



## Capítulo 3

# Programación estructurada

### 3.1. Resultados de aprendizaje

**1. Reconoce la estructura de un programa informático, identificando y relacionando los elementos propios del lenguaje de programación utilizado.**

- b) Se han creado proyectos de desarrollo de aplicaciones.
- c) Se han utilizado entornos integrados de desarrollo.
- d) Se han identificado los distintos tipos de variables y la utilidad específica de cada uno.
- e) Se ha modificado el código de un programa, para crear y utilizar variables.
- f) Se han creado y utilizado constantes y literales.
- g) Se han clasificado, reconocido y utilizado en expresiones los operadores del lenguaje.
- h) Se ha comprobado el funcionamiento de las conversiones de tipo explícitas e implícitas.
- i) Se han introducido comentarios en el código.

**2. Escribe y prueba programas sencillos, reconociendo y aplicando los fundamentos de la programación orientada a objetos.**

- a) Se han identificado los fundamentos de la programación orientada a objetos.
- b) Se han escrito programas simples.
- c) Se han instanciado objetos a partir de clases predefinidas.

- d) Se han utilizado métodos y propiedades de los objetos.
- e) Se han escrito llamadas a métodos estáticos.
- f) Se han utilizado parámetros en la llamada a métodos.
- g) Se han incorporado y utilizado librerías de objetos.
- h) Se han utilizado constructores.
- i) Se ha utilizado el entorno integrado de desarrollo en la creación y compilación de programas

**3. Escribe y depura código, analizando y utilizando las estructuras de control del lenguaje.**

- a) Se ha escrito y probado código que haga uso de estructuras de selección.
- b) Se han utilizado estructuras de repetición.
- c) Se han reconocido las posibilidades de las sentencias de salto.
- d) Se ha escrito código utilizando control de excepciones.
- e) Se han creado programas ejecutables utilizando diferentes estructuras de control.
- f) Se han probado y depurado los programas.
- g) Se ha comentado y documentado el código.

**5. Realiza operaciones de entrada y salida de información, utilizando procedimientos específicos del lenguaje y librerías de clases.**

- a) Se ha utilizado la consola para realizar operaciones de entrada y salida de información.
- h) Se han escrito programas que utilicen interfaces gráficos para la entrada y salida de información.

## 3.2. Estructura básica de un programa

---

```
1  /*
2   * Comentario indicando que es nuestro primer ejemplo.
3   * Ejemplo que quiere mostrar la estructura basica de un programa.
4   */
5  package ejemploestructura; // Paquete
6
7  /**
8   * Comentario indicando el nombre del autor del programa
9   * @author Nieves
10  */
11
12  public class EjemploEstructura { // Clase principal
13
14      /**
15       * Comentario
16       * @param args the command line arguments
17       */
18      public static void main(String[] args) { // Método main = inicial
19          // TODO code application logic here (Comentario que indica que
20              aquí tenemos que empezar a desarrollar el código
21
22          System.out.println("Hola mundo"); // Sentencia o expresión
23      }
24  }
```

---

## 3.3. Comentarios

Un comentario es una construcción destinada a integrar información adicional en el código fuente de un programa. Cuando el código fuente es procesado por un compilador o intérprete, los comentarios no se toman en cuenta.

Los comentarios tienen una amplia gama de posibles usos: desde la mejora del código fuente con descripciones básicas hasta la generación de documentación externa (javadoc). También se utilizan para la integración con sistemas de control de versiones y otros tipos de herramientas de programación externas.

En Java hay tres tipos de comentarios:

// comentarios para una sola línea

/\* comentarios de una o más líneas \*/

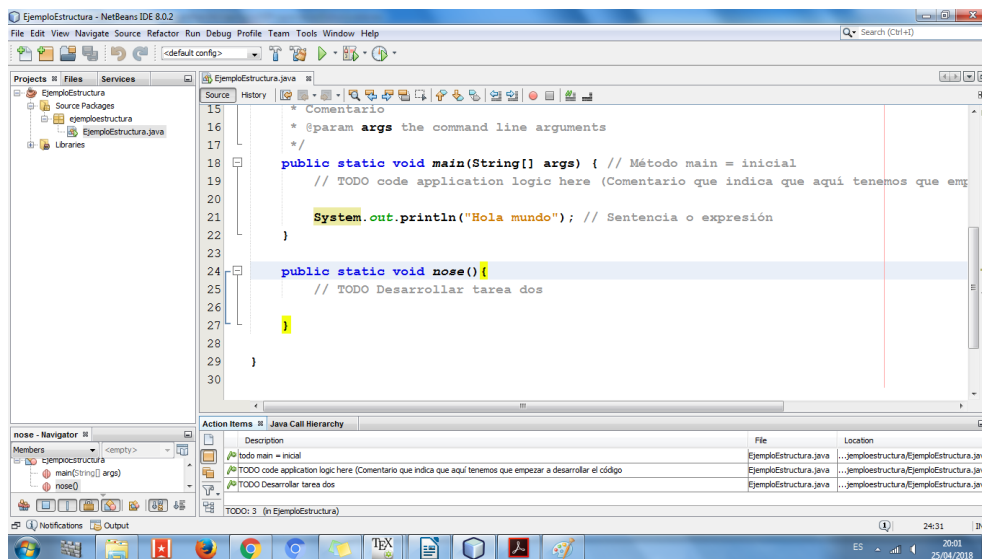
/\*\* comentario de documentación, de una o más líneas \*/

Los comentarios de documentación, colocados inmediatamente antes de una declaración (de variable o función), indican que ese comentario ha de

ser colocado en la documentación que se genera automáticamente cuando se utiliza la herramienta de Java *javadoc*. Dichos comentarios sirven como descripción del elemento declarado permitiendo generar una documentación al mismo tiempo que se genera el código.

Los comentarios que llevan la palabra TODO (por hacer) nos permiten estructurar un programa (decir que hay que hacer) y posteriormente hacerlo. Nos permiten documentar las tareas futuras.

Posteriormente para verlas y completarlas podemos ir a Task, Action items o pulsar ctrl 6.

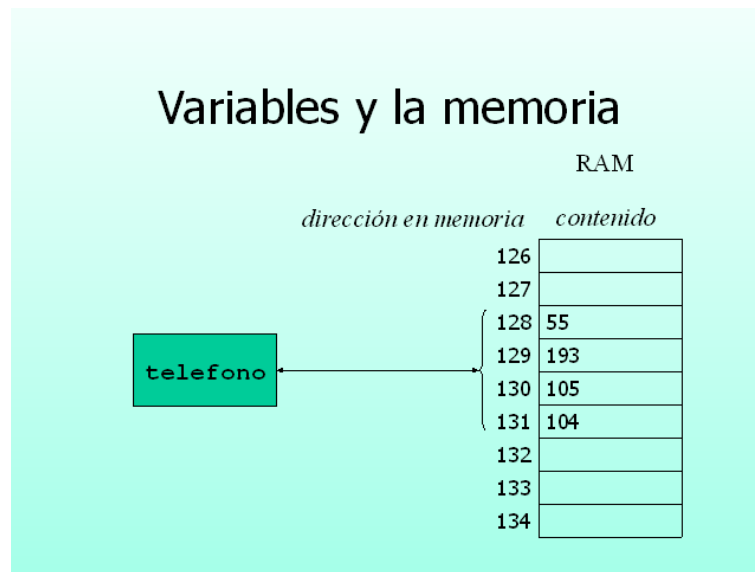


### 3.4. Variables y constantes

Una **variable** es una especie de caja o celda de memoria, que se distingue por un nombre (identificador), en la que se almacena un dato que puede ser modificado. Una variable no es más que un nombre simbólico que identifica una dirección de memoria.

Una variable es la forma de guardar un dato en un programa.

Para usar una variable en un programa hay que **declararla**. Al declarar una variable, se reserva el espacio de memoria necesario para almacenar un valor del tipo de la variable. El identificador o nombre asociado a la



variable se puede utilizar para acceder al dato almacenado en memoria y para modificarlo.

Una **constante** es una especie de caja o celda de memoria, que se distingue por un nombre (identificador), en la que se almacena un dato que **NO** puede ser modificado. Se definen igual que cuando se declara una variable y se inicializa su valor. Con la palabra reservada *final* se impide la modificación del valor almacenado.

La **declaración de variables** en java consta de dos partes: el tipo de datos que va a contener (determina la cantidad de espacio que se reserva) y el identificador o nombre. En una misma declaración se pueden declarar varias variables, siempre que sean del mismo tipo. En este caso, los nombres de las variables se separan por comas.

```

1
2 tipoDato identificador1, identificador2;
3
4 tipoDato2 identificador3;

```

### 3.4.1. Identificadores (nombres) en Java

En Java, un identificador comienza con una letra, un subrayado o un símbolo de dólar. Los siguientes caracteres pueden ser letras o dígitos. Se distinguen las mayúsculas de las minúsculas y no hay longitud máxima.

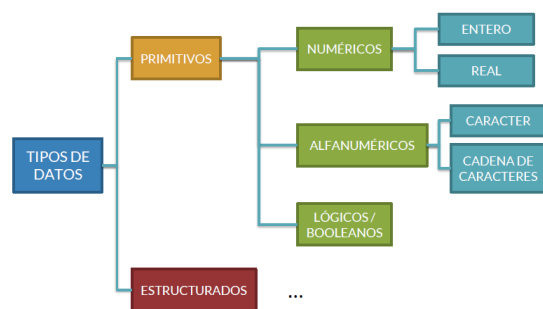
Las siguientes palabras son las palabras clave que están definidas en Java y que no se pueden utilizar como identificadores: *abstract continue for new switch boolean default goto null synchronized break do if package this byte double implements private threadsafe byvalue else import protected throw case extends instanceof public transient catch false int return true char final interface short try class finally long static void const float native super while*

Hay que tener en cuenta que a veces la letra ñ da problemas.

Los identificadores deben ser **autodescriptivos**, es decir, deben hacer referencia al significado de aquello a lo que se refieren.

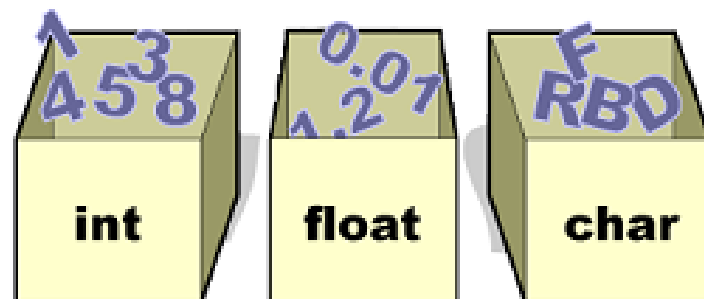
Se usa la notación camelCase, es decir, los nombres asociados a las variables se ponen en minúsculas. Cuando está formado por varias palabras, la primera palabra va en minúsculas y el resto de palabras se inician con una letra mayúscula. Para el identificador de las constantes se suelen utilizar letras mayúsculas separadas por el carácter subrayado.

### 3.4.2. Tipos de datos en Java



Los tipos de datos en Java pueden dividirse en dos categorías: simples o primitivos y compuestos o estructurados. Los simples son tipos que no se derivan de otros tipos, como los enteros, de coma flotante, booleanos y de carácter. Los tipos compuestos se basan en los tipos simples, e incluyen las cadenas, las matrices y tanto las clases como las interfaces en general.

- Tipos de datos **numéricos enteros**



Lógicos	boolean	true, false
Caracteres	char	caracteres unicode
Numéricos	byte	entero de 8 bits
	short	entero de 16 bits
	int	entero de 32 bits
	long	entero de 64 bits
	float	real de 32 bits
	double	real de 64 bits

Se usan para representar números enteros con signo. Hay cuatro tipos: **byte** (8 bits) (valores numéricos de -128 a 127), **short** (16 bits) (de -32.768 a 32.767), **int** (32 bits) (de -2.147.483.648 a 2.147.483.647) y **long** (64 bits) (sin límite).

- Tipos de datos en **numéricos reales o coma flotante**

Se usan para representar números con partes fraccionarias. Hay dos tipos de coma flotante: **float** y **double**. El primero reserva almacenamiento para un número de precisión simple de 4 bytes (valores numéricos hasta 38 cifras) y el segundo lo hace para un número de precisión doble de 8 bytes (valores numéricos hasta 308 cifras).

- Tipo de datos **boolean**

Se usa para almacenar variables que presenten dos estados que serán representados por los valores `true` y `false`.

- Tipo de datos **char**.

Se usa para almacenar caracteres Unicode simples. Debido a que el conjunto de caracteres Unicode se compone de valores de 16 bits, el tipo de datos `char` se almacena en un entero sin signo de 16 bits.

Representan un único carácter y aparecen dentro de un par de comillas simples.

- Adicionalmente también vamos a trabajar con cadenas de caracteres. No son un tipo básico. Se tratan como una clase especial llamada **`java.lang.String`**. Las cadenas se gestionan internamente por medio de una instancia de la clase `String`. Una instancia de la clase `String` es un objeto que ha sido creado siguiendo la descripción de la clase. Las cadenas son inmutables, es decir, no se cambia su contenido sino que se crea una nueva.

---

```
1
2 // Declaración de variables.
3
4     boolean resultado;
5
6     char letraMayuscula;
7
8     byte by;
9
10    short sh;
11
12    int in;
13
14    long lo;
15
```



```
16    String s;  
17  
18    float f;  
19  
20    double d;
```

---

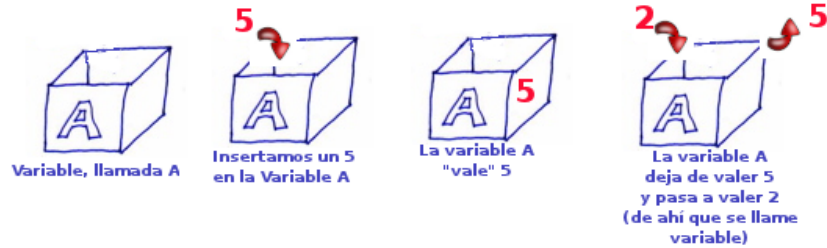


---

*En java 10 se pueden crear variables en algunos casos con **var** sin indicar el tipo*

---

### 3.5. Instrucciones de asignación



La asignación de un valor a una variable o constante se realiza a través del operador igual (=).

En el momento de la declaración es obligatorio inicializar una constante. En el caso de las variables es opcional, es decir, se puede realizar en el momento de la declaración o posteriormente dependiendo de la operación que se vaya a realizar con ella.

Ejemplos:

```

1 package declaracionvariables;
2
3 public class DeclaracionVariables {
4
5     public static void main(String[] args) {
6
7         // Declaración de una variable de tipo byte sin asignación
8         byte primeraVariable;
9
10        // Declaración de una variable de tipo short sin asignación
11        short enteroPequeño;
12
13        // Declaración de una variable de tipo int asignando valor
14        int enteroNormal=655;
15
16        // Declaración de una variable de tipo long asignando valor
17        long enteroLargo=7000;
18
19        // Asignamos valor a variables después de haberla declarado
20        primeraVariable = 1;
21        enteroPequeño = 3;
22
23        // Declaramos tres variables de tipo int en una línea
24        int numeroUno, numeroDos, numeroTres;
25

```

```
26      // Declaramos dos variables de tipo int e inicializamos sólo una
27      int n1=3, n2 = 4;
28
29      // Declaración e inicialización de una variable de tipo float
30      // sufijo f para que considere el valor float y no double que es
        lo que supone por defecto
31      float decimalesPequeño = 2.3f;
32
33      // Declaración e inicialización de una variable de tipo double
34      double decimallargo = 1234765.34;
35
36      // Declaración e inicialización de una variable booleana
37      boolean entero = false;
38
39      // Modificación del contenido de una variable
40      entero = true;
41
42      // Declaración e inicialización de una variable de tipo char
43      char letra = 's';
44
45      // Declaración e inicialización de una variable de tipo String
46      String nombre = "abc def";
47
48      // Declaración e inicialización de una constante de tipo float
49      final float PORCENTAJE_DESCUENTO = 0.25f;
50
51      // DIFERENTES SISTEMAS NUMERICOS
52      //Valor 26, en decimal
53      int decVal = 26;
54
55      //Valor 26, en hexadecimal
56      int hexVal = 0x1a;
57
58      //Valor 26, en binario
59      int binVal = 0b11010;
60
61      // Desde Java 7 se puede usar el guión bajo para delimitar las partes
        de un literal
62      // Java no lo procesará como parte del literal
63      // El sufijo L significa Long
64      long l = 3_000_000_000L;
65      long numeroTarjeta = 1234_5678_9012L;
66      }
67 }
```

### 3.6. Conversiones de tipo de datos (Castings)

En Java es posible transformar el tipo de dato de una variable u objeto en otro diferente al original. En muchos casos estos cambios se realizan de forma implícita, pero también podemos realizarlos de forma explícita. La conversión se lleva a cabo colocando el tipo destino entre paréntesis a la izquierda del valor que queremos convertir.

Ejemplo:

---

```
1 // Entrada de datos por teclado
2
3 // System.in.read siempre devuelve int
4 char c = (char)System.in.read();
5
6 Scanner sc;
7 sc = new Scanner(System.in);
8
9 // sc.nextLine() siempre devuelve String
10 opcion = Integer.parseInt(sc.nextLine());
11
12
13 System.out.println(5/9); // Muestra 0
14 System.out.println((double)5/9); // Muestra 0.5555555555555556
```

---

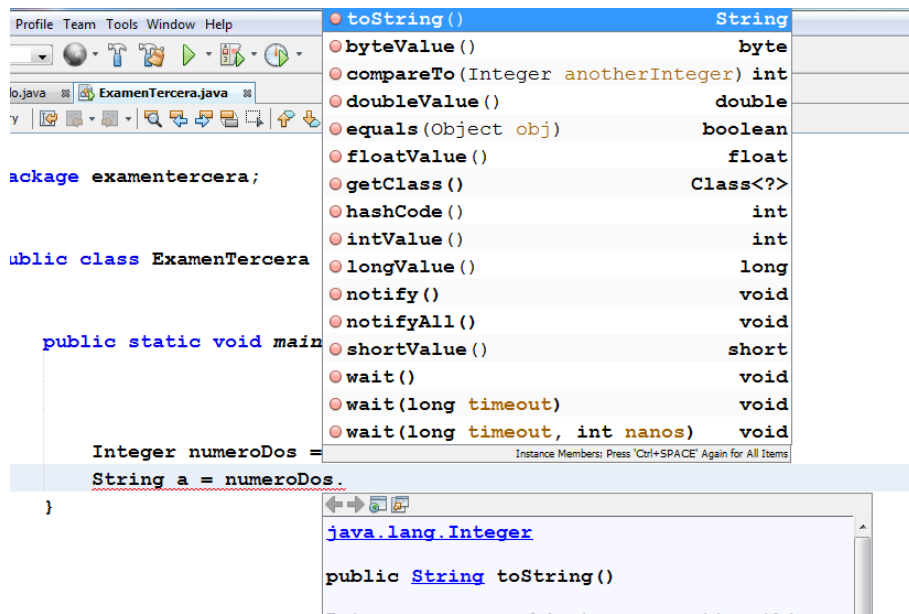
También existen métodos codificados dentro de cada una de las clases asociadas (también denominadas clases envoltorio) a los tipos de datos.

- byte – Byte
- short – Short
- int – Integer
- long – Long
- float – Float
- double – Double
- boolean – Boolean
- char – Character

---

```
1 int numeroUno = 7;
2 numeroUno. // No se puede poner nada tras el punto
3 int. // No se puede poner nada tras el punto
```

---



```

package examentercera;

public class ExamenTercera {

    public static void main(
        String[] args) {

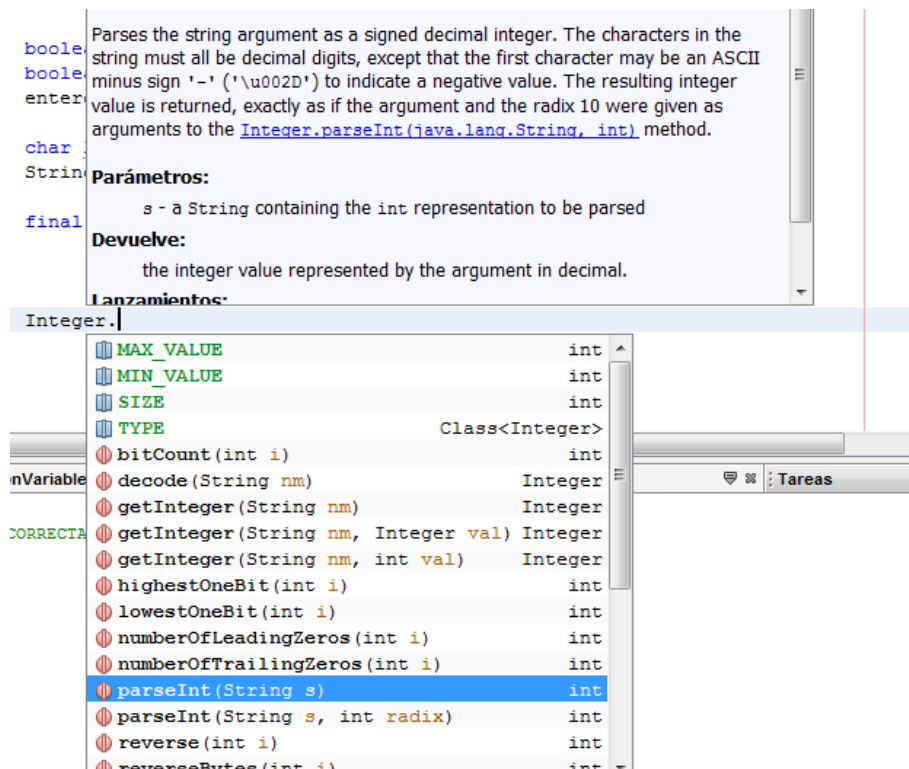
        Integer numeroDos = 12345;
        String a = numeroDos.toString();
    }
}

```

Instance Members: Press 'Ctrl+SPACE' Again for All Items

java.lang.Integer

public String toString()



```

boolean b = Integer.parseInt("12345");
boolean b = Integer.parseInt("-12345");
char c = Integer.parseInt("12345").charAt(0);
String s = Integer.parseInt("12345").toString();
final int i = Integer.parseInt("12345");

```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign '-' ('\u002D') to indicate a negative value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `Integer.parseInt(String, int)` method.

**Parámetros:**

s - a String containing the int representation to be parsed

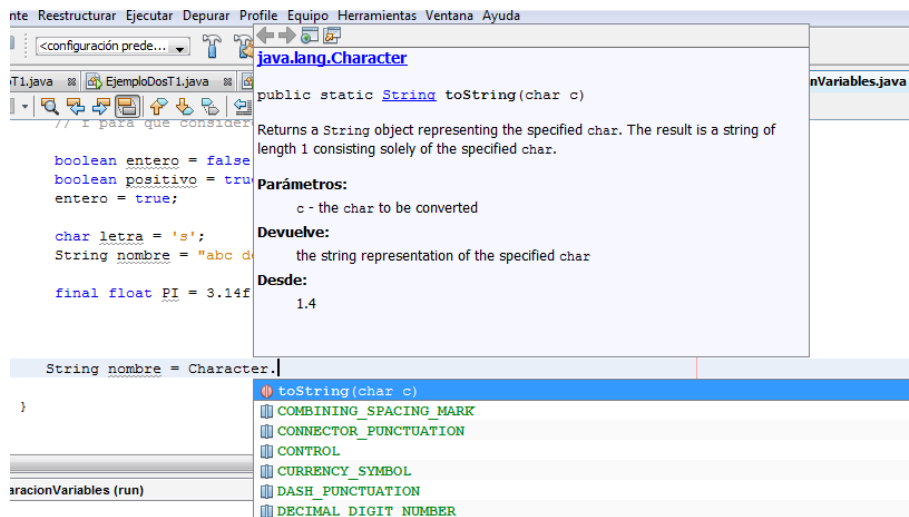
**Devuelve:**

the integer value represented by the argument in decimal.

**Lanzamientos:**

Integer.

- MAX\_VALUE int
- MIN\_VALUE int
- SIZE int
- TYPE Class<Integer>
- bitCount(int i) int
- decode(String nm) Integer
- getInteger(String nm) Integer
- getInteger(String nm, Integer val) Integer
- getInteger(String nm, int val) Integer
- highestOneBit(int i) int
- lowestOneBit(int i) int
- numberOfLeadingZeros(int i) int
- numberOfTrailingZeros(int i) int
- parseInt(String s) int
- parseInt(String s, int radix) int
- reverse(int i) int
- reverseBytes(int i) int



### 3.6.1. Ámbito de una variable

El ámbito de una variable es el período de tiempo que el programa almacena dicho valor en memoria. Como norma general, el ámbito de una variable es el bloque donde ha sido definida.

Los bloques siempre vienen delimitados por las llaves.

```
public class ExamenTercera {

    private static int v1;
    public static void main(String[] args) {
        v1 = 7;
        int v2;

    }

    {
        v2 =
        v1 = 8;
    }

}
```

cannot find symbol  
symbol: variable v2  
location: class ExamenTercera  
----  
(Alt-Enter shows hints)

### 3.7. Operadores aritméticos

Un operador es un símbolo especial que indica al compilador que debe efectuar una operación matemática o lógica.

Java reconoce los siguientes operadores aritméticos:

- + Suma
- - Resta
- \* Multiplicación
- / División
- Resto o módulo %

Para resolver operaciones más complejase n java existe toda una librería de instrucciones o funciones matemáticas llamada **Math**.

Ejemplo:

Tipo	Operador	Precedencia	Operación realizada
Prefix, postfix	-- ++	expr++ expr-- ++expr --expr	Incremento/Decremento en una unidad.
Unarios	+ -	+ -	Cambio de signo
Multiplicativos	* / %	* / %	Multiplicación, división y resto
Aditivos	+ -	+ -	Suma, resta
De movimiento	<< >> >>>	<< >> >>>	Desplazamiento a nivel de bits.

```

1 package operadoresaritmético;
2
3 public class OperadoresAritmetico {
4
5     public static void main(String[] args) {
6         //Declaración e inicialización de las variables numéricas
7         int n1 = 20;
8         int n2 = 3;
9
10        int suma = n1 + n2;
11        System.out.println("El resultado de la suma es: " + suma);
12
13
14        int resta = n1 - n2;
15        System.out.println("El resultado de la resta es: " + resta);
16
17        /*
18         * El resultado de la operación lo guardamos en una variable,
19         * sólo si nos interesa
20         */
21
22        System.out.println("El resultado de la multiplicación es: " + (n1
23            * n2));
24
25        System.out.println("El resultado de la división es: " + (n1/n2));
26
27        System.out.println("El resto de la division es: " + (n1%n2));
28
29        System.out.println("El resultado de la potencia es: " +
30            Math.pow(n1, n2));
31    }
32 }

```

El principal operador de asignación es el igual pero hay más operadores de asignación unidos a los aritméticos:

- '+='



`op1 += op2` es lo mismo que `op1 = op1 + op2`

- `'-='`

`op1 -= op2` es lo mismo que `op1 = op1 - op2`

- `'*='`

`op1 *= op2` es lo mismo que `op1 = op1 * op2`

- `'/='`

`op1 /= op2` es lo mismo que `op1 = op1 / op2`

### 3.7.1. Operadores aritméticos incrementales

Los operadores aritméticos incrementales son operadores unarios (un único operando). El operando puede ser numérico o de tipo `char` y el resultado es del mismo tipo que el operando. Estos operadores pueden emplearse de dos formas dependiendo de su posición con respecto al operando.

- `++` Incremento

`i++` primero se utiliza la variable y luego se incrementa su valor `++i` primero se incrementa el valor de la variable y luego se utiliza

- `--` decremento

---

```
1      int n1 = 5;
2      int n2 = ++n1;
3      System.out.println(n1 + " " + n2); // 6 6
4      int n3 = n1++;
5      System.out.println(n1 + " " + n3); // 7 6
```

---

### 3.7.2. Operadores a nivel de bits

Nosotros no los vamos a usar pero también existen operadores que desplazan los bits a la derecha o a la izquierda.

[https://es.wikibooks.org/wiki/Programaci%C3%B3n\\_en\\_Java/Operadores\\_de\\_bits](https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_Java/Operadores_de_bits)

---

```
1
2      //OPERADORES A NIVEL DE BITS
3      int bitmask = 0b0011;
4      int val = 0b1111;
5
```

---

```
6      int res = val & bitmask; //0011
7      System.out.print("AND ");
8      System.out.println(Integer.toBinaryString(res));
9
10     res = val ^ bitmask; //1100
11     System.out.print("OR exclusivo ");
12     System.out.println(Integer.toBinaryString(res));
13
14     res = val | bitmask; //1111
15     System.out.print("OR inclusivo ");
16     System.out.println(Integer.toBinaryString(res));
17
18     // 0b1111
19
20     res = val << 1; //11110
21     System.out.print("left shift ");
22     System.out.println(Integer.toBinaryString(res));
23
24     res = val >> 2; //0011
25     System.out.print("Signed righth shift ");
26     System.out.println(Integer.toBinaryString(res));
27
28
29     res = (-val) >> 2; //11111111111111111111111111111100
30     System.out.print("Signed righth shift ");
31     System.out.println(Integer.toBinaryString(res));
32
33     res = val >>> 1; //111
34     System.out.print("Unsigned righth shift ");
35     System.out.println(Integer.toBinaryString(res));
36
37     res = ~val; //1111111111111111111111111111110000
38     System.out.print("Inverso o complementario ");
39     System.out.println(Integer.toBinaryString(res));
```

---

### 3.8. Instrucciones de entrada/salida

En java para poder escribir (mostrar datos al usuario) se emplea el objeto **System.out** y para leer del teclado es necesario emplear **System.in**.

El objeto que se crea con System.in pertenece a la clase InputStream y uno de sus métodos es **read**, que permite leer un carácter. Siempre devuelve tipo de dato int.

---

```
1 char caracter = (char) System.in.read();
```

---

Es muy incómodo leer de letra en letra. Para leer una línea completa emplearemos el siguiente código:

---

```
1 BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
2
3 String línea = br.readLine();
```

---

De la primera manera creamos un InputStreamReader a partir de System.in y pasamos dicho InputStreamReader al constructor de BufferedReader, el resultado es que las lecturas que hagamos sobre br son en realidad realizadas sobre System.in, pero con la ventaja de que se permite leer una línea completa.

Es necesario realizar un import de java.io para poder emplear esta lectura de líneas. Además la línea del readLine puede lanzar excepciones, es por ello que hay que meterla entre instrucciones try/catch para poder gestionar el posible error.

---

```
1 String línea;
2
3 try{
4
5     BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
6
7     línea = br.readLine();
8
9 }
10 catch(Exception e){
11     e.printStackTrace();
12 }
13
14 System.out.println(línea);
```

---

Ejemplo: Sumar dos números

---

```
1 package sumardosnumeros;
2
3 import java.io.*;
4
5 public class SumarDosNumeros {
```

---

```
6
7
8 public static void main(String[] args) {
9
10     InputStreamReader isr = new InputStreamReader(System.in);
11     BufferedReader br = new BufferedReader (isr);
12
13     try
14     {
15         System.out.print("Sumando 1 : ");
16         int s1 = Integer.parseInt(br.readLine());
17
18         System.out.print("Sumando 2 : ");
19         int s2 = Integer.parseInt(br.readLine());
20
21         int suma=s1+s2;
22         System.out.println ("La suma es " + s1 + "+" + s2 +"="+ suma);
23     }
24     catch (Exception e)
25     {
26         System.out.println("Problemas");
27     }
28 }
29 }
```

---

Otra manera más simple de hacer más o menos se consigue utilizando la clase **Scanner**.

---

```
1
2 Scanner sc;
3 sc = new Scanner(System.in);
4
5 String dato = sc.nextLine();
```

---

El método `nextLine` siempre devuelve un tipo de datos `String`.

Existen formas más gráficas de llevar a cabo la entrada y salida de datos utilizando, por ejemplo, los cuadros de diálogo que nos proporciona **javax.swing.JOptionPane**. El método **showMessageDialog** nos permite mostrar una cadena de caracteres y el método **showInputDialog** recoger una cadena de caracteres.

Ejemplo:

---

```
1 package entradasalidagrafica;
2
3 import javax.swing.JOptionPane;
4
5 public class EntradaSalidaGrafica {
6
7     public static void main(String[] args) {
8         String datosEntrada;
9         datosEntrada = JOptionPane.showInputDialog(null,"Sumando uno: ");
10        int sumandoUno = Integer.parseInt(datosEntrada);
11    }
```

```

12         int sumandoDos=
            Integer.parseInt(JOptionPane.showInputDialog(null,"Sumando
            dos: "));
13
14         int resultado = sumandoUno + sumandoDos;
15
16         JOptionPane.showMessageDialog(null, "El resultado de la suma es:
            "+ resultado);
17     }
18 }

```

```

public class EntradaSalidaGrafica {

    public static void main(String[] args) {
        String datosEntrada;
        datosEntrada = JOptionPane.showInputDialog(null,"Sumando uno: ");
        int sumandoUno = Integer.parseInt(datosEntrada);

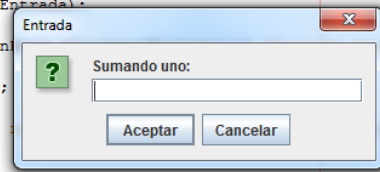
        int sumandoDos= Integer.parseInt(JOptionPane

        int resultado = sumandoUno + sumandoDos;

        JOptionPane.showMessageDialog(null, "El

    }
}

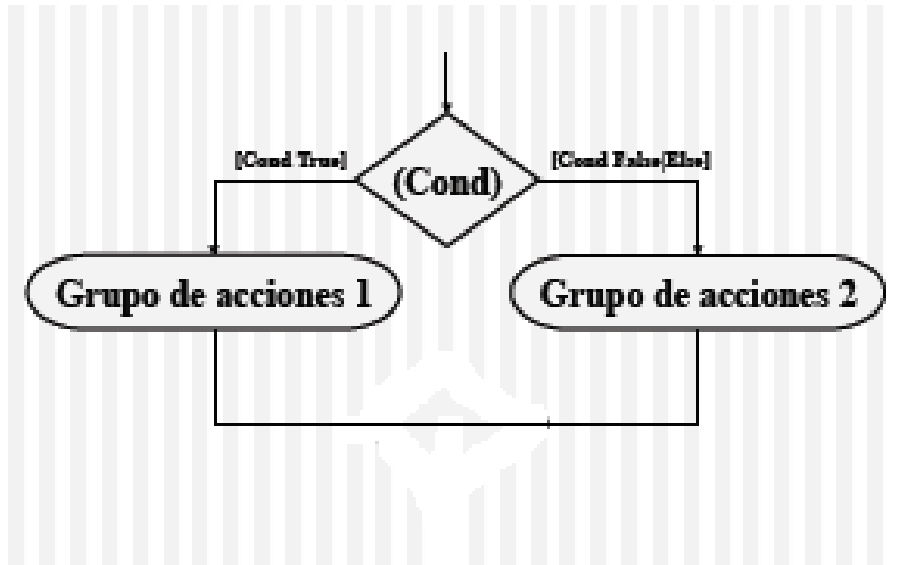
```



### 3.9. Estructuras alternativas

Estas instrucciones evalúan las expresiones lógicas (condiciones) con el objetivo de controlar la ejecución de otras instrucciones o alterar el orden de ejecución normal de las instrucciones de un programa.

Nos permiten evaluar una expresión como verdadera o falsa. En caso de que sea verdadera, se ejecuta un bloque de código. En caso de que no lo sea, dicho bloque no es ejecutado.



La estructura de selección simple en Java se realiza mediante la sentencia if (si, en inglés). La sintaxis es la siguiente :

```

1  if (condición) sentencia;
2
3  if (condición)
4      sentencia 1;
5  else
6      sentencia 2;
7
8  if (condición)
9  {
10     sentencias;
11 }
12 else
13 {
14     sentencias;
15 }
  
```

### 3.9.1. Operadores relacionales

Los operadores relacionales permiten expresar condiciones comparando variables según relación de igualdad/desigualdad o relación mayor/menor. Devuelven siempre un valor boolean (true o false).

Operador	Descripción
==	Devuelve verdadero si ambos valores son verdaderos
!=	Devuelve el valor inverso a ==
>	Devuelve verdadero si el valor de la izquierda es mayor estricto que el de la derecha.
>=	Devuelve verdadero si el valor de la izquierda es mayor o igual que el de la derecha.
<	Devuelve verdadero si el valor de la izquierda es menor estricto que el de la derecha.
<=	Devuelve verdadero si el valor de la izquierda es menor o igual que el de la derecha.

Los operadores lógicos (and &&, or ||) nos permiten construir expresiones lógicas formadas por más de una condición.

Operador	Descripción
!	Realiza la negación del operando
&&	Realiza la operación lógica condicional AND
	Realiza la operación lógica condicional OR
?:	(Ternario) Si el primer operando es verdadero, devuelve el valor del segundo; en otro caso, devuelve el tercero

Si dos operadores se encuentran en la misma expresión, el orden en el que se evalúan puede determinar el valor de la expresión. En la siguiente tabla se muestra el orden o prioridad en el que se ejecutan los operadores que se encuentren en la misma sentencia. Los operadores de la misma prioridad se evalúan de izquierda a derecha dentro de la expresión.

También existe el operador ! que niega el operando que se le pasa.

Prior.	Operador	Tipo de operador	Operación
1	++	Aritmético	Incremento previo o posterior (unario)
	--	Aritmético	Incremento previo o posterior (unario)
	+, -	Aritmético	Suma unaria, Resta unaria
	~	Integral	Cambio de bits (unario)
	!	Booleano	Negación (unario)
2	(tipo)	Cualquiera	
3	*, /, %	Aritmético	Multiplicación, división, resto
4	+, -	Aritmético	Suma, resta
	+	Cadena	Concatenación de cadenas
5	<<	Integral	Desplazamiento de bits a izquierda
	>>	Integral	Desplazamiento de bits a derecha con inclusión de signo
	>>>	Integral	Desplazamiento de bits a derecha con inclusión de cero
6	<, <=	Aritmético	Menor que, Menor o igual que
	>, >=	Aritmético	Mayor que, Mayor o igual que
	instanceof	Objeto, tipo	Comparación de tipos
7	==	Primitivo	Igual (valores idénticos)
	!=	Primitivo	Desigual (valores diferentes)
	===	Objeto	Igual (referencia al mismo objeto)
	!==	Objeto	Desigual (referencia a distintos objetos)
8	&	Integral	Cambio de bits AND
	&&	Booleano	Producto booleano
9	^	Integral	Cambio de bits XOR
	^^	Booleano	Suma exclusiva booleana
10		Integral	Cambio de bits OR
		Booleano	Suma booleana
11	&&&	Booleano	AND condicional
12		Booleano	OR condicional
13	? :	Booleano, cualquiera, cualquiera	Operador condicional (ternario)
14	=	Variable, cualquiera	Asignación
	*, /=, %=		Asignación con operación
	+=, -=		
	<<=, >>=		
	>>>=		
	&=, ^=,  =		



El operador ternario ? nos permite escribir una sentencia if de forma comprimida.

---

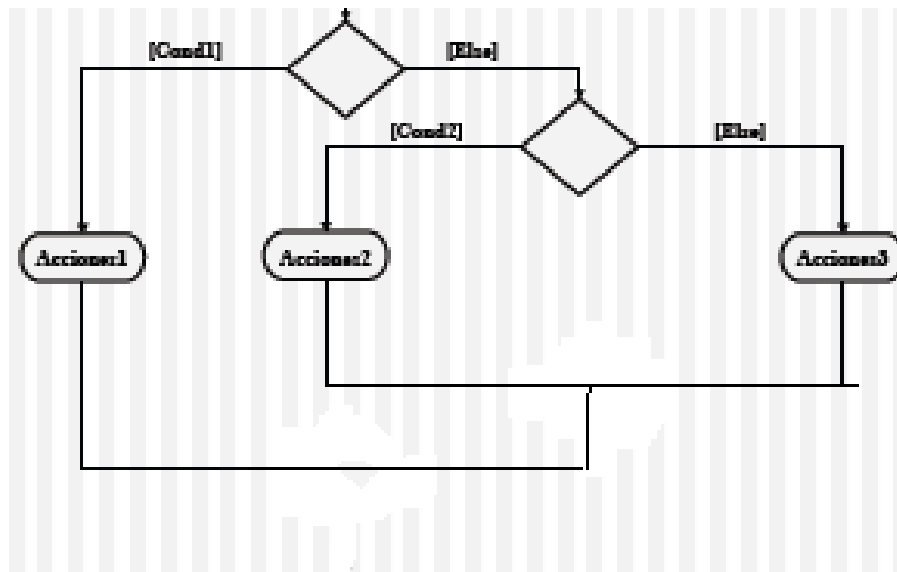
```
1 resultado = (condicion)?valor1:valor2;
```

---

```
1 if (x>y)
2     mayor = x;
3 else
4     mayor = y;
5
6 mayor = (x>y)?x:y;
```

---

Dentro del bloque de sentencias del if (o del bloque else) puede encontrarse otra sentencia if (if anidados).



Ejemplos;

```

1 package ejemplosif;
2
3 public class EjemplosIf {
4
5
6     public static void main(String[] args) {
7         //Entrada de datos
8         int n1= Integer.parseInt(JOptionPane.showInputDialog("Teclea un
9             número"));
10        int n2= Integer.parseInt(JOptionPane.showInputDialog("Teclea otro
11            número"));
12
13        //if sin else y con sólo una sentencia.
14        if (n1 > n2)
15            JOptionPane.showMessageDialog(null, "El primer número es mayor
16                que el segundo");
17
18        JOptionPane.showMessageDialog(null, "Primer if finalizado");
19
20        //if sin else y con varias sentencias
21        if (n1 > n2)
22        {
23            JOptionPane.showMessageDialog(null, "El primer número es
24                mayor que el segundo");
25            JOptionPane.showMessageDialog(null, "El primer número es " +
26                n1);
27            JOptionPane.showMessageDialog(null, "El segundo número es " +
28                n2);
29        }
30    }
31 }
  
```

```
24 JOptionPane.showMessageDialog(null, "Segundo if finalizado");
25
26 //if con else y una sentencia
27 if (n1 > n2)
28     JOptionPane.showMessageDialog(null, "El primer número es mayor
29         que el segundo");
30 else
31     JOptionPane.showMessageDialog(null, "El primer número no es
32         mayor que el segundo");
33
34 JOptionPane.showMessageDialog(null, "Tercer if finalizado");
35
36 //if con else y varias sentencias
37 if (n1 > n2)
38 {
39     JOptionPane.showMessageDialog(null, "El primer número es
40         mayor que el segundo");
41     JOptionPane.showMessageDialog(null, "El primer número es el "
42         + n1);
43 }
44 else
45 {
46     JOptionPane.showMessageDialog(null, "El primer número no es
47         mayor que el segundo");
48     JOptionPane.showMessageDialog(null, "El segundo número es el "
49         + n2);
50 }
51
52 javax.swing.JOptionPane.showMessageDialog(null, "Cuarto if
53     finalizado");
54
55 //if con else y varias sentencias sólo en una rama
56 if (n1 > n2)
57 {
58     JOptionPane.showMessageDialog(null, "El primer número es
59         mayor que el segundo");
60     JOptionPane.showMessageDialog(null, "El primer número es el "
61         + n1);
62 }
63 else
64     JOptionPane.showMessageDialog(null, "El primer número no es
65         mayor que el segundo");
66
67 JOptionPane.showMessageDialog(null, "Quinto if finalizado");
68
69 //Varias condiciones
70 if (n1 > n2 && n1 != 3)
71     JOptionPane.showMessageDialog(null, "El primer número es
72         mayor que el segundo y no es el tres");
73
74 if (n1 > n2 || n1 != 3)
75     JOptionPane.showMessageDialog(null, "El primer número es mayor
76         que el segundo o no es el tres");
77
78 if (n1 > n2 && n1 != 3 || n1==5)
79     JOptionPane.showMessageDialog(null, "Tres condiciones");
80
81 //if anidados
```

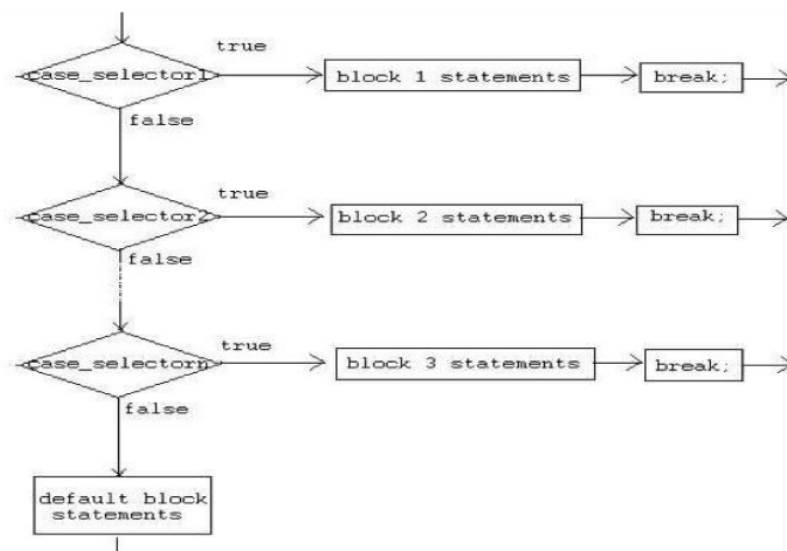
```

71         if (n1 > n2)
72             if (n1 == 3)
73                 JOptionPane.showMessageDialog(null,"Dentro del segundo
74                     if");
75             else
76                 JOptionPane.showMessageDialog(null,"Dentro del segundo
77                     if en el else");
78         else
79             if (n1 == 3)
80                 JOptionPane.showMessageDialog(null,"Dentro del primer
81                     else y segundo if");
82     }
83 }

```

### 3.9.2. Sentencia alternativa múltiple

La instrucción **switch** es una forma de expresión de un anidamiento múltiple de instrucciones **if ... else**. Su uso no es estrictamente necesario ya que siempre podrá ser sustituida por el uso de **if**. No obstante, nos resultará útil ya que introduce mayor claridad en el código.



```

1  switch( variable )
2  {
3      case valor1:
4          sentencial;
5          ...
6          break;
7          ...
8      case valorN:
9          sentenciaN;

```

```
10     ...
11     break;
12 default:
13     sentencia;
14     ...
15     break;
16 }
```

---

```
1  switch (expresión) {
2
3      // En los tres casos quiero hacer lo mismo
4      case valor1: case valor2: case valor3:
5          instrucciones;
6          break;
7
8      case valor4:
9          instrucciones;
10         break;
11     .
12     .
13     .
14     default:
15         sentencias;
16         break;
17 }
18
19 // En varios cases quiero hacer lo mismo
20 switch(x){
21     case 2:
22     case 4:
23     case 6:
24     case 8:
25     case 10:
26         System.out.println("x es es numero par");
27         break;
28     case 1:
29     case 3:
30     case 5:
31     case 7:
32     case 19:
33         System.out.println("x es es numero impar");
34         break;
35 }
36 }
```

---

### Ejemplos;

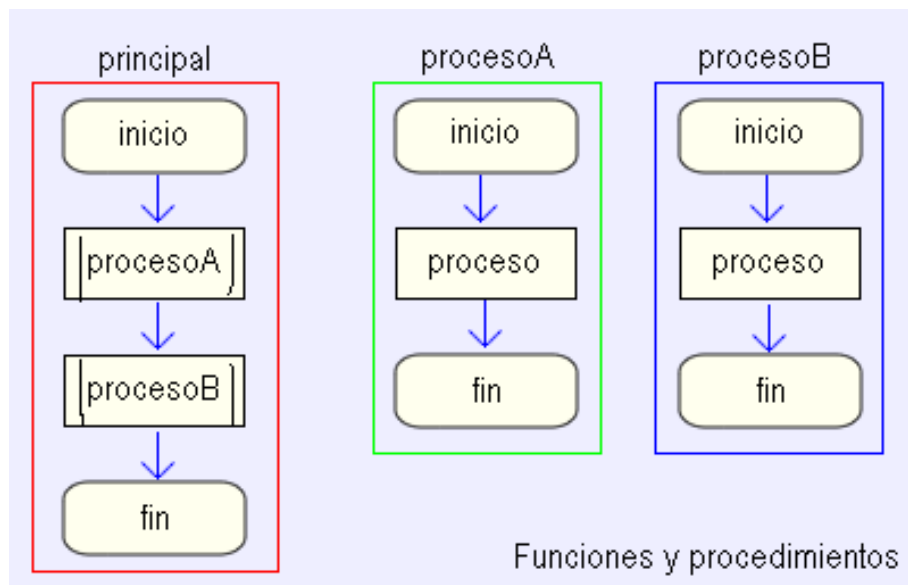
---

```
1  package ejemploswitch;
2
3  import javax.swing.JOptionPane;
4
5  public class EjemploSwitch {
6
7      public static void main(String[] args){
8
9          int numero=Integer.parseInt(JOptionPane.showInputDialog("Teclea
10             un número"));
11     }
```

```
11      switch(numero){
12
13          case 1:
14              JOptionPane.showMessageDialog(null,"Has tecleado el número
15                  uno");
16              break;
17          case 2:
18              JOptionPane.showMessageDialog(null,"Has tecleado el número
19                  dos");
20              break;
21          case 3:
22              JOptionPane.showMessageDialog(null,"Has tecleado el número
23                  tres");
24              break;
25          case 4:
26              JOptionPane.showMessageDialog(null,"Has tecleado el número
27                  cuatro");
28              break;
29          case 5:
30              JOptionPane.showMessageDialog(null,"Has tecleado el número
31                  cinco");
32              break;
33          default:
34              JOptionPane.showMessageDialog(null,"Has tecleado un número
35                  no comprendido entre 1 y 5");
36              break;
37      }
38  }
```

---

### 3.10. Funciones, procedimientos o métodos

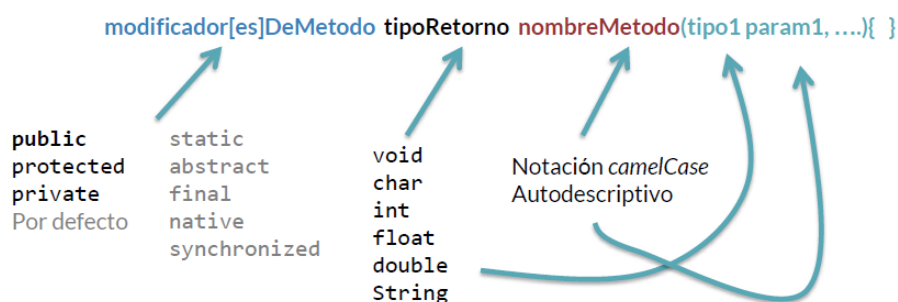


Una función es una parte de un programa que realiza una tarea determinada.

Las funciones son utilizadas para descomponer grandes problemas en tareas simples y para implementar operaciones que son comúnmente utilizadas durante un programa y de esta manera reducir la cantidad de código.

#### 3.10.1. Definición de una función

**visibilidad** **static** **tipoDevolucion** **nombre**(**parametros o argumentos**)



- modvisib o visibilidad: public, private, protected

MODIFICADOR	CLASE	PACKAGE	SUBCLASE	TODOS
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
No especificado	Sí	Sí	No	No
private	Sí	No	No	No

- tipodatodevolucion: Tipo de dato que devuelve la función. Sino devuelve ninguno se escribe void. Si devuelve en el cuerpo de la función habrá que usar la sentencia **return**.
- Parámetros: Lista de datos de entrada que se pasan a la función junto con sus tipos.  
Paso de un parámetro de tipo int: (int dia)  
Paso de dos parámetros (int num, float sueldo)  
Sin parámetros ()



*El static no es obligatorio, pero de momento lo utilizaremos.*

### 3.10.2. Utilización o llamada a la función

Llamada a una función que no devuelve ningún valor y que tampoco recibe parámetros.

```
nombreFuncion();
```

Llamada a una función que no devuelve ningún valor y que recibe parámetros.

```
nombrefuncion(valores);
```

Llamada a una función que devuelve un valor y que no recibe parámetros.

```
variable = nombreFuncion();
```

Llamada a una función que devuelve un valor y que recibe parámetros.

```
variable = nombreFuncion(valores);
```



### 3.10.3. Ejemplos

- No pasamos nada y no devuelve nada.

---

```
1
2 //Llamada
3 obtenerNombre();
4
5 //Definición
6 public static void obtenerNombre()
7 {
8     .....
9 }
```

---

- Pasamos valores y no devuelve nada.

---

```
1
2 // Llamada
3 obtenerNombre(i,f,c);
4
5
6 private static void obtenerNombre(int i, float f, char c)
7 {
8     .....
9 }
```

---

- No pasamos ningún dato y nos devuelve un valor.

---

```
1
2
3
4 int rdo=obtenerNumero();
5
6
7
8 private static int obtenerNumero()
9 {
10     int v1=Integer.parseInt(JOptionPane.showInputDialog("Teclea un
        número"));
11     .....
12
13
14     return v1;
15 }
```

---

- Necesita parámetros y devuelve un valor.

---

```
1
2
3 int rdo=obtenerNumero(i,f,c);
4
5 private static int obtenerNumero(int i, float f, char c)
6 {
7     int v4= 0;
8     // Operaciones con i, f y c
9 }
```

---

```
10         return v4;
11     }
```

---

Cuidado con el **ámbito de las variables**. Recordad que sólo existen en el bloque de código en el que han sido declaradas.

---

```
1  public class Prueba{
2
3      private static int variableGlobal=10;
4
5      public static void metodoPrueba (){
6
7          int variableMetodo=40;
8
9          int resultado = variableMetodo + variableGlobal;
10
11         System.out.println("El resultado de la suma es" + resultado);
12
13         // variableLocal1 aquí no existe.
14     }
15
16     public static void main(String[] args) {
17
18         int variableLocal1=20;
19
20         if (variableGlobal==10)
21         {
22             int variableLocal2=30;
23
24             System.out.println("El valor de las distintas variables es: "
25                               + variableGlobal + variableLocal1 + variableLocal2);
26         }
27
28         System.out.println("El valor de las distintas variables es: " +
29                           variableGlobal + variableLocal1);
30         // variableLocal2 ya no existe
31         metodoPrueba();
32     }
33 }
34 }
```

---

#### 3.10.4. Varargs

Desde hace algunas versiones, Java incluye la opción de usar varargs para indicar que un método recibirá un número arbitrario de argumentos de un tipo. Estos son útiles cuando no sabemos a priori la cantidad de argumentos que recibiremos. Para usarlos usaremos la sintaxis de tres puntos seguidos (...), justo después del tipo de dato, y separados por un espacio del nombre del argumento.

Dentro del método, un varargs se trata igual que un array, es decir, se va por posiciones. (En el tema siguiente hablaremos de los arrays)

Un método que reciba varios argumentos de diferentes tipos, y además, un varargs , debe incluir este como el último en orden de recepción. De otra forma, sería imposible identificar el número de argumentos recibidos mediante el varargs .

---

```
1 public int sumar(int... numero)
2 {
3     int resultado = 0;
4     for(int i = 0; i < numero.length; i++)
5     {
6         resultado += numero[i];
7     }
8     return resultado;
9 }
10
11
12 System.out.println(sumar(3,5));
13 System.out.println(sumar(3,5,7));
14 System.out.println(sumar(3,5,6,5));
```

---

### 3.10.5. Recursividad

Es una técnica utilizada en programación que nos permite que un bloque de instrucciones se ejecute un cierto número de veces (el que nosotros determinemos).

Una función se llama a si misma hasta que se cumpla una condición.

---

```
1 public int factorial(int n) {
2     if (n == 0) {
3         return 1;
4     } else {
5         return n * factorial(n - 1);
6     }
7 }
```

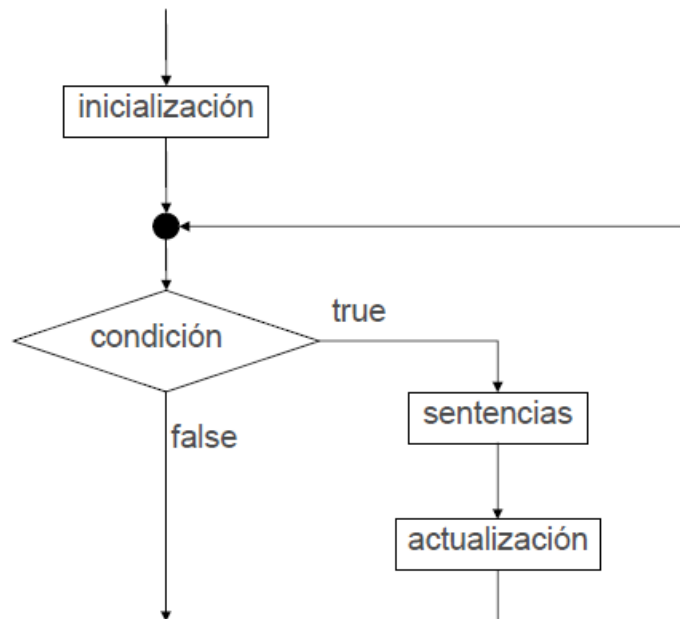
---

## 3.11. Estructuras repetitivas

Los bucles, iteraciones o sentencias repetitivas modifican el flujo secuencial de un programa permitiendo la ejecución reiterada de una sentencia o grupo de sentencias. En Java hay tres tipos de bucles: for, while y do-while.

### 3.11.1. For

Nos permite repetir un bloque de código un **número conocido a priori de veces**.




---

```

1  for({valor inicial};{condición de término};{modificación del valor
    inicial})
2  {
3    sentencias;
4  }
  
```

---

Ejemplo:

---

```

1  package ejemplofor;
2
3  public class EjemploFor {
4
5      public static void main(String[] args) {
  
```

---

```

6      int x;
7      // valor inicial x = 0
8      // condición de término x < 10
9      // modificación del valor inicial x++
10     for(x=0;x<10;x++)
11         System.out.println(x);
12 }
13 }

```

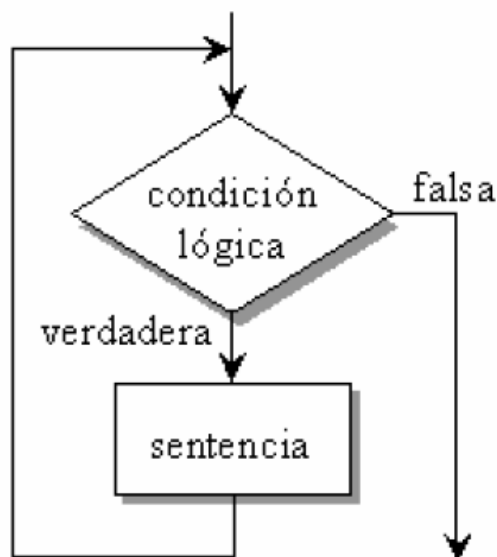


Cuando trabajemos con arrays podremos usar una variante denominada *forEach*

### 3.11.2. While

Nos permite repetir la ejecución de un bloque de sentencias de **0 a N veces**. La repetición se realiza durante un número indeterminado de veces, mientras una expresión sea cierta.

Una de las sentencias del cuerpo del bucle debe modificar alguna de las variables de la condición, para que, en alguna ocasión, la expresión sea falsa.



```

1 while ( condición ) sentencia ;
2
3 while ( condición )
4 {
5     sentencias ;

```

```
6 .....  
7 }
```

---

Ejemplo:

```
1 package ejemplowhile;  
2  
3 public class EjemploWhile {  
4  
5     public static void main(String[] args) {  
6         int x=0;  
7         while (x < 10)  
8         {  
9             System.out.println(x);  
10            x++;  
11        }  
12    }  
13 }
```

---

### 3.11.3. Do while

Nos permite repetir la ejecución de un bloque de sentencias de **1 a N veces**. La condición, a diferencia de la estructura while, se evalúa al final del bucle.

```
1 do  
2 {  
3     sentencias;  
4     .....  
5 }  
6 while ( condición );
```

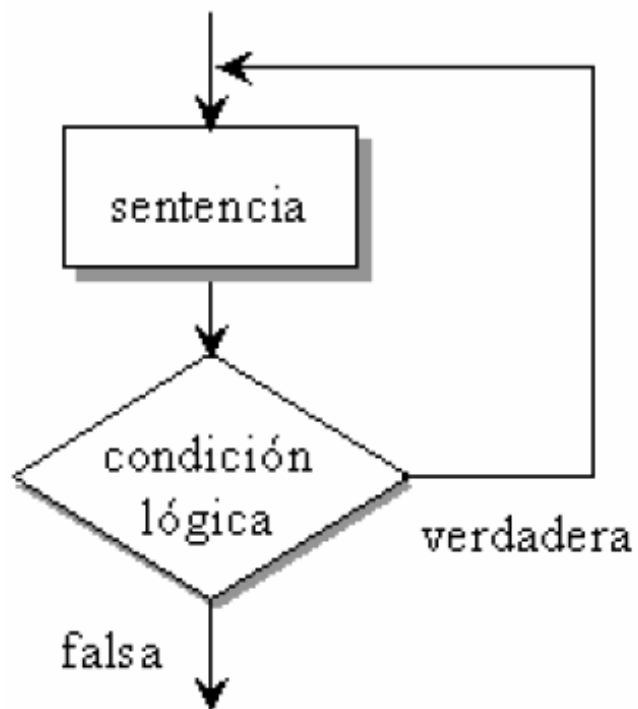
---

Ejemplo:

```
1 package ejemplodowhile;  
2  
3 public class EjemploDoWhile {  
4  
5     public static void main(String[] args) {  
6         int x=0;  
7         do  
8         {  
9             System.out.println(x);  
10            x++;  
11        }  
12        while (x < 10);  
13    }  
14 }
```

---

En cualquiera de los tres casos (for, while, do while) fijaros que se acaba cuando deja de cumplirse la condición.



## 3.12. String

Java crea un String con un literal encerrado entre dos comillas dobles.

---

```
1 String saludo = "Hola Mundo";
```

---

Los objetos de tipo String son inmutables. Todas las operaciones que hagamos con ellos, darán como resultado uno nuevo, no la modificación del anterior (no es muy eficiente).

Dentro de la clase String hay métodos varios que nos permitirán realizar distintas operaciones:

Concatenar es yuxtaponer un String a continuación de otro. Se puede hacer con el operador + y el método **concat()**.

---

```
1 "Cadena " + "concatenada"
2 "Mi nombre es ".concat("Pepe")
```

---

La longitud de un String (**length()**) es el número de caracteres que lo forman. Cuenta también los espacios en blanco, tabuladores, signos de puntuación, etc...

---

```
1 String saludo = "Hola Mundo!";
2 System.out.println(saludo.length()); //Debe imprimir 11
```

---

Los métodos **toLowerCase()** y **toUpperCase()** transforman una cadena completa a mayúsculas o minúsculas.

---

```
1 System.out.println(cadena.toLowerCase());
```

---

Las **cadenas formateadas** nos permiten insertar valores dentro de una cadena a posteriori. Evitan concatenaciones tediosas.

---

```
1 String nombre = "Pepe";
2 String apellidos = "Pérez";
3 String mensaje = "¿qué tal estás?";
4
5 System.out.println("Hola, soy " + nombre + " " + apellidos + ", y quiero
    saludarte diciéndote " + mensaje);
6
7 String str = String.format("Hola, soy %s %s y quiero saludarte diciéndote
    %s", nombre, apellidos, mensaje);
8 System.out.println(str);
9
10 System.out.printf("Hola, soy %s %s y quiero saludarte diciéndote %s",
    nombre, apellidos, mensaje);
```

---

Para **comparar cadenas** existen métodos como **compareTo()**, **compareToIgnoreCase()**, **equals()**, ....



---

```
1      String cadenaUno = "Hola";
2      String cadenaDos = "hola";
3      System.out.println(cadenaUno.equals(cadenaDos)); // false
4      System.out.println(cadenaUno.equalsIgnoreCase(cadenaDos)); // true
5      System.out.println(cadenaUno.compareTo(cadenaDos)); // -32
6      System.out.println(cadenaUno.compareToIgnoreCase(cadenaDos)); //0
7      cadenaDos = "Hola";
8      System.out.println(cadenaUno.equals(cadenaDos)); // true
9      System.out.println(cadenaUno.equalsIgnoreCase(cadenaDos)); // true
10     System.out.println(cadenaUno.compareTo(cadenaDos)); // 0
11     System.out.println(cadenaUno.compareToIgnoreCase(cadenaDos)); // 0
12
13     String quijote = "En un lugar de La Mancha";
14     System.out.println(quijote.endsWith("La Mancha")); //true
15     System.out.println(quijote.startsWith("En un lugar")); // true
16     System.out.println(quijote.contains("lugar")); // true
```

---

Más métodos de la clase String <https://docs.oracle.com/javase/8/docs/api/index.html>

### 3.13. StringBuilder

La clase `StringBuilder` es similar a la clase `String` en el sentido de que sirve para almacenar cadenas de caracteres. No obstante, presenta algunas diferencias relevantes:

- Su tamaño y contenido pueden modificarse. Los objetos de éste tipo son mutables.
- Debe crearse con alguno de sus constructores asociados. No se permite instanciar directamente a una cadena como sí permiten los `String`.

```

1
2 // Construye un StringBuilder vacío y con una capacidad por defecto de 16
   caracteres.
3 StringBuilder s = new StringBuilder();
4
5
6 // Se le pasa la capacidad (número de caracteres) como argumento.
7 StringBuilder s = new StringBuilder(55);
8
9 // Construye un StringBuilder en base al String que se le pasa como
   argumento.
10 StringBuilder s = new StringBuilder("hola");

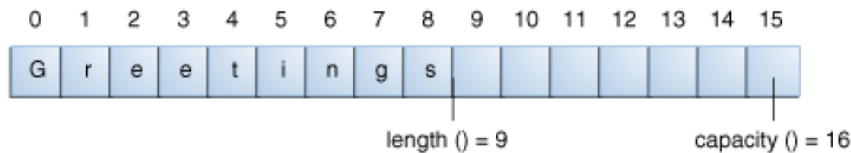
```

#### 3.13.1. Tamaño y capacidad

```

1 // Crea un objeto vacío, con capacidad para 16 caracteres
2 StringBuilder sb = new StringBuilder();
3
4 // Añade 9 caracteres a partir de la posición cero.
5 sb.append("Greetings");

```



Algunos métodos, como `append()`, pueden aumentar la capacidad de nuestro `StringBuilder`.

Más métodos de la clase `StringBuilder` <https://docs.oracle.com/javase/8/docs/api/index.html>

## 3.14. Tratamiento de fechas y horas

Trabajar con fechas en java no es algo del otro mundo ni tiene demasiadas complicaciones, pero la cantidad de formas que hay para hacerlo puede confundirnos.

### 3.14.1. La clase java.util.Date

<https://docs.oracle.com/javase/8/docs/api/java/sql/Date.html>

java.util.Date representa un instante de tiempo específico, con precisión de milisegundos.

```
1
2 // Lunes 21 de mayo de 2018
3
4 public static void main(String[] args) {
5     Date fechaUno = new Date();
6     System.out.println("Hoy es: " + fechaUno); //Hoy es: Mon May 21
7         10:36:59 CEST 2018
8     Date fechaDos = new Date();
9     // Unos milisegundos más que la fechaUno
10    System.out.println(fechaDos); // Mon May 21 10:36:59 CEST 2018
11
12    // Comparar fechas
13    // Son diferentes.
14    if (fechaUno.compareTo(fechaDos)==0)
15        System.out.println("Son iguales");
16    else
17        System.out.println("Son diferentes");
18
19    // Periodo de tiempo (milisegundos) entre dos fechas.
20    System.out.println(fechaUno.getTime() - fechaDos.getTime()); //
21        -47
22
23    if (fechaUno.before(fechaDos))
24        System.out.println("Fecha uno es anterior");
25    else
26        System.out.println("Fecha dos es anterior");
27
28    // Convertir un String en una fecha
29    String fechaString = JOptionPane.showInputDialog("Teclea una
30        fecha dd-MM-yyyy");
31    SimpleDateFormat formatter = new SimpleDateFormat("dd-MM-yyyy");
32
33    try
34    {
35        Date fecha = formatter.parse(fechaString);
36        System.out.println(fecha);
37        // Fri Dec 12 00:00:00 CET 1980 tecleando 12-12-1980
38        // Fri Jan 12 00:00:00 CET 2001 tecleando 12-13-2000
39    }
40    catch(Exception e )
41    {
42    }
```

```
39         System.out.println("Problemas al convertir String en Date");
40         // Escribiendo aaaa
41     }
42 }
43 }
```

---

### 3.14.2. La clase Calendar

Según la documentación del API de java, la clase Calendar es una clase abstracta base para convertir entre un objeto de tipo Date (java.util.Date) y un conjunto de campos enteros como YEAR (año), MONTH (mes), DAY (día), HOUR (hora), etc. Una subclase de Calendar representa una fecha de acuerdo a las reglas de un calendario específico. La plataforma provee una subclase concreta de Calendar: GregorianCalendar.

La clase Calendar tiene mucho del comportamiento que esperaríamos de la clase java.util.Date, es decir, cuando obtenemos una instancia de la clase Calendar obtenemos un instante de tiempo específico con gran precisión similar a lo que obtenemos con la clase date. Los milisegundos juegan un papel fundamental en esta clase; pero el verdadero sentido de la clase Calendar no es obtener un instante de tiempo sino **extraerle datos**.

La clase java.util.Date tiene métodos que permiten obtener el año, mes y día, pero estos métodos están obsoletos precisamente por que para eso existe Calendar y de hecho cuando usamos el método getYear() de la clase java.util.Date esta recurre a las funcionalidades que posee la clase Calendar.

El método getInstance() de la clase nos devuelve una subclase de Calendar con el tiempo ajustado a la hora actual, y usamos el método set() para forzarlo a tomar la fecha deseada.

Por default, el Calendar opera en modo permisivo (lenient). En este modo, el calendario permite que le seteen valores inválidos, y los ajusta a la fecha real. Por ejemplo, setear el día 32 al mes de enero nos ubicará en realidad en el primero de febrero.

En el modo no-permisivo (non-lenient) el Calendar tirará una java.lang.IllegalArgumentException cuando se intente setear un valor fuera de rango.

*calendar.setLenient(false);* deshabilita el modo permisivo.

---

```
1 public static void main(String[] args) {
2     Calendar fechaUno = Calendar.getInstance();
3     System.out.println("Hoy es: " + fechaUno); //Hoy es:
        java.util.GregorianCalendar[time=1526892563583,
        areFieldsSet=true,areAllFieldsSet=true,lenient=true,
        zone=sun.util.calendar.ZoneInfo[id="Europe/Paris",offset=3600000,dstSavings=3600000,
        useDaylight=true,transitions=184,lastRule=java.util.SimpleTim.....
```

```
4
5      System.out.println("Hoy abreviado: " + fechaUno.getTime());
6      // Hoy abreviado: Mon May 21 10:49:23 CEST 2018
7
8      Calendar fechaDos = Calendar.getInstance();
9      fechaDos.set(2015, 2, 16);
10     System.out.println("Segunda fecha " + fechaDos.getTime());
11     // Segunda fecha Mon Mar 16 10:49:23 CET 2015
12
13     System.out.println("Por partes: " +
14         fechaDos.get(Calendar.DAY_OF_MONTH) +
15         "-" + fechaDos.get(Calendar.MONTH) +
16         " " + fechaDos.get(Calendar.YEAR));
17
18     // Por partes: 16-2 2015
19
20     // Comparar fechas
21     if (fechaUno.compareTo(fechaDos)==0)
22         System.out.println("Son iguales");
23     else
24         System.out.println("Son diferentes");
25
26     // Periodo de tiempo (dias) entre dos fechas.
27     // Un día 86,400,000 milisegundos
28     System.out.println("Días de diferencia: " +
29         (fechaUno.getTimeInMillis() - fechaDos.getTimeInMillis())/
30         86400000 );
31     // Días de diferencia: 1161
32
33     if (fechaUno.before(fechaDos))
34         System.out.println("Fecha uno es anterior");
35     else
36         System.out.println("Fecha dos es anterior");
37
38     // Convertir un String en una fecha Calendar
39     try
40     {
41         String fecha = JOptionPane.showInputDialog("Teclea una
42             fecha");
43         Calendar cal = Calendar.getInstance();
44         SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
45         cal.setTime(sdf.parse(fecha));
46         System.out.println(cal.getTime());
47         // Tue Dec 12 00:00:00 CET 2000 tecleando 12-12-2000
48         // Fri Jan 12 00:00:00 CET 2001 tecleando 12-13-2000
49         // El modo permisivo está habilitado
50     }
51     catch(Exception e)
52     {
53         System.out.println("Problemas con la conversión");
54     }
55
56     // Sumar datos a un fecha
57     fechaUno.add(Calendar.HOUR, 2);
58     fechaUno.add(Calendar.YEAR, 1);
59     System.out.println(fechaUno.getTime());
60     fechaUno.add(Calendar.YEAR, -10);
61     System.out.println(fechaUno.getTime());
```

```
59     }  
60 }
```

---

### 3.14.3. Las clases `LocalDate`, `LocalTime`, etc...

A partir de Java 8, la nueva Date API es más amigable y está basada en el proyecto Joda-Time la cual fue adoptada como JSR-310.

En el paquete `java.time`, se puede encontrar las nuevas clases del Date API como `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Duration` y `Period`, de las cuales hablaremos a continuación.

#### **LocalDate**

La clase `LocalDate` representa una fecha sin tener en cuenta la hora. Haciendo uso del método `of(int year, int month, int dayOfMonth)`, se puede crear un `LocalDate`.

---

```
1 LocalDate date = LocalDate.of(1989, 11, 11); //1989-11-11  
2 System.out.println(date.getYear()); //1989  
3 System.out.println(date.getMonth()); //Noviembre  
4 System.out.println(date.getDayOfMonth()); //11
```

---

Se puede capturar el `LocalDate` actual se puede usar el método `now()`:

---

```
1 LocalDate date = LocalDate.now();
```

---

#### **LocalTime**

De manera similar se tiene `LocalTime` que representa un tiempo determinado. Haciendo uso del **método** `of()`, esta clase puede crear un `LocalTime` teniendo en cuenta la hora y minuto; hora, minuto y segundo y finalmente hora, minuto, segundo y nanosegundo.

---

```
1 LocalTime time = LocalTime.of(5, 30, 45, 35); //05:30:45:35  
2 System.out.println(time.getHour()); //5  
3 System.out.println(time.getMinute()); //30  
4 System.out.println(time.getSecond()); //45  
5 System.out.println(time.getNano()); //35
```

---

Para capturar el `LocalTime` actual se puede usar el **método** `now()`:

---

```
1 LocalTime time = LocalTime.now();
```

---

#### **LocalDateTime**

`LocalDateTime` es una clase compuesta que combina las clases anteriormente mencionadas `LocalDate` y `LocalTime`.

En el siguiente fragmento de código se puede apreciar como construir un `LocalDateTime` haciendo uso de todos los campos (año, mes, día, hora, minuto, segundo, nanosegundo):

---

```
1 LocalDateTime dateTime = LocalDateTime.of(1989, 11, 11, 5, 30, 45, 35);  
   //1989-11-11T05:30:45.000000035
```

---

También, se puede crear un objeto `LocalDateTime` basado en los tipos `LocalDate` y `LocalTime`, a continuación se muestra el uso del método `of(LocalDate date, LocalTime time)`:

---

```
1 LocalDate date = LocalDate.of(1989, 11, 11);  
2 LocalTime time = LocalTime.of(5, 30, 45, 35);  
3 LocalDateTime dateTime = LocalDateTime.of(date, time);
```

---

Finalmente, se puede capturar el `LocalDateTime` exacto de la ejecución usando el método `now()`, como se muestra a continuación.

---

```
1 LocalDateTime dateTime = LocalDateTime.now();
```

---

### Instant

`Instant` representa el número de segundos desde 1 de Enero de 1970. Es un modelo de fecha y tiempo fácil de interpretar para una máquina.

---

```
1 Instant instant = Instant.ofEpochSecond(120);  
2 System.out.println(instant); //1970-01-01T00:02:00Z
```

---

De igual manera que las clases ya mencionadas, `Instant` provee el método `now()`.

---

```
1 Instant instant = Instant.now();
```

---

### Duration

`Duration` hace referencia a la diferencia que existe entre dos objetos de tiempo.

En el siguiente ejemplo, la duración se calcula haciendo uso de dos objetos `LocalTime`:

---

```
1 LocalTime localTime1 = LocalTime.of(12, 25);  
2 LocalTime localTime2 = LocalTime.of(17, 35);  
3 Duration duration = Duration.between(localTime1, localTime2);
```

---

Otra opción de calcular la duración entre dos objetos es usando dos objetos `LocalDateTime`:

---

```
1 LocalDateTime localDateTime1 = LocalDateTime.of(2016, Month.JULY, 18, 14,  
   13);  
2 LocalDateTime localDateTime2 = LocalDateTime.of(2016, Month.JULY, 20, 12,  
   25);  
3 Duration duration = Duration.between(localDateTime1, localDateTime2);
```

---



También, se puede crear una duración basada en el método `of(long amount, TemporalUnit unit)`. En el siguiente ejemplo, se muestra como crear un `Duration` de un día.

---

```
1 Duration oneDayDuration = Duration.of(1, ChronoUnit.DAYS);
```

---

Se puede apreciar el uso del enum `ChronoUnit`, la cual es una implementación de `TemporalUnit` y nos brinda una serie de unidades de periodos de tiempo como `ERAS`, `MILLENNIA`, `CENTURIES`, `DECADES`, `YEARS`, `MONTHS`, `WEEKS`, etc.

También, se puede crear `Duration` basado en los métodos `ofDays(long days)`, `ofHours(long hours)`, `ofMillis(long millis)`, `ofMinutes(long minutes)`, `ofNanos(long nanos)`, `ofSeconds(long seconds)`. El ejemplo anterior puede ser reemplazado por la siguiente línea:

---

```
1 Duration oneDayDuration = Duration.ofDays(1);
```

---

### Period

`Period` hace referencia a la diferencia que existe entre dos fechas.

---

```
1 LocalDate localDate1 = LocalDate.of(2016, Month.JULY, 18);
2 LocalDate localDate2 = LocalDate.of(2016, Month.JULY, 20);
3 Period period = Period.between(localDate1, localDate2);
```

---

Se puede crear `Period` basado en el método `of(int years, int months, int days)`. En el siguiente ejemplo, se crea un período de 1 año 2 meses y 3 días:

---

```
1 Period period = Period.of(1, 2, 3);
```

---

Del mismo modo que `Duration`, se puede crear `Period` basado en los métodos `ofDays(int days)`, `ofMonths(int months)`, `ofWeeks(int weeks)`, `ofYears(int years)`.

---

```
1 Period period = Period.ofYears(1);
```

---

Una de las ventajas de la nueva API de fechas es que en todas las clases los métodos son los mismos.

Nombre	Tipo	Uso
of	static	Crear una instancia del objeto a partir de los datos de entrada
from	static	Convertir el parámetro de entrada en una instancia de la clase
parse	static	Procesar la cadena de entrada y construir una instancia
format	instancia	Procesar los datos para producir una cadena
get	instancia	Devolver una parte del objeto
is	instancia	Consultar el estado del objeto
with	instancia	Devolver una copia con uno o varios elementos modificados

Nombre	Tipo	Uso
plus	instancia	Devolver una copia del objeto con una cantidad de tiempo añadido
minus	instancia	Devolver una copia del objeto con una cantidad de tiempo sustraído
to	instancia	Convertir el objeto en otro tipo
at	instancia	Combinar el objeto con otro objeto

Operaciones varias realizadas de dos maneras según la versión de java.

```
1 package ejemplo10;
2
3 import java.text.DateFormat;
4 import java.text.ParseException;
5 import java.text.SimpleDateFormat;
6 import java.time.Duration;
7 import java.time.Instant;
8 import java.time.LocalDate;
9 import java.time.LocalDateTime;
10 import java.time.LocalTime;
11 import java.time.Period;
12 import java.time.format.DateTimeFormatter;
13 import java.util.Calendar;
14 import java.util.Date;
15 import java.util.Locale;
16 import javax.swing.JOptionPane;
17
18 public class Ejemplo10 {
19
20     public static void verificar(int version) {
21         if (version == 7) {
22             Calendar fecha1 = Calendar.getInstance();
23             Calendar fecha2 = Calendar.getInstance();
24             fecha1.set(1991, 0, 21);
25             System.out.println(fecha1.after(fecha2));
26         } else
27         if (version == 8) {
28             LocalDate fecha1 = LocalDate.of(1991, 01, 21);
29             LocalDate fecha2 = LocalDate.now();
30
31             System.out.println(fecha1.isAfter(fecha2));
32             System.out.println(fecha1.isBefore(fecha2));
33
34             LocalTime tiempo1 = LocalTime.of(22, 30, 50);
35             LocalTime tiempo2 = LocalTime.now();
36
37             System.out.println(tiempo1.isAfter(tiempo2));
38             System.out.println(tiempo1.isBefore(tiempo2));
39
40             LocalDateTime fechaTiempo1 = LocalDateTime.of(1991, 1, 21, 22,
41                 30, 50);
42             LocalDateTime fechaTiempo2 = LocalDateTime.now();
43
44             System.out.println(fechaTiempo1.isAfter(fechaTiempo2));
45             System.out.println(fechaTiempo1.isBefore(fechaTiempo2));
46         }
47     }
48
49     // currentTimeMillis
50     public static void medirTiempo(int version) throws InterruptedException
51     {
52         if (version == 7) {
53             long ini = System.currentTimeMillis();
54             Thread.sleep(1000);
55             long fin = System.currentTimeMillis();
56             System.out.println(fin - ini);
57         }
58     }
59 }
```

```
56     } else
57     {
58         if (version == 8) {
59             Instant ini = Instant.now();
60             Thread.sleep(1000);
61             Instant fin = Instant.now();
62             System.out.println(Duration.between(ini, fin));
63         }
64     }
65     // Calendar
66     public static void periodoEntreFechas(int version) {
67         if (version == 7) {
68             Calendar nacimiento = Calendar.getInstance();
69             Calendar actual = Calendar.getInstance();
70
71             nacimiento.set(1991, 0, 21);
72             actual.set(2017, 2, 04);
73
74             int anios = 0;
75
76             while (nacimiento.before(actual)) {
77                 nacimiento.add(Calendar.YEAR, 1);
78                 if (nacimiento.before(actual)) {
79                     anios++;
80                 }
81             }
82             System.out.println(anios);
83         } else
84         {
85             if (version == 8) {
86                 LocalDate nacimiento = LocalDate.of(1991, 1, 21);
87                 LocalDate actual = LocalDate.now();
88
89                 Period periodo = Period.between(nacimiento, actual);
90                 System.out.println("Han transcurrido " + periodo.getYears() + "
91                     años y " + periodo.getMonths() + " meses y "
92                     + periodo.getDays() + " dias, desde " + nacimiento + " hasta "
93                     + actual);
94             }
95         }
96     }
97     // Conversion
98     public static void convertir(int version) throws ParseException {
99         if (version == 7) {
100             String fecha = "21/01/1991";
101             DateFormat formateador = new SimpleDateFormat("dd/MM/yyyy");
102             Date fechaConvertida = formateador.parse(fecha);
103             System.out.println(fechaConvertida);
104
105             Date fechaActual = Calendar.getInstance().getTime();
106             formateador = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss a");
107             String fechaCadena = formateador.format(fechaActual);
108             System.out.println(fechaCadena);
109         } else
110         {
111             if (version == 8) {
112                 String fecha = "21/01/1991";
113                 DateTimeFormatter formateador =
114                     DateTimeFormatter.ofPattern("dd/MM/yyyy");
115                 LocalDate fechaLocal = LocalDate.parse(fecha, formateador);
116                 System.out.println(fechaLocal);
117             }
118         }
119     }
120 }
```

```
112
113         formateador = DateTimeFormatter.ofPattern("ddMMyyyy");
114         System.out.println(formateador.format(fechaLocal));
115     }
116 }
117
118 public static void main(String[] args) {
119     try
120     {
121         verificar(8);
122         medirTiempo(8);
123         periodoEntreFechas(8);
124         convertir(8);
125     }
126     catch (Exception e)
127     {
128         e.printStackTrace();
129     }
130 }
131 }
```

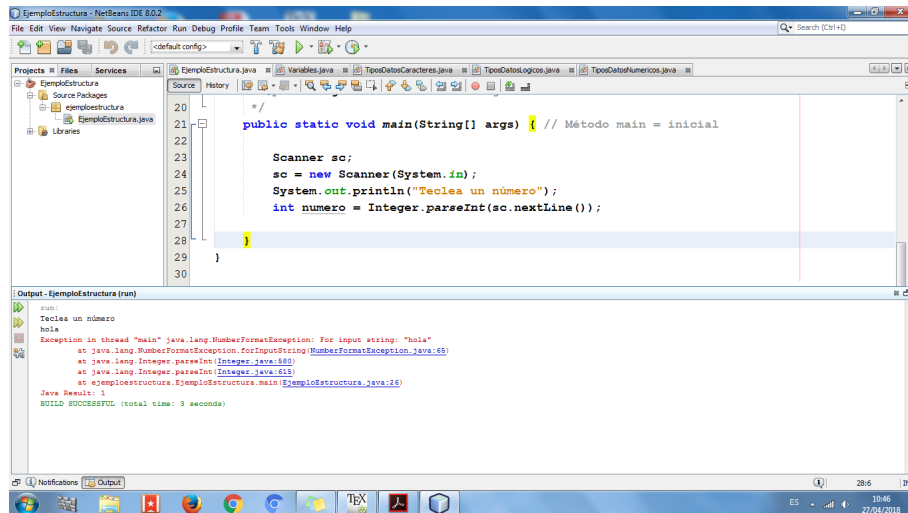
---

## 3.15. Excepciones

Una excepción o exception es un error o una condición anormal que se ha producido durante la ejecución de un programa .

Ejemplos:

- El usuario escribe una palabra cuando esperabamos un número.



```

20
21
22
23
24 Scanner sc;
25 sc = new Scanner(System.in);
26 System.out.println("Teclea un número");
27 int numero = Integer.parseInt(sc.nextLine());
28
29
30

```

Output: EjemploEstructura (run)

```

com
Teclea un número
hola
Exception in thread "main" java.lang.NumberFormatException: For input string: "hola"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:850)
    at java.lang.Integer.parseInt(Integer.java:615)
    at ejemploEstructura.EjemploEstructura.main(EjemploEstructura.java:26)
Java Result: 1
BUILD SUCCESSFUL (total time: 3 seconds)

```

- El programa intenta leer un fichero que no existe.
- El programa intenta realizar una división por cero.
- Se intenta calcular la raíz cuadrada de un número negativo.
- El programa sale de los límites de un array.
- El programa accede a los miembros de un objeto inexistente.

La clase **Exception** es la clase que Java proporciona para la gestión de errores.

Con la clase Exception podremos controlar los errores que se produzcan y decidir como tratarlos.

Si no tratamos las excepciones se encarga de ellas la máquina virtual java.

### 3.15.1. Bloque try..catch: Captura de excepciones.

Con este bloque podremos programar las acciones que queremos llevar a cabo cuando se produzca un error en el código situado dentro del bloque try.

---

```
1
2 try
3 {
4     //Código que nos puede generar algún error en la ejecución.
5
6 }
7 catch (Tipol de error)
8 {
9
10    //Sentencias que se realizarán si se produce un error de Tipol
11
12 }
```

---

Se capturan las excepciones (errores) que se produzcan en el código situado entre el try y el catch.

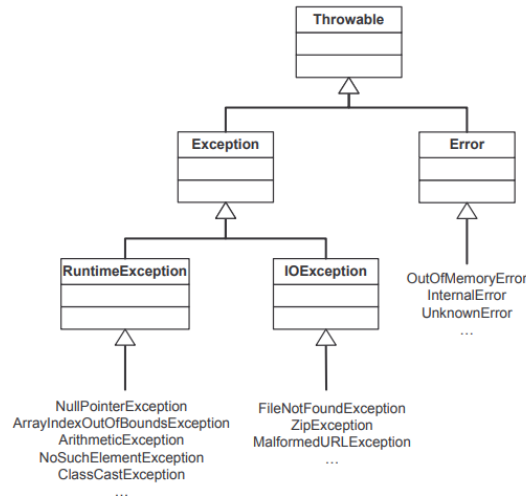
En cuanto se produce la excepción, la ejecución del bloque try termina. Se salta a un bloque catch.

Puede haber más de un catch. En este caso los tipos de excepción deben ir de más concretos a más genéricos.

El operador or nos permite tratar más de un tipo de excepción en un bloque catch.

### 3.15.2. Tipos de excepciones predefinidas más frecuentes

*Jerarquía de clases para el manejo de excepciones en Java*



**Throwable** es la clase base que representa todo lo que se puede lanzar en java.

**Error** es una subclase de throwable que indica problemas graves que un aplicación no debería intentar solucionar, por ejemplo, memoria agotada o error de la JVM.

**Exception** y sus subclases indican situaciones que una aplicación debería tratar de forma razonable. Los dos tipos principales de excepciones son: **RuntimeException** (errores del programador, como una división por cero) y **IOException** (errores que no puede evitar el programador, generalmente relacionados con el entrada/salida del programa).

<https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html>

Los nombres de las excepciones indican la condición de error que representan. Las siguientes son las excepciones predefinidas más frecuentes que se pueden encontrar:

**ArithmeticException** Las excepciones aritméticas son típicamente el resultado de división por 0. `int i = 12 / 0;`

**NullPointerException** Se produce cuando se intenta acceder a una variable o método antes de ser inicializado.



**IncompatibleClassChangeException** El intento de cambiar una clase afectada por referencias en otros objetos, específicamente cuando esos objetos todavía no han sido recompilados.

**ClassCastException** El intento de convertir un objeto a otra clase que no es válida.

`y = (Prueba)x; // donde x no es de tipo Prueba`

**NegativeArraySizeException** Puede ocurrir si hay un error aritmético al cambiar el tamaño de un array.

**OutOfMemoryException** Esta excepción no debería producirse nunca. El intento de crear un objeto con el operador `new` ha fallado por falta de memoria. Y siempre tendría que haber memoria suficiente porque el **garbage collector** se encarga de proporcionarla al ir liberando objetos que no se usan y devolviendo memoria al sistema.

**NoClassDefFoundException** Se hace referencia a una clase que el sistema es incapaz de encontrar.

**ArrayIndexOutOfBoundsException** Se genera al intentar acceder a un elemento de un array más allá de los límites definidos inicialmente para ese array.

**UnsatisfiedLinkException** Se hizo el intento de acceder a un método que no existe.

**NumberFormatException**

Se intenta convertir un `String` en un número y es imposible.

### 3.15.3. Secuencia de ejecución de los bloques

---

```
1 //Bloque uno
2 try
3 {
4     //Bloque dos
5 }
6 }
7 catch (Tipo1 de error)
8 {
9 }
10 //Bloque tres
11
12 }
13 //Bloque cuatro
```

---

Si no se producen excepciones: Bloque uno, bloque dos, bloque cuatro.

Con una excepción en el bloque uno: Bloque uno.

Con una excepción en el bloque dos: Bloque uno, bloque dos, bloque tres y bloque cuatro

---

```
1 //Bloque uno
2 try
3 {
4     //Bloque dos
5 }
6 catch (ArithmeticException ae)
7 {
8     //Bloque tres
9 }
10 catch (NullPointerException ne)
11 {
12     //Bloque cuatro
13 }
14 //Bloque cinco
```

---

Si no se producen excepciones: Bloque uno, bloque dos, bloque cinco

Con una excepción en el bloque uno: Bloque uno.

Con una excepción en el bloque dos de tipo aritmético : Bloque uno, bloque dos, bloque tres y bloque cinco

Con una excepción en el bloque dos de tipo referencia nula: Bloque uno, bloque dos, bloque cuatro y bloque cinco.

Con una excepción en el bloque dos de otro tipo: Bloque uno y bloque dos.

---

```
1 //Bloque uno
2 try
3 {
4     //Bloque dos
5 }
6 catch (ArithmeticException ae)
7 {
8     //Bloque tres
9 }
10 catch (Exception e)
11 {
12     //Bloque cuatro
13 }
14 //Bloque cinco
```

---

Si no se producen excepciones: Bloque uno, bloque dos, bloque cinco

Con una excepción en el bloque uno: Bloque uno.

Con una excepción en el bloque dos de tipo aritmético : Bloque uno, bloque dos, bloque tres y bloque cinco

Con una excepción en el bloque dos de otro tipo: Bloque uno, bloque dos, bloque cuatro y bloque cinco.

---

```
1 //Bloque uno
2 try
3 {
4     //Bloque dos
5 }
6 catch(Exception e)
7 {
8     //Bloque tres
9 }
10 }
11 catch (ArithmeticException ae)
12 {
13
14     //Bloque cuatro
15 }
16 //Bloque cinco
```

---

Si no se producen excepciones: Bloque uno, bloque dos, bloque cinco

Con una excepción en el bloque uno: Bloque uno.

Con una excepción en el bloque dos de tipo aritmético : Bloque uno, bloque dos, bloque tres y bloque cinco

Con una excepción en el bloque dos de otro tipo: Bloque uno, bloque dos, bloque tres y bloque cinco.

EL BLOQUE CUATRO NO SE EJECUTA NUNCA.

#### 3.15.4. La clausual finally

En ocasiones nos interesa ejecutar un fragmento de código independiente-mente si se produce o no una excepción.

Ejemplo: se produzca un error o no durante la manipulación de un fichero nos interesa cerrarlo.

---

```
1 try
2 {
3     //Código que nos puede generar algún error en la ejecución.
4 }
5 }
6 catch (Tipol de error)
7 {
8
```

---

```
9 //Sentencias que se realizarán si se produce un error de Tipo1
10
11 }
12 finally
13 {
14 //Sentencias que siempre se ejecutaran ocurra o no una //excepción
15 }
```

---

Si el cuerpo del bloque try llega a comenzar su ejecución, el bloque finally siempre se ejecutará. Detrás del bloque try si no se produce excepciones. o después de un bloque catch si éste captura la excepción. Justo después de que se produzca la excepción si ninguna cláusula catch captura la excepción.

### 3.15.5. Lanzamiento de excepciones.

Java contiene excepciones diseñadas e implementadas que podemos utilizar y que se lanzan cuando se produce algún error.

*¿ Por qué no puede LANZARSE una excepción que me indique en mi aplicación que no puedo introducir un nuevo producto en el almacén porque está llenó? EN EL ÁMBITO DE MI APLICACIÓN ESO ES UN ERROR.*

Se puede provocar el lanzamiento de excepciones a través de la sentencia **throw**.

### 3.15.6. Sentencia throw.

Se utiliza para lanzar objetos de tipo Throwable(Exception).

Sintaxis: `throw new Exception(" Mensaje de error.");`

---

```
1 public class Main {
2
3     public static void main(String[] args) {
4         //supongamos que el numerador nos lo introduce el usuario por
           teclado
5         int numerador=12;
6         //supongamos que el denominador nos lo introduce el usuario por
           teclado
7         int denominador=0;
8         try
9         {
10             if (denominador==0)
11                 throw new ArithmeticException("Error en la operación");
12             double c=numerador/denominador;
13             System.out.println(c+4);
14         }
15         catch(ArithmeticException ex)
16         {
17             System.out.println(ex.getMessage());
18         }
19     }
20 }
```

```
18         }
19     }
20 }
```

---

### 3.15.7. Relanzar una excepción, propagación.

Existen algunos casos en los cuales el código de un método puede generar una *Exception* y no se desea incluir en dicho método la gestión del error. Java permite que este método pase o relance(*throws*) la *Exception* al método desde el que ha sido llamado, sin incluir en el método los bucles *try/catch* correspondientes.

Esto se consigue mediante la adición de *throws* más el nombre de la *Exception* concreta después de la lista de argumentos del método. A su vez el método superior deberá incluir los bloques *try/catch* o volver a pasar la *Exception*. De esta forma se puede ir pasando la *Exception* de un método a otro hasta llegar al último método del programa, el método *main()*.

---

```
1
2 public void static main(String args[]){
3     ..
4     metodo1()
5     ..
6 }
7
8 void metodo1(){
9     ...
10    try {
11        // Código que puede lanzar las excepciones
12        // IOException
13    } catch (IOException e1) {
14        // Se ocupa de IOException
15    }...
16 } // Fin del metodo1
17
18
19 //No me interesa tratar la excepción en el método, y la propago
20
21 void metodo1() throws IOException{
22     .
23     // Código que puede lanzar las excepciones
24
25 }
```

---

### 3.15.8. Creación de excepciones.

Un programador puede crear sus propias excepciones sólo con heredar de la clase *Exception* o de una de sus clases derivadas. Lo lógico es heredar de

la clase de la jerarquía de Java que mejor se adapte al tipo de excepción. Las clases Exception suelen tener dos constructores:

- Un constructor sin argumentos.
- Un constructor que recibe un String como argumento. En este String se suele definir un mensaje que explica el tipo de excepción generada. Conviene que este constructor llame al constructor de la clase de la que deriva `super(String)`.

Al ser clases como cualquier otra se podrían incluir variables y métodos nuevos. Por ejemplo:

---

```
1 class AlmacenLlenoException extends Exception
2 {
3     public AlmacenLlenoException()
4     { // Constructor por defecto
5         super();
6     }
7     public AlmacenLlenoException(String s)
8     { // Constructor con mensaje
9         super(s);
10    }
11 }
12
13 public class Almacen{
14     ..
15     public void añadirProducto(Producto p) throws AlmacenLlenoException{
16         if (!(listaProductos.size()<20))
17             throw new AlmacenLlenoException(Almacén lleno. );
18         listaProductos.addElement(p);
19     }
20     .
21 }
22
23 Versión 1: PROPAGANDO EXCEPCIONES
24
25 public class Main{
26     ..
27     try{
28         miAlmacen.añadirProducto(p1);
29     }catch(AlmacenLlenoException ex)
30     { //Trato la excepción
31         System.out.println(ex.getMessage());
32     }
33     .
34 }
35
36 Versión dos: TRATANDO EXCEPCIONES
37
38 public class Almacen{
39     ..
40     public void añadirProducto(Producto p) {
41         try{
42             if (!(listaProductos.size()<20))
```

```
43         throw new AlmacenLlenoException(Almacén lleno. );
44     listaProductos.addElement(p);
45     }
46     catch(AlmacenLlenoException ex)
47     {
48         System.out.println(ex.getMessage());
49     }
50 }
51 .
52 }
53
54 public class Main{
55     ..
56     miAlmacen.añadirProducto(p1);
57     .
58 }
```

---