MANUAL DE JAVA

Manual de Java

Contenido:

Introducción a Java

Java es un lenguaje de programación de propósito general, tipado, orientado a objetos,… que permite el desarrollo desde aplicaciones básicas, pasando por aplicaciones empresariales hasta aplicaciones móviles.

Java nacía como un lenguaje de programación que pudiese ser multiplataforma y multidispositivo, bajo el paradigma *"Write Once Run Anywhere" (WORA)*

De esta forma un programa Java escrito una vez podemos ejecutarlo sobre diferentes plataformas, siendo soportados los sistemas operativos Windows, MacOs y UNIX. Y a su vez en diferentes tipos de dispositivos.

Para poder seguir este paradigma la compilación de un programa Java no genera código fuente, sino que genera bytecodes. Estos bytecodes son interpretados por una máquina virtual o JVM (Java Virtual Machine). Dicha máquina ya está escrita para cada uno de los sistemas operativos en cuestión.

Características del lenguaje Java

Dentro de las características del lenguaje Java encontramos:

Independiente de Plataforma

Cuando compilamos código fuente Java no se genera código máquina específico, si no que se genera bytecodes, los cuales son interpretados por la Java Virtual Machine (JVM), posibilitando que un mismo código fuente pueda ser ejecutado en múltiples plataformas.

Orientado a Objetos

Cualquier elemento del lenguaje Java es un objeto. Dentro de los objetos se encapsulan los datos, los cuales son accedidos mediante métodos.

Sencillo

Java está enfocado para ser un lenguaje fácil de aprender. Simplemente se deberán de entender los conceptos básicos de la programación orientada a objetos (POO).

Seguro

Es seguro ya que los programas se ejecutan dentro de la Java Virtual Machine (JVM) en un formato de *"caja de arena"*, de tal manera que no pueden acceder a nada que esté fuera de ella.

Tiene una validación sobre los bytecodes para comprobar que no hay códigos de fragmento ilegal.

Arquitectura Neutral

Independientemente de que se ejecute en una arquitectura de 32 bits o de 64 bits. En Java los tipos de datos siempre ocupan lo mismo.

Portable

Java no tiene nada que dependa de la plataforma, lo cual le hace que sea portable a diferentes plataformas.

Robusto

El lenguaje Java intenta controlar las situaciones de error en los procesos de compilación y de ejecución, reduciendo de esta manera el riesgo de fallo.

Además Java realiza el control total de la memoria alocándola y retirandola mediante un garbage colletor, de tal manera que no podemos utilizar punteros para acceder a ella.

Multi-hilo

Java nos permite la programación concurrente, de tal manera que un único programa puede abrir diferentes hilos de ejecución.

Interpretado

Los bytecodes son interpretados en tiempo real a código máquina.

Alto Rendimiento

Java ofrece compiladores Just-In-Time que permiten tener un alto rendimiento.

Distribuido

El lenguaje Java está pensado para ser ejecutado en arquitecturas distribuidas, como pueda ser Internet.

Variables en Java

¿Qué son las variables en Java?

Las variables Java son un espacio de memoria en el que guardamos un determinado valor (o dato). Para definir una variable seguiremos la estructura:

```
[privacidad] tipo_variable identificador;
```

Java es un lenguaje de tipado estático. Por lo cual todas las variables tendrán un tipo de dato (ya sea un tipo de dato primitivo o una clase) y un nombre de identificador.

El tipo de dato se asignará a la hora de definir la variable. Además, en el caso de que las variables sean propiedades de objetos tendrán una privacidad.

Ejemplos de variables serían…

```
int numero = 2;

String cadena = "Hola";

long decimal = 2.4;

boolean flag = true;
```

Las variables son utilizadas como propiedades dentro de los objetos.

```
class Triangulo {

    private long base;

    private long altura;

}
```

Tipos de variables en Java

Dentro de Java podemos encontrar los siguientes tipos de variables:

- ***Variables de instancia (campos no estáticos), son las variables que están definidas dentro de un objeto pero que no tienen un modificador de estáticas (static). Suelen llevar un modificador de visibilidad (public, private, protected) definiéndose.***

```
class Triangulo {

    private long base;

    private long altura;

}
```

- ***Variables de clase (campos estáticos), son aquellas variables que están precedidas del modificador static. Esto indica que solo hay una instancia de dicha variable. Es decir, aunque tengamos N objetos de la clase, la variable estática solo se instancia una vez.***

```
class Triangulo {

    static long lados = 3;

}
```

Si además queremos que el valor no pueda cambiar nunca la definiremos como final.

```
class Matematicas {

    final static long PI = 3.14159;

}
```

- ***Variables locales, son variables temporales cuyo ámbito de visibilidad es el método sobre el que están definidas. No pueden ser accedidas desde otra parte del código. Se las distingue de las variables de instancia ya que estas no llevan modificadores de visibilidad delante.***

```
int variable = 2;
```

- ***Parámetros, son las variables recibidas como parámetros de los métodos. Su visibilidad será el código que contenga dicho método.***

```
public Triangulo(long base, long altura){...}
```

Clase String: Representando una cadena

Una cadena de texto no deja de ser más que la sucesión de un conjunto de caracteres alfanuméricos, signos de puntuación y espacios en blanco con más o menos sentido.

Podemos encontrarnos desde la archiconocida cadena "Hola Mundo" y la no menos "Mi primera cadena de texto", pasando por las cadenas de texto personalizadas "Víctor", "Víctor Cuervo", las cadenas de depuración "¿Aquí?", "Paso 1", "Paso 2",… hasta las inclasificables "asdf".

Todas ellas serán representadas en java con la clase `String` y `StringBuffer`. Aunque de momento nos centraremos en la primera.

Para encontrar la clase `String` dentro de las librerías de Java tendremos que ir a `java.lang.String`.

Creando una cadena

Para crear una cadena tenemos dos opciones:

- *Instanciamos la clase `String`. Que sería una creación explicita de la clase*

```
String sMiCadena = new String("Cadena de Texto");
```

- *Crear implícitamente la cadena de texto. Es decir, simplemente le asignamos el valor al objeto.*

```
String sMiCadena = "Cadena de Texto";
```

En este caso, Java, creará un objeto `String` para tratar esta cadena.

Crear una cadena vacía

Podemos tener la necesidad de crear una cadena vacía. Puede darse el caso que no siempre sepamos lo que vamos a poner de antemano en la cadena de texto. ¿A quién no le surgen dudas? ;-) … Fuera de bromas, muchas veces la cadena de texto nos la proporcionará el usuario, otro sistema,….

Para poder crear la cadena vacía nos bastará con asignar el valor de "", o bien, utilizar el constructor vacío.

```
String sMiCadena = "";
```

```
String sMiCadena = new String();
```

Constructores String

Visto lo visto podemos resumir que tenemos dos tipos de constructores principales de la clase `String`:

- *String(), que construirá un objeto `String` sin inicializar.*
- *String(String original), construye una clase `String` con otra clase `String` que recibirá como argumento.*

Aunque tenemos algunos más que iremos viendo….

Volcando una cadena de texto a la consola

Solo nos quedará saber cómo volcar una cadena por pantalla. Esto lo haremos con la clase `System.out.println` que recibirá como parámetro el objeto `String`.

Por ejemplo:

```
System.out.println("Mi Cadena de Texto");ó
```

```
String sMiCadena = new String("Mi Cadena de Texto");
```

```
System.out.println(sMiCadena);
```

Nombres de las variables Java

Cuando vayamos a dar un nombre a una variable deberemos de tener en cuenta una serie de normas. Es decir, no podemos poner el nombre que nos dé la gana a una variable.

Los identificadores son secuencias de texto unicode, sensibles a mayúsculas cuya primer carácter solo puede ser una letra, número, símbolo dolar $ o subrayado _ . Si bien es verdad que el símbolo dolar no es utilizado por convención.

Es recomendable que los nombres de los identificadores sean legibles y no acrónimos que no podamos leer. De tal manera que a la hora de verlos se auto-documenten por sí mismos. Además estos identificadores nunca podrán coincidir con las palabras reservadas.

Algunas reglas no escritas, pero que se han asumido por convención son:

- *Los identificadores siempre se escriben en minúsculas. (pe. nombre). Y si son dos o más palabras, el inicio de cada siguiente palabra se escriba en mayúsculas (pe. nombrePersona)*
- *Si el identificador implica que sea una constante. Es decir que hayamos utilizado los modificadores final static, dicho nombre se suele escribir en mayúsculas (pe. LETRA). Y si la constante está compuesta de dos palabras, estas se separan con un subrayado (pe. LETRA_PI).*

Operadores de Asignación y Aritméticos Java

Operador de Asignación

El operador Java más sencillo es el operador de asignación. Mediante este operador se asigna un valor a una variable. El operador de asignación es el símbolo igual.

La estructura del operador de asignación es:

```
variable = valor;
```

Así podemos asignar valores a variables de tipo entero, cadena,…

```
int numero = 3;
```

```
String cadena = "Hola Mundo";
```

```
double decimal = 4.5;
```

```
boolean verdad = true;
```

Operadores Aritméticos

Los operadores aritméticos en Java son los operadores que nos permiten realizar operaciones matemáticas: suma, resta, multiplicación, división y resto.

Los operadores aritméticos en Java son:

Operador	Descripción
+	Operador de Suma. Concatena cadenas para la suma de String
-	Operador de Resta
*	Operador de Multiplicación
/	Operador de División
%	Operador de Resto

Los operadores aritméticos en Java los utilizaremos entre dos literales o variables y el resultado, normalmente lo asignaremos a una variable o bien lo evaluamos.

```
variable = (valor1|variable1) operador (valor2|variable2);
```

Así podemos tener los siguientes usos en el caso de que queramos asignar su valor.

```
suma = 3 + 7;              // Retorna 10
resta = 5 - 2;             // Retorna 3
multiplicacion = 3 * 2;    // Retorna 6
division = 4 / 2;          // Retorna 2
resto = 5 % 3;             // Retorna 2
```

Ten en cuenta que pueden ser valores o variables:

```
suma = vble1 + 3;    // Sumamos 3 al valor de la variable vble1

resta = vble1 - 4;   // Restamos 4 al valor de la variable vble1

...
```

O podríamos utilizarlo en una condición

```
if (variable > suma + 3) { ... }
```

En este caso no asignamos el resultado de la suma a una variable, solo lo evaluamos.

Operadores Igualdad y Relacionales en Java

Los operadores de igualdad y relacionales en Java son aquellos que nos permiten comparar el contenido de una variable contra otra atendiendo a si son variables con un valor igual o distinto o bien si los valores son mayores o menores.

El listado de operadores de igualdad y relacionales en Java es:

Operador	Descripción
==	igual a
!=	no igual a
>	mayor que
>=	mayor o igual que
<	menor que
<=	menor o igual que

Operadores de Igualdad

Mediante los operadores de igualdad podemos comprobar si dos valores son iguales (operador ==) o diferentes (operador !=).

La estructura de los operadores de igualdad es la siguiente:

```
vble1 == vble2
```

```
vble1 != vble2
```

Podemos utilizar estos operadores de igualdad en Java de la siguiente forma:

```
int vble1 = 5;//Se crea una variable vble1 y se le asigna el valor de 5
```

```
int vble2 = 3;//Se crea una variable vble2 y se le asigna el valor de 3
```

```
if (vble1 == vble2)
  System.out.println("Las variables son iguales");
```

```
if (vble1 != vble2)
  System.out.println("Las variables son distintas");
```

Operadores relacionales

Permiten comprobar si un valor es mayor que (operador >), menor que (operador <), mayor o igual que (>=) y menor o igual que (<=).

Al final el operador lo valida entre dos valores o variables con la estructura:

```
vble1 > vble2
```

```
vble1 < vble2
```

```
vble1 >= vble2
```

```
vble1 <= vble2
```

De esta forma podemos tener un código fuente que nos ayude a realizar estas validaciones de relación:

```
int vble1 = 5;

int vble2 = 3;


if (vble1 > vble2)

  System.out.println("La variable 1 es mayor que la variable 2");


if (vble1 < vble2)

  System.out.println("La variable 1 es menor que la variable 2");


if (vble1 >= vble2)

  System.out.println("La variable 1 es mayor o igual que la variable 2");


if (vble1 <= vble2)

  System.out.println("La variable 1 es menor o igual que la variable 2");
```

Operadores Condicionales Java

Los operadores condicionales en Java son aquellos que evalúan dos expresiones booleanas.

Dentro de los operadores condicionales en Java tenemos:

Operador	Descripción
&&	Operador condicional AND
||	Operador condicional OR

Operadores Condicionales

La estructura de los operadores condicionales en Java es:

```
(expresion_booleana1 && expresion_booleana2)
```

```
(expresion_booleana1 || expresion_booleana2)
```

En el caso del operador condicional AND el resultado será true siempre y cuando las dos expresiones evaluadas sean true. Si una de las expresiones es false el resultado de la expresión condicional AND será false.

Para el operador condicional OR el resultado será true siempre que alguna de las dos expresiones sea true.

Los operadores booleanos funcionan mediante la evaluación por cortocircuito. Es decir, que dependiendo del valor de la expresión 1 puede que no sea necesario evaluar la expresión 2.

Para el caso del operador condicional AND, si la primera expresión es false ya devuelve false sin evaluar la segunda expresión. Y en el caso del operador condicional OR si la primera expresión es true ya devuelve true sin evaluar la segunda expresión.

Podríamos ver el uso de los operadores condicionales en el siguiente código:

```
int vble1 = 5;
```

```
int vble2 = 3;
```

```
if ((vble1 == 5) && (vble2 ==3))
```

```
  System.out.println("Las dos variables mantienen sus valores iniciales");
```

```
if ((vble1 == 5) || (vble2 ==3))
```

```
  System.out.println("Al menos una variable mantiene su valor inicial");
```

Sentencias Control en Java

Un programa en Java se ejecuta en orden desde la primera sentencia hasta la última.

Si bien existen las sentencias de control de flujo las cuales permiten alterar el flujo de ejecución para tomar decisiones o repetir sentencias.

Dentro de las sentencias de control de flujo tenemos las siguientes:

- ***Sentencias de decisión***
- ***Sentencias de bucle***
- ***Sentencias de ramificación***

Sentencias de Decisión

Las sentencias de decisión son sentencias que nos permiten tomar una decisión para poder ejecutar un bloque de sentencias u otro.

Las sentencias de decisión son: `if-then-else` y `switch`.

if-then-else

La estructura de las sentencias `if-then-else es`:

```
if (expresion) {
  // Bloque then
} else {
  // Bloque else
}
```

Se evalua la expresión indicada en la sentencia `if`. En el caso de que la expresión sea `true` se ejecutará el bloque de sentencias `then` y en el caso de que la expresión sea `false` se ejecutará el bloque de sentencias `else`.

La parte del `else` no tiene por qué existir. En este caso tendríamos una sentencia `if-then`.

```
if (expresion) {
  // Bloque then
}
```

De esta forma podríamos tener el siguiente código fuente:

```
int valor = 4;


if (valor < 10) {
  System.out.println("El número es menor de 10");
} else {
  System.out.println("El número es mayor de 10");
}
```

Las sentencias `if-then-else` pueden estar anidadas y así nos encontraríamos con una sentencia if-then-elseif, la cual tendría la siguiente estructura:

```
if (expresion) {
  // Bloque then
} else if {
  // Bloque else
} else if {
  // Bloque else
} else if {
  // Bloque else
} ...
```

De esta forma podemos tener el siguiente código:

```
int valor = 14;


if (valor < 10) {
  System.out.println("El valor es una unidad");
} else if (valor < 100) {
  System.out.println("El valor es una decena");
} else if (valor < 1000) {
```

```
  System.out.println("El valor es una centena");
} else if (valor < 10000) {
  System.out.println("El valor es un millar");
} else {
  System.out.println("Es un número grande");
}
```

switch

Para los casos en los que se tienen muchas ramas o caminos de ejecución en una sentencia `if` tenemos la sentencia `switch`. La sentencia `switch` evalúa una expresión y ejecutará el bloque de sentencias que coincida con el valor de la expresión.

El valor de la expresión tiene que ser numérico. Aunque a partir de Java SE 7 ya se pueden utilizar expresiones cuya evaluación sean cadenas.

La estructura de la sentencia switch es:

```
switch (expresion) {
  case valor1:
    bloque1;
    break;
  case valor2:
    bloque2;
    break;
  case valor3:
    bloque3;
    break;
  ...
  default:
      bloque_por_defecto;
}
```

Es importante ver que se utiliza la sentencia break. La sentencia `break` hace que se salga de la sentencia `switch` y por lo tanto no se evalúe el resto de sentencias. Por lo tanto su uso es obligatorio al final de cada uno de los bloques.

Un ejemplo claro en el que podemos utilizar la sentencia `switch` es para evaluar el valor de un mes en numérico y convertirlo a cadena. Este código quedaría de la siguiente forma:

```
int iMes = 3;

String sMes;


switch (iMes) {
    case 1:
        sMes = "Enero";
        break;
    case 2:
        sMes = "Febrero";
        break;
    case 3:
        sMes = "Marzo";
        break;
    case 4:
        sMes = "Abril";
        break;
    case 5:
        sMes = "Mayo";
        break;
    case 6:
        sMes = "Junio";
        break;
    case 7:
        sMes = "Julio";
```

```
            break;
        case 8:
            sMes = "Agosto";
            break;
        case 9:
            sMes = "Septiembre";
            break;
        case 10:
            sMes = "Octubre";
            break;
        case 11:
            sMes = "Noviembre";
            break;
        case 12:
            sMes = "Diciembre";
            break;
        default:
            sMes = "Mes incorrecto";
}


System.out.println(sMes);
```

Este mismo modelo lo podríamos haber implementado mediante una estructura `if-then-else`. Si bien, como podemos ver en el código queda más complejo

```
if (iMes == 1){
    sMes = "Enero";
} else if (iMes == 2) {
    sMes = "Febrero";
} else if (iMes == 3) {
    sMes = "Marzo";
```

```
} else if (iMes == 4) {
    sMes = "Abril";
} else if (iMes == 5) {
    sMes = "Mayo";
} else if (iMes == 6) {
    sMes = "Junio";
} else if (iMes == 7) {
    sMes = "Julio";
} else if (iMes == 8) {
    sMes = "Agosto";
} else if (iMes == 9) {
    sMes = "Septiembre";
} else if (iMes == 10) {
    sMes = "Octubre";
} else if (iMes == 11) {
    sMes = "Noviembre";
} else if (iMes == 12) {
    sMes = "Diciembre";
} else {
    sMes = "Mes incorrecto";
}

System.out.println(sMes);
```

Otra cosa que tenemos que saber de la sentencia `switch` es que las evaluaciones case pueden ser múltiples. La estructura en este caso sería:

```
switch (expresion) {
  case valor1: case valor2: case valor3:
    bloque1;
```

```
    break;
  case valor4: case valor5: case valor6:
    bloque2;
    break;
  ...
  default:
    bloque_por_defecto;
}
```

Esto podemos utilizarlo para saber los días del mes. El código sería el siguiente:

```
int iMes = 3;
String sDias;

switch (iMes) {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        sDias = "El mes tiene 31 días";
        break;
    case 4: case 6: case 9: case 11:
        sDias = "El mes tiene 30 días";
        break;
    case 2:
        sDias = "El mes tiene 28 días (o 29 días si es año bisiesto)";
        break;
    default:
        sDias = "Mes incorrecto";
}
```

Como vemos tenemos diferentes evaluaciones con la sentencia `case`.

Sentencias de Bucle

Las sentencias de bucle nos van a permitir ejecutar un bloque de sentencias tantas veces como queramos, o tantas veces como se cumpla una condición.

En el momento que se cumpla esta condición será cuando salgamos del bucle.

Las sentencias de bucle en Java son: `while`, `do-while` y `for`.

En el caso de la sentencia `while` tenemos un bucle que se ejecuta mientas se cumple la condición, pero puede que no se llegue a ejecutar nunca, si no se cumple la condición la primera vez.

```
while (expresion) {

  bloque_sentencias;

}
```

Por otro lado, si utilizamos `do-while`, lo que vamos a conseguir es que el bloque de sentencias se ejecute, al menos, una vez.

```
do {

  bloque_sentencias;

} while (expresion)
```

La sentencia `for` nos permite escribir toda la estructura del bucle de una forma más acotada. Si bien, su cometido es el mismo.

```
for (sentencias_inicio;expresion;incremento) {

  bloque_sentencias;}
```

Las sentencias de bucle nos van a permitir ejecutar un bloque de sentencias tantas veces como queramos, o tantas veces como se cumpla una condición.

Las sentencias de bucle en [Java][ManualJava] son: `while`, `do-while` y `for`.

while

La estructura repetitiva `while` realiza una primera evaluación antes de ejecutar el bloque. Si la expresión es `true` pasa a ejecutar de forma repetida el bloque de sentencias.

Cada vez que termina de ejecutar el bloque de sentencias vuelve a evaluar la expresión. Si la expresión sigue siendo `true` vuelve a ejecutar el bloque. En el caso de que la expresión sea `false` se saldrá del bucle.

Es por ello que dentro del bloque de sentencias deberán de existir sentencias que modifiquen la evaluación de la expresión, ya que de no hacerse se podría entrar en un bucle infinito.

La estructura de la sentencia `while` es la siguiente:

```
while (expresion) {
  bloque_sentencias;
}
```

Los casos de uso de una sentencia repetitiva `while` son variados, pero principalmente se utiliza para recorrer estructuras de datos o tener contadores.

Por ejemplo podemos realizar un contador de 1 a 10 de la siguiente forma:

```
int contador = 1;
while (contador <= 10) {
  System.out.println(contador);
  contador++;
}
```

do-while

En el caso de la estructura repetitiva `do-while` el funcionamiento es el mismo que el de `while`. Pero con una diferencia, primero se ejecuta el bloque de sentencias y luego se evalua la expresión. Por lo tanto siempre se ejecutará, al menos una vez, el bloque de sentencias.

La estructura de la sentencia `do-while` es:

```
do {
  bloque_sentencias;
} while (expresion)
```

Al igual que anteriormente, en el bloque de sentencias deberemos de modificar alguna de las condiciones de la expresión para poder salir del bucle.

Un ejemplo claro del bucle `do-while` sería el ejemplo en el que le pedimos al usuario que introduzca números por teclado, los cuales mostraremos en forma de eco por pantalla, hasta que introduzca el cero. En ese caso saldremos del bucle.

Utilizaremos la estructura `do-while` en vez de la `while` ya que al menos vamos a pedirle al usuario un número.

El código sería el siguiente:

```
Scanner reader = new Scanner(System.in);
int iNumero;

do {
    System.out.println("Introduce carácter por consola");
    iNumero = reader.nextInt();
    System.out.println(iNumero);
} while (iNumero <> 0);
```

En el caso de haberlo realizado con un bucle `while` tendríamos que repetir la captura y salida de datos. Veamos como quedaría para que puedas ver las diferencias.

```
Scanner reader = new Scanner(System.in);
int iNumero;

System.out.println("Introduce carácter por consola");
iNumero = reader.nextInt();
System.out.println(iNumero);
```

```
while (iNumero <> 0) {
  System.out.println("Introduce carácter por consola");
  iNumero = reader.nextInt();
  System.out.println(iNumero);
}
```

for

Otra de las sentencias repetitivas que tenemos, a parte de los bucles `while` y `do-while`, es la sentencia `for`.

La sentencia `for` tiene la característica de que tiene bien definido el inicio del bloque, la evaluación de la expresión, el incremento de valor y el bloque de sentencias.

La estructura del bucle `for` es:

```
for (sentencias_inicio;expresion;incremento) {
  bloque_sentencias;
}
```

Tanto las sentencias_inicio, expresión como incremento son opcionales y pueden estar o no. Aunque normalmente aparecerán en la estructura.

Esta estructura la podríamos reproducir mediante una sentencia while de la siguiente forma:

```
sentencias_inicio;
while (expresion) {
  bloque_sentencias;
  incremento;
}
```

Las funcionalidades en las que utilizaremos la sentencia `for` serán las mismas que las sentencias `while` y `do-while`, que serán contadores, recorrer estructuras,…

Si queremos definir un contador de 1 a 10 mediante una sentencia `for` utilizaremos el siguiente código:

```
for (int x=1;x<=10;x++=) {
  System.out.println("Valor del contador: " + x);
}
```

En pantalla obtendremos el siguiente resultado:

```
Valor del contador: 1
Valor del contador: 2
Valor del contador: 3
Valor del contador: 4
Valor del contador: 5
Valor del contador: 6
Valor del contador: 7
Valor del contador: 8
Valor del contador: 9
Valor del contador: 10
```

Sentencias de ramificación

Las sentencias de ramificación son aquellas que nos permiten romper con la ejecución lineal de un programa.

El programa se va ejecutando de forma lineal, sentencia a sentencia. Si queremos romper esta linealidad tenemos las sentencias de ramificación.

Las sentencias de ramificación en Java son: `break` y `continue`.

En el caso de `break` nos sirve para salir de bloque de sentencias, mientras que `continue` sirve para ir directamente al siguiente bloque.

break

Ya vimos que en la sentencia selectiva `switch` se utilizaba la sentencia `break` para salir de las evaluaciones y así solo ejecutar el bloque de la opción correspondiente. Si bien podemos utilizar la sentencia `break` con las sentencias repetitivas `while`, `do-while` y `for`. Esta es la que se conoce como sentencia break sin etiquetar.

Cuando utilicemos el `break` dentro de uno de estos bucles lo que se conseguirá es salirse de la ejecución del bucle hasta el siguiente bloque de sentencias. Mismo efecto que si la expresión de evaluación hubiese dado `false`.

Así podremos encontrarnos códigos como el siguiente:

```
while (expresion) {
  sentencia(s);
  break;
  sentencias(s);
}
```

Al ejecutar la sentencia `break` ya no ejecutaremos las sentencias que vayan después.

El uso del `break` dentro de estructuras repetitivas suele aparecer cuando estamos realizando la búsqueda de un elemento por una estructura de datos y lo hemos encontrado.

Por ejemplo, si tenemos un array y queremos buscar un número dentro del array podríamos tener el siguiente código:

```
int[] numeros = {12,3,4,5,6,7,9,10};
int posicion = 0;
boolean encontrado = false;

while (posicion < numeros.length) {
  if (numeros[posicion] == 5) {
        encontrado = true;
        break;
  }
```

```
  posicion++;
}

if (encontrado) {
    System.out.println("El número está en la posición: " + posicion);
} else {
    System.out.println("Número no encontrado");
}
```

Las sentencias `break` se pueden cambiar por variables bandera. Estas variables bandera actúan como cortocircuitos de las expresiones de validación y hacen que salgamos de los bucles.

En este caso podríamos haber utilizado la variable "encontrado" como variable bandera. Y podríamos reescribir el código de la siguiente forma:

```
int[] numeros = {12,3,4,5,6,7,9,10};
int posicion = -1;
boolean encontrado = false;

while ((!encontrado) && (posicion&lt;numeros.length)) {
  posicion++;
  if (numeros[posicion] == 5) {
        encontrado = true;
  }
}

if (encontrado) {
    System.out.println("El número está en la posición: " + posicion);
} else {
    System.out.println("Número no encontrado");
}
```

Como puedes ver el código es muy parecido y solo aparece la condición de la variable bandera.

Una de las cosas que tenemos que tener en cuenta a la hora de utilizar las sentencias break sin etiquetar es que estas generan que se rompa la secuencia de ejecución de sentencias hasta el primer bloque anidado.

Pero, ¿qué sucedería si queremos salir de un conjunto de bucles anidados? Aunque podríamos utilizar múltiples `break` *existe la posibilidad de utilizar sentencias break etiquetadas.*

Las sentencias break etiquetadas funcionan igual que las `break` *pero al ejecutarse se salen a la siguiente sentencia después del bloque etiquetado.*

La sintaxis es:

```
break nombre_etiqueta;
```

Veamos como podría ser una estructura de uso de las sentencias break etiquetadas.

```
sentencia(s) iniciales;


etiqueta:
  while (expresion) {
    sentencia(s) bloque1;
    while (expresion) {
      sentencia(s) bloque2;
      break etiqueta;
    }
  }


sentencias(s) finales;
```

Al ejecutarse se sale de todo el bloque etiquetado como etiqueta y ejecuta las sentencias finales.

Esto podemos encontrarlo si estamos recorriendo una matriz para buscar un elemento. Ya que para recorrer una matriz vamos a necesitar dos bucles anidados.

```
int[][] matriz = {
  {1,2,3,4},
  {5,6,7,8},
  {9,10,11,12}
};
int numeroBuscado = 5;

busqueda:
for (int x=0; x &lt; matriz.length; x++) {
  for (int y=0; y &lt; matriz[x].length; y++) {
    if (matriz[x][y] = numeroBuscado) {
      encontrado = true;
      break busqueda;
    }
  }
}

if (encontrado) {
  System.out.println(x + "," + y);
} else {
  System.out.println("No encontrado");
}
```

continue

Otra sentencia que podemos utilizar en los bucles es la sentencia `continue`. A ejecutar una La sentencia `continue` dejaremos de ejecutar las sentencias que quedan para acabar el bloque dentro de un bucle para volver a evaluar una expresión.

La estructura de unas sentencia `continue` sería:

```
while (expresion) {
  sentencia(s) iniciales;
  continue;
  sentencias(s) finales;
}
```

Al ejecutarse la sentencia `continue` nunca se ejecutarán las sentencias finales.

De igual manera que sucedía con la sentencia `break`, podemos realizar continue etiquetados. En este caso la sentencia `continue` nos llevará directamente a la primera condición de evaluación del bloque.

La estructura en este caso sería la siguiente:

```
etiqueta:
  while (expresion) {
    sentencia(s) iniciales;
    while (expresion) {
      sentencia(s) iniciales;
      continue etiqueta;
      sentencia(s) finales;
    }
    sentencia(s) finales;
  }
```

Arrays Java

¿Qué es un array en Java?

Un array Java es una estructura de datos que nos permite almacenar una ristra de datos de un mismo tipo. El tamaño de los arrays se declara en un primer momento y no puede cambiar en tiempo de ejecución como puede producirse en otros lenguajes. La declaración de un array en Java y su inicialización se realiza de la siguiente manera:

```
tipo_dato nombre_array[];

nombre_array = new tipo_dato[tamanio];
```

Por ejemplo, podríamos declarar un array de caracteres e inicializarlo de la siguiente manera:

```
char arrayCaracteres[];

arrayCaracteres = new char[10];
```

Los arrays Java se numeran desde el elemento cero, que sería el primer elemento, hasta el tamaño-1 que sería el último elemento. Es decir, si tenemos un array de diez elementos, el primer elemento sería el cero y el último elemento sería el nueve. Para acceder a un elemento especifico utilizaremos los corchetes de la siguiente forma. Entendemos por acceso, tanto el intentar leer el elemento, como asignarle un valor.

```
arrayCaracteres[numero_elemento];
```

Por ejemplo, para acceder al tercer elemento lo haríamos de la siguiente forma:

```
// Lectura de su valor.

char x = arrayCaracteres[2];


// Asignación de un valor. Como se puede comprobar se pone el  número dos, que coincide con el tercer elemento. Ya que como  dijimos anteriormente el primer elemento es el cero.

arrayCaracteres[2] = 'b';
```

El objeto array, aunque podríamos decir que no existe como tal, posee una variable, la cual podremos utilizar para facilitar su manejo.

Tamaño del array: .length

Este atributo nos devuelve el número de elementos que posee el array. Hay que tener en cuenta que es una variable de solo lectura, es por ello que no podremos realizar una asignación a dicha variable. Por ejemplo esto nos serviría a la hora de mostrar el contenido de los elementos de un array:

```
char array[];

array = new char[10];


for (int x=0;x<array.length;x++)

  System.out.println(array[x]);
```

Uno de los axiomas de la orientación a objetos es la ocultación, es decir, que no podemos acceder a una variable declarada dentro de una clase a no ser que lo hagamos a través de un método de la clase. Aquí estamos accediendo a una variable. ¿Quizás sea por que no consideran a los arrays como objetos?.

Matrices o Arrays de varios subindices

Podremos declarar arrays Java de varios subíndices, pudiendo tener arrays Java de dos niveles, que serían similares a las matrices, arrays Java de tres niveles, que serían como cubos y así sucesivamente, si bien a partir del tercer nivel se pierde la perspectiva geométrica. Para declarar e inicializar un array de varios subíndices lo haremos de la siguiente manera:

```
tipo_dato nombre_array[][];

nombre_array = new tipo_dato[tamanio][tamanio];
```

De esta forma podemos declarar una matriz Java de 2x2 de la siguiente forma:

```
int matriz[][];

matriz = new int[2][2];
```

El acceso se realiza de la misma forma que antes:

```
int x = matriz[1][1]; // Para leer el contenido de un elemento

matriz[1][1] = x;     // Para asignar un valor.
```

Hay que tener en cuenta que para mostrar su contenido tendremos que utilizar dos bucles. Para saber el número de columnas lo haremos igual que antes mediante la variable `.length`*, pero para saber el numero de filas que contiene cada columna lo tendremos que realizar de la siguiente manera:*

```
matriz[numero_elemento].lenght;
```

Nuestra lectura de los elementos de una matriz quedaría de la siguiente forma:

```
int matriz[][];

matriz = new int[4][4];

for (int x=0; x < matrix.length; x++) {

  for (int y=0; y < matriz[x].length; y++) {

      System.out.println (matriz[x][y]);

  }

}
```

Inicialización de Arrays en Java

Existe una forma de inicializar un array en Java con el contenido, amoldándose su tamaño al número de elementos a los que le inicialicemos. Para inicializar un array Java utilizaremos las llaves de la siguiente forma:

```
tipo_dato array[] = {elemento1,elemento2,...,elementoN};
```

Así, por ejemplo, podríamos inicializar un array Java o una matriz Java:

```
// Tenemos un array de 5 elementos.

char array[] = {'a','b','c','d','e'};

// Tenemos un array de 4x4 elementos.

int array[][] = { {1,2,3,4}, {5,6,7,8}};
```

Programación orientada a objetos (POO)

La programación orientada a objetos (POO) es un paradigma de programación que usa objetos para crear aplicaciones. Está basada en tres pilares fundamentales: herencia, polimorfismo, encapsulación. Su uso se popularizó a principios de la década de 1990. En la actualidad, existe una gran variedad de lenguajes de programación que soportan la orientación a objetos, entre ellos Java.

Ventajas

- ***Fomenta la reutilización y ampliación del código.***
- ***Permite crear sistemas más complejos.***
- ***La programación se asemeja al mundo real.***
- ***Agiliza el desarrollo de software.***
- ***Facilita el trabajo en equipo.***

Lo interesante de la POO es que proporciona conceptos y herramientas con las cuales se modela y representa el mundo real tan fielmente como sea posible.

Si estás empezando con el lenguaje Java hay una serie de conceptos básicos Java de la orientación a objetos que debes de manejar para poder desarrollar con este lenguaje.

Aquí los vamos a ver por encima y dedicaremos un capítulo entero a ellos entrando en detalle sobre todas sus características

- **Objeto**
- **Clase**
- **Paquete**
- **Interface**
- **Herencia**

Objeto

Es un elemento de software que intenta representar un objeto del mundo real. De esta forma un objeto tendrá sus propiedades y acciones a realizar con el objeto. Estas propiedades y acciones están encapsuladas dentro del objeto, cumpliendo así los principios de encapsulamiento.

El *paradigma de la orientación a objetos* aparece como contraste a la *programación estructurada* que se venía utilizando desde los años 60.

Un objeto tiene su estado (o estados) y su comportamiento. Esto se modela mediante propiedades (o variables) y métodos. Incluso un objeto puede contener a su vez otro tipo de objeto.

Encapsulación de datos

Las interacciones con los objetos se hacen mediante los métodos. Es decir, si queremos conocer información del estado del objeto deberemos de llamar a uno de sus métodos y no directamente a las propiedades.

Esta encapsulación nos permitiría cambiar las propiedades del objeto sin que los consumidores se vean afectados siempre y cuando les sigamos retornando el mismo resultado.

Si bien hay objetos que tienen propiedades públicas, por lo cual podremos acceder directamente a dichas propiedades sin necesidad de utilizar un método.

El uso de objetos nos proporciona los siguientes beneficios:

1. **Modularidad, el objeto y sus propiedades puede ser pasado por diferentes estructuras del código fuente, pero el objeto es el mismo.**
2. **Encapsular datos, ocultamos la implementación de propiedades del objeto ya que accederemos a través de los métodos del objeto.**
3. **Reutilización de Código, podemos tener diferentes instancias de un objeto de tal manera que esas diferentes instancias están compartiendo el mismo código.**
4. **Reemplazo, podemos reemplazar un objeto por otro siempre y cuando estos objetos tengan el mismo comportamiento.**

Ejemplos de objetos

Cualquier concepto del mundo real se puede modelar como un objeto con su estado y comportamiento. Por ejemplo *un televisor es un objeto, cuyos estados pueden ser: *encendida*, *apagada*, *en el canal1*, *en el canal2*, *grabando*,… y sus acciones serán *"encender televisor"*, *"apagar televisor"*, *"cambiar de canal"*, *"iniciar la grabación"*,…**

Por ejemplo, imaginemos una figura geométrica como podría ser un *triángulo*. Un triángulo podemos definirlo por varias propiedades como pueden ser: *base*, *altura*, *el lado* y las *coordenadas x,y del centro del triángulo*. Como métodos de un triángulo podemos *"calcular el área del triángulo"*, *"calcular el perímetro del triángulo"*.

Clase

Las clases representan los prototipos de los objetos que tenemos en el mundo real. Es decir, es una generalización de un conjunto de objetos. A su vez los objetos serán instancias de una determinada clase.

Si volvemos al ejemplo del televisor, existen múltiples tipos de televisores y cada uno con sus características. Si bien existe un esquema o prototipo que define el televisor. Este prototipo es lo que conocemos de la clase.

En la clase es dónde realmente definimos las propiedades y métodos que podrán contener cada una de las instancias de los objetos.

Por ejemplo, para nuestro caso de las figuras geométricas podríamos definir un triángulo de la siguiente forma:

```
class Triangulo {
```

```
    private long base;
    private long altura;

    public Triangulo(long base, long altura) {
        this.base = base;
        this.altura = altura;
    }

    public long area() {
        return (base*altura)/2;
    }
}

Triangulo t1 = new Triangulo(2.0,3.0);
Triangulo t2 = new Triangulo(4.0,7.0);

t1.area(); // Área 3.0
t2.area(); // Área 14.0
```

De momento no te preocupes por entender el código del todo, pero verás que hemos definido una clase triángulo la cual tiene dos propiedades base y altura. Estas propiedades las hemos definido como "private" lo cual hace que no puedan ser visibles desde fuera.

```
private long base;

private long altura;
```

Luego tenemos lo que se conoce como un método constructor. Es el método que tiene el mismo nombre que la clase: `Triangulo ()` *y que nos sirve para inicializar las propiedades desde el exterior.*

```
public Triangulo(long base, long altura) {

    this.base = base;

    this.altura = altura;

}
```

Además hemos creado un método que nos calcula el área de un triángulo (base x altura / 2). Este método ya es público y podrá ser invocado de forma externa.

```
public long area() {

  return (base*altura)/2;
```

```
}
```

Vemos cómo creamos diferentes objetos del tipo `Triángulo`. A estos objetos los pasamos diferentes valores.

```
Triangulo t1 = new Triangulo(2.0,3.0);

Triangulo t2 = new Triangulo(4.0,7.0);
```

Y por último hemos invocado al método que nos devuelve el área del triángulo del objeto en concreto.

```
t1.area(); // Área 3.0

t2.area(); // Área 14.0
```

Interface

Un interface es una forma de establecer un contrato entre dos elementos. Un interface indica qué acciones son las que una determinada clase nos va a ofrecer cuando vayamos a utilizarla.

Cuando implementemos un interface (cuando lo usemos) deberemos de implementar todas las acciones (métodos) que este contenga.

Por ejemplo podríamos definir un interface `Figura` el cual indique qué métodos son obligatorios cuando vayamos a definir una figura. El interface se define mediante la palabra interface.

```
interface Figura {

    ...

}
```

Dentro del interface definimos los métodos que serán obligatorios. Por ejemplo, que de una figura se pueda calcular su área y calcular su perímetro.

```
interface Figura {

    public long area();

    public long perimetro();
```

```
}
```

Cuando queramos que una clase implemente un determinado interface deberemos de utilizar el operador implements indicando el nombre del interface a implementar.

Así, si un triángulo queremos que implemente el interface Figura lo definiremos de la siguiente forma:

```
public Triangulo implements Figura {

    ...

}
```

En este momento la clase `Triangulo` deberá de implementar los métodos calcular área y calcular perímetro.

Paquete

Un paquete es una forma de organizar elementos de software mediante un espacio de nombres. Así podremos afrontar desarrollos grandes de software facilitando la forma de encontrar o referirnos a un elemento.

Podríamos entender el sistema de paquetes como si fuese un sistema de carpetas. De tal manera que colocaremos cada una de las clases (o ficheros) en un paquete (o directorio).

Los paquetes se definen mediante el modficiador `package` seguido del nombre del paquete. El paquete lo definiremos en la primera línea de cada una de las clases.

Una definición de paquete podría ser:

```
package net.manual.java.ejemplos;
```

El lenguaje Java nos proporciona un conjunto de paquetes por defecto (conocido como API Java) en los que se pueden encontrar múltiples utilidades del lenguaje. Por ejemplo, la clase Java que nos ayuda a manipular las cadenas de texto es la clase String. La clase String la podemos encontrar en el paquete java.lang.

Herencia

La herencia es una forma de estructurar el software. Mediante la herencia podemos indicar que una clase hereda de otra. Es decir la clase extiende las capacidades (propiedades y métodos) que tenga y añade nuevas propiedades y acciones.

Digamos que las nuevas clases especializan más aquellas clases de las que heredan al añadir nueva funcionalidad. Aunque también pueden reescribir el funcionamiento de dichos elementos.

En nuestro ejemplo del triángulo, este podría heredar de una clase polígono.

```
public class Triangulo extends Poligono {

    ...

}
```

De igual manera de esta clase general `Poligono` podrían heredar otras clases que representasen un polígono, por ejemplo las clases `Cuadrado`, `Pentagono`,...

```
public class Cuadrado extends Poligono {

    ...

}
```

```
public class Pentagono extends Poligono {

    ...

}
```

La herencia entre clases se indica mediante el operador `extends`.

La clase superior de la que heredan las figuras puede definir una serie de propiedades y métodos que heredarán las clases hijas y que por ende podrán utilizar.

Por ejemplo, la clase `Poligono` puede tener una propiedad que sean las longitudes de los lados del polígono y que utilice esas longitudes para calcular el perímetro del polígono.

```
public class Poligono {

    private long[] lados;

    public Poligono(long[] lados) {
        this.lados = lados;
    }

    public long perimetro() {
        ...
    }

}
```

Cuando ahora indiquemos que la clase `Triangulo` hereda de la clase `Poligono`.

```
public class Triangulo extends Poligono {
    ...

    public Triangulo (long base, long altura, int[] lados) {
        super(lados);
        this.base = base;
        this.altura = altura;
    }

}
```

Veremos que los objetos instanciados como triángulos tendrán acceso a los métodos del polígono.

```
Triangulo t1 = new Triangulo(2.0,3.0);
t1.perimetro();
```

En este caso accedemos al método perímetro que heredamos de la clase Polígono.

Una de las cosas que tienes que saber en la herencia es que en el constructor de la clase que hereda (o clase hija) se deberá de llamar al constructor de la clase padre. Para ello se utiliza el método especial `super()`;

instanceof

El operador instanceof es un operador especial para los objetos. Mediante el operador instanceof podemos comprobar si un objeto es de una clase concreta.

La estructura del operador instanceof es:

```
objeto instanceof clase
```

El operador instanceof devolverá true siempre y cuando el objeto sea del tipo clase o de alguna de las clases de las que herede.

Así podríamos definir una secuencia de clases:

```
class Poligono {}

interface Figura {}

class Triangulo extends Poligono implements Figura {}
```

Ahora definimos un par de objetos:

```
Poligono p = new Poligono();

Triangulo t = new Triangulo();
```

Podemos, mediante el uso del operador instanceof, comprobar que t es instancia de tipo Triangulo, Poligono y Figura. Mientras que p es instancia de tipo Polígono, pero no de Triangulo, ni Figura.

```
System.out.println("p es instancia de ");

if (p instanceof Poligono)

  System.out.println("Poligono");

if (p instanceof Triangulo)
```

```
    System.out.println("Triangulo");
if (p instanceof Figura)
    System.out.println("Figura");


System.out.println("t es instancia de ");
if (t instanceof Poligono)
    System.out.println("Poligono");
if (t instanceof Triangulo)
    System.out.println("Triangulo");
if (t instanceof Figura)
    System.out.println("Figura");
```

Para entender este modelo vamos a revisar 2 conceptos fundamentales:

- ***clases***
- ***objetos***

Las clases

En el mundo real, normalmente tenemos muchos objetos del mismo tipo. Por ejemplo, nuestro teléfono móvil es sólo uno de los miles que hay en el mundo. Si hablamos en términos de la programación orientada a objetos, podemos decir que nuestro objeto móvil es una instancia de una clase conocida como "móvil". Los móviles tienen características (marca, modelo, sistema operativo, pantalla, teclado, etc.) y comportamientos (hacer y recibir llamadas, enviar mensajes multimedia, transmisión de datos, etc.).

Cuando se fabrican los móviles, los fabricantes aprovechan el hecho de que los móviles comparten esas características comunes y construyen modelos o plantillas comunes, para que a partir de esas se puedan crear muchos móviles del mismo modelo. A ese modelo o plantilla le llamamos clase, y a los equipos que sacamos a partir de ella la llamamos objetos.

Esto mismo se aplica a los objetos de software, se puede tener muchos objetos del mismo tipo y mismas características.

Definición teórica: La clase es un modelo o prototipo que define las variables y métodos comunes a todos los objetos de cierta clase. También se puede decir que una clase es una plantilla genérica para un conjunto de objetos de similares características.

Por otro lado, una instancia de una clase es otra forma de llamar a un objeto. En realidad no existe diferencia entre un objeto y una instancia. Sólo que el objeto es un término más general, pero los objetos y las instancias son ambas representación de una clase.

Objetos

Entender que es un objeto es la clave para entender cualquier lenguaje orientado a objetos.

Tomemos como ejemplo un ordenador. No necesitamos ser expertos para saber que un ordenador está compuesto internamente por varios componentes: placa base, procesador, disco duro, tarjeta de video, etc…

El trabajo en conjunto de todos estos componentes hace funcionar un ordenador, sin embargo no necesitamos saber cómo trabajan cada uno de estos componentes. Cada componente es una unidad autónoma, y todo lo que necesitamos saber es cómo interactúan entre sí los componentes, saber por ejemplo si el procesador y las memorias son compatibles con la placa base, o conocer donde se coloca la tarjeta de video. Cuando conocemos como interaccionan los componentes entre sí, podremos armar fácilmente un ordenador.

¿Qué tiene que ver esto con la programación? La programación orientada a objetos trabaja de esta manera. Todo el programa está construido en base a diferentes componentes (objetos), cada uno tiene un rol específico en el programa y todos los componentes pueden comunicarse entre ellos de formas predefinidas.

Todo objeto del mundo real tiene 2 componentes: variables de clase y métodos.

Por ejemplo, los automóviles pueden tener como variables de clase (marca, modelo, color, velocidad máxima, etc.) y como métodos (frenar, acelerar, retroceder, llenar combustible, cambiar llantas, etc.).

Los objetos de Software, al igual que los objetos del mundo real, también tienen variables de clase y métodos. Un objeto de software mantiene sus características en una o más "variables", e implementa su comportamiento con "métodos". Un método es una función o subrutina asociada a un objeto.

Imaginemos que tenemos aparcado en el garaje un Ford Focus color azul que corre hasta 260 km/h. Si pasamos ese objeto del mundo real al mundo del software, tendremos un objeto Automóvil con sus características predeterminadas:

Marca = Ford

Modelo = Focus

Color = Azul

Velocidad Máxima = 260 km/h

Por lo tanto dentro de una clase tendremos como variables de clase las características descritas anteriormente y como métodos tendremos en este caso concreto (frenar, acelerar, retroceder, llenar combustible, etc…).

Vamos a ver un ejemplo práctico de cómo se hace esto en Java:

```
class Automovil {

   // VARIABLES DE CLASE
   private String marca;
   private String modelo;
   private String color;
   private String velocidadMaxima;

   // CONSTRUCTOR QUE INICIALIZA LAS VARIABLES DE CLASE
   public Automovil(String marca, String modelo, String color,
String velocidadMaxima) {
      this.marca = marca;
      this.modelo = modelo;
      this.color = color;
      this.velocidadMaxima = velocidadMaxima;
   }


   // METODOS GETTER Y SETTER PARA PODER RECUPERAR O CAMBIAR
   // LOS DATOS DE LAS VARIABLES DE CLASE

   public String getMarca() {
      return marca;
```

```
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public String getModelo() {
        return modelo;
    }

    public void setModelo(String modelo) {
        this.modelo = modelo;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public String getVelocidadMaxima() {
        return velocidadMaxima;
    }

    public void setVelocidadMaxima(String velocidadMaxima) {
        this.velocidadMaxima = velocidadMaxima;
    }

}
```