return notes.size();  
    }  
    ...  
}
```

Fijate: Sólo cambia la implementación, no el interfaz de la clase

Contenido

- ❑ Colecciones
 - Tamaño fijo: *Array*
 - Tamaño indefinido: *ArrayList*
 - Iteración
- ❑ La importancia de la ocultación de la información
- ❑ Igualdad vs. Identidad

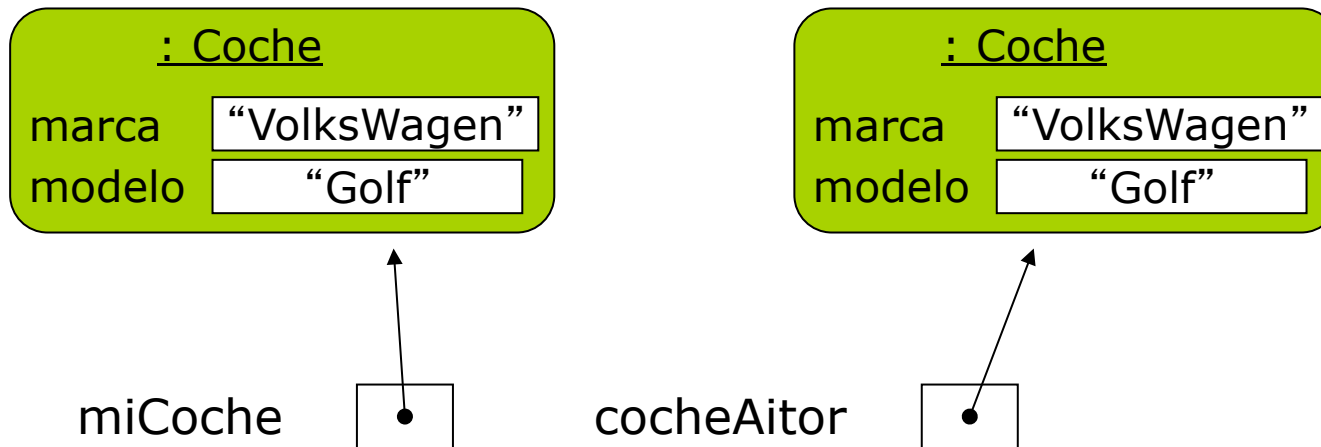
Objetos: identidad

- Los objetos son instancias de las clases
 - Cada objeto tiene sus **propios valores de los atributos** definidos en la clase
 - Cada objeto tiene su **propia identidad**

Igualdad vs. Identidad

□ Situación:

- El coche que tengo yo es un Volkswagen Golf
- ¿Qué pensáis si os comento que Aitor y yo tenemos el mismo coche?

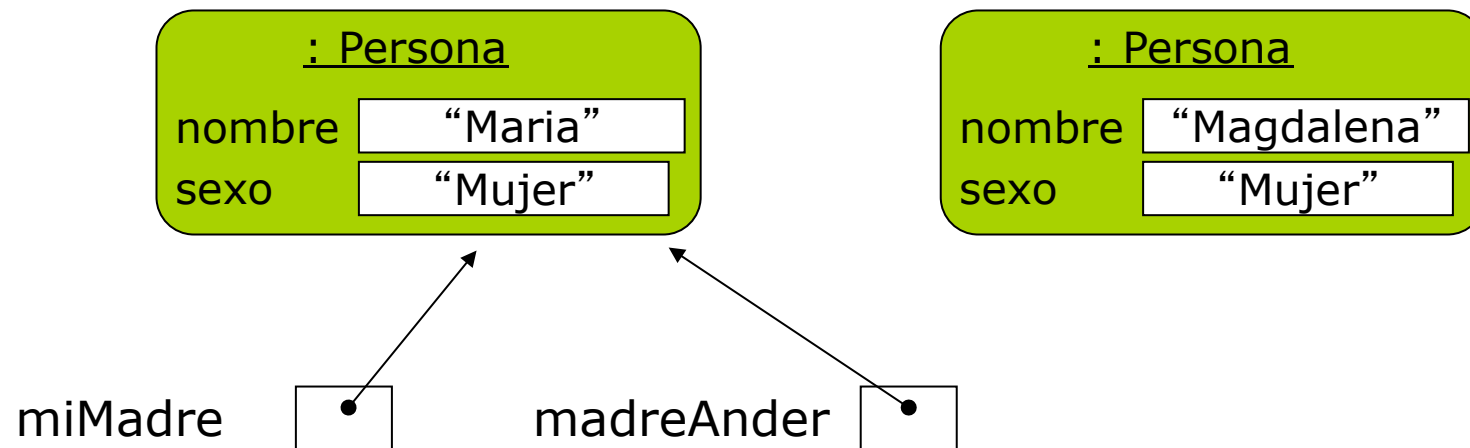


`miCoche == cocheAitor ? _____`

Igualdad vs. Identidad (2)

□ Situación:

- ¿Qué pensáis si os comento que Ander y yo tenemos la misma madre?



miMadre == madreAnder ? _____

Igualdad vs. Identidad (3)

```
if (cadena == "hola") {  
    . . .  
}
```

Prueba la
identidad

```
if (cadena.equals("hola")) {  
    . . .  
}
```

Prueba la
igualdad

Regla:

- ❑ Los tipos básicos (incluidos referencias a objetos) se comparan con `==`
- ❑ Todos los objetos Java se deben comparar siempre con `.equals`

Igualdad vs. Identidad (4): ejemplo

```
public class Coche {  
    private String marca;  
    private String modelo;  
    ...  
    public boolean equals(Coche coche){  
        return this.marca.equals(coche.marca)  
            && this.modelo.equals(coche.modelo);  
    }  
    ...  
}
```

Definición de la
propiedad de
igualdad

```
...  
if (miCoche.equals(cocheAitor)){  
    ...  
}
```

Prueba de la
igualdad

Igualdad vs. Identidad (5): ejemplo

```
public class PruebaIgualdad {  
    public static void main(String args[]) {  
        String str1, str2;  
        str1 = "Texto de prueba.";  
        str2 = str1;  
  
        System.out.println("String1: " + str1);  
        System.out.println("String2: " + str2);  
        System.out.println("El mismo objeto? " + (str1 == str2));  
  
        str2 = new String(str1);  
  
        System.out.println("String1: " + str1);  
        System.out.println("String2: " + str2);  
        System.out.println("El mismo objeto? " + (str1 == str2));  
        System.out.println("El mismo valor? "+ str1.equals(str2));  
    }  
}
```

Igualdad vs. Identidad (6): ejemplo

```
class ArrayIsObject {  
    public static void main (String [] args) {  
        double [] a = { 100.5, 200.5, 300.5 };  
        double [] b = { 100.5, 200.5, 300.5 };  
  
        double [] c = b;  
  
        System.out.println ("a's class is " + a.getClass ());  
  
        System.out.println ("a and b are " + ((a == b) ? "" : "not ") + "the same");  
        System.out.println ("a and b are " + ((a.equals (b)) ? "" : "not ") + "equal ");  
  
        System.out.println ("a and b are " + ((a == b) ? "" : "not ") + "the same");  
        System.out.println ("b and c are " + ((b.equals (c)) ? "" : "not ") + "equal ");  
  
        double [] d = (double []) c.clone ();  
  
        System.out.println ("a and b are " + ((a == b) ? "" : "not ") + "the same");  
        System.out.println ("c and d are " + ((c.equals (d)) ? "" : "not ") + "equal ");  
  
        for (int i = 0; i < d.length; i++)  
            System.out.println (d [i]);  
    }  
}
```

Igualdad vs. Identidad (7): ejemplo

```
import java.util.ArrayList;
public class ArrayListIsObject {
    public static void main (String [] args)
    {
        ArrayList<String> a = new ArrayList<String>();
        a.add("x"); a.add("y"); a.add("z");

        ArrayList<String> b = new ArrayList<String>();
        b.add("x"); b.add("y"); b.add("z");

        ArrayList<String> c = b;

        System.out.println ("a's class is " + a.getClass ());

        System.out.println ("a and b are " + ((a == b) ? "" : "not ") + "the same");
        System.out.println ("a and b are " + ((a.equals (b)) ? "" : "not ") + "equal ");

        System.out.println ("b and c are " + ((b == c) ? "" : "not ") + "the same");
        System.out.println ("b and c are " + ((b.equals (c)) ? "" : "not ") + "equal ");

        ArrayList<String> d = (ArrayList<String>) c.clone ();
        System.out.println ("c and d are " + ((c == d) ? "" : "not ") + "the same");
        System.out.println ("c and d are " + ((c.equals (d)) ? "" : "not ") + "equal ");

        for (String s: d)
            System.out.println (s);
    }
}
```