General Java

Java se creó como una herramienta de programación para ser usada en un proyecto de set-top-box en una pequeña operación denominada the Green Project en Sun Microsystems en 1991 (**fecha oficial de creación/lanzamiento en 1995**) con el nombre "OAK", posteriormente cambiado por Green por problemas legales, y finalmente con la denominación actual JAVA. El objetivo de java era crear un lenguaje de programación parecido a C++ en estructura y sintaxis, fuertemente orientado a objetos, pero con una máquina virtual propia.

El lenguaje de programación Java fue desarrollado originalmente por **James Gosling**.

Características de Java

- Año de creación (1995).
- Es un lenguaje multiplataforma, no está diseñado para un equipo en concreto.
- Es un lenguaje de programación orientado a objetos.
- Se ha creado para usarse en internet.
- Cualquier equipo/dispositivo con una JMV (Máquina Virtual Java) puede leer el código y ejecutarlo.

Lema de Java

Write Once, Run Everywhere -- Escribir una vez, ejecutar en todas partes

Conceptos Básicos

1.1. Algoritmo

Un **algoritmo** es un conjunto finito de instrucciones bien definidas que nos ayudan a resolver un problema.

1.2. Lenguajes de programación

Un **lenguaje de programación** puede definirse como un idioma artificial diseñado para que sea fácilmente entendible por un humano e interpretable por una máquina. Un algoritmo escrito utilizando las instrucciones de un lenguaje de programación se le denomina **programa**.

Consta de una serie de reglas y de un conjunto de órdenes o instrucciones. Cada una de estas instrucciones realiza una tarea determinada. A través de una secuencia de instrucciones podemos indicar a una computadora el algoritmo que debe seguir para solucionar un problema dado.

1.2.1. Lenguajes compilados e interpretados

Un lenguaje de programación esta diseñado para que una persona escriba fácilmente algoritmos, para que un ordenador comprenda lo que se ha escrito se utiliza una herramienta llamada **compilador**, que transforma el conjunto de ordenes en los ceros y unos comprensibles por la máquina (denominado como **código máquina**).

Existen dos enfoques para realizar el proceso de traducción del lenguaje de programación (código fuente) al código máquina.

- **Compilación:** Traduce todas las instrucciones del código fuente y almacena el código máquina generado. Permite ejecutar el programa, sin volver a compilarlo, tantas veces como se necesite y sin disponer del código fuente.
- **Interpretación:** Consiste en traducir el código fuente instrucción a instrucción e ir ejecutando. Solo se traduce y ejecuta la siguiente instrucción que necesitamos. El proceso continuará sucesivamente hasta que la ejecución termine.

A partir del proceso de compilación e interpretación introducimos dos nuevos conceptos:

- **Tiempo de compilación:** Es el intervalo de tiempo durante el cual se compila un programa.
- **Tiempo de ejecución:** Es el intervalo de tiempo durante el cual un programa se ejecuta.

1.2.2. Lenguajes Multiplataforma

El lenguaje C necesita que un mismo programa se compile para cada combinación de tipo de máquina y sistema operativo donde se va a ejecutar. Aunque se pueda ejecutar en distintas plataformas, requiere del compilado específico para cada una de ellas. Por este motivo, **no se considera un lenguaje multiplataforma**.

Java se considera un **lenguaje multiplataforma**. Un mismo programa, una vez compilado, se puede ejecutar en cualquier ordenador que tenga instalada la Máquina Virtual Java, que no es más que un intérprete de bytecode.

Para el caso concreto de Java, vamos a afinar los conceptos de:

- **Tiempo de compilación:** Es el espacio de tiempo en el que se traduce el código fuente al bytecode.
- **Tiempo de ejecución:** Es el tiempo durante el cual el bytecode se interpreta (por la JVM) y se ejecuta por la plataforma correspondiente.

1.5. El programa principal

Para escribir los primeros programas usaremos:

```
import java.util.Scanner;
import java.lang.Math;

public class Nombre del archivo{
    public static void main(String []args){
```

Al escribir un programa en Java usaremos literalmente la fórmula anterior.

1.6. Palabras reservadas

En java existe una serie de palabras con un significado especial, como **package**, **class** o **public**. Estas se denominan palabras reservadas y definan la gramática del lenguaje.

Abstract	Continue	Float	Native	Strictfp	Void
Assert	Default	For	New	Super	volatile
Boolean	Do	If	Package	Switch	While
Break	Double	Implements	Private	Synchronized	yield

Byte	Else	Import	Protected	This
Case	Enum	Instanceof	Public	Throw
Catch	Extends	Int	Return	Throws
Char	Final	Interface	Short	Transient
Class	Finally	Long	Static	try

Al conjunto anterior hay que sumar otras palabras reservadas: **const**, **goto**, **true**, **false** y **null**. Las palabras reservadas solo pueden escribirse en determinado lugar de un programa y no pueden ser utilizadas como identificadores.

1.7. Concepto de variable

Una **variable** es una representación, mediante un identificador, de un valor, que puede cambiar durante la ejecución de un programa. A las variables se les asignan valores concretos por medio del operador de asignación (**=**).

1.7.1. Identificadores

El nombre con el que se identifica cada variable se denomina **identificador**. Hay que tener en cuenta que Java distingue entre mayúscula y minúscula (edad es distinto a eDaD) Los identificadores deben seguir las siguientes reglas:

- Comienzan siempre por una letra, un subrayado (_) o un dólar ($).
- Los siguientes caracteres pueden ser letras, dígitos, subrayado (_) o dólar ($).
- Se hace distinción entre mayúsculas y minúsculas.
- No hay una longitud máxima para el identificador.

1.8. Tipos primitivos

En un programa de ejecución, las variables se almacenan en la memoria del ordenador. Cada una de ellas necesita un tamaño para guardar sus valores. Un tamaño demasiado pequeño no permite guardar valores grandes o muy precisos, y se corre el riesgo de que el valor que se va a guardar no quepa en el espacio reservado. Utilizar un tamaño excesivamente grande desaprovecha la memoria, haciendo un uso ineficiente de ella.

La solución a este problema no es definir un tamaño para cada variable, sino definir unos tipos de variables, con unos tamaños y rangos de valores conocidos, y que las variables utilizadas en nuestros programas se ciñan a estos tipos.

1.8.1 Variables de tipo primitivo

Al escribir un programa, hemos de indicar a qué tipo pertenece cada variable. Este proceso recibe el nombre de declaración de variables y se hará forzosamente antes de su primer uso.

Todas las declaraciones de variables terminan en punto y coma (;), aunque es posible declarar a la vez varias del mismo tipo, separándolas por comas (,).

1.8.2. Rangos

Tipo	Uso	Tamaño	Rango
Byte	Entero corto	8 bits	De -128 a 127
Short	Entero	16 bits	De -32768 a 32767
Int	Entero	32 bits	De -2147483648 a 2147483647

Long	Entero largo	64 bits	± 9 223 372 036 854 775 808
Float	Real precisión sencilla	32 bits	De -10^{32} a 10^{32}
Double	Real precisión doble	64 bits	De -10^{300} a 10^{300}
Boolean	Lógico	1 bit	**True** 0 **false**
Char	Texto	16 bits	Cualquier carácter

1.9. Variables de objeto

Es posible declarar variables cuyo tipo no sea un tipo primitivo, sino una clase. A estas variables se les denomina **variables de objeto** y las utilizaremos para poder aprovechar las herramientas que Java proporciona. Se inicializan por defecto, si es su declaración no se les asigna un valor.

1.10. Constantes

Las constantes son un caso especial de variables, donde, una vez se les asigna su primer valor, este permanece inmutable durante el resto del programa. Cualquier dato que no cambie es candidato a guardarse en una constante.

La declaración de constantes es similar a la de variables, pero añadiendo la palabra reservada **final**.

```
final tipo nombreConstante; // Constante "final"
final byte mayoriaEdad = 18;
final int numeroAlumnos;

numeroAlumnos = aulas * 30;
```

1.11. Comentarios

- **Comentario multilínea:** Cualquier texto incluido entre los caracteres /* (apertura de comentario) y */ (cierre de comentario) será interpretado como un comentario y puede extenderse a través de varias líneas.

- **Comentario hasta final de la línea:** Todo lo que sigue a los caracteres // hasta el final de la línea se considera un comentario.

- **Comentario de documentación:** Similar al comentario multilínea, con la diferencia de que, para iniciarlo, se utilizar los caracteres /**. Existen herramientas que generan documentación automática a partir de ese tipo de comentarios.

1.12. API de Java

Una de las grandes ventajas de los lenguajes de programación modernos es que disponen de una amplia biblioteca de herramientas que realizan tareas complejas de forma transparente al programador que las utiliza, facilitando su tarea.

A estas herramientas, en Java, se les denomina clases y facilitan multitud de tareas. Algunos ejemplos de las funcionalidades que nos brindan son:

- **Lectura de datos:** Leen información desde el teclado, desde un fichero o desde otros dispositivos.
- **Cálculos complejos:** Realizan operaciones matemáticas como raíces cuadradas, logaritmos, cálculos trigonométricos, etcétera.

- **Manejo de errores:** Controlan la situación cuando se produce un error de algún tipo.
- **Escritura de datos:** Escriben información relevante en dispositivos de almacenamiento, impresoras, monitores, etcétera.

1.12.1. Paquetes

A una agrupación de clases se le denomina **paquete**. Los paquetes pueden agruparse, a su vez, en otros paquetes (Por ejemplo, **Math**).

```
import java.util.Scanner;
import java.lang.Math;
import java.time.LocalTime;
```

Cada clase que compone la API puede utilizarse de dos modos:

- **De forma estática:** Se usa directamente el método. Por ejemplo, la case **Math** se utiliza de forma estática.
- **De forma no estática:** Esta manera de utilizar las clases requiere del operador **new**, que se verá en profundidad en unidades posteriores. Un ejemplo de clase que se utiliza de esta forma es **Scanner**, que permite que el usuario introduzca datos en una aplicación.

1.12.2. Salida por consola

Una de las operaciones más básicas que proporciona la API es aquella que permite mostrar un mensaje en el monitos, con idea de aportar información al usuario.

```
        System.out.println("Hola");
        System.out.print("Hola holita");
        System.out.printf("Que pasó");
```

Print que muestra literalmente el mensaje en el monitor.

Println igual que el anterior, pero tras el mensaje, inserta un retorno de carro (nueva línea).

1.12.3. Entrada de datos

Otra operación muy utilizada, consiste en recabar información del usuario a través del teclado. Cuando se hace de forma simple, en modo texto, sin ratón ni interfaz gráfica, se dice que obtenemos datos por consola.

Scanner es una clase de la API que se utiliza de forma no estática, es decir, necesita del operador **new**. Y la forma de trabajar con ella es siempre la misma: en primer lugar tendremos que crear un nuevo escáner.

```
        Scanner teclado = new Scanner(System.in);

        variable = teclado.next"TipoVariable"();
```

Este fragmento de código se utilizará como una fórmula literal cada vez que necesitemos introducir información por teclado. Según deseemos leer un entero, un real o una cadena de caracteres, solo será necesario modificar el nombre y tipo de la variable que leer y el método.

1.13. Operaciones Básicas

Java dispone de multitud de operadores con los que se pueden crear expresiones utilizando como operandos variables, constantes, números y otras expresiones.

1.13.1. Operador de Asignación

El operador = se usa para modificar el valor de una variable. A la variable se le asigna como valor el resultado de la expresión, si en el momento de la asignación la variable tuviera un valor anterior, este se pierde.

1.13.2. Operadores Aritméticos

El operador – (menos unario) sirve para cambiar el signo de la expresión que le sigue, que estará formada por cualquier secuencia de operaciones aritméticas.

```
a = 1;
b = -a // b vale -1
```

El operador % devuelve el resto de dividir el primer operando entre el segundo. Por ejemplo **7%3** (se lee 7 módulo de 3) vale 1, ya que al dividir 7 entre 3 el resto (el módulo) es 1.

Símbolo	Descripción
+	Suma
+	Más unario: positivo
-	Resta
-	Menor unario: negativo
*	Multiplicación
/	División
%	Módulo
++	Incremento en 1
--	Decremento en 1

Los operadores **++** y **--** se utilizan para incrementar o decrementar una variable en 1.

```
a ++; // a = a + 1;
b --; // b = b - 1;
```

1.13.3 Operadores Relacionales

Son aquellos que producen un resultado lógico o booleano a partir de las comparaciones de expresiones numéricas. El resultado solo permite dos posibles valores: **true** o **false.**

Símbolo	Descripción
==	Igual que
!=	Distinto que
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que

1.13.4 Operadores Lógicos

Permiten operar a partir de expresiones lógicas, formando expresiones más complejas, que devuelven, a su vez, un valor lógico. Existen los operadores:

Símbolo	Descripción
&&	Operador **and**: Y
||	Operador **or**: O
!	Operador **not**: negación

La expresión formada a partir de otras dos unidas por el operador será **true** cuando ambas expresiones utilizadas se evalúen como ciertas. Y en caso contrario, si alguna o las dos expresiones se evalúan como falsas, la expresión resultante también será falsa.

1.13.5. Operadores Opera y Asigna

Por simplicidad existen otros operadores de asignación llamados opera y asigna, que realizan la operación indicada tomando como operandos el valor de la variable a la izquierda y el valor a la derecha del =. El resultado se asigna a la misma variable utilizada como primer operando.

Símbolo	Descripción
+=	Igual que
-=	Distinto que
*=	Menor que
/=	Menos o igual que
%=	Mayor que

Todos tienen el mismo funcionamiento. Utilizan la misma variable para operar con su valor y asignarle el resultado.

```
variable += 3; // variable = variable + 3;
x *= 2; //X = x * 2;
```

1.13.6 Operador Ternario

Este operador devuelve un valor que se selecciona de entre dos posible. La selección dependerá de la evaluación de una expresión relacional o lógica que, como hemos visto, piuede tomar dos valores: **verdadero** o **falso**.

```
//Ejemplo ternario
Scanner teclado = new Scanner(System.in);
int a = teclado.nextInt();
int b = teclado.nextInt();
int maximo = a > b ? a : b; // A mayor que B o contrario
System.out.println("El máximo es: " + maximo);
```

1.14. Conversión de tipos

Todas las variables en Java tienen asociado un tipo. Cuando asignamos un valor a una variable, ambos deben ser del mismo tipo. A una variable tipo **int** se le puede asignar un valor **int** y a una variable **double** se le puede asignar un valor **double**.

```
        int a = 2;
        double x = 2.3;
```

Java permite la asignación de variables de un tipo a otra. Para pasar de **int** a **double** Java convierte de forma automática el valor entero (3, por ejemplo) a valor **double** (3.0) antes de asignarlo a la variable X. Esto es posible porque la variable double es de mayor tamaño que el valor int.

Muy distinto es que intentemos asignar un valor double a una variable int, que no tiene sitio suficiente para guardarlo. En este caso, java no hará ninguna conversión automática. Se limitará a darnos un error de compilación, aunque a veces es interesante guardar la parte entera de un número con decimales en una variable entera. Para ellos deberemos colocar un cast o molde delante del valor que queremos asignar.

```
    int a = (int) 2.6; //(int) indica el tipo al que se convertirá el valor
```

Condicionales

Un programa no tiene por qué ejecutar siempre la misma secuencia de instrucciones. Puede darse el caso de que, dependiendo del valor de alguna expresión o de alguna condición, interese ejecutar o evitar un conjunto de sentencias (**if**, **if-else** y **switch**).

2.1. Expresiones Lógicas

Una condición no es más que una expresión relacional o lógica. El valor de una condición siempre es de tipo booleano: verdadero o falso.

2.1.1. Operadores Relacionales

Los operadores relacionales son aquellos que comparan expresiones numéricas para generar valores booleanos. Solo pueden tener dos valores: verdadero o falso.

Símbolo	Descripción
==	Igual que

!=	Distinto que
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que

2.1.2. Operadores Lógicos

Los operadores lógicos permiten construir condiciones más complejas, ya que estos operan y generan valores booleanos.

Símbolo	Descripción
&&	Operador **and**: Y
||	Operador **or**: O
!	Operador **not**: negación

- **Operador &&:** Será cierto si ambos operandos son ciertos.
- **Operador ||:** Es cierto si cualquiera (uno o ambos) de los operandos es cierto.
- **Operador !:** Niega ---cambia--- el valor al que se aplica, convirtiendo **true** en **false** y viceversa.

2.2. Condicional Simple: if

La sentencia **if** proporciona un control sobre un conjunto de instrucciones que pueden ejecutarse o no, dependiendo de la evaluación de una condición. Los dos posibles valores (true o false) de esta determinan si el bloque de instrucciones (**{}**) de if se ejecuta o no.

```
a = 3;
if (a + 1 < 10) {
    a = 0;
    System.out.println("Hola");
}
System.out.println("El valor de a es " + a);
```

2.3 Condicional doble: if-else

Existe otra versión de la sentencia **if**, denominada **if-else**, donde se especifican dos bloques de instrucciones. El primero (true) se ejecutará cuando la condición resulte verdadera y el segundo (false), cuando la condición resulte falsa. Ambos bloques son mutuamente excluyentes, en cada ejecución de la instrucción **if-else** solo se ejecutará uno de ellos.

```
if (a > 0) {
    System.out.println("Valor positivo");
} else {
    System.out.println("Valor negativo o cero");
```

```
}
```

El operador ternario permite seleccionar un valor de entre dos posible, dependiendo de la evaluación de una condición.

```
variable = condición ? valor1 : valor2

//Es lo mismo que usar

if (condición) {
    variable = valor1;
} else {
    variable = valor2;
}
```

Un ejemplo sería calcular el máximo de dos números introducidos:

```
Scanner teclado = new Scanner(System.in);
int a = teclado.nextInt();
int b = teclado.nextInt();
int maximo = a > b ? a : b; // A mayor que B o contrario
System.out.println("El máximo es: " + maximo);
```

2.3.2. Anidación de condicionales

Cuando debamos realizar múltiples comprobaciones, podemos anidar tantos **if** o **if-else** como necesitemos, unos dentro de otros. La anidación de condicionales hace que las comprobaciones sean excluyentes, y resulta más eficiente.

```
if (a - 2 == 1) {
    System.out.println("Hola ");
} else {
    if (a - 2 == 5) {
        System.out.println ("Me ");
    } else {
        if (a - 2 == 8) {
            System.out.println("Alegro ");
        } else {
            if (a - 2 == 9) {
                System.out.println("De ");
            } else {
                if (a - 2 == 11) {
                    System.out.println("Conocerte.");
                } else {
                    System.out.println("Sin coincidencia");
                }
            }
        }
```

```
            }
        }
```

Cuando el bloque de instrucciones del condicional está formado por una única sentencia, no es necesario utilizar llaves (**{}**), aunque son recomendables. Cuando hay muchos casos alternativos es habitual eliminar las llaves de los bloques **else**, consiguiendo un código más compacto.

```
        if (a - 2 == 1) {
            System.out.println("Hola ");
        } else if (a - 2 == 5) {
            System.out.println("Me ");
        } else if (a - 2 == 8) {
            System.out.println("Alegro ");
        } else if (a - 2 == 9) {
            System.out.println("De ");
        } else if (a - 2 == 11) {
            System.out.println("Conocerte.");
        } else {
            System.out.println("Sin coincidencia.");
        }
```

2.4. Condicional Múltiple: Switch

El hecho de utilizar muchos **if** o **if-else** anidados suele producir un código poco legible y difícil de mantener. Para estos casos Java dispone de la sentencia **switch**. La evaluación de expresión debe dar un resultado entero, convertible en entero o un valor de tipo **String**. La cláusula **default** es opcional.

- Evalúa la expresión y obtiene su valor.
- Compara el valor obtenido con cada valor de las cláusulas **case**. Se puede utilizar en un mismo **case** varios valores separados por coma.
- En el momento en que coincide con alguno de ellos, ejecuta el conjunto de instrucciones de esa cláusula **case** y de todas las siguientes.
- Si no existe coincidencia alguna, se ejecuta el conjunto de instrucciones de la cláusula **default**, siempre y cuando esté presente.
- Las ejecuciones continuaran hasta encontrarse con un **break**.

```
        switch (nota) {
            case 0,1,2,3,4: {//Bloque formado por dos instrucciones: entre
llaves
                System.out.println("Suspenso.");
                System.out.println("Ánimo...");
            }
            break;
            case 5: //Bloque de una única instrucción: podemos obviar las
llaves
                System.out.println("Suficiente.");
```

```
            break;
            case 6:
                System.out.println("Bien.");
            break;
            case 7,8:
                System.out.println("Notable.");
            break;
            case 9,10: {
                System.out.println("Sobresaliente.");
                System.out.println("Enhorabuena.");
            }
            break;
            default:
                System.out.println("Nota incorrecta.");
        }
```

Bucles

Un bucle es un tipo de estructura que contiene un bloque de instrucciones que se ejecuta repetidas veces; cada ejecución o repetición del bucle se llama iteración.

El uso de bucles simplifica la escritura de programas, minimizando el código duplicado.

3.1. Bucles controlados por condición

El control del número de iteraciones se lleva a cabo mediante una condición.

3.1.1. While

Al igual que **if**, **while** depende de la evaluación de una condición. Decide si realizar una nueva iteración basándose en el valor de la condición.

1. Se evalúa **condición**.
2. Si la evaluación resulta **true**, se ejecuta el bloque de instrucciones.
3. Tras ejecutarse el bloque de instrucciones, se vuelve al primer punto.
4. Si la condición es **false**, terminamos la ejecución del bucle.

```
5.          int i = 1; //Valor inicial
6.
7.
8.          while (i <=3) { /*El bucle iterará mientras i sea menos
9.                          o igual que 3*/
10.             System.out.println(i); //Mostramos i
11.             i++; /*Incrementamos i para la siguiente vuelta
12.                  del bucle*/
13.         }
```

Un bucle **while** puede realizar cualquier número de iteraciones, desde cero, cuando la primera evaluación de la condición resuelta falsa, hasta infinitas, en el caso de que la condición sea siempre cierta. Esto es lo que se conoce como **bucle infinito**.

3.1.2. Do-while

El bucle **do-while** es similar al **while**, con la diferencia de que primero se ejecuta el bloque de instrucciones y después se evalúa la condición para decidir si se realiza una nueva iteración.

```
int i = 1;

do {
    System.out.println(i);
    i++;
} while (i <=10);
```

Es el único bucle que termina en punto y coma (;). Mientras **while** se puede ejecutar de 0 a infinitas veces, el **do-while** lo hace de 1 a infinitas veces.

3.2. Bucles controlados por: for

El bucle **for** permite controlar el número de iteraciones mediante una variable (que suele recibir el nombre de contador).

Consta de 3 partes:

- **Inicialización:** Es una lista de instrucciones, separadas por comas, donde generalmente se inicializan las variables que van a controlar el bucle. Se ejecutan una sola vez antes de la primera iteración.
- **Condición:** Es una expresión booleana que controla las iteraciones del bucle. Se evalúa antes de cada iteración; el bloque de instrucciones se ejecutará solo cuando el resultado sea **true**.
- **Incremento:** Es una lista de instrucciones, separadas por comas, donde se suelen modificar las variables que controlan la condición. Se ejecuta al final de la iteración.

Aunque **for** está controlado por una condición que, en principio, puede ser cualquier expresión booleana, la posibilidad de configurar la inicialización y el incremento de las variables que controlan el bucle permite determinar de antemano el número de iteraciones.

```
for (int i = 1; i <= 2; i++) {
    System.out.println("La i vale " + i);
}
```

3.3. Salidas anticipadas

Dependiendo de la lógica que implementemos en el programa, puede ser interesante terminar un bucle antes de tiempo y no esperar a que termine por su condición (realizando las iteraciones). Para poder hacer esto disponemos de:

- **Break:** Interrumpe completamente la ejecución del bucle.

- **Continue:** Detiene la iteración actual y continúa con la siguiente.

Cualquier programa puede escribirse sin **break** ni **continue**; se recomienda evitarlos, ya que rompen la secuencia natural de las instrucciones.

```
        i = 1;
        while (i <= 10) {
            System.out.println("La i vale" + );
            if (i == 2) {
                break; /*El bucle esta para valor 10
                        pero con break lo cortamos en 2 */
            }
            i++;
        }
```

```
        i = 0;

        while (i < 10) {
            i++;
            if (i % 2 == 0) { //Si i es par
                continue;
            }
            System.out.println("La i vale " + i);
        }
```

3.4. Bucles anidados

En el uso de los bucles es muy frecuente la anidación, que consiste en incluir un bucle dentro de otro.

Al hacer esto se multiplica el número de veces que se ejecuta el bloque de instrucciones de los bucles internos. Los bucles anidados pueden encontrarse relacionados cuando las variables de los bucles más externos intervienen en el control de la iteración de un bucle interno; o independientes, cuando no existe relación alguna entre ellos.

3.4.1. Bucles independientes

Cuando los bucles anidados no dependen, en absoluto, unos de otros para determinar el número de iteraciones, se denominan **bucles anidados independientes**.

```
        for (i = 1; i <= 4; i++) {
            for (j = 1; j <= 3; j++) { //j va independiente de i
                System.out.println("Ejecutando...");
            }
        }
```

3.4.2 Bucles dependientes

Puede darse el caso de que el número de iteraciones de un bucle interno no sea independiente de la ejecución de los bucles exteriores, y dependa de sus variables de control. Decimos entonces que son **bucles anidados dependientes.**

```
        for (i = 1; i <=3; i++) {
            System.out.println("Bucle externo, i=" + i);
            j = 1;
            while (j <= i) { //j depende del valor de i
                System.out.println("...Bucle interno, j=" + j);
                j++;
            }
        }
```

Funciones / Métodos

4.1 Conceptos Básicos

La solución para cuando necesitamos la misma funcionalidad en distintos lugares de nuestro código no es más que etiquetar con un nombre de fragmento de código y sustituir en el programa dicho fragmento, en todos los lugares donde aparezca, por el nombre que le hemos asignado.

En general, la sintaxis para definir una función es:

```
    static tipo nombreFuncion() {
        cuerpo de la función
    }
```

Definimos algunos conceptos necesarios para seguir trabajando con funciones:

- **Llamada a la función:** Es el nombre de la función, seguido de (). Se convierte en una nueva instrucción que podemos utilizar para invocarla.
- **Prototipo de la función:** Es la declaración de la función, donde se especifica su nombre, el tipo que devuelve y, entre paréntesis, los parámetros de entrada que utiliza “static tipo nombreFuncion()”.
- **Cuerpo de la función:** Es el bloque de código que ejecuta la función cada vez que se invoca y que aparece entre llaves después del prototipo (contenido de la función).
- **Definición de una función:** Está formada por le prototipo más el cuerpo de la función (función al completo).

Con esto evitamos:

- La duplicidad del código: ya que el código se escribe una única vez.
- La dificultad en el mantenimiento, en el caso de realizar modificaciones, solo se realizarán en la función, no en donde se aplica.

El comportamiento de una llamada a una función consiste en:

1. Las instrucciones del programa principal se ejecutan hasta que encuentran la llamada a la función.
2. La ejecución salta a la definición de la función.
3. Se ejecuta el cuerpo de la función.
4. Cuando la ejecución del cuerpo termina, retornamos al punto del programa desde donde se invocó la función.
5. El programa continúa su ejecución.

En la programación estructurada se llaman **funciones**, y en la programación orientada a objetos, se denominan **métodos**.

4.2 Ámbito de las variables

En el cuerpo de la función podemos declarar variables, que se conocen como **variables locales**. Su ámbito es la propia función donde se declaran, no pudiéndose utilizar fuera de ella. Se puede declarar variables con el mismo nombre en **diferentes funciones**.

4.3 Paso de información a una función

Cuando una función necesita conocer información externa para poder llevar a cabo su tarea. Se transmite esa información mediante una llamada a la función, indicando las variables que tienen esa información, por ejemplo “nombreFunción(2)”. Es lo que denominamos un **parámetro de entrada**, no es más que una variable local a la que se le asignan valores en cada llamada.

4.3.1 Valores en la llamada

En la llamada a una función se pueden pasar valores que provienen de literales, expresiones o variables.

```
variosSaludos(2); //llamada con un literal
int n = 3;
variosSaludos(2+n); //llamada con una expresión
```

4.3.2 Parámetros de entrada

Una función puede definirse para recibir tantos datos como necesita. Cada dato utilizado en la llamada a una función será asignado a un parámetro de entrada, especificado en la definición de la función con de la siguiente manera:

```
tipo nombreFuncion(tipo1 parametro1, tipo2 parametro2...) {
    cuerpo de la función
}
```

El primer parámetro de entrada lo hemos llamado parametro1 y se le puede asignar un valor del tipo tipo1, lo mismo ocurre con el resto de parámetros. El número de parámetros definidos en la función determina el número de valores que hay que utilizar en cada llamada.

En Java, los parámetros toman su valor como una copia del valor de la expresión o variable utilizada en la llamada; este mecanismo de paso de parámetros se denomina **paso de parámetros por valor** o **por copia**.

4.4 Valor devuelto por una función

Es posible realizar paso de información de la función a nuestro programa principal, realizando una llamada desde el cuerpo de la función hacia el código don de se hace la llamada principal. Con esto conseguimos que la llamada a una función se convierta en un valor cualquiera. Este puede ser utilizado desde el lugar donde se invoca.

```
int a = suma(2, 3);
int b = suma(7, 1) * 5;
```

Para que la función devuelva el la información, disponemos de la instrucción **return**, que finaliza la ejecución de la función y devuelve el valor indicado.

```
tipo nombreFunción(parámetros) {
    ...
    return (valor);
}
```

Debe existir una concordancia entre el tipo declarado en la función y el tipo del valor devuelto con **return**. Nada impide utilizar varios *return* en una misma función, pero es una práctica desaconsejable, ya que una función debe tener un único punto de entrada y de salida; el uso de varios *return* rompe esta norma.

4.5 Sobrecarga de funciones

Java permite que dos o más funciones compartan el mismo identificador en un mismo programa. Esto es lo que se conoce como **sobrecarga de funciones**. La forma de distinguir entre las distintas funciones sobrecargadas es mediante sus listas de parámetros, que deben ser distintas, ya sean en número o en tipo.

Las funciones sobrecargadas pueden devolver tipos distintos, aunque estos no sirven para distinguir una función sobrecarga de otra.

```
// Función sobrecargada
static int suma(int a, int b) {
    int suma;
    suma = a + b;
    return (suma);
}

// Función sobrecargada
static double suma(int a, double pesoA, int b, double pesoB) {
    double suma;
    suma = a * pesoA / (pesoA + pesoB) + b;
    return (suma);
}
```

4.6 Recursividad

Una función puede ser invocada desde cualquier lugar: desde el programa principal, desde otra función e incluso desde dentro de su propio cuerpo de instrucciones. En este último caso, cuando una función se invoca a sí misma, diremos que es una **función recursiva**.

```
static int funcionRecursiva() {
    if (caso base) {
        resultado = valorBase;
    } else {
        resultado = funcionRecursiva(nuevosDatos); //Llamada recursiva
        ...
    }
    return (resultado);
}
```

Tablas -- Pedro

Una tabla es una variable que permite guardar más de un valor simultáneamente.

Podemos ver una tabla como una variable más grande(supervariable) que engloba a otras variables, llamadas elementos o componentes.

El problema es como diferenciar cada uno de los elementos que constituyen la tabla.

Para poder hacer esto necesitamos asignar un número de orden a cada elemento.

En el momento de crear una tabla, deberemos tener en cuenta lo siguiente:

-El tipo de datos a almacenar y los elementos que necesitamos.

-Declarar una variable para la tabla.

-Crear la propia tabla

Una tabla se define mediante dos características fundamentales: Su longitud(número de elementos que tiene) y su tipo(datos que almacena en todos sus elementos)

Una referencia se le identifica como una dirección de memoria.

La forma de que una variable sepa dónde esta la tabla en la memoria es asignándole una referencia de la primera posición que ocupa.

Las variables de tabla son referencias, también se pueden llamar como variables de referencia.

Las referencias se modifican en cada ejecución, dependiendo de la ocupación de la memoria

Las variables pueden verse como medios para acceder a las tablas a las que referencian.

Se puede acceder a una misma tabla mediante más de una variable (debe de estar referenciada la tabla por las variables)

Si una tabla no está referenciada por ninguna variable, tiene dos problemas:

-La tabla es inútil

-Ocupa espacio en la memoria.

Existe un recolector de basura que comprueba todas las tablas construidas y si encuentra tablas inservibles las elimina dejando libre el espacio ocupado en la memoria.

Hay una forma de asignar una variable a null para que no haga referencia a nada. Ejemplo:

t1 = null; //no referencia a nada

Cuando creamos una tabla, mantiene su longitud constante y no es posible cambiar el número de elementos que contienen.

Para modificar la tabla hay que crear una segunda tabla con el número de elementos necesarios y copiar en ella los datos que nos interesa de la primera tabla.

Si la nueva tabla se referencia con la misma variable que referenciaba a la original, es como si la tabla original hubiera modificado su longitud

Ordenar una tabla consiste en cambiar de posición los datos.

Buscar una tabla consiste en averiguar si entre los elementos de una tabla se encuentra y en que posición se encuentra un valor llamado clave de búsqueda.

La búsqueda secuencial consiste en un recorrido de la tabla donde se comprueban los valores de los elementos.

El algoritmo de búsqueda dicotómica(búsqueda binaria) comprueba si la clave de búsqueda se encuentra en el elemento central de la tabla.

Para poder copiar una tabla debemos de:

-Crear una nueva tabla del mismo tipo y longitud que la tabla original

-Recorrer la tabla originar, copiando el valor de cada elemento en su lugar correspondiente en la tabla destino.

Para poder hacer una inserción en la tabla hemos de ver si está ordenada o no.

Si el orden no importa, incrementamos la longitud de la tabla e insertamos el nuevo dato en el último elemento.

Para poder eliminar un elemento de la tabla tendremos que buscarlo para conocer en que índice se encuentra

Podemos hacer tablas de dos dimensiones, teniendo ahora una tabla con longitud y anchura.

Una tabla bidimensional se denomina matriz.

También hay tablas de tres o más dimensiones.

Cadena de Caracteres – Pedro

Definimos texto como palabra, frase o párrafos de cualquier longitud.

Un texto es una secuencia de caracteres.

Para manipular textos tenemos las clases Character y String.

- Char: Un carácter se define como una letra de cualquier alfabeto, un número, un ideograma o cualquier símbolo.

- Unicode: Es un estándar de codificación de caracteres que identifica a cada carácter mediante un número entero único llamado code point, cuyo valor se puede representar en decimal o hexadecimal.

Para evitar confusión se le antepone la secuencia U+ o \u cuando se representa en hexadecimal y se completan con ceros a la izquierda si es necesario (siempre se usan como mínimo 4 dígitos)

Para codificar cualquier code point necesitamos 3 bytes.

Solo los code points cuyo valor es inferior o igual a 65535 se pueden asignar a un tipo char.

Secuencias de escape:

Se conoce como secuencia de escape a un carácter precedido de una barra invertida (\)

Cada carácter posee un significado especial

Conversión char-int:

Cada code point solo es un número entero.

Que un carácter se identifique con un número crea una relación entre el tipo char y el tipo int.

Se puede asignar un valor entero a un char siempre cuando este entre 0 y 65535 y asignar un carácter a una variable de tipo int.

Ejemplo:

```
int e= ‘a’;
```

Si lo mostramos mostrará 97.

```
char c= ‘a’
System.out.println((int)c);
```

Aritmética de caracteres.

Se pueden realizar operaciones aritméticas entre un carácter y su representación numérica en Unicode.

Clase Character

La clase Character amplía la funcionalidad de char.

Clasificación de caracteres.

Un carácter puede clasificarse dentro de algunos de los grupos siguientes:

Dígitos: Este grupo está formado por los caracteres del 0 al 9

Letras: Formado por el alfabeto tanto en minúscula como en mayúscula.

Caracteres blancos: El espacio, tabular, etc.

Otros caracteres: Signos de puntuación, matemáticos, etc.

Métodos:

boolean isDigit(char c):Indica si c es un dígito. Devuelve true si lo es y false si no.

boolean isLetter(char c): Indica si c es una letra. Devuelve true si lo es y false si no.

boolean isLetterOrDigit(char c): Indica si el carácter es una letra o un dígito.Devuelve true si lo es y false si no.

boolean isLowerCase(char c): Indica si es una letra minúscula. Devuelve true si lo es y false si no.

boolean is UpperCase (char c): Indica si es una letra mayúscula. Devuelve true si lo es y false si no.

boolean is SpaceChar(char c):Indica si c es un espacio.

Devuelve true si lo es y false si no.

boolean isWhitespace(char c):Indica si c es cualquier carácter blanco.Devuelve true si lo es o false si no Estos son:

Espacio en blanco.

Retorno de carro(\r)

Nueva línea(\n)

Tabulador(\t)

Otros

Conversion.

Los métodos que realizan conversiones son aquellos que devuelven transformado el valor que se les pasa como parámetro.

También existen los que realizan la operación inversa(convierten un valor de otro tipo en un carácter).

Conversion entre caracteres.

Son métodos que transforman un carácter en otro

char toLowerCase(char c):Convierte una letra a minúscula

char toUpperCase(char c):Convierte una letra a mayúscula

Clase String.

La clase String manipula conjuntos secuenciales de caracteres,cadenas,etc.

Una variable de tipo String almacenará una cadena de caracteres que viene de manipular otra cadena o de un literal.

Un literal cadena consiste en un texto entre comillas simples.

Los literales carácter y cadena se diferencian en el tipo de comillas usado. “a” es una cadena compuesta por único carácter mientras ‘a’ es un carácter.

Inicialización de cadenas.

Podemos usar new para crear y asignar un valor a una variable String.

También se puede abreviar.

Valores de otros tipos.

Cuando necesitemos representar un valor de un tipo primitivo en forma de cadena representamos como una cadena formada por cada secuencia de caracteres.

El método estático que construye una cadena para representar un valor es:

static String valueOf(tipo valor): Devuelve una cadena con la representación del valor pasado como parámetro.

Comparación.

Los operadores de comparación no se encuentran disponibles directamente para comparar cadenas de caracteres. Tenemos métodos que realizan las comparaciones oportunas.

Igualdad.

Un error común es comparar dos variables de tipo cadena usando ==.

Este operador no se puede usar con String.

Usaremos:

boolean equals(String otra): Compara la cadena con otra.

Devuelve true si son iguales o false si no lo son.

Para comparar cadenas sin tener en cuenta mayúsculas y minúsculas se realiza:

boolean equalsIgnoreCase(String otraCadena): Compara la cadena con otra sin distinguir entre mayúsculas o minúsculas. Devuelve true si son iguales o false si no lo son.

Para comparar un trozo de cadena tenemos:

boolean regionMatches(int incio, String otraCad, int inicioOtra, int longitud):

Compara dos fragmentos de cadenas, el primero es la cadena invocante y comienza con índice inicio y el segundo corresponde a la cadena otraCad y comienza en el carácter con índice inicioOtra.

Devuelve true si son iguales o false si no lo son

boolean regionMatches(boolean ignora, int incio, String otraCad, int inicioOtra, int longitud):

Realiza lo mismo que el anterior método con la diferencia de que si el valor del parámetro que ignora es true, la comprobación se realiza considerando iguales las mayúsculas y minúsculas.

Comparación alfabética.

Otra forma de comparar dos cadenas es alfabéticamente, según el orden de un diccionario.

Los métodos son:

int compareTo(String cadena):

Compara alfabéticamente una cadena y la que se le pasa devolviendo 0 si las cadenas comparadas son iguales, un número negativo si la cadena es menor alfabéticamente que la otra cadena y un número positivo si la cadena es mayor alfabéticamente que la otra cadena.

int compareToIgnoreCase (String cadena):

Realiza lo mismo de antes sin distinguir entre minúsculas o mayúsculas.

Concatenación.

El operador + une o concatena dos cadenas.

Las une pero no inserta nada entre ellas.

Obtención de caracteres.

Todos los caracteres pueden ser identificados mediante la posición que ocupan, cada carácter se número en un índice único que comienza en 0.

Obtención de un carácter

char charAt(int posicion): Devuleve el carácter que ocupa el índice posicion.

Obtención de una subcadena.

Una subcadena es un fragmento de una cadena(subconjunto de caracteres contiguos de una cadena).

String substring(int inicio): Devuelve una subcadena formada desde la posicion inicio hasta el final de la cadena.

Devuelve una copia.

String substring(int inicio, int final): Devuelve una subcadena formada desde la posicion inicio hasta la posicion final de la cadena.

Devuelve una copia.

Para eliminar caracteres blancos usamos:

String strip():

Devuelve una copia de la cadena sin caracteres blancos del principio y del final.

String stripLeading(): igual que strip() pero solo elimina los espacios en blanco del principio.

String stripTrailing(): solo elimina los espacios en blanco del final.

Longitud de una cadena

En ciertos métodos es necesario usar algunos índices para localizar los caracteres que forman una cadena.

Tenemos:

int length(): Devuelve el número de caracteres (longitud)

de una cadena.

Busqueda.

Dentro de una cadena podemos buscar un carácter o una subcadena. Si lo encuentra, devuelve el número del índice donde se encuentra si no, devuelve -1.

int indexOf(int c): Busca la primera ocurrencia de c en la cadena empezando por el principio. Si lo encuentra, devuelve el número del índice donde se encuentra si no, devuelve -1.

int indexOf(String cadena): Busca la primera ocurrencia de una cadena.

int indexOf(int c, int inicio): Busca la primera ocurrencia de c a partir de inicio.

int indexOf(String cadena): Busca la primera ocurrencia de una cadena a partir de inicio.

int lastIndexOf(int c): Devuelve el índice de la última ocurrencia de c o -1 si no se encuentra.

int lastIndexOf(String cadena): Devuelve el índice de la última ocurrencia de la cadena o -1 si no se encuentra.

int lastIndexOf(int c, int inicio): Devuelve el índice de la última ocurrencia de c a partir de inicio buscando desde la última posicion o -1 si no se encuentra.

int lastIndexOf(String cadena, int inicio): Devuelve el índice de la última ocurrencia de la cadena a partir de inicio buscando desde la última posicion o -1 si no se encuentra.

Comprobaciones.

Es posible realizar comprobaciones con una cadena de caracteres, por ejemplo si esta vacía, si tiene cierta subcadena, etc.

Cadena vacía.

Una cadena vacía es aquella que no está formada por ningún carácter y se representa mediante “”.

Su longitud es 0.

Para comprobar si es una cadena vacía:

boolean isEmpty():

Indica mediante true si está vacía o no si no lo está.

Contiene.

Si una cadena contiene otra subcadena se usa:

boolean contains(CharSequence subcadena): Devuelve true si se encuentra la subcadena en cualquier posición.

Prefijos y sufijos.

Los prefijos y sufijos son subcadenas que van al principio o al final de una cadena. Para comprobar si hay prefijos o sufijos se usa:

boolean startsWith(String prefijo): Comprueba si la cadena invocada comienza con la cadena prefijo.Devuelve true si asi es o false si no es.

boolean startsWith(String prefijo, int inicio): Comprueba si la cadena invocada comienza con la cadena prefijo comenzando desde la posición incio..Devuelve true si asi es o false si no es.

boolean endsWith(String sufijo): Indica si la cadena termina con el sufijo que le pasamos.

Conversión.

Una cadena puede transformarse sustituyendo todas las letras que la componen a minúsculas o mayúsculas. Para ello, usamos los siguientes métodos.

String toLowerCase(): Devuelve una copia de la cadena en letras minúsculas

String toUpperCase() Devuelve una copia de la cadena en letras mayúsculas

String replace(char original, char otro): Devuelve una copia de la cadena invocante donde se han sustituido todos los caracteres del carácter original por otro.

Separación en partes.

Una cadena se puede descomponer en partes si definimos un separador. Usamos:

String[] split(String separador): Devuelve las subcadenas resultantes de dividir la cadena invocante con el separador pasado como parámetro.

Cadenas y tablas de caracteres.

Existe una relación entre las cadenas,clase String y las tablas de caracteres char[]. En aquellas ocasiones en que interese manipular o cambiar de lugar los caracteres dentro de una cadena resulta más cómodo trabajar con una tabla.

El método que crea una tabla de caracteres tomando como base una cadena es:

char[] to CharArray(): Crea y devuelve una tabla de caracteres con el contenido de la cadena invocante.

El método que realiza lo inverso es:

static String valueOf(char[] tabla): Devuelve un String con el contenido de la tabla de caracteres.

static String valueOf(char[] tabla, int inicio, int cuantos): Devuelve un String con el contenido de la tabla de caracteres con la diferencia de que devuelve la cadena formada por un subconjunto de caracteres consecutivos de la tabla t. inicio es el índice el primer elemento de la tabla y cuantos determina el número de caracteres que compondrán la cadena.

Clases – Ismael

Definicion de Clase y conceptos básicos

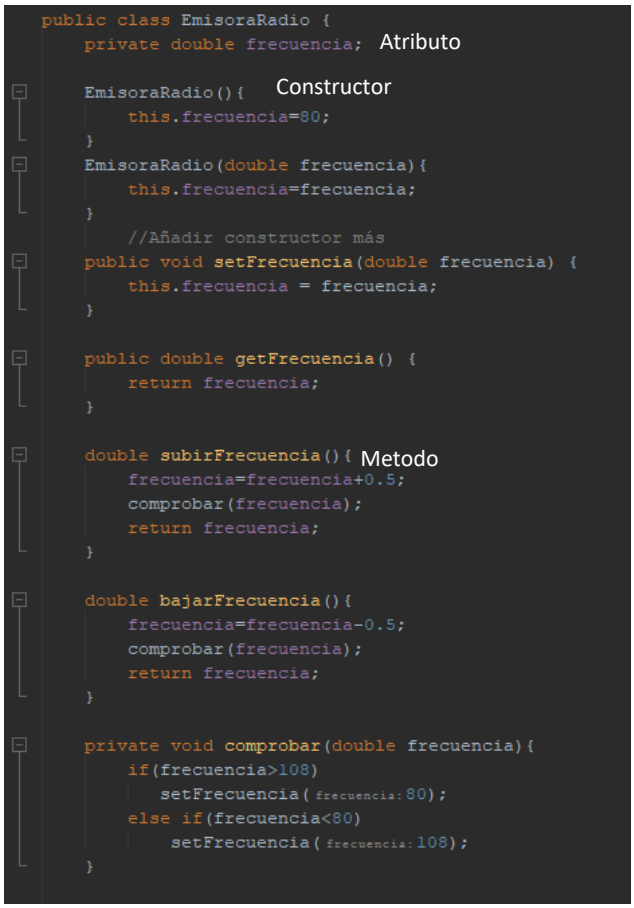
➢ **Clase:** Descripción abstracta de elementos genéricos relevantes en nuestro sistema.
Por ejemplo: Persona, Libreta pueden ser clases necesarias en la aplicación para un banco.

La descripción de la clase incluye atributos y métodos.
Una clase es una plantilla (un molde) para construir objetos. Cuando se crea un objeto (instanciación) se tiene que usar una clase y el objeto pertenecerá a esa clase.
De esta manera el compilador comprenderá las características del objeto.

➢ **Atributos:** Son las características de la clase, sus propiedades, los datos que definen al objeto (variables de clase en Java). A la hora de determinar los atributos hay que procurar que sean lo más genéricos posible. Atributos de la clase Persona podrían ser nombre, edad y dni.

➢ **Métodos:** Definen el comportamiento (las cosas que puede hacer) de la clase (métodos en Java). Por ejemplo, una de las cosas que puede hacer una persona es presentarse ("Hola, me llamo Daniel") o informarte de su edad ("Tengo 27 años").

Todas las clases tienen al menos un **método constructor**, que es el que se usa para generar un objeto de esa clase.
El constructor tiene unas reglas muy precisas: Debe ser público. Su nombre coincide exactamente con el de la clase. No devuelve ningún valor (ni siquiera void).




```java
public class EmisoraRadio {
    private double frecuencia;

    EmisoraRadio(){
        this.frecuencia=80;
    }
    EmisoraRadio(double frecuencia){
        this.frecuencia=frecuencia;
    }
        //Añadir constructor más
    public void setFrecuencia(double frecuencia) {
        this.frecuencia = frecuencia;
    }

    public double getFrecuencia() {
        return frecuencia;
    }

    double subirFrecuencia(){
        frecuencia=frecuencia+0.5;
        comprobar(frecuencia);
        return frecuencia;
    }

    double bajarFrecuencia(){
        frecuencia=frecuencia-0.5;
        comprobar(frecuencia);
        return frecuencia;
    }

    private void comprobar(double frecuencia){
        if(frecuencia>108)
           setFrecuencia( frecuencia: 80);
        else if(frecuencia<80)
            setFrecuencia( frecuencia: 108);
    }
```

Inicializacion de objetos

Un ejemplo de inicialización de objetos es asignando un valor por defecto.
Esto se haría creando un constructor por defecto asignándole un valor por defecto a los atributos de la clase.
Ejemplo de constructor por defecto:

```java
public Calendario(){
    this.dia=1;
    this.mes=1;
    this.año=2000;
}
```

Estos valores se podrán modificar durante la ejecución, salvo que estos sean declarados con el modificador “final”, que en ese caso sería una constante, un atributo que no se puede modificar durante la ejecución

Objetos

Objeto es la denominación a los elementos que pertenecen a una clase. Una clase solo es el molde para crear objetos.

Cada objeto tiene sus propios valores de los atributos definidos en la clase. Nuestro sistema no trabajará con las clases, si no que utilizará los objetos.

Variables de referencia

Son los nombres utilizados para inicializar un objeto. Estos siguen las mismas reglas que las variables para los tipos primitivos.

```
Calendario fechal;
```

En este ejemplo **fecha1** sería una variable de referencia de la clase **Calendario**

Operador new

El operador **new** es el encargado de crear objetos y tablas.

```
fechal=new Calendario();
```

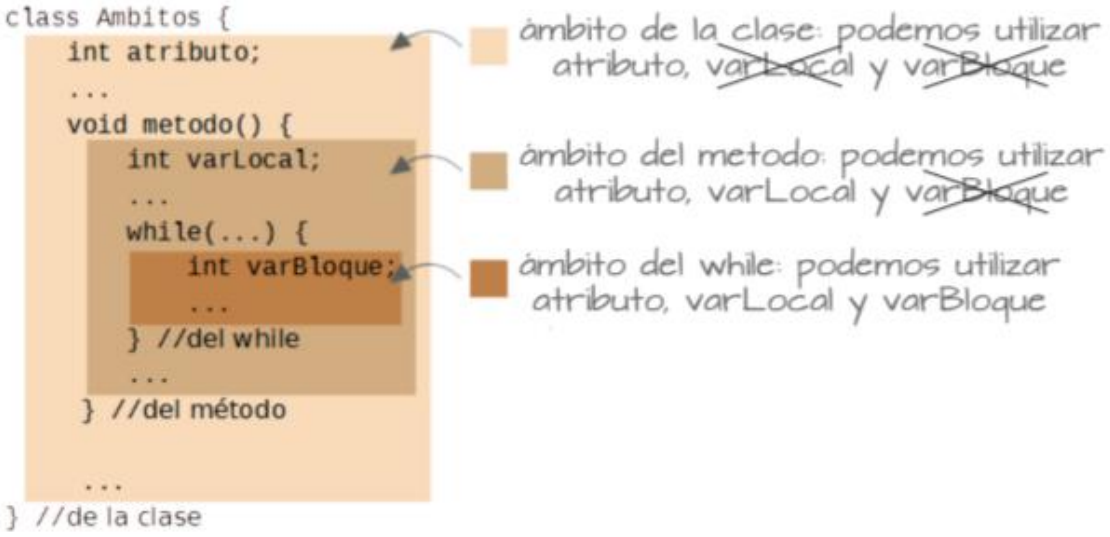
En este caso se crea el objeto **fecha1** , siendo esté del tipo **Calendario**.

Referencia null

El valor literal **null** es una referencia nula. Esta se usa al declarar una variable de referencia, ya que esta inicializa por defecto a **null.**

Ámbitos de variables

- **Ámbito de clase:** Son las variables declaradas como globales o atributos si hablamos de clases. Estás pueden usarse en cualquier lugar de la clase
- **Ámbito de método:** Son variables creadas en un método/función. Estas se pueden utilizar en todo el método, pero no fuera de este
- **Ámbito de estructura de control:** Son las variables que solo se pueden utilizar dentro de una estructura. Por ejemplo en un **for** , la **“i”** es una variable de este tipo



Ocultacion de atributos

En POO hay una excepción sobre cuando una var. Local tiene el mismo identificador que un atributo de clase.
En este caso se dice que dentro variable local tiene prioridad sobre el atributo, ya que accede antes a la var local, produciendo la ocultación de atributos.

Objeto this

La palabra reservada **this** permite utilizar el atributo cuando ha sido ocultado por una variable local. Esta palabra reservada se utilizar para referenciarse a sí mismo como yo.

Al escribir this en el ámbito de una clase se interpreta como la propia clase, y permite acceder a los atributos aunque se encuentren ocultos

Modificadores de acceso

Son palabras reservadas que se utilizan para controlar la visibilidad de clases entre paquetes.

- **Clases vecinas**: cuando ambas clases pertenecen al mismo paquete
- **Clases externas**: cuando la clase se ha definido en paquetes distintos

Tipos de visibilidades

- **Visibilidad por defecto**: Cuando se define la clase sin ningún modificador de acceso. Estas clases solo serán visibles dentro del mismo paquete donde se encuentran.
- **Visibilidad total**: Utilizan el modificador **public**. Estas serán visibles desde otros paquetes. Estas se llamarán a otros paquetes mediante un **import**
- **Visibilidad "privada" *:** Se utiliza el modificador **private.** Estas solo serán visibles dentro de la clase, es decir si un método tiene el modificador private no se podrá. modificar fuera de la clase que lo contiene.

***Está visibilidad no sale definida como tal con un nombre en el tema**

	Visible desde…		
	la propia clase	clases vecinas	clases externas
private	✓		
sin modificador	✓	✓	
public	✓	✓	✓

Métodos get/set

Set

Se utiliza para asignar un valor a un atributo de clase

Get

Se utiliza para ver el valor del atributo, ya que este devuelve el atributo únicamente

La principal de ventaja de utilizar estos métodos es la encapsulacion de la clase que los contiene, ya que permite controlar que atributos son accesible para lectura y cuáles para escritura, así como los valores asignados.