Apuntes - [Java]

Java se basa en la **Programación Orientada a Objetos (POO)**. 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.

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 estructura se organiza sobre el código pero en la programación orientada a objetos el código 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.

Sintaxis básica

- Compilar código Java: *$ javac filename.java*
- Ejecutar código: *$ java filename*
- Start a graphical console to monitor and manage Java applications: *jconsole*

Tools and Commands Reference

```
// Comentarios de una sóla línea

/*
Comentarios multilínea
*/

/**
* Comentarios JavaDoc lucen así. Suelen describir la clase o varios atributos de
una clase.
*/

// Todos los programas importan automáticamente el paquete 'java.lang' que define
la clase 'System'
// Importa la clase 'ArrayList' dentro del paquete 'java.util'
import java.util.ArrayList;
// Importa todas las clases dentro del paquete 'java.security'
import java.security.*;

// Para Java un archivo es una unidad de compilación. Pueden contener una o varias
clases.
// Por convención, el nombre de la clase principal (declarada como public) debe
coincidir con el nombre del archivo que contiene el programa.
public class Sample {

    // Un programa debe tener un método 'main' como punto de entrada
    public static void main (String[] args) {
        // Usa 'System.out.println' para imprimir líneas
        System.out.println("¡Hola mundo!");
        System.out.println(
            " Integer (int): " + 10 +
            " Double (double): " + 3.14 +
            " Boolean (boolean): " + true);
```

```
        // Para imprimir sin el salto de línea, usa 'System.out.print'
        System.out.print("Hola ");
        System.out.print("Mundo");
    }
}
```

Tipos & Variables

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

En Java se declara una variable usando *'<tipo> <nombre>'*. Es necesario declarar la variable antes de poder hacer referencia a ella. A partir de que se declaran se pueden utilizar, y no 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.

```
// [Tipos primitivos]
// ------------------
// [Byte] - Entero complemento a dos con signo de 8-bit (-128 <= byte <= 127)
byte fooByte = 100;

// [Short] - Entero complemento a dos con signo de 16-bit (-32,768 <= short <=
32,767)
short fooShort = 10000;

// [Integer] - Entero complemento a dos con signo de 32-bit (-2,147,483,648 <= int
<= 2,147,483,647)
int fooInt = 1;

// [Long] - Entero complemento a dos con signo de 64-bit
(-9,223,372,036,854,775,808 <= long <= 9,223,372,036,854,775,807)
long fooLong = 100000L;
// 'L' es usado para denotar que el valor de esta variable es del tipo Long;
cualquier cosa sin ella es tratado como un entero por defecto.

// Nota: Java no tiene tipos sin signo

// [Float] - Número de coma flotante IEEE 754 de precisión simple de 32-bit
float fooFloat = 234.5f;
// 'f 'es usado para denotar que el valor de esta variable es del tipo float; de
otra manera es tratado como un double.

// [Double] - Número de coma flotante IEEE 754 de precisión doble de 64-bit
double fooDouble = 123.4;

// [Boolean] - true & false
boolean fooBoolean = true;
```

```
boolean barBoolean = false;

// [Char] - Un simple carácter unicode de 16-bit.
/* Como char es un tipo sin signo de 16 bits, se pueden realizar operaciones
aritméticas. Las constantes de carácter se incluyen entre comillas simples. */
char fooChar = 'A';
fooChar++; // now fooChar == 'B'
```

En Java, un literal es un valor fijo representado en formato legible para los humanos. Por ejemplo, el número 100 es un literal. Los literales también suelen denominarse constantes. De forma predeterminada, 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. Java también admite los literales de cadena. Una cadena es un conjunto de caracteres includos entre comillas dobles.

```
int a = 100;
long b = 100L;
double c = 100.5;
float d = 100.5f;
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
```

Secuencias de escape de caracteres:

- *`\'`* - 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)

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

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

Se usa la palabra clave *`'final'`* para hacer **inmutable** las variables. Por convención el nombre de la variable se declara en mayúsculas:

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

Notación abreviada para declarar (e inicializar) múltiples variables:

```
int x, y, z;
int i1 = 1, i2 = 2;
int a = b = c = 100; // el símbolo '=' retorna el valor de su derecha y por tanto
lo podemos usar para de esta forma.
```

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 pueden empezar por *cualquier letra, guión bajo o $*. El siguiente carácter puede ser *cualquier letra, dígito, guión bajo o $*. Por lo tanto no pueden empezar con un dígito ni emplear palabras clave de Java.

Un bloque de código es un grupo de dos o más instrucciones definidas entre llaves {}. Tras 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 en 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 se destruyen al finalizar el bloque. Además, si se define una variable al inicio de un bloque estará disponible para el código de ese bloque a partir de su definición. Por lo tanto si se define al final no se podrá utilizar.

Los bloques se pueden anidar, de forma que un bloque de código es contenido por otro bloque de código. Desde el bloque interior se pueden acceder a las variables definidas en el bloque exterior pero el exterior no puede acceder a las variables definidas en el bloque interior.

Operadores

```
// 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
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
```

Operadores lógicos

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 el segundo operando sólo se evalúa cuando es necesario.

Cadenas

```
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:

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

Control de flujo

```
/*
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 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 estática

Java admite la importación estática, que tiene la forma *'import static'* y al usarla, se puede hacer referencia directamente a miembros estáticos por sus nombres, sin necesidad de calificarlos con el nombre de su clase.

```
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>];
```

```
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, ...};
```

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

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

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

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

```
System.out.println("Year: " + sample[0]); // => 2015
```

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

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

```
class Bicicleta {

    // Campos o variables de instancia
    public int ritmo; // Public: Puede ser accedido desde cualquier parte
    private int velocidad;  // Private: Accesible sólo desde esta clase
    protected int engranaje; // Protected: Accesible desde esta clases y sus
subclases
    String nombre; // default: Sólo accesible desde este paquete

    // Constructores son la manera de crear clases
    // Este es un constructor por defecto
    public Bicicleta() {
        engranaje = 1;
        ritmo = 50;
        velocidad = 5;
        nombre = "Bontrager";
    }

    // Este es un constructor específico (contiene argumentos)
    public Bicicleta(int ritmoInicial, int velocidadInicial, int engranajeInicial,
String nombre) {
        this(); // llamada al constructor sin parámetros 'Bicicleta()';
        this.engranaje = engranajeInicial;
        this.ritmo = ritmoInicial;
        this.velocidad = velocidadInicial;
        this.nombre = nombre;
    }

    // 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 int getRitmo() {
        return ritmo;
```

```
    }

    // ....

    //Método para mostrar los valores de los atributos de este objeto.
    @Override
    public String toString() {
        return "engranaje: " + engranaje +
                " ritmo: " + ritmo +
                " velocidad: " + velocidad +
                " nombre: " + nombre;
    }
}
```

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 por *'list-args'* se **ejecuta 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; // Ahora 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:

> *<public/private/protected> <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.

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 matriz. 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**.

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:

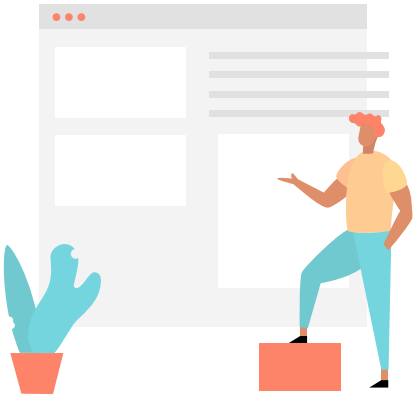
- Sólo pueden invocar directamente otros métodos *'static'*
- Sólo pueden acceder directamente a datos *'static'*
- Carecen de una referencia *'this'*

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 a 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() {
        Vehicle vehicle = new Car(); // Un variable de tipo 'Vehicle' hace
referencia a un objeto de tipo 'Car', que es una subclase de 'Vehicle'
        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 reemplazado en una subclase. Si la subclase no lo reemplaza, 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) -> Una clase sin modificador sólo será visible por otras clases **dentro del mismo paquete**.
- *'public'* -> Una clase pública es **visible desde cualquier lugar**.

NOTA: 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**.

NOTA: 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 por ejemplo *class Car extends Superclass implements Vehicle {}*

Importante: 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 o clases 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) {
        Vehicle car = new Car(); // Al igual que con la herencia, podemos declarar
una variable de referencia de un tipo de interfaz.
        car.getWheels(); // 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.
    }
}
```

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 y 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.

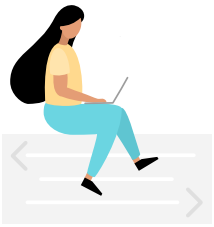
Si una clase hereda de dos interfaces que implementan un método predeterminado con el mismo nombre, la clase está obligada a implementar dicho método ya que si no lo hace el compilador genera un error. La versión implementada en la clase tiene preferencia sobre las versiones implementadas en las interfaces.

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) {
  // controlador para TipoException
}
catch (Tipo2Exception exception) {
  // controlador para Tipo2Exception
}
catch (Exception exception) { // Captura del resto de excepciones no capturadas anteriormente
  // controlador para el resto de excepciones
}
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 execpciones de Java como excepciones personalizadas. Una vez hecho 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 MyExcpetion("Message");
  }
}

class MyException extends Exception { }
```

En JDK 7 se amplió el mecanismo de excepciones al permite 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 necesario. 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' (ya no es necesario invocar a
'close()') */
try(FileInputStream fin = New FileInputStream(args[0])) {
  // código
}
catch (IOException e) {
  // Controlador
}
```

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

```
/* El siguiente código usa un 'try-with-resources' para abrir un archivo y después
cerrarlo automáticamente al salir del bloque 'try' (ya no es necesario invocar a
'close()') */
try(FileInputStream fin = New FileInputStream(args[0]); FileOutputStream fout =
New FileOutputStream(args[1])) {
  // 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'*:

```
// 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);
    }
}
```

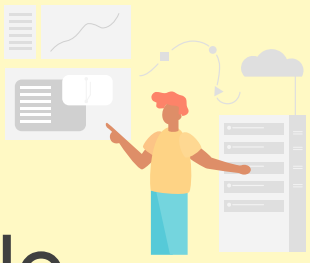
Escribir en un fichero con *FileWriter*

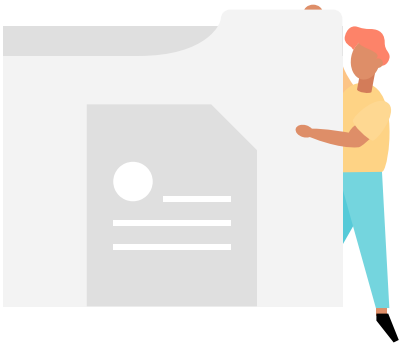
```
// Demonstrate 'FileWriter'.
// This program uses 'try-with-resources'. It requires JDK 7 or later.
import java.io.*;

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*

```
// 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);
        }
    }
}
```

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;

    synchronized int sumArray(int nums[]) { // 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
        // 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.
- Ni puede actuar como superclase de otra clase.

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

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

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

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

- *public static tipo-enum[] values()* => devuelve una matriz 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 {
    CAR(66), TRUCK(12), AIRPLANE(600), BOAT(12); // valores de inicialización. A
destacar el ';' necesario cuando se definen variables, constructores, etc..

    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()); // => 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

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();

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

        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.

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

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. Es decir permite limitar los parámetros de tipo a únicamente tipos numéricos por ejemplo, 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**.

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

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.

```
static <T> void sample( /* lista-params */ ) { /* code... */ }
<T, V> boolean sample( /* lista-params */ ) { /* code... */ }
<T, V extends T> int sample(/* lista-params */ ) { /* code... */ }
static <T extends Comparable<T>, V extends T> boolean sample( /* lista-params */ )
{ /* 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.

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

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

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

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

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

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

Genéricos y código legado

Antes de 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> {
    // code...
}

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 JDK 7 es posible reducir la sintaxis a la hora de crear una instancia de un tipo genérico. La creación de instancias solo usa '<>', una lista vacía de argumentos que indica al compilador que infiera los argumentos de tipo que necesita el constructor. Para código compatible con versiones anteriores a JDK 7 se usa 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 JDK 7 y
posteriores
```

Restricciones y ambigüedad

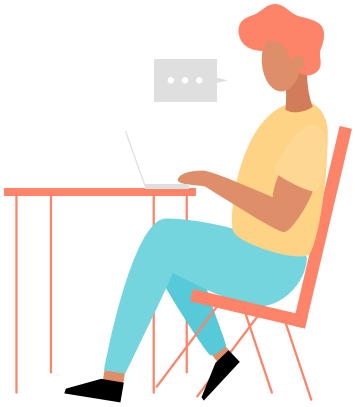
El uso de genéricos puede crear situaciones de ambigüedad, sobretodo en casos de sobrecarga de métodos:

```
class Gen<T, V> {
    // variables de instancia

    // Estos dos métodos se sobrecargan pero dado que T y V pueden ser del mismo
tipo, esto generaria 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 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(); // Error, no se puede crear instancias de un parámetro de
tipo
    }

    static T sample () {} // Error, 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**. Las expresiones lambda también suele denominarse clausuras (*'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 abstracto
}
```

Fundamentos

El nuevo operador para las expresiones lambda se denomina **operador lambda** y tiene la forma `'->'`. 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.

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

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

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.

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

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.

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

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.

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

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

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

            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 JDK 8:

- Sintaxis para métodos *'static'*: *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

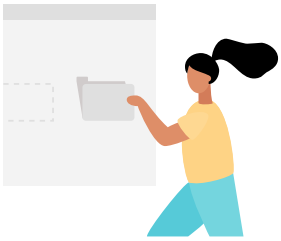
Java proporciona una serie de **interfaces funcionales predefinidas** preparadas para utilizar:

- *`UnaryOperator<T>`* -- Aplica una operación unaria a un objeto de tipo 'T' y devuelve el resultado, que también es de tipo 'T'. Su método es *`'apply()'`*.
- *`BinaryOperator<T>`* -- Aplica una operación a dos objetos de tipo 'T' y devuelve el resultado, que también es de tipo 'T'. su método es *`'apply()'`*.
- *`Consumer<T>`* -- Aplica una operación a un objeto de tipo 'T'. Su método es *`'accept()'`*.
- *`Supplier<T>`* -- Devuelve un objeto de tipo 'T'. Su método es *`'get()'`*.
- *`Function<T, V>`* -- Aplica una operación a un objeto de 'T' y devuelve el resultado como objeto de 'V'. Su método es *`'apply()'`*.
- *`Predicate<T>`* -- Determina si un objeto de tipo 'T' cumple una restricción. Devuelve un valor boolean que indica el resultado. Su método es *`'test()'`*.

```
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));
```

```
    }
}
```

Collections



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 marcos de trabajo 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 los métodos que realizan cálculos útiles, como la búsqueda y clasificación, en objetos que implementan 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 apropiada. En esencia, los algoritmos son funciones reutilizables.

The 'Collection' Interface

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)`, y `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)`, y `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'), métodos que añaden y eliminan un elemento de la colección ('add', 'remove'), y métodos que proporcionan un iterador sobre la colección ('iterator').

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.

Por tanto es recomendable usar iteradores en vez de una construcción *for-each* cuando tengamos que eliminar el elemento actual.

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

The 'Set' Interface

Un *Set* o conjunto es una colección que **no puede contener elementos duplicados**. Modela la abstracción del conjunto matemático. La interfaz *Set* sólo contiene métodos heredados de *Collection* y añade la restricción de que los elementos duplicados están prohibidos.

La interfaz *Set* también añade un contrato más fuerte sobre el comportamiento de las operaciones *equals* y *hashCode*, permitiendo que las instancias de *Set* sean comparadas de forma significativa incluso si sus tipos de implementación difieren. Dos instancias de *Set* son iguales si contienen los mismos elementos.

La plataforma Java contiene tres implementaciones de *Set* de propósito general:

- **HashSet** que almacena sus elementos en una tabla hash, es la mejor implementación; sin embargo, no ofrece garantías en cuanto al orden de iteración.

- **TreeSet** que almacena sus elementos en un árbol *'red-black'*, ordena sus elementos en función de sus valores; es sustancialmente más lento que **HashSet**.
- **LinkedHashSet**: que se implementa como una tabla hash con una lista enlazada que la recorre, ordena sus elementos según el orden en que se insertaron en el conjunto (orden de inserción). Tiene un coste algo más elevado que un **HashSet** pero soluciona el problema del orden.

La interfaz `Set` tiene una subinterface `SortedSet`, que es un `Set` que mantiene sus elementos en orden ascendente, ordenados de acuerdo al orden natural de los elementos o de acuerdo a un `Comparator` proporcionado a la hora de creación del `SortedSet`.

The 'List' Interface

Una `List` es una **colección ordenada que pueden contener elementos duplicados** (a veces llamada secuencia). Además de las operaciones heredadas de `Collection`, la interfaz `List` incluye operaciones para lo siguiente:

- **Acceso por posición** para manipular los elementos de la lista. Esto incluye métodos como `get`, `set`, `add`, `addAll` y `remove`.
- **Búsqueda** de elementos específicos dentro de la lista y la devolución de su posición numérica dentro de ella. Métodos como `indexOf` y `lastIndexOf`.
- Extensión de la **iteración** para obtener ventaja de la naturaleza secuencial de las listas con `listIterator`.
- Operaciones arbitrarias en secciones de la lista con el método `subList`.

La plataforma Java contiene dos implementaciones de `List` de propósito general:

- **ArrayList**, que suele ser la implementación con mejor rendimiento.
- **LinkedList**, que ofrece un mejor rendimiento en determinadas circunstancias.

The 'Queue' Interface

Una `Queue` o cola es una colección que contiene elementos antes de ser procesados. Además de las operaciones básicas de una `Collection`, las colas proporcionan operaciones adicionales de inserción, extracción e inspección.

Una **LinkedList** implementa la interfaz `Queue`. La clase '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
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Las colas ordenan típicamente, aunque no necesariamente, los elementos de una manera **FIFO** (first-in-first-out). Entre las excepciones se encuentran las colas de prioridad, que ordenan los elementos según sus valores.

Cualquiera que sea el orden que se utilice, la cabeza de la `Queue` es el elemento que sería eliminado por una llamada a `remove()` o `poll()`. En una cola FIFO, todos los elementos nuevos se insertan en la cola de la cola. Otros tipos de colas pueden utilizar reglas de colocación diferentes. Cada implementación de cola debe especificar sus propiedades de ordenación.

Es posible que una implementación de `Queue` restrinja el número de elementos que contiene; tales colas se conocen como *bounded*.

El método `add()`, que `Queue` hereda de `Collection`, inserta un elemento a menos que viole las restricciones de capacidad de la cola, en cuyo caso lanza `IllegalStateException`. El método `offer()`, que se utiliza únicamente en colas limitadas (*'bounded'*), difiere de `add()` solo en que devuelve `false` si no se inserta el elemento.

Los métodos `remove()` y `poll()` eliminan y devuelven la cabecera o *'head'* de la cola. Los métodos `remove()` y `poll()` difieren en su comportamiento sólo cuando la cola está vacía. En estas circunstancias, `remove()` lanza `NoSuchElementException`, mientras que `poll()` devuelve nulo.

Los métodos `element()` y `peek()` devuelven, pero no eliminan, la cabecera de la cola.

The 'Deque' Interface

Una `Deque` es una cola de dos extremos. Este tipo de cola es una colección lineal de elementos que soporta la inserción y extracción de elementos en **ambos extremos**.

La interfaz `Deque` es un tipo de datos abstractos más rico que `Stack` y `Queue` porque implementa tanto stacks como colas al mismo tiempo. Clases predefinidas como 'ArrayDeque' y 'LinkedList' implementan la interfaz `Deque`.

La interfaz `Deque` define métodos para acceder a los elementos en ambos extremos de la instancia. Se proporcionan métodos para insertar, quitar y examinar los elementos. Tendremos métodos que lancen una excepción o devuelvan el valor nulo.

Operación	First Element	Last Element
Insert (Exception)	`addFirst(e)`	`addLast(e)`
Insert (boolean)	`offerFirst(e)`	`offerLast(e)`
Remove (Exception)	`removeFirst()`	`removeLast()`
Remove (null)	`pollFirst()`	`pollLast()`
Examine (Exception)	`getFirst()`	`getLast()`
Examine (null)	`peekFirst()`	`peekLast()`

The 'Map' Interface

Un `Map` es un objeto que asigna claves a valores. Un mapa **no puede contener claves duplicadas**. Cada clave puede asignarse a un valor como máximo. Modela la abstracción de la función matemática.

La plataforma Java contiene tres implementaciones de `Map` de propósito general y cuyo comportamiento y rendimiento son análogos a las implementaciones de la interfaz `Set` como son **HashSet**, **TreeSet** y **LinkedHashSet**:

- **HashMap** que almacena sus elementos en una tabla *hash*, es la mejor implementación; sin embargo, no ofrece garantías en cuanto al orden de iteración.
- **TreeMap** que almacena sus elementos en un árbol *'red-black'*, ordena sus elementos en función de sus valores; es sustancialmente más lento que **HashMap**.
- **LinkedHashMap**: que se implementa como una tabla *hash* con una lista enlazada que la recorre, ordena sus elementos según el orden en que se insertaron en el mapa (orden de inserción). Tiene un coste algo más elevado que un **HashMap** pero soluciona el problema del orden.

La interfaz `Map` tiene una subinterface `SortedMap`, que es un `Map` que mantiene sus elementos en orden ascendente, ordenados de acuerdo al orden natural de las claves o de acuerdo a un `Comparator` proporcionado a la hora de creación del `SortedMap`.

Pruebas unitarias con JUnit

Un sistema con test unitarios será más fácil modificarlo ya que tendremos la seguridad de que no vamos a romper nada. Nos permitirá refactorizar el código sin miedo ya que podremos detectar posibles errores y fallos de forma más rápida. El uso de test también nos permite detectar malas prácticas de diseño en fases tempranas del desarrollo, lo que permite su solución dando lugar a software de mejor calidad.

Tipos de pruebas

- **Test unitarios**: prueban una funcionalidad única y se basan en el principio de responsabilidad única (la S de los principios de diseño SOLID)
- **Integración**: prueban la conexión entre componentes, sería el siguiente paso a los test unitarios.
- **Funcionales (o Sistema)**: prueban la integración de todos los componentes que desarrollan una funcionalidad concreta (por ejemplo, la automatización de pruebas con Selenium serían test funcionales).
- **Aceptación de Usuarios**: Pruebas definidas por el *Product Owner* basadas en ejemplos (BDD con Cucumber).
- **Regresión**: Prueban que los test unitarios y funcionales siguen funcionando a lo largo del tiempo (se pueden lanzar tanto de forma manual como en sistemas de Integración Continua).
- **Carga**: Prueban la eficiencia del código.

Características de los tests unitarios

Los tests unitarios deben cumplir los siguientes puntos denominados **Principios FIRST**:

- **Fast**: Rápida ejecución.
- **Isolated**: Independiente de otros test.
- **Repeatable**: Se puede repetir en el tiempo.
- **Self-Validating**: Cada test debe poder validar si es correcto o no a sí mismo.
- **Timely**: ¿Cuándo se deben desarrollar los test? ¿Antes o después de que esté todo implementado? Sabemos que cuesta hacer primero los test y después la implementación

(TDD: Test-driven development), pero es lo suyo para centrarnos en lo que realmente se desea implementar.

Además podemos añadir estos dos puntos más:

- Sólo pruebas de los **métodos públicos** de cada clase.
- No se debe hacer uso de las **dependencias** de la clase a probar. Esto quizás es discutible porque en algunos casos donde la dependencias son clases de utilidades y se puede ser menos estricto. Se recomienda siempre aplicar el sentido común.
- Un test no debe implementar ninguna lógica de negocio (nada de if...else...for...etc)

Framework JUnit4 / JUnit5

JUnit es un framework Java para implementar test en Java. JUnit 5 requiere Java 8 (o superior). JUnit se basa en anotaciones:

- *@Test*: indica que el método que la contiene es un test: expected y timeout.
- *@Before* (JUnit4) / *@BeforeEach* (JUnit5): ejecuta el método que la contiene justo antes de cada test.
- *@After* (JUnit4) / *@AfterEach* (JUnit5): ejecuta el método que la contiene justo después de cada test.
- *@BeforeClass* (JUnit4) / *@BeforeAll* (JUnit5): ejecuta el método (estático) que la contiene justo antes del primer test.
- *@AfterClass* (JUnit4) / *@AfterAll* (JUnit5): ejecuta el método (estático) que la contiene justo después del último test.
- *@Ignore* / *@Disabled*: evita la ejecución del tests. No es muy recomendable su uso porque puede ocultar test fallidos. Si dudamos si el test debe estar o no, quizás borrarlo es la mejor de las decisiones.
- *@DisplayName("cadena")* (JUnit5): Declara un nombre de visualización personalizado para la clase de prueba o el método de prueba. En lugar de usar esta anotación es recomendable utilizar nombres para los métodos lo suficientemente descriptivos como para que no sea necesario usar esta anotación.

```
@DisplayName("Aserciones soportadas")
class StandardTests {

    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @Test
```

```
    void regular_testi_method() {
        // ...
    }

    @Test
    @DisplayName("Verdadero o falso?")
    void regular_testi_method() {
        // ...
    }

    @Test
    void failingTest() {
        fail("a failing test");
    }

    @Test
    @Disabled("este tests no se ejecuta")
    void skippedTest() {
        // not executed
    }

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }
}
```

Las condiciones de aceptación del test se implementa con las aserciones. Las más comunes son los siguientes:

- **assertTrue/assertFalse (condición a testear)**: Comprueba que la condición es cierta o falsa.
- **assertEquals/assertNotEquals (valor esperado, valor obtenido)**: Es importante el orden de los valores esperado y obtenido.
- **assertNull/assertNotNull (object)**: Comprueba que el objeto obtenido es nulo o no.
- **assertSame/assertNotSame(object1, object2)**: Comprueba si dos objetos son iguales o no.
- **fail()**: Fuerza que el test termine con fallo. Se puede indicar un mensaje.

```
class AssertionsTest {

    @Test
    void standardAssertions() {
        assertEquals(2, 2);
        assertEquals(4, 4, "Ahora el mensaje opcional de la aserción es el último
parámetro.");
        assertTrue(2 == 2, () -> "Al usar una lambda para indicar el mensaje, "
```

```
                + "esta se evalúa cuando se va a mostrar (no cuando se ejecuta el
assert), "
                + "de esta manera se evita el tiempo de construir mensajes
complejos innecesariamente.");
    }

    @Test
    void groupedAssertions() {
        // En un grupo de aserciones se ejecutan todas ellas
        // y ser reportan todas los fallos juntos
        assertAll("user",
            () -> assertEquals("Francisco", user.getFirstName()),
            () -> assertEquals("Pérez", user.getLastName())
        );
    }

    @Test
    void exceptionTesting() {
        Throwable exception = expectThrows(IllegalArgumentException.class, () -> {
            throw new IllegalArgumentException("a message");
        });
        assertEquals("a message", exception.getMessage());
    }
}
```

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.

Histórico de versiones

This JEP is the index of all JDK Enhancement Proposals, known as JEPs.

JDK 1.0 (23 de Enero de 1996)

- Primera versión

JDK 1.1 (19 de Febrero de 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 (8 de Diciembre de 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 (8 de Mayo de 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 (6 de Febrero de 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 de Septiembre de 2004)

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

Java SE 6 (11 de Diciembre de 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 (7 de Julio de 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 (18 de Marzo de 2014)

- Lista completa de características
- 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 de Septiembre de 2017)

- Lista completa de características
- JEP 200: The Modular JDK
- JEP 222: 'jshell': The Java Shell (Read-Eval-Print Loop)
- JEP 295: Compilación *Ahead-of-Time*
- JEP 282: Herramienta *jlink* que puede ensamblar y optimizar un conjunto de módulos y sus dependencias en una imagen personalizada en tiempo de ejecución. De manera efectiva, permite producir un ejecutable totalmente utilizable que incluye la JVM para ejecutarlo.
- JEP 266: More Concurrency Updates. Interfaces supporting the Reactive Streams publish-subscribe framework.

- JEP 263: Gráficos HiDPI
- JEP 224: HTML5 Javadoc
- JEP 275: Modular Java Application Packaging

Java 10 (20 de Marzo de 2018)

- Lista completa de características
- JEP 286: Local-Variable Type Inference
- JEP 317: Experimental Java-Based JIT Compiler. This is the integration of the Graal dynamic compiler for the Linux x64 platform
- JEP 310: Application Class-Data Sharing. This allows application classes to be placed in the shared archive to reduce startup and footprint for Java applications
- 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 (25 de Septiembre de 2018)

- Lista completa de características
- 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
- JavaFX, Java EE and CORBA modules have been removed from JDK

Java 12 (19 de Marzo de 2019)

- Lista completa de características
- JEP 189: Shenandoah: A Low-Pause-Time Garbage Collector (Experimental)

- JEP 230: Microbenchmark Suite
- JEP 325: Switch Expressions (Preview)
- JEP 334: JVM Constants API

Reference

- https://docs.oracle.com/javase/tutorial/index.html
- https://docs.oracle.com/javase/tutorial/java/TOC.html
- https://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html
- https://docs.oracle.com/en/java/javase/11/
- http://openjdk.java.net/
- https://en.wikipedia.org/wiki/Java_version_history
- https://www.adictosaltrabajo.com/2016/11/24/primeros-pasos-con-junit-5/
- http://innovationlabs.softtek.co/testing-unitario

License

ANEXO: Effective Java

(todo)

Creating and Destroying Objects

Este capítulo trata de la creación y destrucción de objetos: cuándo y cómo crearlos, cuándo y cómo evitar su creación, cómo asegurar que se destruyan a tiempo y cómo gestionar cualquier acción de limpieza que deba preceder a su destrucción.

Item 1: Consider static factory methods instead of constructors

La forma tradicional de que una clase permita a un cliente obtener una instancia es proporcionar un constructor público. Pero hay otra técnica y es proveer un método público *'static factory'* que es simplemente un **método estático que retorna una instancia** de la clase.

```
// Retorna una instancia de Boolean usando el parámetro de tipo boolean
public static Boolean valueOf(boolean b) {
    return b ? Boolean.TRUE : Boolean.FALSE;
}
```

Hay que tener en cuenta que *'static factory method'* no es lo mismo que el patrón ***'Factory Method'*** de los patrones de diseño *"Design Patterns: Elements of Reusable Object-Oriented Software"*.

No es incompatible que una clase suministre *'static factory methods'* además de constructores públicos.

El uso de estos métodos tiene ventajas:

- **Una ventaja de los *'static factory methods'* es que, a diferencia de los constructores, tienen nombres**. Podemos elegir nombres que sean mucho más descriptivos que los constructores.
- **Una segunda ventaja es que, a diferencia de los constructores, no tienen que crear un nuevo objeto cada vez que se les invoca.** Esto permite clases inmutables que retornen

instancias ya creadas, mejorando el rendimiento ya que podemos evitar la creación de nuevos objetos.

- **Una tercera ventaja es que, a diferencia de los constructores, estos métodos pueden devolver un objeto de cualquier subtipo de su tipo de devolución.** Esto da una gran flexibilidad para elegir la clase del objeto devuelto.
- **Una cuarta ventaja de las fábricas estáticas es que la clase del objeto devuelto puede variar de una llamada a otra en función de los parámetros de entrada.** Se permite cualquier subtipo del tipo de retorno declarado. La clase del objeto devuelto también puede variar de una liberación a otra.
- **Una quinta ventaja de las fábricas estáticas es que la clase del objeto devuelto no necesita existir cuando se escribe la clase que contiene el método.**

Como inconvenientes destacar:

- **La principal limitación de proporcionar sólo métodos estáticos de fábrica es que las clases sin constructores públicos o protegidos no pueden ser heredadas.**
- **Otra limitación es que no es fácil detectar estas factorías en la documentación de la clase.** Esto es debido a como funciona la herramienta de Javadoc. Los constructores aparecen en un lugar destacado a diferencia de los métodos. Normalmente, estos métodos suelen seguir ciertas convenciones:
    - **from**: un método *'type-conversion'* que toma un parámetro y retorna la correspondiente instancia de ese tipo:

```
Date d = Date.from(instant);
```

    - **of**: un método de agregación que toma múltiples parámetros y devuelve una instancia de ese tipo que los incorpora:

```
Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);
```

    - **valueOf**: una forma más descriptiva de *'from'* y *'of'*:

```
BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);
```

    - **instance** or **getInstance**: retorna una instancia que se describe por sus parámetros (si los hay) pero que no puede decirse que tenga el mismo valor:

```
```

```
StackWalker luke = StackWalker.getInstance(options);
```

- **create** or **newInstance**: como el anterior salvo que esta vez de garantiza que en cada llamada se devuelve una nueva instancia

```
Object newArray = Array.newInstance(classObject, arrayLen);
```

- **getType**: como **getInstance**, pero se usa si el método de fábrica está en una clase diferente. *'Type'* es el tipo de objeto devuelto por el método de fábrica:

```
FileStore fs = Files.getFileStore(path);
```

- **newType** como **newInstance**, pero se usa si el método de fábrica está en una clase diferente. *'Type'* es el tipo de objeto devuelto por el método de fábrica:

```
BufferedReader br = Files.newBufferedReader(path);
```

- **type** una alternativa concisa a **getType** y **newType**:

```
List<Complaint> litany = Collections.list(legacyLitany);
```

Item 2: Consider a builder when faced with many constructor parameters

Las factorías estáticas y los constructores comparten una limitación: no se adaptan bien a un gran número de parámetros opcionales.

Tradicionalmente los programadores han usado el patrón *'telescoping constructor'* en el cual se provee a la clase de un constructor con los parámetros requeridos, otro constructor con uno de los parámetros opcionales, otro con dos y así sucesivamente hasta completar la lista y tener un constructor con todos los opcionales. De esta forma cuando se desea crear una instancia, se utiliza el constructor con la lista de parámetros más corta que contiene todos los parámetros que se desean configurar. Los parámetros que no se utilizan se suele pasar como 0, 'null', etc..

```
public class NutritionFacts {
  private final int servingSize;  // (mL) required
  private final int servings;     // (per container) required
  private final int calories;     // (per serving) optional
  private final int fat;          // (g/serving) optional
```

```
  private final int sodium;       // (mg/serving) optional
  private final int carbohydrate; // (g/serving) optional

  public NutritionFacts(int servingSize, int servings) {
    this(servingSize, servings, 0);
  }

  public NutritionFacts(int servingSize, int servings, int calories) {
    this(servingSize, servings, calories, 0);
  }

  public NutritionFacts(int servingSize, int servings, int calories, int fat) {
    this(servingSize, servings, calories, fat, 0);
  }

  public NutritionFacts(int servingSize, int servings, int calories, int fat, int
sodium) {
    this(servingSize, servings, calories, fat, sodium, 0);
  }

  public NutritionFacts(int servingSize, int servings, int calories, int fat, int
sodium, int carbohydrate) {
    this.servingSize = servingSize;
    this.servings = servings;
    this.calories = calories;
    this.fat = fat;
    this.sodium = sodium;
    this.carbohydrate = carbohydrate;
  }
}
```

En resumen, el patrón *'telescoping constructor'* funciona, pero es difícil escribir código cliente cuando hay muchos parámetros, y es más difícil de leer. Además, es propenso a errores ya que cuanto más extensa es la lista de parámetros mayores probabilidades de equivocarse en el orden de los mismos al invocar un constructor. Si los parámetros son del mismo tipo, el compilador no mostrará ningún error.

Otro patrón que permite trabajar con muchos parámetros opcionales en un constructor es el patrón *'JavaBean'*. En este patrón se invoca un constructor sin parámetros para crear un objeto y luego se invocan los metódos ***'setters'*** de cada parámetro tanto requerido como opcional que sea necesario para construir correctamente el objeto:

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

Este patrón es más fácil de leer y mantener pero tiene el inconveniente de que debido a que la construcción se divide en múltiples llamadas, un *'JavaBean'* puede estar en un estado inconsistente a lo largo de su construcción. La clase no tiene la opción de hacer cumplir la consistencia simplemente comprobando la validez de los parámetros del constructor. Intentar usar un objeto cuando está en un estado inconsistente puede causar fallos que están lejos del código que contiene el fallo y por lo tanto son difíciles de depurar.

Afortunadamente, existe una tercera alternativa que combina la seguridad *'telescoping constructor'* con la legibilidad del patrón *'JavaBeans'*. Es una forma del patrón ***'Builder'*** incluido en *"Design Patterns: Elements of Reusable Object-Oriented Software"*.

En lugar de hacer el objeto deseado directamente, el cliente llama a un constructor (o fábrica estática) con todos los parámetros requeridos y consigue un objeto **'Builder'**. Luego el cliente llama a los métodos similares a los ***'setters'*** en el objeto constructor para establecer cada parámetro opcional de interés. Finalmente, el cliente llama a un método ***build()*** sin parámetros para generar el objeto, que es típicamente inmutable.

```
// Builder Pattern
public class NutritionFacts {
  private final int servingSize;
  private final int servings;
  private final int calories;
  private final int fat;
  private final int sodium;
  private final int carbohydrate;

  public static class Builder {
    // Required parameters
    private final int servingSize;
    private final int servings;
    // Optional parameters - initialized to default values
    private int calories = 0;
    private int fat = 0;
    private int sodium = 0;
    private int carbohydrate = 0;

    public Builder(int servingSize, int servings) {
      this.servingSize = servingSize;
      this.servings = servings;
    }

    public Builder calories(int val) {
      calories = val;
      return this;
    }

    public Builder fat(int val) {
```

```
        fat = val;
        return this;
      }

      public Builder sodium(int val) {
        sodium = val;
        return this;
      }

      public Builder carbohydrate(int val) {
        carbohydrate = val;
        return this;
      }

      public NutritionFacts build() {
        return new NutritionFacts(this);
      }
    }

    private NutritionFacts(Builder builder) {
      servingSize = builder.servingSize;
      servings = builder.servings;
      calories = builder.calories;
      fat = builder.fat;
      sodium = builder.sodium;
      carbohydrate = builder.carbohydrate;
    }
  }
```

Este código de cliente es fácil de escribir y, lo que es más importante, fácil de leer. La clase es inmutable, y todos los valores por defecto de los parámetros están en un solo lugar. Los métodos *'set'* del 'Builder' devuelven al constructor mismo (con *return this*) para que las invocaciones puedan ser encadenadas, resultando en una API fluida. Para detectar parámetros no válidos lo antes posible, podemos verificar la validez de los parámetros en el constructor y los métodos del constructor:

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240,
8).calories(100).sodium(35).carbohydrate(27).build();
```

El patrón *'Builder'* simula los parámetros opcionales con nombre que se encuentran en Python, Kotlin o Scala.

Item 3: Enforce the singleton property with a private constructor or an enum type

Una clase *singleton* es simplemente una clase que se instancia exactamente una vez. Los *'singletons'* normalmente representan un objeto sin estado, como una función o un componente del sistema que es intrínsecamente único. Hacer que una clase sea un *'singleton'* puede dificultar la prueba de sus clientes porque es imposible sustituir una implementación simulada por un *'singleton'* a menos que implemente una interfaz que sirva como su tipo.

Hay dos formas comunes de implementar *'singletons'*. Ambos se basan en mantener el constructor privado y exportar un miembro estático público para proporcionar acceso a la única instancia.

En primer lugar, hacer que la variable miembro sea un campo final:

```
// Singleton with public final field
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }
    public void leaveTheBuilding() { ... }
}
```

El constructor privado es llamado una única vez para inicializar el campo públic, estático y final ***`Elvis.instance`***. En teoría sólo habrá un único ***`Elvis`*** aunque mediante reflexión, un cliente con suficientes privilegios podría invocar al método privado haciéndolo accesible. Para evitar esto, hay que modificar el constructor para que lance una excepción si se intenta crear una segunda instancia.

La principal ventaja del enfoque de campo público es que la API deja claro que la clase es una clase *'singleton'*: el campo estático público es final, por lo que siempre contendrá la misma referencia de objeto. La segunda ventaja es que es más simple.

Una segunda forma es hacer que el miembro público sea un método *'static factory'*:

```
// Singleton with static factory
public class Elvis {
    private static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }
    public static Elvis getInstance() { return INSTANCE; }
    public void leaveTheBuilding() { ... }
}
```

Todas las llamadas a ***`Elvis.getInstance()`*** devuelven la misma referencia de objeto, y nunca se creará ninguna otra instancia de Elvis (con el mismo problema mencionado anteriormente).

Una de las ventajas de este enfoque es que brinda la flexibilidad de cambiar de opinión sobre si la clase es un singleton sin cambiar su API. El método *'static factory'* devuelve la única instancia, pero podría modificarse para devolver, por ejemplo, una instancia separada para cada hilo que lo

invoque. Una segunda ventaja es que puede escribir una fábrica de singleton genérica si su aplicación lo requiere.

A menos que una de estas ventajas sea relevante, **el primer enfoque de campo público es preferible al segundo enfoque**.

Una tercera forma de implementar una clase *'singleton'* es declarar una enumeración de un solo elemento. Ese enfoque es parecido al enfoque de campo público sin el inconveniente del problema de la reflexión. Es un enfoque más conciso y directo pero es también un enfoque poco natural. Un tipo de enumeración de un solo elemento es a menudo la mejor manera de implementar un *'singleton'*. Tenga en cuenta que no puede usar este enfoque si su *'singleton'* debe extender una superclase que no sea `Enum`.

Item 4: Enforce noninstantiability with a private constructor

Ocasionalmente, querrá escribir una clase que sea solo una agrupación de métodos estáticos y campos estáticos. Estas clases han adquirido una mala reputación debido a que algunas personas abusan de ellas para evitar pensar en términos de objetos, pero tienen usos válidos.

Se pueden utilizar para agrupar métodos relacionados en valores primitivos o arrays como en `java.lang.Math` o `java.util.Arrays`. También se pueden usar para agrupar métodos estáticos, incluidas factorías estáticas, para objetos que implementan alguna interfaz como en `java.util.Collections`. Por último, estas clases se pueden usar para agrupar métodos en una clase final, ya que no se pueden colocar en una subclase.

Tales clases de utilidad no fueron diseñadas para ser instanciadas: una instancia no tendría sentido. Sin embargo, en ausencia de constructores explícitos, el compilador proporciona un constructor público, sin parámetros y predeterminado. Para un usuario, este constructor es indistinguible de cualquier otro.

Intentar imponer la no instabilidad haciendo que la clase sea abstracta no funciona. Se podría instanciar una subclase. Además, induce a error al usuario al pensar que la clase fue diseñada para herencia.

Existe, sin embargo, un modismo simple para garantizar la no instanciación. Un constructor predeterminado se genera solo si una clase no contiene constructores explícitos, por lo que se puede hacer que una clase no sea instanciable al incluir un constructor privado:

```
// Noninstantiable utility class
public class UtilityClass {
    // Suppress default constructor for noninstantiability
    private UtilityClass() {
        throw new AssertionError();
    }
    // ....
}
```

Debido a que el constructor explícito es privado, es inaccesible fuera de la clase. El `AssertionError` no se requiere estrictamente, pero proporciona un mecanismo seguro en caso de que el constructor sea invocado accidentalmente desde dentro de la clase. Garantiza que la clase nunca será instanciada bajo ninguna circunstancia.

Este modismo es ligeramente contrario a la intuición porque el constructor se proporciona expresamente para que no se pueda invocar. Por lo tanto, es aconsejable incluir un comentario, como se mostró en el ejemplo.

Como efecto secundario, este modismo también evita que la clase sea heredada. Todos los constructores deben invocar un constructor de superclase, explícita o implícitamente, y una subclase no tendría un constructor de superclase accesible para invocar.