

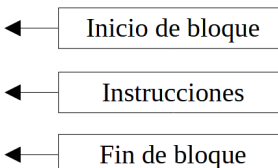
Introducción a la programación

Programar es el proceso de crear un *programa* escrito en un *lenguaje de programación* para que un dispositivo lo *ejecute* para realizar una tarea. Los archivos que escribimos se llaman *código fuente* del programa y el archivo que se genera a partir de este código se llama *archivo ejecutable* del programa. Algunos lenguajes no generan archivos ejecutables, como veremos más adelante, pero en *Java*, que es el lenguaje que usaremos, sí. Pasemos a definir algunos conceptos fundamentales en programación.

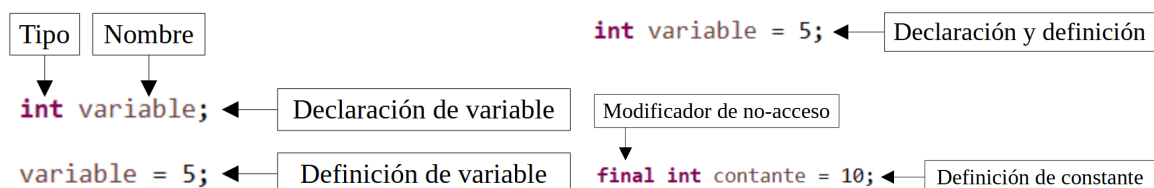
Algoritmo, datos y funciones

Un *algoritmo* es un conjunto de instrucciones ordenadas para resolver una tarea dada. En la vida cotidiana se emplean algoritmos para resolver problemas determinados, como puede ser un manual para ensamblar un mueble o una receta de cocina. En Java los algoritmos cuentan de instrucciones terminadas por punto y como y bloques delimitados por llaves.

```
{  
    int numero;  
    numero = 7;  
}
```

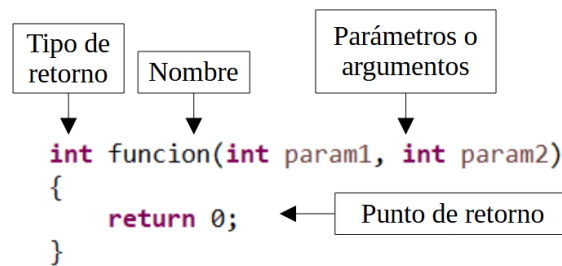


Los *datos* son la información que se le proporciona a un algoritmo para encontrar una solución: se pueden organizar en *estructuras de datos* que los agrupen en una estructura determinada. Los datos se almacenan en *variables* y *constantes*, las primeras permiten que su contenido cambie durante la ejecución del programa y las segundas mantienen el mismo valor todo el tiempo de ejecución. Las variables se declaran al indicar su tipo y nombre, y se definen cuando se les asigna un valor. Las constantes se definen con un valor dado y su definición se precede del modificador de *no-acceso*, *final*.



```
int variable = 5; ← Declaración y definición  
int variable; ← Declaración de variable  
variable = 5; ← Definición de variable  
final int contante = 10; ← Definición de constante
```

Las *funciones* son una porción de código que realiza una tarea específica, estas pueden contar de varios algoritmos distintos y hacer uso o *llamar a* otras funciones. Las funciones tienen un *tipo de retorno* que es el tipo de variable que dan como resultado, un nombre y una lista de *parámetros* o *argumentos* que son los datos que recibe la función para realizar su tarea.



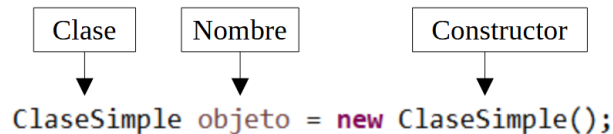
Clases, instancias y objetos

Las *clases* son plantillas genéricas que describen un objeto que encapsula datos y algoritmos relacionados con dicho objeto. Mientras que una *instancia* de dicha clase se refieren a un objeto específico con valores específicos para los datos de la clase. Las funciones que pertenecen a una clase se denominan *métodos*. Los métodos y variables que pertenecen a la clase se llaman *miembros*.

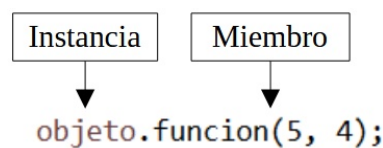
Por otro lado, se le llama *objeto* tanto a una clase como a una instancia de la clase. Ya que en *Java* todo está relacionado con un objeto, se considera un lenguaje de *programación orientada a objetos* (POO).

Por ejemplo, una clase puede ser *silla*, que hace referencia a una silla genérica con sus atributos, y una instancia de esa clase es una silla en particular con sus atributos definidos. Mientras que tanto el concepto de silla como la silla específica pueden referirse como objetos.

Como con las variables, las instancias se declaran con el tipo (clase) y el nombre y se definen asignando un valor, en este caso se usa el operador *new* y la llamada al *constructor* que es un tipo de función especial que pertenece a la clase.



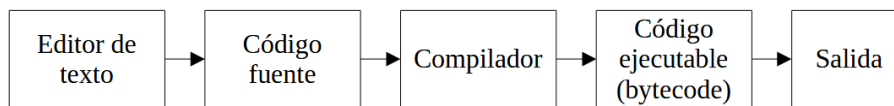
Para acceder a los miembros de un objeto se usa el punto (.).



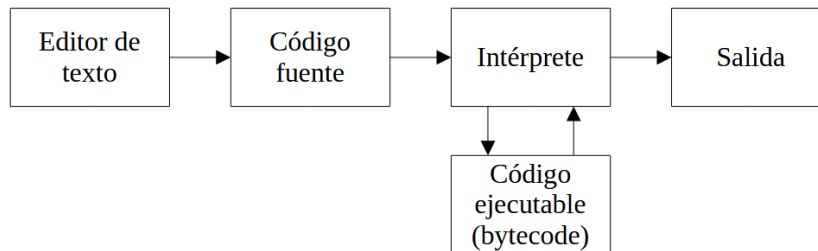
Tipos de lenguajes de programación

El *código fuente* de un programa está escrito en un *lenguaje de programación* específico que luego puede ser compilado o interpretado según el tipo de lenguaje. Algunos lenguajes de programación son: *C*, *C++*, *Java*, *Python*, *Php*, *JavaScript* y *SQL*.

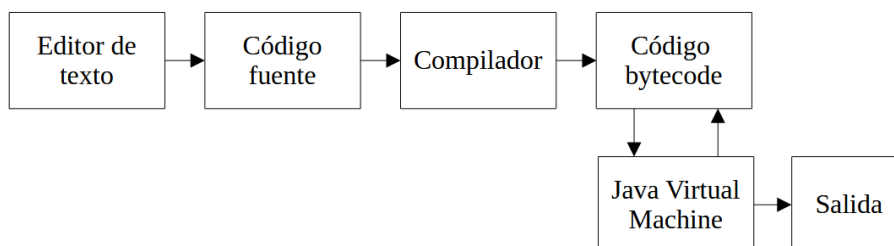
Por un lado, un *lenguaje de programación compilado* es aquel donde el código fuente se traduce, con un programa llamado *compilador*, para generar un *archivo ejecutable* (archivo binario en lenguaje de máquina o bytecode), por ejemplo, en Windows sería un archivo *exe* o, en Android, un archivo *apk*. Una vez compilado, el programa puede ejecutarse sin necesidad de traducir el código nuevamente. Un ejemplo de lenguaje compilado es el *lenguaje C*.



Por otro lado, un *lenguaje de programación interpretado* es aquel donde código fuente no es usado para generar un archivo ejecutable, sino que existe un programa *intérprete* que traduce el código fuente línea por línea a bytecode para ejecutarlo. También son llamados *lenguajes scripting* y a su código fuente, *script*. Un ejemplo de lenguaje interpretado es *Python*.



Java es considerado un *lenguaje intermedio*, ya que el código fuente se compila completamente para generar un archivo como en un lenguaje compilado. La diferencia es que el compilador de Java no produce un ejecutable con código máquina que el sistema operativo pueda cargar directamente para que el microprocesador lo ejecute. En cambio, se genera un archivo binario (*.class*) que es ejecutado (interpretado) por la *Máquina Virtual de Java*, *JVM* (*Java Virtual Machine*) que debe estar instalada en el sistema operativo.



Para ejecutar un programa *Java* en Windows debe estar instalada la JVM que se instala con el programa *Java Runtime Environment (JRE)*, por otro lado, Android convierte esto binarios en otro tipo de archivo (*.dex*) que puede interpretar su propia máquina virtual *Dalvik Virtual Machine (DVM)*.

Pasos del desarrollo de un programa

Más ampliamente podemos decir que *programar* es el proceso de crear un software (programa) mediante la escritura, puesta a punto, compilación o interpretación, y documentación del código fuente de dicho programa.

Desarrollo lógico

Este paso implica planificar como va a funcionar el programa y el flujo que sigue para resolver un problema.

Codificación

Este paso es la escritura del código fuente en el lenguaje de programación que hayamos elegido.

Compilación o interpretación

Compilación o interpretación del programa según corresponda si es un lenguaje compilado o interpretado.

Depuración

Este paso implica la búsqueda y corrección de errores del programa, es decir, la prueba y depuración (*debug*) del mismo.

Documentación

Esta puede realizarse en un documento aparte o en el mismo código fuente con comentarios, que es texto que no se interpreta como código. En Java hay dos tipos de comentarios según si ocupa una línea o varias. Los primeros van luego de una doble barra y los segundos entre barra y asterisco.

```
// Comentario en una línea

/* Comentario que
 * ocupa varias
 * líneas
 */
```

Los comentarios de documentación, también conocidos como *Javadoc*. Por medio de herramientas externas, podremos generar de forma automática la documentación de un proyecto Java, a partir de estos comentarios de documentación o *Javadocs*.

Estos comentarios, a diferencia de los comentarios multilínea, van cerrados entre `/**` y `*/`, decir que inician con `/**` en lugar de `/*`. Además, se recomienda que cada línea del bloque empiece con `*`.

Estos comentarios requieren etiquetas, que comienzan con `@`, para indicar los componentes del código fuente (parámetros, tipos de retorno, etc.). Los más comunes son `@param` y `@return`, para indicar comentario sobre un parámetro y valor de retorno de una función respectivamente.

```
/**
 * Descripción de la función
 * @param param1 Comentario sobre el primer parámetro
 * @param param2 Comentario sobre el segundo parámetro
 * @return Comentario sobre el valor de retorno
 */
private int funcion(int param1, int param2)
{
    return 0;
}
```

Entorno de desarrollo

Un entorno de desarrollo integrado o *IDE* (Integrated Development Environment) es un programa que facilita al programador el desarrollo de software. Normalmente, un IDE consiste de un editor donde escribir el código fuente, herramientas de reemplazo y generación de código automáticas y un depurador (*debugger*) para la búsqueda de errores. La mayoría de los IDE tienen autocompletado inteligente de código, un compilador y/o un intérprete. Algunos IDE son *Eclipse*, *Android Studio*, *Visual Studio* y *Visual Studio Code*.

Estructura de un programa en Java

Paquete (*package*)

Cada programa Java puede, opcionalmente, comenzar con una declaración de paquete. Los paquetes agrupan clases relacionadas y ayudan a organizar el código. El uso de paquetes ayuda a evitar conflictos de nombres entre clases y reutilizar de código.

```
package com.cursojava;    // Nombre del paquete
```

Importaciones (*import*)

Después de la declaración de paquete, se pueden incluir importaciones. Estas permiten utilizar clases de otros paquetes. Esto permite acceder a clases y métodos de bibliotecas externas sin necesidad de escribir el nombre completo del paquete.

```
import java.util.Scanner;    // Importa la clase Scanner del paquete java.util
```

Clase principal y método de entrada (*main*)

Todo programa Java necesita al menos una clase. La clase principal es el punto de entrada del programa, y debe contener el método *main* que es el punto de inicio de ejecución del programa. Un programa mínimo, que no haga nada, pero que se pueda ejecutar, sería el siguiente:

```
public class SimpleJavaApp
{
    // Punto de entrada del programa, método main
    public static void main(String[] args)
    {
        // Nuestro código
    }
}
```

Las sintaxis y palabras claves usadas aquí las veremos en detalle más adelante.

Tipos y estructuras de datos

Cuando declaramos una variable o constante debemos darle un nombre y un tipo de dato que pueda almacenar. Los tipos de datos básicos son:

- Entero: Número sin parte decimal. Ejemplo de uso: contadores, edad, año.
- Flotante: Número con parte decimal. Ejemplo de uso: peso, distancia, precio.
- Caracter: Representa una letra o símbolo. Ejemplo de uso: departamento, opción en una lista.
- Booleano: Valor de dos estados: verdadero o falso. Ejemplo de uso: opciones binarias.

En Java, las palabras reservadas para los tipos de datos comunes o *primitivas*, son:

Nombre	Tipo	Tamaño	Rango de valores
byte	Entero	1 byte	-128 a 127
short	Entero	2 bytes	-32.768 a 32.767
int	Entero	4 bytes	-2^{31} a $2^{31}-1$
long	Entero	8 bytes	-2^{63} a $2^{63}-1$
float	Flotante	4 bytes	1.4×10^{-45} a 3.4×10^{38} (aproximado)
double	Flotante	8 bytes	4.9×10^{-324} a 1.8×10^{308} (aproximado)
boolean	Booleano	4 bytes	Verdadero (true) o Falso (false)
char	Carácter	2 bytes	Caracteres Unicode UTF-16

A las variables se las *declara* cuando se les da un nombre y un tipo y se *definen* cuando se les asigna un valor por primera vez. Veamos algunos ejemplos:

```
byte bData = 12;
char cPiso = 'A';           // Caracter
int iCont = 0;              // Entero
short iMes = 7;             // Entero corto
long lNumSerie = 72845741L; // Entero largo
float fPeso = 87.5f;        // Flotante
double fPrecio = 15.2;      // Flotante de doble precisión
boolean bSelected = true;   // Booleano
```

Cuando se asigna un valor a un caracter se lo hace entre apóstrofes, cuando se quiere indicar que un valor es flotante simple se usa la letra *f* al final del mismo y cuando se quiere indicar que un valor es *long* se coloca una *L*.

Además, existen tipos de datos compuestos:

- Lista (array): Los arrays son una colección ordenada de elementos del mismo tipo que se pueden acceder mediante un índice.

- Matriz (array de dos dimensiones): Colección de elementos del mismo tipo organizados en dos dimensiones (filas y columnas) donde cada elemento puede accederse por su fila y columna.
- Cadena de caracteres: Arrays de caracteres. En Java se manejan con un objeto tipo `String`.
- Enumeración: Conjunto de constantes que se pueden acceder por un nombre dado.

```
// Declaración del enum eNombres que tiene una lista fija de nombres
enum eNombres { JUAN, PEDRO, ANA };

public static void main(String[] args)
{
    // Array con letras que representan los días de la semana
    char[] arrDias = { 'L', 'M', 'X', 'J', 'V', 'S', 'D' };
    // Array con dos filas de números, la primera con números impares
    // y la segunda con números pares
    int [][]mat = { {1, 3, 5}, {2, 4, 6} };
    // Cadena de caracteres con la frase: Hola mundo
    String sSaludo = "Hola mundo";
    // Definición de eUsuario con el valor JUAN que pertenece al
    // enum eNombre que declaramos previamente declarado
    eNombres eUsuario = eNombres.JUAN;
```

Para indicar que nos referimos a una constante y no a una variable, usamos la palabra reservada *final*. Por ejemplo:

```
final long lClaveNum = 135791; // Entero largo contante
```

Con estos tipos de datos podemos realizar gran parte del trabajo que necesitemos en nuestro programa, pero también hay otras formas de organizar los datos, llamadas *estructuras de datos*.

Conversión de datos

La conversión de datos en programación se llama *cast*, y en Java se realiza anteponiendo el tipo al que queremos *castear* entre paréntesis. En algunos casos esto no es necesario, ya que se hace automáticamente. Por ejemplo, si asignamos una variable entera a otra de mayor capacidad, pero no al revés.

```
char cChar = 'B';
long lLong = 65452114L;

cChar = lLong;           // Esto es incorrecto
cChar = (char)lLong;     // Esto es correcto
lLong = cChar;           // Esto también es correcto
```

Cuando casteamos a un tipo de datos de menor tamaño, lógicamente el valor se recorta para ajustarse, por ejemplo:

```
double fDoble = 18.124;
int iEntero = (int) 18.724; // En iEntero se guarda 18
```

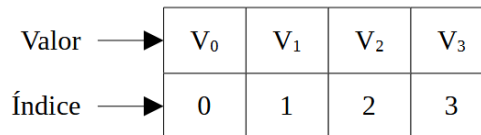
Es decir, si asignamos un flotante a un entero, este se trunca y se guarda solo la parte entera del mismo.

Estructuras de datos

Las *estructuras de datos* son formas específicas de organizar y almacenar datos para facilitar su manipulación y acceso. Veamos algunas de las más usadas:

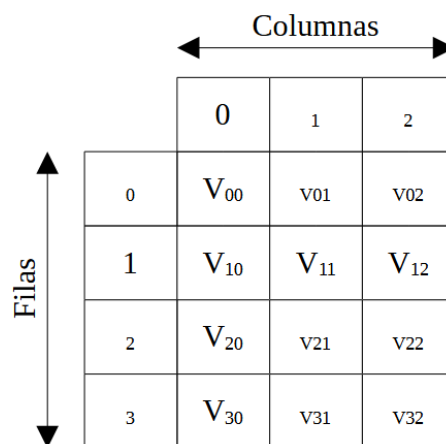
Arreglo (*array*)

Los *arrays* son colecciones ordenadas de elementos del mismo tipo y accesibles mediante un índice.



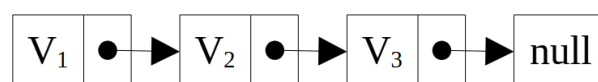
Matriz (*matrix*)

Las *matrices* son estructuras bidimensional que organizan datos en filas y columnas.



Listas enlazadas

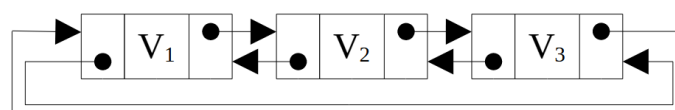
Las *listas enlazadas* son secuencias de *nodos*, donde cada nodo contiene un valor y una referencia al siguiente.



También existen las *listas doblemente enlazadas* donde cada nodo apunta al nodo siguiente y al anterior.

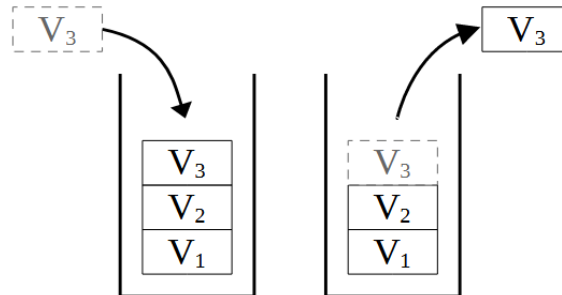


Y finalmente, las *listas circulares* son listas doblemente enlazadas donde el último nodo apunta al primero y viceversa.



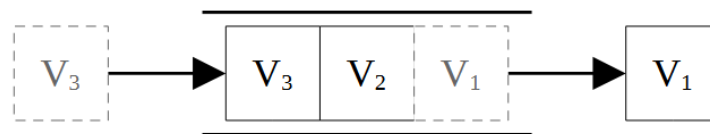
Pila (stack)

Colección de elementos apilados donde se puede acceder o eliminar el último elemento agregado (el que está sobre la pila), a este sistema se lo llama *LIFO* (*Last In, First Out*), que significa que el último valor en entrar a la pila será el primero en salir.



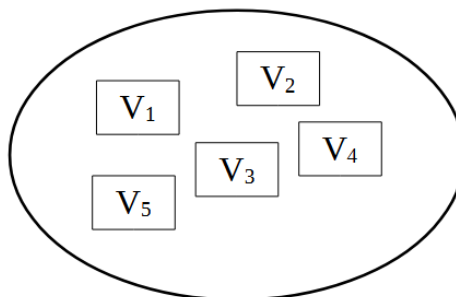
Cola (queue):

Colección de elementos encolados donde el primer elemento agregado será el primero en ser accedido, a este sistema se lo llama *FIFO* (*First In, First Out*), ya que el primer elemento en entrar a la cola será el primero en salir.



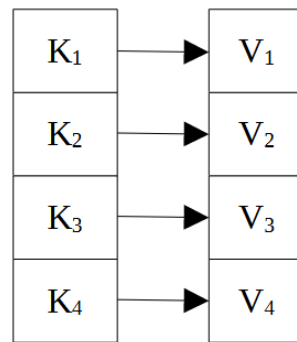
Conjunto (set)

Colección de elementos únicos sin un orden específico.



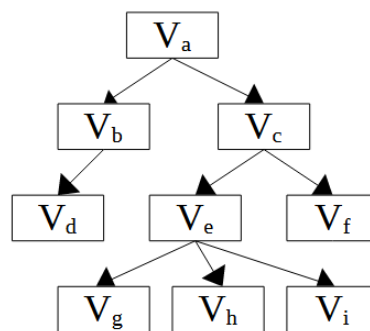
Diccionario (map)

Almacena pares de clave-valor, donde cada clave es única y asociada a un valor, o sea, que cada valor se puede acceder con una clave única.



Árbol (tree)

Estructura jerárquica que consta de nodos conectados por bordes. Incluye árboles binarios, árboles de búsqueda, etc.



Manejo de cadenas de caracteres en Java

En Java, las cadenas de caracteres se manejan con la clase *String*, que sirve para almacenar y manipular las mismas. Los textos entrecomillados son tratados como objetos de la clase *String* y puede asignarse a objetos de este tipo. La clase *String* puede contener todo tipo de caracteres UNICODE.

```
String str;  
  
str = "A";  
str = "Texto largo";  
str = "user@mail.com";
```

Podemos inicializar el *String* con el operador *new* o directamente asignando la cadena de caracteres.

```
String str1 = new String("Algún texto");  
String str2 = "Algún otro texto";
```

En *Java* las cadenas se pueden concatenar usando el símbolo de suma (+).

```
String str1 = new String("Hola ");
String str2 = "Mundo";
String str3 = str1 + str2;

System.out.println(str3);

// Es equivalente a hacer lo siguiente
System.out.println("Hola " + str2);
System.out.println(str1 + "mundo");
System.out.println("Hola " + "mundo");
```

Salida

```
Hola Mundo
Hola Mundo
Hola mundo
Hola mundo
```

Para mostrar una cadena por consola usaremos la función *println* de la clase *System.out*. Explicaremos más sobre la clase y sus funciones en el capítulo de entrada y salida de datos. La concatenación también se puede hacer con el método *concat*.

```
str3 = str1.concat(str2);
// Guardando el resultado en str3
System.out.println(str3);
// O mostrando directamente
System.out.println(str1.concat(str2));
```

Salida

```
Hola Mundo
Hola Mundo
```

String ofrece varios métodos para obtener información de las cadenas y compararlas con otras.

Método	Descripción
<code>cadena.length()</code>	Devuelve el número de caracteres que contiene la cadena
<code>cadena.isEmpty()</code>	Comprueba si la cadena está vacía
<code>cadena1.equals(cadena2)</code>	Compara si una cadena es igual a otra
<code>cadena1.compareTo(cadena2)</code>	Compara dos cadenas
<code>cadena1.contains(cadena2)</code>	Comprueba si la cadena contiene otra indicada
<code>cadena.indexOf(ch)</code>	Devuelve la posición que ocupa un carácter en la cadena

Veamos algunos ejemplos de uso.

```
String str1 = "usuario@dominio.com";
String str2 = "usuario@dominio2.com";
String str3 = "usuario@dominio.com";
String str4 = "";

// Cantidad de caracteres
System.out.println("str1 tiene " + str1.length() + " caracteres");
System.out.println("str2 tiene " + str2.length() + " caracteres");
// Comprobación de contenido vacío
System.out.println("str3 vacía: " + str3.isEmpty());
System.out.println("str4 vacía: " + str4.isEmpty());
// Comprobación de igualdad
System.out.println("str1 es igual a str2: " + str1.equals(str2));
System.out.println("str1 es igual a str3: " + str1.equals(str3));
```

Salida

```
str1 tiene 19 caracteres
str2 tiene 20 caracteres
str3 vacía: false
str4 vacía: true
str1 es igual a str2: false
str1 es igual a str3: true
```

Veamos como funciona la comparación de cadena con un ejemplo:

```
String str1 = "Almendras";
String str2 = "Banana";
String str3 = "Calabaza";

System.out.println("str1 comparad con str2: " + str1.compareTo(str2));
System.out.println("str3 comparad con str2: " + str3.compareTo(str2));
System.out.println("str1 comparad con str1: " + str1.compareTo(str1));
```

Salida

```
str1 comparad con str2: -1
str3 comparad con str2: 1
str1 comparad con str1: 0
```

Si la cadena que llama al método es anterior a la del argumento se retorna -1, si es posterior se retorna 1 y si son iguales se retorna 0. Veamos un ejemplo de los dos métodos de búsqueda dentro de la cadena.

```
String str1 = "Catarata";

System.out.println(str1 + " contine 'rata': " + str1.contains("rata"));
System.out.println("Posición de 'rata' en " + str1 + ": " + str1.indexOf("rata"));
System.out.println(str1 + " contine 'gato': " + str1.contains("gato"));
System.out.println("Posición de 'gato' en " + str1 + ": " + str1.indexOf("gato"));
System.out.println("Posición de 'a' " + str1 + ": " + str1.indexOf('a'));
```

Salida

```
Catarata contine 'rata': true
Posición de 'rata' en Catarata: 4
Catarata contine 'gato': false
Posición de 'gato' en Catarata: -1
Posición de 'a' Catarata: 1
```

Como en el caso de los arrays, la primera posición es cero (0). Pasemos a ver otros métodos muy usados que ofrece la clase *String*.

Método	Descripción
<code>cadena.toLowerCase()</code>	Devuelve la cadena en minúsculas
<code>cadena.toUpperCase()</code>	Devuelve la cadena en mayúsculas
<code>cadena.substring(i)</code>	Devuelve sub cadena desde la posición <i>i</i> al final
<code>cadena.substring(i, f)</code>	Devuelve sub cadena desde <i>i</i> , inclusive, hasta antes de <i>f</i>
<code>cadena.replace(s1, s2)</code>	Sustituye todas las apariciones de <i>s1</i> por <i>s2</i>

Ejemplos de uso

```
String str1 = "Jirafa";

System.out.println(str1 + " en mayúsculas: " + str1.toUpperCase());
System.out.println(str1 + " en minúsculas: " + str1.toLowerCase());

System.out.println(str1 + " sub cadena 1: " + str1.substring(2));
System.out.println(str1 + " sub cadena 2: " + str1.substring(2, 6));

System.out.println(str1 + " reemplazo de cadena: " + str1.replace("Ji", "Gar"));
```

Salida

```
Jirafa en mayúsculas: JIRAFa
Jirafa en minúsculas: jirafa
Jirafa sub cadena 1: rafa
Jirafa sub cadena 2: rafa
Jirafa reemplazo de cadena: Garrafa
```

Operadores

En Java, como en todos los lenguajes, existen operadores, que son símbolos que permite realizar operaciones entre valores, y se pueden dividir en: aritméticos, de asignación, de comparación y lógicos.

Operadores aritméticos

Son aquellos símbolos que nos permiten hacer operaciones o cálculos simples. Los operadores de decremento (--) e incremento (++), suman o restan 1 al valor de la variable antes o después de usarla, según si están antes o después de la misma respectivamente.

Operador	Acción	Ejemplo	Resultado
-	Resta	X = 5 - 3;	2
+	Suma	X = 2 + 3;	5
*	Multipliación	x = 2*3;	6
/	División	x = 6/2;	3
%	Módulo (resto entero)	x = 5%2;	1
--	Decremento	x = 1; x--;	0
++	Incremento	x = 1; x++;	2

Operadores de asignación

Los operadores de asignación se utilizan para asignar valores a variables. En Java, el operador de asignación básico es =, pero también existen operadores compuestos que realizan una operación y luego asignan el resultado a la variable en una sola instrucción.

Operador	Acción	Ejemplo	Resultado (dado X = 6)
-=	Asigna Resta	X -= 1;	5
*=	Asigna Producto	X *= 5;	30
/=	Asigna División	X /= 2;	3
+=	Asigna Suma	X += 4;	10

Operadores relacionales o de comparación

Permiten evaluar si dos variables son iguales, diferentes o menores, a su vez permite comprobar o bien probar la veracidad de una condición, la respuesta es de valor booleano.

Operador	Acción	Ejemplo
<	Menor que	x = 5 y=3; (y < x)
>	Mayor que	x =7 y=9; (y> x)
<=	Menor o igual que	x = 5 y= 3; (y <= x)
>=	Mayor o igual que	x =7 y=5; (x >= y)
==	Igual a	x = 5 y =5; (x == y)
!=	Distinto a	X=9 y=5; (x != y)

Operadores lógicos

Producen un resultado booleano relacionando distintos valores booleanos o relaciones que generen un valor booleano. Funcionan de la siguiente manera:

Operador	Acción	Ejemplo
&&	Da verdadero si ambos lados son verdadero	C && D
	Da verdadero si un lado es verdadero	C D
!	Negación	!C

Operadores binarios

Los operadores binarios permiten manipular los números bit a bit.

Operador	Descripción
&	Realiza la operación <i>AND</i> entre bits
^	Realiza la operación <i>XOR</i> entre bits
	Realiza la operación <i>OR</i> entre bits
~	Realiza la operación <i>NOT</i> a los bits
<<	Operador desplazamiento izquierda
>>	Operador desplazamiento derecha
>>>	Operador desplazamiento derecha sin signo

Ejemplos de uso:

```
int num1, num2;
int res;

num1 = 9;           // en binario es 0000 1001
num2 = 10;          // en binario es 0000 1010
res = num1 & num2;   // en binario es 0000 1000 = 8

num1 = 9;           // en binario es 0000 1001
num2 = 10;          // en binario es 0000 1010
res = num1 | num2;   // en binario es 0000 1011 = 11

num1 = 9;           // en binario es 0000 1001
num2 = 10;          // en binario es 0000 1010
res = num1 ^ num2;   // en binario es 0000 0011 = 3

num1 = 9;           // en binario es 0000 1001
res = num1 << 2;     // num1 queda en 0010 0100 = 36

num1 = 9;           // en binario es 0000 1001
res = num1 >> 1;     // num1 queda en 0000 0100 = 4
```

Para otras operaciones más complejas como raíces cuadradas y potenciación debemos hacer uso de la clase `Math` que veremos a continuación.

Clase Math

La clase *Math* es una clase de utilidad, lo que significa que todos sus métodos y constantes son estáticos (declarados como *static*), lo que nos permite acceder a ellos directamente sin necesidad de crear una instancia de la clase.

Listemos algunos de los métodos más usados:

Método	Descripción
<code>abs(x)</code>	Retorna el valor absoluto de x
<code>max(x, y)</code>	Retorna el mayor entre x e y
<code>min(x, y)</code>	Retorna el menor entre x e y
<code>random()</code>	Retorna un número aleatorio (<i>double</i>) entre 0 y 1
<code>sqrt(x)</code>	Retorna la raíz cuadrada de x
<code>pow(x, y)</code>	Retorna x elevado a la potencia de y
<code>round(x)</code>	Redondea x al entero más cercano
<code>ceil(x)</code>	Redondea x al menor entero mayor o igual a x .
<code>floor(x)</code>	Redondea x al mayor entero menor o igual a x .

Ejemplos

```
System.out.println("abs(-5): " + Math.abs(-5));
System.out.println("abs(7): " + Math.abs(7));
System.out.println();
System.out.println("Math.max(11, 5): " + Math.max(11, 5));
System.out.println("Math.min(11, 5): " + Math.min(11, 5));
System.out.println();
System.out.println("Math.round(9.38): " + Math.round(9.38));
System.out.println("Math.round(13.56): " + Math.round(13.56));
System.out.println("Math.ceil(15.53): " + Math.ceil(15.53));
System.out.println("Math.floor(15.53): " + Math.floor(15.53));
System.out.println();
System.out.println("Math.sqrt(4): " + Math.sqrt(4));
System.out.println("Math.pow(2, 3): " + Math.pow(2, 3));
System.out.println();
System.out.println("Math.random(): " + Math.random());
System.out.println("Math.random(): " + Math.random());
System.out.println("Math.random(): " + Math.random());
```

Salida

```
abs(-5): 5
abs(7): 7

Math.max(11, 5): 11
Math.min(11, 5): 5

Math.round(9.38): 9
Math.round(13.56): 14
Math.ceil(15.53): 16.0
Math.floor(15.53): 15.0

Math.sqrt(4): 2.0
Math.pow(2, 3): 8.0

Math.random(): 0.6214480036061215
Math.random(): 0.9961763466552057
Math.random(): 0.4620313343089767
```

Estructuras de control

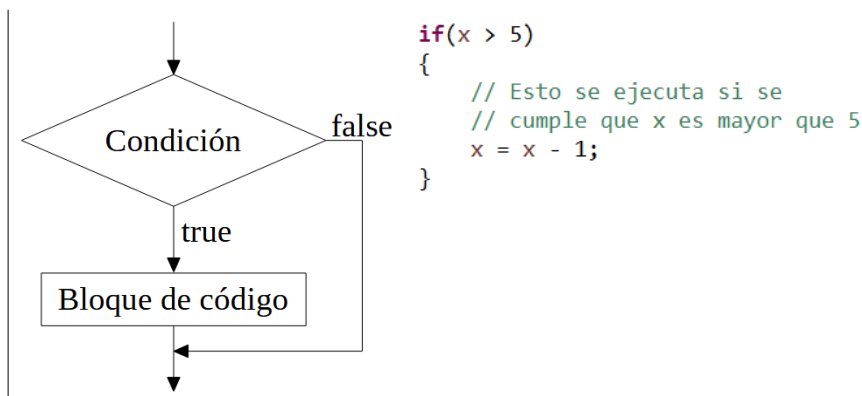
Las estructuras de control son el conjunto de reglas que permiten controlar el flujo de ejecución de un programa. Las estructuras de control tienen un único punto de entrada y se pueden clasificar en: condicionales e iterativas (bucles).

Estructuras condicionales o de control de selección

Las estructuras condicionales ejecutan un bloque de instrucciones u otro, según se cumpla o no una condición. En Java estas estructuras son:

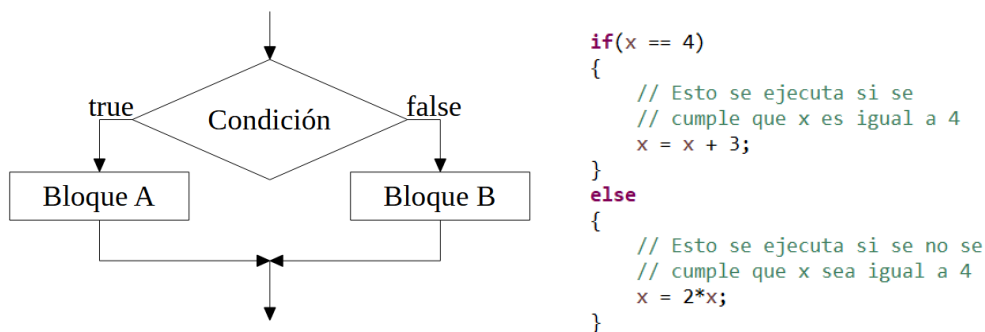
Si simple (if)

Permite que un bloque de código se ejecute o no según se cumpla o no una condición.



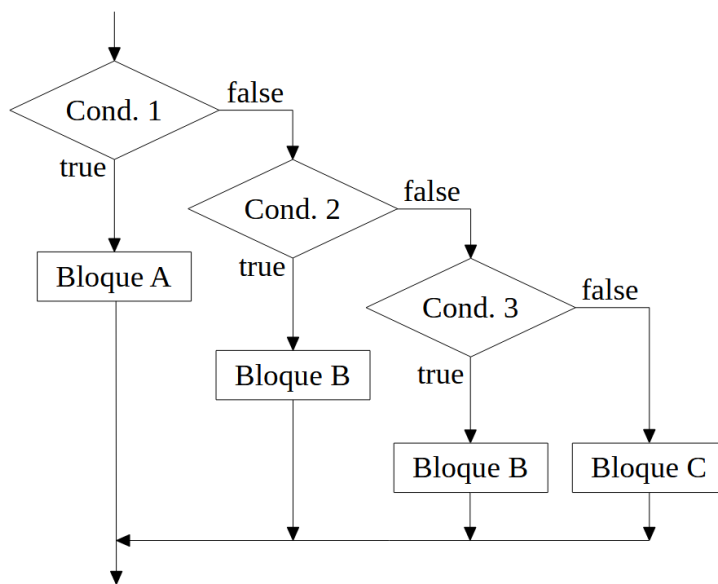
Si-Sino (if-else)

Permite que según se cumpla o no una condición se ejecuten dos diferentes bloques de código.



Si múltiple (if-else if-else)

Permite que según se ejecuten múltiples bloques de código según se cumplan diferentes condiciones.

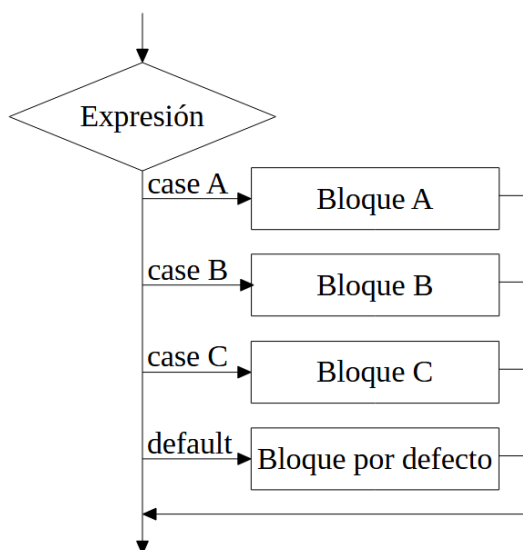


```

if(x < 0)
{
    // Se ejecuta si se x es negativa
    y = -1*x;
}
else if (x > 0)
{
    // Se ejecuta si se x es positiva
    y = x;
}
else
{
    // Se ejecuta si se x es cero
    y = 0;
}
  
```

Selector por caso (switch case)

Permite ejecutar uno de entre varios bloques de código según el valor de una expresión. Es una alternativa a si múltiple cuando se compara la misma expresión con diferentes valores.



```

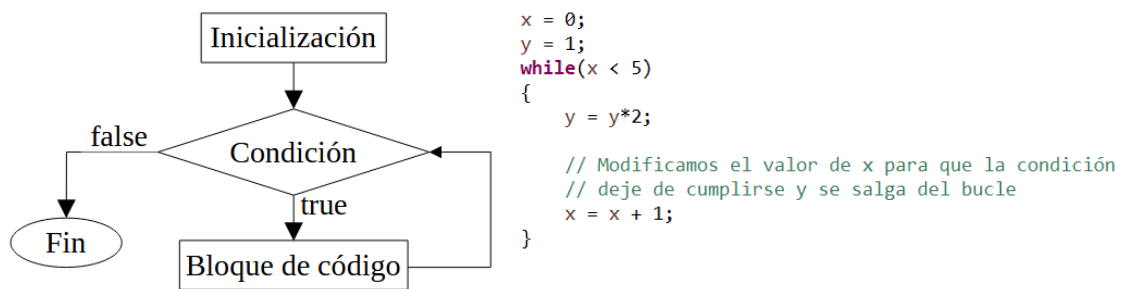
switch(x)
{
    case 0:
    {
        // Esto se ejecuta si x vale 0
        y = 0;
    }
    break;
    case 1:
    {
        // Esto se ejecuta si x vale 1
        y = x*2;
    }
    break;
    default:
    {
        y = x/10;
        // Esto se ejecuta en cualquier otro caso
    }
}
  
```

Estructuras de control de flujo o bucles

Las estructuras de control de flujo o bucles sirven para ejecutar un conjunto específico de instrucciones múltiples veces. La cantidad de veces que se repite dicho bloque de acciones puede ser fijado de antemano o puede depender del valor de alguna variable o de alguna condición específica que debe cumplirse antes de que se finalice el bucle.

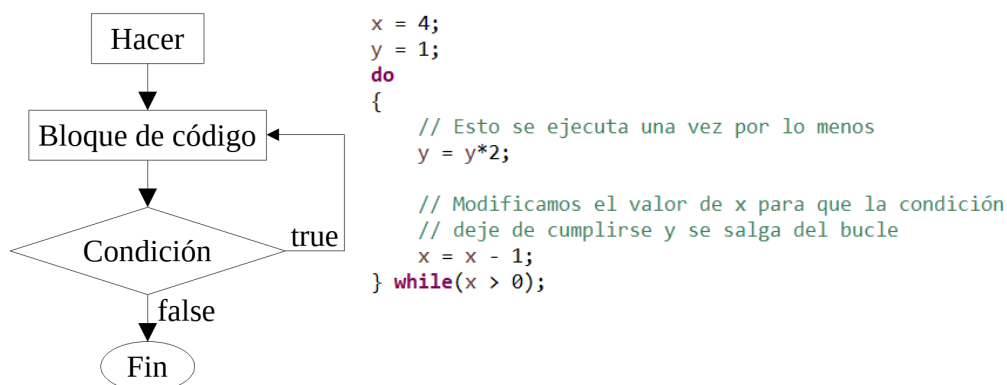
Mientras (while)

El bucle *while* permite repetir un bloque de código mientras se cumpla una condición dada, es decir, mientras la expresión lógica se mantenga verdadera.



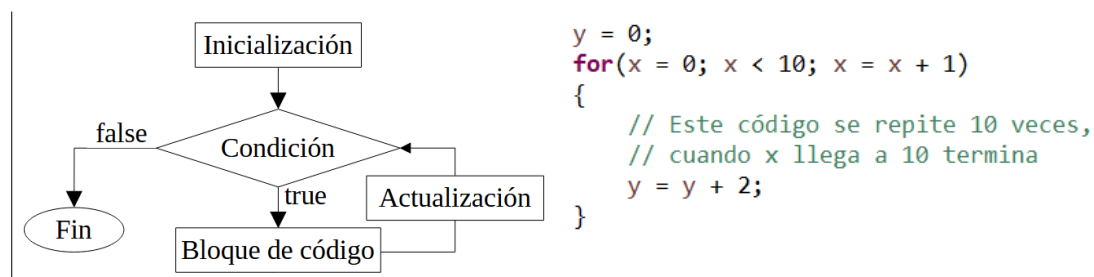
Hacer mientras (do while)

Es similar al bucle *while* con la diferencia que la condición se comprueba al final, por lo tanto, el bloque de código se ejecuta siempre una vez por lo menos.



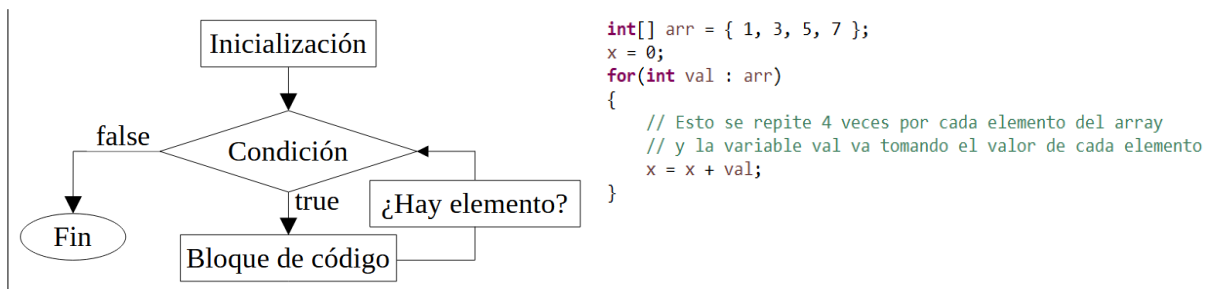
Para (for)

El bucle *for* permite repetir un bloque de código (iterar) una cantidad predefinida de veces. En este bucle hay lugar para inicializar una variable, comprobar su valor e incrementarla o decrementarla.



Para cada (for)

Este es un uso particular del bucle *for* que se usa en arrays. En este caso el bucle se repite una vez por cada elemento que tenga el array.



Instrucciones de control de flujo

Las instrucciones de control de flujo modifican la forma en que se ejecuta el código haciendo que se fuerce, continúe o interrumpa el flujo. En Java tenemos las siguientes:

- Interrumpir (*break*): Sirve para salir de una estructura. Por ejemplo, dentro de un bucle se puede poner para salir del mismo si se da una determinada condición.
- Continuar (*continue*): Sirve para continuar desde el principio de un bucle omitiendo el código que viene después.
- Salir del proceso (*return*): Sirve para salir de la función que se está ejecutando actualmente desde cualquier parte de su código. Veremos más su uso cuando veamos funciones.

```

// Este for no se va a ejecutar 10 veces sino 7, ya que si se
// cumple la condición se sale del mismo con un break
for(int i = 0; i < 10; i++)
{
    if(i == 7)
    {
        break;
    }
}

// Este for no se va a ejecutar 10 veces sino 5, ya que si se
// cumple la condición se le asigna 10 a i y al volver al principio
// se cumple la condición de salida
for(int i = 0; i < 10; i++)
{
    if(i == 5)
    {
        i = 10;
        continue;
    }
}
  
```

Funciones, métodos y procedimientos

Diferencia entre función, método y procedimiento

Es muy común entre programadores que se hable indistintamente de estos tres términos, pero tienen diferencias fundamentales.

Funciones: Las funciones son un conjunto de líneas de código (instrucciones), encapsuladas en un bloque, usualmente reciben parámetros o argumentos, cuyos valores se utilizan para efectuar operaciones y luego retornan un valor. En otras palabras, una función puede recibir parámetros o argumentos (algunas no reciben nada), hace uso de dichos valores recibidos y retorna un valor usando la instrucción *return*, si no retorna algo, entonces no es una función. En Java las funciones usan el modificador *static*.

Métodos: Los métodos y las funciones en Java pueden realizar las mismas tareas, es decir, son funcionalmente idénticos. La diferencia radica en que un método está asociado siempre a un objeto, es decir, es una función que pertenece a un objeto, mientras que una función puede usarse sin la necesidad de una instancia de una clase.

Procedimientos: Los procedimientos son básicamente un conjunto de instrucciones que se ejecutan sin retornar ningún valor. En Java un procedimiento es básicamente un método cuyo tipo de retorno es *void* que no nos obliga a utilizar una sentencia *return*.

Función	Método	Procedimiento
Pertenece a una clase	Pertenece a un objeto	Pertenece a un objeto
Puede o no recibir argumentos	Puede o no recibir argumentos	Puede o no recibir argumentos
Retorna un resultado	Retorna un resultado	No retorna un resultado

Más allá de estas diferencias, es común referirse siempre a funciones.

Ámbito de las variables

El *ámbito* de una variable es la parte del código donde existe y puede ser usada. Fuera de su ámbito, la variable no existe y, por tanto, no podemos usarla. La vida de una variable consta de tres etapas:

- Declaración de la variable.
- Uso de la variable.
- Destrucción de la variable.

La variable existe desde que se crea, hasta que se acaba el bloque en el que está declarada.

```
// Aquí NO está disponible
{
    // Aquí TAMPOCO no está disponible
    int iMiVariable = 5;

    // Aquí SÍ o está disponible
    {
        // En este bloque interior TAMBIÉN está disponible
    }
    // Aquí NO está disponible, ya se destruyó
}
```

Funciones recursivas

Una función recursiva es una función que se llama a sí misma para resolver un problema de manera repetitiva. Siempre debe haber una condición para que deje de llamarse a sí misma y no bloquear el programa.

Un ejemplo clásico es el cálculo del factorial de un número. El factorial de un número entero positivo se define como el producto de todos los números enteros positivos desde 1 hasta dicho número, es decir: $n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$

Para calcular el factorial de un número podemos hacer ese número por el factorial del número anterior. Por lo que podemos usar una función recursiva.

```
package com.cursojava;

public class FactorialApp
{
    public static void main(String[] args)
    {
        int res = factorial(5);

        System.out.println("resultado: " + res);
    }

    public static int factorial(int num)
    {
        if(num < 0)
        {
            // num nunca debería ser negativo, retornamos 0
            // y debería tomarse como un error
            return 0;
        }
        else if(num == 0 || num == 1)
        {
            return 1;
        }
        else
        {
            int result = num*factorial(num - 1);
            return result;
        }
    }
}
```

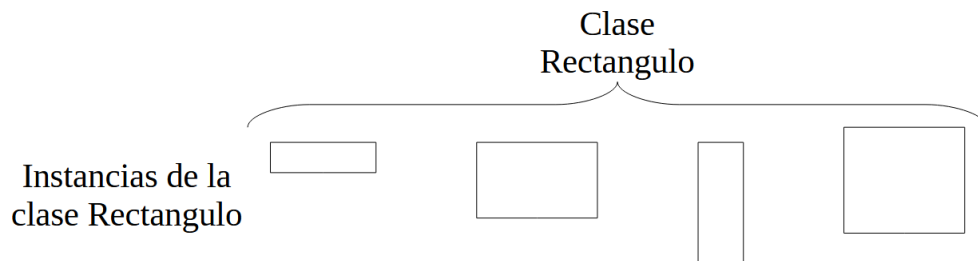
Otro uso frecuente de la recursividad es para procesos que implican recorrer el contenido de un directorio para procesar los archivos.

Clases y objetos

Clases y objetos

Como definimos anteriormente, las clases son plantillas genéricas, mientras que una instancia se refiere a un objeto específico de esa clase.

Por ejemplo, si pensamos en un rectángulo, no pensamos en ninguno en particular, sino en la idea del mismo, esto sería una *clase*. Pero si nos referimos a un rectángulo con un alto y ancho dado, estamos hablando de una instancia de la clase rectángulo.



Cada elemento de la clase se llama miembro, y puede ser, por ejemplo, una variable destinada a almacenar un dato de la clase, una constante, una función que realice alguna tarea con los datos de la clase e incluso instancias de otras *clases*. Los miembros se acceden con el operador punto (.) en la forma: *instancia.miembro*.

Visibilidad de los miembros

Cuando creamos una clase declaramos cada uno de sus miembros y además de indicar el tipo de dato, también podemos indicar la visibilidad, es decir, si puede ser o no accedido desde otros archivos. Para eso usamos los modificadores de acceso que son las palabras reservadas:

Público (*public*): El miembro de la clase es accesible desde cualquier lugar del código.

Privado (*private*): El miembro es accesible sólo desde la propia clase.

Protegido (*protected*): El miembro es accesible por la propia clase y las clases descendientes.

```
package com.cursojava;

@SuppressWarnings("unused")
public class Persona
{
    // Constante pública
    public static final double DEF_ID = 1;

    // Variable privada
    private String nombre;

    // Variable protegida
    protected long dni;

    // Variable pública
    private double altura;
}
```

Constructor de la clase

Un constructor es una función que se ejecuta automáticamente cuando se crea un objeto de una clase específica y se usa para inicializar valores del objeto recién creado. En una clase podemos crear varios constructores que reciben distintos argumentos o parámetros. Si no escribimos ninguno, el objeto tendrá un constructor por defecto, sin argumentos y que no hace nada.

```
public class Rectangulo
{
    private static final double LADO_POR_DEFECTO = 0;

    private double lado1;
    private double lado2;

    public Rectangulo()
    {
        lado1 = LADO_POR_DEFECTO;
        lado2 = LADO_POR_DEFECTO;
    }
    public Rectangulo(double l1, double l2)
    {
        lado1 = l1;
        lado2 = l2;
    }
}
```

Una instancia de dicha clase puede ser creada en otro archivo donde se haya importado el paquete que contiene la clase usando cualquiera de los constructores. La creación se realiza con el operado *new* antes de la llamada al constructor.

```
Rectangulo rect1 = new Rectangulo();
Rectangulo rect2 = new Rectangulo(2, 3);
```

Un objeto que no ha sido inicializado tiene el valor de *null*, que es la palabra reservada que representa que la variable no hace referencia a ningún objeto. Este valor es útil si queremos inicializar una variable, pero no tenemos un objeto concreto que asignarle en ese momento. Además, es útil para comprobar si el objeto es válido antes de usarlo, lo que hacemos comparando con *null*.

```
String nombre = null;

if(nombre == null)
{
    System.out.println("El nombre no ha sido asignado");
}
```

Objeto *this*

Dentro de la clase podemos referirnos al objeto que actualmente instancia como *this*. Esto nos sirve, por ejemplo, para diferenciar variables locales de un método de variables de la clase si su nombre es el mismo. Se suele usar en los constructores cuando los argumentos se llaman igual que las variables de la clase. Pero también se puede usar en cualquier otro método para acceder a cualquier miembro propio del objeto.

El objeto *this* también puede usarse para llamar a un constructor desde otro. Esto debe hacerse siempre en la primera línea, de esta forma reutilizamos el código de otro constructor.

```
public class Rectangulo
{
    private static final double LADO_POR_DEFECTO = 0;

    private double lado1, lado2;

    public Rectangulo()
    {
        this(LADO_POR_DEFECTO, LADO_POR_DEFECTO);
    }
    public Rectangulo(double lado1, double lado2)
    {
        this.lado1 = lado1;
        this.lado2 = lado2;
    }
}
```

Dentro de los métodos del objeto es indiferente llamar a un miembro por su nombre solo o usando el objeto *this*.

Encapsulamiento

El *encapsulamiento* es la capacidad de ocultar el estado interno de un objeto y restringir el acceso a sus miembros. Esto nos permite agrupar datos y funciones en una sola entidad (*clase*), que encapsulan los datos y los métodos que operan sobre esos datos. De esta manera, el encapsulamiento evita que otros objetos accedan directamente a los datos internos y los modifiquen de manera inapropiada, lo que podría causar errores en el programa.

Un ejemplo simple es el uso de los métodos para escribir (*set*) o leer (*get*) el valor de una variable miembro.

```
/**
 * Establecer el lado 1
 * @param lado1 Lado 1
 */
public void setLado1(double lado1)
{
    // Comprobar que el valor pasado sea positivo
    if(lado1 > 0)
    {
        this.lado1 = lado1;
    }
}
/**
 * Obtener el lado 1
 * @return Lado 1
 */
public double getLado1()
{
    return lado1;
}
```

Herencia

La *herencia* es una propiedad de los lenguajes orientados a objetos, como Java, que nos permite crear clases derivadas a partir de una clase que llamamos *padre* (o *superclase*). La clase *hija* (o *descendiente*) hereda los atributos y métodos de su clase *padre* que sean visibles para ella, es decir, los miembros públicos y protegidos.

Esto permite aprovechar y extender la funcionalidad de una clase, ahorrando tiempo y esfuerzo al reutilizar código ya probado y adaptarlo a nuevas necesidades partiendo de una base sólida. Desde la clase hija o descendiente se puede acceder a los miembros de la clase padre con la palabra reservada *super*.

Veamos un ejemplo de una nueva clase que modela un rectángulo y de ahí heredamos una clase que modele un prisma que descienda de ella.

```
public class MyRect
{
    protected double lado1;
    protected double lado2;

    public MyRect()
    {
        this(0, 0);
    }
    public MyRect(double lado1, double lado2)
    {
        this.lado1 = lado1;
        this.lado2 = lado2;
    }
    public void setLado1(double lado1)
    {
        if(lado1 > 0)
        {
            this.lado1 = lado1;
        }
    }
    public double getLado1()
    {
        return lado1;
    }
    public void setLado2(double lado2)
    {
        this.lado2 = lado2;
    }
    public double getLado2()
    {
        return lado2;
    }
    public double calcArea()
    {
        return lado1*lado2;
    }
}

public class MyPrism extends MyRect
{
    protected double alto;

    public MyPrism()
    {
        this(0, 0, 0);
    }
    public MyPrism(double ancho, double largo, double alto)
    {
        super(ancho, largo);
        this.alto = alto;
    }
    public double calVolumen()
    {
        return alto*calcArea();
    }
    public double getAncho()
    {
        return getLado1();
    }
    public void setAncho(double ancho)
    {
        super.setLado1(ancho);
    }
    public double getLargo()
    {
        return getLado2();
    }
    public void setLargo(double lado2)
    {
        setLado2(lado2);
    }
    public double getAlto()
    {
        return alto;
    }
    public void setAlto(double alto)
    {
        this.alto = alto;
    }
}
```

La clase hija *MyPrism* toma de la clase padre *MyRect* las variables *lado1* y *lado2* para usarlas como ancho y largo de la base y toma el cálculo del área para cálculo de la base del prisma en el cálculo del volumen del mismo.

```
MyPrism prisma = new MyPrism(2, 4, 6);
MyRect rect = new MyRect(3, 5);

System.out.println("Area rectángulo: " + rect.calcArea());
System.out.println("Volumen prisma: " + prisma.calVolumen());
```

Polimorfismo

El *polimorfismo* es una propiedad de la POO que permite que los objetos de diferentes clases respondan a un mismo mensaje de diferentes maneras. El polimorfismo permite crear una jerarquía de clases relacionadas que se comportan de manera diferente pero que comparten una interfaz común. Esto hace que el código sea más fácil de mantener y actualizar, ya que se puede agregar una nueva clase sin afectar el código existente. Hay dos tipos principales de polimorfismo, de sobrecarga y de sobrescritura.

Polimorfismo de sobrecarga

La sobrecarga de métodos es una técnica que permite que una clase tenga varios métodos con el mismo nombre, pero con diferentes parámetros. En tiempo de compilación, el compilador identifica el método adecuado a partir de los parámetros utilizados.

Un ejemplo de polimorfismo de sobrecarga ya lo vimos al ver que una clase puede tener varios constructores, pero veamos un ejemplo con otras funciones.

Clase	Uso
<pre>public class CalcPromedio { public CalcPromedio(){} public double prom(double n1, double n2) { return (n1 + n2)/2; } public double prom(double n1, double n2, double n3) { return (n1 + n2 + n3)/3; } public double prom(double[] nums) { double sum = 0; for(int i = 0; i < nums.length; ++i) { sum += nums[i]; } return sum/nums.length; } }</pre>	<pre>public class PoliApp { public static void main(String[] args) { CalcPromedio calcProm = new CalcPromedio(); double promedio; promedio = calcProm.prom(1, 2); System.out.println("promedio: " + promedio); promedio = calcProm.prom(1, 2, 3); System.out.println("promedio: " + promedio); double[] numeros = { 1, 3, 5 }; promedio = calcProm.prom(numeros); System.out.println("promedio: " + promedio); } }</pre>
	<p>Salida</p> <pre>promedio: 1.5 promedio: 2.0 promedio: 3.0</pre>

En este caso tenemos varios métodos llamados *prom*, pero que reciben distintos parámetros, así al llamarlo se elige el adecuado.

Polimorfismo de sobrescritura

La sobrescritura de métodos es una técnica que permite que una clase hija redefina un método de su clase padre o superclase. Esto significa que el método de la subclase reemplaza al método de la superclase y se ejecuta en su lugar cuando se lo llama.

Retomando el ejemplo de la clase *MyRect* y *MyPrism*, sobrescribimos el método *calcArea* en *MyPrism* para que calcule el área lateral del prisma.

Llamada

Métodos de *MyPrism*

```
MyPrism prisma = new MyPrism(2, 4, 6);
MyRect rect = new MyRect(3, 5);

System.out.println("Area rectángulo: " + rect.calcArea());
System.out.println("Area prisma: " + prisma.calcArea());
System.out.println("Volumen prisma: " + prisma.calVolumen());
```

Salida

```
Area rectángulo: 15.0
Area prisma: 88.0
Volumen prisma: 48.0
```

```
@Override
public double calcArea()
{
    return 2*lado1*lado2 + 2*lado1*alto + 2*lado2*alto;
}
public double calVolumen()
{
    return alto*super.calcArea();
}
```

Como *MyPrism* hereda de *MyRect*, podemos declarar el objeto como del tipo padre pero crearlo como un hijo.

```
MyRect prisma = new MyPrism(2, 4, 6);
MyRect rect = new MyRect(3, 5);

System.out.println("Area rectángulo: " + rect.calcArea());
System.out.println("Area prisma: " + prisma.calcArea());
```

Aunque los dos objetos están declarados del mismo tipo, se instancian diferente, por lo tanto, la llamada a la función *calcArea* se hace a la que corresponde en cada caso.

Clases abstractas

Una clase abstracta en Java es una clase que no puede ser instanciada directamente, es decir, no se pueden crear objetos de esa clase. Las clases abstractas sirven como plantillas para otras clases, lo que permite establecer una estructura común que las clases derivadas (hijas) deben seguir.

Por ejemplo, podemos tener una clases abstracta que modele los polígonos, y por lo tanto, todas las clases hijas deberían tener métodos que, por ejemplo, retornen la cantidad de lados y el perímetro. Por esto, la clase abstracta madre debería tener estos métodos a ser obligatoriamente implementados en los hijos. Veamos un ejemplo.

```

                                Clase abstracta padre
public abstract class MyPolygon
{
    public abstract String getNombre();
    public abstract int getLados();
    public abstract double calcPerimetro();
}

Clase hija                                Clase hija
```

```

public class MyTriangle extends MyPolygon
{
    private double l1, l2, l3;

    public MyTriangle(double l1, double l2, double l3)
    {
        this.l1 = l1;
        this.l2 = l2;
        this.l3 = l3;
    }
    @Override
    public String getNombre()
    {
        return "Triángulo";
    };
    @Override
    public int getLados()
    {
        return 3;
    };
    @Override
    public double calcPerimetro()
    {
        return l1 + l2 + l3;
    }
}

public class MySquare extends MyPolygon
{
    private double l1;

    public MySquare(double l1)
    {
        this.l1 = l1;
    }
    @Override
    public String getNombre()
    {
        return "Cuadrado";
    };
    @Override
    public int getLados()
    {
        return 4;
    };
    @Override
    public double calcPerimetro()
    {
        return 4*l1;
    };
}

```

Creamos un array con instancias de las clases para mostrar sus datos.

```

public static void main(String[] args)
{
    MyPolygon[] poligons = new MyPolygon[2];

    poligons[0] = new MySquare(2);
    poligons[1] = new MyTriangle(2, 3, 4);

    for(int i = 0; i < 2; ++i)
    {
        System.out.println(poligons[i].getNombre() +
            ": Lados " + poligons[i].getLados() +
            ". Perímetro: " + poligons[i].calcPerimetro());
    }
}

```

Salida

```

Cuadrado: Lados 4. Perímetro: 8.0
Triángulo: Lados 3. Perímetro: 9.0

```

Interfaces

Una interfaz en Java es una colección de métodos abstractos y propiedades constantes. Las interfaces especifican qué se debe hacer pero no su implementación. Serán las clases las que *implementen* estas interfaces las que describan la lógica del comportamiento de los métodos. La principal diferencia entre *interface* y *abstract* es que un interface proporciona un mecanismo de encapsulamiento de los métodos sin forzar al usuario a utilizar la herencia. Apliquemos esto al ejemplo de los polígonos.

```

public interface IPolygon
{
    public String getNombre();
    public int getLados();
    public double calcPerimetro();
}

```

Implementación de la interface

```
public class MyTriangle2 implements IPolygon
{
    private double l1, l2, l3;

    public MyTriangle2(double l1, double l2, double l3)
    {
        this.l1 = l1;
        this.l2 = l2;
        this.l3 = l3;
    }
    @Override
    public String getNombre()
    {
        return "Triángulo";
    }
    @Override
    public int getLados()
    {
        return 3;
    }
    @Override
    public double calcPerimetro()
    {
        return l1 + l2 + l3;
    }
}
```

```
public class MySquare2 implements IPolygon
{
    private double l1;

    public MySquare2(double l1)
    {
        this.l1 = l1;
    }
    @Override
    public String getNombre()
    {
        return "Cuadrado";
    }
    @Override
    public int getLados()
    {
        return 4;
    }
    @Override
    public double calcPerimetro()
    {
        return 4*l1;
    }
}
```

Entrada y salida y datos

Entrada y salida por consola

Antes de profundizar en los conceptos de entrada y salida de datos, veamos como leer y escribir datos desde y hacia la consola, con lo que ya podremos realizar programas sencillos. Para escribir usamos la clase *System.out*, y para leer datos desde la consola usamos la clase *Scanner*.

Entrada de datos (clase *Scanner*)

Para usar la clase *Scanner* creamos una instancia de la clase pasando como argumentos el objeto *System.in* y usamos cualquiera de sus métodos disponibles. Al final del uso llamamos al método *close()* para cerrar el flujo de datos de entrada.

Método	Descripción
<code>nextBoolean()</code>	Lee un a valor tipo <code>boolean</code>
<code>nextByte()</code>	Lee un a valor tipo <code>byte</code>
<code>nextDouble()</code>	Lee un a valor tipo <code>double</code>
<code>nextFloat()</code>	Lee un a valor tipo <code>float</code>
<code>nextInt()</code>	Lee un a valor tipo <code>int</code>
<code>nextLine()</code>	Lee un a valor tipo <code>String</code>
<code>nextLong()</code>	Lee un a valor tipo <code>long</code>
<code>nextShort()</code>	Lee un a valor tipo <code>short</code>

Salida de datos (*System.out*)

Para usar la clase *System.out* creamos una instancia y usamos cualquiera de sus métodos:

Método	Descripción
<code>print()</code>	Imprime texto o valores a la consola
<code>printf()</code>	Imprime texto formateado a la consola
<code>println()</code>	Imprime texto o valores a la consola, seguida por un salto de línea

Ya hemos visto *print* y *println* en ejemplos anteriores. La función *printf* permite imprimir texto formateado usando etiquetas (que empiezan con %) que se reemplazan por valores de las variables pasadas como parámetros. Algunas de las etiquetas más usadas son:

Etiqueta (sin el %)	Imprime
<code>%</code>	Caracter <code>%</code> .
<code>n</code>	Salto de línea.
<code>b</code> ó <code>B</code>	Valor booleano en minúscula o mayúsculas respectivamente.
<code>c</code> ó <code>C</code>	Caracter Unicode en minúscula o mayúsculas respectivamente.
<code>s</code> ó <code>S</code>	Cadena de caracteres en minúscula o mayúsculas respectivamente.
<code>d</code>	Número entero.

f	Número flotante.
---	------------------

```
public class PrintfApp
{
    public static void main(String[] args)
    {
        String nombre = "Juan";
        short edad = 25;
        double altura = 1.78 ;
        boolean empleado = true;

        System.out.printf("Nombre: %s.%nEdad: %d.%nAltura: %f.%nEs empleado: %b.%n",
            nombre, edad, altura, empleado);
    }
}
```

```
Nombre: Juan.
Edad: 25.
Altura: 1.780000.
Es empleado: true.
```

Streams

En Java, la entrada y salida de datos (I/O) se hace a través de *streams* (flujos de datos). Un *stream* es una secuencia de datos que fluye desde una fuente (entrada) o hacia un destino (salida). Los streams se dividen en dos grandes categorías: *streams de bytes* para datos binarios y *streams de caracteres* para texto.

Característica	Streams de Bytes	Streams de Caracteres
Tipo de datos	Bytes (8 bits por operación)	Caracteres (16 bits por operación)
Clases principales	<i>InputStream</i> , <i>OutputStream</i>	<i>Reader</i> , <i>Writer</i>
Uso principal	Datos binarios (imágenes, videos, etc.)	Texto (archivos de texto, consola)
Ejemplos de clases	<i>FileInputStream</i> , <i>FileOutputStream</i>	<i>FileReader</i> , <i>FileWriter</i>

Los *streams de bytes* permiten leer y escribir datos byte a byte sin importar el tipo de dato o el formato del archivo. Las clases principales son:

- ***InputStream***: Superclase abstracta para todos los streams de entrada de bytes. Las clases más usadas son: *ByteArrayInputStream* y *FileInputStream*.
- ***OutputStream***: Superclase abstracta para todos los streams de salida de bytes. Las clases más usadas son: *ByteArrayOutputStream* y *FileOutputStream*.

Los *streams de caracteres* están diseñados para trabajar con datos de texto y se utilizan para procesar caracteres Unicode. Las clases principales son:

- ***Reader***: Superclase abstracta para todos los streams de entrada de caracteres. La clase más usada es *FileReader*.
- ***Writer***: Superclase abstracta para todos los streams de salida de caracteres. La clase más usada es *FileWriter*.

Siempre que trabajemos con streams realizaremos estos pasos: abrir el flujo (crear el objeto), escribir los datos o leerlos y procesarlos, y por último cerrar el stream.

Leer y escribir bytes en un archivo

Para leer y escribir bytes en un archivo usamos las clases *FileInputStream* y *FileOutputStream* respectivamente. Veamos un ejemplo de escritura usando *FileOutputStream*:

```
String filePath = "archivo.bin";    // Nombre del archivo
byte[] data = { 1, 3, 5, 7, 9 };    // Datos a escribir
FileOutputStream fos;                // Stream de escritura

// Escribir bytes al archivo
try
{
    // Creo el objeto (se abre el archivo/stream de datos)
    fos = new FileOutputStream(filePath);
    // Escribo los datos
    fos.write(data);
    // Cierro el archivo/stream de datos
    fos.close();
    System.out.println("Bytes escritos en el archivo.");
}
catch(IOException e)
{
    e.printStackTrace();
}
```

FileOutputStream se puede construir pasando un parámetro adicional, booleano, que si es *true* agrega los datos al final del archivo si no, por defecto crea o sobrescribe el archivo si ya existe. Veamos un ejemplo de lectura usando *FileInputStream*:

```
FileInputStream fis;                // Stream de lectura
try
{
    // Creo el objeto (se abre el archivo/stream de datos)
    fis = new FileInputStream(filePath);
    // Byte leído
    int readed;
    System.out.println("Leyendo bytes del archivo:");
    while((readed = fis.read()) != -1)
    {
        // Si la función read() retorna -1 se llegó al final del archivo
        // Muestro el dato leído
        System.out.printf("%H ", readed);
    }
    // Cierro el archivo/stream de datos
    fis.close();
    System.out.println();
}
catch(IOException e)
{
    e.printStackTrace();
}
```

Salida
Leyendo bytes del archivo:
1 3 5 7 9

Para leer datos con *FileInputStream* lo hacemos byte a byte usando el método *read* que nos retorna el dato leído o nos retorna -1 si se llegó al final del archivo.

Leer y escribir texto en un archivo

Para leer y escribir bytes en un archivo usamos las clases *FileReader* y *FileWriter* respectivamente. Veamos un ejemplo de escritura usando *FileWriter*:


```
String filePath = "archivo.txt";
String data = "Este es un ejemplo de Streams de Caracteres en Java.";

// Escribir texto al archivo
FileWriter writer;
try
{
    // Abrir stream
    writer = new FileWriter(filePath);
    // Escribir datos
    writer.write(data);
    // Cerrar el stream
    writer.close();
    System.out.println("Datos escritos en el archivo.");
}
catch(IOException e)
{
    e.printStackTrace();
}
```

FileWriter, al igual que *FileOutputStream*, se puede construir pasando un parámetro booleano adicional para indicar si los datos se agregan al final o no. Veamos como leer texto, caracter por carácter, usando *FileReader*.

<pre>// Leer texto desde el archivo FileReader reader; try { reader = new FileReader(filePath); int content; System.out.println("Leyendo el archivo:"); while((content = reader.read()) != -1) { System.out.print((char) content); } reader.close(); } catch(IOException e) { e.printStackTrace(); }</pre>	<p>Salida</p> <p>Leyendo el archivo: Este es un ejemplo de Streams de Caracteres en Java.</p>
--	---

En todos estos ejemplos usamos la estructura *try-catch* que es para manejar *excepciones*. En la siguiente sección profundizaremos en el tema.

Excepciones

En Java los errores en tiempo de ejecución (cuando se está ejecutando el programa) se denominan *excepciones*. Estas ocurren cuando se produce un error en nuestro programa, como por ejemplo cuando se hace una división por cero, cuando un objeto es *null* y no puede serlo, cuando no se abre correctamente un archivo, etc. Cuando se produce una excepción, se muestra un mensaje de error en la consola y finaliza la ejecución del programa.

Existen muchos tipos de excepciones y hay que decir que se aprenden con la experiencia de encontrarse con ellas y solucionarlas. Cuando se produce una excepción se crea un objeto con la información sobre el error. Estos son de tipos descendientes de la clase *Throwable*. Algunas excepciones son:

Excepción	Motivo
ArithmeticException	Típicamente es el resultado de división por 0:
NullPointerException	Se produce cuando se intenta acceder a una variable o método antes de ser definido.
ClassCastException	El intento de convertir un objeto a otra clase que no es válida.
NegativeArraySizeException	Puede ocurrir si hay un error aritmético al cambiar el tamaño de un array.
OutOfMemoryException	La creación de un objeto con el operador <i>new</i> ha fallado por falta de memoria.
NoClassDefFoundException	Se referenció una clase que el sistema es incapaz de encontrar.
ArrayIndexOutOfBoundsException	Intento de acceder a un elemento de un array más allá de los límites definidos inicialmente.
InternalException	Este error se reserva para eventos que no deberían ocurrir. Por definición, esta excepción no debería lanzarse.

Java nos permite hacer un control de las excepciones para que nuestro programa no se pare inesperadamente y se siga su ejecución. Para ello tenemos la estructura *try-catch-finally*,

```
try
{
    // Esto se ejecuta cuando no hay una excepción
}
catch(Exception e)
{
    // Esto se ejecuta cuando se produce una excepción
}
finally
{
    // Esto se ejecuta tanto si hay o no excepciones
}
```

La clase *Throwable* y, por lo tanto todas sus clases descendientes, tiene varios métodos para obtener información de la excepción.

Streams estándar

Los *streams estándar* en Java permiten que los programas interactúen con la consola como si fuese un archivo. Estos streams están predefinidos y se accede a ellos a través de la clase *System*. Java proporciona tres streams estándar:

- **Standard Input (Entrada estándar):** Se utiliza para leer la entrada del usuario (por ejemplo, desde el teclado).
- **Standard Output (Salida estándar):** Se utiliza para escribir la salida (normalmente en la consola).
- **Standard Error (Error estándar):** Se utiliza para escribir mensajes de error o información adicional sobre excepciones.

Manejo de archivos y directorios con la clase *File*

La clase principal para gestionar directorios y archivos es la clase *File*, con la que se puede crear, listar, verificar y eliminar directorios, entre otras operaciones.

Obtener información de archivos y directorios

La clase *File* tiene varios métodos para comprobar el estado del archivo o directorio y de la ruta que representa.

Método	Comprobación que se hace sobre el archivo/directorio
<code>isDirectory</code>	Si es de un directorio
<code>isFile</code>	Si es un archivo
<code>isHidden</code>	Si está oculto
<code>exists</code>	Si existe
<code>canRead</code>	Si puede ser leído
<code>canWrite</code>	Si puede ser escrito

Creación de Directorios

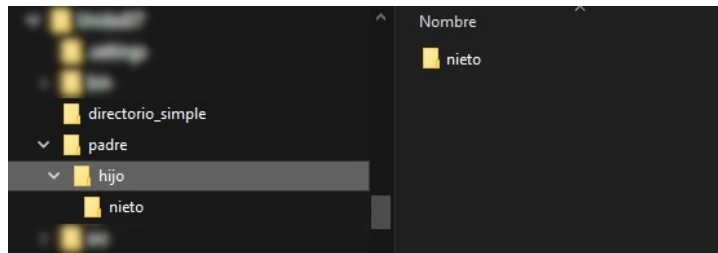
Hay dos métodos principales para la creación de directorios: *mkdir* para crear un único directorio y *makedirs* para crear además los directorios padres que no existan.

```
// Ruta del directorio a crear
String dirPath = "directorio_simple";

// Crear un directorio utilizando mkdir()
File dir = new File(dirPath);
if(dir.mkdir())
{
    System.out.println("Directorio '" + dirPath + "' creado exitosamente.");
}
else
{
    System.out.println("No se pudo crear el directorio '" + dirPath + "'.");
}

// Crear una estructura de directorios utilizando mkdirs()
String nestedDirPath = "padre/hijo/nieto";
File nestedDir = new File(nestedDirPath);
if(nestedDir.mkdirs())
{
    System.out.println("Estructura de directorios '" + nestedDirPath + "' creada exitosamente.");
}
else
{
    System.out.println("No se pudo crear la estructura de directorios '" + nestedDirPath + "'.");
}
```

Resultado:



Listado de contenido de directorios

Los métodos *list* y *listFiles* permiten obtener los nombres de los archivos o los objetos *File* de los archivos y directorios dentro de un directorio.

- **list:** Devuelve un array de nombres de los archivos y subdirectorios.
- **listFiles:** Devuelve un array de objetos *File* de los archivos y subdirectorios.

```
String dirPath = "padre"; // Ruta del directorio

// Crear un objeto File para el directorio
File dir = new File(dirPath);
// Verificar si es un directorio y listar su contenido
if(dir.isDirectory())
{
    System.out.println("Contenido del directorio '" + dirPath + "':");
    // Utilizar list() para obtener los nombres de los archivos
    String[] fileNames = dir.list();
    if(fileNames != null && fileNames.length > 0)
    {
        for(String fileName : fileNames)
        {
            System.out.println(fileName);
        }
    }
    else
    {
        System.out.println("El directorio está vacío.");
    }
    // Utilizar listFiles() para obtener objetos File
    File[] files = dir.listFiles();
    if(files != null && files.length > 0)
    {
        for(File file : files)
        {
            System.out.println(file.getName() + " (es archivo: " + file.isFile() + ")");
        }
    }
}
else
{
    System.out.println(dirPath + " no es un directorio válido.");
}
```

Eliminar Directorios

Para eliminar archivos y directorios usamos el método *delete*. Este método elimina archivos y directorios vacíos. Para eliminar un directorio que contiene archivos, primero hay que eliminar su contenido.

```
String dirPath = "directorio_simple"; // Ruta del directorio a eliminar
File dir = new File(dirPath);

// Eliminar el directorio si está vacío
if(dir.delete())
{
    System.out.println("Directorio '" + dirPath + "' eliminado exitosamente.");
}
else
{
    System.out.println("No se pudo eliminar el directorio '" + dirPath + "'. Asegúrate de que esté vacío.");
}
```

Multitarea e Hilos

La multitarea permite que un sistema ejecute múltiples procesos o subprocesos (threads o hilos) en paralelo.

Tipos de multitarea

Multitarea por procesos

Ocurre cuando el sistema operativo permite que múltiples procesos se ejecuten simultáneamente. Cada proceso es una aplicación independiente con su propio espacio de memoria. Los procesos están completamente aislados entre sí y cada uno tiene su propio espacio de memoria. Los procesos pueden comunicarse entre sí, pero esto suele requerir mecanismos más complejos (como la comunicación interprocesos, IPC). Es más costoso en términos de recursos del sistema porque cada proceso consume más memoria y recursos del procesador.

Multitarea por subprocesos (Multithreading)

Este tipo de multitarea se enfoca en la concurrencia dentro de un mismo proceso. En lugar de ejecutar múltiples procesos independientes, la multitarea por hilos permite que varias tareas (hilos) se ejecuten en paralelo dentro de una sola aplicación. Los hilos comparten el mismo espacio de memoria dentro del proceso. Los hilos comparten el mismo espacio de memoria y recursos dentro del proceso por lo que es más eficiente en términos de uso de memoria que la multitarea por procesos. Por lo tanto diferentes partes de un programa se ejecutan en paralelo dentro del mismo.

Hilos (Threads)

Un hilo (thread) es una secuencia de instrucciones que puede ejecutarse en paralelo con otros hilos dentro de un proceso.

Características

- **Ejecutan tareas concurrentes:** Los hilos pueden ejecutar diferentes tareas dentro de un programa de forma concurrente, lo que mejora el rendimiento de las aplicaciones que deben realizar varias operaciones al mismo tiempo.
- **Comparten recursos:** Los hilos dentro de un mismo proceso comparten el mismo espacio de memoria y recursos del sistema, lo que permite una comunicación más rápida entre ellos, pero también aumenta el riesgo de errores de concurrencia.
- **Menor costo que los procesos:** Crear y gestionar hilos es menos costoso en términos de recursos del sistema que crear procesos. Esto se debe a que no necesitan su propio espacio de memoria aislado.
- **Sincronización:** Los hilos deben ser sincronizados cuando acceden a recursos compartidos para evitar problemas como condiciones de carrera (race conditions), donde varios hilos intentan modificar el mismo recurso al mismo tiempo.

Tipos de Hilos

- **Hilos del usuario (User Threads):** Son los hilos que son creados y gestionados directamente por la aplicación.
- **Hilos del sistema (Daemon Threads):** Son hilos que se ejecutan en segundo plano para realizar tareas de soporte. Por ejemplo, un hilo que realiza recolección de basura es un hilo daemon.

Implementación de hilos

Hay dos formas de implementar hilos en Java. Una es extender la clase *Thread* e implementar el método *run* y la otra es implementar la interface *Runnable*.

Veamos un ejemplo del uso de una clases hija de *Thread*.

```
public class ThreadApp1
{
    public static void main(String[] args)
    {
        MyThread thread = new MyThread();

        thread.start();
    }
}
```

Salida

```
Inicio cuenta regresiva (Thread)
10 9 8 7 6 5 4 3 2 1
Fin de la cuenta regresiva
```

```
public class MyThread extends Thread
{
    public void run()
    {
        int i = 10;
        System.out.println("Inicio cuenta regresiva (Thread)");
        while(i > 0)
        {
            System.out.print ((i--) + " ");
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        System.out.println();
        System.out.println("Fin de la cuenta regresiva");
    }
}
```

Implementando *Runnable*:

```
public class ThreadApp2
{
    public static void main(String[] args)
    {
        MyRunnable runnable = new MyRunnable();
        runnable.run();
    }
}
```

Salida

```
Inicio cuenta regresiva (Runnable)
10 9 8 7 6 5 4 3 2 1
Fin de la cuenta regresiva
```

```
public class MyRunnable implements Runnable
{
    public void run()
    {
        int i = 10;
        System.out.println("Inicio cuenta regresiva (Runnable)");
        while(i > 0)
        {
            System.out.print ((i--) + " ");
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        System.out.println();
        System.out.println("Fin de la cuenta regresiva");
    }
}
```

Si el código que queremos correr es corto como en estos casos, y no queremos hacer una clase aparte, podemos crear una instancia de *Thread* pasando una instancia de *Runnable* donde implementamos el método *run* con el código correspondiente como en el siguiente ejemplo.

```

public static void main(String[] args)
{
    Thread thread = new Thread(new Runnable()
    {
        @Override
        public void run()
        {
            int i = 10;
            System.out.println("Inicio cuenta regresiva (Thread con Runnable como parámetro)");
            while(i > 0)
            {
                System.out.print ((i--) + " ");
                try
                {
                    Thread.sleep(1000);
                }
                catch (InterruptedException e)
                {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            System.out.println();
            System.out.println("Fin de la cuenta regresiva");
        }
    });
    thread.start();
}

```

Salida

```

Inicio cuenta regresiva (Thread con Runnable como parámetro)
10 9 8 7 6 5 4 3 2 1
Fin de la cuenta regresiva

```

Esperar fin de hilos

Hay ocasiones en las que es necesario asegurarse que uno o más hilos terminen antes de que el hilo principal o algún otro hilo continúe su trabajo. El método *join* se utiliza para eso: hace que el hilo que lo llama espere a que el hilo sobre el que se invoca termine su ejecución. Veamos un ejemplo donde se llama a dos hilos y el principal, espera que terminen.

```

MyThreadCounter thread1 = new MyThreadCounter("Hilo 1");
MyThreadCounter thread2 = new MyThreadCounter("Hilo 2");

thread1.start();
thread2.start();
try
{
    thread1.join();
    thread2.join();
}
catch (InterruptedException e)
{
    e.printStackTrace();
}
System.out.println("Fin de ambos hilos");

```

Salida

```

Inicio Hilo 1
Inicio Hilo 2
[Hilo 2] 5
[Hilo 1] 5
[Hilo 1] 4
[Hilo 2] 4
[Hilo 2] 3
[Hilo 1] 3
[Hilo 1] 2
[Hilo 2] 2
[Hilo 2] 1
[Hilo 1] 1
Fin Hilo 1
Fin Hilo 2
Fin de ambos hilos

```

```

public class MyThreadCounter extends Thread
{
    private String name;

    public MyThreadCounter(String name)
    {
        this.name = name;
    }

    public void run()
    {
        int i = 5;
        System.out.println("Inicio " + name);
        while(i > 0)
        {
            System.out.println "[" + name + "] " + (i--);
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
        System.out.println("Fin " + name);
    }
}

```


Otra forma de hacer esto es con la clase *ExecutorService* que brinda varios métodos útiles para manejar múltiples hilos o un *pool* de hilos. Veamos un sencillo ejemplo que da el mismo resultado que el ejemplo anterior:

<pre> public static void main(String[] args) { ExecutorService executor = Executors.newFixedThreadPool(3); MyThreadCounter thread1 = new MyThreadCounter("Hilo 1"); MyThreadCounter thread2 = new MyThreadCounter("Hilo 2"); executor.execute(thread1); executor.execute(thread2); // Cerrar el pool después de ejecutar todas las tareas executor.shutdown(); } </pre>	<p>Salida</p> <pre> Inicio Hilo 1 Inicio Hilo 2 [Hilo 2] 5 [Hilo 1] 5 [Hilo 2] 4 [Hilo 1] 4 [Hilo 1] 3 [Hilo 2] 3 [Hilo 1] 2 [Hilo 2] 2 [Hilo 2] 1 [Hilo 1] 1 Fin Hilo 2 Fin Hilo 1 </pre>
---	--

Sincronizar recursos

Uno de los problemas más comunes en la programación con hilos ocurre cuando dos o más hilos intentan acceder o modificar un recurso compartido de manera simultánea, esto se llama *Condiciones de Carrera (Race Conditions)*.

Otro problema es el llamado *deadlock*. Este ocurre cuando dos o más hilos se bloquean mutuamente esperando recursos que los otros hilos tienen bloqueados, por lo que los hilos no pueden continuar porque están esperando que se libere algún recurso.

Una de las formas más simples de salvar estos errores es usando la palabra clave *synchronized* que permite bloquear métodos o bloques de código para que solo un hilo acceda a un recurso a la vez.

<pre> public static void main(String[] args) { MyCounter counter = new MyCounter(); MyThreadIncCounter thread1 = new MyThreadIncCounter(counter); MyThreadIncCounter thread2 = new MyThreadIncCounter(counter); thread1.start(); thread2.start(); try { thread1.join(); thread2.join(); } catch (InterruptedException e) { e.printStackTrace(); } System.out.println("Valor final del contador: " + counter.getCount()); } </pre>	<pre> public class MyCounter { private int count = 0; // Método sincronizado para incrementar el contador public synchronized void incCount() { count++; } public int getCount() { return count; } } public class MyThreadIncCounter extends Thread { private MyCounter counter; public MyThreadIncCounter(MyCounter counter) { this.counter = counter; } public void run() { for(int i = 0; i < 1000; i++) { counter.incCount(); } } } </pre>
---	--

Salida

Valor final del contador: 2000

En este caso, el método *incCount* que incrementa la variable privada *count*, está sincronizado para que no se intente acceder a dicha variable por dos hilos al mismo tiempo, esto se llama *sincronización por exclusión mutua*.

Comunicación entre Hilos

Cuando un hilo debe esperar a que otro hilo termine una tarea para continuar necesitamos coordinar su trabajo. Para esto usamos los métodos *wait*, *notify* y *notifyAll* dentro de un bloque sincronizado. Veamos un ejemplo donde un hilo genera un dato y luego notifica que se produjo mientras que otro hilo espera la notificación para hacer algo con ese dato, en este caso, imprimirlo.

```
public class MyLoadDataThread extends Thread {
    private MyBuffer buffer;

    public MyLoadDataThread(MyBuffer buffer) {
        this.buffer = buffer;
    }
    public void run() {
        for(int i = 1; i <= 5; i++) {
            buffer.loadData();
        }
    }
}

public class MyLogDataThread extends Thread {
    private MyBuffer buffer;

    public MyLogDataThread(MyBuffer buffer) {
        this.buffer = buffer;
    }
    public void run() {
        for(int i = 1; i <= 5; i++) {
            buffer.logData();
        }
    }
}

public class MyBuffer {
    private int data;
    private boolean loaded = false;

    public synchronized void loadData() {
        while(loaded) {
            try {
                wait(); // Espera a que el consumidor consuma el dato
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // Cargo un dato aleatorio en data
        data = (int)(Math.random()*10);
        System.out.println("Datos cargado: " + data);
        loaded = true;
        notify(); // Notifica al consumidor que el dato está disponible
    }

    public synchronized void logData() {
        while(!loaded) {
            try {
                wait(); // Espera a que el productor produzca un dato
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Dato: " + data);
        loaded = false;
        notify(); // Notifica al productor que puede producir otro dato
    }
}

public class ThreadApp7 {
    {
        public static void main(String[] args) {
            MyBuffer buffer = new MyBuffer();
            MyLoadDataThread productor = new MyLoadDataThread(buffer);
            MyLogDataThread consumidor = new MyLogDataThread(buffer);

            productor.start();
            consumidor.start();
        }
    }
}
```

Datos cargado: 7
Dato: 7
Datos cargado: 8
Dato: 8
Datos cargado: 0
Dato: 0
Datos cargado: 3
Dato: 3
Datos cargado: 5
Dato: 5

Apéndice A - Estructuras de datos

Listas y colecciones

Hay varias clases que implementan la interface *List*, que es hija de la interface *Collection*. Todas estas clases tendrás métodos en común como los siguientes:

Método	Descripción
<code>add(index, element)</code>	Agregar objeto en la posición indicada
<code>addAll(index, collection)</code>	Agrega todos los elementos de la colección a la lista
<code>size()</code>	Obtener la cantidad de elementos del array
<code>clear()</code>	Eliminar todos los elementos del array
<code>remove(index)</code>	Eliminar el objeto en la posición indicada
<code>remove(element)</code>	Eliminar el objeto indicado
<code>get(index)</code>	Obtener objeto en la posición indicada
<code>set(index, element)</code>	Poner el objeto en la posición indicada
<code>indexOf(objet)</code>	Retorna el índice del objeto indicado
<code>lastIndexOf(element)</code>	Retorna el índice de la última aparición del objeto dado
<code>equals(element)</code>	Compara el elemento dado con los de la lista
<code>isEmpty()</code>	Retorna <i>true</i> si el array está vacío
<code>contains(element)</code>	Comprueba si contiene el objeto indicado
<code>containsAll(collection)</code>	Comprueba si contiene la lista de objetos indicados

Arrays

Entre estas clases tenemos *ArrayList* y *Vector* que manejan arrays, *LinkedList* para listas enlazadas y *Stack* para pilas. Veamos un ejemplo usando *ArrayList*.

```
ArrayList<String> list = new ArrayList<>();
ArrayList<String> listOdd = new ArrayList<>();

listOdd.add("Uno");
listOdd.add("Tres");
System.out.println("subList: " + listOdd);

list.add("Cero");
list.add("Dos");
list.addAll(listOdd);
System.out.println("list: " + list);

System.out.println("list.get(1): " + list.get(1));

list.set(1, "II");
System.out.println("list: " + list);

System.out.println("list.contains(\"Uno\") : " + list.contains("Uno"));
System.out.println("list.contains(\"Cinco\") : " + list.contains("Cinco"));
System.out.println("list.containsAll(listOdd): " + list.containsAll(listOdd));

System.out.println("list.indexOf(\"Dos\") : " + list.indexOf("Dos"));
System.out.println("list.indexOf(\"Cinco\") : " + list.indexOf("Cinco"));

System.out.println("list.isEmpty(): " + list.isEmpty());
```

Salida

```
subList: [Uno, Tres]
list: [Cero, Dos, Uno, Tres]
list.get(1): Dos
list: [Cero, II, Uno, Tres]
list.contains("Uno"): true
list.contains("Cinco"): false
list.containsAll(listOdd): true
list.indexOf("Dos"): -1
list.indexOf("Cinco"): -1
list.isEmpty():false
```

Listas enlazadas

Los métodos más importantes de *LinkedList* para manejar listas enlazadas son:

Método	Descripción
<code>peekFirst()</code>	Obtener el primer elemento de la lista
<code>peekLast()</code>	Obtener el último elemento de la lista
<code>offerFirst(element)</code>	Agregar elemento al principio de la lista
<code>offerLast(element)</code>	Agregar elemento al final de la lista
<code>pollFirst()</code>	Quitar el primer elemento de la lista
<code>pollLast()</code>	Quitar el último elemento de la lista

```
LinkedList<String> list = new LinkedList<>();
```

```
list.add("Cero");
list.add("Uno");
list.add("Dos");
list.add("Tres");
System.out.println("list: " + list);
System.out.println("First: " + list.peekFirst() + ". Last: " + list.peekLast());

System.out.println("Agregar al principio: 'Primero'");
list.offerFirst("Primero");
System.out.println("list: " + list);

System.out.println("Agregar al final: 'Último'");
list.offerLast("Último");
System.out.println("list: " + list);

System.out.println("Quitar el primero:");
list.pollFirst();
System.out.println("list: " + list);

System.out.println("Quitar el último:");
list.pollLast();
System.out.println("list: " + list);
```

Salida

```
list: [Cero, Uno, Dos, Tres]
First: Cero. Last: Tres
Agregar al principio: 'Primero'
list: [Primero, Cero, Uno, Dos, Tres]
Agregar al final: 'Último'
list: [Primero, Cero, Uno, Dos, Tres, Último]
Quitar el primero:
list: [Cero, Uno, Dos, Tres, Último]
Quitar el último:
list: [Cero, Uno, Dos, Tres]
```

Pilas

Los métodos más importantes de *Stack* para manejar pilas son:

Método	Descripción
<code>push(element)</code>	Agregar elemento a la parte superior de la pila
<code>search(element)</code>	Buscar un elemento en la pila (el primero tiene índice 1)
<code>pop()</code>	Quitar el elemento de la parte superior de la pila

Ejemplo de uso

```
Stack<String> stack = new Stack<>();
```

```
stack.push("Primero");
stack.push("Segundo");
stack.push("Tercero");
System.out.println("stack: " + stack);
```

```
System.out.println("Buscar 'Primero': " + stack.search("Primero"));
```

```
System.out.println("Quitar elementos de la pila:");
```

```
while(!stack.isEmpty())
{
    System.out.println("Quitar elemento superior de la pila: " + stack.pop());
    System.out.println("stack: " + stack);
}
```

Salida

```
stack: [Primero, Segundo, Tercero]
Buscar 'Primero': 3
Quitar elementos de la pila:
Quitar elemento superior de la pila: Tercero
stack: [Primero, Segundo]
Quitar elemento superior de la pila: Segundo
stack: [Primero]
Quitar elemento superior de la pila: Primero
stack: []
```

Colas

Las clases que manejan cola implementan la interface *Queue*, cuyos métodos principales son:

Método	Descripción
offer(element)	Agregar elemento al final de la cola
peek(element)	Consultar el elemento del principio de la cola
poll()	Quitar el elemento del principio de la cola

Una de las clases que implementa esta interface es *PriorityQueue*. Ejemplo de uso

```
Queue<String> queue = new PriorityQueue<>();
```

```
queue.offer("Primero");
queue.offer("Segundo");
queue.offer("Tercero");
System.out.println("Cola: " + queue);
```

```
System.out.println("Primero de la cola: " + queue.peek());
```

```
System.out.println("Quitar elementos de la cola:");
```

```
while(!queue.isEmpty())
{
    System.out.println("Quitar de la cola: " + queue.poll());
    System.out.println("queue: " + queue);
}
```

Salida

```
Cola: [Primero, Segundo, Tercero]
Primero de la cola: Primero
Quitar elementos de la cola:
Quitar de la cola: Primero
queue: [Segundo, Tercero]
Quitar de la cola: Segundo
queue: [Tercero]
Quitar de la cola: Tercero
queue: []
```

Matrices

En Java, no hay una clase particular para manejar matrices, simplemente se hace con arrays de dos dimensiones. Veamos un ejemplo de creación de una matriz y del uso de sus datos.

```
final int FILAS = 3;
final int COLUMNAS = 4;

int[][] matrix = new int[FILAS][COLUMNAS];
int value = 0;

// Llenar filas con número consecutivos por fila
for(int f = 0; f < FILAS; ++f)
{
    for(int c = 0; c < COLUMNAS; ++c)
    {
        matrix[f][c] = value;
        ++value;
    }
}
// Mostrar matriz
for(int f = 0; f < FILAS; ++f)
{
    for(int c = 0; c < COLUMNAS; ++c)
    {
        System.out.print(matrix[f][c] + " ");
    }
    System.out.println();
}
```

Salida

```
0 1 2 3
4 5 6 7
8 9 10 11
```

Conjuntos

Para manejar los conjuntos en Java, tenemos las clases que implementar la interface *Set*, como lo es, por ejemplo, la clase *HashSet*. Veámoslo con un ejemplo:

```
Set<String> set = new HashSet<>();

set.add("Perro");
set.add("Gato");
set.add("Pez");
System.out.println("set: " + set);

set.remove("Perro");
System.out.println("set: " + set);

System.out.println("Datos del conjunto:");
Iterator<String> iterator = set.iterator();
while(iterator.hasNext())
{
    System.out.println(iterator.next());
}
```

Salida

```
set: [Gato, Pez, Perro]
set: [Gato, Pez]
Datos del conjunto:
Gato
Pez
```

Diccionarios (mapas)

Para manejar los diccionarios o mapas en Java, tenemos las clases que implementar la interface *Map*, como lo es, por ejemplo, la clase *HashMap*.

```
HashMap<String, Integer> map = new HashMap<>();

map.put("Juan", 13579);
map.put("Ana", 456877);
map.put("Pedro", 24680);
map.put("María", 36901);
System.out.println("set: " + map);

System.out.println("Valor para la clave 'Juan': " + map.get("Juan"));
System.out.println("Valor para la clave 'Lisa': " + map.get("Lisa"));

System.out.println("Contiene la clase 'Ana': " + map.containsKey("Ana"));
System.out.println("Contiene la clase 'Julio': " + map.containsKey("Julio"));

// Acceder a todos los valores
System.out.println("Acceder a todos los valores:");
Set<String> claves = map.keySet();
Iterator<String> iterator = claves.iterator();
while(iterator.hasNext())
{
    String clave = iterator.next();

    System.out.println("{ " + clave + " = " + map.get(clave) + " }");
}
// Acceder a todos los valores (otro método)
System.out.println("Acceder a todos los valores (otro método):");
Set<Entry<String, Integer>> entradas = map.entrySet();
for(Entry<String, Integer> ent : entradas)
{
    System.out.println("{ " + ent.getKey() + " = " + ent.getValue() + " }");
}
```

Salida

```
set: {Ana=456877, María=36901, Pedro=24680, Juan=13579}
Valor para la clave 'Juan': 13579
Valor para la clave 'Lisa': null
Contiene la clase 'Ana': true
Contiene la clase 'Julio': false
Acceder a todos los valores:
{ Ana = 456877 }
{ María = 36901 }
{ Pedro = 24680 }
{ Juan = 13579 }
Acceder a todos los valores (otro método):
{ Ana = 456877 }
{ María = 36901 }
{ Pedro = 24680 }
{ Juan = 13579 }
```

Apéndice B - Excepciones

La base de las *excepciones* es la clase *Throwable*, también es base de los *errores*, que son fallos graves que están fuera del control del programador, como la falta de memoria disponible en el sistema. Las excepciones pueden clasificarse según cuando y como deben tratarse:

- **Excepciones verificadas (*Checked Exceptions*):** son aquellas que el compilador obliga al programador a manejar en tiempo de compilación. Estas excepciones surgen de situaciones que pueden preverse y, por lo tanto, deben ser manejadas antes de la ejecución.
- **Excepciones No Verificadas (*Unchecked Exceptions*):** estas ocurren en tiempo de ejecución y el compilador no obliga a manejarlas. En general, son causadas por errores de lógica del programador, como intentar dividir un número por cero o acceder a una posición de un array que no existe.

Veamos los principales métodos la clase *Throwable*:

Método	Descripción
<code>getMessage()</code>	Recupera un mensaje descriptivo de la excepción que se crea con el objeto <i>Throwable</i> .
<code>getCause()</code>	Devuelve la causa raíz de la excepción. Es útil cuando una excepción es causada por otra, permitiendo rastrear la cadena de excepciones hasta el problema original.
<code>toString()</code>	Devuelve una representación textual del objeto <i>Throwable</i> , que incluye el nombre de la clase de la excepción y el mensaje devuelto por <code>getMessage()</code> .
<code>printStackTrace()</code>	Imprime el resultado de <code>toString()</code> con la traza completa de la pila de error estándar (<code>System.err</code>).
<code>getStackTrace()</code>	Devuelve un array de objetos <i>StackTraceElement</i> , donde cada elemento representa una llamada en la pila de ejecución en el momento en que ocurrió la excepción.
<code>fillInStackTrace()</code>	Llena el objeto <i>Throwable</i> con la traza de la pila, sobrescribiendo cualquier traza previa. Se utiliza principalmente cuando se quiere capturar el estado de la pila en el momento en que se lanza una nueva excepción.

Veamos algunos ejemplos de uso de estos métodos.


```
try
{
    throw new Exception("Error de ejemplo");
}
catch(Exception e)
{
    System.out.println(e.getMessage());
}
```

Salida
Error de ejemplo

```
try
{
    Throwable cause = new IllegalArgumentException("Creada como ejemplo");
    throw new Exception("Excepción de ejemplo", cause);
}
catch(Exception e)
{
    System.out.println("Causa: " + e.getCause());
}
```

Salida
Causa: [java.lang.IllegalArgumentException](#): Creada como ejemplo

```
try
{
    throw new NullPointerException("Objeto nulo");
}
catch(Exception e)
{
    System.out.println(e.toString());
}
```

Salida
[java.lang.NullPointerException](#): Objeto nulo

```
try
{
    throw new Exception("Error grave");
}
catch(Exception e)
{
    e.printStackTrace();
}
```

Salida
[java.lang.Exception](#): Error grave
at com.cursojava.ExcepcionesApp.main([ExcepcionesApp.java:36](#))

```
try
{
    throw new Exception("Excepción lanzada");
}
catch(Exception e)
{
    e.fillInStackTrace(); // Captura el estado actual de la pila
    e.printStackTrace();
}
```

Salida
[java.lang.Exception](#): Excepción lanzada
at com.cursojava.ExcepcionesApp.main([ExcepcionesApp.java:61](#))

Además, podemos formatear la salida de los datos de la pila usando el método `getStackTrace` y escribir en la salida cada línea del stack con el formato que nos parezca mejor.

```
try
{
    throw new Exception("Error con pila");
}
catch(Exception e)
{
    StackTraceElement[] elementos = e.getStackTrace();
    for(StackTraceElement elemento : elementos)
    {
        System.out.println(elemento);
    }
}
```

Salida
com.cursojava.ExcepcionesApp.main([ExcepcionesApp.java:45](#))

Múltiples bloques catch y múltiples excepciones en un bloque catch

Java permite que se utilicen múltiples bloques catch para manejar diferentes tipos de excepciones. Cada bloque catch se evalúa en orden, y el primero que coincida con la excepción lanzada será ejecutado.

```
try
{
    int[] numeros = { 1, 2, 3 };
    System.out.println(numeros[5]); // Genera ArrayIndexOutOfBoundsException
}
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println("Error: Índice fuera de los límites del array.");
}
catch(ArithmeticException e)
{
    System.out.println("Error: División por cero.");
}
```

Salida
Error: Índice fuera de los límites del array.

También se pueden capturar múltiples excepciones en un solo bloque catch. Para lograrlo, simplemente se separan los tipos de excepciones con el operador | (pipe).

```
try
{
    File file = new File("archivo.txt");
    FileInputStream fis = new FileInputStream(file);
}
catch(NullPointerException | FileNotFoundException e)
{
    System.out.println("Ocurrió un error al manejar el archivo: " + e.getMessage());
}
```

Esto tiene varias ventajas:

- **Código más legible:** Al usar un único bloque catch, se reduce la necesidad de duplicar código, combinando estas excepciones en un solo bloque.
- **Mejor rendimiento:** El manejo de múltiples excepciones en un solo bloque reduce el uso de bloques catch y el compilador puede optimizar mejor el flujo de manejo de excepciones.
- **Reducción de código redundante:** Diferentes excepciones pueden requerir el mismo tratamiento lo que implicaba duplicar código. Con la captura múltiple, es posible reducir ese código repetido y hacerlo más eficiente y fácil de mantener.

Diferencias entre *throw* y *throws* en Java

Como vimos en los ejemplos, la palabra clave *throw* se usa para lanzar una excepción explícitamente dentro de un método. Mientras que la palabra clave *throws* se usa para indicar que un método puede disparar una excepción. Por ejemplo:

```
public static void main(String[] args)
{
    EjemploThrows et = new EjemploThrows();
    try
    {
        et.leerArchivo(null);
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
}
```

```
public class EjemploThrows
{
    // Método que puede lanzar una IOException
    public void leerArchivo(String nombreArchivo) throws IOException
    {
        if(nombreArchivo == null)
        {
            throw new IOException("El archivo no puede ser nulo.");
        }
        // Resto del método
    }
}
```

Salida

```
java.io.IOException: El archivo no puede ser nulo.
    at com.cursojava.EjemploThrows.leerArchivo(EjemploThrows.java:12)
    at com.cursojava.ThrowsApp.main(ThrowsApp.java:12)
```

Creación de Excepciones Personalizadas

Además de las excepciones predefinidas que proporciona el lenguaje, también podemos crear excepciones propias. Al definir una excepción propia, es importante seguir ciertos lineamientos. Algunos puntos fundamentales:

- **Herencia de *Throwable*:** Cualquier excepción que crees debe ser una subclase de *Throwable*, ya que es la clase raíz de todas las excepciones y errores en Java.
- **Excepciones verificadas:** Para crear una excepción sea del tipo verificada (checked exception), la clase debe extender *Exception*.
- **Excepciones en tiempo de Ejecución:** Para crear una excepción no verificada (*runtime exception*), que no requiera manejo explícito en tiempo de compilación, se debe extender *RuntimeException*.

Definamos una excepción personalizada y una clase que la lance.

```
public class MyBankAccount
{
    private double saldo;

    public MyBankAccount(double saldoInicial)
    {
        this.saldo = saldoInicial;
    }

    public void retirar(double cantidad) throws MyNotFoundException
    {
        if(cantidad > saldo)
        {
            throw new MyNotFoundException();
        }
        saldo -= cantidad;
    }
}
```

```
public class MyNotFoundException extends Exception
{
    private static final long serialVersionUID = 1L;

    // Constructor por defecto
    public MyNotFoundException()
    {
        super("No hay fondos suficientes.");
    }
    // Constructor con mensaje personalizado
    public MyNotFoundException(String mensaje)
    {
        super(mensaje);
    }
}
```

Uso:

```
public static void main(String[] args)
{
    MyBankAccount cuenta = new MyBankAccount(1000.0);
    try
    {
        cuenta.retirar(1500.0);
    }
    catch(MyNotFoundException e)
    {
        e.printStackTrace();
    }
}
```

[com.cursojava.MyNotFoundException: No hay fondos suficientes.](#)
at com.cursojava.MyBankAccount.retirar(MyBankAccount.java:16)
at com.cursojava.CustomExceptionApp.main(CustomExceptionApp.java:10)

Apéndice C - Networking

El Networking en Java hace referencia a la programación de redes, permitiendo que diferentes dispositivos se comuniquen entre sí usando una red. Es fundamental para crear aplicaciones cliente-servidor, implementar protocolos de red y trabajar con sockets. Java soporta dos protocolos de red:

- **TCP (Transmission Control Protocol):** Permite una comunicación confiable entre dos aplicaciones. Se usa comúnmente sobre el protocolo IP (Internet Protocol) y es conocido como TCP/IP.
- **UDP (User Datagram Protocol):** Un protocolo sin conexión que permite la transmisión de datos entre aplicaciones sin necesidad de mantener una conexión constante.

Programación con Sockets

Un socket proporciona un mecanismo de comunicación entre dos máquinas utilizando el protocolo TCP. Este es uno de los conceptos más usados en la programación de redes y permite establecer conexiones bidireccionales.

Proceso para Establecer una Conexión TCP

- El servidor crea un objeto *ServerSocket* que define en qué puerto se llevará a cabo la comunicación.
- El servidor invoca el método *accept* para esperar que un cliente se conecte.
- El cliente crea un objeto *Socket*, especificando el nombre del servidor y el puerto al que desea conectarse.
- Si la conexión es exitosa, el cliente y el servidor pueden comunicarse a través de los flujos de entrada y salida (*InputStream* y *OutputStream*).

Ejemplo de cliente y servidor

La aplicación servidor quedaría:

```
public class ServerTest extends Thread
{
    private static int port = 6543;

    public static void main(String[] args)
    {
        try
        {
            MyServer t = new MyServer(port);
            t.start();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}

public class MyServer extends Thread
{
    private ServerSocket servidorSocket;

    public MyServer(int puerto) throws IOException
    {
        servidorSocket = new ServerSocket(puerto);
        servidorSocket.setSoTimeout(60*1000);
    }

    public void run()
    {
        while(true)
        {
            try
            {
                System.out.println("[server] Esperando conexión en el puerto " + servidorSocket.getLocalPort() + "...");
                Socket servidor = servidorSocket.accept();

                System.out.println("[server] Conectado a " + servidor.getRemoteSocketAddress());
                DataInputStream entrada = new DataInputStream(servidor.getInputStream());

                System.out.println(entrada.readUTF());
                DataOutputStream salida = new DataOutputStream(servidor.getOutputStream());
                salida.writeUTF("[server] Se ha conectado a " + servidor.getLocalSocketAddress());

                servidor.close();
            }
            catch (SocketTimeoutException s)
            {
                System.out.println("[server] Tiempo de espera agotado.");
                break;
            }
            catch (IOException e)
            {
                e.printStackTrace();
                break;
            }
        }
    }
}
```

Esta aplicación debemos exportarla y ejecutarla aparte de la aplicación cliente. La aplicación cliente quedaría:

```
public static void main(String[] args)
{
    String servidor = "localhost";
    int puerto = 6543;
    try
    {
        System.out.println("Conectando a " + servidor + ":" + puerto + "...");
        Socket cliente = new Socket(servidor, puerto);

        System.out.println("Conectado a " + cliente.getRemoteSocketAddress());
        OutputStream salida = cliente.getOutputStream();
        DataOutputStream salidaServidor = new DataOutputStream(salida);

        salidaServidor.writeUTF("Hola desde " + cliente.getLocalSocketAddress());
        InputStream entrada = cliente.getInputStream();
        DataInputStream entradaServidor = new DataInputStream(entrada);

        System.out.println("El servidor dice: " + entradaServidor.readUTF());
        cliente.close();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
}
```

Conectando a localhost:6543...
Conectado a localhost/127.0.0.1:6543
El servidor dice: [server] Se ha conectado a /127.0.0.1:6543

URL y URLConnection

Una URL (*Uniform Resource Locator*) es una dirección que representa un recurso en la Web, como una página web o un archivo FTP. En Java, la clase `URL` permite crear, manipular y analizar URLs. Pasos para Conectar a una URL:

- Usar el método `openConnection` de la clase `URL` para obtener un objeto de conexión.
- Configurar los parámetros de la conexión.
- Crear una conexión al recurso remoto usando el método `connect`.
- Una vez conectado, acceder al contenido y a los encabezados del recurso.

Veamos un ejemplo práctico donde nos conectamos a una página y leemos su contenido.

```
public static void main(String[] args)
{
    try
    {
        // Crear URL y conectarse
        URL urlObj = new URL("https://www.mercadolibre.com");
        URLConnection connection = urlObj.openConnection();

        // Crear los streams de entrada y salida
        InputStream inputStream = connection.getInputStream();
        BufferedInputStream reader = new BufferedInputStream(inputStream);

        // Leer desde la URL y escribir al archivo
        byte[] buffer = new byte[4096];
        int bytesRead = -1;
        while((bytesRead = reader.read(buffer)) != -1)
        {
            String out = "";
            for(int i = 0; i < bytesRead; ++i)
            {
                out += ((char)buffer[i]);
            }
            System.out.print(out);
        }
        reader.close();
    }
    catch(MalformedURLException e)
    {
        System.out.println("URL con formato no válido: " + e.getMessage());
    }
    catch(IOException e)
    {
        System.out.println("Error de entrada/salida: " + e.getMessage());
    }
}
```

```
<!DOCTYPE html>
<html lang="es">
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Mercado Libre - Envíos Gratis en el día</title>
  <meta name="description" content="Compra productos con Envíos Gratis en el día en Mercado Libre. Encuentra miles de marcas y productos a precios increíbles.">
  <meta name="google-site-verification" content="8ksh-10j9m6c-H16d0T3K2p6c-30350C1dM0c"/>
  <link rel="shortcut icon" href="https://http2.mlstatic.com/ui/navigation/5.18.1/mercadolibre/favicon.ico">
  <link rel="apple-touch-icon" href="https://http2.mlstatic.com/ui/navigation/5.6.1/mercadolibre/0b0d0-precomposed.png">
  <link rel="apple-touch-icon" sizes="76x76" href="https://http2.mlstatic.com/ui/navigation/5.6.1/mercadolibre/76x76-precomposed.png">
  <link rel="apple-touch-icon" sizes="120x120" href="https://http2.mlstatic.com/ui/navigation/5.6.1/mercadolibre/120x120-precomposed.png">
  <link rel="apple-touch-icon" sizes="152x152" href="https://http2.mlstatic.com/ui/navigation/5.6.1/mercadolibre/152x152-precomposed.png">
  <link rel="canonical" href="https://www.mercadolibre.com" data-head-react="true">
  <link rel="alternate" hreflang="es-es" href="https://www.mercadolibre.com.ar"/>
  <link rel="alternate" hreflang="es-bo" href="https://www.mercadolibre.com.bo"/>
  <link rel="alternate" hreflang="es-cl" href="https://www.mercadolibre.com.cl"/>
  <link rel="alternate" hreflang="es-co" href="https://www.mercadolibre.com.co"/>
```

Tabla de contenidos

Introducción a la programación.....	1
Algoritmo, datos y funciones.....	1
Clases, instancias y objetos.....	2
Tipos de lenguajes de programación.....	2
Pasos del desarrollo de un programa.....	3
Desarrollo lógico.....	3
Codificación.....	3
Compilación o interpretación.....	3
Depuración.....	3
Documentación.....	4
Entorno de desarrollo.....	4
Estructura de un programa en Java.....	4
Paquete (<i>package</i>).....	4
Importaciones (<i>import</i>).....	5
Clase principal y método de entrada (<i>main</i>).....	5
Tipos y estructuras de datos.....	6
Conversión de datos.....	7
Estructuras de datos.....	7
Arreglo (<i>array</i>).....	8
Matriz (<i>matrix</i>).....	8
Listas enlazadas.....	8
Pila (<i>stack</i>).....	8
Cola (<i>queue</i>):.....	9
Conjunto (<i>set</i>).....	9
Diccionario (<i>map</i>).....	9
Árbol (<i>tree</i>).....	9
Manejo de cadenas de caracteres en Java.....	10
Operadores.....	13
Operadores aritméticos.....	13
Operadores de asignación.....	13
Operadores relacionales o de comparación.....	13
Operadores lógicos.....	14
Operadores binarios.....	14
Clase Math.....	15
Estructuras de control.....	16
Estructuras condicionales o de control de selección.....	16
Si simple (<i>if</i>).....	16
Si-Sino (<i>if-else</i>).....	16
Si múltiple (<i>if-else if-else</i>).....	16
Selector por caso (<i>switch case</i>).....	17
Estructuras de control de flujo o bucles.....	17
Mientras (<i>while</i>).....	17
Hacer mientras (<i>do while</i>).....	18
Para (<i>for</i>).....	18

Para cada (<i>for</i>).....	18
Instrucciones de control de flujo.....	19
Funciones, métodos y procedimientos.....	20
Diferencia entre función, método y procedimiento.....	20
Ámbito de las variables.....	20
Funciones recursivas.....	21
Clases y objetos.....	22
Clases y objetos.....	22
Visibilidad de los miembros.....	22
Constructor de la clase.....	23
Objeto <i>this</i>	23
Encapsulamiento.....	24
Herencia.....	24
Polimorfismo.....	25
Polimorfismo de sobrecarga.....	26
Polimorfismo de sobrescritura.....	26
Clases abstractas.....	27
Interfaces.....	28
Entrada y salida y datos.....	29
Entrada y salida por consola.....	29
Entrada de datos (clase <i>Scanner</i>).....	29
Salida de datos (<i>System.out</i>).....	29
Streams.....	30
Leer y escribir bytes en un archivo.....	30
Leer y escribir texto en un archivo.....	31
Excepciones.....	32
Streams estándar.....	33
Manejo de archivos y directorios con la clase <i>File</i>	34
Obtener información de archivos y directorios.....	34
Creación de Directorios.....	34
Listado de contenido de directorios.....	35
Eliminar Directorios.....	35
Multitarea e Hilos.....	36
Tipos de multitarea.....	36
Multitarea por procesos.....	36
Multitarea por subprocesos (<i>Multithreading</i>).....	36
Hilos (<i>Threads</i>).....	36
Características.....	36
Tipos de Hilos.....	36
Implementación de hilos.....	37
Esperar fin de hilos.....	38
Sincronizar recursos.....	39
Comunicación entre Hilos.....	40
Apéndice A - Estructuras de datos.....	41
Listas y colecciones.....	41
Arrays.....	41

Listas enlazadas.....	42
Pilas.....	42
Colas.....	43
Matrices.....	43
Conjuntos.....	44
Diccionarios (mapas).....	44
Apéndice B - Excepciones.....	46
Múltiples bloques catch y múltiples excepciones en un bloque catch.....	47
Diferencias entre <i>throw</i> y <i>throws</i> en Java.....	48
Creación de Excepciones Personalizadas.....	49
Apéndice C - Networking.....	50
Programación con Sockets.....	50
Proceso para Establecer una Conexión TCP.....	50
Ejemplo de cliente y servidor.....	50
URL y URLConnection.....	51