Librerías de Java

Java y Servicios Web I
Master en Ingeniería Matemática

Manuel Montenegro
Dpto. Sistemas Informáticos y Computación

Desp. 467 (Mat)

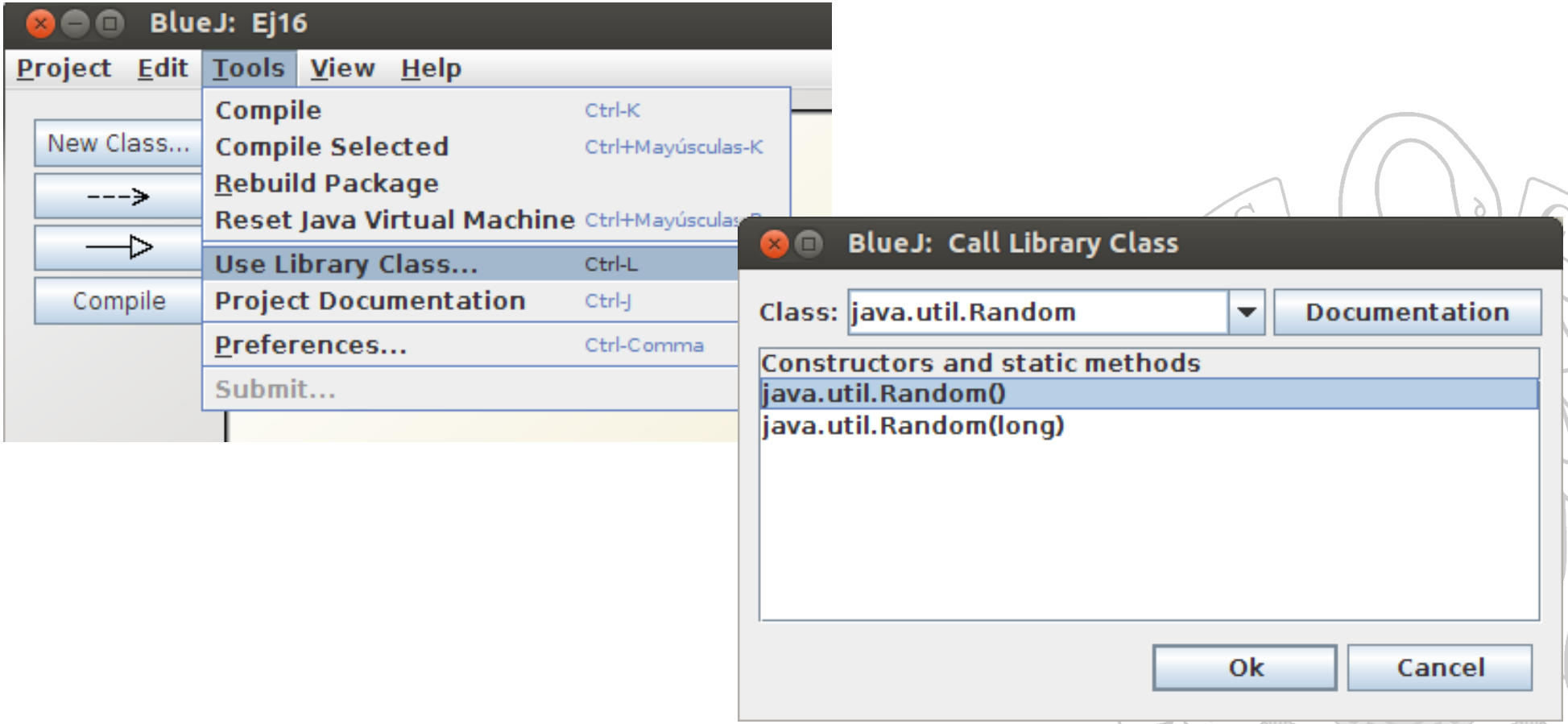
montenegro@fdi.ucm.es

Contenidos

- Clases envoltorio: `Integer`, `Character`, ...
- `String` y `StringBuffer`
- Números aleatorios
- Fechas y horas
- Expresiones regulares
- Utilidades con arrays
- *Java Collections Framework*
- Clases genéricas

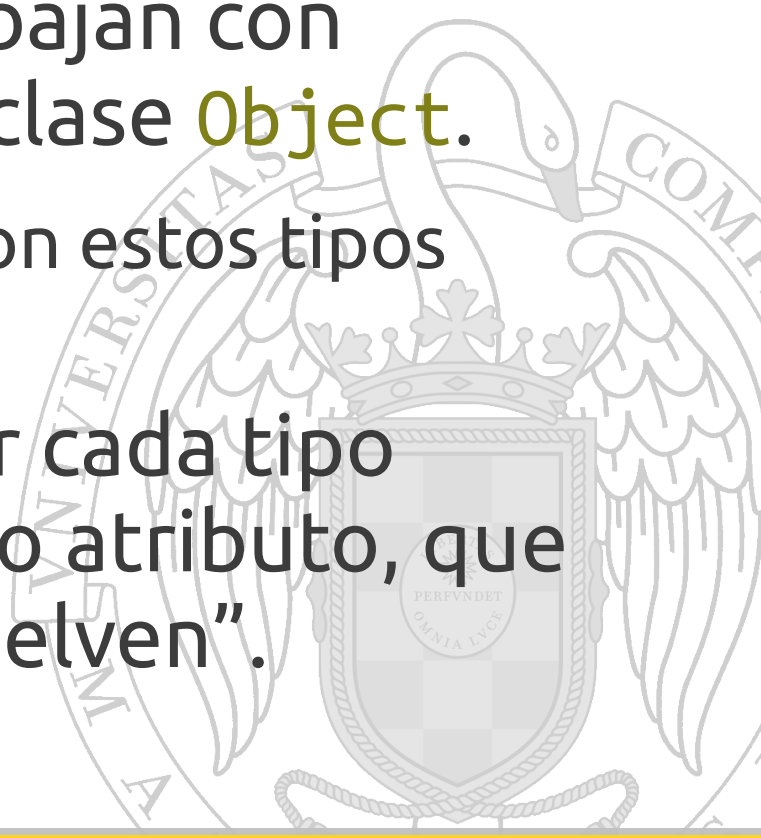
Librerías de Java

- El entorno BlueJ permite crear instancias de las clases contenidas en la librería de Java



Clases envoltorio (*wrapper*)

- En ocasiones es útil tratar los tipos de datos básicos como objetos.
  - `int, char, boolean, byte, float, double,...`
- Muchas funciones y clases trabajan con elementos que heredan de la clase `Object`.
  - No funcionarán directamente con estos tipos básicos.
- Existe una clase envoltorio por cada tipo básico. Cada una tiene un único atributo, que es del tipo básico al que "envuelven".



Clases envoltorio (*wrapper*)

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

La clase Integer

- Constructores:
  - `Integer(int valor)`
  - `Integer(String valor)`
- Método de acceso al valor básico:
  - `int getValue()`
  - `String toString()` (heredado de `Object`)

```
Integer x = new Integer(5), y = new Integer(9);
int z = x.intValue() + y.intValue();

System.out.printf("%s + %s = %d", x, y, z);
```

Boxing y *Unboxing* automáticos

- Desde la versión 5 de Java, se convierte automáticamente entre las clases envoltorios y sus correspondientes tipos básicos.
  - Si se introduce un tipo básico donde se espera un objeto de una clase envoltorio, se llama al constructor correspondiente (boxing).
  - Si se introduce un objeto de una clase envoltorio donde se espera un tipo básico, se llama al método de acceso correspondiente (unboxing).

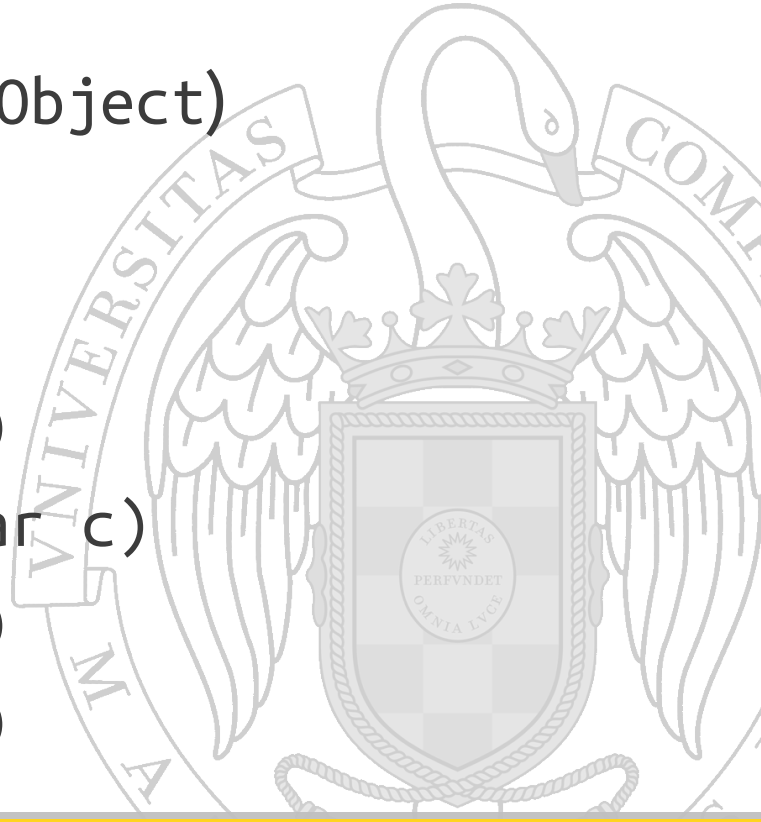
```
Integer x = 5, y = 9;
int z = x + y;

System.out.printf("%s + %s = %d", x, y, z);
```

Boxing

Unboxing

La clase `Character`

- Constructor:
  - `Character(char valor)`
- Método de acceso al valor básico:
  - `char charValue()`
  - `String toString()` (heredado de `Object`)
- Métodos de utilidad:
  - `static boolean isDigit(char c)`
  - `static boolean isLetter(char c)`
  - `static boolean isWhiteSpace(char c)`
  - `static char toLowerCase(char c)`
  - `static char toUpperCase(char c)`



Contenidos

- Clases envoltorio: `Integer`, `Character`, ...
- `String` y `StringBuffer`
- Números aleatorios
- Fechas y horas
- Expresiones regulares
- Utilidades con arrays
- *Java Collections Framework*
- Clases genéricas

La clase String

- Métodos de utilidad:
  - `char charAt(int indice)`
  - `int compareTo(String otra)`
  - `int compareToIgnoreCase(String otra)`
  - `int indexOf(String str)`
  - `int indexOf(String str, int inicio)`
  - `int length()`
  - `String substring(int inicio, int fin)`
  - `String toUpperCase()`
  - `String toLowerCase()`
  - `String trim()`

La clase String

```
String cadena = "Esto es un ejemplo";
System.out.println(cadena.charAt(2));
System.out.println(cadena.indexOf("es"));
System.out.println(cadena.toLowerCase());
System.out.println(cadena.toUpperCase());
System.out.printf("|%s|", "   cadena   ".trim());
```

t

5

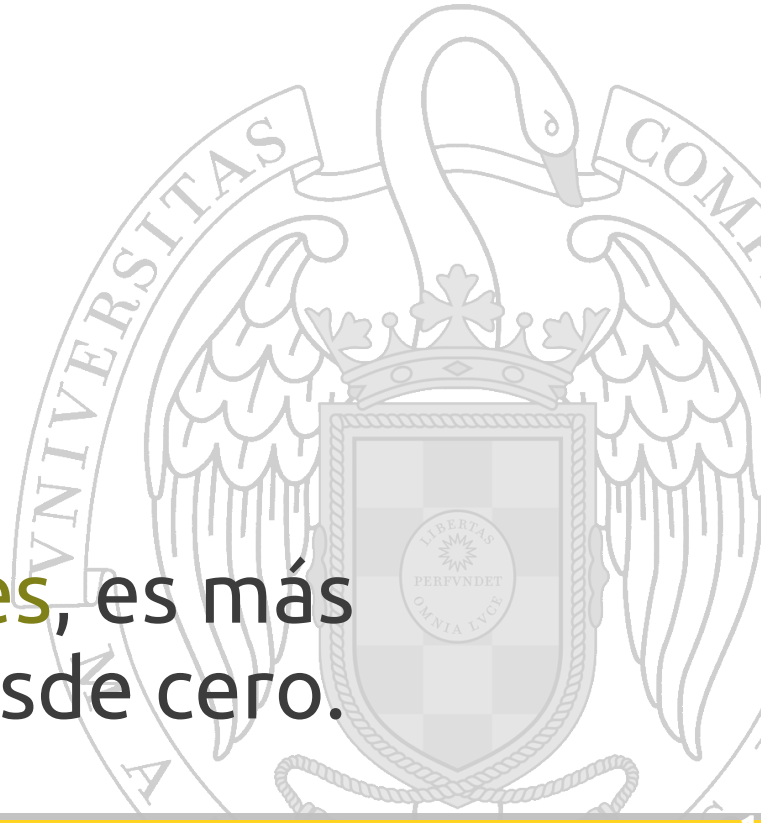
esto es un ejemplo

ESTO ES UN EJEMPLO

|cadena|

La clase `StringBuffer`

- Los objetos de la clase `String` son inmutables.
  - No pueden cambiarse una vez creados.
- Un objeto `StringBuffer` puede ser modificado tras su creación.
  - Método `append()`
  - Método `delete()`
  - Método `insert()`
  - ...
- En el caso de cadenas mutables, es más eficiente que crear `Strings` desde cero.



Contenidos

- Clases envoltorio: `Integer`, `Character`, ...
- `String` y `StringBuffer`
- Números aleatorios
- Fechas y horas
- Expresiones regulares
- Utilidades con arrays
- *Java Collections Framework*
- Clases genéricas

La clase Random

- Paquete `java.util`.
- Representa una secuencia de números aleatorios.
- Constructores:
  - `Random()`
  - `Random(long semilla)`
- Métodos:
  - `int nextInt(int n)` `// Unif. en {0,...,n-1}`
  - `double nextDouble()` `// Unif(0,1)`
  - `double nextGaussian()` `// N(0,1)`

Contenidos

- Clases envoltorio: `Integer`, `Character`, ...
- `String` y `StringBuffer`
- Números aleatorios
- Fechas y horas
- Expresiones regulares
- Utilidades con arrays
- *Java Collections Framework*
- Clases genéricas

La clase `Date`

- Sus instancias almacenan una fecha y una hora.
- Constructores:
  - `Date()`
  - `Date(long tiempo)`
- Métodos:
  - `long getTime()`
  - `String toString()`
- Los valores `long` indicados en el constructor indican el número de milisegundos transcurridos desde el 1 de Enero de 1970, 00:00:00 GMT.

La clase `GregorianCalendar`

- Permiten convertir la fecha/hora contenida en un objeto `Date` a una representación estructurada en dia, mes, año, horas, minutos, segundos, etc.
- Constructor:
  - `GregorianCalendar()`
- Métodos:
  - `void setTime(Date date)`
  - `int get(int campo)`

La clase GregorianCalendar

- El argumento campo del método get() recibe una de las constantes definidas como atributos estáticos en la clase GregorianCalendar.

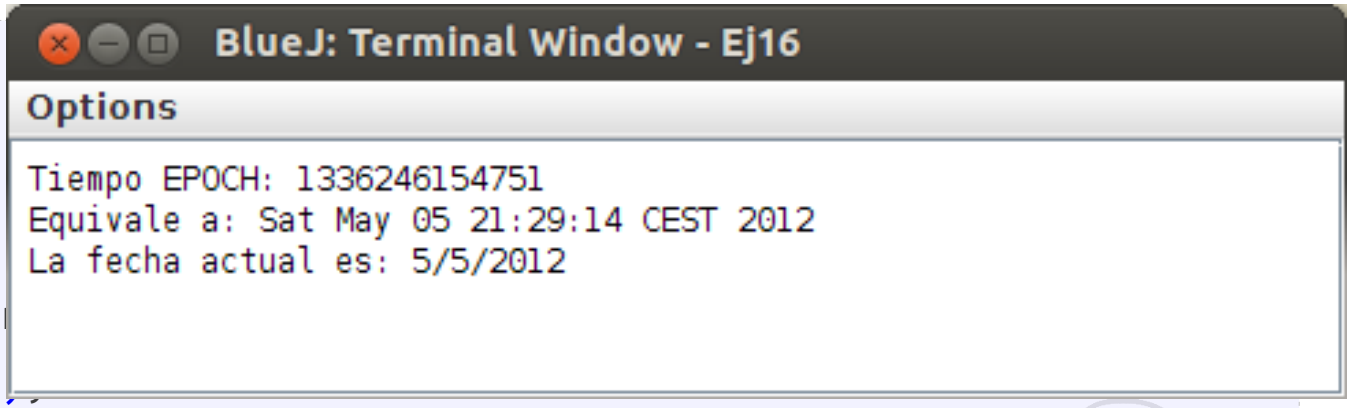
Campos disponibles
GregorianCalendar.ERA
GregorianCalendar.YEAR
GregorianCalendar.MONTH
GregorianCalendar.DAY_OF_MONTH
GregorianCalendar.DAY_OF_WEEK
GregorianCalendar.WEEK_OF_YEAR
GregorianCalendar.HOUR
GregorianCalendar.HOUR_OF_DAY
GregorianCalendar.MINUTE
GregorianCalendar.SECOND
GregorianCalendar.MILLISECOND
...

Ejemplo

```
import java.util.*;

public class DateTest
{
    public static void main(String[] args) {
        Date d = new Date();
        System.out.printf("Tiempo EPOCH: %d\n", d.getTime());
        System.out.printf("Equivale a: %s\n", d.toString());
        GregorianCalendar gc = new GregorianCalendar();
        gc.setTime(d);
        System.out.printf("La fecha actual es: %d/%d/%d",
            gc.get(GregorianCalendar.DAY_OF_MONTH),
            gc.get(GregorianCalendar.MONTH) + 1,
            gc.get(GregorianCalendar.YEAR));
    }
}
```

Ejemplo

```
import java.util.*;

public class DateTest
{
    public static void mai
        Date d = new Date();
        System.out.printf("Tiempo EPOCH: %d\n", d.getTime());
        System.out.printf("Equivale a: %s\n", d.toString());
        GregorianCalendar gc = new GregorianCalendar();
        gc.setTime(d);
        System.out.printf("La fecha actual es: %d/%d/%d",
            gc.get(GregorianCalendar.DAY_OF_MONTH),
            gc.get(GregorianCalendar.MONTH) + 1,
            gc.get(GregorianCalendar.YEAR));
    }
}
```

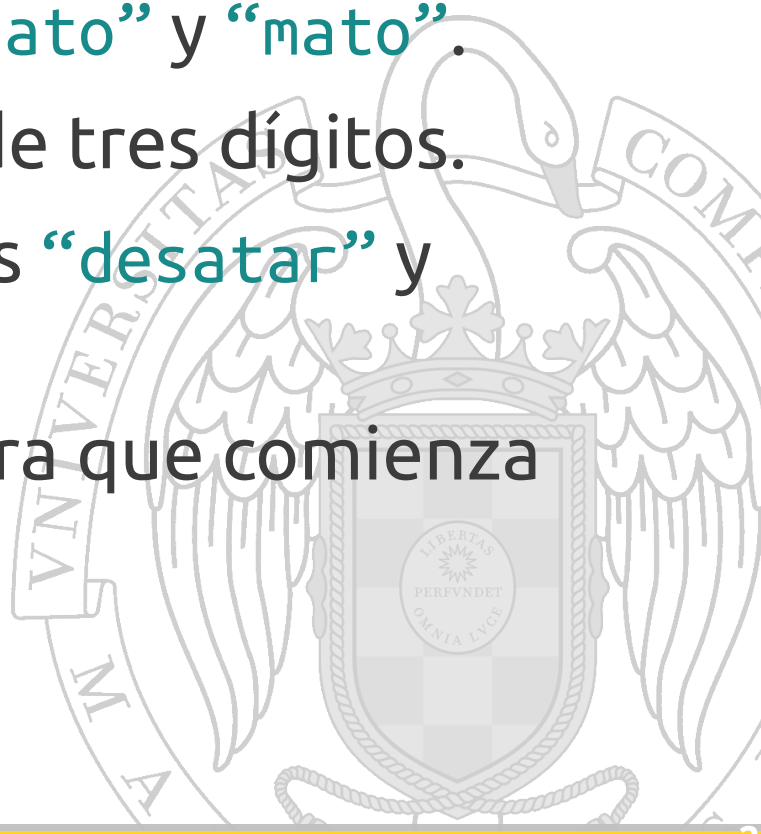


Contenidos

- Clases envoltorio: `Integer`, `Character`, ...
- `String` y `StringBuffer`
- Números aleatorios
- Fechas y horas
- Expresiones regulares
- Utilidades con arrays
- *Java Collections Framework*
- Clases genéricas

Expresiones regulares

- Una expresión regular es un patrón que describe un conjunto de cadenas.
- Por ejemplo:
  - `[gm]ato` describe las palabras "`gato`" y "`mato`".
  - `\d\d\d` describe una secuencia de tres dígitos.
  - `(des)?atar` describe las palabras "`desatar`" y "`atar`".
  - `[A-Z][a-z]*` describe una palabra que comienza con letra mayúscula.



Sintaxis de expresiones regulares

Clases de caracteres

Símbolo	Caracteres admisibles
`[abc]`	a,b,c
`[^abc]`	Cualquier carácter excepto a,b,c
`[a-z]`	Carácter de *a* a *z*
`[a-z0-9]`	Carácter de *a* a *z*, y de *0* a *9*.
`.`	Cualquier carácter
`\d`	Carácter numérico
`\D`	Carácter no numérico (= [^\d])
`\s`	Carácter blanco, tabulador, salto de línea, etc.
`\S`	Carácter no blanco, tabulador, salto de línea, etc.
`\w`	Carácter alfanumérico, o símbolo de subrayado.
`\W`	Carácter no alfanumérico, ni símbolo de subrayado

Sintaxis de expresiones regulares

Capturadores de límites

Símbolo	Captura
`^`	Inicio de línea
`$`	Fin de línea
`\b`	Límite de palabra
`\A`	Inicio de entrada
`\G`	Fin de entrada

Operadores

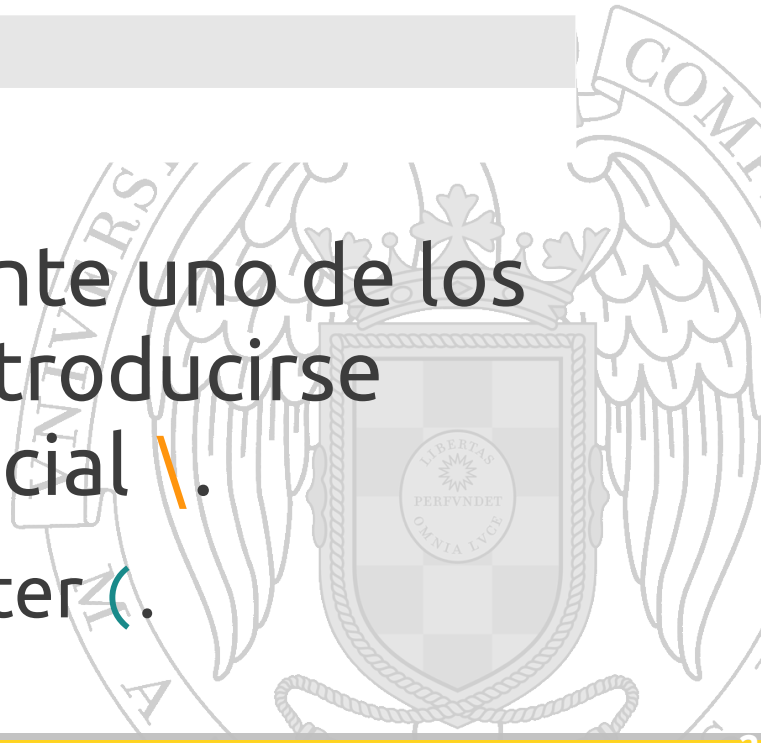
Símbolo	Captura
`XY`	X seguido de Y
`X|Y`	X o Y
`(X)`	Agrupamiento: X como un grupo de captura
`\número`	Referencia a grupo de captura anterior

Sintaxis de expresiones regulares

Cuantificadores

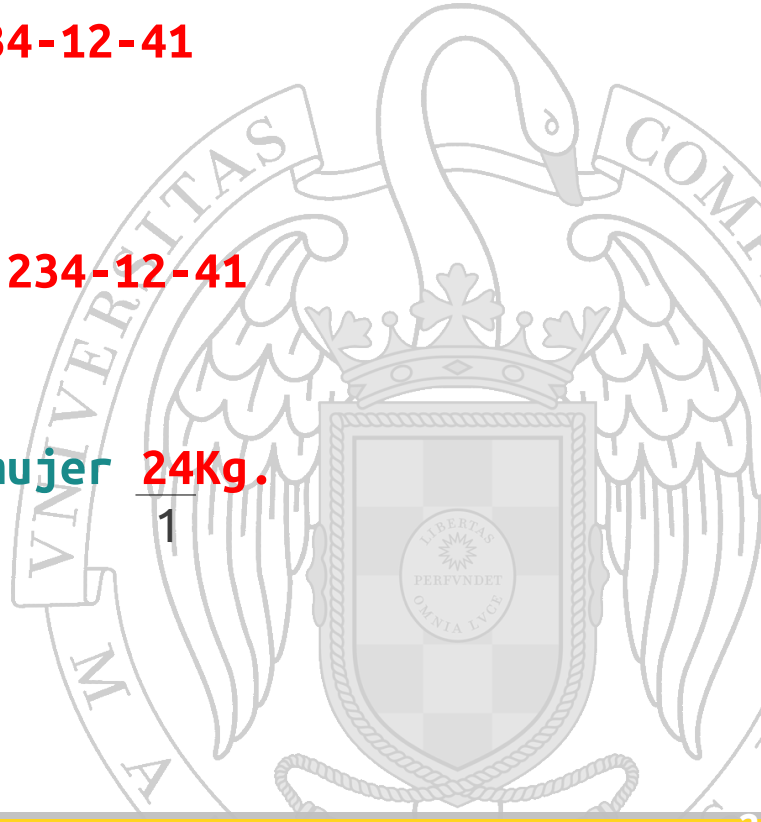
Expr	Captura
`X?`	X una vez, o ninguna.
`X*`	X cero o más veces.
`X+`	X una o más veces.
`X{n}`	X repetido n veces.
`X{n,}`	X repetido n veces o más.
`X{n,m}`	X repetido de n a m veces.

- Si se quiere capturar literalmente uno de los caracteres especiales, ha de introducirse precedido por el carácter especial `\`.
  - Por ejemplo, `\(` captura el carácter `(`.



Sintaxis de expresiones regulares

Expresión regular	Ajuste
`(des)?hace`	`hacer por deshacer.` (1)
`^(des)?hace`	`hacer por deshacer.`
`[0-9-]{3,}$`	`Llama al número 91-234-12-41`
`\(([0-9]{2,3})\) [0-9-]{3,}`	`Llama al número (91) 234-12-41` (1)
`([0-9]+)\w?Kg\.`	`Yo peso 63 Kg. y mi mujer 24Kg.` (1) (1)
`([cb]at)\1`	`catbat, but batbat` (1)



Sintaxis de expresiones regulares

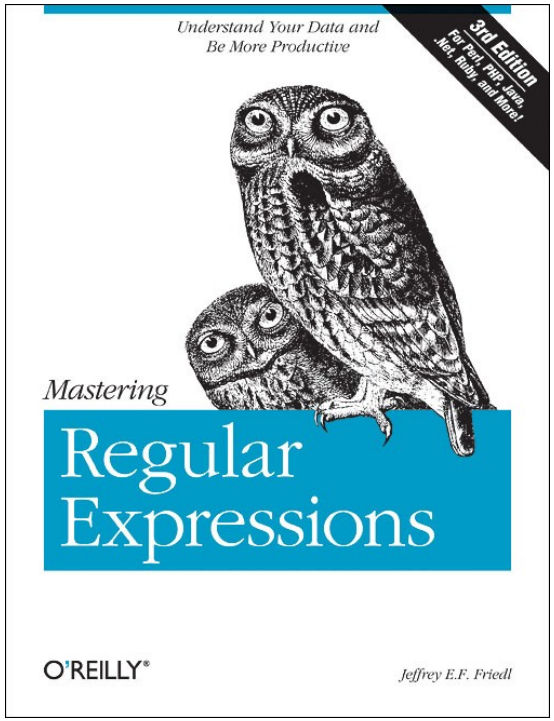
Cuantificadores (II)

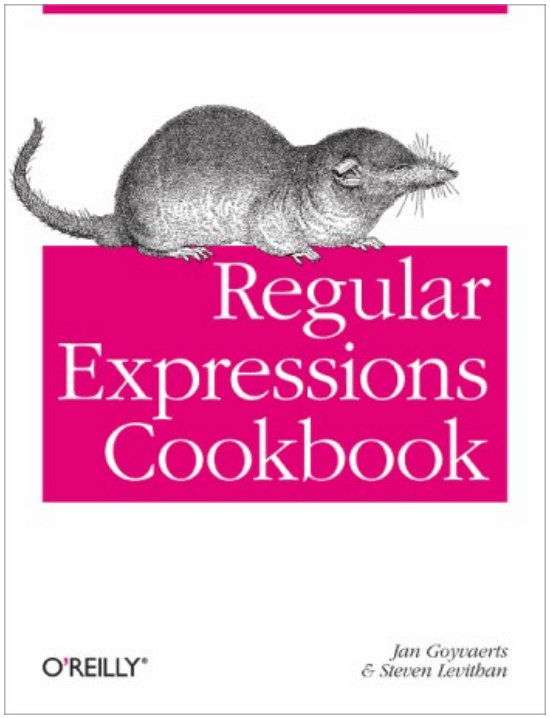
Voraces	Reticentes	Posesivos
X?	X??	X?+
X*	X*?	X*+
X+	X+?	X++
X{n}	X{n}?	X{n}+
X{n,}	X{n,}?	X{n,}+
X{n,m}	X{n,m}?	X{n,m}+

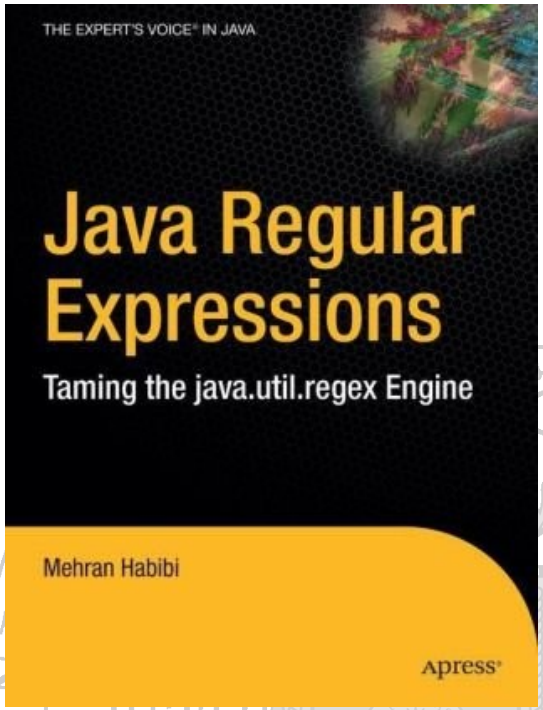
`[ab]*b`	**aabaabaa**	Intenta ajustar la cadena total. Si no es posible, retrocede hasta lograr un ajuste.
`[ab]*?b`	**aabaabaa**	Intenta ajustar la cadena vacía. Si no es posible, avanza hasta lograr un ajuste.
`[ab]*+b`	**aabaabaa**	Intenta ajustar la cadena total. Si no es posible, no hay ajuste.

Más información

- http://www.regular-expressions.info

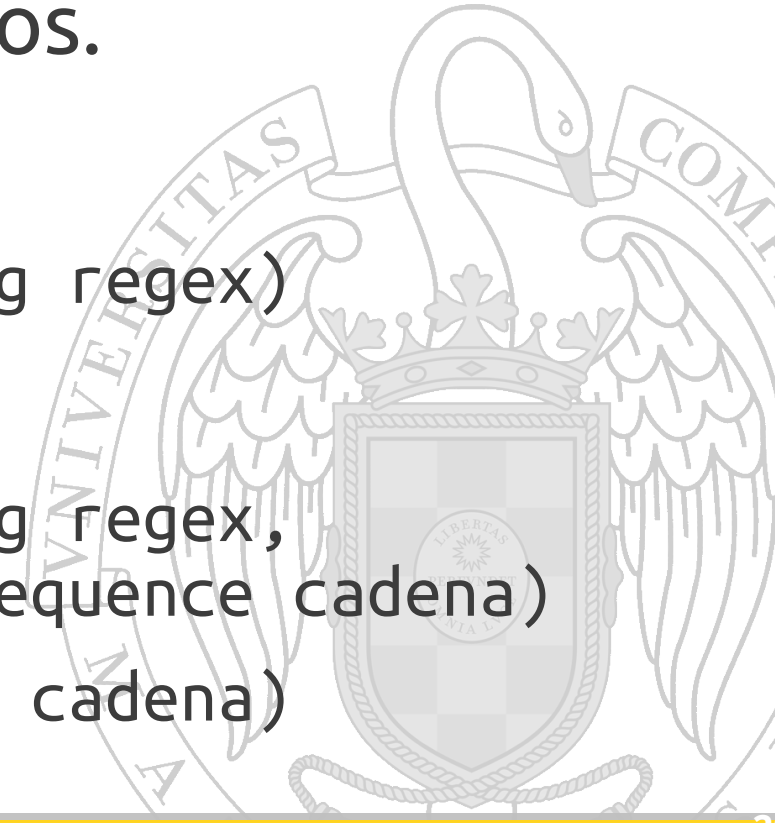






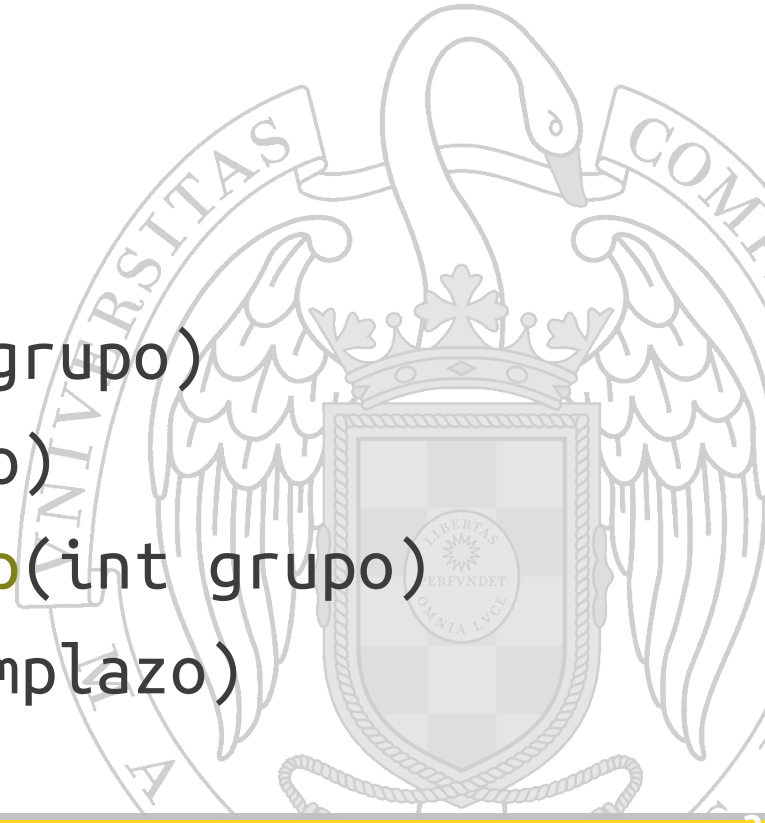
La clase Pattern

- Paquete `java.util.regex`
- Sus objetos representan expresiones regulares compiladas.
- No tiene constructores públicos.
- Creación:
  - `static Pattern compile(String regex)`
- Métodos:
  - `static boolean matches(String regex, CharSequence cadena)`
  - `Matcher matcher(CharSequence cadena)`



La clase `Matcher`

- Realiza el reconocimiento de una expresión regular a una cadena específica
- No tiene constructor público.
- Métodos:
  - `boolean matches()`
  - `boolean find()`
  - `int start() / int start(int grupo)`
  - `int end() / int end(int grupo)`
  - `String group() / String group(int grupo)`
  - `String replaceAll(String reemplazo)`



Ejemplo

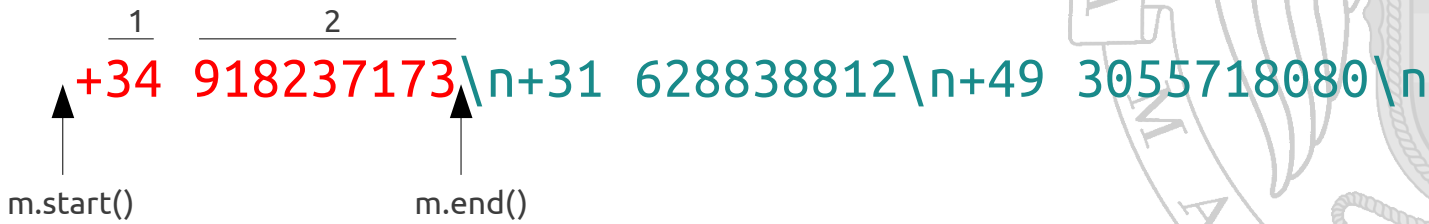
```
import java.util.regex.*;


public class RegexTest
{
    private static final String cadena = "+34 918237173\n" +
                                         "+31 628838812\n" +
                                         "+49 3055718080\n";

    private static final String regex = "\\+(\\d+)\\s+(\\d+)" ;
    public static void main(String[] args) {

        Pattern p = Pattern.compile(RegexTest.regex);
        Matcher m = p.matcher(RegexTest.cadena);
        while (m.find()) {
            System.out.printf("Ajuste encontrado desde %d hasta %d\n", m.start(), m.end());
            System.out.printf("Prefijo: %s, Teléfono: %s\n", m.group(1), m.group(2));
        }

    }
}
```

```
+34 918237173\n+31 628838812\n+49 3055718080\n
```

Ejemplo

```
import java.util.regex.*;


public class RegexTest
{
    private static final String cadena = "+34 918237173\n" +
                                         "+31 628838812\n" +
                                         "+49 3055718080\n";

    private static final String regex = "\\+(\\d+)\\s+(\\d+)" ;
    public static void main(String[] args) {

        Pattern p = Pattern.compile(RegexTest.regex);
        Matcher m = p.matcher(RegexTest.cadena);
        while (m.find()) {
            System.out.printf("Ajuste encontrado desde %d hasta %d\n", m.start(), m.end());
            System.out.printf("Prefijo: %s, Teléfono: %s\n", m.group(1), m.group(2));
        }

    }
}
```

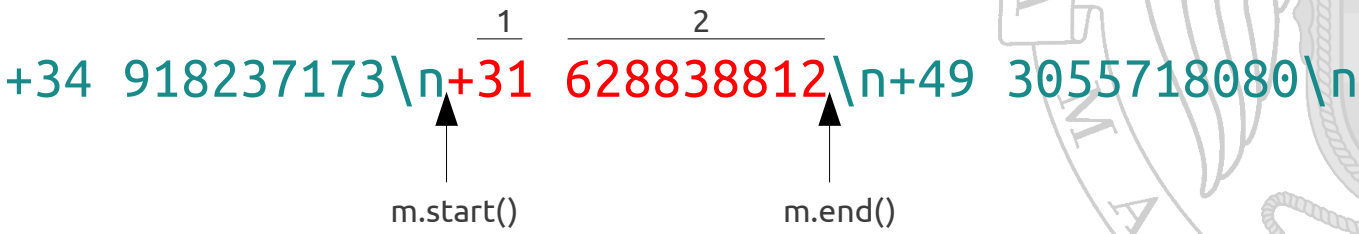
m.find() = true



Ejemplo

```
import java.util.regex.*;


public class RegexTest
{
    private static final String cadena = "+34 918237173\n" +
                                         "+31 628838812\n" +
                                         "+49 3055718080\n";

    private static final String regex = "\\+(\\d+)\\s+(\\d+)" ;
    public static void main(String[] args) {

        Pattern p = Pattern.compile(RegexTest.regex);
        Matcher m = p.matcher(RegexTest.cadena);
        while (m.find()) {
            System.out.printf("Ajuste encontrado desde %d hasta %d\n", m.start(), m.end());
            System.out.printf("Prefijo: %s, Teléfono: %s\n", m.group(1), m.group(2));
        }

    }
}
```

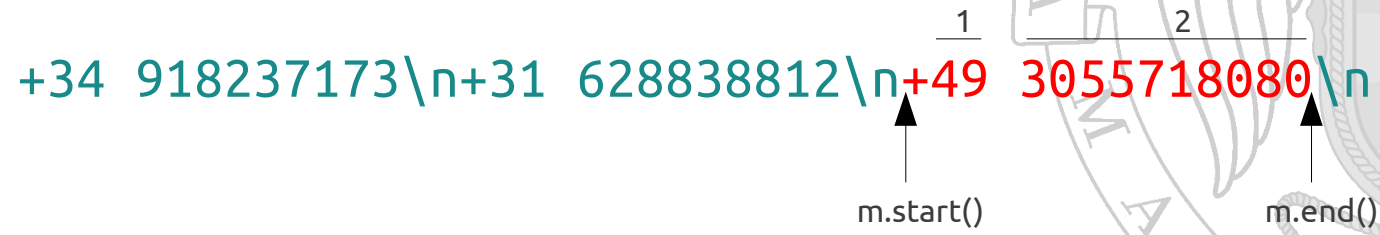
m.find() = true



Ejemplo

```
import java.util.regex.*;


public class RegexTest
{
    private static final String cadena = "+34 918237173\n" +
                                         "+31 628838812\n" +
                                         "+49 3055718080\n";

    private static final String regex = "\\+(\\d+)\\s+(\\d+)" ;
    public static void main(String[] args) {

        Pattern p = Pattern.compile(RegexTest.regex);
        Matcher m = p.matcher(RegexTest.cadena);
        while (m.find()) {
            System.out.printf("Ajuste encontrado desde %d hasta %d\n", m.start(), m.end());
            System.out.printf("Prefijo: %s, Teléfono: %s\n", m.group(1), m.group(2));
        }

    }
}
```

m.find() = true



Ejemplo

```
import java.util.regex.*;


public class RegexTest
{
    private static final String cadena = "+34 918237173\n" +
                                         "+31 628838812\n" +
                                         "+49 3055718080\n";

    private static final String regex = "\\+(\\d+)\\s+(\\d+)" ;
    public static void main(String[] args) {

        Pattern p = Pattern.compile(RegexTest.regex);
        Matcher m = p.matcher(RegexTest.cadena);
        while (m.find()) {
            System.out.printf("Ajuste encontrado desde %d hasta %d\n", m.start(), m.end());
            System.out.printf("Prefijo: %s, Teléfono: %s\n", m.group(1), m.group(2));
        }

    }
}
```

m.find() = false

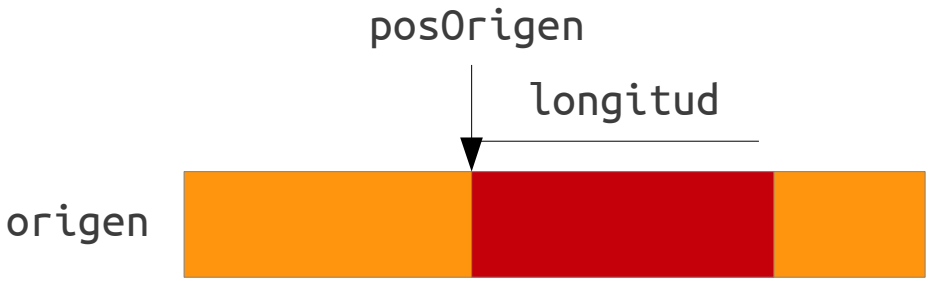
+34 918237173\n+31 628838812\n+49 3055718080\n

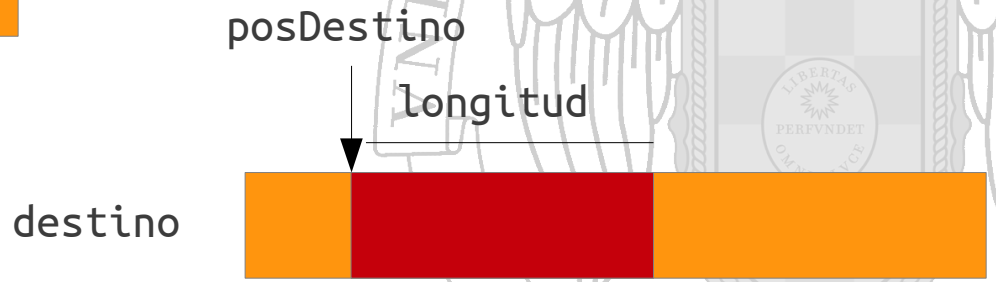
Contenidos

- Clases envoltorio: `Integer`, `Character`, ...
- `String` y `StringBuffer`
- Números aleatorios
- Fechas y horas
- Expresiones regulares
- Utilidades con arrays
- *Java Collections Framework*
- Clases genéricas

Arrays

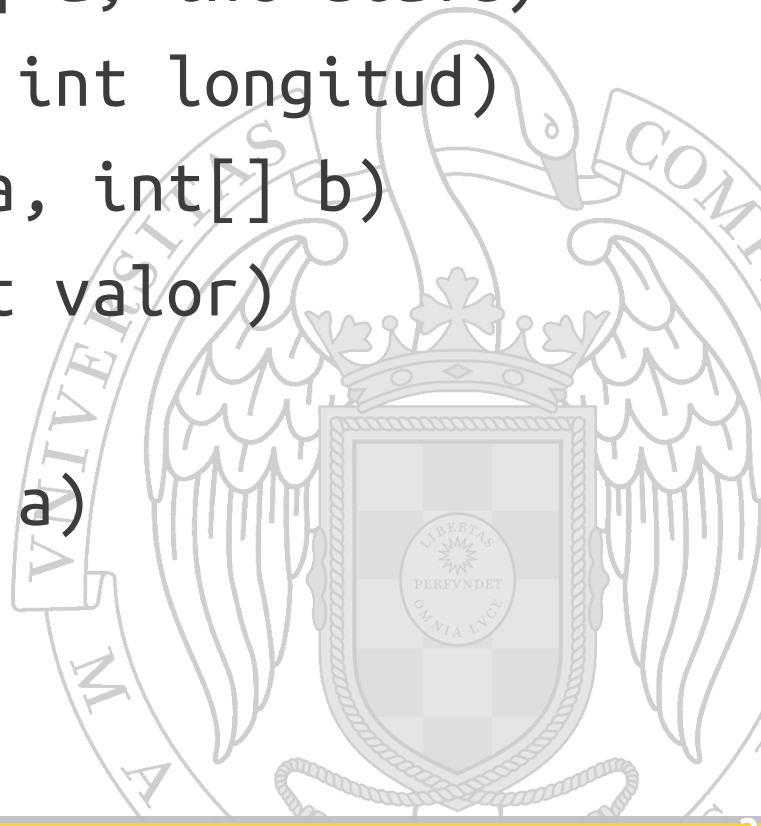
- El método `System.arrayCopy` permite copiar elementos de un array en otro.
  - `static void arrayCopy(Object origen, int posOrigen, Object destino, int posDestino, int longitud);`





Arrays

- La clase `java.util.Arrays` contiene una batería de métodos estáticos útiles para el manejo de arrays de cualquier tipo.
  - `static int binarySearch(int[] a, int clave)`
  - `static int[] copyOf(int[] a, int longitud)`
  - `static boolean equals(int[] a, int[] b)`
  - `static void fill(int[] a, int valor)`
  - `static void sort(int[] a)`
  - `static String toString(int[] a)`



Arrays

```
public class ArraysTest
{
    public static void main(String[] args) {
        int[] arr = new int[10];
        Random r = new Random();
        for (int i = 0; i < arr.length; i++) {
            arr[i] = r.nextInt(100);
        }
        System.out.println("Array inicial: " + Arrays.toString(arr));
        Arrays.sort(arr);
        System.out.println("Array ordenado: " + Arrays.toString(arr));
        System.out.print("Introduce número a buscar: ");
        int num = new Scanner(System.in).nextInt();
        System.out.print("La búsqueda binaria devuelve: ");
        System.out.println(Arrays.binarySearch(arr, num));
    }
}
```

Contenidos

- Clases envoltorio: `Integer`, `Character`, ...
- `String` y `StringBuffer`
- Números aleatorios
- Fechas y horas
- Expresiones regulares
- Utilidades con arrays
- *Java Collections Framework*
- Clases genéricas

Java Collections Framework

- Un array permite tener un conjunto de objetos indexados por un número.
- El tamaño de un array queda fijo en el momento de su creación.
- En la práctica, el número de elementos que ha de contener un array es desconocido.
- Existen tipos de datos más sofisticados para almacenar un número variable de elementos.
- La *Java Collections Framework* es un conjunto de clases (contenidas en el paquete `java.util`) que implementan estos tipos de datos.

Java Collections Framework

Contenedores

Colecciones

Elementos individuales.
Búsqueda por elemento

Listas

Secuencia (con duplicados)
Importa orden

Conjuntos

Secuencia (sin duplicados)
No importa orden

Arrays asociativos (*Maps*)

Asocian claves con valores
Búsqueda por clave

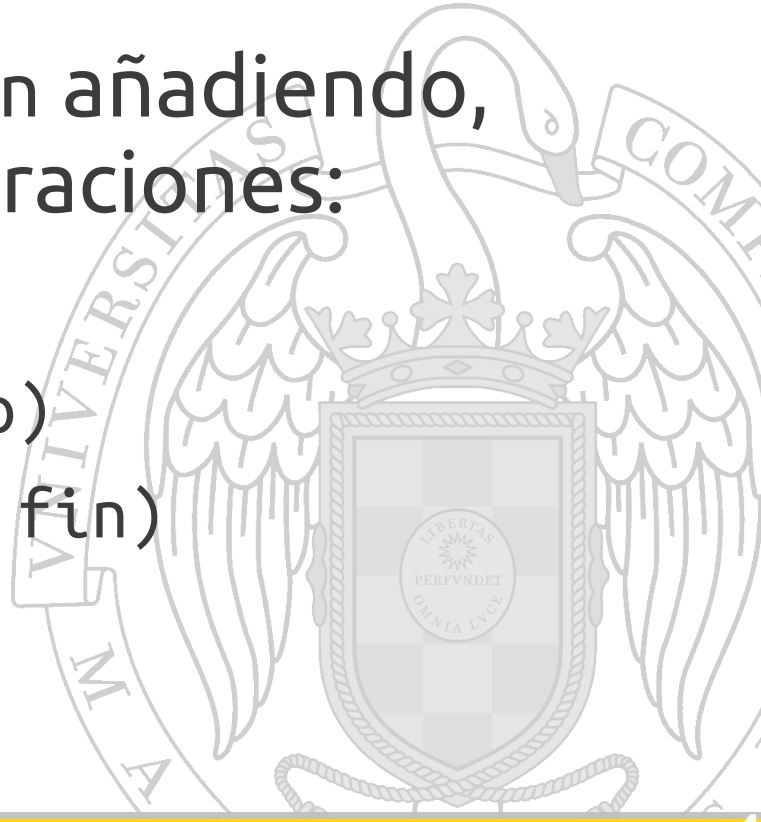
Interfaz Collection

- Especifica funciones para manejar grupos de objetos, conocidos como elementos.
  - `boolean add(Object o)`
  - `boolean addAll(Collection c)`
  - `void clear()`
  - `boolean contains(Object o)`
  - `boolean isEmpty()`
  - `boolean remove(Object o)`
  - `int size()`
  - `Iterator iterator()`
  - `Object[] toArray()`

Interfaz *Collection*

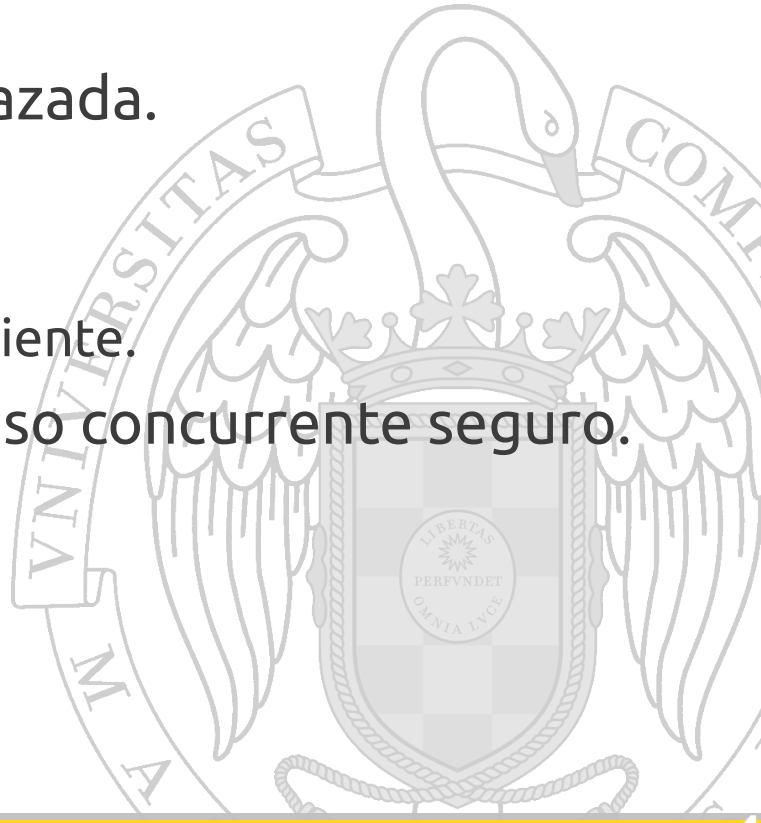
Interfaz *List*

- Especifica las operaciones de una colección ordenada.
  - Proporciona control sobre dónde insertar los elementos.
- Extiende la interfaz `Collection` añadiendo, entre otras, las siguientes operaciones:
  - `Object get(int indice)`
  - `void set(int indice, Object o)`
  - `List subList(int inicio, int fin)`



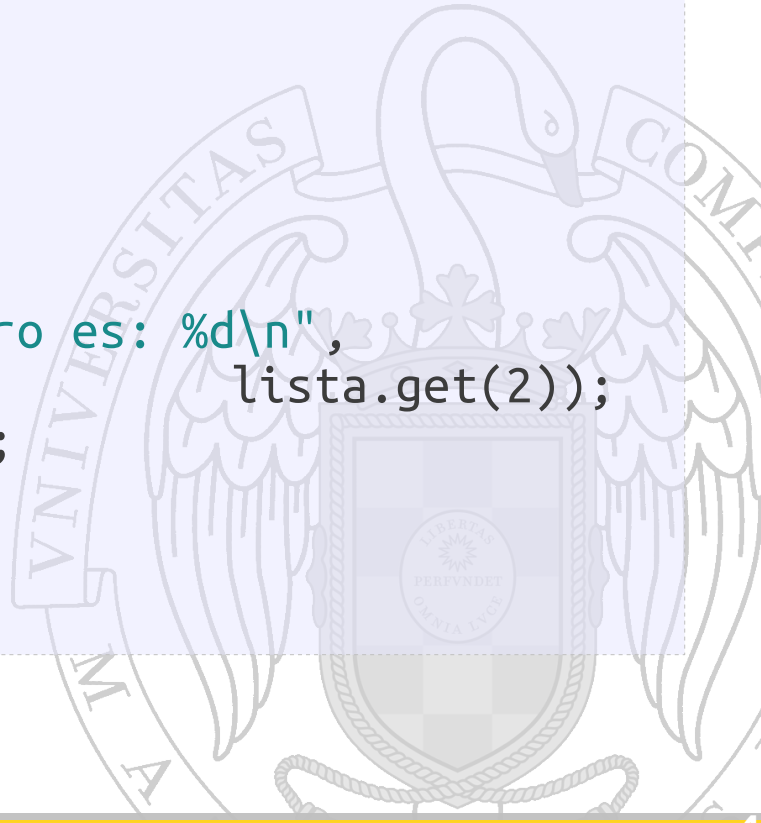
Implementaciones de *List*

- `ArrayList`: Implementación como array.
  Cuando el array se llena, se reserva otro más grande y se mueven los elementos del array antiguo al array nuevo.
  - Ventaja: Acceso a elementos i-ésimos eficiente.
  - Desventaja: Inserción y borrado ineficientes, cuando se realizan en medio del array.
- `LinkedList`: Implementación como lista enlazada.
  Enfoque tradicional.
  - Ventaja: Inserción y borrado eficientes.
  - Desventaja: Acceso a elementos i-ésimos ineficiente.
- `Vector`: Parecido a `ArrayList`, pero con acceso concurrente seguro.
  - Ventaja: Acceso concurrente.
  - Descentaja: Más ineficiente que ArrayList.



Ejemplo con listas

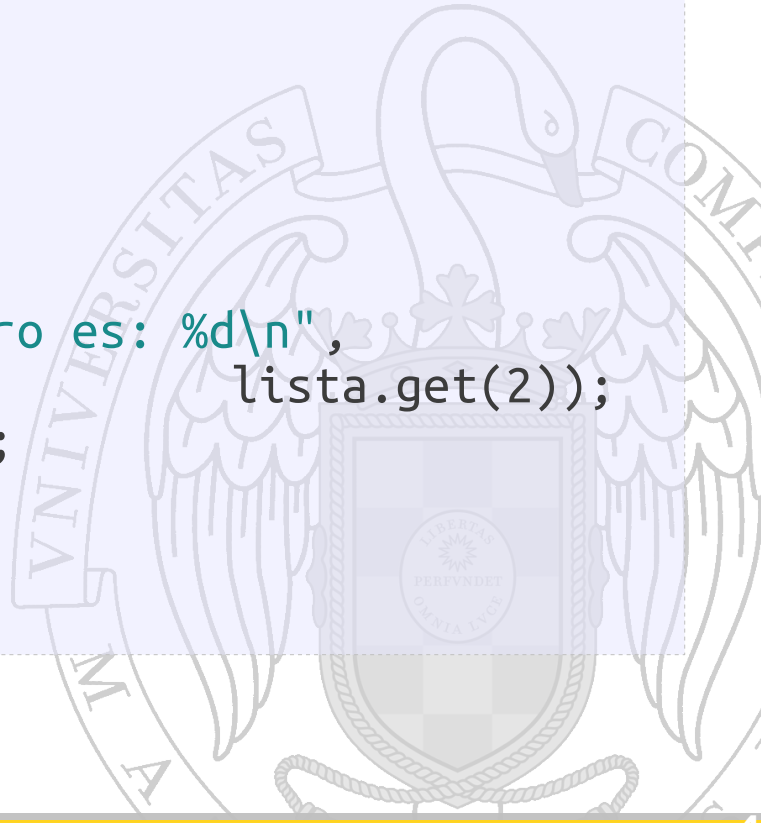
```
import java.util.ArrayList;


public class ArrayListTest
{
    public static void main(String[] args) {
        ArrayList lista = new ArrayList();
        lista.add(new Integer(3));
        lista.add(new Integer(20));
        lista.add(new Integer(10));
        lista.add(new Integer(15));
        System.out.printf("El elemento tercero es: %d\n",
                                                lista.get(2));
        System.out.println(lista.toString());
    }
}
```



Ejemplo con listas

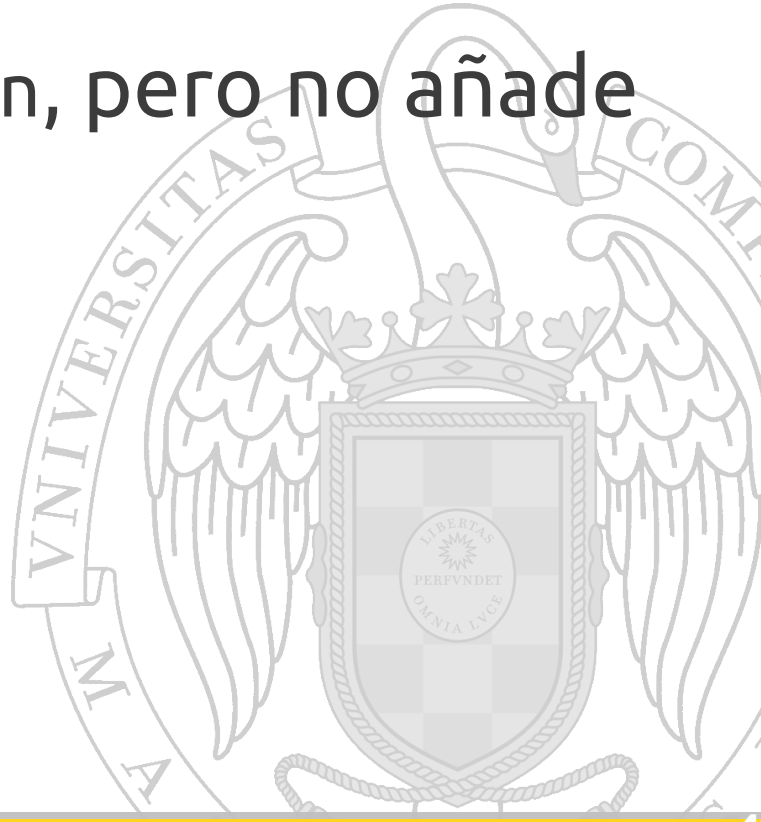
```
import java.util.ArrayList;


public class ArrayListTest
{
    public static void main(String[] args) {
        ArrayList lista = new ArrayList();
        lista.add(3);
        lista.add(20);
        lista.add(10);
        lista.add(15);
        System.out.printf("El elemento tercero es: %d\n",
                                                lista.get(2));

        System.out.println(lista.toString());
    }
}
```

Boxing



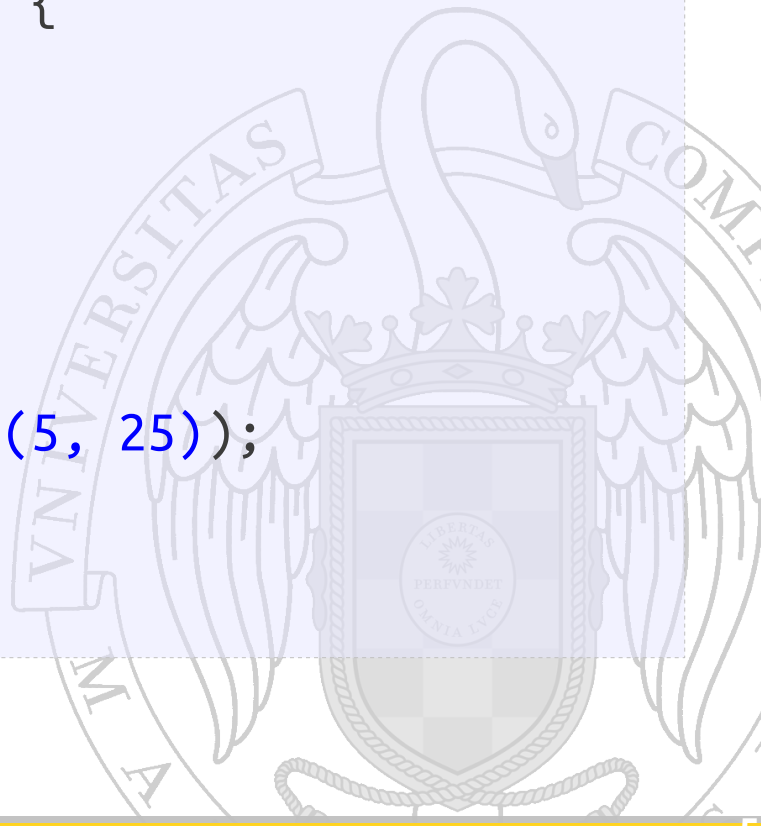
Interfaz *Set*

- Especifica las operaciones de una colección en la que no puede haber duplicados.
  - Los elementos no guardan ningún orden en particular.
- Extiende la interfaz `Collection`, pero no añade ninguna operación nueva.



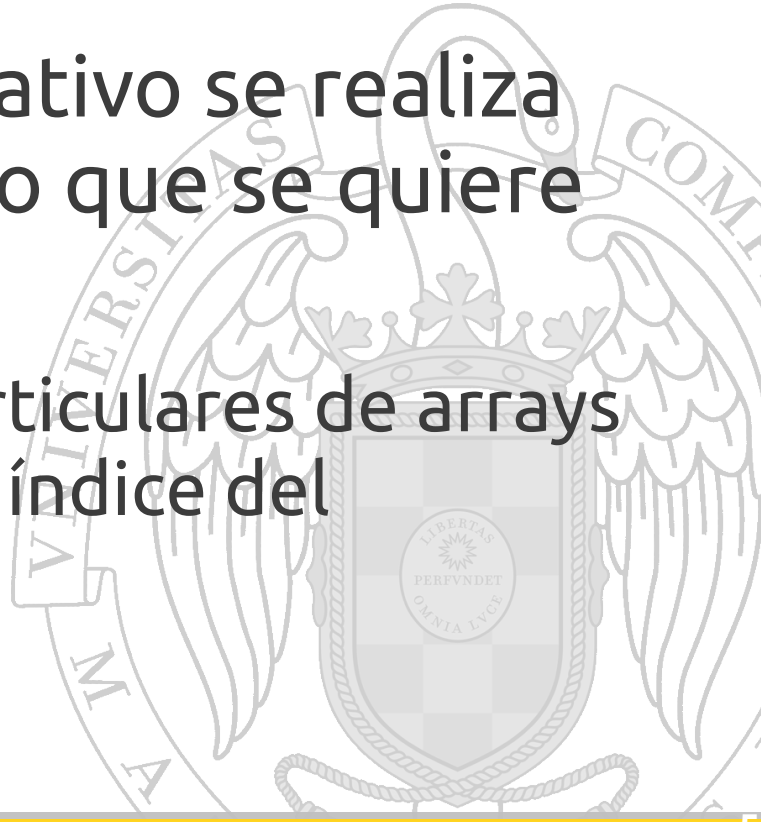
Ejemplo con conjuntos

```
import java.util.TreeSet;


public class TreeSetTest
{
    public static void main(String[] args) {
        TreeSet conjunto = new TreeSet();
        conjunto.add(3);
        conjunto.add(15);
        conjunto.add(1);
        conjunto.add(20);
        conjunto.add(30);
        System.out.println(conjunto.subSet(5, 25));
    }
}
```



Interfaz *Map*

- Especifica un objeto que asocia claves con valores (array asociativo)
  - Ejemplo: Base de datos que asocia DNIs con datos de personas. Clave: `int`, Valor: `Persona`.
- La búsqueda en un array asociativo se realiza mediante la clave del elemento que se quiere buscar.
  - Los arrays de Java son casos particulares de arrays asociativos, donde la clave es el índice del elemento dentro del array.



Interfaz *Map*

- Métodos de la interfaz:
  - `void clear()`
  - `boolean containsKey(Object clave)`
  - `Set entrySet()`
  - `Object get(Object clave)`
  - `boolean isEmpty()`
  - `Set keySet()`
  - `void put(Object clave, Object valor)`
  - `Object remove(Object clave)`
  - …

Implementaciones de *Map*

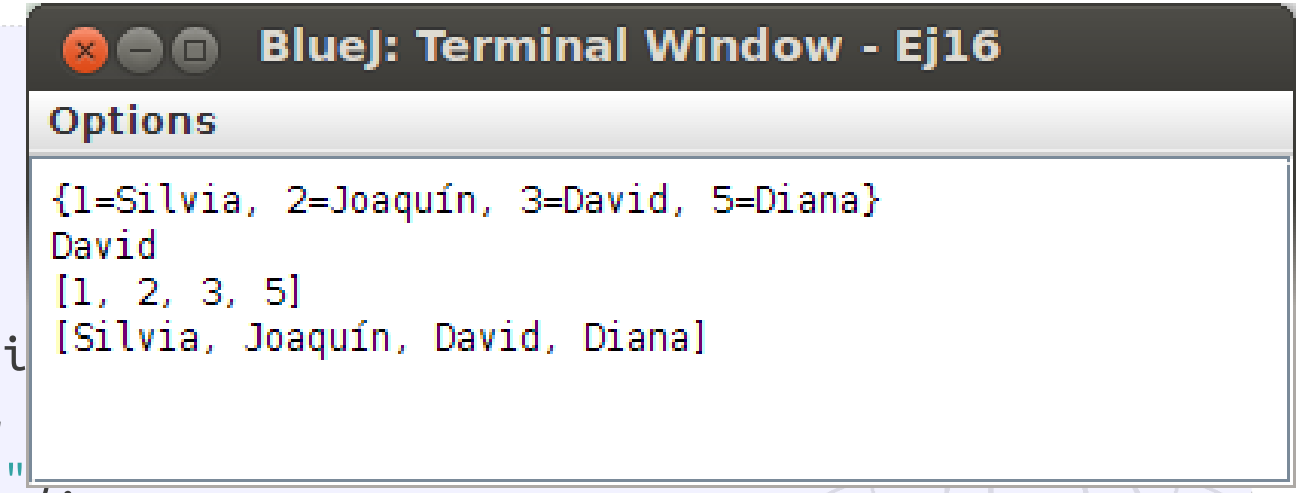
- Son análogas a las implementaciones de la interfaz Set:
- `TreeMap`
  Guarda los pares (clave, valor) ordenados por clave. Requiere una operación de comparación entre claves.
- `HashMap`
  Guarda los pares (clave, valor) en una tabla *hash* utilizando el método `hashCode()` de la clase de la clave.

Ejemplo con conjuntos

```
import java.util.TreeMap;


public class TreeMapTest
{
    public static void main(String[] args) {
        TreeMap map = new TreeMap();
        map.put(3, "David");
        map.put(1, "Silvia");
        map.put(2, "Joaquín");
        map.put(5, "Diana");
        System.out.println(map);
        System.out.println(map.get(3));
        System.out.println(map.keySet());
        System.out.println(map.values());
    }
}
```

Ejemplo con conjuntos

```
import java.util.TreeMap;


public class TreeMapTest
{
    public static void mai
        TreeMap map = new
        map.put(3, "David");
        map.put(1, "Silvia");
        map.put(2, "Joaquín");
        map.put(5, "Diana");
        System.out.println(map);
        System.out.println(map.get(3));
        System.out.println(map.keySet());
        System.out.println(map.values());
    }
}
```



Iterar sobre una colección

- Las subclases de `Collection` proporcionan métodos para recorrer todos los elementos de la colección correspondiente.
- El recorrido se realiza mediante un iterador.
- Conceptualmente, un iterador es como un puntero que señala un determinado elemento de la lista y proporciona tres operaciones:
  - Devolver el elemento apuntado por el iterador.
  - Borrar el elemento apuntado por el iterador.
  - Mover el iterador al siguiente elemento.

La interfaz *Iterator*

```
public interface Iterator {
  public boolean hasNext();
  public Object next();
  public void remove();
}
```

```
public void imprimirElementos(Collection c) {
  // Imprime los elementos de una colección,
  // uno por cada línea.

  Iterator it = c.iterator();
  while (it.hasNext())
    System.out.println(it.next());
}
```

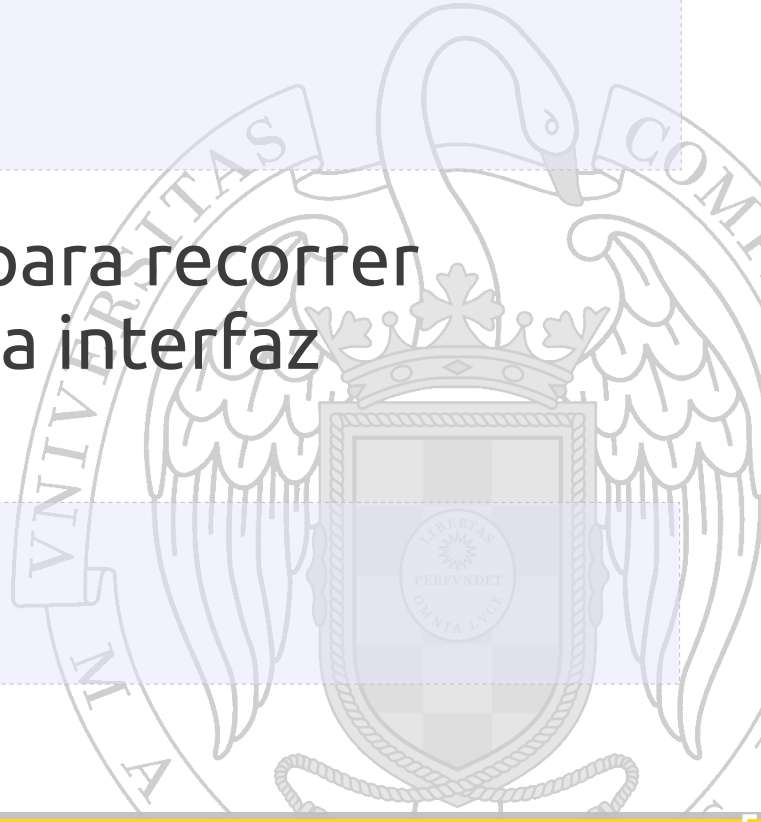
Bucles sobre colecciones

- Desde la versión 5 de Java, se pueden utilizar bucles `for` para iterar sobre los elementos de una colección.

```
public void imprimirElementos(Collection c) {
  // Imprime los elementos de una colección,
  // uno por cada línea.

  for(Object e : c)
    System.out.println(e);
}
```

- En general, esta sintaxis es válida para recorrer cualquier objeto que implemente la interfaz `Iterable`.

```
public interface Iterable {
  public Iterator iterator();
}
```



Contenidos

- Clases envoltorio: `Integer`, `Character`, ...
- `String` y `StringBuffer`
- Números aleatorios
- Fechas y horas
- Expresiones regulares
- Utilidades con arrays
- *Java Collections Framework*
- Clases genéricas

Clases genéricas

- En versiones anteriores de Java, las colecciones trabajaban con elementos que eran de tipo `Object`.
  - Ventaja: Genericidad.
  - Desventaja: Los métodos para acceder a los elementos de la colección devuelven un objeto de tipo Object → es necesaria una conversión (*downcasting*) al tipo que nos interese.

```
Perro p = new Perro();
ArrayList l = new ArrayList();
l.add(p);
p = (Perro) l.get(0);
```

Clases genéricas

- La versión 5 de Java introduce los genéricos, que permiten construir clases paramétricas con respecto a un tipo.
- Todas las clases de la *Java Collection Framework* son paraḿetricas con respecto al tipo **T** de elementos que se almacenan.

```
ArrayList<Perro> l = new ArrayList<Perro>();
l.add(new Perro());
```

- Los métodos que acceden a los elementos de las colecciones devuelven valores del tipo **T**.

```
Perro p = l.get(0);
```

Interfaz `Collection<E>`

- Es genérica con respecto al tipo de elementos que almacena.
  - `boolean add(E o)`
  - `boolean addAll(Collection<? extends E> c)`
  - `Iterator<E> iterator()`
  - `E[] toArray()`
  - …
- Interfaz `Iterator<E>`
  - `E next()`
  - …

Interfaz `List<E>`

- Extiende la interfaz `Collection<E>` añadiendo, entre otras, las siguientes operaciones:
  - `E get(int indice)`
  - `void set(int indice, E o)`
  - `List<E> subList(int inicio, int fin)`
  - …
- Implementaciones de `List<E>`
  - `ArrayList<E>`
  - `LinkedList<E>`

Interfaz Map<K,V>

- Métodos de la interfaz:
  - `Set<Map.Entry<K,V>> entrySet()`
  - `V get(K clave)`
  - `Set<K> keySet()`
  - `void put(K clave, V valor)`
  - …
- Implementaciones de `Map<K,V>`
  - `HashMap<K,V>`
  - `TreeMap<K,V>`

Ejemplo con conjuntos genéricos

```
import java.util.TreeMap;


public class TreeMapTest
{
    public static void main(String[] args) {
        TreeMap<Integer,String> map = new TreeMap<Integer,String>();
        map.put(3, "David");
        map.put(1, "Silvia");
        map.put(2, "Joaquín");
        map.put(5, "Diana");
        System.out.println(map);
        System.out.println(map.get(3));
        System.out.println(map.keySet());
        System.out.println(map.values());
    }
}
```

Referencias

- B. Eckel
  Thinking in Java (3rd Edition)
  Cap. 11 (no incluye *Generics*!)
- P. Deitel, H. Deitel
  Java. How to Program (9th Edition)
  Cap. 20, 21, 22
- M. Naftalin and P. Wadler
  Java generics and collections
- Documentación de librerías de Java
  http://docs.oracle.com/javase/6/docs/api/