

# UT5

## Colecciones de tamaño fijo: arrays. Librería de clases: la clase String.

(2ª presentación)

Módulo – Programación (1º)

Ciclos – Desarrollo de Aplicaciones Multiplataforma | Desarrollo de Aplicaciones Web

CI María Ana Sanz

---

# Contenidos

- Arrays en Java
  - declarar un array
  - crear un array
  - dar valores a un array
  - arrays como parámetros
  - arrays como valor de retorno
  - arrays como atributos
  - arrays de objetos
  - arrays bidimensionales
- Operaciones con arrays
  - búsqueda en arrays
  - ordenación de arrays
- La API de Java
  - documentación de una clase
- La clase String
- La clase StringBuilder
- Signatura del método main()
- La clase Scanner como tokenizer
- El tipo enum

# Uso de la API de Java

- **API de Java (Application Programmer's Interface)**
  - clases proporcionadas por Java para poder incluir en nuestros proyectos
- **Ejer 5.21**
  - Localiza la clase Random
  - Busca el método **round()** en la clase Math
  - Busca el método **arraycopy()** de la clase System
  - Busca el método **compareTo()** de la clase String

# Interface versus implementación

- **Interface** de una clase
  - muestra su parte pública, lo que la clase puede hacer (servicios que ofrece)
  - proporciona información sobre cómo utilizar la clase
  - no muestra implementación (código)
  - parte visible
- **Implementación** de la clase
  - código fuente
  - un programador trabaja en la implementación de una clase y al mismo tiempo hace uso de otras clases vía sus interfaces.

# Interface versus implementación

- La **documentación** de una clase incluye:
  - el nombre de la clase
  - una descripción general de la clase
  - la lista de los constructores y métodos de la clase
  - los valores de retorno y parámetros para los constructores y métodos
  - una descripción del propósito de cada constructor y método
- Toda esta información constituye la interface de la clase.
- interfaz puede aplicarse también a un método individual.
  - La interfaz de un método consiste en su signature
    - `public int maximo(int x, int y)`

# Modificadores de acceso: public / private

- Definen la visibilidad de
  - un atributo, constructor, método e incluso una clase
- Método **public**
  - puede ser invocado desde dentro de la clase donde está definido y fuera de ella
- Método **private**
  - sólo puede ser llamado dentro de la clase en la que se ha declarado.

# Modificadores de acceso: public / private

- **Ocultar la información** – el principio de ocultamiento de la información
  - asegura la modularización de la aplicación,
    - una clase así no depende de cómo esté implementada otra, lo que conduce a un bajo acoplamiento entre las clases y, por tanto, a un más fácil mantenimiento.
  - los objetos exhiben responsabilidades (*qué*) pero ocultan la implementación (*cómo*)
- Métodos privados
  - descomponer una tarea en tareas más pequeñas
    - facilitar la implementación
    - evitar repetir código

principios de un buen  
diseño OO

# Modificadores de acceso: *public* / *private*

---

- Atributos
  - siempre privados
    - un atributo público rompe el principio de ocultamiento
  - acceder a ellos a través de los accesorios y mutadores



# Paquetes y sentencia import

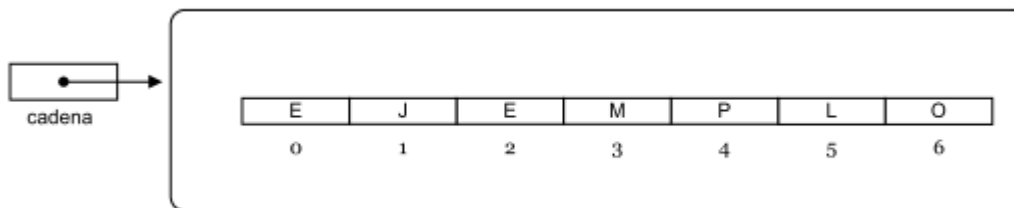
- Las clases de la API de Java no están automáticamente disponibles
  - Hay que indicar en el código fuente que queremos utilizar una clase de la librería.
  - A esto se llama **importar la clase** y se realiza con la sentencia **import**
    - `import java.util.Random;` ◀ **mejor esta**
    - `import java.util.*;` Todas las clases de este paquete
    - se incluye al principio del código fuente de una clase
  - Java utiliza **paquetes** para organizar en grupos la multitud de clases de su librería
  - Un **paquete**
    - conjunto de clases lógicamente relacionadas. Los paquetes pueden **anidarse** (un paquete puede contener otro paquete).

# Paquetes y sentencia import (cont..)

- la clase Random está dentro del paquete java.util
  - El nombre completo o nombre calificado de la clase es el nombre del paquete que la contiene seguido por un punto y el nombre de la clase: **java.util.Random**
- Si no especificamos import
  - `private java.util.Scanner teclado` `= new java.util.Scanner(System.in);`
- paquete **java.lang**
  - se importa **automáticamente** en cada clase.
  - la clase **String** está en el paquete java.lang.
- A partir de la versión Java 1.5 se puede **importar elementos static de una clase**.
  - `import static java.lang.Math.random;`

# La clase String

- Un string es una secuencia de caracteres.
- En Java **un string es un objeto**.
- La clase String
  - en java.lang
  - modela una secuencia de caracteres y
  - proporciona múltiples constructores y métodos para tratar de forma individual los caracteres,
    - comparar cadenas, buscar subcadenas, extraer subcadenas, ....
  - el valor de un string se escribe entre "".



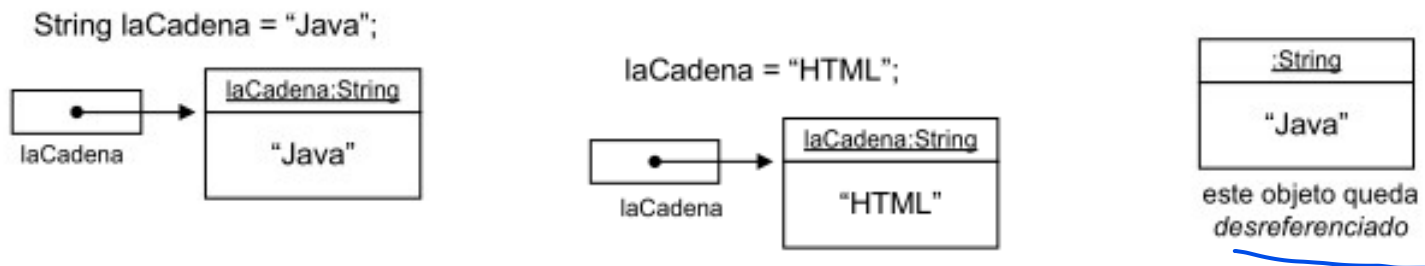
String
– value
– count
+ String()
+ String(in s : String)
+ length()
+ charAt()
+ indexOf()
+ valueOf(in n : int) : String
+ valueOf(in d : double) : String
+ charAt(in n : int) : char
+ equals(in o : Object) : boolean
+ indexOf(in ch : int) : int
+ indexOf(in ch : int, in start : int) : int
+ indexOf(in s : String) : int
+ indexOf(in s : String, in start : int) : int
+ substring(in strt : int) : String
+ substring(in strt : int, in end : int) : String

# Creando un String

- Llamar al constructor de la clase
  - `String cadena = new String();`     `//crea una cadena vacía`
  - `String cadena = new String("Ejemplo de cadena");`
- **Forma abreviada** de crear e inicializar un String
  - `String cadena = "Bienvenido a Java";`
- Si únicamente hacemos
  - `String cadena;`
    - se crea una referencia que vale *null*.

# Inmutabilidad de String

- Un objeto String es **inmutable**.
  - Una vez **creado su contenido no puede cambiar**, contendrá esa cadena toda su vida.
  - Si se asigna un nuevo valor a la cadena **Java crea un nuevo objeto y descarta el anterior**
  - `String laCadena = "Java";`  
`laCadena = "HTML";`



# Longitud de una cadena y acceso a caracteres individuales

- Longitud de una cadena – **int length()**
  - String cadena = "Hola";
  - int longitud = cadena.length(); // longitud = 4
- Acceso a los caracteres de un String - **char charAt(int posicion)**
  - String cadena = new String("Hola qué tal");  
char carac = cadena.charAt(5); // carac = 'q'  
int indice = 6;  
System.out.println(cadena.charAt(indice + 3)); //escribe t

Método accesor,  
no es una  
atributo público

El primer carácter en un String  
está en la posición 0.  
El último en la posición  
**cadena.length() - 1**

Indexes

0	1	2	3	4	5	6	7
↓	↓	↓	↓	↓	↓	↓	↓
S	o	c	r	a	t	e	s

# Concatenación de cadenas

- **String concat(String cadena)**

- String str1 = "Bienvenido ";  
String str2 = "al lenguaje Java ";  
String str3 = str1.concat(str2); // str3 es "Bienvenido al lenguaje  
// Java"

- **operador +** para concatenar

- String str1 = "Hola";  
String str2 = " tal";  
String resul = str1 + " que" + str2;

Java proporciona un mecanismo  
mas cómodo a través del operador  
+

# Concatenación de cadenas (cont ...)

- **operador +**
  - también concatena un String combinado con un tipo primitivo
    - un int, double o char.
  - El valor del tipo primitivo se convierte automáticamente a String y luego se concatena.
    - `int valor = 3;`  
`String str = "Resultado = " + valor;`  
`System.out.println(str);`



# Extrayendo subcadenas

- **String substring(int posInicial, int posFinal)**
  - devuelve una subcadena (es un string) que comienza en posInicial y llega hasta posFinal (sin incluir esta posición final)
- **String substring(int posInicial)**
  - devuelve una subcadena (es un string) desde posInicial hasta el final de la cadena
- ```
String mensaje = "Hola";  
String str1 = mensaje.substring(1); //str1 vale "ola"  
String str2 = mensaje.substring(1, 3); // str2 vale "ol"
```

# Comparación de cadenas

- **boolean equals(Object obj)**

- permite comparar dos objetos cualesquiera, también objetos String

- true si son iguales, false en otro caso

- String mensaje = “Hola”;
  - if (mensaje.equals(“Hola”))

true

Se compara el valor, no la referencia. Para eso usaremos ==

- **boolean equalsIgnoreCase(String s)**

- comparar dos objetos String ignorando mayúsculas / minúsculas

- **int compareTo(String s)**

- if (str1.compareTo(str2) == 0) ..... devuelve
  - 0 si las dos cadenas que se comparan son iguales
  - < 0 si str1 es menor que str2
  - > 0 si str1 mayor que str2

$\text{str1.compareTo(str2)} = -1 * \text{str2.compareTo(str1)}$

# Comparación de cadenas (cont...)

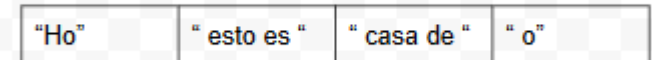
- Al comparar dos cadenas
  - comparación lexicográfica y sensible a mayúsculas (la comparación se hace carácter a carácter, por ejemplo, “abc” es menor que “abg” y teniendo en cuenta la longitud, por ejemplo, “aba” es menor que “abaaa”)
- **!!ATENCIÓN!!**
  - Nunca hay que utilizar el operador `==` para comparar dos cadenas
  - `if (str1 == str2) ....` compara las referencias `str1` y `str2`, se comprueba si apuntan al mismo objeto, no si se trata de la misma secuencia de caracteres.
    - El operador `==` sirve para comparar tipos primitivos y no tipos objeto.

# Búsqueda de cadenas


- **int indexOf(String s)**
- **int indexOf(char c) / int indexOf(int c)**
  - devuelve la posición de la primera ocurrencia de s o -1 si no se encuentra
    - String mensaje = “Hola”;
    - int pos = mensaje.indexOf(“la”); // devuelve 2
    - pos = mensaje.indexOf(“pa”); // devuelve -1
    - pos = mensaje.indexOf(‘o’); // devuelve 1
- **int indexOf(String s, int from)**
  - devuelve la posición de la primera ocurrencia de s a partir de *from* o -1 si no se encuentra
    - String mensaje = “Hola esto es un ejemplo”;
    - int pos = mensaje.indexOf(“es”, 7) // devuelve 10
    - pos = mensaje.indexOf(“la”, 5) // devuelve -1

# Otros métodos de String - split()

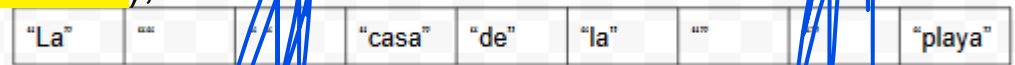
- **String[] split(String delimitador) /**
- **String[] split(String expresión\_regular)**
  - trocea (parsea) una cadena según un delimitador devolviendo un array con los trozos (*tokens*) obtenidos
    - String mensaje = “Hola esto es la casa de la ola”;
    - String[] tokens = mensaje.split(“la”);
  - String lineaDatos = “Ana:23:1.80:3000”;
  - String[] datos = lineaDatos.split(“:”);
- final String ESPACIO = “ ”;
- String frase = “la casa de la playa”;
- String[] palabras = frase.split(ESPACIO);



|      |             |             |      |
|------|-------------|-------------|------|
| “Ho” | “ esto es ” | “ casa de ” | “ o” |
|------|-------------|-------------|------|



|       |      |         |        |
|-------|------|---------|--------|
| “Ana” | “23” | “1.80 ” | “3000” |
|-------|------|---------|--------|

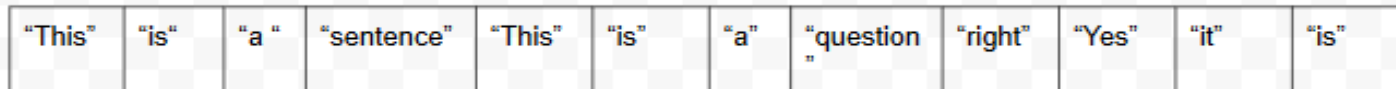


|      |     |     |        |      |      |     |     |         |
|------|-----|-----|--------|------|------|-----|-----|---------|
| “La” | “ ” | “ ” | “casa” | “de” | “la” | “ ” | “ ” | “playa” |
|------|-----|-----|--------|------|------|-----|-----|---------|

# Otros métodos de String - split()

- **String[] split(String expresión\_regular)**

- en realidad el **delimitador** es una **expresión regular**
  - **patrón codificado utilizado para buscar coincidencias en una cadena de texto** (cumple o no el patrón la cadena) e incluso realizar transformaciones en ella
  - String str = "This is a sentence. This is a question, right? Yes! It is.";
  - String delimitadores = "[.,?!]+";
  - String[] tokens = str.split(delimitadores);



|        |      |      |            |        |      |      |            |         |       |      |      |
|--------|------|------|------------|--------|------|------|------------|---------|-------|------|------|
| "This" | "is" | "a " | "sentence" | "This" | "is" | "a " | "question" | "right" | "Yes" | "it" | "is" |
|--------|------|------|------------|--------|------|------|------------|---------|-------|------|------|

- String frase = "la.casa.de.la.playa";
- String[] palabras = frase.split("\\.");

# Ejer 5.22

- Ejer 5.22

de la clase String (static),  
no necesita instancias de  
dicha clase



|                    |                          |                |
|--------------------|--------------------------|----------------|
| toLowerCase()      | indexOf()                | isEmpty()      |
| toUpperCase()      | contains()               | valueOf()      |
| startsWith()       | compareToIgnoreCase()    | replaceFirst() |
| endsWith()         | replaceAll()             | lastIndexOf()  |
| equalsIgnoreCase() | split()                  |                |
| trim()             | format() -<br>(estático) |                |

## Ejer 5.23

- Ejer 5.23

- a) `String unaCadena = new String("Ejemplo");`

- `unaCadena = unaCadena.toUpperCase();`

- a.1) ¿quién es el receptor del mensaje `toUpperCase()`?

- `unaCadena`

- a.2) ¿sería correcto hacer: `unaCadena.toUpperCase()` ?

- `no, la nueva cadena con mayúsculas se perdería`

- b) `String str = "Aprendiendo cadenas en Java";`

- `String resul;`

- `String otra = "cadena de ejemPLO";`

- `char character;`

- `int pos;`



## Ejer 5.23

- Ejer 5.23

a) String str = "Aprendiendo cadenas en Java";  
String resul;  
String otra = "cadena de ejemPLO";  
char character;  
int pos;

1. la cadena str convertida a mayúsculas  
`resul = str.toUpperCase();`
2. el carácter de la posición 6 de str  
`character = str.charAt(6);`
3. el último carácter de la cadena str  
`carácter = str.charAt(str.length() - 1);`

## Ejer 5.23

- Ejer 5.23

4. compara str con la cadena otra sin tener en cuenta mayúsculas ni minúsculas

`if (str.compareToIgnoreCase(otra)) ....`

5. pregunta si str empieza por "Ba"

`if (str.startsWith("Ba")) ...`

6. sustituye en str todas las 'e' por '\*'

`str = str.replace("e", "*");`

7. devuelve la primera aparición de la 'd' en str

`pos = str.indexOf("d");`

## Ejer 5.23

### ■ Ejer 5.23

8. localiza la última aparición de la 'c' en str. Extrae , a partir de ahí, la subcadena existente hasta el final.

```
pos = str.lastIndexOf('c');  
String otra = str.substring(pos);
```

9. convierte a String el valor 66

```
String resul = String.valueOf(66);
```

10. pregunta si la cadena str es vacía (Hazlo de varias formas)

```
if (str.isEmpty()) ..  
if (str.equals("")) ...  
if (str.length() == 0) ...
```

## Ejer 5.24

- Ejer 5.24
  - `public int aparicionesDe(String str)`
    - cuántas veces aparece la cadena `str` en `this.palabra`
    - si `palabra = "esto es un ejemplo de esas cosas"`, `aparicionesDe("es")` devolvería 3
  - `public String insertarGuion()`
    - si `palabra="prueba"` el método devolverá "p-r-u-e-b-a"

## Ejer 5.24

```
public int aparicionesDe(String str)
{
    String origen = this.palabra.toLowerCase();
    int lenBuscada = str.length();
    String strBuscada = str.toLowerCase();
    int cuantas = 0;
    int pos = origen.indexOf(strBuscada);
    while (pos != -1) {
        cuantas++;
        int inicio = pos + lenBuscada;
        origen = origen.substring(inicio);
        pos = origen.indexOf(strBuscada);
    }

    return cuantas;
}
```

## Ejer 5.24

```
public String insertarGuion()
{
    String nueva = "";
    int i = 0;
    while (i < this.palabra.length() - 1) {
        nueva = nueva + this.palabra.charAt(i) + "-";
        i++;
    }
    return nueva + this.palabra.charAt(this.palabra.length() - 1);
}
```

# Ejercicios

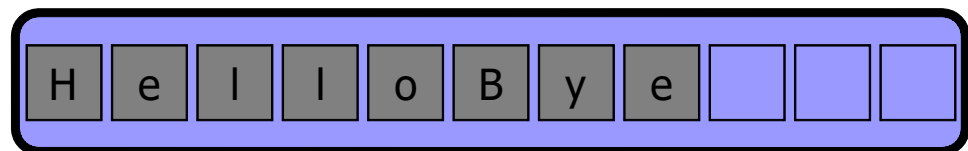
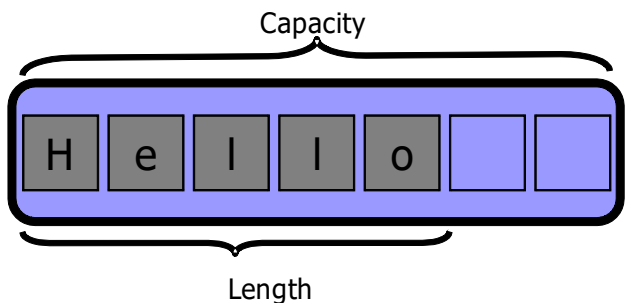
---

Ejer 5.25  
Ejer 5.26

# La clase StringBuilder

- Desventajas de String
  - los objetos String son inmutables
    - cada vez que se concatena, por ejemplo, se genera un nuevo objeto ¡POCA EFICIENCIA!
    - en situaciones con muchas concatenaciones es mejor utilizar StringBuilder
- **StringBuilder**
  - modela cadenas que pueden ser modificadas
  - en java.lang
  - como StringBuffer pero no sincronizada (más eficiente)

Cada vez que llamamos a `alumno[i].toString()` se crea un nuevo objeto





# Cuándo utilizar StringBuilder?

- se utiliza como clase de apoyo
- en **toString()** si hay muchas concatenaciones
  - en el método toString() de una clase que encapsula un array
- si se van a hacer muchas operaciones de inserción de caracteres / borrado en una cadena

```
public String toString() {  
    StringBuilder sb = new StringBuilder();  
    for (int i = 0; i < pos; i++) {  
        sb.append(listaPalabras[i].toString()).append("\t");  
    }  
    return sb.toString();  
}
```

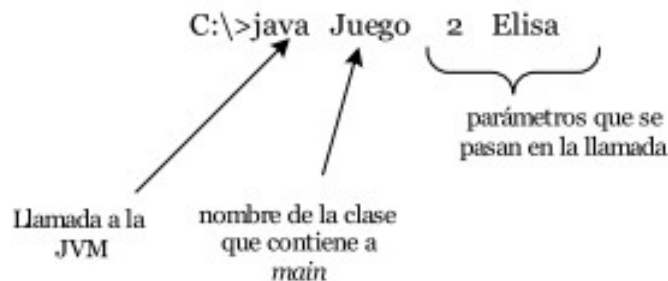
# Métodos de StringBuilder

|                                                     |                                                           |
|-----------------------------------------------------|-----------------------------------------------------------|
| <code>new StringBuilder()</code>                    | <code>StringBuilder deleteCharAt(int indice)</code>       |
| <code>new StringBuilder(String s)</code>            | <code>StringBuilder reverse()</code>                      |
| <code>char charAt(int indice)</code>                | <code>StringBuilder insert(int posicion, String s)</code> |
| <code>int length()</code>                           | <code>StringBuilder delete(int inicio, int final)</code>  |
| <code>void setCharAt(int indice, char c)</code>     | <code>StringBuilder append(String s)</code>               |
| <code>StringBuilder deleteCharAt(int indice)</code> | <code>String toString()</code>                            |

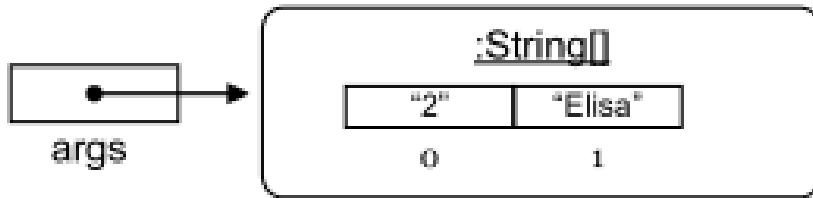
# Signatura del método main() - argumentos en la línea de comandos

- Método **main()**
  - iniciar la ejecución de una aplicación desde fuera del entorno
  - estático
  - *C:.....>java AppCalculadora*
- Signatura de main()
  - **public static void main(String[ ] args)**

para pasar  
argumentos  
desde la línea de  
comandos



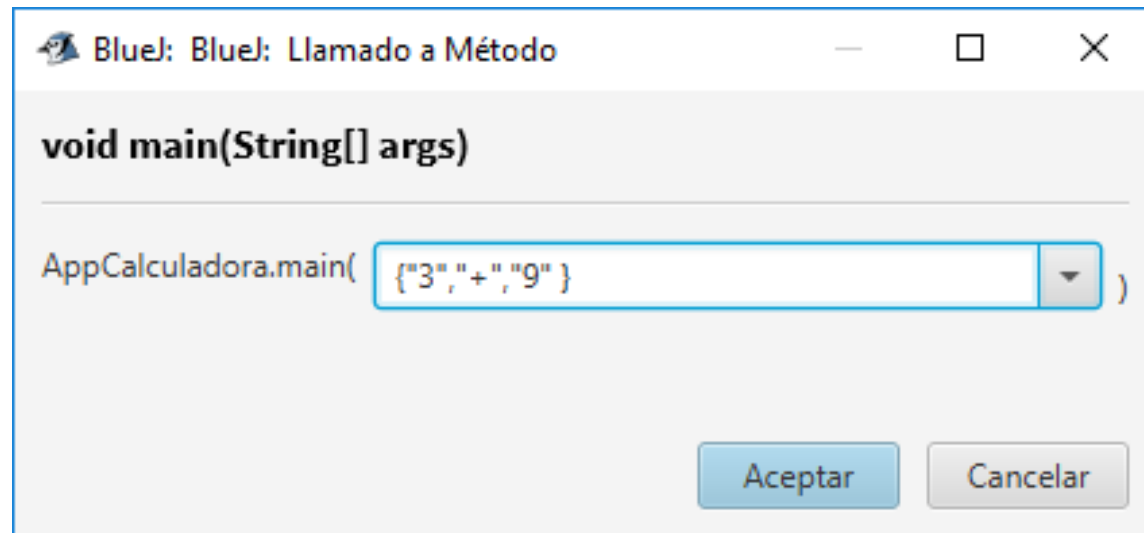
# Signatura del método main() - argumentos en la línea de comandos



```
public class PruebaMain
{
    public static void main(String[] args) {
        if (args.length == 2)
        {
            int edad = Integer.parseInt(args[0]);
            String nombre = args[1];
            Persona p = new Persona(nombre, edad);
        }
        else
        {
            .....
        }
    }
}
```

# Signatura del método main() - argumentos en la línea de comandos

- Desde BlueJ



# La clase Scanner como 'tokenizer'

- partir una cadena según un criterio de separación
  - **pepe:23:Pamplona:Navarra**      4 tokens (el : es el separador)
- Cómo extraer tokens?
  - método `split()` de la clase `String`
  - clase `Scanner`
- Clase `Scanner`
  - para separar *tokens* de un `String`
  - si queremos que el delimitador sea una palabra completa hay que utilizar esta clase

# La clase Scanner como 'tokenizer'

`new Scanner(String s)`

`boolean hasNext()`

`String next()`

`boolean hasNextInt()`

`String nextLine()`

`int nextInt()`

`useDelimiter(String s)`

# La clase Scanner como 'tokenizer'

```
public void ejemplo01Scanner()
{
    String s = "Ejemplo de Java";
    Scanner sc = new Scanner(s);
    sc = sc.useDelimiter("de");
    while (sc.hasNext()) {
        System.out.println(sc.next());
    }
}
```

```
public void ejemplo02Scanner()
{
    String s = "1 2 3 4 5 6 7";
    Scanner sc = new Scanner(s); //por defecto el
                                //delimitador es el espacio

    int suma = 0;
    while (sc.hasNextInt()) {
        suma += sc.nextInt();
    }
    System.out.println("La suma es " + suma);
}
```



# El tipo enum

- Tipo enumerado (introducido en Java 1.5)
- Crea un tipo enumerando cada uno de sus valores

no hay garantía de que el parámetro nivel sea un valor correcto

Lo que hemos hecho hasta ahora

```
public class Control
{
    private static final int BAJO = 0;
    private static final int MEDIO = 1;
    private static final int ALTO = 2;
    private int nivel;
    /*
     * Inicializa nivel a BAJO
     */
    public Control()
    {
        nivel = BAJO;
    }
    public void setNivel(int nivel)
    {
        this.nivel = nivel;
    }
}
```

# El tipo enum

```
public class Control
{
    private Nivel nivel;
    /*
     * Inicializa nivel a BAJO
     */
    public Control()
    {
        nivel = Nivel.BAJO;
    }
    /*
     * devuelve el valor actual de nivel
     */
    public Nivel getNivel()
    {
        return nivel;
    }
}
```

```
/**
 * Enumera el conjunto de valores
 * disponibles para un objeto
 * Control
 */
public enum Nivel
{
    BAJO, MEDIO, ALTO;
}
```

```
/*
 * establece el nivel actual
 */
public void setNivel(Nivel nivel)
{
    this.nivel = nivel;
}
```

# Consideraciones sobre el tipo enum

- son clases
- no son enteros
- no tienen constructor público
- implícitamente los valores del tipo son *public, static, final*
- incluyen método **ordinal()** -
  - valor ordinal (posición en la lista) asociado al enumerado

# Consideraciones sobre el tipo enum

- se comparan con `==` o con `equals()`
- se pueden comparar en términos de orden con `compareTo()`
- heredan el método `toString()`
  - `Nivel.MEDIO.toString()` devuelve la cadena “MEDIO”.
  - Este método se puede redefinir dentro del tipo enumerado

```
Nivel nivel = Nivel.MEDIO;  
System.out.println(nivel.toString());  
// o System.out.println(nivel);
```

# Consideraciones sobre el tipo enum

- proporcionan un método estático `valueOf()` – complementa a `toString()`.
  - `Nivel.valueOf("MEDIO")` devuelve `Nivel.MEDIO`
- cada enumerado define un método público estático `values()` que devuelve un array cuyos elementos contienen los valores del tipo

```
Nivel[ ] niveles = Nivel.values();
for (int i = 0; i < niveles.length; i++) {
    System.out.println(niveles[i].toString());
}
```

# El tipo enum con switch

```
switch (nivel)  
{
```

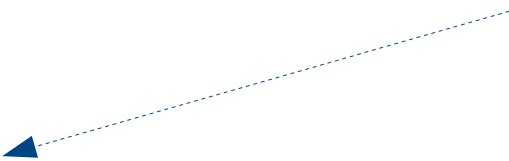
```
    case BAJO: .....  
        break;
```

```
    case MEDIO: .....  
        break;
```

```
    case ALTO: .....  
        break;
```

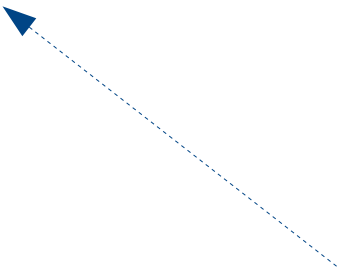
```
}
```

no se especifica  
nombre del tipo



# El tipo enum con if

```
if (nivel == Nivel.MEDIO)
{
    .....
}
```



se especifica  
nombre del tipo

# Constructores, atributos y métodos en un tipo enum

```
public enum Nivel
{
    BAJO (10), MEDIO (25), ALTO (50);

    private int nivel;
    private Nivel(int nivel)
    {
        this.nivel = nivel;
    }

    public int getValorNivel()
    {
        return nivel;
    }
}
```

- constantes con valor asociado
  - argumento
- constructor privado
  - se invoca al declarar la constante
- el argumento se pasa al constructor
- podemos incluir otros métodos
  - accedores
  - redefinir toString()



# El tipo enum – Ejer 5.27 5.28 5.29

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

```
public enum Estacion
{
    PRIMAVERA, VERANO, OTOÑO, INVIERNO;
}
```

```
public enum Direccion
{
    NORTE, SUR, ESTE, OESTE;
}
```

## El tipo enum – Ejer 5.29

```
public enum Valor
{
    AS(1), DOS(2), TRES(3), CUATRO(4), CINCO(5),
    SEIS(6), SIETE(7), SOTA(10), CABALLO(11), REY(12);

    private int valor;

    private Valor(int valor)
    {
        this.valor = valor;
    }

    public int getValor()
    {
        return valor;
    }
}
```