

# Elementos de Lógica de Programación

Jorge I. Meza  
[jimezam@autonoma.edu.co](mailto:jimezam@autonoma.edu.co)

2024



## Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International

This license requires that reusers give credit to the creator. It allows reusers to distribute, remix, adapt, and build upon the material in any medium or format, for noncommercial purposes only. If others modify or adapt the material, they must license the modified material under identical terms.

👤 **BY:** Credit must be given to you, the creator.

🚫💰 **NC:** Only noncommercial use of your work is permitted.

Noncommercial means not primarily intended for or directed towards commercial advantage or monetary compensation.

🔄 **SA:** Adaptations must be shared under the same terms.

# Contenidos

- Herramientas
- Concepto de *algoritmo*
- Unidad mínima de compilación
- Comentarios
- Variables
- Constantes
- Operadores
- Entrada y salida de datos
- Condicionales
- Iteraciones
- Arreglos y Matrices
- Funciones y procedimientos

# Herramientas

# Software Development Kit

## ➤ *Java Development Kit (JDK)*

- <https://jdk.java.net/>
- <https://learn.microsoft.com/en-us/java/openjdk/download>  
(para Windows)

Para desarrolladores

## ➤ *Java Runtime Edition (JRE)*

- <https://www.java.com/download/manual.jsp>

Para usuarios finales

# *Integrated Development Environment (IDE)*

Esta herramienta es opcional para desarrollar pero muy recomendada ya que facilita y agiliza las tareas del desarrollador.

Durante el curso se utilizará **Netbeans**, el cual puede ser descargado desde los siguientes URL.

1) Versión oficial

<https://netbeans.apache.org/front/main/download>

2) Versión empaquetada por Codelerety

<https://www.codelerity.com/netbeans/>

# Conceptos básicos

# Definición de algoritmo

"Es un conjunto de instrucciones o pasos definidos y finitos que se siguen para resolver un problema o realizar una tarea específica".



# Características de un algoritmo

- **Finitud:** Debe terminar después de un cierto número de pasos.
- **Claridad:** Cada paso debe estar claramente definido y ser comprensible sin ambigüedades.
- **Entrada:** Puede tener cero o más entradas.
- **Salida:** Produce una o más salidas.
- **Eficiencia:** Deben ser *eficientes* en términos de tiempo y recursos utilizados.

# Representaciones de un algoritmo

- **Pseudocódigo:** Una forma textual de describir un algoritmo con un estilo que mezcla elementos del lenguaje natural y elementos de programación.
- **Diagramas de flujo:** Representaciones gráficas que usan símbolos para mostrar los pasos y decisiones en el algoritmo.
- **Código en un lenguajes de programación:** Implementaciones concretas del algoritmo en un lenguaje de programación específico.

# Elementos generales de un algoritmo

- *Sentencias* → Ordenes que se ***ejecutan***
- *Expresiones* → Instrucciones que se ***evalúan***
- Variables
- Constantes
- Comentarios
- Funciones y procedimientos

# Unidad mínima de compilación

# Unidad mínima de compilación

- Consiste en la estructura de código mínima para considerar de un programa, el cual puede ser compilado, enlazado (si es el caso) y ejecutado.
- En Java, consiste en una *clase* con su respectivo *método* `main` como se muestra a continuación.
- Estos conceptos de clase y método se verán más adelante en el curso de Fundamentos de Programación Orientada a Objetos.

# Unidad mínima de compilación

```
class MiClase {  
    public static void main(String[] args) {  
        // Código fuente  
    }  
}
```

# Comentarios

# Comentarios

Los comentarios en general, son ignorados por el compilador, por lo tanto su contenido no tiene relevancia para el funcionamiento del programa.

Su utilidad radica en tareas como estas:

- Desactivar temporalmente secciones de código, lo cual puede resultar útil durante el desarrollo y la depuración.
- Documentar el código, es decir, dejar "mensajes técnicos" en el código que faciliten su futura lectura y comprensión.



# Comentarios

Comentario de una sola línea

```
// ...
```

Comentario de documentación  
(JavaDoc)

```
/**
```

```
...
```

```
...
```

```
*/
```

Comentario multi-línea

```
/*
```

```
...
```

```
...
```

```
*/
```

# Variables y constantes

# Concepto de variable

"Representan un segmento de la memoria reservado para el programa, en el cual se pueden asignar y obtener (guardar) valores".

# Concepto de variable

- En un lenguaje de **tipeado estático**, las variables deben *declararse antes de poderse utilizar*, ya que sus operaciones se verifican en compilación.
- En un lenguaje **fuertemente tipeado**, las variables una vez tienen un *tipo de datos* definido, este no puede ser modificado.

# Declaración de una variable

```
<tipo> <identificador> [= <valor inicial>];
```

## Ejemplos:

```
int cantidad;  
float precio;  
double masa = 340000;  
boolean estado = true;
```

# Declaración de una constante

```
final <tipo> <identificador> = <valor inicial>;
```

## Ejemplos:

```
final int CANTIDAD_MAXIMA = 50;  
final float IVA = 16;  
final double MASA_PLANETA = 340000;  
final char CARACTER_SALIDA = 'x';
```

# Tipos de datos simples o primitivos

byte	8 bits	-128 a 127
short	16 bits	-32,768 a 32,767
int	32 bits	-2,147,483,648 a 2,147,483,647
long	64 bits	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
float	32 bits	Precisión: Aproximadamente 6-7 dígitos decimales
double	64 bits	Precisión: Aproximadamente 15-16 dígitos decimales
boolean	1 bit	true o false
char	16 bits	0 a 65,535 (caracteres Unicode)

# Reglas para los identificadores

- El carácter Inicial debe ser una letra (mayúscula o minúscula), un signo de dólar (\$) o un guion bajo (\_). No puede empezar con un dígito.
- Los demás caracteres pueden ser letras, dígitos, signos de dólar (\$) o guiones bajos (\_).
- No se deben utilizar *palabras reservadas del lenguaje* como identificadores.
- Son sensibles a mayúsculas y minúsculas.



# Palabras reservadas de Java

<code>abstract</code>	<code>assert</code>	<code>boolean</code>	<code>break</code>	<code>byte</code>
<code>case</code>	<code>catch</code>	<code>char</code>	<code>class</code>	<code>const</code>
<code>continue</code>	<code>default</code>	<code>do</code>	<code>double</code>	<code>else</code>
<code>enum</code>	<code>extends</code>	<code>final</code>	<code>finally</code>	<code>float</code>
<code>for</code>	<code>goto</code>	<code>if</code>	<code>implements</code>	<code>import</code>
<code>instanceof</code>	<code>int</code>	<code>interface</code>	<code>long</code>	<code>native</code>
<code>new</code>	<code>null</code>	<code>package</code>	<code>private</code>	<code>protected</code>
<code>public</code>	<code>return</code>	<code>short</code>	<code>static</code>	<code>strictfp</code>
<code>super</code>	<code>switch</code>	<code>synchronized</code>	<code>this</code>	<code>throw</code>
<code>throws</code>	<code>transient</code>	<code>try</code>	<code>void</code>	<code>volatile</code>
<code>while</code>				

# Ejemplos identificadores variables

## Válidos

```
int edad;  
double _salario;  
String $nombre;  
float valorTotal2;  
char caracterEspecial;  
boolean isTrue;
```

## Inválidos

```
int 2edad;  
String nombre#;  
float valor total;  
char char;  
boolean true;
```

# Acerca de los identificadores variables

Modelan propiedades o elementos, como por ejemplo `nombre`, `velocidad`, `tiempo`, por ende son **sustantivos en singular** y se escriben en **minúsculas**.

Si su nombre es compuesto, se utiliza la nomenclatura *Camel Case*: `nombreCompleto`, `velocidadPromedio` y `tiempoTotal` por ejemplo.

# Acerca de los identificadores constantes

Modelan propiedades o elementos con valor fijo, como por ejemplo IVA o GRAVEDAD por ende son **sustantivos en singular** y se escriben en **mayúsculas**.

Si su nombre es compuesto, se utiliza la nomenclatura *Snake Case*:

IMPUESTO\_AGREGADO, GRAVEDAD\_PLANETA y LIMITE\_SUPERIOR **por ejemplo**.

# Alcance (*scope*) de una variable

Indica el ámbito o segmento del código fuente en el cual, una variable específica puede ser manipulada. Este segmento de código depende del lugar en el cual fue declarada y los bloques de código que tenga el programa.

# Alcance (*scope*) de una variable

```
void miFuncion() {  
    int v1 = 1;  
  
    if(expr1) {  
        int v2 = 2;  
    }  
    else {  
        int v3 = 3;  
    }  
  
    int v4 = 4;  
}
```

```
void miFuncion() {  
    int v1 = 1;  
  
    for(int i=0; i<10; i++) {  
        int v2 = 2;  
  
        if(i%2 == 0) {  
            int v3 = 3;  
        }  
        else {  
            int v4 = 4;  
  
            while(x<j) {  
                int v5 = 5;  
            }  
        }  
    }  
}
```

# Operador de asignación (=)

```
variable = (variable | valor |  
            expresión);
```

Siempre y cuando:

Tipo de datos(*variable*) == Tipo de datos(*expresión*)

# Operador de asignación (=)

Por ejemplo:

```
// 10 es un int  
int cantidad = 10;
```

```
// estado debe ser una variable boolean  
boolean tmp = estado;
```

```
// precioUnitario * cantidad debe ser un float  
float precioTotal = precioUnitario * cantidad;
```



# Conversión entre tipos de datos

# Conversión de tipos

Permite convertir un valor de un tipo de datos específico a otro (*aplican restricciones*).

Esta conversión se puede hacer **entre tipos familiares** (*casting* formal) o **entre tipos disímiles** como String (requiere API).

# *Casting* o promoción de datos

- Este tipo de conversión se puede realizar entre tipos de datos "familiares" entre sí, como por ejemplo `byte`, `short`, `int`, `long`, `float`, `double`, `char` (código ASCII). Nótese que todos son numéricos.
- Debe tenerse cuidado ya que la conversión de un valor desde un tipo de mayor tamaño hacia un tipo de menor tamaño, inexorablemente producirá la pérdida de información.

# *Casting* o promoción de datos

La sintáxis general de la promoción es la siguiente:



```
TipoOriginal entrada;
```

```
TipoResultado salida = (TipoResultado) entrada;
```


# Casting o promoción de datos

```
int d = 125;
```

```
float f = d;
```

```
float f = (float) d;   // 125.0
```

```
float r = 75.25;
```

```
int i = (int) r;  // 75
```

¡Hay pérdida de información!

# Conversión a String

```
otroTipo variable;  
String cadena = String.valueOf(variable);
```

Por ejemplo:

```
float velocidad = 12.5;  
String texto = String.valueOf(velocidad);  
// "12.5"
```

# Conversión a String (opción menos elegante)

```
otroTipo variable;  
String cadena = "" + variable;
```

Por ejemplo:

```
float velocidad = 12.5;  
String texto = "" + velocidad;  
// "12.5"
```

# Conversión de String a otro tipo

```
String cadena;  
otro variable = Otro.parseOtro(cadena);
```

Por ejemplo:

```
String codigo = "1237";  
int numero = Integer.parseInt(codigo);  
// 1237
```



# Conversión de String a otro tipo

<code>byte</code>	→	<code>Byte</code>	→	<code>parseByte</code>
<code>short</code>	→	<code>Short</code>	→	<code>parseShort</code>
<code>int</code>	→	<code>Integer</code>	→	<code>parseInt</code>
<code>long</code>	→	<code>Long</code>	→	<code>parseLong</code>
<code>float</code>	→	<code>Float</code>	→	<code>parseFloat</code>
<code>double</code>	→	<code>Double</code>	→	<code>parseDouble</code>
<code>boolean</code>	→	<code>Boolean</code>	→	<code>parseBoolean</code>

# Operadores

# Operadores aritméticos

+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo o residuo

# Operadores de autoincremento

$a = a + b$	$a += b$
$a = a - b$	$a -= b$
$a = a * b$	$a *= b$
$a = a / b$	$a /= b$
$a = a \% b$	$a \% = b$

# Operadores de autoincremento

<code>a ++</code>	<code>a = a + 1</code>	<code>a += 1</code>
<code>b --</code>	<code>b = b - 1</code>	<code>b -= 1</code>

# Operadores relacionales

>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
==	Igual a
!=	Diferente de

# Operadores lógicos o booleanos

&	Y	Conjunción	<i>AND</i>
	O	Disyunción	<i>OR</i>
!	NO	Negación	<i>NOT</i>

& &	Y en cortocircuito
	O en cortocircuito

# Tabla de verdad de la negación (*NOT*)

!	q
1	0
0	1



# Tabla de verdad de la Y (*AND*)

p	&	q
0	0	0
0	0	1
1	0	0
1	1	1

# Tabla de verdad de la O (*OR*)

p		q
0	0	0
0	1	1
1	1	0
1	1	1

# Entrada y salida de datos estándar

# Flujos de datos estándar

- Salida estándar  
STDOUT → Pantalla (*consola*)
- Salida de error  
STDERR → Pantalla (*consola*)
- Entrada estándar  
STDIN → Teclado

# Salida de datos

- La salida de datos consiste en mostrar información del programa al usuario a través de la pantalla, es decir, salida estándar (STDOUT).
- En Java, esto se realiza mediante el método `System.out.print` y `System.out.println`.
- La diferencia entre `print` y `println` radica en que el segundo agrega automáticamente un salto de línea (`'\n'`) a la cadena impresa.

# Salida de datos: ejemplo

```
System.out.println("Hola");      Hola  
System.out.println("Mundo");     Mundo
```

```
System.out.print("Hola");        HolaMundo  
System.out.print("Mundo");
```

# Salida de datos: ejemplo

```
String nombre = "Jorge";  
int edad = 25;
```

```
System.out.println("¡Bienvenido!");
```

¡Bienvenido!

```
System.out.println();
```

Línea en blanco

```
System.out.println("Hola " +  
    nombre + ", ¿Cómo estás?");
```

Hola Jorge, ¿Cómo estás?

```
System.out.println("Tu tienes " +  
    edad + " años.");
```

Tu tienes 25 años.

# Entrada de datos

- La entrada de datos consiste en ingresar información al programa a través del teclado, es decir, entrada estándar (STDIN).
- En Java, existen varias formas de hacerlo, una de ellas es utilizando la clase `java.util.Scanner` como se muestra a continuación.



# Entrada de datos

```
import java.util.Scanner;

Scanner teclado = new Scanner(System.in);

System.out.print("Ingresa tu nombre: ");
String nombre = teclado.nextLine();
System.out.println("Hola, " + nombre + "!");

System.out.print("Ingresa tu edad: ");
int edad = teclado.nextInt();
System.out.println("Tienes " + edad + " años.");

System.out.print("Ingresa tu altura en metros: ");
double altura = teclado.nextDouble();
System.out.println("Mides " + altura + " metros.");

teclado.close();
```

# Entrada de datos

Investigar que otros útiles métodos `nextXXX` tiene la clase `java.util.Scanner` según su documentación.

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP			Java SE 21 & JDK 21
SUMMARY: NESTED   FIELD   CONSTR   METHOD		DETAIL: FIELD   CONSTR   METHOD	SEARCH <input type="text" value="Search"/>
String	next()	Finds and returns the next complete token from this scanner.	
String	next(String pattern)	Returns the next token if it matches the pattern constructed from the specified string.	
String	next(Pattern pattern)	Returns the next token if it matches the specified pattern.	
BigDecimal	nextBigDecimal()	Scans the next token of the input as a BigDecimal.	
BigInteger	nextBigInteger()	Scans the next token of the input as a BigInteger.	
BigInteger	nextBigInteger(int radix)	Scans the next token of the input as a BigInteger.	
boolean	nextBoolean()	Scans the next token of the input into a boolean value and returns that value.	
byte	nextByte()	Scans the next token of the input as a byte.	
byte	nextByte(int radix)	Scans the next token of the input as a byte.	
double	nextDouble()	Scans the next token of the input as a double.	
float	nextFloat()	Scans the next token of the input as a float.	
int	nextInt()	Scans the next token of the input as an int.	
int	nextInt(int radix)	Scans the next token of the input as an int.	
String	nextLine()	Advances this scanner past the current line and returns the input that was skipped.	
long	nextLong()	Scans the next token of the input as a long.	
long	nextLong(int radix)	Scans the next token of the input as a long.	
short	nextShort()	Scans the next token of the input as a short.	
short	nextShort(int radix)	Scans the next token of the input as a short.	

# Generación de números pseudoaleatorios

# Generación de números al azar

La forma más simple consiste en utilizar el método `Math.random()` el cual genera valores pseudoaleatorios entre 0 y 1 más específicamente entre  $[0, 1)$ .

```
double valorAzar = Math.random();
```

# Generación de números al azar

Generar un valor al azar entre 0 y 15 (sin incluirlo).

```
double valor = Math.random() * 15;
```

Generar un valor **entero** al azar entre 0 y 15 (sin incluirlo).

```
int valor = (int) (Math.random() * 15);
```

Generar un valor **entero** al azar entre **5** y 15 (sin incluirlo).

```
int valor = (int) (Math.random() *  
                  (15 - 5 + 1)) + 5;
```

# Sentencias condicionales

# Sentencias condicionales

"Permite que el programa tome decisiones a partir de la evaluación de una expresión y con esto, elija que bloque de código ejecutar".

Tipos:

- 1) IF-ELSE
- 2) SWITCH
- 3) IF ternario

# Sentencias condicionales: IF

```
if (expresión) {  
    BloqueA  
}
```

- La *expresión* debe ser booleana, es decir, al evaluarse deberá dar como resultado un valor booleano: `true` o `false`.
- Si la expresión se evalúa como `true`, el BloqueA se ejecutará.



# Sentencias condicionales: IF-ELSE

```
if (expresión) {  
    BloqueA  
} else {  
    BloqueB  
}
```

- La *expresión* debe ser booleana.
- Si la expresión se evalúa como `true`, el BloqueA se ejecutará.
- De lo contrario, si se evalúa como `false`, el BloqueB se ejecutará.

# Sentencias condicionales: anidamiento

El anidamiento o *nesting*, permite tener sentencias (condicionales o de iteración) como contenido de los *bloques* de otras sentencias.

# Sentencias condicionales: anidamiento

```
if(encendido == true) {  
    if(velocidad > 0) {  
        // A  
    }  
    else {  
        // B  
    }  
}  
else {  
    if(velocidad > 0) {  
        // C  
    }  
}
```

¿Cuál debe ser el estado de las variables para que se ejecute el bloque A, el B o el C de este programa?

# Sentencias condicionales: anidamiento

```
if(encendido == true) {  
    if(velocidad > 0) {  
        // A  
    }  
    else {  
        // B  
    }  
}  
else {  
    if(velocidad > 0) {  
        // C  
    }  
}
```

```
boolean encendido = true;
```

```
if(encendido) {  
    ...  
}
```

# Sentencias condicionales: bloques

Para las sentencias condicionales (IF), al igual que con las sentencias de iteración (WHILE, DO-WHILE, FOR y FOREACH) revisadas más adelante, es necesario especificar el bloque de instrucciones que cobijarán. Este bloque cumple las siguientes reglas.

- 1) Si el bloque tiene **una sola sentencia**, no es obligatorio utilizar { ... }. En caso de hacerse, es inócuo.
- 2) Si el bloque tiene **más de una setencia**, es obligatorio utilizar { ... }.

# Sentencias condicionales: bloques

```
if (salud > 50) {  
    print("Bien");  
}
```

```
if (salud > 50)  
    print("Bien");
```



```
if (funcionando) {  
    for (int i=0; i<5; i++) {  
        print (maquina[i]);  
    }  
}
```

```
if (funcionando)  
    for (int i=0; i<5; i++) {  
        print (maquina[i]);  
    }
```



```
if (funcionando)  
    for (int i=0; i<5; i++)  
        print (maquina[i]);
```



# Sentencias condicionales: bloques

```
if(funcionando) {  
    int longitud = maquinas.length;  
  
    for(int i=0; i<longitud; i++) {  
        double valor = maquina[i];  
        print(valor);  
    }  
}
```



# Sentencias condicionales: SWITCH

```
switch (expresión) {  
    case 1:  
        break;  
  
    case 2:  
        break;  
  
    case 7:  
        break;  
  
    default:  
        break;  
}
```

- La expresión debe ser numérica, de carácter (`char`) o cadena (`String`).
- Cada bloque `case` identifica el valor que lo "activará" si coincide con el de expresión. Debe terminar con un `break`.
- El bloque `default` es opcional y se "activa" cuando ningún bloque `case` coincidió con su valor.



# Ejemplo: descripción de nota con IF-ELSE

```
if(nota == 1)
    System.out.println("Te fue mal");
else
    if(nota == 2)
        System.out.println("Te fue regular");
    else
        if(nota == 3)
            System.out.println("Te fue bien");
        else
            System.out.println("ERROR, nota
inválida: " + nota);
```

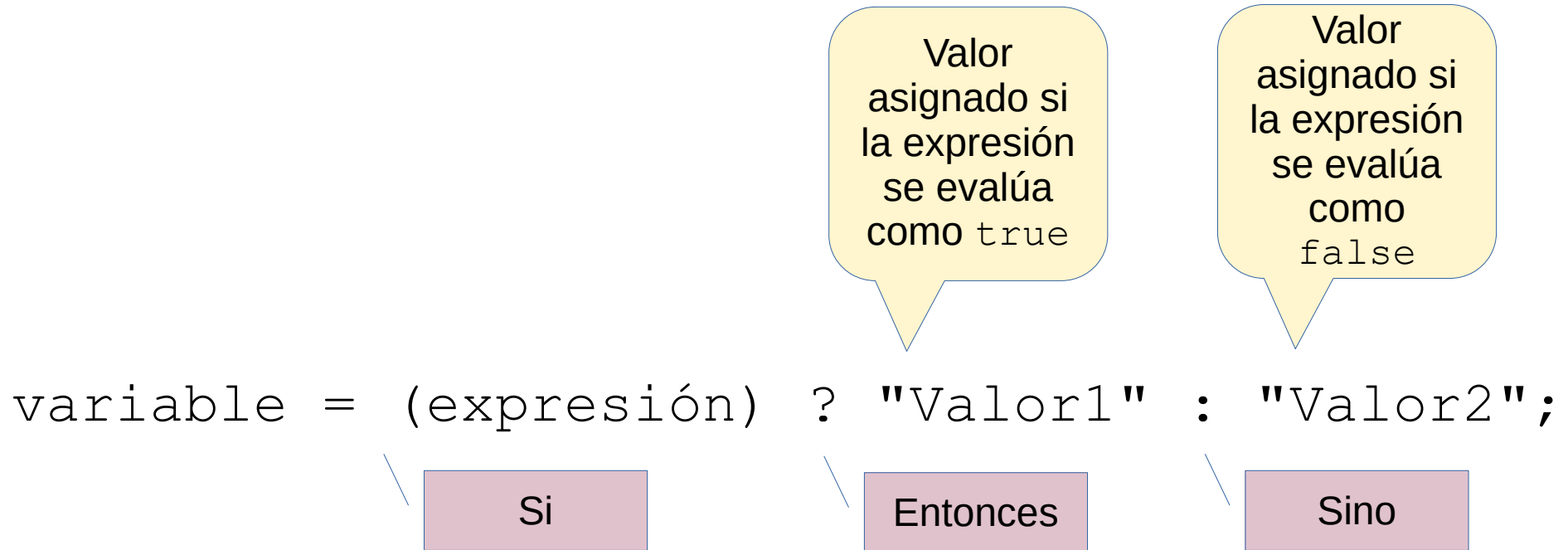
# Ejemplo: descripción de nota con SWITCH

```
switch(nota) {  
    case 1: System.out.println("Te fue mal");  
        break;  
    case 2: System.out.println("Te fue regular");  
        break;  
    case 3: System.out.println("Te fue bien");  
        break;  
    default: System.out.println("ERROR, nota  
inválida: " + nota);  
        break;  
}
```

# IF ternario

- A diferencia del IF ... ELSE que es una sentencia, el IF ternario es una expresión, así que se debe evaluar y obtener un valor resultante de él.
- Es útil cuando para hacer un código más compacto cuando se va a determinar el valor de una variable a partir de otra o de una expresión.

# IF ternario



# IF ternario

```
variable = (expresión) ? "Valor1" : "Valor2";
```

Es equivalente a:

```
if (expresión)  
    variable = "Valor1";  
else  
    variable = "Valor2";
```

# IF ternario

```
boolean estado;  
String mensaje = (estado) ? "Presente" :  
"Ausente";  
  
float velocidad;  
int movimiento = (velocidad > 0) ? 1 : 0;  
  
int año;  
boolean reciente = (año >= 2020) ? true :  
false;
```

# Sentencias de iteración

# Sentencias de iteración o ciclos

- Permiten ejecutar en múltiples ocasiones a ciertas sentencias específicas.
- Existen diferentes tipos de ciclos con la misma finalidad pero con variaciones en su comportamiento.
  - WHILE ... DO
  - DO ... WHILE
  - FOR ... NEXT
  - FOR EACH (más adelante)



# Iteraciones con WHILE ... DO

- No se conoce cuántas iteraciones máximo se van a realizar. Mínimo se realizan cero iteraciones.
- El bloque de código se ejecuta *mientras que* la evaluación de la expresión sea `true`.
- Es necesario garantizar el cambio de valor de la expresión.

```
while (expresión)
{
    // ...
}
```

# Iteraciones con WHILE ... DO



```
int cantidad = 3;
int i = 0;
int acum = 0;

while (i < cantidad) {
    acum +=
teclado.nextInt();
    i++;
}

float promedio =
acum / cantidad;
```



```
int i = 0;
int r = 0;

while (i < 10) {
    r += arr[i];
}

System.out.println
(arr[i]);
```

# Iteraciones con DO ... WHILE

- No se conoce cuántas iteraciones máximo se van a realizar. Mínimo se realiza una iteración.
- El bloque de código se ejecuta *mientras que* la evaluación de la expresión sea `true`.
- Es necesario garantizar el cambio de valor de la expresión.

```
do {  
    // ...  
} while (expresión);
```

# Iteraciones con DO ... WHILE

```
int dato;  
int buscado;  
  
do {  
    dato = leer();  
    if(dato == buscado)  
        return true;  
} while(hayDatos());  
  
return false;
```

# Iteraciones con FOR ... NEXT

- Se conoce cuántas iteraciones se van a realizar.
- El bloque de código se ejecuta *mientras que* la evaluación de la expresión sea `true`.
- La expresión se crea en función de la variable de iteración que se actualiza con el incremento especificado.

```
for (int i=0; i<10; i++)  
{  
    // ...  
}
```

Condición

Incremento

Variables de  
iteración  
Estado inicial

# Iteraciones con FOR ... NEXT

```
int acum = 1;
for(int i=0; i<5; i++)
{
    acum *= (i + 1);
}
```

```
int limInf = 3;
int limSup = 9;
for(int j=limInf;
    j<limSup; i+=2) {
    if(a[j] % 2 == 0)
        acum += a[j];
}
```

# Ciclos infinitos ⚠

```
while(true) {  
    // ...  
}
```

```
for ( ; ; ) {  
    // ...  
}
```

```
do {  
    // ...  
} while(true);
```

# Modificadores de flujo

Como su nombre lo indica, permiten alterar el flujo normal de las sentencias de iteración.

1) `break`

2) `continue`



# Modificadores de flujo: `break`

Permite romper para siempre o terminar abruptamente el ciclo cuando se desee.

```
int i = 0;

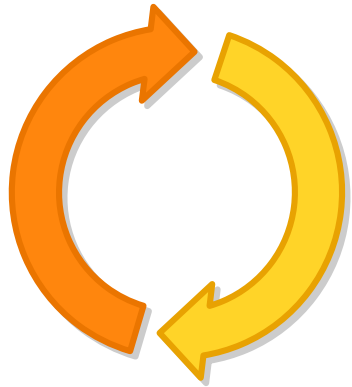
while (i < 100) {

    if (i % 2 == 0 & i > 10)
        break;

    System.out.println(i);
    i++;

}
```

# Modificadores de flujo: `break`



```
for ( ... ) {
```

```
    If( ... )
```

```
        break;
```

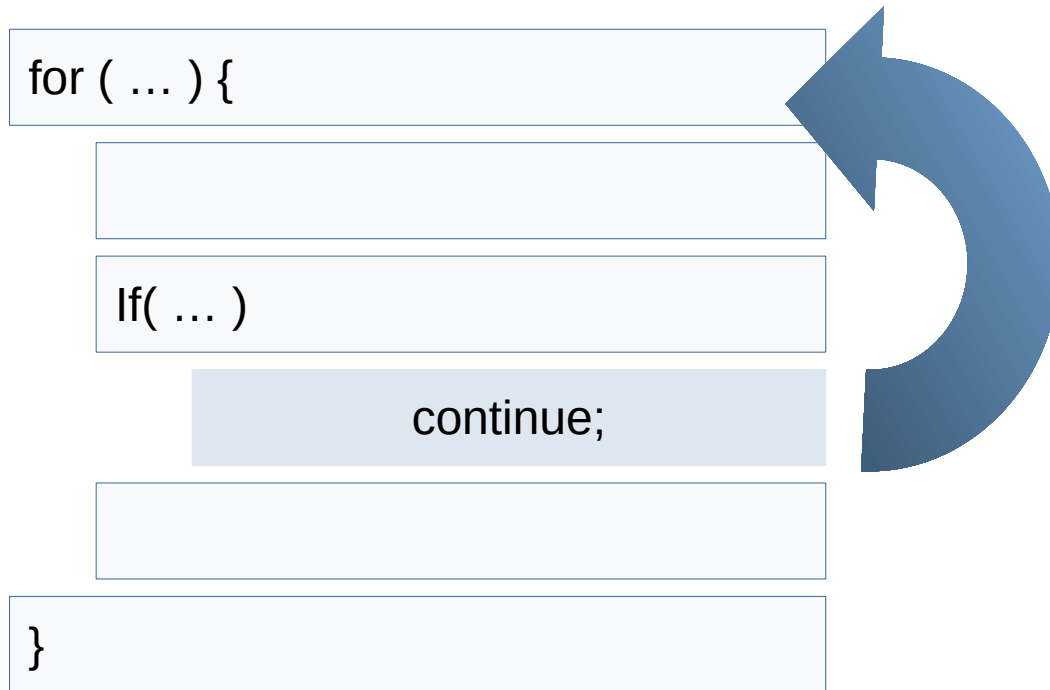
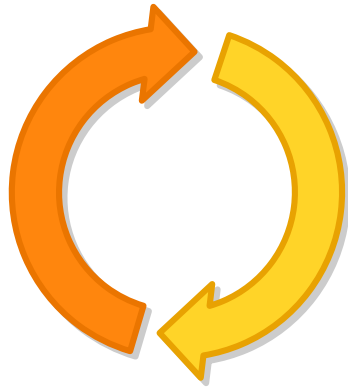
```
}
```

# Modificadores de flujo: `continue`

Permite saltar la iteración activa pero continúa en la siguiente. No termina ni se sale del ciclo.

```
int i = 0;
while (i<100) {
    i ++;
    if (i%2 == 0 & i>10)
        continue;
    System.out.println(i);
}
```

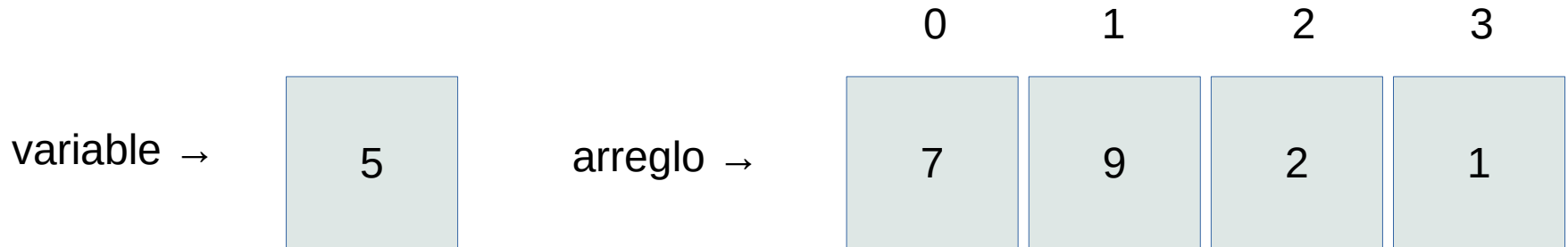
# Modificadores de flujo: `continue`



# Arreglos y matrices

# Arreglos

- También llamados *arrays* o vectores.
- Sus valores modelan características por lo tanto son **sustantivos**, en **plural** y se escriben en **minúsculas**.
- Permiten almacenar múltiples valores bajo un mismo identificador.



# Características de los arreglos

- Su longitud es estática, no pueden crecer ni encogerse una vez son creados.
- Los valores ubicados en cada una de sus celdas deben ser del mismo tipo de datos del cual fue declarado el arreglo.
- Para acceder a los valores ubicados en las celdas del arreglo, se utiliza un índice que empieza en cero y se utiliza el operador corchetes para enunciarlo.

# Declaración de arreglos

```
tipo [] identificador = new tipo [tamaño] ;
```

Por ejemplo:

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

```
double[] precios = new double[10];
```

```
String[] nombres = new String[55];
```



# Declaración de arreglos

`tipo [] identificador;`

Declaración

// ...

En este segmento del código, el valor del arreglo (`identificador`) es `null` y por ende no puede utilizarse aún.

`identificador = new tipo[tamaño];`

Creación

# Acceso a las celdas de los arreglos

```
String[] nombres = new String[4];
```

```
nombres[0] = "Jorge";
```

```
nombres[1] = "Pedro";
```

```
nombres[2] = "Rosa";
```

```
nombres[3] = "María";
```

```
System.out.println("Mi mejor amigo es " +  
nombres[0]);
```

```
for(int i=1; i<4; i++)  
    System.out.println("- " + nombres[i]);
```

# Declaración y valores iniciales de un arreglo

- Paso #1: Crear el arreglo

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

- Paso #2: Dar valores a las celdas del arreglo

```
datos[0] = 3;
```

```
datos[1] = 5;
```

```
datos[2] = 7;
```

```
datos[3] = 9;
```

```
datos[4] = 2;
```

# Declaración y valores iniciales de un arreglo (versión rápida)

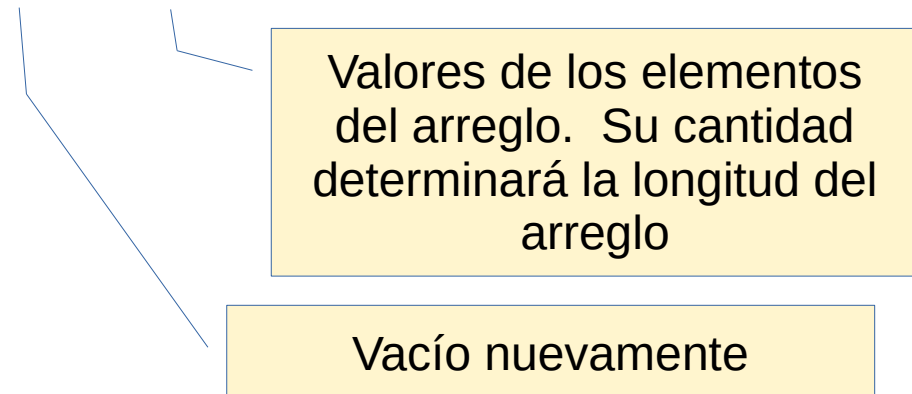
- Esta opción puede elegirse siempre y cuando, desde el momento de la creación del arreglo se conozca la cantidad de elementos y los valores individuales de los elementos que se almacenarán en el arreglo.
- Una vez hecho esto es posible modificar los valores de las celdas del arreglo pero no su longitud (¡es estático!).

# Declaración y valores iniciales de un arreglo (versión rápida)

- Paso #1: Crear el arreglo, especificando los valores a las celdas del arreglo

```
int[] datos = new int[] {3, 5, 7, 9, 2};
```

- Paso #2: Disfrutar :-)



Valores de los elementos del arreglo. Su cantidad determinará la longitud del arreglo

Vacío nuevamente

# Recorrido de los arreglos

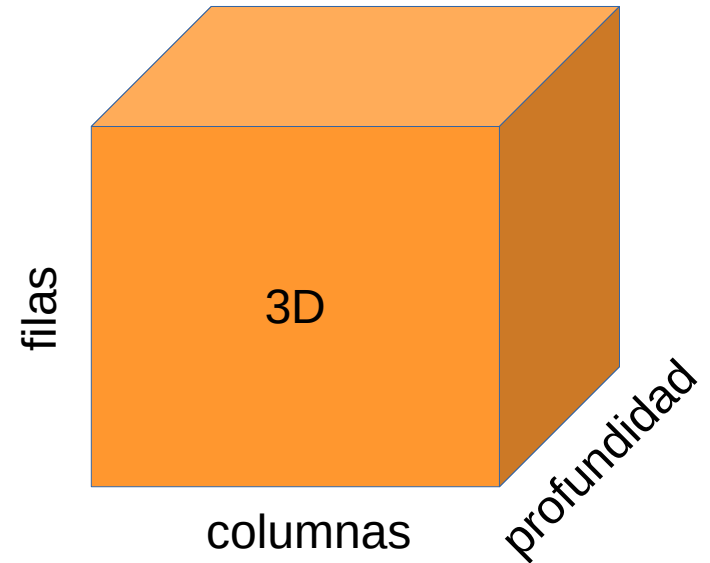
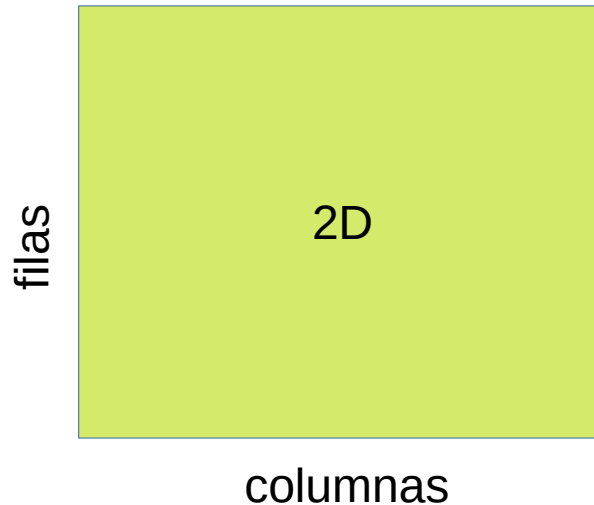
```
void adivinal(int[] datos) {  
    for(int i=0; i<datos.length; i++) {  
        System.out.println(datos[i]);  
    }  
}  
  
boolean adivina2(char[] datos, char x) {  
    for(int i=0; i<datos.length; i++) {  
        if(datos[i] == x)  
            return true;  
    }  
    return false;  
}
```

# Matrices de n-dimensiones

- Permiten generalizar el concepto de arreglo a  $n$  dimensiones ( $n \geq 2$ ).
- Para acceder a una celda, se requiere de un índice por cada dimensión.
  - Arreglos                      - índice
  - Matrices 2D                - fila/columna
  - Matrices 3D                - **fila/columna/profundidad**

# Matrices de n-dimensiones

Índice - longitud



4D – no se puede graficar  
en este universo.



# Matrices de n-dimensiones: declaración

```
// Arreglos
```

```
int[] velocidades = new int[10];
```

```
// Matrices
```

```
char[][] ubicaciones = new char[7][5];
```

```
// Cubos (matrices 3D)
```

```
double[][][] valores = new double[20][5][8];
```

# Matrices de n-dimensiones: recorrido

```
// Arreglos
```

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

```
// Matrices 2D
```

```
for(int f=0; f<ubicaciones.length; f++) {  
    for(int c=0; c<ubicaciones[0].length; c++) {  
        System.out.println(ubicaciones[f][c]);  
    }  
}
```

# Matrices de n-dimensiones: recorrido

```
// Matrices 3D
```

```
for(int f=0; f<valores.length; f++) {  
    for(int c=0; c<valores[0].length; c++) {  
        for(int p=0; p<valores[0][0].length; p++) {  
            System.out.println(valores[f][c][p]);  
        }  
    }  
}
```

# Sentencias de iteración (continuación)

# Iteraciones con FOREACH

- Este tipo de iteración tiene una sintáxis corta y es útil cuando se desea recorrer una colección o arreglo.
- Tiene algunas restricciones:
  - Debe recorrerse el arreglo completo, no permite elegir los índices de inicio o terminación
  - No es posible conocer el índice de la iteración en que se obtuvo algún elemento
  - Es de sólo lectura, es decir, no deben realizarse modificaciones al arreglo mientras se itera

# Iteraciones con FOREACH

```
for(tipo elemento : arreglo) {  
    // ...  
}
```



Tomado de <https://www.redalyc.org/journal/5055/505554802003/html/>

# Iteraciones con FOREACH

```
int[] arreglo = new int[5];
int acumulador = 0;

for(int e : arreglo) {
    if(e > 10)
        acumulador += e;
}

double resultado =
    acumulador / arreglo.length;
```



```
int[] arreglo = new int[5];
int acumulador = 0;
int indice = 0;

for(int e : arreglo) {
    if(e > 10)
        acumulador += e;
    else
        arreglo[i] = 0;

    indice ++;
}
```



# Funciones y procedimientos



# Modularidad

- Claridad en el desarrollo
- Facilidad de depuración y mantenimiento
- Reutilización
- Mayor velocidad de construcción
- Mayor escalabilidad



# Paradigma procedimental

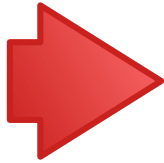
- Propone fortalecer la modularidad del código fuente para hacer mejores programas.
- Divide la implementación del programa en diferentes *subrutinas* que se pueden utilizar de manera independiente.
- Esto mejora la legibilidad, el mantenimiento y la reutilización del código.

# Paradigma procedimental: ventajas

- Claridad y organización
  - módulos con responsabilidad única.
- Reutilización de código
  - no partir desde cero cada proyecto
- Facilidad de depuración
  - claridad en las funcionalidades y responsables
- Facilidad de mantenimiento
  - cambios con mínimo impacto

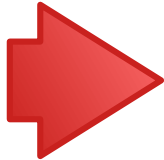
# Paradigma procedimental

- Datos



- Variables
- Constantes
- Arreglos
- Matrices

- Funcionalidad



- Subrutinas
  - Funciones
  - Procedimientos

# Subrutinas

Bloques de código independientes y reutilizables con un objetivo específico y único

## Funciones

Realizan su procesamiento y **retornan** un valor

## Procedimientos

Realizan su procesamiento y **NO retornan** un valor

# Elementos de las subrutinas

- Identificador
- Parámetros
- Tipo de retorno
- Bloque de código
- Valor de retorno

Sólo para funciones

Especificación

(qué se *promete* hacer)

Implementación

(cómo se va a hacer)

# Elementos de las subrutinas: identificador

- Es el *nombre* de la subrutina, este se utiliza para *invocarla*.
- Como denota una acción, su nombre debe ser un **verbo** y se debe escribir en **infinitivo**.
- En Java (métodos) debe escribirse en minúsculas y si el nombre es compuesto, la primera letra de la segunda palabra en adelante, debe escribirse en mayúsculas (*Caml Case*).

# Elementos de las subrutinas: identificador

Por ejemplo:

- dibujar
- buscar
- ordenar
- calcularArea
- determinarLimite
- hallarEstudianteMasJoven

## Sugerencia

No utilizar caracteres especiales en los identificadores como tildes o ñ, a pesar de UTF8

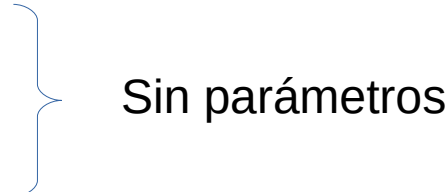


# Elementos de las subrutinas: parámetros

- Son los **datos externos** que la subrutina **necesita** recibir **para poder realizar su trabajo**
- No todas las subrutinas requieren de parámetros
- De cada parámetro se conoce su *tipo de datos* y su *identificador*
- Los parámetros deben ser resueltos cuando se invoca la subrutina y se envían los datos que los suplirán, llamándose **argumentos**.

# Elementos de las subrutinas: parámetros

Por ejemplo:

- buscar (int[] valores, int dato)
  - ordenar (String[] nombres)
  - calcularArea (double base, double altura)
  - buscarMasJoven (Persona[] personas)
  - reiniciarValores ()
  - apagarVehiculo ()
- 
- Sin parámetros

# Elementos de las subrutinas: tipo de retorno

- Indica el **tipo de datos que puede retornar la subrutina**
- Pueden utilizarse tipos de datos primitivos o compuestos (como clases)
- Sólo aplica para subrutinas de tipo *función*
- Para subrutinas de tipo *procedimiento*, siempre es `void` para indicar que nunca retornan *nada*

# Elementos de las subrutinas: parámetros

Por ejemplo:

- **int** buscar (int[] valores, int dato)
- **void** ordenar (String[] nombres)
- **double** calcularArea (double base, double altura)
- **Persona** buscarMasJoven (Persona[] personas)
- **void** reiniciarValores()
- **boolean** apagarVehiculo()

# Elementos de las subrutinas: bloque de código

- Determina la **implementación de la subrutina**, es decir, el como se realizará la tarea específica
- En esta implementación se puede **acceder a los parámetros** enviados como si fueran variables locales convencionales
- Las variables locales declaradas en este bloque, desaparecen tan pronto como se cierra
- En el caso de las *funciones*, es necesario que se garantice siempre un valor retornado

## Elementos de las subrutinas: valor de retorno

- Este valor indica la **respuesta que se le va a dar a quien invoque** la subrutina
- Sólo **aplica para las *funciones*** ya que todas los *procedimientos* retornan siempre `void`
- Si la implementación de la función utiliza sentencias condicionales o de iteración, es necesario garantizar que **todos los posibles caminos de código retornen un valor específico**

# Ejemplos de subrutinas

```
void reiniciarValores() {  
    distancia = 0;  
    velocidad = 0;  
}
```

Invocación:

```
reiniciarValores();
```

# Ejemplos de subrutinas

```
boolean apagarVehiculo() {  
    if(estado == true) {  
        estado = false;  
        return true;  
    }  
    else  
        return false;  
}
```

**Invocación:**

```
boolean retorno = apagarVehiculo();
```



# Ejemplos de subrutinas

```
double calcularArea (double base, double altura) {  
    double resultado = base * altura / 2;  
    return resultado;  
}
```

Invocación:

```
double area = calcularArea(20, 10);
```

# Ejemplos de subrutinas

```
int buscar (int[] valores, int dato) {  
    for(int i=0; i<valores.length; i++) {  
        if(valores[i] == dato)  
            return i;  
    }  
    return -1;  
}
```

## Invocación:

```
int[] elementos = new int[] {10, 20, 50, 70};  
int r1 = buscar (elementos, 50);  
int r2 = buscar (elementos, 17);
```

# Ejemplos de subrutinas

```
Persona buscarMasJoven (Persona[] personas) {  
    int posicion = 0;  
    for(int i=1; i<personas.length; i++) {  
        if(personas[i].obtEdad() <  
            personas[posicion].obtEdad())  
            posicion = i;  
    }  
    return personas[posicion];  
}
```

## Invocación:

```
Persona p = buscarMasJoven (personas);
```

# Ejemplos de subrutinas

```
void ordenar (String[] nombres) {  
    int n = nombres.length;  
    boolean intercambio;  
    do {  
        intercambio = false;  
        for (int i=0; i<n-1; i++) {  
            if (nombres[i].compareTo(nombres[i+1]) > 0) {  
                String temp = nombres[i];  
                nombres[i] = nombres[i+1];  
                nombres[i+1] = temp;  
                intercambio = true;  
            }  
        }  
        n--;  
    } while (intercambio);  
}
```

Invocación:

```
buscarMasJoven (personas);
```

¿Fin?