Java

Frogames

Fundamentos del lenguaje

Conceptos básicos

PROGRAMA
Conjunto de instrucciones.

PROGRAMACIÓN
Proceso de realización de un programa.

OBJETIVOS DE LA PROGRAMACIÓN
FIABILIDAD
LEGIBILIDAD
EFICIENCIA

Conceptos básicos

LENGUAJES DE PROGRAMACIÓN
Lenguajes utilizados para la realización de un programa.

LENGUAJE MÁQUINA
Lenguaje que codifica un programa al más bajo nivel del ordenador. Cada procesador presenta su propio lenguaje máquina.

LENGUAJES DE ALTO NIVEL
Utilizados para representar programas de manera simbólica muy cercana al lenguaje natural. Son independientes de las máquinas en las que se vayan a utilizar.

Conceptos básicos

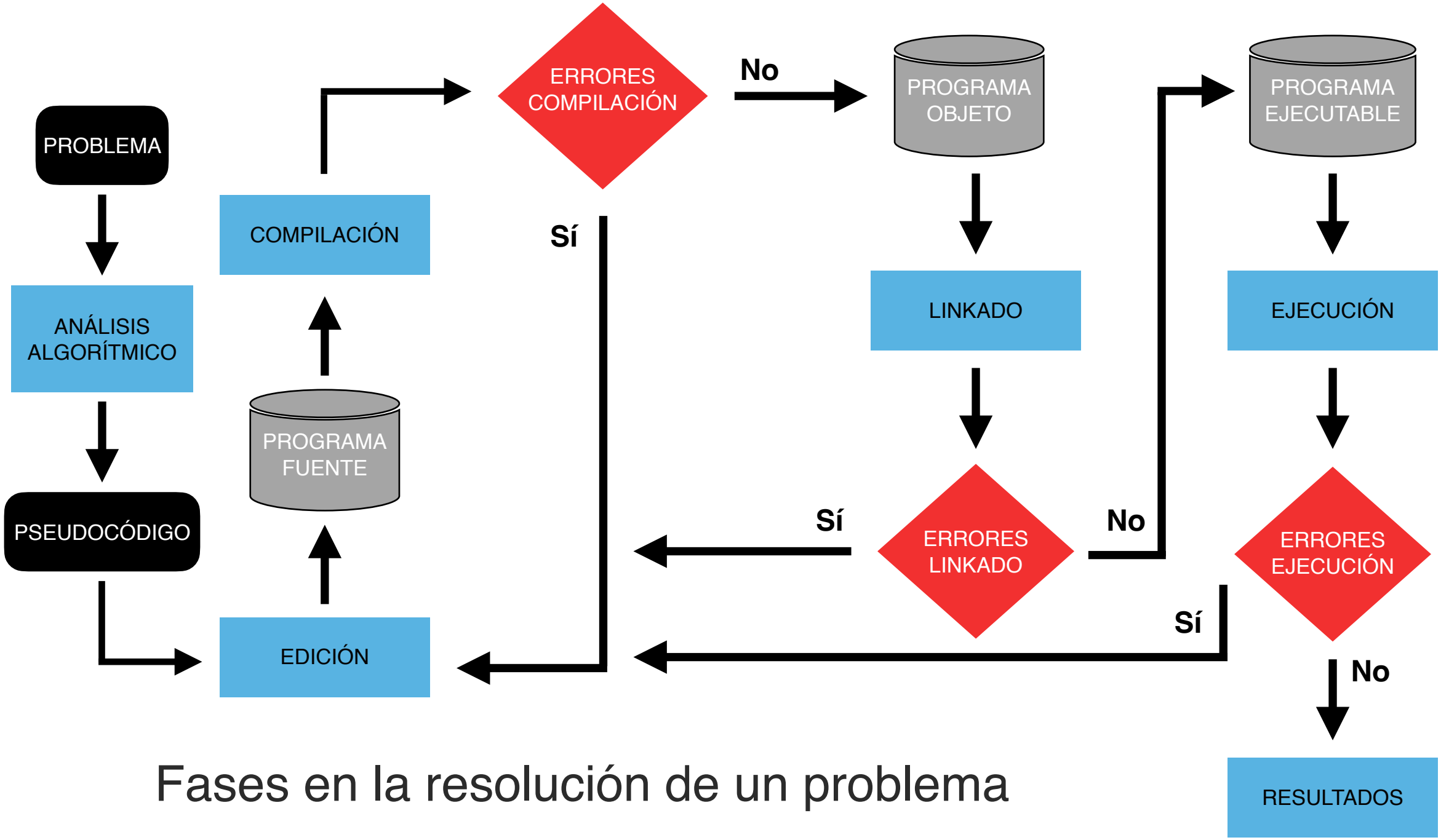
COMPILADORES

- Traduce programas de un lenguaje de programación simbólico a código máquina.
- La representación del programa en lenguaje simbólico se le llama programa fuente, y su representación en código máquina se le llama programa objeto.
- El lenguaje simbólico también se le llama lenguaje fuente y al lenguaje máquina lenguaje objeto.

Conceptos básicos

ACCIÓN PRIMITIVA
Acción directamente realizable por el procesador.

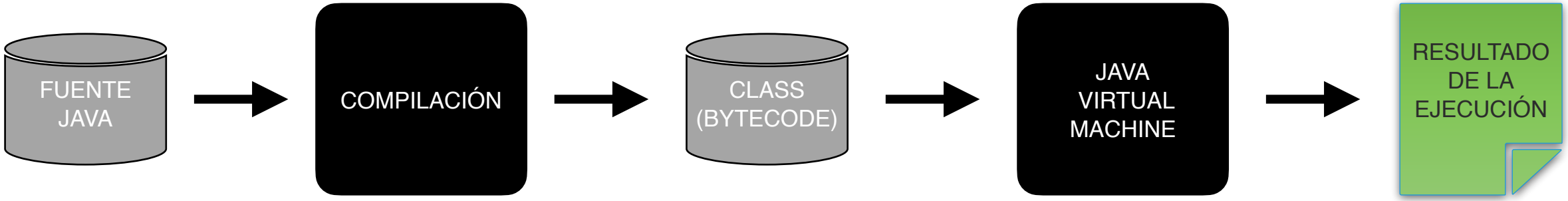
ACCIÓN COMPUESTA
Conjunto de acciones simples que realizan una tarea más o menos larga. La descripción de un programa en acciones compuestas puede facilitar su comprensión.



Fases en la resolución de un problema

Introducción a Java

- Java es un lenguaje multiplataforma.
- ***Java Virtual Machine*** (JVM) constituye una plataforma de ejecución virtual.
- No existe proceso de linkado, los enlaces se resuelven en ejecución.



Fundamentos del lenguaje

Comentarios

COMENTARIOS DE UNA SOLA LÍNEA

```
// Esto es un comentario de una sola línea
```

COMENTARIOS DE MÚLTIPLES LÍNEAS

```
/*
Esto es un comentario
de más de una
Línea
*/
```

Comentarios

Javadoc es el estándar de la industria para documentar clases en Java.

COMENTARIOS DE DOCUMENTACIÓN (JAVADOC)

```
/**
* Esto es un
* comentario para
* javadoc
*/
```

Etiquetas

Etiquetas principales que se usan en javadoc:

ETIQUETA	DESCRIPCIÓN
@author	Autor del elemento a documentar.
@version	Versión actual del componente.
@param	Código para documentar cada uno de los parámetros.
@return	Indica los parámetros de salida.
@exception	Indica la excepción que puede generar.
@deprecated	El método, clase u otro componente está obsoleto.
@see	Referencia a otra clase o utilidad.

Ejemplos de comentarios

EJEMPLO DE COMENTARIOS DE CLASES

```
/**
* Descripción de la clase
* @author Nombre / Empresa
*  @version 0.1
*/
```

EJEMPLO DE COMENTARIOS DE MÉTODOS

```
/**
* Breve descripción
* del método
*@param descripción del parámetro param1
*@return qué devuelve el método
*@exception tipo de excepción que lanza un método y en qué caso
*@see
*/
```

Ejemplos de comentarios

EJEMPLO DE COMENTARIOS DE VARIABLES

```
/**
* Descripción de la variable
* Valores válidos (si aplica)
*  Comportamiento en caso de que sea null (si aplica)
*/
```

Sentencias

Una sentencia es una orden que se le da al programa para realizar una tarea específica.

El carácter ‘;’ se utiliza como separador de sentencias.

SENTENCIAS

```
Sentencia A;
Sentencia B;
...
```

Bloques de código

Un bloque de código es un grupo de sentencias que se comportan como una unidad.

Las llaves de apertura ‘{’ y cierre ‘}’ delimitan el bloque de código.

BLOQUE DE CÓDIGO

```
{
    Sentencia A;
    Sentencia B;
    ...
    Sentencia N;
}
```

Método main

El método main es el punto de entrada de la aplicación de java.

MÉTODO MAIN

```
public class <identificador> {
   public static void main(String [] args){
      // Bloque de acciones
   }
}
```

Fundamentos del lenguaje

Variables

Una variable representa un valor almacenado en memoria principal y viene definida por un tipo, un identificador y un valor.

DECLARACIÓN DE UNA VARIABLE

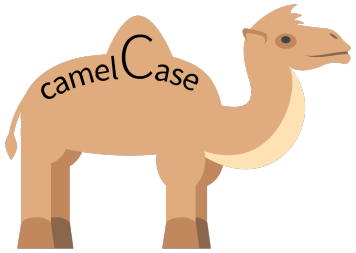
```
<tipo> <identificador> = <valor_inicial>;
```

- Un identificador debe comenzar por un carácter que no sea dígito ni carácter especial. No se pueden utilizar las palabras reservadas del lenguaje.
- Java es un lenguaje *case sensitive.*
- Los identificadores deben ser mnemotécnicos.

Convenciones de nombres (camelCase)

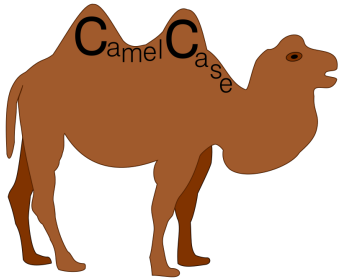
- camelCase:

  Consiste en unir dos o más palabras sin espacios entre ellas, separando las palabras con una letra mayúscula inicial a partir de la segunda palabra.



- PascalCase:

  Idéntico al caso anterior, con la excepción de que la primera letra es mayúscula.



Convenciones de nombres (snake_case)

- snake_case:

  Une las palabras por una barra baja (“_”) y con todas las letras en minúscula

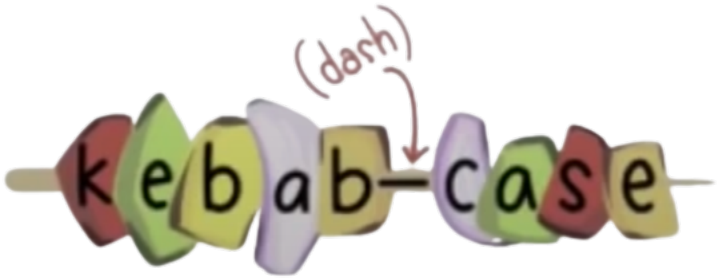
- SCREAMING_SNAKE_CASE:

  Igual que el anterior, excepto que todas las letras son en mayúscula. Se suele usar para definir constantes.

Convenciones de nombres (kebab-case)

- kebab-case:

  Igual a snake case pero con un guión (“-”) como separador.



- Train-Case:

  Variedad del kebab case con la primera letra de cada palabra en mayúscula.

Otras convenciones de nombres

- Notación húngara:

  Consiste en añadir prefijos en minúscula al nombre de la variable para indicar el tipo. Por ejemplo: strNombre, intContador, ...

Ejemplos de variables

EJEMPLO

```
int contador_eventos = 0;

int contadorEventos = 0;
```

También podemos declarar una variable sin dar un valor inicial:

EJEMPLO

```
int contadorEventos;
```

Múltiples variables

Para declarar más de una variable del mismo tipo debemos indicar el tipo, seguido de los identificadores y valores iniciales separados por comas.

DECLARACIÓN DE MÚLTIPLES VARIABLES

```
<tipo> <identificador1> = <valor1> , ..., <identificadorN> = <valorN>;
```

EJEMPLO

```
int dia = 23, mes = 5, año = 1995;
int dia, mes, año;
```

Tipos de datos primitivos

- Enteros:

TIPO	DESCRIPCIÓN
byte	Entero de 8 bits. Puede almacenar los valores numéricos [—128,127].
short	Entero de 16 bits. Puede almacenar los valores numéricos [—32768, 32767].
int	Entero de 32 bits. Puede almacenar los valores numéricos $[-2^{31}, 2^{31}-1]$.
long	Entero de 64 bits. Puede almacenar los valores numéricos $[-2^{63}, 2^{63}-1]$.

- Reales:

TIPO	DESCRIPCIÓN
float	Puede almacenar números en coma flotante con precisión simple de 32 bits.
double	Puede almacenar números en coma flotante con doble precisión de 64 bits.

Tipos de datos primitivos

- Caracteres:

TIPO	DESCRIPCIÓN
char	Tipo de dato que representa a un solo carácter Unicode de 16 bits

- Booleanos:

TIPO	DESCRIPCIÓN
boolean	Tipo de dato que solo puede tomar dos valores: true o false. Ocupa 1 bit de información.

Constantes

Una constante representa un valor en memoria principal que no puede ser modificado.

DECLARACIÓN DE UNA CONSTANTE

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

Fundamentos del lenguaje

Strings

La clase string nos permite trabajar con cadenas de texto.

DECLARACIÓN DE UN STRING

```
String <identificador> = new String(<cadena>);
```

Java también acepta la siguiente forma para declarar un string:

DECLARACIÓN DE UN STRING

```
String <identificador> = <cadena>;
```

Métodos más utilizados

MÉTODO	DESCRIPCIÓN
length()	Longitud de la cadena.
indexOf(<caracter>)	Índice de la primera aparición del carácter buscado.
lastIndexOf(<caracter>)	Índice de la última aparición del carácter buscado.
charAt(<índice>)	Carácter ubicado en la posición indicada.
toCharArray()	Convierte la cadena en un array de caracteres.
toUpperCase()	Convierte toda la cadena a mayúsculas.
toLowerCase()	Convierte toda la cadena a minúsculas.
substring(<índice1>, <índice2>)	Subcadena que se encuentra entre las posiciones índice1 y índice2-1.

Métodos más utilizados

MÉTODO	DESCRIPCIÓN
startsWith(<expresión>)	Comprueba si la cadena empieza con la expresión proporcionada.
endsWith(<expresión>)	Comprueba si la cadena termina con la expresión proporcionada.
replace(<caracterViejo>, <carácterNuevo>)	Reemplaza todas las ocurrencia del carácter viejo por el carácter nuevo.
replaceAll(<cadenaVieja>, <cadenaNueva>)	Reemplaza todas las ocurrencia de la cadena vieja por la cadena nueva.
split(<expresión>)	Divide la cadena entorno a las coincidencias de la expresión proporcionada.
concat(<cadena>)	Concatena la cadena especificada al final de la cadena inicial.
copyValueOf(<arrayCaracteres>)	Copia la secuencia de caracteres proporcionada en una cadena.

Métodos más utilizados

MÉTODO	DESCRIPCIÓN
equals(<cadena>)	Compara dos cadenas. Devuelve true si son iguales y false en caso contrario.
equalsIgnoreCase(<cadena>)	Compara dos cadenas. Devuelve true si son iguales y false en caso contrario. No considera mayúsculas y minúsculas.
trim()	Elimina los espacios en blanco de los extremos, pero no modifica los intermedios.
valueOf(<argumento>)	Proporciona la representación a string del argumento proporcionado.

NOTA: Para consultar más información la clase String, podemos consultar la documentación oficial de java (documentación).

Fundamentos del lenguaje

Operadores aritméticos

OPERADOR	DESCRIPCIÓN	USO
+	Operador unario suma. Indica un número positivo.	+ exp
+	Suma.	exp + exp
—	Operador unario de cambio de signo.	— exp
—	Resta.	exp — exp
*	Producto.	exp * exp
/	División (tanto entera como real).	exp / exp
%	Resto de la división entera.	exp % exp

Operadores aritméticos incrementales

OPERADOR	DESCRIPCIÓN	USO
++	Operador postincremento. Primero se utiliza la variable y luego se incrementa su valor en una unidad.	exp ++
++	Operador preincremento. Primero se incrementa el valor de la variable en una unidad y luego se utiliza.	++ exp
— —	Operador postdecremento. Primero se utiliza la variable y luego se decrementa su valor en una unidad.	exp — —
— —	Operador predecremento. Primero se decrementa el valor de la variable en una unidad y luego se utiliza.	— — exp

Operadores relacionales

OPERADOR	DESCRIPCIÓN	USO
==	Igual que	exp == exp
!=	Distinto que	exp != exp
>	Mayor que	exp > exp
<	Menor que	exp < exp
>=	Mayor o igual que	exp >= exp
<=	Menor o igual que	exp <= exp

Operadores de asignación

OPERADOR	DESCRIPCIÓN	USO
=	Asignación de un valor a una variable.	var = exp
+=	Suma y asignación.	var += exp (var <— var + exp)
—=	Resta y asignación.	var —= exp (var <— var — exp)
*=	Multiplicación y asignación.	var *= exp (var <— var * exp)
/=	División y asignación.	var /= exp (var <— var / exp)
%=	Módulo y asignación.	var %= exp (var <— var % exp)

Operadores lógicos

- &&

exp	exp	exp && exp
true	true	true
true	false	false
false	true	false
false	false	false

- ||

exp	exp	exp || exp
true	true	true
true	false	true
false	true	true
false	false	false

- !

exp	!exp
true	false
false	true

Operador condicional ternario

OPERADOR	DESCRIPCIÓN	USO
?	Evalua una expresión ‘a’ y toma el valor ‘b’ si la expresión es verdadera, en caso contrario toma el valor ‘c’.	a ? b: c

Fundamentos del lenguaje

Entrada y salida de datos

- Para poder acceder a los dispositivos de entrada y salida en java es necesario invocar el sistema ***System***.

- Para poder realizar operaciones de entrada y salida es necesario importar ***java.io***

IMPORTAR EL PAQUETE java.io

```
import java.io.*;
```

Entrada de datos

- Lectura de un carácter:

LECTURA DE UN CARÁCTER DEL TECLADO

```
char <identificador> = (char)System.in.read();
```

- Lectura de un String:

LECTURA DE UNA LÍNEA DE CARACTERES DEL TECLADO

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String <identificador> = in.readLine();
```

Entrada de datos

- Lectura de un entero:

LECTURA DE UN NÚMERO ENTERO DEL TECLADO

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String <identificadorLinea> = in.readLine();
int <identificadorNumero> = Integer.parseInt(<identificadorLinea>);
```

- Lectura de un real:

LECTURA DE UN NÚMERO REAL DEL TECLADO

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String <identificadorLinea> = in.readLine();
float <identificadorNumero> = Float.parseFloat(<identificadorLinea>);
```

Lectura de datos con Scanner

Scanner es una clase que facilita el proceso de lectura de datos.

CLASE SCANNER

```
import java.util.Scanner;
```

Métodos más utilizados

MÉTODO	DESCRIPCIÓN
next()	Lee un String hasta encontrar un delimitador, generalmente un espacio.
nextLine()	Lee un String hasta encontrar un salto de línea.
nextByte()	Lee valores byte.
nextShort()	Lee valores short.
nextInt()	Lee valores int.
nextLong()	Lee valores long.
nextFloat()	Lee valores float.
nextDouble()	Lee valores double.
nextBoolean()	Lee valores booleanos.

Ejemplos de uso

- Lectura de un entero:

EJEMPLO DE LECTURA DE UN NÚMERO ENTERO DEL TECLADO

```
Scanner <identificadorScanner> = new Scanner(System.in);
int <identificadorInt> = <identificadorScanner>.nextInt();
```

- Lectura de un String:

EJEMPLO DE LECTURA DE UN STRING ENTERO DEL TECLADO

```
Scanner <identificadorScanner> = new Scanner(System.in);
String <identificadorString> = <identificadorScanner>.nextLine();
```

Ejemplos de uso

- Lectura de un carácter:

EJEMPLO DE LECTURA DE UN CARÁCTER ENTERO DEL TECLADO

```
Scanner <identificadorScanner> = new Scanner(System.in);
char <identificadorChar> = <identificadorScanner>.next().charAt(0);
```

Salida de datos

- Imprimir sin salto de línea:

VISUALIZACIÓN POR MONITOR

```
System.out.print(<exp>);
```

- Imprimir con un salto de línea:

VISUALIZACIÓN POR MONITOR

```
System.out.println(<exp>);
```

Fundamentos del lenguaje

Condicionales

Los condicionales permiten ejecutar una de entre varias acciones en función del valor de una expresión lógica o relacional.

- If:

IF

```
if(<condición>){
   // Bloque de acciones
}
```

NOTA: Las llaves no son necesarias si solamente hay una sentencia en el if.

Condicionales

- If-else:

IF ELSE

```
if(<condición>){
   // Bloque de acciones
}else{
   // Bloque de acciones
}
```

Condicionales

- If-else if:

ELSE IF

```
if(<condición>){
   // Bloque de acciones
}else if(<condición>){
   // Bloque de acciones
}else{
   // Bloque de acciones
}
```

Condicionales

- switch:

SWITCH

```
switch(<expresión>){
   case valor1: // Bloque de acciones
                break;
   case valor2: // Bloque de acciones
                break;
   ...
   case valorN: // Bloque de acciones
                break;
   default: // Bloque de acciones
}
```

Características del switch

- Cada sentencia **case** se corresponde con un único valor de la expresión.
- Los valores no comprendidos en ninguna sentencia case se pueden gestionar en default, que es opcional.
- En ausencia de **break**, cuando se ejecuta una sentencia **case** se ejecutan también todas las **case** que van a continuación, hasta que se llega a un **break** o se termina el **switch**.

Condicionales

- switch (arrow case):

  Eliminan la necesidad de usar la sentencia **break** para evitar la caída.

SWITCH

```
switch(<expresión>){
   case valor1-> {// Bloque de acciones}
   case valor2-> {// Bloque de acciones}
   ...
   case valorN-> {// Bloque de acciones}
   default-> {// Bloque de acciones}
}
```

NOTA: Las llaves no son necesarias si solamente hay una sentencia.

Condicionales

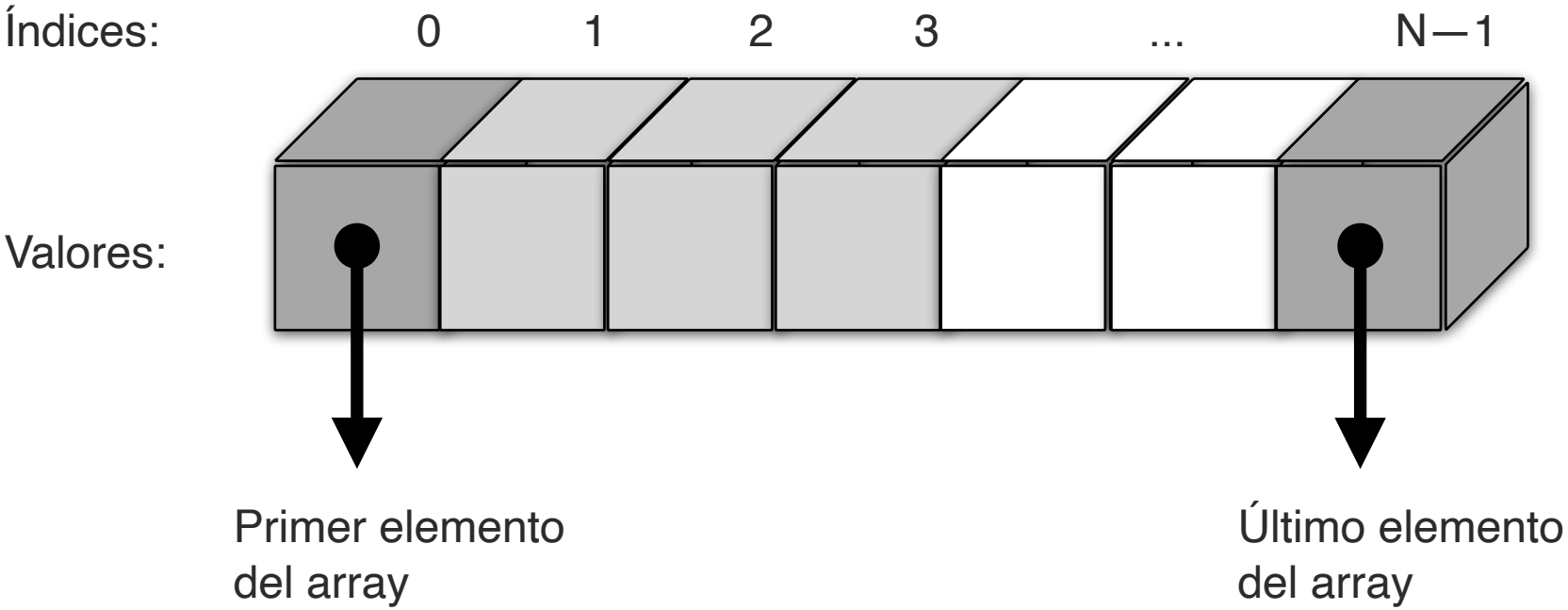
En ocasiones nos puede interesar ejecutar el mismo código en más de un caso:

SWITCH

```
case valor1, valor2, ..., valorN -> {// Bloque de acciones}
```

Fundamentos del lenguaje

¿Qué es un array?



Declarar un array

DECLARACIÓN DE UN ARRAY

```
<tipo> <identificador> [] = new <tipo> [<cantidad_datos>];
```

DECLARACIÓN DE UN ARRAY

```
<tipo> <identificador> [];
<identificador> = new <tipo> [<cantidad_datos>];
```

Inicialización de arrays

INICIALIZAR UN ARRAY

```
<tipo> <identificador> [] = {<elemento1>, ... , <elementoN>};
```

Operaciones con arrays

ACCEDER A UN ELEMENTO DEL ARRAY

```
<identificador> [<índice>];
```

ASIGNAR UN VALOR EN UNA POSICIÓN DEL ARRAY

```
<identificador> [<índice>] = <valor>;
```

Atributo .length

- El atributo **.length** devuelve el número de elementos que tiene el array.

OBTENER LA LONGITUD DE UN ARRAY

```
<identificador>.length;
```

Fundamentos del lenguaje

¿Qué es un bucle?

- Los bucles sirven para realizar un proceso de manera repetida.
- El código de un bucle se incluye entre las llaves { } (son opcionales si el bucle tiene una sola línea de código).
- El código del bucle se ejecutará mientras se cumpla una determinada condición.
- ¡Hay que prestar atención a los bucles infinitos!

Bucle for

- El bucle **for** ejecuta un bloque de acciones un número determinado de veces.

FOR

```
for(<inicialización>; <condición_parada>; <actualización>){
   // Bloque de acciones en cada iteración
}
```

NOTA: La <inicialización> y <actualización> pueden tener varias expresiones separadas por comas.

Bucle for-each

- El bucle **for each** se usa cuando no se necesita un índice de control y cada elemento de la colección debe procesarse.

FOR EACH

```
for(<tipo> <identificador_temporal> : <identificador_colección>){
   // Bloque de acciones en cada iteración
}
```

Bucle while

- El bucle **while** ejecuta un bloque de acciones repetidamente mientras se cumpla la condición.

WHILE

```
while(<condición>){
   // Bloque de acciones mientras se cumpla <condición>
}
```

Bucle do while

- El bucle **do while** ejecuta un bloque de acciones repetidamente mientras se cumpla la condición.

DO WHILE

```
do{
   // Bloque de acciones mientras se cumpla <condición>
}while(<condición>);
```

NOTA: La diferencia entre un bucle while y un bucle do while es que el bucle do while ejecuta el bloque de acciones y luego evalúa la condición. En cambio, en un bucle while primero se comprueba la condición y luego se ejecuta el bloque de código.

Uso de break y continue

- La sentencia **break** puede utilizarse para salir de un bucle. Así como también para salir de un caso de la sentencia switch.

BREAK
`break;`

Uso de break y continue

- La sentencia **continue** salta una iteración del bucle, si se da una condición especificada, y continúa con la siguiente iteración en el bucle.

CONTINUE
`continue;`

Fundamentos del lenguaje

¿Qué es una función?

Una función es un conjunto de líneas de código encapsuladas en un bloque, que opcionalmente reciben parámetros, cuyos valores se usan para realizar operaciones y devolver valores.

El objetivo de las funciones es automatizar tareas que requerimos con frecuencia.

VENTAJAS
Separación del código en pequeñas porciones.
Facilidad de mantener.
Facilidad de uso.
Reutilización a futuro.

Declaración de una función

- La sintaxis para declarar una función es la siguiente:

DECLARACIÓN DE UNA FUNCIÓN

```
<acceso> static <tipo> <identificador>(<parámetros>){
   // Bloque de acciones
   return <valor>
}
```

Modificadores de acceso

- Los modificadores de acceso ayudan a restringir el alcance de una clase, constructor, variable, función o método.

Modificador de acceso	Clase	Paquete	Subclase de otro paquete	Todos
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
default	Sí	Sí	No	No
private	Sí	No	No	No

NOTA: Cuando no se especifica un modificador de acceso, por defecto está teniendo el modificador default.

Tipos

- Al declarar una función, debemos indicar el tipo de dato que devuelve dicha función. Puede ser cualquier tipo de dato primitivo (int, boolean, char, ...) o también complejo (String, array, …).

EJEMPLO TIPO SIMPLE

```
public static boolean <identificador>(<parámetros>){
   // Bloque de acciones
   return <valor_booleano>
}
```

Ejemplo

EJEMPLO TIPO COMPLEJO

```
public static String [] <identificador>(<parámetros>){
   // Bloque de acciones
   return <array_strings>
}
```

Parámetros

Los parámetros son los valores que recibe una función para realizar su objetivo.

Debemos tener en cuenta los siguiente detalles sobre los parámetros de las funciones:

- Una función puede tener una cantidad cualquiera de parámetros.
- Si hay más de un parámetro, deben ir separados por comas.
- Cada parámetro tiene un tipo y un identificador. El tipo puede ser cualquiera.

PARÁMETROS

```
<función>(<tipo1> <identificador1>, ..., <tipoN> <identificadorN>){
   // Bloque de acciones
}
```

Argumentos de longitud variable (varags)

Algunas veces tendremos la necesidad de programar una función que tome una longitud variable de parámetros. Para ello deberemos usar las funciones varargs.

SINTAXIS DE VARARGS

```
<función>(<tipo>...<identificador>){
   // Bloque de acciones
}
```

NOTA: Internamente el compilador guarda los parámetros en un array del tipo indicado.

NOTA: Una función puede combinar parámetros normales con un solo parámetro variable. Y este debe ser el último.

Return

- La instrucción **return** permite devolver datos. Cualquier instrucción que se encuentre después de un return no será ejecutada.

RETURN

```
<acceso> static <tipo> <identificador>(<parámetros>){
   // Bloque de acciones
   return <valor>
}
```

NOTA: Es común encontrarse con funciones que tengan múltiples sentencias **return** en el interior de condicionales.

Procedimientos

- Los procedimientos no devuelven valores. Por tanto no tienen un tipo específico y no hacen uso de la sentencia return para devolver valores.

PROCEDIMIENTOS

```
<acceso> static void <identificador>(<parámetros>){
   // Bloque de acciones
}
```

NOTA: Los procedimientos pueden opcionalmente usar la sentencia **return**, pero no con un valor. Se utiliza para finalizar allí la ejecución.

Invocar funciones y procedimientos

- Para invocar una función o un procedimiento simplemente necesitamos el identificador y enviar los valores de los parámetros. El orden y el tipo de los parámetros que enviamos deben coincidir con los declarados.

INVOCAR FUNCIONES O PROCEDIMIENTOS

```
<identificador>(<valor1>, ..., <valorN>);
```

NOTA: Si una función o procedimiento no recibe parámetros, no ponemos nada en el interior de los paréntesis. Pero siempre debemos añadir los paréntesis.

Fundamentos del lenguaje

Lectura de ficheros con FileReader

FileReader es una clase que facilita la lectura de ficheros de caracteres con Java.

IMPORTAR CLASE FILEREADER

```
import java.io.FileReader;
```

Métodos más utilizados

MÉTODO	DESCRIPCIÓN
read()	Lee un solo carácter. Devuelve un -1 si ya no quedan caracteres por leer en el fichero.
close()	Libera los recursos del sistema asociados al fichero.

Para poder hacer uso de las funciones anteriores primero necesitamos crear una “variable” de la clase FileReader.

DECLARAR UNA VARIABLE FILEREADER
`FileReader <identificador> = new FileReader(<nombreFichero>);`

Ejemplo de uso

- Lectura de un carácter:

LECTURA DE UN CARÁCTER

```
char <identificadorCaracter> = (char) <identificadorFichero>.read();
```

- Cierre:

CIERRE

```
<identificadorFichero>.close();
```

Escritura de ficheros con FileWriter

FileWriter es una clase que facilita la escritura de ficheros de caracteres con Java.

IMPORTAR CLASE FILEWRITER

```
import java.io.FileWriter;
```

Métodos más utilizados

MÉTODO	DESCRIPCIÓN
write(<cadena>)	Escribe una cadena de texto en el fichero.
close()	Libera los recursos del sistema asociados al fichero.

Para poder hacer uso de las funciones anteriores primero necesitamos crear una “variable” de la clase FileWriter.

DECLARAR UNA VARIABLE FILEWRITER
`FileWriter <identificador> = new FileWriter(<nombreFichero>);`

Ejemplo de uso

- Escritura de una cadena de texto:

LECTURA DE UN CARÁCTER

```
<identificadorFichero>.write(<cadena>);
```

- Cierre:

CIERRE

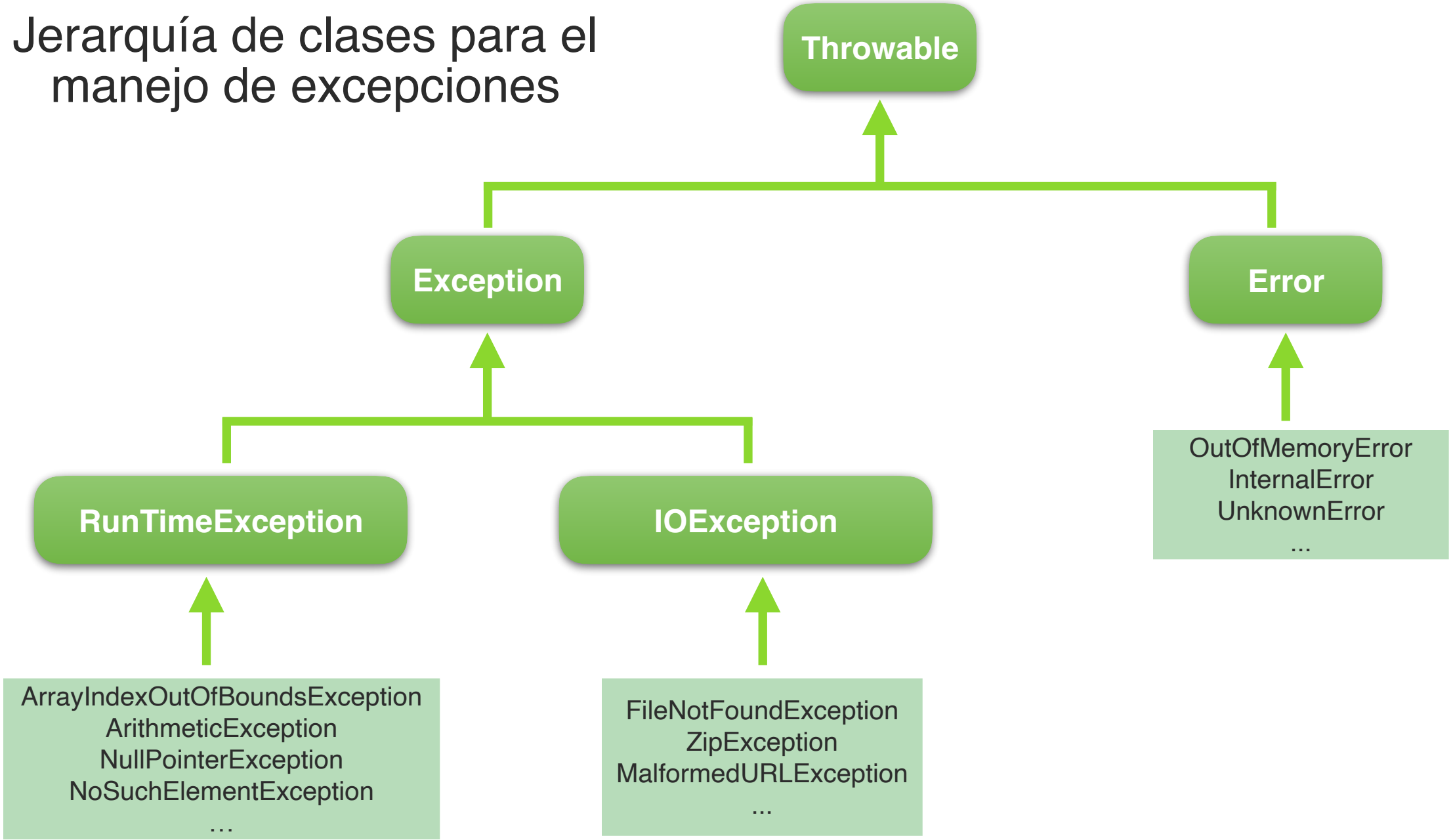
```
<identificadorFichero>.close();
```

Fundamentos del lenguaje

¿Qué es una excepción?

Las excepciones son errores o condiciones anormales que se producen en la ejecución de un programa.

Jerarquía de clases para el manejo de excepciones



Bloques try y catch

El bloque try catch se utiliza para capturar excepciones que se puedan producir en el bloque de código delimitado por try.

BLOQUE TRY CATCH

```
try{
   // Bloque de acciones
}catch(<tipoExcepcion> <identificador>){
   // Bloque de acciones
}
```

NOTA: Para un mismo bloque try, pueden producirse diferentes tipos de excepciones. Por lo que podemos tener más de un catch sucesivo. Aunque debemos tener en cuenta que las cláusulas catch se comprueban en orden.

Uso del finally

En ocasiones puede nos puede interesar ejecutar un bloque de código independientemente de si se produce una excepción o no.

FINALLY

```
try{
   // Bloque de acciones
}catch(<tipoExcepcion> <identificador>){
   // Bloque de acciones
}finally{
   // Bloque de acciones
}
```

Funciones para usar con excepciones

MÉTODO	DESCRIPCIÓN
getMessage()	Extrae el mensaje asociado con la excepción.
toString()	Devuelve un string que describe la excepción.
printStackTrace()	Indica el método donde se lanzó la excepción.

Propagación de excepciones

Si un método lanza alguna excepción (y no utiliza los bloques **try-cath**), en su declaración debemos especificar la lista de los diferentes tipos de excepciones que puede lanzar.

THROWS

```
<metodo> throws <identificadorExcepcion1>, ..., <identificadorExcepcionN>{
   // Bloque de acciones
}
```

NOTA: Se puede poner únicamente una superclase de excepciones para indicar que se pueden lanzar excepciones de cualquiera de sus clases derivadas.

NOTA: No hace falta avisar de que se pueden lanzar excepciones implícitas (RunTimeException o Error).

Sentencia throw

La sentencia throw se utiliza para lanzar objetos de tipo throwable.

THROW

```
throw new <tipoExcepción>(“<mensajeError>”);
```

- También podemos lanzar la excepción en dos pasos:

THROW

```
<tipoExcepcion> <identificador> = new <tipoExcepción>(“<mensajeError>”);
throw <identificador>;
```