Java

Introducción

Java es un **lenguaje orientado a objetos**. En la década de los 60 nació la programación estructurada impulsada por lenguajes como Pascal o C. Con el aumento de la complejidad de los programas se adoptó un nuevo enfoque como es la programación orientada a objetos o POO.

Java fue desarrollado por James Gosling, Patrick Naughton, Chris Warth, Ed Frank y Mike Sheridan en Sun Microsystems en 1991. Inicialmente, el lenguaje se denominó *Oak* aunque se rebautizó como **Java** en 1995. La concepción de Java se basa en C y C++.

Su objetivo inicial era crear un lenguaje independiente y portátil capaz de ser ejecutado en diferentes plataformas y con diferentes CPUs. La clave está en que el resultado de la compilación de código Java no es código ejecutable, sino código de bytes o *bytecode*. Es un conjunto optimizado de instrucciones diseñadas para ejecutarse sobre la JVM o *Java Virtual Machine*. Esta máquina virtual es la que interpreta el *bytecode*. De esta forma sólo es necesario implementar la JVM para cada plataforma.

Desde un punto de vista general, un programa se puede organizar de dos formas: sobre su código (lo que sucede) y sobre sus datos (lo que se ve afectado). En la programación estructurada se organiza sobre el código pero en la programación orientada a objetos el programa se estructura alrededor de los datos, definiendo estos datos y las rutinas que permiten actuar sobre los mismos.

Para complementar los principios de la programación orientada a objetos, se aplican los conceptos de **encapsulación, herencia y polimorfismo**.

- La **encapsulación** es un mecanismo que combina el código con los datos que manipula, al tiempo que los protege de interferencias externas. La unidad básica de encapsulación es la **clase**. La clase define la forma de un objeto y especifica los datos y el código que actúa sobre ellos. Los objetos son instancias de una clase.

- El **polimorfismo** es la propiedad que permite a una interfaz acceder a una clase general de acciones. Este concepto suele expresarse como "una interfaz, múltiples métodos". El compilador en tiempo de ejecución será el encargado de seleccionar el método correcto a invocar.

- La **herencia** es el proceso mediante el cual un objeto puede adquirir las propiedades de otro. Gracias a la herencia un objeto solo tiene que definir los atributos que lo hacen único dentro de la clase y heredar los atributos generales.

Comandos de Java

```
# Compilar código Java en bytecode
$ javac filename.java

# Ejecutar aplicación Java
$  java filename

# Generar páginas HTML de documentación de la API
$ javadoc

# Inicia el intérprete de comandos
$ jshell

# Iniciar una consola gráfica para monitorear y gestionar aplicaciones Java
$ jconsole
```

- Java® Development Kit Version 22 Tool Specifications

Sintaxis básica

Comentarios

```
// Comentarios de una sóla línea

/*
Comentarios multilínea
*/

/**
 * Los comentarios JavaDoc suelen describir la clase o varios atributos de una clase.
 */
public class Sample {
    // ...
}
```

Identificadores

En Java, un **identificador** es un nombre asignado a un método, variable u otro elemento definido por el usuario. Pueden tener uno o varios caracteres de longitud.

Los nombres de variable deben cumplir ciertas restricciones:

- Pueden empezar por **cualquier letra, guión bajo(`_`) o el símbolo** `$`.
- El siguiente carácter puede ser **cualquier letra, dígito, guión bajo(`_`) o el símbolo** `$`.
- Por lo tanto no pueden empezar con un dígito ni emplear palabras clave de Java.
- No pueden incluir espacios ni caracteres especiales distintos de `_` y `$`.

> **No es recomendable usar el símbolo** `$` **en nombres de variables comunes**, ya que generalmente está reservado para usos internos o generados automáticamente por el compilador.

Java es ***"case sensitive"*** lo que significa que Java distingue entre mayúsculas y minúsculas.

En Java, las convenciones de nombres más comunes incluyen:

- **camelCase**: se utiliza para nombres de variables y métodos, como `edadDelCapitan`, `nombreUsuario`, `calcularTotal()`.
- **PascalCase**: similar a la anterior, pero la primera letra también es mayúscula. Se usa para nombres de clases y interfaces, como `Persona`, `CuentaBancaria`.
- **UPPER_SNAKE_CASE**: se utiliza para las constantes, es decir, variables que son `static final`. En esta convención todas las letras son mayúsculas y se separan con guiones bajos, como `MAX_VALUE`, `PI` o `DAYS_OF_WEEK`.

Tipos y variables

Los tipos de datos son especialmente importantes en Java por tratarse de un lenguaje de **tipado fuerte**. Es decir, el compilador comprueba la compatibilidad de los tipos en todas las operaciones. Para realizar la comprobación de tipos, todas las variables, expresiones y valores tienen un tipo.

Al determinar el tipo para una variable, estamos indicando cuál es la información que vamos a poder almacenar en esta variable y las operaciones que podremos efectuar con ella.

En Java se declara una variable usando `<tipo> <nombre>`. Es necesario declarar la variable antes de poder hacer referencia a ella. Una vez se ha declarado ya se puede utilizar, nunca antes.

Por lo general, debe asignar un valor a una variable antes de poder usarla aunque en determinados casos Java puede inicializar el valor de las variables, como por ejemplo en **variables de instancia**.

- **Variables locales**: se declaran dentro de un método, constructor o bloque, y solo están accesibles dentro de ese contexto. No tienen un valor por defecto, por lo que deben ser inicializadas antes de usarse.
- **Variables de instancia**: se declaran dentro de una clase pero fuera de cualquier método, constructor o bloque. Cada objeto de la clase tiene su propia copia de estas variables. Si no se inicializan explícitamente, las variables de instancia reciben un valor por defecto basado en su tipo.
- **Variables de clase (o estáticas)**: se declaran con la palabra clave `static`. Solo hay una copia de estas variables que es compartida por todas las instancias de la clase. Se almacenan en memoria cuando la clase es cargada por el ClassLoader.

Valores por defecto

En el caso de las **variables de instancia**, si no se inicializan de forma explícita, el compilador establecerá un valor basado en su tipo:

- Tipos primitivos:
  - `int`, `short`, `byte`, `long`: 0
  - `float`: 0.0f
  - `double`: 0.0d
  - `char`: `'\u0000'` (carácter nulo)
  - `boolean`: false
- Objetos y referencias (como `String` o cualquier objeto): `null`

> Sin embargo, confiar en estos valores predeterminados generalmente se considera una **práctica poco recomendable**. Aunque estos valores por defecto evitan errores de compilación, la inicialización explícita mejora la legibilidad y evita posibles errores sobre el estado inicial de una variable.

Tipos primitivos

Los tipos primitivos en Java tienen **rangos definidos**, y al intentar asignar directamente un valor fuera del rango, el compilador **generará un error**. Sin embargo, para los tipos primitivos numéricos, si una operación aritmética excede el rango permitido, se producirá un **desbordamiento**, resultando en un valor inesperado, pero sin generar un error de compilación.

> El desbordamiento ocurre en **tiempo de ejecución** y puede llevar a resultados incorrectos sin advertencias del compilador.

Java no tiene tipos sin signo (excepto `int` y `long` con métodos específicos introducidos en Java SE 8), por lo que todos los tipos numéricos tienen signo.

byte

El tipo primitivo `byte` es un entero de complemento de dos con signo de 8 bits. Es una opción eficiente para ahorrar memoria en grandes colecciones de datos numéricos pequeños. También se utiliza frecuentemente en la manipulación de datos binarios y en aplicaciones que requieren optimización de memoria.

```
// (-128 <= byte <= 127)
byte foo = 100;
```

> El rango de valores permitidos para `byte` es de **-128** a **127** (ambos inclusive). Si se intenta asignar un valor fuera de este rango, el compilador generará un error.

short

El tipo primitivo `short` es un entero de complemento de dos con signo de 16 bits. Es una opción intermedia entre `byte` e `int` para almacenar números pequeños en aplicaciones que requieren optimización de memoria.

java

```
// (-32,768 <= short <= 32,767)
short foo = 10000;
```

> El rango de valores permitidos para `short` es de **-32,768** a **32,767** (ambos inclusive). Si se intenta asignar un valor fuera de este rango, el compilador generará un error.

int

El tipo primitivo `int` es un entero de complemento de dos con signo de 32 bits. Es el tipo predeterminado para los literales enteros.

A partir de Java SE 8 y versiones posteriores, se puede utilizar este tipo para representar un entero de 32 bits sin signo, que tiene un valor mínimo de 0 y un valor máximo de 2^32-1.

```
// (-2,147,483,648 <= int <= 2,147,483,647)
int foo = 1;
```

> El rango de valores permitidos para `int` es de **-2^31** a **2^31-1** (ambos inclusive). Si se intenta asignar un valor fuera de este rango, el compilador generará un error.

long

El tipo primitivo `long` es un entero de complemento de dos con signo de 64 bits.

A partir de Java SE 8 y versiones posteriores, se puede utilizar este tipo para representar un entero de 64 bits sin signo, que tiene un valor mínimo de 0 y un valor máximo de 2^64-1.

```
// (-9,223,372,036,854,775,808 <= long <= 9,223,372,036,854,775,807)
long foo = 100000L;
```

El sufijo 'L' o 'l' (recomendable en mayúsculas por claridad) se utiliza para indicar que el valor literal asignado a la variable es de tipo `long`. Cualquier literal sin este sufijo se considera un entero por defecto (tipo `int`).

> El rango de valores permitidos para `long` es de **-2^63** a **2^63-1** (ambos inclusive). Si se intenta asignar un valor fuera de este rango, el compilador generará un error.

float

El tipo primitivo `float` es un número de coma flotante IEEE 754 de precisión simple de 32 bits. Puede consultarse el rango de valores en la documentación oficial.

```
float foo = 234.5f;
```

El sufijo 'f' o 'F' se utiliza para indicar que el valor literal asignado a la variable es de tipo `float`. Cualquier literal sin este sufijo se considerará como un `double` por defecto.

> Este tipo de dato nunca debe utilizarse para valores que requieran alta precisión, como la moneda. Para estos casos, es recomendable utilizar el tipo `java.math.BigDecimal` en su lugar.

double

El tipo primitivo `double` es un número de coma flotante IEEE 754 de precisión doble de 64 bits. Para los valores decimales, este tipo de datos es la opción predeterminada.

Puede consultarse el rango de valores en la documentación oficial.

```
double foo = 123.4;
```

El sufijo 'd' o 'D' es opcional ya que cualquier literal sin este sufijo se considerará como un `double` por defecto.

boolean

Este tipo de dato sólo admite dos valores posibles: `true` o `false`.

El tipo de dato `boolean` en Java sólo admite dos valores posibles: `true` o `false`. Los valores booleanos se utilizan comúnmente en condiciones y estructuras de control.

```
boolean foo = true;
boolean bar = false;

boolean baz = !foo; // baz será false
```

Además, se pueden combinar expresiones booleanas utilizando operadores lógicos:

```
boolean condition1 = (foo && bar); // AND: será false
boolean condition2 = (foo || bar); // OR: será true
```

char

El tipo primitivo `char` representa un **único carácter sin signo de 16 bits**. Esto permite realizar operaciones aritméticas sobre valores de tipo `char`. Este tipo está basado en la codificación UTF-16, lo que le permite representar una amplia variedad de caracteres, incluyendo caracteres especiales mediante secuencias de escape.

```
char fooChar = 'A';
fooChar++; // now fooChar == 'B'
```

Las constantes de carácter se incluyen entre comillas simples.

A continuación, se presentan algunas de las secuencias de escape más comunes:

- `\'` - Comilla simple
- `\"` - Comilla doble
- `\\` - Barra invertida
- `\r` - Retorno de carro
- `\n` - Nueva línea
- `\f` - Salto de formulario
- `\t` - Tabulación horizontal
- `\b` - Retroceso
- `\ddd` - Constante octal (donde 'ddd' es una constante octal)
- `\uxxxx` - Constante hexadecimal (donde 'xxxx' es una constante hexadecimal)

Valores literales

En Java, un literal es un valor fijo representado en un formato legible para los humanos. Por ejemplo, el número 100 es un literal. Los literales también suelen denominarse constantes.

Por defecto, los literales enteros son de tipo `int` y los literales de coma flotante son de tipo `double`.

Los literales de carácter se incluyen entre comillas simples, mientras que los literales de cadena son conjuntos de caracteres que se encierran entre comillas dobles.

```
int a = 100;
long b = 100L;
double c = 100.5;
float d = 100.5f;
char f = 'f';
String str = "Literal de cadena";

int hexadecimal = 0xFF; // Formato hexadecimal que corresponde a 255 en decimal
int octal = 011; // Formato octal que corresponde a 9 en decimal
```

A partir de JDK 7, se pueden emplear guiones bajos para mejorar la legibilidad de literales enteros o flotantes:

```
int x = 123_456_789;
double z = 123_456_789.5;
```

La palabra clave `final` se utiliza para hacer que las variables sean **inmutable**. Por convención, el nombre de la variable se declara en mayúsculas:

```
final int HORAS_QUE_TRABAJO_POR_SEMANA = 9001;
```

También existe una notación abreviada para declarar (e inicializar) múltiples variables del mismo tipo, aunque no se aconseja su uso por motivos de legibilidad:

```java
// Declarar las tres variables del mismo tipo
int x, y, z;

// Declarar e inicializar las variables
int i1 = 1, i2 = 2;

// El símbolo '=' retorna el valor de su derecha, por lo que esta forma es válida
int a = b = c = 100;
```

Bloques

Un **bloque de código es un grupo de dos o más instrucciones definidas entre llaves (`{}`)**. Al crear un bloque de código, se convierte en una unidad lógica que se puede usar como si fuera una instrucción independiente.

Un bloque de código define un **ámbito**. Las variables definidas dentro de un ámbito o bloque de código no son accesibles fuera de ese ámbito. Cada vez que se accede a un bloque, las variables contenidas en ese bloque se inicializan, y cuando el bloque finaliza, se destruyen.

Una variable está disponible a partir de su definición. Por lo tanto, si se define una variable al final de un bloque, no se podrá utilizar (y tampoco tiene sentido).

Los **bloques se pueden anidar**, lo que significa que un bloque de código puede contener a otro bloque de código. Desde un bloque interior se puede acceder a las variables definidas en el bloque exterior, pero el bloque exterior no puede acceder a las variables definidas en el bloque interior.

```java
public class Bloques {
    public static void main(String ... args) {
        String exterior = "Bloque exterior";

        {
            String interior = "Bloque interior";
            System.out.println(interior); // Correcto
            System.out.println(exterior); // Correcto
        }

        System.out.println(exterior); // Correcto
        // System.out.printf(interior); // Error ya que 'interior' no es accesible
    }
}
```

Operadores

```java
// La aritmética es directa
System.out.println("1 + 2 = " + (1 + 2)); // => 3
System.out.println("2 - 1 = " + (2 - 1)); // => 1
System.out.println("2 - 1 = " + (2 - 1)); // => 2
System.out.println("1 / 2 = " + (1 / 2)); // => 0 (0.5 truncado)

// Módulo
System.out.println("11%3 = " + (11 % 3)); // => 2

// Operadores de comparación
System.out.println("3 == 2 " + (3 == 2)); // => false
System.out.println("3 != 2 " + (3 != 2)); // => true
System.out.println("3 > 2 " + (3 > 2)); // => true
System.out.println("3 < 2 " + (3 < 2)); // => false
System.out.println("2 <= 2 " + (2 <= 2)); // => true
System.out.println("2 >= 2 " + (2 >= 2)); // => true

// Asignaciones abreviadas
```

```java
int x += 10; // x = x + 10;
int x -= 10; // x = x - 10;
int x *= 10; // x = x - 10;
int x /= 10; // x = x / 10;
int x %= 10; // x = x % 10;
boolean bool &= true; // bool = bool & true;
boolean bool |= true; // bool = bool | true;
boolean bool ^= true; // bool = bool ^ true;

// Incrementos y decrementos
int y, x = 10;
y = x++; // y = 10. Primero se asigna el valor y luego se aumenta
y = ++x; // y = 11. Primero se aumenta y luego se asigna
y = x--; // y = 10. Primero se asigna el valor y luego se resta
y = --x; // y = 9. Primero se resta y luego se asigna
```

Los operadores lógicos son herramientas fundamentales para realizar evaluaciones condicionales y tomar decisiones en el flujo de un programa.

Estos operadores permiten combinar o modificar expresiones booleanas, que son aquellas que pueden evaluarse como verdaderas o falsas:

`A`	`B`	`A|B`	`A&B`	`A^B`	`!A`
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Los operadores lógicos AND y OR pueden funcionar **en modo cortocircuito (&&) y (||)**. En este modo se evalúa el primer operando y si fuera necesario, se evaluaría el segundo.

Cadenas

```java
String fooString = "¡Mi String está aquí!";

// \n es un carácter escapado que inicia una nueva línea
String barString = "¿Imprimiendo en una nueva linea?\n¡Ningun problema!";

// \t es un carácter escapado que añade un carácter tab
String bazString = "¿Quieres añadir un 'tab'?\t¡Ningun problema!";
```

Conversión de tipos numéricos primitivos en cadenas y viceversa:

```java
Integer.parseInt("123"); // retorna una versión entera de "123"
String.valueOf(123); // retorna una version string de 123
```

Control de flujo

```java
/*
if (expr booleana) {
    bloque de intrucciones;
} else if (expr booleana) {
    bloque instrucciones;
```

```
} else {
    intrucciones en caso de que ninguna condición anterior se cumpla;
} */


/*
while(expr booleana) {
    bloque de instrucciones;
    contador++;  // actualizar la variable usada para evaluar la condición
} */

/*
do {
    bloque de intrucciones
    contador++; // actualizar la variable usada para evaluar la condición
}while(expr booleana);
*/

/*
for(<declaración_de_inicio>; <condicional>; <paso>) {
    bloque de instrucciones;
} */
```

En Java, el cuerpo asociado a un bucle `for` o de otro tipo puede estar vacío ya que una instrucción vacía es sintácticamente válida. Puede ser útil en algunos casos:

```
int sum = 0;
for(int i = 1; i<= 5; sum += i++);
// Se usa el bucle for para incrementar la variable sum
```

En la JDK 5 se añadió los bucles `for-each` que permiten iterar por matrices, clases del paquete 'Collections', etc…

```
/*
for(tipo var-iteración : collection) {
    bloque instrucciones;
} */
```

La estructura `switch` funciona con tipos numéricos simples como `byte`, `short`, `char` e `int`. También funciona con tipos enumerados, la clase `String` y unas pocas clases especiales que envuelven tipos primitivos: `Character`, `Byte`, `Short` e `Integer`.

```
int mes = 3;
switch (mes) {
    case 1:
        System.out.println("Enero");
        break;
    case 2:
        System.out.println("Febrero");
        break;
    case 3:
        System.out.println("Marzo");
        break;
    default:
        break;
}
```

Break

Por medio de la instrucción `break` se puede forzar la salida inmediata de un bucle e ignorar el código restante del cuerpo y la prueba condicional. El control del programa se pasa a la siguiente instrucción después del bucle.

Continue

Con la instrucción `continue` se fuerza una iteración del bucle, es decir, se ignora el código comprendido entre esta instrucción y la expresión condicional que controla el bucle.

Tanto `break` como `continue` pueden funcionar junto a una etiqueta permitiendo dirigir el control del programa al bloque de código indicado por la etiqueta. Un `break` o `continue` etiquetados se declaran con `break {etiqueta}` y `continue {etiqueta}`. El único requisito es que el bloque de código con la etiqueta debe contener la instrucción `break` o `continue`. Es decir, no se puede utilizar un `break` como si fuera una instrucción `goto`.

```
public class Sample{
    public static void main(String ... args){
        for (int i = 0; i < 4; i++) {
            one: {
                two: {
                    if(i == 1) break one;
                    if(i == 2) break two;
                }
                System.out.println("After two");
            }
            System.out.println("After one");
        }
    }
}
```

Paquetes

Todas las clases en Java pertenecen a un paquete. Si no se especifica uno se usa el paquete predeterminado (o global).

Al definir una clase en un paquete, se añade el nombre de dicho paquete a cada clase, lo que evita colisiones de nombres con otras clases. El paquete debe coincidir con la jerarquía de directorios. Los nombres de paquetes se escriben en minúsculas para evitar conflictos con los nombres de clases o interfaces.

Para definir un paquete se utiliza la palabra clave `package`:

```
package paquete1.paquete2....paqueteN;
```

Importación

Cuando se usa una clase de otro paquete, se debe cualificar su nombre con el nombre de su paquete, como por ejemplo `java.util.ArrayList`.

Sin embargo, podemos usar la palabra clave `import` para importar uno o varios miembros de un paquete. El paquete `java.lang` es exclusivo ya que se importa automáticamente en todos los programas Java.

Por tanto se puede importar una **clase concreta** o importar **todas las clases** de un paquete con el astericos (`*`).

La sentencia `import` se utiliza después de la sentencia `package` si existe.

```
package com.example;

//import java.lang.*; // importación automática por defecto
```

```java
import java.util.HashMap;

public class example {
    public static void main(String[] args) {
        // Al no realizar la importación hay que cualificar el nombre
        java.util.ArrayList aList = new java.util.ArrayList();

        // Al realizar la importación podemos usar 'HashMap'
        HashMap<Integer, String> hMap = new HashMap<>();
    }
}
```

Importación estática

Java admite la importación de campos estáticos finales (constantes) y de métodos estáticos usando la forma `import static`. Al usar este tipo de importación, se puede hacer referencia directamente a miembros estáticos por sus nombres, sin necesidad de calificarlos con el nombre de su clase.

```java
import static java.lang.Math.sqrt;
// import static java.lang.Math.pow;
// import static java.lang.Math.*; // importa todos los miembros estáticos

void operation () {
    sqrt(9); // con importación estática
    Math.pow(5, 8); // sin importación estática
}
```

Arrays

Notación para la declaración de un array (el tamaño del array debe decidirse en la declaración):

- `<tipo_de_dato> [] <nombre_variable> = new <tipo_de_dato>[<tamaño>];`

```java
int[] sample = new int[10];
int sample[] = new int[5];
String[] sample = new String[1];
boolean[] sample = new boolean[100];
int[] sample1, sample2, sample3;
```

Notación para la declaración e inicialización de un array:

- `<tipo_de_dato> [] <nombre_variable> = {value, value, ...};`

```java
int[] sample = {2015, 2016, 2017};
```

Los arrays comienzan su indexación en cero y son **mutables**:

```java
sample[1] = 2018;
System.out.println("Year: " + sample[1]); // => 2018
```

Acceder un elemento dentro de un array (un intento de acceso fuera de los límites del array lanza un `ArrayIndexOutOfBoundsException`):

Al asignar una referencia de una matriz a otra referencia no se crea una copia de la matriz ni se copian los contenidos. Sólo se crea una referencia a la misma matriz, al igual que sucede con cualquier otro objeto. Por lo tanto, a partir de ambas referencias se accede al **mismo array**:

```
int[] nums = {1, 2, 3};
int[] other = nums; // Ahora 'other' apunta a la misma matriz que 'nums'.
```

Clases y objetos

Una definición de clase crea un **nuevo tipo de datos**:

```
class Bicicleta {

    // Campos o variables de instancia
    public String nombre; // Puede ser accedido desde cualquier parte
    private double precio;  // Accesible sólo desde esta clase
    protected int velocidad; // Accesible desde esta clase, sus subclases o el mismo paquete
    int numMarchas; // default: Sólo accesible desde este paquete

    // Constructores son la manera de crear clases
    // Este es un constructor por defecto
    public Bicicleta() {
        numMarchas = 18;
        precio = 2495.99;
        velocidad = 45;
        nombre = "Bontrager";
    }

    // Este es un constructor específico (contiene argumentos)
    public Bicicleta(String nombre) {
        super(); // llamada al constructor sin parámetros 'Bicicleta()';
        this.nombre = nombre;
    }

    // Este es un constructor específico (contiene argumentos)
    public Bicicleta(String nombre, double precio) {
        this(nombre); // llamada al constructor 'Bicicleta(String nombre)';
        this.precio = precio;
    }

    // Sintaxis de método:
    // <public/private/protected> <tipo_de_retorno> <nombre_funcion>(<argumentos>)

    // Las clases de Java usualmente implementan métodos 'get' (obtener)
    // y 'set' (establecer) para sus campos

    // Sintaxis de declaración de métodos
    // <alcance> <tipo_de_retorno> <nombre_metodo>(<argumentos>)
    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }

    // ....

    //Método para mostrar los valores de los atributos de este objeto.
    @Override
    public String toString() {
        return "Bicicleta{" +
                "nombre='" + nombre + '\'' +
                ", precio=" + precio +
                ", velocidad=" + velocidad +
```

```
                ", numMarchas=" + numMarchas +
                '}';
    }
}
```

Todas las clases tienen al menos un constructor predeterminado ya que Java ofrece automáticamente un constructor que inicializa todas las variables miembro en sus valores predeterminados que son **cero(0), 'null' y 'false'**. Cuando se crea un constructor el predeterminado deja de usarse.

Hay otra forma de `this` que permite que un constructor invoque a otro dentro de la misma clase. Cuando se ejecuta `this(lista-args)`, el constructor sobrecargado que encaja con la lista de parámetros especificada se ejecutará y se **ejecutará primero**. Por tanto no se puede usar `this()` y `super()` al mismo tiempo ya que ambos deben ser la primera instrucción.

El operador `new` asigna dinámicamente, es decir, en tiempo de ejecución, memoria para un objeto y devuelve una referencia al mismo. Esta referencia es, ni más ni menos, que la dirección en memoria del objeto asignado por `new`, que después se almacena en una variable para poder ser utilizada posteriormente.

```
Bicicleta bicicleta = new Bicicleta();
Bicicleta bicicleta2 = bicicleta; // Ambas variables hacen referencia al mismo objeto
```

Clases anidadas

La clases anidadas no estáticas también se denominan **clases internas**. Una clase interna no existe independientemente de su clase contenedora, ya que el ámbito de una clase interna lo define la clase externa. También se pueden definir clases que sean locales de un bloque.

Una clase interna tiene acceso a todas las variables y métodos de su clase externa y puede hacer referencia a los mismos directamente como hacen otros miembros no estáticos de la clase externa.

```
class Outern {
    int a = 5;
    int b = 10;

    void sum() {
        Intern intern = new Intern();
        System.out.println(intern.operation());
    }

    class Intern {
        int operation() {
            return a + b;
        }
    }
}
```

Métodos

Notación para la definición de un método:

- `<visibilidad> <tipo_de_retorno> <nombre_funcion>(<argumentos>)`

Los **parámetros** aparecen en la definición del método. Cuando un método tiene parámetros la parte de su definición que los especifica se denomina 'lista de parámetros'.

La **firma** de un método se compone del **nombre del método y la lista de parámetros**.

Hablamos de **argumentos** cuando usamos valores concretos para realizar la llamada al método. El valor concreto pasado a un método es el argumento. Dentro del método, la variable que recibe el argumento es el parámetro.

```
int sum(int a, int b) { // lista de parámetros del método. Junto con el nombre forman la firma
    return a + b;
}

sum(10, 20); // Llamada al método usando dos argumentos o valores
```

Para la devolución de un valor en un método se utiliza la palabra clave `return`. La sentencia `return` tiene dos formas: una forma sirve para devolver un valor y la otra sirve para salir de un método cuando retorna `void`:

```
int sum(int a, int b) {
  return a + b;
}

void isEven(int num) {
  if(num % 2 == 0)
    return;
  else
    System.out.println("Num is odd");
}
```

En Java, cuando se pasa como argumento **un tipo primitivo se pasa por valor**, esto es, se crea una copia del argumento y los cambios que suceden dentro del método no afecta al exterior. En cambio, cuando se pasa un **objeto se pasa implícitamente por referencia**, ya que cuando se crea una variable de un tipo de clase se crea una referencia a un objeto y es la referencia y no el objeto lo que se pasa al método. Los cambios realizados en el objeto dentro del método afectan al objeto.

Podemos indicar un parámetro como `final` lo que impedirá que podamos asignar una referencia de un nuevo objeto aunque no impedirá que se realicen cambios en los atributos del objeto:

```
class Car {
  public int speed;

  public Car(int speed) {
    this.speed = speed;
  }
}

class Main {
  public static void main(String[] args) {
    Car car = new Car(100);
    System.out.println("Speed: " + car.speed); // Prints '100'
    bicycle = incrementSpeed(bicycle);
    System.out.println("Speed: " + car.speed); // Prints '125'

  }

  public static Bicycle incrementSpeed(final Bicycle bicycle) {
    bicycle.speed = 125;  // Podemos asignar nuevos valores a los atributos del objeto
    // bicycle = new Bicycle(125); // ERROR! La variable 'bicycle es una variable final
    return bicycle;
  }
}
```

Sobrecarga de métodos

La sobrecarga de métodos es una de las técnicas de Java para implementar el **polimorfismo**. En Java, dos o más métodos de la misma clase pueden compartir el mismo nombre siempre y cuando su **firma sea diferente**. Por tanto, para sobrecargar un

método, basta con declarar métodos con distinta firma. En Java, la firma de un método es el **nombre del método más su lista de parámetros**, sin incluir el tipo devuelto. Por tanto, la sobrecarga de métodos son métodos con el mismo nombre pero distinta lista de parámetros, sin tener en cuenta el tipo de devolución.

Por ejemplo, en la clase `java.lang.Math` se utiliza la sobrecarga de métodos para disponer de varios métodos que realizan la misma operación sobre tipos diferentes:

```
public static double abs(double a)
public static float abs(float a)
public static long abs(long a)
public static int abs(int a)
```

Argumentos de longitud variable: `varargs`

En ocasiones será necesario métodos que acepten una número variable de argumentos. Se define con el símbolo (`...`).

La firma de un método con argumentos de longitud variable es:

- `tipo método(tipo ... var) {}`

Dentro del método esta variable se utiliza como una array. Por lo tanto, para acceder a los parámetros se emplea la misma notación que se emplea en un array. Un método puede tener parámetros normales además de parámetros de longitud variable. En ese caso, **los parámetros normales van delante y por último el parámetro de longitud variable**.

Modificador `static`

Se pueden definir como `static` tanto variables como métodos. Las variables declarados como `static` son básicamente **variables globales**. Todas las instancias de la clase comparten la misma variable.

Los métodos `static` tienen ciertas restricciones:

- Sólo pueden invocar directamente otros métodos `static`
- Sólo pueden acceder directamente a datos `static`
- Carecen de una referencia `this`

Bloque `static`

Cuando una clase requiere de cierta inicialización antes de que pueda crear objetos se puede usar un bloque `static` que se ejecuta al cargar la clase por primera vez:

```
class staticBlock {
  static int a;
  static int b;

  // Este bloque se ejecuta al cargar la clase por primera vez
  // y antes que cualquier otro método 'static'
  static {
    a = 5;
    b = 10;
  }
}
```

Herencia

La **herencia** es uno de los tres principios fundamentales de la programación orientada a objetos ya que permite crear clasificaciones jerárquicas.

Se invoca al constructor de la superclase con `super(lista-parámetros)`. Esta instrucción debe ser **siempre la primera instrucción** ejecutada dentro del constructor de la subclase. El constructor de la superclase inicializa la parte de la superclase y el constructor de la subclase la parte de la subclase. En una jerarquía de clases, los constructores se invocan en orden de derivación, de **superclase a subclase**.

Con `super.miembro` en donde miembro puede ser un método o una variable de instancia, podemos hacer referencia a métodos o variables de la superclase desde una subclase.

Java es un lenguaje de **tipado fuerte**. Por lo tanto una variable de tipo sólo puede hacer referencia a objetos de ese tipo. Sin embargo, existe una excepción cuando aplicamos la herencia. Se puede asignar a una variable de referencia de una superclase una referencia a un objeto de cualquier subclase derivada de dicha superclase. Es decir, una referencia de superclase puede hacer referencia a un objeto de subclase.

Hay que tener en cuenta que cuando se asigna una referencia de un objeto de subclase a una variable de referencia de superclase **sólo** se tiene acceso a las partes del objeto que defina la superclase.

```
class Vehicle {
    void echo() {}
}

class Car extends Vehicle {
    void gamma(){}

    void sample() {
        // El tipo 'Car' es una subclase de 'Vehicle'
        Vehicle vehicle = new Car();
        vehicle.echo(); // Correcto
        // vehicle.gamma(); // Incorrecto.
        // Sólo tenemos acceso a las partes que definen la superclase.
    }
}
```

Sobreescritura de métodos

En una jerarquía de clases, cuando un método de una subclase tiene el mismo tipo de devolución y firma (nombre y parámetros) que un método de su superclase, el método de la subclase reemplaza o sobreescribe al de la superclase.

Si la firma no es exacta, ya no hablamos de sobreescritura de métodos sino de sobrecarga de métodos.

La sobreescritura de métodos es importante porque es la forma de implementar el **polimorfismo** en Java. El compilador, en tiempo de ejecución, será el encargado de invocar el método adecuado.

Si usamos la anotación `@Override` en un método le estamos indicando al compilador que es un método sobreescrito y por tanto puede realizar las comprobaciones pertinentes en tiempo de compilación, como por ejemplo que el método original sigue existiendo en la superclase o que no ha sido modificado.

```
class Vehicle {
  void show() {}
}

class Car extends Vehicle {
  @Override
  void show() {}
}

class Motocycle extends Vehicle {
```

```
  @Override
  void show() {
    super.show(); // Podemos invocar al método 'show()' de la superclase
  }
}

public class Sample {
  public static void main(String ... args) {
    Vehicle vehicle1 = new Car();
    Vehicle vehicle2 = new Motocycle();
    vehicle1.show(); // El compilador invoca el método 'show()' de 'Car'
    vehicle2.show(); // El compilador invoca el método 'show()' de 'Motocycle'

  }
}
```

Clases abstractas

Una clase que defina uno o varios métodos abstractos debe definirse como `abstract`. Un método abstracto carece de cuerpo y debe ser implementado en una subclase. Si la subclase no lo implementa, también deberá marcarse como `abstract`. No se pueden crear objetos de una clase marcada como abstracta.

El modificador `abstract` sólo se puede usar en métodos normales, no se puede aplicar ni en métodos estáticos ni en constructores.

Una clase definida como `abstract` puede tener variables y métodos normales con implementación como cualquier otra clase.

```
abstract class Vehicle {
  void show();
}

class Car extends Vehicle {
  @Override
  void show() {}
}
```

Modificador `final`

Para evitar que un método se reemplace, se especifica `final` como modificador al inicio de su declaración. También se puede evitar que una clase se herede si se precede su declaración como `final`. De esta forma, todos sus métodos son final de forma implícita.

Los modificadores `abstract` y `final` son incompatibles ya que una clase `abstract` debe ser heredada para proporcionar una implementación completa y el modificador `final` no permite la herencia.

Una variable miembro con el modificador `final` es como una constante ya que el valor inicial asignado no se puede cambiar mientras dure el programa.

```
final class Vehicle {}

class SuperCar {
  final int MIN_POWER = 545; // Este valor no cambia mientras dure el programa

  void show() {}
  final void price() {}
}

// class Moto extends Vehicle {} // Una clase final no puede ser heredada
```

```
class Car extends SuperCar {
  @Override
  void show() {} // Correcto

  void price() {} // Incorrecto. No se puede sobreescribir un método 'final'
}
```

Visibilidad

Visibilidad de clases

Visibilidad permitidas para las clases:

- `default` (sin modificador) -> Sólo será visible por otras clases **dentro del mismo paquete**.
- `public` -> Una clase pública es **visible desde cualquier lugar**.

> ⚠ Una clase declarada como `public` debe encontrarse en **un archivo con el mismo nombre**.

```
class Vehicle {} // clase 'default' (sin modificador)
public class Car {} // clase 'public' y en un fichero con el nombre 'Car.java'
```

Visibilidad de una interfaz

Visibilidad permitida para las interfaces:

- `default` (sin modificador) -> Una interfaz sin modificador sólo será visible por otras clases o interfaces **dentro del mismo paquete**.
- `public` -> Una interfaz pública es visible **desde cualquier lugar**.

> ⚠ Una interfaz declarada como `public` debe encontrarse en **un archivo con el mismo nombre**.

```
interface Vehicle {} // interfaz 'default' (sin modificador)
public interface Car {} //  interfaz 'public' y en un fichero con el nombre 'Car.java'
```

Visibilidad de variables y miembros de instancia

	Private	Default	Protected	Public
Visible desde la misma clase	Sí	Sí	Sí	Sí
Visible desde el mismo paquete por una subclase	No	Sí	Sí	Sí
Visible desde el mismo paquete por una no subclase	No	Sí	Sí	Sí
Visible desde un paquete diferente por una subclase	No	No	Sí	Sí
Visible desde un paquete diferente por una no subclase	No	No	No	Sí

Interfaces

Las interfaces son sintácticamente similares a las clases abstractas con la diferencia que **en una interfaz todos los métodos carecen de cuerpo**. Una clase puede implementar todas las interfaces que desee pero tiene que implementar todos los

métodos descritos en la interfaz. Por tanto, el código que conozca la interfaz puede usar objetos de cualquier clase que implemente dicha interfaz. Si una clase no implementa todos los métodos de una interfaz deberá declarase como `abstract`.

Antes de JDK 8 una interfaz no podía definir ninguna implementación pero a partir de JDK 8 se puede añadir una implementación predeterminada a un método de interfaz. La clase o clases que implementen la interfaz podrán **definir su propia implementación o usar la predeterminada**. Un método predeterminado se precede con la palabra clave `default`. Ahora también admite métodos estáticos y, a partir de JDK 9, una interfaz puede incluir métodos `private`.

Una interfaz puede ser `public` (y en un fichero del mismo nombre) o `default` (sin modificador). Los métodos son implícitamente `public` y las variables declaradas en un interfaz no son variables de instancia, sino que son `public`, `final` y `static` y deben inicializarse. Por tanto son constantes.

Cuando una clase implementa varias interfaces, éstas se separan mediante comas. En caso de que una clase implemente una interfaz y que herede de una clase primero se coloca `extends` y luego `implements`.

> ❗ Como hemos dicho en una interfaz los métodos son implícitamente `public`. Cuando una clase implementa dicha interfaz y codifica los métodos de la interfaz, si no indica visibilidad los miembros de la clase son `default` de forma implícita, lo cual genera un error ya que `default` es más restrictivo que `public`. Por tanto, **tenemos que indicar explícitamente como `public` los métodos implementados en la clase**.

```
(public) interface Vehicle {
  public static final String UNITS = "Km/h";

  // Método implícitamente 'public' que será codificado por la clase/s que implementan la interfaz
  void getWheels();

  // Método con una implementación por defecto.
  (public) default boolean start() {
    return true;
  }
}

// Si una clase implementa varias interfaces, estas se separan mediante comas.
class Car extends Superclass implements Vehicle {
    public void getWheels() {} // Es necesario indicar 'public' o se genera un error.
}

class Sample {
    public static void main (String ... args) {
        //Se declara una variable de referencia de un tipo de interfaz.
        Vehicle car = new Car();

        // Se ejecutará la versión implementada por el objeto.
        // Sólo se tiene acceso a los métodos definidos en la interfaz y
        // no a otros métodos que puedan estar definidos en la clase.
        car.getWheels();
    }
}
```

Una interfaz puede heredar a otra interfaz por medio de la palabra reservada `extends`. Cuando una clase implementa una interfaz que hereda de otra interfaz debe proporcionar implementaciones de todos los métodos definidos en la cadena de herencia.

```
interface Vehicle {
  int getWheels();
}

interface Car extends Vehicle {
  int getPassengers();
}
```

```
class MyCar implements Car {
  public int getWheels() { return 4; } // se implementan los métodos de ambas interfaces
  public int getPassengers() { return 5; }
}
```

La inclusión de los métodos predeterminados no varia un aspecto clave de los interfaces, y es que no admiten variables de instancia. Por tanto sigue habiendo una diferencia entre interfaces y una clase normal o abstracta. **Una clase puede mantener información de estado mediante sus variables de instancia mientras que una interfaz no puede**. Por tanto una interfaz sigue siendo útil para definir lo que debe hacer una clase y no como lo debe hacer.

Si una clase hereda de dos interfaces que implementan un método predeterminado con el mismo nombre, la clase está **obligada** a implementar dicho método ya que si no lo hace el compilador genera un error. La versión implementada en la clase tiene preferencia sobre las versiones implementadas en las interfaces.

JDK 8 añade a las interfaces la capacidad de tener uno o varios métodos estáticos. Como sucede con una clase, un método estático definido por una interfaz se puede invocar de forma independiente a cualquier objeto.

```
interface Vehicle {
  static void start() { System.out.println("Starting..."); }
}

public class Sample {
  public static void main (String ... args) {
    Vehicle.start();
  }
}
```

A partir de JDK 9 una interfaz puede incluir un método `private` que solo puede invocarse mediante un método predeterminado u otro método `private` definido por la misma interfaz. Dado que es `private` este código no puede usarse fuera de la interfaz en la que esté definido.

Excepciones

Una excepción es **un error producido en tiempo de ejecución**. En Java, todas las excepciones se representan por medio de clases. Todas las clases de excepción se derivan de `Throwable`. Esta clase tiene dos subclases directas: `Exception` y `Error`.

Las excepciones tipo `Error` son errores producidos en la propia máquina virtual y no se deben controlar. Los programas sólo deben controlar aquellas excepciones de tipo `Exception`.

Mediante la palabra reservada `throw` se pueden lanzar manualmente una excepción.

Las excepciones se tratan en un bloque `try-catch-finally` (`finally` es opcional):

```
try {
  // bloque de código que puede lanzar la excepción
} catch (TipoException exception) {
  // bloque de código para TipoException
} catch (Tipo2Exception exception) {
  // bloque de código para Tipo2Exception
} catch (Exception exception) {
  // Captura del resto de excepciones no capturadas anteriormente
} finally {
  // Código que se ejecutará siempre, tanto si
  // se produce una excepción como si no se produce.
}
```

Si un método genera una excepción que no se va a controlar, debemos declarar dicha excepción en una cláusula `throws`. Con esta cláusula podemos 'relanzar' tanto excepciones de Java como excepciones personalizadas. Una vez lanzada esta excepción deberá ser capturada en un bloque `try-catch` superior o por la JVM:

```
int divide(int a, int b) throws ArithmeticException, MyException {
  if (b == 0) {
    throw new ArithmeticException();
  } else {
    throw new MyException("Message");
  }
}

class MyException extends Exception { }
```

En **JDK 7** se amplió el mecanismo de excepciones al permitir la **captura múltiple**. Con la captura múltiple se permite la captura de dos o más excepciones dentro de la misma cláusula `catch`. Cada tipo de excepción de la lista se separa con el operador `|` (`'OR'`). Cada parámetro es `final` de forma implícita.

```
try {
  // código
} catch (final ArithmeticException | ArrayIndexOutOfBoundsException e) {
  // Controlador
}
```

En **JDK 7** se añadió otro mecanismo denominado `try-with-resources` o ***try con administración automática de recursos***. Es un tipo de `try` que evita situaciones en que un archivo (u otro recurso como bases de datos, etc..) no se libera después de ser utilizado. Un `try-with-resources` de este tipo también puede incluir cláusulas `catch` o `finally`.

Los recursos que se pueden emplear con este tipo de `try-with-resources` son recursos que implementen la interfaz `AutoCloseable` que a su vez hereda de `Closeable`. La interfaz `AutoCloseable` define el método `close()`. Además, el recurso declarado en la instrucción `try` es **'final'** de forma implícita, de forma que no puede ser asignado ni modificado una vez creado y su ámbito se limita al propio `try`.

```
/*
 - El siguiente código usa un 'try con recursos' para abrir un archivo
 - y después cerrarlo automáticamente al salir del bloque 'try'.
 - Por tanto ya no es necesario invocar a 'close()'
 */
try (FileInputStream fin = New FileInputStream(args[0])) {
  // bloque de código
} catch (IOException e) {
  // Controlador
}
```

Se pueden gestionar más de un recurso que estarán separados por un punto y coma ';':

```
try (FileInputStream fin = New FileInputStream(args[0]); FileOutputStream fout = New FileOutputStream(args[1])) {
  // bloque de código
} catch (IOException e) {
  // Controlador
}
```

Entrada/Salida (E/S)

En Java el sistema E/S se define en dos sistemas completos: uno para **E/S de bytes** y otro para **E/S de caracteres**. En el nivel inferior toda la E/S sigue orientada a bytes. La E/S de caracteres es una especialización y una forma más cómoda de trabajar con caracteres.

Los programas en Java realizan la E/S a través de **flujos** (*'streams'*).

Todos los programas de Java importan automáticamente el paquete `java.lang` que define la clase `System`. Esta clase contiene, entre otros elementos, tres variables de flujo predefinidos:

- `System.in` - hace referencia al flujo estándar de entrada, que es el teclado.
- `System.out` - hace referencia al flujo estándar de salida, que es la consola.
- `System.err` - hace referencia al flujo de error estándar que también es la consola de forma predeterminada.

Flujos de bytes

Los flujos de bytes se definen en dos jerarquías de clases. En la parte superior hay dos clases abstractas que definen las características comunes: `InputStream` y `OutputStream`.

A partir de estas clases se crean subclases concretas con distinta funcionalidad:

- InputStream:
  - `BufferedInputStream` - Flujo de entrada en búfer
  - `ByteArrayInputStream` - Flujo de entrada desde una matriz de bytes
  - `DataInputStream` - Flujo de entrada que contiene métodos para leer los tipos de datos estándar de Java
  - `FileInputStream` - Flujo de entrada que lee desde un archivo
  - `FilterInputStream` - Implementa 'InputStream'
  - `ObjectInputStream` - Flujo de entrada de objetos
  - `PipedInputStream` - Conducción de entrada
  - `PushbackInputStream` - Flujo de entrada que permite devolver bytes al flujo
  - `SequenceInputStream` - Flujo de entrada que combina dos o más flujos de entrada que se leen secuencialmente, uno tras otro
- OutputStream:
  - `BufferedOutputStream` - Flujo de salida en búfer
  - `ByteArrayOutputStream` - Flujo de salida que escribe en una matriz de bytes
  - `DataOutputStream` - Flujo de salida que contiene métodos para escribir los tipos de datos estándar de Java
  - `FileOutputStream` - Flujo de salida que escribe en un archivo
  - `FilterOutputStream` - Implementa 'OutputStream'
  - `ObjectOutputStream` - Flujo de salida para objetos
  - `PipedOutputStream` - Conducción de salida
  - `PrintStream` - Flujo de salida que contiene `print()` y `println()`

Leer entradas de consola

Aunque usar el flujo de caracteres para leer de consola es preferible debido a la internacionalización y al mantenimiento de programas, la lectura de flujo de bytes sigue siendo usado:

```
// Leer una matriz de bytes desde el teclado
import java.io.*;

class ReadBytes {
    public static void main(String args[]) throws IOException {
        byte[] data = new byte[10];
        System.out.println("Enter some characters:");
        System.in.read(data); // leer una matriz de bytes desde el teclado
```

```
        System.out.print("You entered: ");
        for(int i = 0; i < data.length; i++) {
            System.out.print((char)data[i]);
        }
    }
}
```

Escribir la salida en la consola con `PrintStream`

Para escribir en consola se utiliza `print()` o `println()` que se definen en `PrintStream` aunque también tiene métodos como `write()`

```
class WriteDemo {
    public static void main(String args[]) {
        int b = 'X';
        System.out.write(b); // Escribir un byte en la pantalla
        System.out.write('\n');
    }
}
```

Leer archivos con `FileInputStream`

```
/* Display a text file.
 To use this program, specify the name
 of the file that you want to see.
 For example, to see a file called TEST.TXT,
 use the following command line.
 java ShowFile TEST.TXT
*/
import java.io.*;

class ShowFile {
    public static void main(String args[]) {
        int i;
        FileInputStream fin = null;
        try {
            fin = new FileInputStream(args[0]);
            do {
                i = fin.read();
                if (i != -1)
                    System.out.print((char) i);
            } while (i != -1);
        } catch (IOException e) {
            System.out.println("I/O Error: " + e);
        } finally {
            // Close file in all cases.
            try {
                if (fin != null)
                    fin.close();
            } catch (IOException e) {
                System.out.println("Error Closing File");
            }
        }
    }
}
```

Escribir archivos con `FileOutputStream`

```
/* Copy a file.
 To use this program, specify the name
 of the source file and the destination file.
```

```
 For example, to copy a file called FIRST.TXT
 to a file called SECOND.TXT, use the following
 command line.
 java CopyFile FIRST.TXT SECOND.TXT
*/
import java.io.*;

class CopyFile {
    public static void main(String args[]) throws IOException {
        int i;
        FileInputStream fin = null;
        FileOutputStream fout = null;
        // First, confirm that both files have been specified.
        if (args.length != 2) {
            System.out.println("Usage: CopyFile from to");
            return;
        }
        // Copy a File.
        try {
            // Attempt to open the files.
            fin = new FileInputStream(args[0]);
            fout = new FileOutputStream(args[1]);
            do {
                i = fin.read();
                if (i != -1)
                    fout.write(i);
            } while (i != -1);
        } catch (IOException e) {
            System.out.println("I/O Error: " + e);
        } finally {
            try {
                if (fin != null)
                    fin.close();
            } catch (IOException e2) {
                System.out.println("Error Closing Input File");
            }
            try {
                if (fout != null)
                    fout.close();
            } catch (IOException e2) {
                System.out.println("Error Closing Output File");
            }
        }
    }
}
```

Flujos de caracteres

Los flujos de caracteres se definen en dos jerarquías de clases. En la parte superior hay dos clases abstractas que definen las características comunes: `Reader` y `Writer`. Las clases concretas derivadas de estas clases operan en flujos de caracteres *Unicode*.

A partir de estas clases se crean subclases concretas con distinta funcionalidad:

- Reader:
    - `BufferedReader` - Flujo de caracteres entrada en búfer
    - `CharArrayReader` - Flujo de entrada que lee desde una matriz de caracteres
    - `FileReader` - Flujo de entrada que lee desde un archivo
    - `FilterReader` - Lector filtrado
    - `InputStreamReader` - Flujo de entrada que traduce bytes en caracteres
    - `LineNumberReader` - Flujo de entrada que cuenta líneas
    - `PipedReader` - Conducción de entrada
    - `PushbackReader` - Flujo de caracteres que permite devolver caracteres al flujo de entrada

- `StringReader` - Flujo de entrada que lee desde una cadena

- Writer:

  - `BufferedWriter` - Flujo de caracteres de salida en búfer
  - `CharArrayWriter` - Flujo de salida que escribe en una matriz de caracteres
  - `FileWriter` - Flujo de salida que escribe en un archivo
  - `FilterWriter` - Escritor filtrado
  - `OutputStreamWriter` - Flujo de salida que traduce caracteres en bytes
  - `PipedWriter` - Conducción de salida
  - `PrintWriter` - Flujo de salida que contiene `print()` y `println()`
  - `StringWriter` - Flujo de salida que escribe en una cadena

Leer caracteres desde la consola con `BufferedReader`

Como `System.in` es un flujo de bytes, se convierte en flujo de caracteres mediante un `InputStreamReader`. Este `InputStreamReader` se pasa a `BufferedReader`, que es una clase óptima que admite un flujo de entrada en búfer.

```
// Use a 'BufferedReader' to read characters from the console.
import java.io.*;

class BRRead {
    public static void main(String args[]) throws IOException {
        char c;
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter characters, 'q' to quit.");
        // read characters
        do {
            c = (char) br.read();
            System.out.println(c);
        } while (c != 'q');
    }
}
```

Leer cadenas desde la consola con `BufferedReader`

```
// Read a string from console using a 'BufferedReader'.
import java.io.*;

class BRReadLines {
    public static void main(String args[]) throws IOException {
        // create a 'BufferedReader' using 'System.in'
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str;
        System.out.println("Enter lines of text.");
        System.out.println("Enter 'stop' to quit.");
        do {
            str = br.readLine();
            System.out.println(str);
        } while (!str.equals("stop"));
    }
}
```

Salida en consola con `PrintWriter`

Aunque para programas pequeños y tareas de depuración se puede utilizar `System.out`, en programas reales es recomendable usar un flujo `PrintWriter`:

```java
// Demonstrate 'PrintWriter'
import java.io.*;

public class PrintWriterDemo {
    public static void main(String args[]) {
        PrintWriter pw = new PrintWriter(System.out, true);
        pw.println("This is a string");
        int i = -7;
        pw.println(i);
        double d = 4.5e-7;
        pw.println(d);
    }
}
```

Escribir en un fichero con `FileWriter`

```java
// Demonstrate 'FileWriter'.
// This program uses 'try-with-resources'. It requires JDK 7 or later.
import java.io.*;

class FileWriterDemo {
    public static void main(String args[]) throws IOException {
        String source = "Now is the time for all good men\n" + " to come to the aid of their country\n"
                + " and pay their due taxes.";
        char buffer[] = new char[source.length()];
        source.getChars(0, source.length(), buffer, 0);
        try (FileWriter f0 = new FileWriter("file1.txt");
                FileWriter f1 = new FileWriter("file2.txt");
                FileWriter f2 = new FileWriter("file3.txt")) {
            // write to first file
            for (int i = 0; i < buffer.length; i += 2) {
                f0.write(buffer[i]);
            }
            // write to second file
            f1.write(buffer);
            // write to third file
            f2.write(buffer, buffer.length - buffer.length / 4, buffer.length / 4);
        } catch (IOException e) {
            System.out.println("An I/O Error Occurred");
        }
    }
}
```

Leer de un fichero con `FileReader`

```java
// Demonstrate 'FileReader'.
// This program uses 'try-with-resources'. It requires JDK 7 or later.
import java.io.*;

class FileReaderDemo {
    public static void main(String args[]) {
        try (FileReader fr = new FileReader("FileReaderDemo.java")) {
            int c;
            // Read and display the file.
            while ((c = fr.read()) != -1)
                System.out.print((char) c);
        } catch (IOException e) {
            System.out.println("I/O Error: " + e);
        }
    }
}
```

Manipular ficheros y directorios

Con el paquete `java.io` se trabaja con *streams*, leyendo y escribiendo ficheros a bajo nivel.

Existe otra forma de manipular ficheros a más alto nivel gracias a las utilidades del paquete `java.nio`.

El primer paso es obtener una referencia al fichero o directorio mediante la clase `Path`:

```
// Obtener una referencia mediante la clase de ayuda 'java.nio.file.Paths'
Path path = Paths.get("/home/test.txt"); // Sistemas UNIX

Path path2 = Paths.get("c:\\home\\test.txt"); // Sistemas Windows
```

Para operar con ficheros o directorios es necesario utilizar la clase `Files`:

```
// Verificar si existe el fichero
Path path = Paths.get("/home/test.txt");

Files.exists(path); // Devuelve 'true' si existe

// Verificar su accesibilidad
Files.isReadable(path);
Files.isWritable(path);
Files.isExecutable(path);

// Copiar ficheros (o directorios)
Files.copy(path, Paths.get("copy.txt"));

// Borrar fichero (o directorio, que deberán estar VACÍOS)
Files.delete(Paths.get("copy.txt"));
```

Trabajar con fechas

Hasta Java 8, para trabajar con fechas se utilizaban las clases `Date` y `GregorianCalendar`.

Las nuevas herramientas para trabajar con fechas a partir de Java 8 se encuentran en el paquete `java.time`:

```
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

LocalDate fechaActual = LocalDate.now();
System.out.println(fechaActual);  // Imprime '2024-01-19'

DateTimeFormatter dtF1 = DateTimeFormatter.ofPattern("dd-MM-yyyy");
System.out.println(dtF1.format(fechaActual)); // Imprime '19-01-2024'
```

Programación de subprocesamiento múltiple

Existen dos tipos de multitarea: la basada en **procesos** y la basada en **subprocesos**.

Un proceso es básicamente un programa que se ejecuta. Por tanto la multitarea basada en procesos permite al equipo ejecutar dos o más programas a la vez. En un entorno multitarea basado en subprocesos, el subproceso es la unidad de código menor que se entrega, lo que significa que un mismo programa puede realizar dos o más tareas al mismo tiempo.

Java no controla la multitarea basada en procesos pero **sí controla la basada en subprocesos**.

Una ventaja del subprocesamiento múltiple es que permite programas más eficaces ya que se utiliza el tiempo de inactividad en la mayoría de programas. En sistemas de un sólo núcleo, los subprocesos de ejecución simultánea comparten la CPU y cada subproceso recibe una porción de tiempo de CPU. En sistemas multinúcleo, dos o más subprocesos se pueden ejecutar simultáneamente.

Un subproceso puede estar en varios estados, como por ejemplo en ejecución o puede estar bloqueado a la espera de un recurso, etc…

Junto a la multitarea basada en subprocesos surge la necesidad de una función especial denominada **sincronización**, que permite coordinar la ejecución de subprocesos de determinadas formas.

El sistema de subprocesamiento múltiple de Java se base en la clase `Thread` y en su interfaz `Runnable`, ambas de `java.lang`. Para crear un nuevo subproceso, su programa debe ampliar `Thread` o implemetar la interfaz `Runnable`.

```
// Create a second thread.
class NewThread implements Runnable {
    Thread thread;

    NewThread() {
        // Create a new, second thread
        thread = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + thread);
        thread.start(); // Start the thread
    }

    // This is the entry point for the second thread.
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ThreadDemo {
    public static void main(String args[]) {
        new NewThread(); // create a new thread
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

```
// Controlling the main Thread.
class CurrentThreadDemo {
    public static void main(String args[]) {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
        // change the name of the thread
        t.setName("My Thread");
        System.out.println("After name change: " + t);
```

```
        try {
            for (int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
}
```

`Thread` ofrece dos formas para saber si un subproceso ha finalizado. Por un lado se puede invocar `isAlive()` en el subproceso, que devolverá 'true' si el subproceso en el que se invoca sigue en ejecución.

```
do {
    // code
} while (thread.isAlive())
```

Otra forma de esperar a que un subproceso termine consiste en invocar `join()`. Este método espera a que termine el subproceso en el que se invoca. Su nombre proviene del concepto del subproceso invocador esperando a que se le una el subproceso especificado.

Métodos sincronizados

Al usar varios subprocesos en ocasiones será necesario sincronizar las actividades de los subprocesos para que no accedan a la vez a un mismo recurso. Esto se consigue con la palabra clave `synchronized`.

Al invocar un método sincronizado, el subproceso invocador accede al monitor del objeto, que lo bloquea. Mientras está bloqueado, ningún otro subproceso puede acceder al método ni a otro método sincronizado definido por la clase del objeto.

```
class SumArray {
    private int sum;

    /* Este método está sincronizado.
    Cuando sea invocado por un subproceso quedará bloqueado al resto de subprocesos,
    que deberán esperar a que sea desbloqueado.
    No podrán acceder ni a éste ni a ningún otro método sincronizado de esta clase */
    synchronized int sumArray(int nums[]) {
        // code....
    }
}
```

Bloque sincronizado

No sólo se puede sincronizar métodos si no que Java proporciona un **bloque sincronizado**. Tras entrar en un bloque `synchronized`, ningún otro subproceso puede invocar un método sincronizado en el objeto al que hace referencia la variable pasada como parámetro hasta que se salga del bloque.

```
synchronized(refObj) { // 'refObj' es una referencia al objeto sincronizado
    // instrucciones que sincronizar
}
```

```
// This program uses a synchronized block.
class Callme {
    void call(String msg) {
```

```
        System.out.print("[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    // synchronize calls to call()
    public void run() {
        synchronized (target) { // synchronized block
            target.call(msg);
        }
    }
}

class Sample {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");
        // wait for threads to end
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}
```

Enumeraciones

Básicamente, una enumeración es una **lista de constantes con nombre** que definen un nuevo tipo de datos. Un objeto de un tipo de enumeración solo puede albergar los valores definidos por la lista. Por tanto, una enumeración le permite definir con precisión un nuevo tipo de datos con un número fijo de valores.

Desde una perspectiva de programación, las enumeraciones son muy útiles cuando hay que definir un grupo de valores que representan una colección de elementos. Es importante entender que una **constante de enumeración es un objeto de su tipo de enumeración**. Una enumeración se crea con la palabra clave `enum`.

Las constantes de la enumeración son `public` y `static` de forma implícita.

```
enum Transport {
    CAR, TRUCK, AIRPLANE, TRAIN, BOAT // constantes de enumeración
}
```

Estas constantes tienen el tipo de la enumeración que las contiene. Una vez definida la enumeración para crear una variable de este tipo no usamos `new` como una clase sino que se declaran y usan las enumeraciones como si fueran tipos primitivos.

Sin embargo **Java implementa las enumeraciones como si fueran clases**, permitiendo que tengan constructores, métodos, etc.. aunque con dos limitaciones que las diferencia del resto de clases en Java:

- Una enumeración no puede heredar de otra clase.
- Una enumeración no puede actuar como superclase de otra clase.

```
// Las constantes, al ser 'static' se invocan de esta forma: 'Enumeration.constante'
Transport transport = Transport.TRUCK;

if (transport == Transport.TRUCK) { // Comparar la igualdad de dos constantes de enumeración
    System.out.println(transport) // => TRUCK
}

//Podemos usar una enumeración para controlar una instrucción 'switch'
switch (transport) {
    // No es necesario usar 'Transport.CAR' ya que implícitamente ya se especifica
    case CAR:
    // code ....
    break;
    case TRUCK:
    // code ....
    break;
    default:
    // code...
    break;
}
```

Las enumeraciones cuentan con dos métodos predefinidos `values()` y `valueOf()` cuyo formato es:

- `public static tipo-enum[] values()` ➜ devuelve un array que contiene una lista de las constantes de enumeración
- `public static tipo-enum valueOf(String cadena)` ➜ devuelve las constantes de enumeración cuyo valor se corresponde a la cadena pasada como argumento.

```
// Uso de values() en un for-each
for (Transport transport : Transport.values()) {
    System.out.println(transport);
}
```

Al definir un constructor en una enumeración, el constructor se invoca al crear cada una de las constantes de enumeración. Cada constante puede invocar todos los métodos definidos por la enumeración. Cada constante dispone de su propia copia de las variables de instancia definidas por la enumeración.

```
enum Transport {
    /* Valores de inicialización.
    A destacar el ';' necesario cuando se definen variables, constructores, etc.. */
    CAR(66), TRUCK(12), AIRPLANE(600), BOAT(12);

    private int speed; // variable de instancia. Cada constante dispone de su propia copia

    Transport(int s) { // constructor. Es invocado por cada constante
        speed = s;
    }

    int getSpeed() { // método de instancia. Se invocaría con Transport.CAR.getSpeed();
        return speed;
```

```
    }
}
```

Las enumeraciones tienen un método llamado `ordinal()` que devuelve un valor que indica la posición de la constante dentro de la enumeración. Los valores ordinales empiezan en 0:

```
enum Transport {
    CAR, AIRPLANE, TRUCK, BOAT
}

System.out.println(Transport.TRUCK.ordinal()); // Print 3
```

Autoboxing y unboxing

En Java los tipos primitivos no forman parte de la jerarquía de objetos por motivos de eficiencia. Sin embargo existen clases que actuan como envoltorios (*'wrapper'*) para tipos primitivos como `Float`, `Double`, `Byte`, `Short`, `Integer`, `Long`, `Character` y `Boolean`.

Todos los envoltorios de tipos numéricos heredan de la clase abstracta `Number`.

Encapsular un tipo primitivo en su envoltorio se denomina **'boxing'**. Por tanto **'autoboxing'** es el proceso de encapsular automáticamente un tipo primitivo en su clase envoltorio y **'auto-unboxing'** es el proceso inverso.

```
Integer num = Integer.valueOf(100) // sin 'autoboxing'

Integer iOb = 100; // 'autobox' de int a Integer

int i = iOb; // unbox
```

Genéricos

El término **"genérico"** significa tipo con parámetros. Los tipos con parámetros permiten crear clases, interfaces y métodos en los que los tipos de datos se especifican como parámetros. Cuando una clase utiliza genéricos se denomina **"clase genérica"**.

```
/* Uso de genéricos en una clase.
'T' es un parámetro de tipo que se sustituye por un tipo real al crear un objeto de la clase */
class Gen<T> {
  T ob; // Declarar un objeto de tipo 'T'.

  Gen(T o) { // Pasar al constructor una referencia a un objeto de tipo 'T'
    ob = o;
  }

  T getOb() { // retorna 'ob' de tipo 'T'
    return ob;
  }

  void showType() {
    System.out.println("Type of T is " + ob.getClass().getName());
  }
}

class GenDemo {
  public static void main(String ... args) {
    Gen<Integer> iOb; // Crear una referencia

    // Crear un objeto Gen<Integer> y asignar la referencia a 'iOb'.
```

```
    // Uso de autoboxing para encapsular el valor entero en un objeto 'Integer'
    iOb = new Gen<Integer>(80);
    iOb.showType();

    /* Esta asignación generaría un error en tiempo de compilación.
    Es una de la ventajas del uso de genéricos */
    // iOB = new Gen<Double>(88.0) // Error

    Gen<String> strOb = new Gen<String>("Generic");
    strOb.showType();
  }
}
```

El compilador no crea diferentes versiones de la clase genérica en función del tipo pasado sino que usa la misma versión. Lo que hace es sustituir el genérico por el tipo real y realiza las conversiones necesarias para que el código se comporte como si hubiera sido escrito con ese tipo.

Al declarar una instancia de un tipo genérico, el argumento de tipo pasado al parámetro de tipo debe ser un tipo de referencia. No se puede usar un tipo primitivo como `int` o `char`.

Destacar sobre los tipos genéricos es que una referencia a una versión concreta de un tipo genérico no es compatible en cuanto a tipo se refiere con otra versión del mismo tipo genérico.

```
/* No se puede asignar una referencia de Gen<String> a una referencia Gen<Integer>
aunque ambas usen la misma clase genérica Get<T> */
iOb = strOb; // Error
```

Se puede declarar más de un parámetro de tipo en un tipo genérico. Basta con usar una lista separada por comas:

```
class Gen<T, V> {
    T ob1;
    V ob2;

    Gen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }
}

// Podemos usar tipos diferentes (<Integer, String>) o tipos iguales (<Integer, Integer>)
Gen<Integer, String> sample = new Gen<Integer, String>(0, "");
Gen<Integer, Integer> sample1 = new Gen<Integer, Integer>(0, 0);
```

Tipos vinculados (o limitados)

Java ofrece los **'tipos vinculados'** que permite, al especificar un parámetro de tipo, crear un vínculo superior que declare la superclase de la que deben derivarse todos los argumentos de tipo. Por ejemplo, podemos limitar los parámetros de tipo a únicamente tipos numéricos, evitando que pasemos parámetros de tipo `String`.

Para ello usamos la cláusula `extends` al especificar los parámetros de tipo:

- `<T extends superclass>`

Esto especifica que 'T' solo se puede reemplazar por *'superclass'* o subclases de *'superclass'*. Por tanto *'superclass'* define un **límite superior e inclusivo**.

> ⚠️ Todos los tipos numéricos heredan de la clase abstracta `Number`.

```
class GenNumeric<T extends Number> { // De esta forma limitamos 'T' a tipos numéricos
    T num;

    GenNumeric(T n) {
        num = n;
    }

    double fraction() {
        // Como hemos limitado el tipo a tipos numéricos podemos emplear métodos de la clase 'Number'
        return num.doubleValue() - num.intValue();
    }
}
```

Los tipos vinculados resultan especialmente útiles para garantizar que un parámetro sea compatible con otro:

```
class Pair<T, V extends T> { // 'V' debe tener el mismo tipo que 'T' o ser una subclase de 'T'
    // code ....
}

Pair<Integer, Integer> x = new Pair<Integer, Integer>(); // Correcto
Pair<Number, Integer> y = new Pair<Number, Integer>(); // Correcto, Integer es una subclase de Number
Pair<Integer, String> z = new Pair<Integer, String>(); // ¡¡INCORRECTO!!, String no es una subclase de Integer
```

Argumentos comodín

Un argumento comodín se representa mediante '?' y representa un tipo desconocido. Destacar que el comodín '?' no afecta al tipo de parámetros. La limitación se crea con la cláusula `extends`. El comodín simplemente equivale a cualquier tipo válido. Por ejemplo, `<T extends Number>` limita a tipos numéricos y por tanto el comodín equivale a utilizar cualquier tipo numérico válido. La limitación a tipos numéricos se ha creado con la cláusula `extends`.

```
class Gen<T extends Number> {
    T num;
    // code...
}

class Sample {
    boolean absEqual(Gen<?> a, Gen<?> b) {
        return Math.abs(a.num.doubleValue()) == Math.abs(b.num.intValue());
    }
}
```

Los argumentos comodín se pueden vincular con cualquier parámetro de tipo. Un comodín vinculado es especialmente importante para crear un método que solo deba operar en objetos que sean subclases de una superclase concreta. Se especifica con la forma:

- `<? extends superclase>`

```
void sample(Gen<? extends Number> a) {  // Tipos que sean 'Number' o subclases de 'Number'
    // code...
}
```

Se puede especificar un límite inferior con la forma `<? super subclase>`. En este caso es un **límite no inclusivo**. Por tanto '?' equivale a superclases de subclase pero no incluye a la propia subclase.

Métodos genéricos

Los métodos de una clase genérica pueden usar el parámetro del tipo de una clase y por tanto son genéricos de forma automática. Sin embargo también podemos declarar métodos genéricos dentro de clases no genéricas.

Los parámetros de tipo en un método se declaran antes que el tipo devuelto del método. Los métodos genéricos puede ser estáticos o no estáticos.

```
class Sample {
    static <T> void sample(T x) { /* code... */ }
    <T, V> boolean sample(T x, V y) { /* code... */ }
    <T, V extends T> int sample(T x, V y) { /* code... */ }
    static <T extends Comparable<T>, V extends T> boolean sample(T x, V y) { /* code... */ }
}
```

Un constructor puede ser genérico aunque su clase no lo sea:

```
class Sample {
    // variables de instancia

    <T extends Number> Sample(T arg) { // Constructor genérico
        // code...
    }
}
```

Interfaces genéricas

Las interfaces genéricas se especifican como una clase genérica. Cualquier clase que implemente una interfaz genérica también debe ser una clase genérica. Si se especifica el tipo entonces no es necesario que sea genérica.

Los parámetros de tipo especificado en una interfaz también se pueden vincular (limitar) con los tipos vinculados. Las clases que implementen dicha interfaz deberán pasarle un argumento de tipo que tenga la misma vinculación.

```
// Interfaz genérica
interface ISample<T> {
    boolean contains(T arg);
}

// Interfaz genérica con tipos vinculados (limitados) por la superclase 'Number'
interface ISample2<T extends Number> {
    // ...
}

// Clase genérica obligada que implementa una interfaz genérica
class Sample<T> implements ISample<T> {
    // ...
}

// Clase no necesariamente genérica que implementa una interfaz con un tipo concreto
class Sample implements ISample<Double> {
    // ...
}

// Clase con parámetros de tipo vinculados
class Sample2<T extends Number> implements ISample2<T> {
    // ....
}

/* No es necesario volver a indicarla en ISample2 */
// class Sample2<T extends Number> implements ISample2<T extends Number> {} // ¡¡INCORRECTO!!.
```

Genéricos y código legado

Antes de la JDK 5 no existían los genéricos. Por tanto, para asegurar la compatibilidad con código legado Java permite usar una clase génerica sin argumetos de tipo. En estos casos se convierte en un tipo sin procesar.

```
class Gen<T> {
    // ...
}

Gen<Integer> iOb = new Gen<Integer>(0); // Objeto 'Gen' para enteros
Gen<String> strOb = new Gen<String>(""); // Objeto 'Gen' para Strings
Gen legacyOb = new Gen(new Double(0.0)); // Objeto 'legacy' con tipo sin procesar

// Dado que el compilador desconoce el tipo sin procesar se producen situaciones potencialmente erróneas
strOb = legacyOb; // Asignación que no produce error de compilación pero insegura
//iOb = strOb; // Asignación errónea que detecta el compilador
```

Inferencia de tipos

A partir de la JDK 7 es posible reducir la sintaxis a la hora de crear una instancia de un tipo genérico. Para la creación de una instancia se emplea una lista vacía de argumentos (`<>`) que indica al compilador que infiera los argumentos de tipo que necesita el constructor. En caso de que sea necesario mantener la compatibilidad con código legado se puede emplear la forma completa:

```
class Gent<T, V> {
    // code...
}

Gen<Integer, Integer> iOb = new Gen<Integer, Integer>(); // Forma completa
Gen<Integer, Integer> iOb = new Gent<>(); // Sintaxis reducida para la JDK 7 y posteriores
```

Restricciones y ambigüedad

El uso de genéricos puede crear situaciones de ambigüedad, sobretodo en casos de sobrecarga de métodos:

```
class Gen<T, V> {
    /* Estos dos métodos se sobrecargan pero dado que T y V pueden ser del mismo tipo, se generarían
    dos métodos iguales por lo que el compilador genera un error y este código no compila. */
    void get(T ob) {}

    void get(V ob) {}
}
```

Una restricción importante es que los parámetros de tipo no se pueden utilizar como si fueran tipos normales ni tampoco declarar variables estáticas de parámetros de tipo:

```
class Gen<T, V> {
    T ob;
    static V ob; // ¡¡INCORRECTO!!, no hay variables estáticas de 'T'

    void sample() {
        ob = new T(); // ¡¡INCORRECTO!!, no se puede crear instancias de un parámetro de tipo
    }

    static T sample () {} // ¡¡INCORRECTO!!, no se puede usar un tipo 'T' como tipo de devolución
```

```
    static <T> boolean sample () // Correcto
}
```

Expresiones lambda

Básicamente **una expresión lambda es un método anónimo**. Sin embargo, este método no se ejecuta por sí solo, sino que se usa para implementar un método definido por una **interfaz funcional**. Estas interfaces funcionales anteriormente se conocían por SAM (*Single Abstract Method*).

Las expresiones lambda también suele denominarse *'closure'*.

Una interfaz funcional es una interfaz que únicamente contiene un método abstracto. Por lo tanto, una interfaz funcional suele representar una única acción.

Una interfaz funcional puede incluir métodos predeterminados y/o métodos estáticos pero en todos los casos solo puede haber **un solo método abstracto** para que la interfaz sea considerada interfaz funcional. Como los métodos de interfaz no predeterminados y no estáticos son implícitamente abstractos, no es necesario utilizar la palabra clave `abstract`.

```
interface Sample { // interfaz funcional
  double getValue(); // método implícitamente abstracto
}
```

Fundamentos

El nuevo operador para las expresiones lambda se denomina **operador lambda** y tiene la forma de flecha `->`. Divide la expresión lambda en dos partes: la parte izquierda especifica los parámetros necesarios y la parte derecha contiene el cuerpo de la expresión.

Este cuerpo puede estar compuesto por una única expresión o puede ser un bloque de código. Cuando es una única expresión se denomina **lambda de expresión** y cuando es un bloque de código se denomina **lambda de bloque**.

```
() -> 98.6;  // Expresión lambda sin parámetros que evalúa un valor constante

(int n) -> 100 - n: // Expresión lambda con un parámetro

(n) -> 100 - n; // Expresión lambda con un parámetro cuyo tipo es inferido

n -> 100 - n; // Cuando sólo hay un parámetro los paréntesis son opcionales
```

Una expresión lambda no se ejecuta por sí misma, sino que forma la **implementación del método abstracto** definido por la interfaz funcional que especifica su tipo de destino. Como resultado, una expresión lambda solo se puede especificar en un contexto en el que se haya definido un tipo de destino. Uno de estos contextos se crea al asignar una expresión lambda a una referencia de interfaz funcional. Otros contextos de tipo de destino son la inicialización de variables, las instrucciones `return` y los argumentos de métodos por ejemplo.

```
interface IFuncional { // interfaz funcional
  double getValue(); // método abstracto
}

// Referencia a una interfaz funcional
IFuncional sample;
```

```
// Usar una expresión lambda en un contexto de asignación a una referencia de interfaz funcional
sample = () -> 98.6;
```

Al invocar el método de la interfaz funcional se ejecuta la implementación de la expresión lambda.

```
// Usamos la referencia para invocar el método de la interfaz y que ha sido implementado por la expresión lambda.
sample.getValue();
```

Por lo general, el tipo del método abstracto definido por la interfaz funcional y el tipo de la expresión lambda deben ser compatibles. Esto es, **el tipo de devolución y la firma del método de la interfaz funcional deben ser iguales o compatibles con la expresión lambda**

```
// Interfaz funcional con un método que acepta dos parámetros y devuelve un booleano
interface IFuncional {
    boolean areEquals(int a, int b);
}

IFuncional sample = (n, m) -> n == m;

// Forma opcional porque el compilador puede inferir los tipos de n y m por el contexto
// IFuncional sample = (int n, int m) -> n == m;

sample.areEquals(10, 15); // Invocar el método.
```

Bloques de expresión lambda

Para crear una lambda de bloque basta encerrar las instrucciones entre llaves. La lambda de bloque funciona igual que las lambda de expresión con la salvedad que hay que incluir en una lambda de bloque una instrucción `return` para devolver un valor.

En una lambda de bloque podemos declarar variables, utilizar bucles, instrucciones `switch`, etc.. Una lambda de bloque funciona como un método.

Interfaces funcionales genéricas

La interfaz funcional asociada a una expresión (o bloque) lambda puede ser genérica. En este caso, el tipo de destino de la expresión lambda se determina, en parte, por el tipo de argumento o argumentos especificados al declarar la interfaz funcional.

```
// Interfaz funcional usando genéricos
interface IFuncional<T, V extends T> {
    boolean areEquals(T a, V b);
}

IFuncional iSample = (int n, int m) -> n == m; // Expresión lambda usando enteros
iSample.areEquals(10, 20);

IFuncional strSample = (String n, String m) -> n.equals(m); // Expresión lambda usando Strings
strSample.areEquals("cad", "cad");
```

Expresiones lambda como argumento de función

Una operación muy habitual es usar las expresiones lambda como argumento de una función.

```java
// Interfaz funcional
interface IFuncional {
    boolean areEquals(int a, int b);
}

class LambdaArgumentDemo {
    // Método estático que acepta una interfaz funcional de tipo IFuncional como primer parámetro.
    static boolean operation(IFuncional sample, int a, int b) {
        return sample.areEquals(a, b);
    }

    public static void main(String...args) {
        IFuncional sample = (int n, int m) -> n == m;

        // Se pasa un referencia una instancia de la interfaz IFuncional creada con una expresión lambda.
        LambdaArgumentDemo.operation(sample, 10, 15);

        // También es posible pasar la expresión lambda directamente a la función
        LambdaArgumentDemo.operation((n, m) -> n == m, 10, 15);
    }
}
```

Expresiones lambda y captura de variables

Las variables definidas por el ámbito contenedor de una expresión lambda son accesibles desde la propia expresión lambda, como por ejemplo variables de instancia o una variable `static` definida por su clase contenedora. Una expresión lambda también tiene acceso a `this`, lo que hace referencia a la instancia de invocación de la clase contenedora de la expresión lambda.

Sin embargo, cuando una expresión lambda usa una variable local desde su ámbito contenedor, se crea una situación especial denominada **captura de variables**. En ese caso, la expresión lambda puede usar únicamente variables locales que sean **eficazmente finales**.

Una variable eficazmente final es aquella cuyo valor no cambia una vez asignada. No es necesario declararla explícitamente como final.

```java
// Interfaz funcional
interface IFuncional {
    int func(int a);
}

class VarCapture {
    public static void main(String...args) {
        int num = 10; // variable local a capturar en la expresión lambda

        IFuncional sample = (n) -> {
            int v = n + num; // Uso correcto. La variable 'num' no se modifica

            /* Uso incorrecto ya que la variable 'num' se modifica dentro de la expresión
            y por tanto ya no es una variable eficazmente final */
            // num++

            return v;
        };

        sample.func(100);  // Uso de la expresión lambda.
    }
}
```

Generar una excepción desde una expresión lambda

Una expresión lambda puede generar una excepción. No obstante, si genera una excepción comprobada, esta tendrá que ser compatible con la excepción (o excepciones) indicadas en la cláusula `throws` del método abstracto de la interfaz funcional.

```
interface IFuncional {
    boolean ioAction(Reader rdr) throws IOException;
}

class LambdaExceptionDemo {
    public static void main(String...args){
        IFuncional sample = (rdr) -> {
            // Como la invocación a 'read()' generaría una IOException, el método 'ioAction()'
            // de la interfaz funcional debe incluir IOException en una cláusula 'throws'
            int ch = rdr.read();

            return true;
        };
    }
}
```

Referencias de métodos 'static' y métodos de instancia

Una referencia de método permite hacer referencia a un método sin ejecutarlo. Al evaluar una referencia de método también se crea una instancia de una interfaz funcional.

El nombre de la clase se separa del método mediante un par de puntos `::`, un nuevo separador añadido a Java en la JDK 8:

- Sintaxis para métodos estáticos: `NombreClase::nombreMétodo`
- Sintaxis para métodos de instancia: `refObj::nombreMétodo`

Si es un método genérico la sintaxis es `NombreClase::<T>nombreMétodo` o `refObj::<T>nombreMétodo`

```
interface IntPredicate {
    boolean areEquals(int n, int m);
}

public class Sample {
    // Método estático que recibe dos parámetros de tipo int y los compara entre sí
    static boolean compare(int a, int b) {
        return a == b;
    }

    // Método miembro
    boolean compare2(int a, int b) {
        return a == b;
    }

    // Este método tiene una interfaz funcional como tipo en su primer parámetro
    static boolean numTest(IntPredicate p, int a, int b) {
        return p.areEquals(a, b);
    }

    public static void main(String...args) {
        // Pasamos a numTest() una referencia de método estático
        System.out.println(Sample.numTest(Sample::compare, 10, 10)); // => true

        Sample sample = new Sample();

        IntPredicate p = sample::compare2; // Se crea una referencia de método

        System.out.println(Sample.numTest(p, 10, 15)); // => false
        System.out.println(Sample.numTest(sample::compare2, 15, 15)); // => true
```

```
    }
}
```

Referencias de constructor

Al igual que se crean referencias de método, se pueden crear referencias a constructores. La sintaxis es `NombreClase::new`. Si es una clase con genéricos la sintaxis es `NombreClase<T>::new`. En el caso de una matriz tiene la sintaxis `tipo[]::new`

```
interface IntPredicate {  // Interfaz funcional
    MyClass create(String n); // Método abstracto que recibe un 'String' como parámetro y retorna 'MyClass'
}

class MyClass {
    String name;

    MyClass(String n) {
        name = n;
    }

    MyClass() {
        name = "";
    }
}

public class Sample {
    public static void main(String...args) {
        IntPredicate p = MyClass::new; // Una referencia de constructor

        MyClass c = p.create("MyClass");

        System.out.println(c.name);
    }
}
```

Interfaces funcionales predefinidas

En Java 8 apareció el paquete `java.util.function` que proporciona una serie de **interfaces funcionales predefinidas** preparadas para utilizar:

- `Consumer<T>`: Representa una operación que acepta un solo argumento de entrada y no devuelve ningún resultado.
- `BiConsumer<T, U>`: Representa una operación que acepta dos argumentos de entrada y no devuelve ningún resultado.
- `Function<T, R>`: Representa una función que acepta un argumento y produce un resultado.
- `BiFunction<T, U, R>`: Representa una función que acepta dos argumentos y produce un resultado.
- `UnaryOperator<T>`: Representa una operación en un solo operando que produce el mismo tipo que su operando.
- `BinaryOperator<T>`: Representa una operación sobre dos operandos del mismo tipo, produciendo un resultado del mismo tipo que los operandos.
- `Supplier<T>`: Representa una función que no acepta argumentos y devuelve un resultado.
- `Predicate<T>`: Representa un predicado (función que se evalua de forma booleana) de un argumento.

```
import java.util.function.Predicate; // Importar la interfaz 'Predicate'

public class Sample {
    public static void main(String...args) {
        Predicate<Integer> isEven = n -> (n % 2) == 0;

        System.out.println("4 es par? " + isEven.test(4));
    }
}
```

Más información

Stream API

La API *Stream*, introducida en la versión 1.8, es un juego de utilidades para la manipulación de grandes agrupaciones de objetos en memoria.

Este mecanismo explota las capacidades de las expresiones lambda. La API *Stream* no es un nuevo tipo de colección sino más bien un envoltorio que facilita su manipulación.

Esta API ofrece dos tipos de operaciones:

- **Operaciones intermedias**: son aquellas que producen o retornan un nuevo *stream*, pudiéndose concatenar unas con otras como por ejemplo:
  - **distinct**: retorna un nuevo *stream* con los elementos diferentes entre sí
  - **filter**: retorna un nuevo *stream* de acuerdo con la expresión pasada como parámetro
  - **limit**: retorna un nuevo *stream* con el número máximo de elementos pasado como parámetro
  - **of**: retorna un *stream* a partir de un array
  - **sorted**: retorna un nuevo *stream* ordenado
- **Operaciones finales**: son aquellas que no producen un *stream*, como por ejemplo:
  - **count**: retorna el número de elementos
  - **findFirst**: retorna el primer elemento
  - **forEach**: realiza una acción sobre cada uno de los elementos
  - **max/min**: retorna el máximo/mínimo elemento

Más información

Crear un stream

Todas las clases de tipo `Stream` tienen un método `of()` que recibe como parámetro un **array de objetos**. La clase `java.util.Arrays` también dispone del método `stream()` con el mismo objetivo:

```
int[] enteros = new int[]{1, 2, 3, 4, 5};

IntStream strEnt = IntStream.of(enteros);
IntStream strEnt3 = IntStream.of(4, 5, 6);
IntStream strEnt2 = Arrays.stream(enteros);
```

La mayoría de las clases del **framework de colecciones** disponen de un método `stream()` para crear un *stream*:

```
List<Empleado> empleados = getListaEmpleados();

Stream strEmp = Stream.of(empleados);
Stream strEmp2 = empleados.stream();
Stream strEmp3 = empleados.parallelStream();
```

Recorrer un stream

Recorrer un *stream* se considera una **operación final**.

Para recorrer o ejecutar una acción sobre cada uno de los elementos de un *stream* se utiliza el método `forEach()` junto con una expresión lambda que representará la implementación de una interfaz funcional de tipo `java.util.function.Consumer`:

```
IntStream.of(1, 2, 3, 4, 5, 6).forEach(e -> System.out.println("Entero: " + e));

empleados.stream().forEach(emp -> System.out.println("Nombre: " + emp.getNombre()));
```

Operaciones de filtrado

Filtrar un *stream* se considera una **operación intermedia**.

Las operaciones de filtrado se realizarán a través del método `filter()` empleando una expresión lambda que representará la implementación de una interfaz funcional de tipo `java.util.function.Predicate` y establecerá las condiciones de filtrado:

```
IntStream.of(5, 20, 32, 8, 14, 24) // se crea el stream
    .filter(e -> e > 10) // retorna un nuevo stream filtrado
    .forEach(e -> System.out.println( e + " es mayor que 10")); // recorrer el stream
```

Operaciones de ordenación

Ordenar un *stream* se considera una **operación intermedia**.

Las operaciones de ordenación de tipos primitivos serán automáticas con el método `sorted()` mientras que para la ordenación de objetos se realiza con la implementación de la interfaz `java.util.Comparator`:

```
IntStream.of(5, 20, 32, 8, 14, 24)
    .sorted()   // ordena el stream
    .forEach(e -> System.out.println(e)); // imprime el stream ordenado

empleados.stream()
    .sorted((emp1, emp2) -> emp1.getAge() - emp2.getAge())
    .forEach(emp -> System.out.println("Empleado: " + emp.getName()));
```

Operaciones de mapeo

Mapear un *stream* se considera una **operación intermedia**.

Las operaciones de mapeo permiten aplicar una función a un *stream* para producir otro *stream* de tipo diferente como por ejemplo `mapToInt()` o `mapToLong()`. La expresión lambda es una implementación de la interfaz funcional `java.util.function.Function`.

El hecho de construir un *stream* de tipos primitivos puede servir para realizar algún tipo de operación como por ejemplo una suma o una media.

```
empleados.stream()
    .sorted((emp1, emp2) -> emp1.getAge() - emp2.getAge()) // ordena el stream
    .mapToInt(emp -> emp.getAge()) // mapea la edad en un nuevo stream
    .forEach(emp -> System.out.println("Edad del empleado: " + emp)); // recorre el stream
```

Operaciones aritméticas

Este tipo de operaciones se consideran una **operación final**.

Las clases *stream* que envuelven tipos primitivos como las subclases `IntStream`, `DoubleStream` o `LongStream` incorporan métodos que permiten realizar algunas operaciones aritméticas como por ejemplo `sum()`, `average()`, `max()`, `min()` o `count()`:

```
empleados.stream()
    .mapToInt(emp -> emp.getAge()) // mapea la edad en un nuevo stream
    .average()  // calcula la media de los enteros del stream
    .getAsDouble(); // imprime la media de edad
```

Operaciones de colección

Este tipo de operaciones se consideran una **operación final**.

Permiten generar un nuevo objeto o una lista de ellos a partir de un *stream*:

```
List<String> nombres = empleados.stream()
    .map(emp -> emp.getName()) // mapea el nombre en un nuevo stream
    .sorted()   // ordena alfabéticamente los nombres
    .collect(Collectors.toList()); // retorna una lista a partir del stream
```

Colecciones

Una **colección** -a veces llamada contenedor- es simplemente un objeto que agrupa múltiples elementos en una sola unidad. Las colecciones se utilizan para almacenar, recuperar, manipular y comunicar datos agregados.

Un framework de colecciones es una arquitectura unificada para representar y manipular colecciones. Todos los *frameworks* de colecciones contienen lo siguiente:

- **Interfaces**: Estos son tipos de datos abstractos que representan colecciones. Las interfaces permiten manipular las colecciones independientemente de los detalles de su representación. En los lenguajes orientados a objetos, las interfases generalmente forman una jerarquía.
- **Implementaciones**: Estas son las implementaciones concretas de las interfaces de colecciones. En esencia, son estructuras de datos reutilizables.
- **Algoritmos**: Estos son métodos que realizan cálculos útiles, como la búsqueda y clasificación, en objetos que implementan las interfaces de colección. Se dice que los algoritmos son **polimórficos**, es decir, que se puede utilizar el mismo método en muchas implementaciones diferentes de la interfaz de colección correspondiente. En esencia, los algoritmos son funciones reutilizables.

La interfaz 'Collection'

Una colección representa un grupo de objetos conocidos como sus elementos. La interfaz `Collection` se utiliza para transmitir colecciones de objetos en las que se desea la máxima generalidad.

La interfaz `Collection` contiene métodos que realizan operaciones básicas como:

- `int size()`
- `boolean isEmpty()`
- `boolean contains(Object element)`
- `boolean add(E element)`
- `boolean remove(Object element)`
- `Iterator<E> iterator()`.

También contiene métodos que operan en colecciones enteras como:

- `boolean containsAll(Collection<?> c)`

- `boolean addAll(Collection<? extends E> c)`
- `boolean removeAll(Collection<?> c)`
- `boolean retainAll(Collection<?> c)`
- `void clear()`.

La interfaz `Collection` hace lo que cabría esperar, dado que una colección representa un grupo de objetos. Tiene métodos que le dicen cuántos elementos hay en la colección (`size()` , `isEmpty()`), métodos que comprueban si un objeto dado está en la colección (`contains(Object element)`), métodos que añaden y eliminan un elemento de la colección (`add(E element)` , `remove(Object element)`), y métodos que proporcionan un iterador sobre la colección (`iterator()`).

`Collection` es la superinterfaz de otras interfaces fundamentales en Java, como:

- `List` : representa una secuencia ordenada de elementos que permite duplicados y acceso por posición.
- `Set` : estructura donde no se permiten elementos duplicados y modela un conjunto matemático.
- `Queue` : ordena sus elementos generalmente de acuerdo con el principio FIFO (*first-in-first-out*), aunque existen excepciones, como las colas de prioridad.

Cada una de estas subinterfaces especializa `Collection` para diferentes estructuras de datos, añadiendo métodos y restricciones adicionales.

Los métodos `toArray()` y `toArray(T[] a)` se proporcionan como un puente entre colecciones y APIs antiguas que esperan matrices en la entrada. Las operaciones de array permiten traducir el contenido de una colección a un array. La forma sencilla sin argumentos crea una nueva matriz de `Object` . La forma más compleja permite al invocador proporcionar un array o elegir el tipo del array de salida en tiempo de ejecución.

```
Object[] a = c.toArray();

String[] a = c.toArray(new String[0]);
```

Hay tres formas de recorrer las colecciones: utilizando operaciones agregadas, con la construcción `for-each` y utilizando iteradores.

En JDK 8 y versiones posteriores, el método preferido para iterar sobre una colección es obtener un flujo y realizar operaciones agregadas en él. Las operaciones agregadas a menudo se usan junto con las **expresiones lambda** para hacer que la programación sea más expresiva, utilizando menos líneas de código.

```
myShapesCollection.stream()
.filter(e -> e.getColor() == Color.RED)
.forEach(e -> System.out.println(e.getName()));

// parallel stream if the collection is large enough
myShapesCollection.parallelStream()
.filter(e -> e.getColor() == Color.RED)
.forEach(e -> System.out.println(e.getName()));

String joined = elements.stream()
.map(Object::toString)
.collect(Collectors.joining(", "));
```

La construcción `for-each` permite recorrer de forma concisa, es decir, de uno en uno, una colección o array utilizando un bucle `for` :

```
for (Object o : collection) {
    System.out.println(o);
}
```

Un `Iterator` es un objeto que permite recorrer una colección y eliminar elementos de la colección de forma selectiva, si se desea. Se obtiene un `iterator` para una colección llamando a su método `iterator()`.

La interfaz `Iterator` tiene esta forma:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

El método `hasNext()` devuelve `true` si hay más elementos y el método `next()` devuelve el siguiente elemento. El método `remove()` elimina el último elemento devuelto por el método `next()`. Por tanto sólo puede ser invocado **una vez** por cada llamada a `next()`. Incumplir la regla lanza una excepción.

Es recomendable usar iteradores en vez de una construcción `for-each` cuando se necesita eliminar el elemento actual.

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); ) {
        if (!cond(it.next())) {
            it.remove();
        }
    }
}
```

Java permite la creación de **colecciones inmutables**, que no se pueden modificar después de su creación. Estas colecciones son útiles en entornos donde se necesita evitar modificaciones accidentales o garantizar la seguridad en entornos multihilo. Las colecciones inmutables pueden crearse mediante métodos como `Collections.unmodifiableCollection(Collection<? extends T> c)` o con las fábricas de colecciones inmutables introducidas en Java 9, como `List.of()`, `Set.of()`, y `Map.of()`.

La clase `Collections` es una clase utilitaria en Java que proporciona métodos estáticos para operar en y devolver colecciones. Esta clase no puede ser instanciada, ya que su constructor es privado.

Java ofrece implementaciones específicas para trabajar con colecciones en entornos multihilo, donde múltiples hilos acceden a la colección simultáneamente. Algunas de las implementaciones más comunes son:

- `ConcurrentHashMap`: un mapa concurrente eficiente que permite acceso seguro en entornos multihilo sin necesidad de sincronización explícita.
- `CopyOnWriteArrayList`: una lista que crea una copia de la estructura subyacente en cada modificación, asegurando la seguridad en concurrencia pero a costa de un mayor uso de memoria.

Estas colecciones concurrentes proporcionan un manejo más eficiente y seguro en situaciones donde las colecciones estándar no son adecuadas.

La interfaz 'Set'

Un `Set` (conjunto) es una colección que **no puede contener elementos duplicados**. Modela la abstracción del conjunto matemático. La interfaz `Set` hereda métodos de `Collection` pero añade la restricción de que los elementos duplicados están prohibidos.

Para garantizar la unicidad de los elementos, ya que no se permiten los duplicados, `Set` depende de las implementaciones correctas de los métodos `equals(Object o)` y `hashCode()`. Estos métodos se utilizan para comparar los elementos dentro del conjunto, asegurando que dos elementos sean considerados iguales si tienen el mismo valor, incluso si provienen de diferentes implementaciones de `Set`.

Dos instancias de `Set` son consideradas iguales si contienen los mismos elementos, sin importar el orden o la implementación subyacente, siempre que se respete el contrato de `equals(Object o)`.

Algunas de las implementaciones más conocidas de la interfaz `Set` son:

- `HashSet`, que almacena los elementos en una tabla *hash*. Es la implementación más eficiente en términos de rendimiento, pero no garantiza ningún orden de iteración.
- `TreeSet` que almacena los elementos en un árbol *'red-black'* (rojo-negro), ordenando los elementos en función de sus valores. Aunque es más lenta que `HashSet`, garantiza el orden natural de los elementos.
- `LinkedHashSet`, que implementa una tabla *hash* vinculada a una lista enlazada. Mantiene el orden de inserción de los elementos, aunque tiene un coste ligeramente superior al de `HashSet`.

La interfaz `Set` cuenta con una subinterface `SortedSet`, que asegura que los elementos se mantengan en orden ascendente, ya sea según el orden natural o de acuerdo con un `Comparator` proporcionado al momento de la creación del `SortedSet`.

La interfaz 'List'

Una `List` es una **colección ordenada que pueden contener elementos duplicados** (también llamada secuencia). Además de las operaciones heredadas de `Collection`, la interfaz `List` incluye funcionalidades adicionales:

- **Acceso por posición**: permite manipular los elementos de la lista por su índice. Los métodos incluyen `get(int index)`, `set(int index, E element)`, `add(E e)` y `remove()`.
- **Búsqueda**: se pueden buscar elementos específicos dentro de la lista y devolver su posición numérica mediante métodos como `indexOf(Object o)` y `lastIndexOf(Object o)`.
- **Iteración avanzada**: extiende las capacidades de iteración, aprovechando la naturaleza secuencial de las listas, con el uso de `listIterator()`, que permite recorrer la lista en ambas direcciones.
- **Operaciones sobre sublistas**: proporciona la posibilidad de realizar operaciones arbitrarias sobre secciones específicas de la lista con el método `subList(int fromIndex, int toIndex)`.

Algunas de las implementaciones más conocidas de la interfaz `List` son:

- `ArrayList`, que suele ser la implementación de mejor rendimiento en la mayoría de los casos.
- `LinkedList`, que puede ofrecer un mejor rendimiento en ciertas situaciones, como inserciones o eliminaciones frecuentes en el medio de la lista.

La interfaz 'Queue'

Una `Queue` (cola) es una colección que contiene elementos antes de ser procesados. Además de las operaciones básicas de una `Collection`, las colas proporcionan operaciones adicionales para insertar, extraer e inspeccionar elementos.

Algunas de las implementaciones más conocidas de la interfaz `Queue` son:

- `LinkedList` implementa tanto la interfaz `Queue` como la interfaz `List`, lo que le permite actuar como una lista enlazada y una cola al mismo tiempo.

- `PriorityQueue` es una cola de prioridad basada en la estructura de pila de datos. Esta cola ordena los elementos según el orden especificado en el momento de la construcción, que puede ser el orden natural de los elementos o el orden impuesto por un comparador explícito.

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

Cada método de `Queue` existe en dos formas: una forma lanza una **excepción** si la operación falla, y la otra forma devuelve un valor **especial** si la operación falla (ya sea nulo o falso, dependiendo de la operación):

Operación	Lanza excepción	Nulo o false
Insertar	`add(e)`	`offer(e)`
Eliminar	`remove()`	`poll()`
Inspeccionar	`element()`	`peek()`

Las colas generalmente siguen un **orden FIFO** (*first-in-first-out*), aunque no siempre. Las excepciones incluyen las colas de prioridad, que ordenan los elementos según sus valores o prioridades.

Independientemente del orden, la cabeza de la `Queue` es el elemento que sería eliminado por una llamada a `remove()` o `poll()`. En una cola FIFO, los nuevos elementos se insertan al final. Otros tipos de colas pueden utilizar diferentes reglas para la colocación de elementos. Cada implementación de cola debe especificar sus propias reglas de orden.

Algunas implementaciones de `Queue` pueden tener una capacidad limitada, conocidas como **colas acotadas** (*bounded*).

El método `add()`, heredado de `Collection`, inserta un elemento a menos que viole las restricciones de capacidad de la cola, en cuyo caso lanza una `IllegalStateException`. Por otro lado, el método `offer()`, comúnmente usado en colas acotadas, devuelve `false` si el elemento no puede ser insertado.

Los métodos `remove()` y `poll()` eliminan y devuelven la cabeza de la cola. Si la cola está vacía, `remove()` lanza una `NoSuchElementException`, mientras que `poll()` simplemente devuelve nulo.

Finalmente, los métodos `element()` y `peek()` devuelven, pero no eliminan, la cabeza de la cola. La diferencia es que `element()` lanza una excepción si la cola está vacía, mientras que `peek()` devuelve nulo.

La interfaz 'Deque'

Una `Deque` (*Double-Ended Queue* o cola de dos extremos) es una estructura de datos que permite **la inserción y extracción de elementos en ambos extremos**. A diferencia de una `Queue` convencional, que solo permite operaciones en un extremo (**FIFO**), una `Deque` admite operaciones en ambos extremos, lo que la hace más flexible.

La interfaz `Deque` es más versátil que `Stack` y `Queue`, ya que puede implementarse tanto como una pila (**LIFO**) como una cola (**FIFO**). Algunas de las implementaciones más conocidas de la interfaz `Deque` son:

- `ArrayDeque`: una implementación basada en un *array* redimensionable, eficiente para la mayoría de las operaciones.
- `LinkedList`: una implementación basada en una lista doblemente enlazada, que ofrece un rendimiento consistente para inserciones y eliminaciones en ambos extremos.

La interfaz `Deque` define métodos específicos para acceder, insertar, eliminar y examinar elementos en ambos extremos de la colección. Existen versiones de métodos que pueden lanzar excepciones o devolver un valor nulo en caso de fallos.

Operación	First Element	Last Element
Insertar (Exception)	`addFirst(e)`	`addLast(e)`
Insertar (boolean)	`offerFirst(e)`	`offerLast(e)`
Eliminar (Exception)	`removeFirst()`	`removeLast()`
Eliminar (null)	`pollFirst()`	`pollLast()`
Examinar (Exception)	`getFirst()`	`getLast()`
Examinar (null)	`peekFirst()`	`peekLast()`

La interfaz 'Map'

Un `Map` es un objeto que asigna claves a valores. Un mapa **no puede contener claves duplicadas**; cada clave puede estar asociada a un valor como máximo. Este comportamiento modela la abstracción de una función matemática.

Es importante notar que, a diferencia de otras interfaces en el marco de colecciones de Java como `List` o `Set`, la interfaz `Map` no hereda de `Collection`. Esto se debe a que trabaja con pares clave-valor, mientras que `Collection` opera sobre elementos individuales. Debido a esta diferencia conceptual, `Map` no incluye métodos como `add()` o `iterator()`, que son característicos de las colecciones.

Algunas de las implementaciones más conocidas de la interfaz `Map` son:

- `HashMap` almacena sus elementos en una **tabla *hash*** y es generalmente la implementación más rápida. Sin embargo, no garantiza el orden de iteración de los elementos.
- `TreeMap` almacena sus elementos en un **árbol *'red-black'*** (árbol rojo-negro) lo que le permite ordenar los elementos en función de sus claves. Esta implementación es más lenta en comparación con `HashMap`.
- `LinkedHashMap` combina una **tabla *hash* con una lista enlazada** que mantiene el orden de inserción de los elementos. Aunque tiene un costo ligeramente mayor que `HashMap`, conserva el orden de inserción de los elementos durante la iteración.

La interfaz `Map` tiene una subinterfaz, `SortedMap`, que garantiza que sus elementos se mantengan en orden ascendente, ya sea según el orden natural de las claves o de acuerdo a un `Comparator` proporcionado al crear el `SortedMap`.

Análisis de complejidad en Java Collections

En la **Java Collections Framework**, se utilizan varias estructuras de datos que proporcionan diferentes niveles de eficiencia en términos de tiempo para las operaciones comunes, como acceso, búsqueda, inserción y eliminación. La elección de una estructura de datos adecuada puede tener un impacto significativo en el rendimiento de una aplicación, dependiendo de los patrones de uso que se requieran.

A continuación, se presenta una tabla con la **complejidad temporal**, expresada en notación O grande, de las operaciones más comunes en las estructuras de datos más utilizadas en Java. Esta notación ayuda a estimar el tiempo de ejecución en función del tamaño de la colección:

Estructura de Datos	Acceso	Búsqueda	Inserción	Eliminación
ArrayList	O(1)	O(n)	O(1)	O(n)

Estructura de Datos	Acceso	Búsqueda	Inserción	Eliminación
LinkedList	O(n)	O(n)	O(1)	O(1)
HashMap	-	O(1)	O(1)	O(1)
TreeMap	O(log n)	O(log n)	O(log n)	O(log n)
HashSet	-	O(1)	O(1)	O(1)
TreeSet	O(log n)	O(log n)	O(log n)	O(log n)
PriorityQueue	O(n)	O(n)	O(log n)	O(log n)
LinkedHashMap	-	O(1)	O(1)	O(1)
Stack (basado en **Vector**)	O(1)	O(n)	O(1)	O(1)
Deque (como **ArrayDeque**)	O(1)	O(n)	O(1)	O(1)

- `ArrayList` tiene tiempo constante para el acceso a elementos, pero las operaciones de búsqueda, inserción o eliminación pueden requerir mover varios elementos.
- `LinkedList` tiene tiempos más costosos de acceso y búsqueda debido a su naturaleza lineal.
- Las estructuras basadas en tablas de *hash* (`HashMap` , `HashSet` , `LinkedHashMap`) ofrecen tiempos de búsqueda, inserción y eliminación constantes promedio.
- Las estructuras ordenadas como `TreeMap` y `TreeSet` tienen operaciones logarítmicas debido a su implementación basada en árboles (generalmente árboles rojos-negros).
- `PriorityQueue` está diseñada para obtener el elemento de mayor prioridad de manera eficiente, pero las búsquedas dentro de la cola son lineales.

Módulos

Con la aparición de JDK 9 se incorporó a Java la característica de los **módulos**. Un módulo es una agrupación de paquetes y recursos a los que se puede hacer referencia conjuntamente a través del nombre del módulo.

La declaración de un módulo son instrucciones en un archivo fuente de Java llamado *'module-info.java'*. Luego `javac` compila ese archivo en un archivo de clase que se conoce como **descriptor de módulo**. Un descriptor de módulo empieza por la palabra clave `module` y tiene la sintaxis:

```
module nombreMódulo {
    // definición de módulo
}
```

Para especificar la dependencia de un módulo se utiliza la sintaxis `requires NombreMódulo` .

Para exportar un módulo y permitir su uso en otros módulos se utiliza la sintaxis `exports NombrePaquete` . Cuando un módulo exporta un paquete, hace que todos los tipos públicos y protegidos del paquete sean accesibles para otros módulos. Además, los miembros `public` y `protected` de esos tipos también son accesibles. Cualquier paquete no exportado es sólo para uso interno de su módulo. Por tanto, la visibilidad `public` que es la menos restrictiva es únicamente visible dentro de su propio módulo hasta que no se *'exporte'*, lo que hace que sea visible para otros módulos.

Tanto `requires` como `exports` deben estar solo dentro de una declaración de módulo.

Módulos de la plataforma

A partir de JDK 9 los paquetes de la API de Java se han incorporado a módulos, permitiendo implementar aplicaciones con únicamente los paquetes necesarios de la JRE, reduciendo considerablemente el tamaño de las aplicaciones.

De los módulos de la plataforma, el más importante es `java.base`. Este módulo incluye y exporta paquetes esenciales de Java como `java.lang`, `java.io` o `java.util` entre otros. Dada su importancia está disponible automáticamente para todos los progamas sin necesidad de usar la instrucción `import` y todos los módulos lo requieren automáticamente y por tanto tampoco es necesario usar la instrucción `requires`.

Módulos y código legado

Para permitir la compatibilidad con código anterior a JDK 9, Java introduce dos características para permitir dicha compatibilidad.

Cuando se usa código legado que no forma parte de un módulo nombrado, pasa automáticamente a formar parte del **"módulo sin nombre"**. Este módulo tiene dos atributos importantes. En primer lugar, todos los paquetes que contiene se exportan de forma automática. En segundo lugar, este módulo puede acceder a todos los demás. Por tanto, cuando un programa no usa módulos, todos los módulos de la API de la plataforma Java se vuelven accesibles automáticamente a través del **"módulo sin nombre"**.

Otra característica que permite la compatibilidad con código legado es el uso automático de la ruta de clase en vez de la ruta de módulo.

Introducción a JShell

A partir del JDK 9 se incluye una herramienta llamada **JShell** que proporciona un entorno interactivo para experimentar de manera rápida y fácil código Java.

JShell implementa lo que se conoce como ejecución **REPL** ***(read-evaluate-print loop)***. Con este mecanismo, se introduce un fragmento de código que se lee y evalúa. A continuación, JShell muestra el resultado del código y queda a la espera del siguiente fragmento o expresión.

Cada secuencia de código introducida se llama *snippet*.

JShell puede evaluar fragmentos de código y expresiones ya que entre bastidores proporciona una clase y un método sintéticos. Es decir, proporciona todo lo necesario para poder ejecutar por ejemplo una instrucción como `System.out.println()`.

En JShell se puede utilizar variables, expresiones, métodos, clases, interfaces, enumeraciones, etcétera…

Además, en JShell se importan por defecto varios paquetes por lo que no es necesario realizar importaciones de los paquetes más comunes. Pueden listarse con `/imports`.

Para iniciar JShell escribimos `jshell` desde la línea de comandos.

Podemos cargar un fichero de cualquier extensión con fragmentos de código o con una sesión previa con `jshell nombrearchivo`.

Una vez iniciada la consola, JShell dispone de una serie de comandos que empiezan por `/`:

- **/help**: muestra la ayuda
- **/exit**: salir de JShell
- **/list**: muestra todos los *snippets* introducidos
- **/edit o /edit n**: permite editar todos los snipets o uno en concreto
- **/save nombrearchivo**: permite guardar la sesión actual de *snippets*
- **/open nombrearchivo**: permite cargar una sesión de *snippets*

- **/types**: muestra clases, interfaces y enumeraciones
- **/imports**: muestra las importaciones
- **/methods**: muestra los métodos
- **/vars**: muestra las variables

Más información aquí o aquí

Histórico de versiones

JDK 1.0 (23/01/1996)

- Primera versión

JDK 1.1 (19/02/1997)

- Reestructuración intensiva del modelo de eventos AWT (Abstract Windowing Toolkit)
- Clases internas (inner classes)
- JavaBeans
- JDBC (Java Database Connectivity), para la integración de bases de datos
- RMI (Remote Method Invocation)

J2SE 1.2 (08/12/1998)

- Palabra reservada (keyword) `strictfp`
- Reflexión en la programación
- API gráfica (Swing) fue integrada en las clases básicas
- Máquina virtual (JVM) de Sun fue equipada con un compilador JIT (Just in Time) por primera vez
- Java Plug-in
- Java IDL, una implementación de IDL (Lenguaje de Descripción de Interfaz) para la interoperabilidad con CORBA
- Colecciones (Collections)

J2SE 1.3 (08/05/2000)

- Inclusión de la máquina virtual de HotSpot JVM (la JVM de HotSpot fue lanzada inicialmente en abril de 1999, para la JVM de J2SE 1.2)
- RMI fue cambiado para que se basara en CORBA
- JavaSound
- Inclusión de 'Java Naming and Directory Interface' (JNDI) en el paquete de bibliotecas principales (anteriormente disponible como una extensión)
- Java Platform Debugger Architecture (JPDA)

J2SE 1.4 (06/02/2002)

- Palabra reservada `assert`
- Expresiones regulares modeladas al estilo de las expresiones regulares Perl
- Encadenación de excepciones. Permite a una excepción encapsular la excepción de bajo nivel original.
- Non-blocking NIO (New Input/Output)
- Logging API
- API I/O para la lectura y escritura de imágenes en formatos como JPEG o PNG
- Parser XML integrado y procesador XSLT (JAXP)
- Seguridad integrada y extensiones criptográficas (JCE, JSSE, JAAS)
- Java Web Start incluido (El primer lanzamiento ocurrió en marzo de 2001 para J2SE 1.3)

J2SE 5.0 (30/09/2004)

- Genéricos
- Anotaciones
- Autoboxing/unboxing
- Enumeraciones
- `Varargs` (número de argumentos variable)
- Bucle `for` mejorado.
- Utilidades de concurrencia
- Clase `Scanner`

Java SE 6 (11/12/2006)

- Incluye un nuevo marco de trabajo y APIs que hacen posible la combinación de Java con lenguajes dinámicos como PHP, Python, Ruby y JavaScript.
- Incluye el motor Rhino, de Mozilla, una implementación de Javascript en Java.
- Incluye un cliente completo de Servicios Web y soporta las últimas especificaciones para Servicios Web, como JAX-WS 2.0, JAXB 2.0, STAX y JAXP.
- Mejoras en la interfaz gráfica y en el rendimiento.

Java SE 7 (07/07/2011)

- Soporte para XML dentro del propio lenguaje.
- Un nuevo concepto de superpaquete.
- Soporte para `closures`.
- Introducción de anotaciones estándar para detectar fallos en el software.
- NIO2.
- Java Module System.
- Java Kernel.
- Nueva API para el manejo de Días y Fechas, la cual reemplazará las antiguas clases `Date` y `Calendar`.
- Posibilidad de operar con clases `BigDecimal` usando operandos.
- Uso de `Strings` en bloques `switch`
- Uso de guiones bajos en literales numéricos (1_000_000)

Java SE 8 LTS (18/03/2014)

- JDK 8 Documentation
- Lista completa de características - JEP
  - JEP 126: Lambda Expressions & Virtual Extension Methods
  - JEP 153: Launch JavaFX Applications
  - JEP 178: Statically-Linked JNI Libraries
  - JEP 155: Concurrency Updates
  - JEP 174: Nashorn Javascript Engine
  - JEP 104: Annotations on Java Types
  - JEP 150: Date & Time API

Java 9 (21/09/2017)

- JDK 9 Documentation
- Java Language Changes for Java SE 9
- Significant Changes in JDK 9 Release
- Lista completa de características - JEP
  - JEP 200: The Modular JDK
  - JEP 222: 'jshell': The Java Shell (Read-Eval-Print Loop)

- JEP 295: Compilación *Ahead-of-Time*
- JEP 282: jlink: The Java Linker
- JEP 266: More Concurrency Updates
- JEP 263: Gráficos HiDPI
- JEP 224: HTML5 Javadoc
- JEP 275: Modular Java Application Packaging
- JEP 261: Module System

Java 10 (20/03/2018)

- JDK 10 Documentation
- Java Language Changes for Java SE 10
- Significant Changes in JDK 10 Release
- Lista completa de características - JEP
  - JEP 286: Local-Variable Type Inference
  - JEP 317: Experimental Java-Based JIT Compiler
  - JEP 310: Application Class-Data Sharing
  - JEP 322: Time-Based Release Versioning
  - JEP 307: Parallel Full GC for G1
  - JEP 304: Garbage-Collector Interface
  - JEP 314: Additional Unicode Language-Tag Extensions
  - JEP 319: Root Certificates
  - JEP 312: Thread-Local Handshakes
  - JEP 316: Heap Allocation on Alternative Memory Devices
  - JEP 313: Remove the Native-Header Generation Tool – javah
  - JEP 296: Consolidate the JDK Forest into a Single Repository

Java 11 LTS (25/09/2018)

- JDK 11 Documentation
- Java Language Changes for Java SE 11
- Significant Changes in JDK 11 Release
- Lista completa de características -JEP
  - JEP 309: Dynamic Class-File Constants
  - JEP 318: Epsilon: A No-Op Garbage Collector
  - JEP 323: Local-Variable Syntax for Lambda Parameters
  - JEP 331: Low-Overhead Heap Profiling
  - JEP 321: HTTP Client (Standard)
  - JEP 332: Transport Layer Security (TLS) 1.3
  - JEP 328: Flight Recorder
  - JEP 335: Deprecate the Nashorn Javascript Engine

Java 12 (19/03/2019)

- JDK 12 Documentation
- Java Language Changes for Java SE 12
- Significant Changes in JDK 12 Release
- Lista completa de características- JEP
  - JEP 230: Microbenchmark Suite
  - JEP 334: JVM Constants API

Java 13 (17/09/2019)

- JDK 13 Documentation

- Java Language Changes for Java SE 13
- Significant Changes in JDK 13 Release
- Lista completa de características - JEP
  - JEP 353: Reimplement the Legacy Socket API

Java 14 (17/03/2020)

- JDK 14 Documentation
- Java Language Changes for Java SE 14
- Significant Changes in JDK 14 Release
- Lista completa de características - JEP
  - JEP 358: Helpful NullPointerExceptions
  - JEP 361: Switch Expressions
  - JEP 349: JFR Event Streaming

Java 15 (15/09/2020)

- JDK 15 Documentation
- Java Language Changes for Java SE 15
- Significant Changes in JDK 15 Release
- Lista completa de características - JEP
  - JEP 371: Hidden Classes
  - JEP 372: Remove the Nashorn JavaScript Engine
  - JEP 373: Reimplement the Legacy DatagramSocket API
  - JEP 378: Text Blocks

Java 16 (16/03/2021)

- JDK 16 Documentation
- Java Language Changes for Java SE 16
- Significant Changes in JDK 16 Release
- Lista completa de características - JEP
  - JEP 347: Enable C++14 Language Features
  - JEP 369: Migrate to GitHub
  - JEP 392: Packaging Tool
  - JEP 394: Pattern Matching for instanceof
  - JEP 395: Records
  - JEP 396: Strongly Encapsulate JDK Internals by Default

Java 17 LTS (13/09/2021)

- JDK 17 Documentation
- Java Language Changes for Java SE 17
- Significant Changes in JDK 17 Release
- Lista completa de características - JEP
  - JEP 356: Enhanced Pseudo-Random Number Generators
  - JEP 409: Sealed Classes
  - JEP 403: Strongly Encapsulate JDK Internals

Java 18 (22/03/2022)

- JDK 18 Documentation
- Java Language Changes for Java SE 18
- Significant Changes in JDK 18 Release

- Lista completa de características - JEP
  - JEP 400: UTF-8 by Default
  - JEP 408: Simple Web Server
  - JEP 413: Code Snippets in Java API Documentation

Java 19 (20/09/2022)

- JDK 19 Documentation
- Java Language Changes for Java SE 19
- Significant Changes in JDK 19 Release
- Lista completa de características - JEP

Java 20 LTS (21/03/2023)

- JDK 20 Documentation
- Java Language Changes for Java SE 20
- Significant Changes in JDK 20 Release
- Lista completa de características - JEP

Java 21 (21/09/2023)

- JDK 21 Documentation
- Java Language Changes for Java SE 21
- Significant Changes in JDK 21 Release
- Lista completa de características - JEP
  - JEP 431: Sequenced Collections
  - JEP 440: Record Patterns
  - JEP 441: Pattern Matching for switch
  - JEP 444: Virtual Threads

Java 22 (19/03/2024)

- JDK 22 Documentation
- Java Language Changes for Java SE 22
- Significant Changes in JDK 22 Release
- Lista completa de características - JEP
  - JEP 454: Foreign Function & Memory API
  - JEP 456: Unnamed Variables & Patterns

NEW Java 23 (17/09/2024)

- JDK 23 Documentation
- Java Language Changes for Java SE 23
- Significant Changes in JDK 23 Release
- Lista completa de características - JEP

Referencias

- Java Platform, Standard Edition Documentation
- Last API Documentation
- The Java™ Tutorials
- OpenJDK

- OpenJDK - Github
- This JEP is the index of all JDK Enhancement Proposals, known as JEPs.
- JDK Release Notes
- The Java Version Almanac
- https://developer.oracle.com/languages/java.html
- https://roadmap.sh/java

Licencia