

# Apuntes de Java

migueluisV

Realizadas: Abril 2022

# Índice

<b>1</b>	<b>Conceptos básicos</b>	<b>4</b>
<b>2</b>	<b>Comentarios</b>	<b>4</b>
<b>3</b>	<b>Clases importadas</b>	<b>5</b>
<b>4</b>	<b>Variables</b>	<b>5</b>
4.1	Concatenar cadenas . . . . .	5
4.2	Reglas para nombrar variables . . . . .	6
<b>5</b>	<b>Operadores primitivos</b>	<b>6</b>
5.1	Operador de incremento y decremento . . . . .	6
5.2	Operador de asignación . . . . .	7
<b>6</b>	<b>Asignación de valores a variables</b>	<b>7</b>
<b>7</b>	<b>Condicionales</b>	<b>9</b>
<b>8</b>	<b>Operadores lógicos</b>	<b>10</b>
<b>9</b>	<b>Ciclos</b>	<b>10</b>
9.1	Ciclo while . . . . .	10
9.2	Ciclo for . . . . .	11
9.3	Ciclo do-while . . . . .	11
9.4	Controles de ciclos . . . . .	11
<b>10</b>	<b>Condicionales switch</b>	<b>12</b>
10.1	Expresión switch . . . . .	13
<b>11</b>	<b>Arreglos</b>	<b>14</b>
11.1	Ciclo mejorado para arreglos . . . . .	15
11.2	Arreglos multidimensionales . . . . .	15
<b>12</b>	<b>POO</b>	<b>16</b>
12.1	Clases y objetos . . . . .	16
12.1.1	Creando una clase en VSCode . . . . .	16
12.1.2	Creando una clase por código . . . . .	16
12.1.3	Creando un objeto . . . . .	17
12.1.4	Métodos . . . . .	17
12.1.5	Atributos de clases . . . . .	18
12.1.6	Modificadores de acceso . . . . .	19
12.1.7	Tipo de retorno de métodos . . . . .	20
12.1.8	Tipos de valores y referencias . . . . .	21
12.1.9	Constructores . . . . .	22
12.1.10	Static y tipos constantes <i>Final</i> . . . . .	22

12.2	Conceptos de POO . . . . .	23
12.2.1	Encapsulación . . . . .	23
12.2.2	Herencia . . . . .	24
12.2.3	Polimorfismo . . . . .	24
12.2.4	Abstracción . . . . .	26
<b>13</b>	<b>Métodos especiales</b>	<b>28</b>
13.1	Getters y Setters . . . . .	28
13.2	La clase Math . . . . .	28
13.3	El método <i>equals()</i> . . . . .	29
13.4	Enums . . . . .	30
<b>14</b>	<b>Paquetes</b>	<b>31</b>
14.1	API de Java . . . . .	31
<b>15</b>	<b>Interfaces</b>	<b>32</b>
<b>16</b>	<b>Casting</b>	<b>33</b>
16.1	Upcasting y Downcasting . . . . .	33
<b>17</b>	<b>Clases anónimas</b>	<b>34</b>
<b>18</b>	<b>Clases anidadas</b>	<b>34</b>
<b>19</b>	<b>Excepciones</b>	<b>36</b>
19.1	Manejo de múltiples excepciones . . . . .	36
19.2	Excepciones lanzadas (throw) . . . . .	37
19.3	Excepciones Unchecked y Checked . . . . .	37
<b>20</b>	<b>Listas</b>	<b>37</b>
20.1	ArrayList . . . . .	38
20.2	LinkedLists . . . . .	39
20.3	HashMap . . . . .	39
20.4	Conjuntos . . . . .	40
20.4.1	LinkedHashMaps . . . . .	40
20.5	Organizando listas . . . . .	40
20.6	Iterators . . . . .	41
<b>21</b>	<b>Hilos</b>	<b>42</b>
<b>22</b>	<b>Trabajando con archivos</b>	<b>43</b>
22.1	Lectura . . . . .	44
22.2	Creación y escritura . . . . .	44

# 1 Conceptos básicos

- Todos los programas de Java están contenidos dentro de una **clase** con algún nombre en particular, como **App**, **Program** o un nombre personalizado.
- Todos los programas de Java tienen un punto de partida, un procedimiento llamado **main** (como en C#).
- El lenguaje incluye temas como:
  - Indicadores de acceso.
  - Tipo de retorno de métodos (procedimientos y funciones).
  - Formas de creación de métodos (estática o no).
  - Parámetros en funciones y procedimientos.
- Cada instrucción debe terminar con un punto y coma (;).
- Cada clase, función u objeto debe abrirse con llaves ({}).

Esta es la estructura inicial o básica de un programa en Java:

```
public class App {  
    public static void main(String[] args) throws Exception {  
        System.out.println("hola mundo");  
    }  
}
```

# 2 Comentarios

Java soporta el poder hacer comentarios de una y varias líneas:

Se usan dos diagonales (//) para comentar una línea de código.

```
// Esto es un comentario.  
x = 5; // Variable comentada.
```

Se usa una diagonal y un asterisco (/\*) al inicio del bloque y un asterisco y diagonal (\*/) al final para comentar varias líneas o un bloque de código.

```
/* System.out.println("Hola mundo");  
x = 5;  
Tres líneas comentadas. */
```

Una herramienta del *Java Development Kit* es que se puede generar documentación automática de un proyecto Java en formato HTML, esta documentación toma el código fuente y lo muestra más presentable para que otro desarrollador entienda el programa. Utilizando los **comentarios de documentación** podemos hacer que esta documentación generada esté aún más presentable, por así decirlo, si insertamos comentarios de documentación en cada clase, función, procedimiento, declaración de variable, cuando generemos la documentación, veremos qué ocurre en cada parte donde pusimos un comentario de este tipo. Para insertar

un comentario de documentación, utilizamos una diagonal y dos asteriscos (/\*\*) al inicio de la instrucción y una diagonal y un asterisco al final (\*/).

```
/** Inicia una clase. */  
class Prueba {}
```

Por último, si escribimos una diagonal y más de un asterisco (/\*\*\*\*\*...), Java lo interpreta como que se quiere hacer un recuadro o una separación entre partes o bloques de código, como se puede ver a continuación:

```
/*  
Comienzo de un método.  
*/
```

### 3 Clases importadas

Podemos importar clases especiales u externas para volver más rico el programa de java, esto es como las librerías o cabeceras en otros lenguajes.

La palabra reservada a utilizar es **import**, seguido de la clase a importar, por ejemplo: java.util.Scanner.

### 4 Variables

Los tipos de datos existentes en Java son:

- **int**: datos numéricos enteros.
- **double**: datos numéricos con punto decimal.
- **String**: una cadena de texto.
- **char**: un único carácter.
- **boolean**: booleano, almacena si o no.

Recordemos que hay variaciones con estas variables, podemos crear una variable entera que tenga menos espacio de bits para almacenar un número (byte y short), lo mismo ocurre con los flotantes (float) o hacer que acepten muchos bits (long).

Se puede poner una coma entre cada variable declarada del mismo tipo para hacer una declaración de variables en una sola línea:

```
double n1, n2, n3, n4, n5, prom;
```

#### 4.1 Concatenar cadenas

Se utiliza el **operador** + para concatenar (unir) más de una cadena.

```
System.out.println("hola" + "mundo" + "en" + "Java");
```

## 4.2 Reglas para nombrar variables

Para administrar mejor el nombre de las variables se pueden seguir los siguientes consejos:

- Ponerle una letra mayúscula o un guión bajo (\_).
- Ponerle un nombre significativo que se pueda recordar fácilmente.
- No se permiten caracteres especiales o espacios en blanco.
- No se permiten palabras reservados como nombres de variables.
- C++ distingue entre mayúsculas y minúsculas, para tener cuidado nombrando así las variables.

## 5 Operadores primitivos

Las operaciones matemáticas que se pueden realizar en Java son las básicas: **igualdad** (=), **suma** (+), **resta** (-), **multiplicación** (\*), **división** (/) y **modulo** (%), estas operaciones se realizan de la misma manera en la que se hacen en ecuaciones u operaciones algebraicas.

```
int num1 = 1, num2 = 2, num3 = 3;
int suma = num1 + num2 + num3;
int resta = num3 - num1 - num2;
int multi = 5 * 4 * num3; // Ejemplos de operaciones matemáticas con los operandos.
int div = num1 / num2; // El valor almacenado en div es un entero, si se busca que se
    almacene el valor decimal real se requiere un float o double.
int mod = num2 % 2; // Almacena el residuo de una división.
```

### 5.1 Operador de incremento y decremento

Como el otros lenguajes, puedes utilizar los operadores de incremento y decremento en variables, se pueden realizar estas operaciones por medio de un **prefijo** o **posfijo**:

- Con prefijo, primero se incrementa o decrementa el valor de la variable, luego se utiliza en un ciclo u operación que se de.

```
int x = 5;
int y = ++x; // Incrementa x a 6 y a la variable y se le asigna 6.
```

- Con posfijo, primero se utiliza el valor de la variable en un ciclo u operación, luego se incrementa o decrementa el valor.

```
int x = 5;
int y = x++; // A la variable y se le asigna 5 y después incrementa x a 6.
```

## 5.2 Operador de asignación

Este operador se utiliza en operaciones o variables acumulativas (como para sumar varias calificaciones y luego calcular un promedio) donde a una variable se le asigna su propio contenido, más otro valor, o para incrementar o decrementar una variable más de una unidad o 1, estos operadores de asignación aplican para los operandos existentes:

```
int os = 1, or = 2, om = 3, od = 4, om = 5;
os += 1; // os = 1 + 1.
or -= 3; // or = 2 - 3.
om *= 5; // om = 3 * 5.
od /= 7; // od = 4 / 7.
om %= 9; // om = 5 % 9.
```

## 6 Asignación de valores a variables

Para dar una entrada de datos en Java, existen varias formas, en esta ocasión, utilizaremos una clase importada para realizar dicha acción. Se debe agregar una clase especial para realizar entrada de datos en un proyecto de Java: la clase **Scanner** y también debemos crear un objeto de la misma para trabajar con esta clase:

```
import java.util.Scanner;
Scanner var = new Scanner(System.in);
```

Esta clase contiene distintos procedimientos para leer entradas de un tipo de dato específico:

- **nextByte()**: lee un entero byte.
- **nextShort()**: lee un entero short.
- **nextInt()**: lee un entero.
- **nextLong()**: lee un decimal long.
- **nextFloat()**: lee un decimal float.
- **nextDouble()**: lee un decimal double.
- **nextBoolean()**: lee una entrada booleana.
- **nextLine()**: lee toda una cadena de texto.
- **next()**: lee únicamente una palabra de una cadena con varias palabras.

```
import java.util.Scanner; //Importa la clase al proyecto.

class App {
    public static void main(String[ ] args) {
        String variable;
        Scanner myVar = new Scanner(System.in); //Crea un objet de la clase importada.
```

```
        variable = myVar.nextLine(); //Asigna lo que se lea en la entrada a una variable
        tipo string.
        System.out.println(myVar.nextLine()); //Despliega lo que se lea en la entrada de
        datos.
        System.out.println(variable); //Despliega la variable string.
    }
}
```



## 7 Condicionales

Las estructuras condicionales se utilizan para realizar una acción dependiendo de alguna condición, existen condicionales dentro de condicionales (anidadas) y condicionales con estructura si- sino si. La estructura básica consiste en la palabra reservada **if**, seguido de la condición entre paréntesis, y unas llaves de apertura y cierre donde se contiene el código:

```
if (Condición){
    // Código.
}
```

Los **operadores condicionales** existentes son:

- Mayor que: `>`
- Menor que: `<`
- Mayor o igual que: `>=`
- Menor o igual que: `<=`
- No es igual: `!=`
- Es igual: `==`

```
class App{
    public static void main(String[] args) throws Exception {
        int num = 5, edad = 16;

        if (num == 5){ // Condicional if sencilla.
            System.out.println("si es igual");
        }

        if (num > 5){ // Condicional if-else.
            System.out.println("es mayor");
        }else{
            System.out.println("es menor");
        }

        if (edad <= 10){ // Condicional anidada.
            System.out.println("tiene menos de diez años");
        }else{
            if (edad > 10 && edad < 18){
                System.out.println("tiene entre diez y dieciocho años");
            }else{
                System.out.println("es mayor de edad");
            }
        }

        if (num == 5){ // Condicional si-sino si.
            System.out.println("es cinco");
        }else if (num > 5){
            System.out.println("es mayor a cinco");
        }
    }
}
```

```
        }else{
            System.out.println("es menor a cinco");
        }
    }
}
```

## 8 Operadores lógicos

Los **operadores lógicos** son utilizados para juntar varias condiciones y que estas sean cumplidas, por ejemplos, queremos mostrar el mensaje "el número está entre 10 y 20" solamente si un número está entre 10 Y el 20, se deben cumplir ambas condiciones para que se muestre el mensaje deseado. Los *operadores lógicos* disponibles son:

- **Y:** AND (&&)
- **O:** OR (—)
- **No:** NOT (!)

Recordemos que, si hay tres condiciones utilizando AND en una condicional, y una de ellas no se cumple, toda la condicional no se cumple, en caso de que se utiliza OR en la condicional, y solamente una de ellas se cumple, toda la condicional se cumple. El caso especial NOT indica que, si una condición se cumple en una condicional, el operador NOT la cancela, en caso de que no se cumpla, la cumple.

## 9 Ciclos

Un ciclo es un bloque de código que se ejecuta hasta que una condición es cumplida. Pueden ser usados para obtener datos de los usuarios repetidas veces, desplegar información de un arreglo o lista, entre otras cosas.

### 9.1 Ciclo while

Un **ciclo while** se repetirá hasta que una condición se cumpla, primero se evalúa la condición y luego se entra al ciclo. Su estructura es:

```
while (Condición){
    // Código.
    // Instrucción de incremento, decremento que dé salida al ciclo, sino, será infinito.
}
```

Ejemplo:

```
int x = 0;
while (x < 5){
    System.out.println(x);
}
```

## 9.2 Ciclo for

Un **ciclo for** se repetirá un número determinado de veces, se establece un contador, una condición para dar salida al ciclo, y un incremento o decremento del contador, todo en una misma instrucción. Su estructura es:

```
while (Contador/Inicialización; Condición; Incremento/Decremento){  
    // Código.  
}
```

Ejemplo:

```
for (int x = 0; x < 5; x++){  
    System.out.println(x);  
}
```

## 9.3 Ciclo do-while

Un **ciclo do-while** es parecido al while, solo que, en este caso, primero se entra al ciclo y luego se evalúa la condición del ciclo, es decir, el ciclo while primero evalúa, dependiendo de esta evaluación, deja entrar o no al ciclo, por otro lado, el ciclo do-while deja entrar al ciclo por lo menos una vez, luego evalúa la condición. Su estructura es:

```
do {  
    // Código.  
    // Instrucción de incremento, decremento que dé salida al ciclo, sino, será infinito.  
} while (Condición)
```

Ejemplo:

```
int x = 0;  
do {  
    System.out.println(x);  
    x++;  
} while (x < 5);
```

*Nota:* no olvidar poner el punto y coma (;) después de la condición entre paréntesis.

## 9.4 Controles de ciclos

Podemos utilizar las siguientes palabras reservadas para controlar el flujo de los ciclos:

- **break:** si se escribe esta instrucción dentro del ciclo, lo rompe o termina, y pasa directamente a la siguiente instrucción después del ciclo.
- **continue:** si se escribe esta instrucción dentro del ciclo, ignora el resto de código a partir de donde se escribió la instrucción, y re evalúa la condición del ciclo, en otras palabras, se salta una iteración.

Ejemplos:

```
int x = 1;

while (x < 10){ // Ciclo que se repite 10 veces.
    System.out.println(x); // Imprime x.
    if (x == 6){ // Si x vale 6.
        break; // Rompe y se sale del ciclo.
    }
    x++;
}
System.out.println(x); // Ejecuta esta instrucción después de salirse del ciclo.

for (x = 10; x <= 50; x += 10){ // Ciclo que se repite hasta que x sea menor igual a 50 y
    avance de 10 en 10.
    if (x == 20){ // Si x vale 20, se salta el resto de instrucciones después de continue
        dentro del ciclo.
        continue;
    }
    System.out.println(x); // Imprime x.
}
```

## 10 Condicional switch

La condicional **switch** evalúa una variable en base a un conjunto de valores y se le asigna un comportamiento o tarea a hacer, los componentes de un *switch* son:

- La entrada: se utiliza la palabra reservada *switch*, seguido de paréntesis, los cuales contendrán la variable a evaluar. En seguida se abren llaves.
- Los casos: son el conjunto de valores con los que se evaluará la variable. Se usa la palabra reservada **case** y **dos puntos** (:).
- El código para un caso exitoso: va después de *case (valor):*, simplemente es código que se realizará en caso de que el caso y el valor de la variable coincidan.
- Instrucción **break**: si esta palabra reservada no se escribe, la condicional switch continuará evaluando la variable por el resto de los casos.
- El caso **default**: esta palabra reservada es utilizada en caso de que la variable evaluada no coincida con ninguno de los casos. No es necesario utilizar el *break*.

La estructura de un switch es:

```
switch (variable){
    case valor1:
        // Código.
        break;
    case valor2:
        // Código.
        break;
    .
}
```

```
.  
.   
case valorn:   
    // Código.   
    break;   
default: // Este caso es opcional.   
    // Código.   
}
```

## 10.1 Expresión switch

Se puede utilizar una **expresión de asignación switch** para asignarle un valor a una variable en base a una estructura switch y sus caso. Lo primero que debemos hacer es tener una variable auxiliar que será utilizada para evaluar en el switch, y en base a los casos existentes, se le asigna un valor a una variable. Ejemplo:

```
import java.util.Scanner; // Importa la clase para entrada de datos.   
  
public class Program   
{   
    public static void main(String[] args) {   
        Scanner sc = new Scanner(System.in); // Crea un objeto de la clase Scanner.   
        int day = sc.nextInt(); // Asignación de un valor a la variable.   
        String dayType = switch(day) { // Asignación de un switch a la variable.   
            case 1, 2, 3, 4, 5 -> "Working day"; // En caso de que el valor de day esté   
                entre 1 y 5, se le asigna una cadena a la variable.   
            case 6, 7 -> "Weekend"; // En caso de que el valor de day sea 6 o 7, se le   
                asigna una otra cadena en la variable.   
            default -> "Invalid day"; // En caso de que no coincida ningún caso con el valor   
                de day, se le asigna otra cadena en la variable.   
        };   
        System.out.println(dayType); // Despliegue de resultados.   
    }   
}
```

*Nota:* hay que darse cuenta que se utiliza `->` para trabajar con los casos, y ninguno de estos tiene la palabra *break*.

## 11 Arreglos

Para declarar un **arreglo sencillo**, es necesario seguir la siguiente estructura:

```
<Tipo dato>[] <Nombre>;  
<Tipo dato>[] <Nombre> = new <Tipo dato>[Núm. de elementos];  
  
int[] numbs = new int[10];
```

Un arreglo tiene **índices**, los cuales hacen referencia a los elementos del arreglo; si un arreglo tiene 10 elementos, su índice va de 0 a 9 (1 - 10), por lo que, si queremos ver o utilizar el contenido del elemento 6 del arreglo, debemos utilizar la siguiente instrucción:

```
numbs[5] = 1;
```

Para **inicializar un arreglo** con tipo de datos nativos utilizamos la siguiente estructura:

```
<Tipo dato>[] <Nombre> = {<Dato1>, <Dato2>, <Dato3>, ..., <DatoN>};  
String[] nombs = {'Luis', 'Mario', 'Enrique'};
```

*La propiedad length:* esta propiedad de arreglos te permite conocer la cantidad de elementos de un arreglo, basta con escribir el nombre del arreglo, seguido de .length y nos regresará como resultado el número de elementos (entero):

```
int[] numbs = new int[5];  
System.out.println(numbs.length); // Imprime 5.
```

Un ejemplo de el uso de arreglos para determinar la cantidad de elementos de un arreglo, que el usuario ingrese los datos, y se calcule la suma de estos:

```
import java.util.Scanner;  
  
public class Main {  
  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        int i, res = 0; // Declaración y asignación de valores a las variables.  
        int length = scanner.nextInt(); // Asignación de un valor al largo del arreglo.  
        int[] array = new int[length]; // Declaración del arreglo.  
        for (i = 0; i < array.length; i++) {  
            array[i] = scanner.nextInt(); // Asignación de un valor a los elementos del arreglo  
            por medio de un ciclo for.  
        }  
  
        for (i = 0; i < array.length; i++){  
            res += array[i]; // Suma acumulativa de los elementos del arreglo por medio de un  
            ciclo for.  
        }  
        System.out.println(res); // Imprime la suma.  
    }  
}
```

## 11.1 Ciclo mejorado para arreglos

Así como el **foreach** en CSharp, en Java existe un ciclo que se encarga de recorrer los elementos de un arreglo sin que exista la posibilidad de errores y se ve más limpio, es llamado **ciclo mejorado** o **ciclo for each**, su estructura es la siguiente:

```
int[]nums = {1, 2, 3, 4, 5}; // Declaración y asignación de valores al arreglo.

for(int i: nums){ // for (variable para navegar en el arreglo : nombre del arreglo).
    System.out.println(i); // Instrucciones.
}
```

*Nota:* el tipo de dato de la variable de navegación debe de coincidir con el tipo de dato de los elementos del arreglo.

## 11.2 Arreglos multidimensionales

Los arreglos pueden tener más de una **dimensión**, si es de una dimensión, podemos verlo como una sola fila de elementos; si es de dos dimensiones, podemos verlo como una tabla con columnas y filas; si es de tres dimensiones, podemos verlo como un cubo, con filas, columnas y profundidad.

Entonces, para declarar un arreglo multidimensional debemos seguir la siguiente estructura:

```
<Tipo dato>[][] <Nombre>;
<Tipo dato>[][] <Nombre> = new <Tipo dato>[Núm. de elementos][Núm. de elementos];

int[][] numbs = new int[3][3];
```

Vemos que el ejemplo anterior hay dos pares de corchetes cuadrados, lo cual significa que el arreglo es de dimensión dos, si hubiera tres pares, sería de dimensión tres. Para poder **inicializarlo** seguimos el siguiente ejemplo:

```
int[][] numbs = {{1,3,5}, {2,4,6}, {7,8,9}};
```

Para un arreglo de dimensión dos, el primer par de corchetes representa el número de columnas que tendrá, y el segundo par de corchetes son la cantidad de filas, por ende, en la inicialización previa, vemos que el arreglo bidimensional es de 3x3, porque hay tres pares de llaves (filas) con tres elementos en cada uno (columnas). Para acceder a los elementos del arreglo también se utilizan índices base 0, y seguimos la lógica de los pares de corchetes cuadrados explicado anteriormente:

*System.out.println(numbs[0, 0];)*

## 12 POO

Java trabaja con la programación orientada a objetos, es decir, clases, objetos, métodos y atributos.

### 12.1 Clases y objetos

Una clase es una plantilla que representa un objeto, situación o estado de la vida real, una clase es una representación de un objeto de la realidad. Una clase está constituida por sus **atributos** (características), **comportamiento** (métodos, que son lo mismo que funciones y procedimientos), **acceso a las clases** (modificadores de acceso), **objetos** (objeto naciente o generado a partir de la clase, con el cual se va a trabajar), **constructores y destructores** (estado cuando se crea un objeto y cuando se elimina), entre otras consideraciones que veremos más adelante.

#### 12.1.1 Creando una clase en VSCode

Dependiendo del editor de texto o IDE donde se esté trabajando, hay distintos pasos para crear una clase de manera visual o por medio de formularios, sin embargo, la forma para hacer en VSCode es ubicándonos en el árbol de trabajo del proyecto en el Explorador lateral, vemos que una de las carpetas del proyecto en Java es *src*, damos clic derecho sobre esa carpeta, y damos clic a la opción de *New File*, le damos el nombre que queramos al archivo, pero tiene que tener una terminación **.java**; una vez hecho esto, tenemos una clase creada.

#### 12.1.2 Creando una clase por código

Si no se está trabajando en VSCode, en el mismo archivo App de código fuente de nuestro código, escribimos lo siguiente:

```
<Modificador acceso> class <Nombre> {  
    // Código.  
}  
  
public class Persona {  
    // Código.  
}
```

Los *modificadores de acceso* se verán más adelante. Ya con nuestra clase creada, le crearemos un método que simplemente despliega el mensaje *Una persona camina*:

```
public class Persona{  
    void Caminar(){  
        System.out.println("Una persona camina");  
    }  
}
```

*Nota:* el modificador de acceso es opcional, tanto en la declaración de la clase, como en la de sus métodos y atributos; si no se le define un modificador de acceso, la clase y su contenido será **public**.



### 12.1.3 Creando un objeto

Con esta clase y método, podemos crear un objeto de la misma para acceder a su contenido. La estructura para crear un objeto es:

```
<Clase> <Nombre> = new <clase>(); // Declara objeto.  
<Nombre>.<Método>;
```

De esta manera, creamos el objeto *persona* de la clase *Persona* y, por medio de un punto (.), podemos utilizar su método *Caminar*:

```
public class Persona{  
    void Caminar(){  
        System.out.println("Una persona camina");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Persona persona = new Persona();  
        persona.Caminar();  
    }  
}
```

### 12.1.4 Métodos

Los métodos definen el comportamiento de una clase y, por ende, sus objetos. Podemos definir los los métodos de una clase por medio de la siguiente estructura:

```
<Modificador de acceso> <Tipo retorno> <Nombre>(<Parámetros>) {  
    // Código.  
}
```

Hemos mencionado mucho sobre modificadores de acceso y otros términos que quizás no están muy claros, por lo que los recopilamos (más adelante serán detallados):

- **Modificador de acceso:** permite el acceso de otros métodos de la clase o partes del código al método que estamos creando.
- **Tipo retorno:** determina si la función regresa un valor o cumple con una serie de instrucciones simplemente, pero no regresa valor.
- **Nombre método:** un nombre para la función. Tomen en cuenta de que un método no suele comenzar con un número, comienza con mayúsculas o tiene un nombre significativo.
- **Parámetros:** un método puede recibir valores por medio de parámetros para utilizarlos en su funcionamiento. Están constituidos por el tipo de dato, y su nombre, están separados por medio de comas (,).

Solo recordemos que los tips anteriores son recomendaciones para nombrar variables, funciones, métodos, procedimientos o atributos, no son reglas establecidas por el lenguaje.

Para poder llamar a un método desde otra sección de código escribimos su nombre, el par de paréntesis, y dentro de ellos, sus parámetros (si el método los tiene). A continuación, ejemplos de métodos:

```
import java.util.Scanner;

public static void Despliegue(int res){ // Crea un método que despliega el resultado de la
    suma.
    System.out.println(res);
}

public static void Suma(int a, int b){ // Crea un método que recibe dos parámetros enteros
    para sumarlos.
    int res; // Declaración de variable de resultado.
    res = a + b; // Asignación de la suma de parámetros a la variable.
    Despliegue(res); // Llamada al método de despliegue de resultado.
}

public class Main{
    public static void main(String[] args){
        int a, b;
        Scanner lectura = new Scanner(System.in); // Declaración de variables.

        a = lectura.nextInt();
        b = lectura.nextInt(); // Asignación de valores a las variables.

        Suma(a, b); // Llamada al método de suma de valores.
    }
}
```

### 12.1.5 Atributos de clases

Los **atributos** de una clase conforman las características que esta tiene. Los atributos están compuestos por:

```
<Modificador de acceso> <Tipo dato> <Nombre>;
```

Siguiente el ejemplo de Persona, crearemos algunos atributos para esta clase.

```
public class Persona{
    double Altura;
    double Peso;
    String Nombre;
    int Edad;

    void Caminar(){
        System.out.println("Una persona camina");
    }
}
```

Ahora, cuando declaramos un objeto Persona, este objeto tendrá los atributos de la clase disponibles para que se les asigne algún valor, como podemos ver en el siguiente ejemplo:

```
import java.util.Scanner;
```

```
public class Persona{ // Declaración de la clase.
    double Altura;
    double Peso;
    String Nombre;
    int Edad; // Declaración de distintos atributos.

    void Caminar(){ // Método que despliega que la persona camina.
        System.out.println("\n" + Nombre + " camina");

    void InfoPersona(){ // Método que despliega la información de la persona ingresada en
        el main.
        System.out.println("El nombre de la persona es: " + Nombre);
        System.out.println("La edad de la persona es: " + Edad);
        System.out.println("El peso de la persona es: " + Peso + " kg");
        System.out.println("La altura de la persona es: " + Altura + " cm\n");
    }
}

public class Main{
    public static void main(String[] args){
        String nom;
        double peso, altura;
        int edad; // Declaración de variables.
        Scanner lectura = new Scanner(System.in);
        Persona persona = new Persona(); // Declaración de objetos.

        nom = lectura.nextLine();
        peso = lectura.nextDouble();
        altura = lectura.nextDouble();
        edad = lectura.nextInt(); // Asignación de valores a las variables.

        persona.Nombre = nom;
        persona.Edad = edad;
        persona.Peso = peso;
        persona.Altura = altura; // Asignación de valores a los atributos del objeto
        persona.

        persona.InfoPersona(); // Llamada al método que despliega la información de la
        persona.
        persona.Caminar(); // Llamada al método que despliega que la persona camina.
    }
}
```

### 12.1.6 Modificadores de acceso

Los **modificadores de acceso** permiten que otras partes del código accedan a un atributo o método de una clase, a una función o procedimiento, para cuidar la integridad de su contenido. Los valores que pueden adoptar los modificadores de acceso son:

- Para declarar clases, existen:

- **public**: el que más hemos visto hasta ahora, permite que otras clases accedan a la que se está declarando.
- **default**: no es una palabra reservada, simplemente se declara la clase sin ningún modificador, permite que todas las clases en el mismo paquete accedan a la que se está declarando.
- Para declarar atributos y métodos de una clase, existen:
  - **default**: no es una palabra reservada, simplemente se declara el atributo o método sin algún modificador, permite que otra clase en el mismo paquete acceda o pueda trabajar con el atributo o método que se está declarando.
  - **public**: permite que otra clase acceda o pueda trabajar con el atributo o método que se está trabajando.
  - **protected**: permite que, si la clase tiene una clase derivada, la clase derivada pueda acceder a los atributos o métodos de su clase padre.
  - **private**: permite que el atributo o método solo sea accedido o trabajado dentro de la misma clase, no por fuera.

### 12.1.7 Tipo de retorno de métodos

Mencionamos que un método puede regresar o no un valor, esto lo determinamos creando el método, después de definir un modificador de acceso, definimos que nuestro método es del tipo `int`, `double`, `float`, `string`, etc. Con esto hecho, el método solicitará que se le asigne algún valor a retornar del mismo tipo que el método, la palabra reservada para lograr esto es **return**, justo como las funciones sencillas fuera de clases.

```
class MyClass{
    static int sum(int val1, int val2){
        return val1+val2;
    }
    public static void main(String[]args){
        intx=sum(2, 5);
        System.out.println(x);
    }
}
```

Por lo que, ahora sabemos o recordamos que:

- Las métodos y procedimientos tienen un modificador de acceso y un tipo de dato de retorno.
- Las funciones siempre regresan un tipo de dato.
- La palabra *return* es la que indica que la función regresa alguna instrucción que represente un tipo de dato que sea el mismo que la del método o función.
- La palabra *void* indica que un procedimiento o método no regresan ningún valor.

*Nota:* un método o función puede tener como parámetros dos doubles, pero regresar un `int`.

### 12.1.8 Tipos de valores y referencias

Algo que debemos tener a consideración es que, cuando hacemos una llamada a un método, función o procedimiento, y les pasamos una variable como parámetro, lo que realmente le pasamos a estas es una copia del valor de la variable, no el valor de la variable en sí, veamos el siguiente ejemplo:

```
public class MyClass{
    public static void main(String[] args){
        int x = 5; // Declara una variable con valor 5.
        addOneTo(x); // Llama al procedimiento de suma mandando el valor 5.
        System.out.println(x); // Imprime 5.
    }
    static void addOneTo(int num){ // Declara una función que recibe un parámetro para
        sumarle 1.
        num=num+1; // Al parámetro le suma 1.
    }
}
```

En realidad, no podemos pensar que la impresión debe ser 6, porque en ningún momento se le regresa 6 a main, el 5 que mandamos de main a addOneTo es solo un valor, no la variable x como tal.

Sin embargo, esto no ocurre con los objetos. Cuando una función o procedimiento tiene como parámetro un objeto de una clase X, el parámetro se convierte en un **valor referencia**. Un valor referencia es la dirección de memoria de un dato, lo podemos ver como un acceso directo de un programa en un sistema operativo, solamente que cualquier cosa que se le haga a la referencia, se verá reflejado en el dato original.

Entonces, si una función o procedimiento tiene como parámetro un tipo de dato, este se convierte en un *valor tipo*, si tiene como parámetro un objeto de clase, se convierte en una *referencia*. Podemos apreciarlo en el siguiente ejemplo (asumiendo que existe una clase llamada Person):

```
public class MyClass{
    public static void main(String[] args){
        Person j; // Declara un objeto Person.
        j=new Person("John"); // Se inicializa con un nombre "John".
        j.setAge(20); // Se le asigna 20 a su atributo Age.
        celebrateBirthday(j); // Llamada al procedimiento que agrega 1 a la edad del objeto
        J.
        System.out.println(j.getAge()); // Imprime la edad aumentada.
    }
    static void celebrateBirthday(Person p){ // Procedimiento que tiene como parámetro un
        objeto Persona, por lo que, cualquier cosa que se le haga al objeto que se le pase
        como parámetro, le pasará al objeto en sí.
        p.setAge(p.getAge()+1);
    }
}
```

*Nota:* los strings y arreglos también son tratados como referencias en los parámetros.

### 12.1.9 Constructores

Un **constructor** es un método que es llamado cuando se crea un objeto de una clase, los constructores están caracterizados porque no requieren modificador de acceso ni algún tipo de retorno (un tipo de dato o void), sin embargo, el constructor debe tener el mismo nombre de la clase donde se está declarando.

Los constructores suelen ser usados para inicializar un objeto de una clase con ciertos valores de atributos, además, una clase puede tener más de un constructor. Para declarar un objeto y verlo en funcionamiento, podemos ver el siguiente ejemplo:

```
import java.util.Scanner;

public class Persona{ // Declaración de clase.
    private String Nombre;
    private int Edad; // Declaración de atributos privados.

    Persona(String nom, int ed){ // Declaración de constructor con atributos para que,
        cuando se llame al constructor, asigne valores a los atributos, es decir,
        inicialice un objeto.
        this.Nombre = nom;
        this.Edad = ed;
    }

    public void Despliegue(){ // Declaración de un método que despliega el contenido de los
        atributos de la clase.
        System.out.println("Nombre: " + Nombre + ". Edad: " + Edad);
    }
}

public class Main{
    public static void main(String[] args){
        Scanner lectura = new Scanner(System.in);
        String nom;
        int ed; // Declaración de variables.

        nom = lectura.nextLine();
        ed = lectura.nextInt(); // Asignación de valores a las variables.

        Persona persona = new Persona (nom, ed); // Declaración de un objeto Persona y se
        le pasa como parámetro las variables nom y edad, para llamar al constructor.
        persona.Despliegue(); // Llamada al método que despliega la información de la
        persona.
    }
}
```

### 12.1.10 *Static* y tipos constantes *Final*

La palabra reservada **static** en una clase es utilizada para crear una variable o arreglo que pueda ser utilizado por todos los objetos que nazcan de dicha clase, a su vez que a las clases derivadas, es decir que, si creamos cinco objetos de la clase Persona, y la clase persona tiene una variable *static*, todos los objetos Persona pueden acceder a la variable estática y

su contenido; si creamos variables estáticas en la clase principal del proyecto (Class, App o algún otro nombre), estas variables pueden ser utilizadas por todas las clases, funciones, procedimientos y el main, en otras palabras, creamos una **variable global**. Este concepto de estático aplica a variables, procedimientos y funciones. Lo veremos en el siguiente ejemplo.

```
import java.util.Scanner;

public class Persona{ // Declaración de clase.
    public static Nombre = "Mario"; // Declaramos una variable estática pública.
}

public class Main{
    public static void main(String[] args){
        Scanner lectura = new Scanner(System.in);
        String nom; // Declaración de variables.

        nom = lectura.nextLine(); // Asignación de valores a las variables.

        Persona persona1 = new Persona();
        Persona persona2 = new Persona();
        Persona persona3 = new Persona(); // Declaración de objetos Persona.

        System.out.println(persona1.Nombre);
        System.out.println(persona2.Nombre);
        System.out.println(persona3.Nombre); // Las tres anteriores instrucciones
            despliegan el nombre Mario.
    }
}
```

Ahora bien, la palabra reservada **final** sirve para declarar variables constantes, es decir, solo pueden tener una asignación, ya compilado y en ejecución, dicho valor no puede cambiar. Funciones y procedimientos pueden ser asignados también como *final*, para evitar la sobrecarga de los mismos. Un ejemplo de declaración:

```
public static final double PI = 3.1416;
```

## 12.2 Conceptos de POO

### 12.2.1 Encapsulación

La idea de la **encapsulación** es que la implementación de un programa y sus detalles estén escondidos del usuario final, por ejemplo, no queremos que un usuario tenga acceso al valor de todo el dinero que tiene ahorrado en una cuenta bancaria, sino, este lo agrandaría para tener más dinero del que tiene. La *encapsulación* controla el acceso a los datos de un sistema o programa, permite que cambiemos algunos valores sin realizar grandes modificaciones a un programa. El ejemplo claro sobre la encapsulación visual es que declaremos atributos o métodos de una clase en **private**, así, ni otras clases fuera de la que se está declarando pueden acceder a dichos atributos. Un ejemplo de esto en código es:

```
class BankAccount { // Declara clase.
    private double balance=0; // Declara atributo privado.
```

```
public void deposit(double x) { // Declara método público para incrementar el valor del
    atributo.
    if(x > 0) {
        balance += x;
    }
}
```

En el ejemplo anterior, no podemos modificar directamente el valor del atributo privado, esto solo es posible por medio de un método público.

### 12.2.2 Herencia

La **herencia** es el proceso de pasar los atributos y métodos de una clase (llamada **clase padre, superclase o base**) a otra clase (llamada **clase derivada, subclase o hija**), estos atributos y métodos se pasan según el *modificador de acceso* que tenga, en Java, si algún atributo o método de una superclase es *private*, este atributo o método no se hereda a una subclase. en cambio, cualquier atributo *public* o *protected* si se heredan. Para heredar de una clase a otra, debemos utilizar la palabra reservada *extends*, como en el siguiente ejemplo:

```
public class Persona{ // Declara una clase Persona con un atributo y método.
    private String Nombre;

    public void setNombre(String nom){
        this.Nombre = nom;
    }
}

class Anciano extends Persona{ // Declara una clase Anciano que hereda de Persona su
    contenido.
    public void Info(){
        System.out.println("Mi nombre es:" + Nombre); // Podemos utilizar el atributo
        Nombre de Persona porque Anciano lo heredó de dicha clase.
    }
}
```

Debes tomar en cuenta que un **constructor** es único para cada clase, cuando se hereda de una clase a otra, el constructor de la superclase no se hereda, sin embargo, cuando se crea un objeto de la subclase, se llama primero al constructor de la superclase, después el constructor de la subclase.

Para **declarar** un objeto de una subclase, utilizamos la instrucción del siguiente ejemplo:

```
public static void main(String[] args){
    Persona viejo = new Anciano(); // Primero se escribe el nombre de la superclase, después
    s de new, el nombre de la subclase.
}
```

### 12.2.3 Polimorfismo

El **polimorfismo** es el pensamiento de que algo tiene "muchas formas", es decir, un ser vivo (a menos que sean seres vivos inertes) se mueve, se desplaza, camina, en nuestro caso,



retomamos la clase *Persona*, una persona camina, ahora, un niño, un adulto, un adolescente y un anciano caminan a distintas velocidades, **caminar** se convierte en algo distinto dependiendo del sujeto o ser que lo lleve a cabo, eso es el *polimorfismo*, que una acción pueda tener distinta implementación o comportamiento dependiendo el objeto u origen. A continuación, un ejemplo en código:

```
class Animal{ // Declara superclase.
    public void Sonido(){
        System.out.println("Grrrrr"); // Declara método a heredar.
    }
}

class Perro extends Animal{ // Declara subclase 1.
    public void Sonido(){
        System.out.println("Grrrrr");
    }
}

class Gato extends Animal{ // Declara subclase 2.
    public void Sonido(){
        System.out.println("Grrrrr");
    }
}

// Ambas subclases tienen el mismo método que la superclase, pero tienen un código
// distinto que las caracteriza.

public static void main(String[] args){
    Animal perro = new Perro(); // Crea un objeto Perro que hereda el contenido de Persona.
    Animal gato = new Gato(); // Crea un objeto Gato que hereda el contenido de Persona.

    perro.Sonido();
    gato.Sonido(); // Ambas llaman al mismo método, pero el resultado es distinto.
}
```

### Sobrecarga y sobre escritura

Otro ejemplo para comprender este término es el sonido de los animales, podemos crear una superclase *Sonido*, la cual tiene un método protegido *Sonido*, sabemos que los animales hacen ciertos ruidos para comunicarse, si declaramos una subclase de *Animal* llamada *Perro* y *Gato*, cada uno hace un ruido distinto, son animales los dos, pero su sonido se diferencia, eso es polimorfismo, pero también lo podemos relacionar con la **sobrecarga** o **sobre escritura** de métodos, porque si hacemos memoria, cuando hacemos alguna de las dos acciones con funciones o procedimientos estáticos, redefinimos el comportamiento del procedimiento o clase según nuestras necesidades, es similar en el polimorfismo.

Para sobre escribir una función o método, debemos seguir las siguientes reglas:

- El nombre del mismo debe ser igual al original y el modificador de acceso de la función/procedimiento sobre escrito debe ser **menos restrictivo** que el del original.
- Una función, procedimiento o método declarado como **final** y **static** no se pueden sobre escribir.

- Si un método no se puede heredar, no se puede sobre escribir.
- Constructores no se pueden sobre escribir.
- El método, función o procedimiento original y sobre escrito deben tener la misma cantidad de parámetros.

Cuando un procedimiento, función o método tiene el mismo nombre, pero distinta cantidad de parámetros, este método es **sobrecargado**, esto es útil cuando el mismo método debe tener distinto comportamiento en base a distintos parámetros. Un ejemplo:

```
import java.util.Scanner;

class Persona{ // Declara clase.
    public void Caminar(){ // Declara método original.
        System.out.println("Camina rápido");
    }
    public void Caminar(String men){ // Declara método sobre escrito.
        System.out.println(men);
    }
}

static void Suma(int a, int b){ // Declara función original.
    return a + b;
}

static void Suma(double a, double b){ // Declara función sobrecargada.
    return a + b;
}

public static void main(String[] args){
    Persona p = new Persona(); // Declara objeto Persona.
    Scanner lectura = new Scanner(System.in);

    String men = lectura.nextLine(); // Declaración y asingación de valor a la variable.

    p.Caminar(); // Llamada al procedimiento original.
    p.Caminar(men); // Llamada al procedimiento sobre escrito.

    System.out.println(Suma(1.1, 2.2)); // Llamada a la función estática original.
    System.out.println(Suma(1, 2)); // Llamada a la función estática sobrecargada.
}
```

#### 12.2.4 Abstracción

La **abstracción** es el pensamiento sobre el concepto de un objeto, más no sus características, por ejemplo, un libro: cuando mencionan dicha palabra, todos sabemos a qué nos referimos, pero no pensamos inmediatamente en cuántas páginas tiene un libro, ni su autor, ni tamaño, no imprenta, pensamos en su concepto, en la programación, esto es aplicable a clases abstractas, como lo puede ser en algún ejemplo anterior donde una superclase Animal tiene el método Caminar, y todas sus subclases heredan dicho método, pero con distinta implementación o bloque de código; la aplicación de la abstracción a esta clase sería hacer que este

método sea abstracto y, por ende, toda la clase `Animal` se vuelve abstracta, y así todas las subclases de esta superclase heredan el método abstracto y pueden darle la implementación que deseen. Esto es un complemento del *polimorfismo* y la *sobrecarga y sobre escritura* de métodos.

Para crear una **clase abstracta**, debemos utilizar la palabra reservada **abstract**, y algunas características sobre este tipo de declaraciones son:

- Si una clase es declarada, esta **no puede instanciarse**, es decir, no podemos declarar objetos de la misma.
- Para utilizar una clase abstracta, debemos primero heredarla de otra superclase.
- Cualquier clase que tenga métodos abstractos debe ser abstracta.
- Un método abstracto es declarado, pero no tiene un bloque de código, esto es porque será reacondicionado por otra subclase (*abstract void Caminar();*).

Un ejemplo de lo anterior:

```
abstract class Animal{ // Declara superclase abstracta, no se pueden crear objetos de la
    misma.
    public abstract void Sonido(){} // Declara método abstracto sin implementación.
}

class Perro extends Animal{ // Declara subclase 1.
    public void Sonido(){
        System.out.println("Grrrrr");
    }
}

class Gato extends Animal{ // Declara subclase 2.
    public void Sonido(){
        System.out.println("Grrrrr");
    }
}

// Ambas subclases tienen el mismo método que la superclase, pero al ser abstracto, se les
// puede dar una implementación distinta a cada uno.

public static void main(String[] args){
    Perro perro = new Perro(); // Crea un objeto Perro.
    Gato gato = new Gato(); // Crea un objeto Gato.

    perro.Sonido();
    gato.Sonido(); // Ambas llaman al mismo método, pero el resultado es distinto.
}
```

## 13 Métodos especiales

### 13.1 Getters y Setters

Como en CSharp, en el cual existe una herramienta que te permite asignar valores a los atributos de una clase mediante la instrucción **get** y **set**, aquí ocurre lo mismo, se crea un método que regresa el valor de un atributo privado donde el nombre del método comienza con *get* y continua con el nombre del atributo; de igual forma, se crea otro método donde el nombre del mismo comienza con *set* y continua con el nombre del atributo; ambos métodos pueden tener el modificador *public*. Declaramos un atributo llamado *Color* con modificador *private*, para cuidar su contenido y acceso, posteriormente, creamos los siguientes métodos:

```
public class Auto{
    private String color; // Atributo de prueba privado.

    public String getColor(){ // Al llamar a este método, obtenemos el valor contenido en
        el atributo color.
        return color;
    }

    public void setColor(String c){ // Al llamar este método, pasándole un valor string por
        el parámetro, asignamos el parámetro al atributo.
        this.color = c;
    }
}
```

La **función** del *getter* y *setter* es que asignemos y obtengamos el valor de atributos que tengan modificador de acceso privado. Una vez declarados, podemos utilizar los getters y setters en el main.

### 13.2 La clase Math

Para realizar distintas operaciones matemáticas en Java, tenemos la clase **Math** para facilitarnos el trabajo, no es necesario declarar un objeto de la clase, simplemente debemos ingresar la palabra reservada *Math* para utilizarla. Entre algunas de los métodos de esta clase tenemos:

- **Math.abs(-20)**: regresa el valor absoluto del parámetro. Regresa 20.
- **Math.ceil(4.75)**: redondea el valor decimal del parámetro a su entero más cercano. El valor de retorno es un **double**. Regresa 5.0
- **Math.floor(4.75)**: redondea el valor decimal del parámetro a su entero más cercano. El valor de retorno es un **entero**. Regresa 5
- **Math.max(3, 4)** y **Math.min(3, 4)**: regresa el mayor o mínimo de una serie de números. Regresa 4 y 3.

### 13.3 El método *equals()*

Cuando se trata de comparar dos objetos de una misma clase, no podemos utilizar el operador de comparación `==`, ya que los objetos hacen referencia a una dirección de memoria, entonces, al compararlos con este operador, verá que la dirección de memoria es distinta, por lo que el resultado siempre será *false*. Para comparar exitosamente dos objetos de una misma clase, se requiere utilizar el método (**`equals()`**), pero no se usa así nada más, este método de ser **sobre escrito**, y acompañarlo con el método **`hashCode()`**.

Para lograr esto, en tu editor de texto o IDE, puedes dar clic derecho sobre el nombre de la clase con la que se desee comparar objetos, en la opción *Source* o *Source Action*, seleccionamos la opción *Generate hashCode() and equals()*..., esta opción te genera automáticamente la sobre escritura de los métodos y su respectivo bloque de código para poder comparar objetos de la misma clase. Un ejemplo de equals:

```
public static class A{ // Declara una clase con atributo tipo int sencillo.
    public int x;

    // Se generó el código de hashCode() y equals() con el procedimiento descrito
    // anteriormente.
    // Ya se puede utilizar el método equals(), si no se genera este código, no se puede
    // utilizar correctamente.

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + x;
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        A other = (A) obj;
        if (x != other.x)
            return false;
        return true;
    }
}

public static void main(String[] args){
    A clase1 = new A(); // Declara tres objetos de la clase A, cada objeto recibe un valor
    // en su atributo.
    clase1.x = 5;
    A clase2 = new A();
    clase2.x = 5
    A clase3 = new A();
    clase3.x = 6;
```

```

    if (clase1.equals(clase2)){ // Despliega Iguales.
        System.out.println("Iguales");
    }
    else{
        System.out.println("No iguales");
    }
    if (clase2.equals(clase3)){ // Despliega No iguales.
        System.out.println("Iguales");
    }
    else{
        System.out.println("No iguales");
    }
}

```

### 13.4 Enums

Los **enums** son un tipo de dato especial, parecido a un arreglo, que se utiliza para declarar una colección de datos constantes (no se puede cambiar el valor), para acceder a los valores de un *enum*, utilizamos un **punto (.)**, en vez de índices. Para declarar un enum, utilizamos la siguiente estructura:

```
<Modificador de acceso> enum <Nombre> {<Val1>, <Val2>, <Val3>, ..., <ValN>;}
```

Y para declarar un objeto de un enum, seguimos el siguiente estructura:

```
<Nombre enum> <Nombre> = <Nombre enum>.<Val>;
```

Podemos seguir el siguiente ejemplo:

```

class Main {
    public static void main(String[] args) {
        Player player1 = new Player(Difficulty.EASY);
        Player player2 = new Player(Difficulty.MEDIUM);
        Player player3 = new Player(Difficulty.HARD); // Se declaran 3 objetos Player que
        se les pasa como parámetro un valor del enum Difficulty.
    }
}

enum Difficulty {EASY, MEDIUM, HARD} // Declara un enum y le asigna valores.

public class Player{ // Declara clase.
    Player(Difficulty diff){ // Declara constructor con parámetro tipo enum.
        switch(diff){ // Crea un switch para evaluar el valor del parámetro.
            case EASY: // No es necesario escribir "" para valores de enum (en caso de que
                sean una cadena o caracter).
                System.out.println("You have 3000 bullets");
                break;
            case MEDIUM:
                System.out.println("You have 2000 bullets");
                break;
            case HARD:
                System.out.println("You have 1000 bullets");

```

```
        break;
    }
}
}
```

## 14 Paquetes

Los **paquetes** podemos verlos como carpetas de clases, así tenemos más orden en los archivos de clases que creamos, además de dar un control adicional de acceso a dichas clases.

Para crear un *paquete*, en nuestro editor de texto o IDE, en la carpeta src, damos clic derecho y damos clic a la opción New Folder, asignamos un nombre y finalizamos el proceso. Cuando una clase es agregada al paquete, aparecerá la instrucción *package <Nombre clase>* para indicar a qué paquete pertenece la clase.

Si queremos utilizar una clase de un paquete, debemos utilizar la siguiente instrucción en la cabecera de nuestro archivo principal:

```
import <Nombre paquete>.<Nombre clase>;
```

Si queremos importar todas las clases de un paquete, podemos utilizar un **asterisco (\*)** después del nombre del paquete (sin el punto) para ahorrarnos líneas de código. Un ejemplo de paquetes (asumiendo que existe la clase Persona en un paquete Clases):

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

public class Main{
    public static void main(String[] args){
        Scanner lectura = new Scanner(System.in);
        String nom; // Declaración de variables.

        nom = lectura.nextLine(); // Asignación de valores a las variables.

        Persona persona1 = new Persona();
        Persona persona2 = new Persona();
        Persona persona3 = new Persona(); // Declaración de objetos Persona.

        System.out.println(persona1.Nombre);
        System.out.println(persona2.Nombre);
        System.out.println(persona3.Nombre); // Las tres anteriores instrucciones
        despliegan el nombre Mario.
    }
}
```

Utilizamos un ejemplo anterior, pero en vez de tener declarada la clase en el archivo principal, está contenida en un paquete y la importamos para trabajar con ella.

### 14.1 API de Java

Java contiene una serie de clases e interfaces generales que pueden ser utilizadas por todos en sus proyectos, es una cortesía del lenguaje que se encuentra en el enlace.

Una vez consultadas todas las clases e interfaces, seleccionamos alguna y la importamos a nuestro proyecto, recordemos que para importar utilizamos la estructura:

```
import java.<Nombre librería API>.<Nombre>;

import java.awt.*;
```

Recordemos que el asterisco (\*) en la instrucción import sirve para seleccionar todas las clases dentro de una librería.

## 15 Interfaces

Una **interface** es una clase completamente abstracta, se utiliza la palabra reservada **interface**, donde alguna de sus características son:

- Contiene nada más variables **static final**.
- No tiene un constructor debido a que este tipo de clases no pueden ser instanciadas.
- Se puede heredar el contenido de una interface a otra.
- Una clase regular puede implementar más de una interface, estas se separan utilizando una coma (,).
- Todos los métodos tienen implícitamente el modificador de acceso **public**.
- Cuando implementas una interface en una clase, tienes que sobre escribir todos los métodos de la interface en la clase.

Seguimos con el ejemplo de la clase Animal, pero implementando interfaces:

```
interface Animal{ // Declara superclase abstracta, no se pueden crear objetos de la misma.
    void Sonido(); // Declara método abstracto sin implementación.
    void Caminata();
}

class Perro implements Animal{ // Declara subclase 1.
    public void Sonido(){
        System.out.println("Guau");
    }
}

class Gato implements Animal{ // Declara subclase 2.
    public void Sonido(){
        System.out.println("Miaw");
    }
}

public static void main(String[] args){
    Perro perro = new Perro(); // Crea un objeto Perro.
    Gato gato = new Gato(); // Crea un objeto Gato.
```



```
perro.Sonido();
gato.Sonido(); // Ambas llaman al mismo método, pero el resultado es distinto.
}
```

## 16 Casting

El término **casting** hacia un valor se refiere a realizar una "conversión rápida" (que en realidad es una re-etiquetación) de un tipo de dato a otro, este tipo de conversiones se realizan con la siguiente instrucción:

```
int a = (int)3.14 // Se asigna el vlaor 3.
```

Como vemos, el valor asignado a la variable **int** es un decimal, un punto flotante, por lo cual no se le puede asignar dicho valor directamente, lo que hacemos es realizar un *casting* de dicho valor a **int**, lo que terminara conteniendo la variable es el número 3, o sea, la parte entera del valor decimal. Este tipo de casting se pueden realizar entre más tipos de datos, no solamente de **int** a **double**.

### 16.1 Upcasting y Downcasting

El *casting* también puede realizarse con clases, no como en tipos de datos regulares, pero similar; como hemos visto anteriormente, los objetos de clases se declaran de la siguiente forma:

```
Persona p = new Persona();
```

Los tipos de casting para las clases son:

- **Upcasting:** se refiere a declarar una subclase como su superclase, es decir, el objeto declarado será tratado como su superclase, sigue siendo un objeto de una subclase, pero el compilador lo ve como su superclase. Asumiendo que existe la superclase *Animal* y la subclase *Perro*, un ejemplo sería:

```
Animal animal = new Perro(); // Upcasting.
```

Esto es lo opuesto a declarar un objeto de una subclase en base a su superclase (*Perro p = new Animal();*). El proceso de *upcasting* es automático en Java, no se requiere algo más para esto.

- **Downcasting:** es el proceso contrario a un upcasting, como vimos en el concepto anterior, el upcasting busca que el compilador vea a un objeto de una subclase como uno de una superclase, un objeto *Gato* será visto como un objeto *Animal*, entonces, el *downcasting* se refiere a hacer que el compilador vuelva a ver al objeto de la subclase como tal (que el objeto *Gato* sea visto como objeto *Gato*, no *Animal*). Este proceso es manual, no automático, por lo que, se deben utilizar las siguientes instrucciones:

```
Animal animal = new Perro() // Upcasting.

((Perro)animal).Sonido(); // Re-etiqueta al objeto animal como perro.
```

## 17 Clases anónimas

Es un complemento de la sobre **escritura de métodos**, las **clases anónimas** es un término utilizado para modificar un método de una clase a la hora de la declaración de un objeto de la misma, como podemos ver en el siguiente ejemplo:

```
class Machine { // Declara una clase.
    public void start() { // Declara un método que despliega un mensaje.
        System.out.println("Starting...");
    }
}

public static void main(String[] args){
    Machinem = new Machine(){ // Declara un objeto Machine y abre llaves.
        @Override public void start(){ // Con la palabra reservada "@Override" y la
            sentencia del método a modificar, se realizan los cambios de código.
            System.out.println("Woooo0"); // Despliega nuevo mensaje.
        }
    };
    m.start(); // Llamada al procedimiento modificado, despliega "Woooo0" en vez de "
        Starting..."
}
```

Podemos ver que es una forma rápida y fácil de comprender para realizar sobre escritura de métodos de clases: las clases anónimas no es una clase como tal, es una herramienta para sobre escribir métodos de clases.

Esta modificación solo es aplicable para el objeto creado en ese momento, no para toda la clase en sí, así que, si creamos otro objeto Machine, el método start desplegará "Starting..." en vez del otro mensaje.

## 18 Clases anidadas

Java permite la **anidación** de clases, es decir, que una clase se convierta en un miembro de otra clase, simplemente escribe la declaración del objeto de la clase donde se requiera y listo (por ejemplo, en el constructor). Debe tomar en cuenta que, si la clase anidada es privada, ningún otro objeto o instrucción puede acceder a la clase anidada, pero la clase anidada puede acceder a todo el contenido (métodos y atributos) de su clase exterior. A continuación, dos ejemplos:

```
// Ejemplo 1.
public class Maquina{ // Declara clase.
    private String Tipo; // Declara atributo privado.
    Maquina(String tipo){ // Declara constructor que le asigna valor al atributo.
        this.Tipo = tipo;

        Robot r = new Robot(); // Declara objeto Robot dentro de la clase Maquina.
        r.Despliegue(); // Llamada al método de despliegue.
    }
}

private class Robot{ // Declara clase.
```

```
    public void Despliegue(){ // Declara método que despliega mensaje.
        System.out.println("Soy un robot");
    }
}

//Ejemplo 2
public class Maquina{ // Declara clase.
    private String Tipo; // Declara atributo privado.

    public class Robot{ // Declara clase dentro de clase.
        public void Despliegue(){ // Declara método que despliega mensaje.
            System.out.println("Soy un robot");
        }
    }
}
```

## 19 Excepciones

Una **excepción** es un error que ocurre durante la ejecución de un programa, puede que no haya ningún error de lógica o de sintaxis en el código, pero a la hora de estar ejecutándose, quizás se ingresó una cadena de texto en lugar de un número en el programa, o se sobrepasó la cantidad de elementos de un arreglo. Para poder **atrapar** errores durante la ejecución de un programa, se requiere la siguiente estructura:

```
try{
    // Bloque de código donde se busca atrapar errores.
} catch (Exception <Nombre de error>){
    // Instrucciones en caso de detección de error.
}
```

Dentro del bloque **try**, escribimos un bloque de código donde imaginamos que llegue a ocurrir un error (inserción de valores en arreglos, asignación de valores a variables, creación de objetos, apertura, lectura y escritura de archivos), y en el bloque **catch** escribimos lo que queremos que pase en caso de detección de un error, vemos que hay paréntesis donde está la palabra reservada **Exception**, esta palabra reservada corresponde a la superclase de excepciones que contiene Java, a partir de esta, existen subclases que se enfocan a atrapar distintos errores, como lo que se han mencionado recientemente, el `¡Nombre del error!` simplemente es asignar un nombre sencillo a la excepción para utilizarla en el bloque *catch* (es como crear una variable: `int variable == Exception e`). Presentamos un ejemplo donde se intenta acceder al índice de un elemento de un arreglo que no existe:

```
public class MyClass{
    public static void main(String[]args){
        try{ // Declara inicio de bloque try.
            int a[] = new int[2]; // Declara arreglo de dos posiciones.
            System.out.println(a[5]); // Imprime el contenido del elemento 5, se presenta un
                error.
        } catch(Exception e) { // Declara inicio de bloque catch, atraparé cualquier error
            que se presente durante la ejecución.
            System.out.println("An error occurred"); // Despliega que ha ocurrido un error.
        }
    }
}
```

### 19.1 Manejo de múltiples excepciones

La palabra reservada **throw** sirve para **lanzar** o llamar a una de las tantas excepciones que maneja Java, es una alternativa a crear el bloque *try-catch*, la estructura de *throw* es:

```
throw new <Excepción>(<Argumentos>)
```

Para poder utilizar esta palabra reservada, debemos hacer que el código que se va a analizar **esté sujeto** a la excepción que queremos utilizar, por ejemplo, si queremos realizar una división, sabemos que no se puede dividir entre 0, por lo que tenemos el siguiente código:

```
public double division(double a, double b) throws ArithmeticException{ // Declara función
    que está sujeta a atrapar excepciones tipo aritméticas.
```

```
if (b == 0){
    throw new ArithmeticException("División entre 0") // Atrapa y llama a la excepción
    con un mensaje a desplegar.
}else{
    return a / b; // Regresa la división exitosa.
}
}
```

## 19.2 Excepciones lanzadas (throw)

Así que, para utilizar throw, debemos primero hacer que el bloque de código a evaluar esté sujeto a *throws* y la excepción a buscar. Podemos hacer que un bloque de código esté sujeto a más de una excepción, estas excepciones se separan por medio de comas (,).

Ahora bien, también podemos hacer que haya más de un bloque catch en una estructura try, cada bloque catch puede atrapar distintas excepciones, es recomendable organizar las excepciones de los catch desde los más específicos hasta los más generales. Un ejemplo sobre múltiples bloques catch:

```
public double division(double a, double b){ // Declara una función.
    try{
        int num1 = 1;
        int num2 = 2; // Declara y asigna valores a las variables.
        System.out.println(num1 / num2); // Imprime la división de las variables.
    } catch(ArithmeticException e){ // Atrapa la excepción en caso de división entre cero.
        System.out.println("División entre 0");
    } catch(InputMismatchException e){ // Atrapa la excepción en caso de tipo de dato
        ingresado distinto a la variable.
        System.out.println("Error de formato")
    }
}
```

## 19.3 Excepciones Unchecked y Checked

Hay dos tipos de excepciones:

1. **Checked:** son aquellas que son atrapadas durante la compilación, y te generan un mensaje de error que evita que el programa sea ejecutado. Por ejemplo, la excepción `InterruptedException` (interrupción por el método `sleep` de los hilos).
2. **Unchecked (o Runtime):** son aquellas excepciones atrapadas durante el tiempo de ejecución de un programa. Por ejemplo, la excepción `ArithmeticException` (errores aritméticos).

## 20 Listas

La API de Java tiene clases especiales para guardar y manipular grupos de objetos.

## 20.1 ArrayList

El **ArrayList** es un tipo de arreglo especial, primero debemos usar la siguiente instrucción para utilizarlo:

```
import java.util.ArrayList;
```

Este tipo de arreglo se diferencia de los otros debido a que, una vez declarado con un tamaño inicial, cuando este tamaño se supera, el arreglo se incrementa, haciendo que incrementemente su tamaño solo. Este es uno de los escenarios de declaración de *ArrayList*, pero también se pueden declarar sin un tamaño, por lo que es considerado como un arreglo de tamaño indeterminado, además, tampoco es necesario declarar el tipo del ArrayList, como se ve a continuación:

```
ArrayList colores = new ArrayList(); // ArrayList sin tamaño fijo ni tipo específico.  
ArrayList<String> colores = new ArrayList<String>(10); // ArrayList con tipo y tamaño  
declarado.
```

Lo que almacena un ArrayList son objetos, por lo tanto, no podemos declarar un arreglo de estos que almacene valores int, double o char, sin embargo, podemos hacer que almacene clases tipo que representen estos valores, es decir, que el arreglo almacene objetos Integer, Double, entre otros.

Otro factor que diferencia a estos arreglos de los normales, son la forma de trabajarlos, para agregar un elemento a un ArrayList se requiere del método **add()** y para eliminar, se utiliza el método **remove()**, como se ve en el siguiente código:

```
import java.util.ArrayList; // Importa la librería necesaria para utilizar ArrayLists.  
  
public class MyClass{  
    public static void main(String[] args){  
        ArrayList<String> colores = new ArrayList<String>(); // Declara el ArrayList de  
        clase tipo String.  
        colors.add("Red");  
        colors.add("Blue");  
        colors.add("Green");  
        colors.add("Orange"); // Agrega 4 colores con el método add().  
        colors.remove("Green"); // Elimina un color con el método remove().  
        System.out.println(colors); // Imprime los colores.  
    }  
}
```

Otros métodos útiles son:

- **contains()**: regresa **true** si el arreglo contiene un objeto especificado.
- **get(índice int)**: regresa el contenido del elemento en la posición especificada.
- **size()**: regresa el número de elementos del arreglo.
- **clear()**: elimina todos los elementos del arreglo.

## 20.2 LinkedLists

Las **LinkedLists** tienen una sintaxis muy similar a las *ArrayLists*, puedes transformar el código de este último tipo de arreglo a un **LinkedList** cambiando las siguientes instrucciones:

```
import java.util.LinkedList; // Librería para trabajar con LinkedLists.
LinkedList ll = new LinkedList(); // Declaración de una LinkedList.
```

La diferencia entre un **ArrayList** y un **LinkedList** es que, el primero trabaja de forma similar a un arreglo, los **ArrayLists** almacenan objetos de la forma en la que un arreglo lo hace, en cambio la **LinkedList**, almacena objetos y crea un enlace o *link* hacia el próximo elemento, es decir, forma una cadena o secuencia con los elementos, además, también almacena la dirección de memoria de los objetos guardados. Las **LinkedLists** son utilizadas para guardar, insertar y eliminar gran cantidad de elementos, y los **ArrayList** son utilizados para guardar menor cantidad de elementos, pero con la posibilidad de acceder a ellos de forma más rápida y sencilla. Un ejemplo de **LinkedLists**

```
import java.util.LinkedList; // Declara librería para trabajar con LinkedLists.
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        LinkedList<String> words = new LinkedList<String>(); // Declara LinkedList tipo
        String.

        while(words.size()<5){
            String word = scanner.nextLine(); // Asigna valor a la variable.
            words.add(word); // Agrega la variable al LinkedList.
        }

        for(String i: words){ // Recorre el LinkedList.
            if(i.length() > 4){
                System.out.println(i); // Si el largo del elemento del LinkedList supera los
                cuatro caracteres, imprime el elemento.
            }
        }
    }
}
```

## 20.3 HashMap

Un **HashMap** es un tipo de almacenamiento de datos donde se utiliza un **par de elementos**, los cuales son una **llave** (como un índice) y un **valor** (el contenido) para guardar y acceder a la información almacenada. De igual forma, se requiere incluir una librería para utilizarlas y la forma para declararlos es similar a las dos listas vistas anteriormente:

```
import java.util.HashMap; // Librería para trabajar con HashMaps.
HashMap<Tipo elemento, Tipo índice> $Nombre$ = new HashMap<Tipo elemento, Tipo índice>(); // Declaración de un HashMap.
```

Los métodos para agregar, eliminar y obtener el valor de un elemento en un `HashMap`, respectivamente, son `put()`, `remove()` y `get(Llave)`. Un ejemplo de un `HashMap` es:

```
import java.util.HashMap; // Declara librería para trabajar con HashMaps.

class A {
    public static void main(String[] args) {
        HashMap<String, String> m = new HashMap<String, String>(); // Declara HashMap con
        llave tipo String y valores tipo String.
        m.put("A", "First");
        m.put("B", "Second");
        m.put("C", "Third"); // Agrega elementos.
        System.out.println(m.get("B")); // Despliega el valor de la llave "B".
    }
}
```

Un *HashMap* no puede tener llaves duplicadas, si escribes un valor y llave sobre una llave ya existe, el valor se sobre escribe. Existen los métodos `containsKey()` y `containsValue()` para saber si en un `HashMap` existe alguna llave o valor, en caso de que alguno de los dos no exista, se regresa un valor `null`.

## 20.4 Conjuntos

Un **conjunto** es una colección de elementos que no puede tener elementos duplicados. Un *HashSet* es la herramienta utilizada para trabajar con conjuntos, como vemos en el siguiente ejemplo:

```
import java.util.HashSet; // Declara librería para trabajar con HashMaps.

class A {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<String>(); //Declara HashMap solamente con llave,
        sin valor
        set.add("A");
        set.add("B");
        set.add("C"); // Agrega elementos que conforman un conjunto.
        System.out.println(set.size()); // Imprime el tamaño del HashMap.
    }
}
```

### 20.4.1 LinkedHashMaps

Los `HashMaps` no llevan un registro u orden sobre como se van agregando los elementos, puedes agregar 100 elementos y no te dice cuál fue primero o cuál es el último. Para resolver eso, se emplea un **LinkedHashMap** para que haya enlaces entre los elementos, logrando así saber el orden en el que se fueron registrando.

## 20.5 Organizando listas

Una forma sencilla de **organizar** listas, es por medio de la clase **Collections**, dentro de la librería `java.util`, es estática, por lo que no debes importarla, ni crear un objeto de la misma.



El método `sort()` de esta clase es la que utilizaremos para lograr nuestro cometido:

```
import java.util.ArrayList; // Importa librería para trabajar con ArrayLists.
import java.util.Collections; // Importa librería para trabajar con el método sort().

public class App{
    public static void main(String[] args){
        ArrayList<Integer> numeros = new ArrayList<Integer>(); // Declara ArrayList de
            tipo Integer.
        ArrayList<String> animales = new ArrayList<String>();
        nums.add(3);
        nums.add(100);
        nums.add(56);
        nums.add(4342);
        nums.add(90);
        nums.add(353);
        nums.add(66);
        nums.add(34);
        nums.add(4);
        nums.add(1); // Agrega muchos elementos.
        animales.add("perro");
        animales.add("serpiente");
        animales.add("leon");
        animales.add("gato"); // Agrega muchos elementos.

        Collections.sort(nums); // Organiza los elementos de menor a mayor.
        Collections.sort(animales); // Organiza elementos en orden alfabético.
        System.out.println(nums);
        System.out.println(animales);
    }
}
```

Otros métodos útiles de la clase `Collections` son:

- `max()`; regresa el elemento máximo de una colección de datos.
- `min()`; regresa el elemento mínimo de una colección de datos.
- `reverse()`; regresa la secuencia invertida de una lista.
- `suffle()`; baraja los elementos de una lista en base a un parámetro (p. e. aleatorio).

## 20.6 Iterators

Los **Iterators** son un objeto que nos permiten navegar a través de una colección de datos, para obtener y eliminar elementos de los mismos. Cada colección de datos posee un método `iterator()`, a partir de este, podemos crear objetos iterators, si no creamos el objeto, no podemos utilizar un iterator; otra cosa a considerar es que debemos importar la librería `java.util.Iterator`; para poder trabajarlos.

Algunos de los métodos disponibles de los iterators son:

- `hasNext()`: regresa *true* en caso de que haya un elemento a continuación del actual, en caso de que no, regresa *false*.

- **next()**: regresa el elemento actual y avanza al siguiente.
- **remove()**: remueve el último elemento regresado en next().

Se pueden combinar iterators y ciclos, como vemos en el siguiente código:

```
import java.util.Iterator; // Librería para trabajar con Iterators.
import java.util.LinkedList; // Librería para trabajar con LinkedLists.

public class MyClass {
    public static void main(String[] args) {
        LinkedList<String> animals = new LinkedList<String>(); // Declara LinkedList del
            tipo String.
        animals.add("fox");
        animals.add("cat");
        animals.add("dog");
        animals.add("rabbit"); //Agrega elementos

        Iterator<String> it = animals.iterator(); // Declara objeto Iterator de la colecció
            n animals.
        while(it.hasNext()) { // Mientras detecte que existe un elemento siguiente al
            actual.
            String value = it.next(); // A value se le asigna el elemento actual del
                iterator y se mueve al siguiente.
            System.out.println(value); // Imprime la variable.
        }
    }
}
```

## 21 Hilos

Java maneja **múltiples hilos** para trabajar, podemos correr algún bloque de código en un hilo mientras se ejecutan otros bloques de código en más hilos. Existen dos formas de crear un hilo:

1. **Declarando una subclase de Thread**: creamos una subclase de la superclase **Thread** para obtener todos sus métodos, pero sobre todo, sobre escribir el método **run()** para ahí escribir el código que buscamos ejecutar, como vemos en el siguiente ejemplo:

```
class Prueba extends Thread{ // Subclase de Thread.
    public void run(){ // Sobre escribe el método run con otro comportamiento.
        System.out.println("Hola mundo");
    }
}

class App{
    public static void main(String[] args){
        Prueba p = new Prueba(); // Declara un objeto Prueba.
        p.start(); // Llamada al procedimiento que ejecuta el método run().
    }
}
```

2. **Declarando una clase con interface Runnable:** declaramos una clase que implemente la interface **Runnable** y sobre escribimos el método *run()* para que haga lo que nosotros busquemos. En el método main, declaramos un objeto Thread, y como argumento en **new**, le pasamos un objeto de nuestra clase creada previamente, una vez declarado el objeto *Thread*, llamamos a su método *start()*, como se ve en el siguiente ejemplo:

```
class Prueba implements Runnable{ // Clase que implementa la interface Runnable
    para hilos.
    public void run(){ // Sobre escribe el método run con otro comportamiento.
        System.out.println("Hola mundo");
    }
}

class App{
    public static void main(String[] args){
        Thread p = new Thread(new Prueba()); // Declara un objeto Thread con un
        objeto Prueba.
        p.start(); // Llamada al procedimiento que ejecuta el método run().
    }
}
```

Cosas a considerar con los hilos:

- Todos los hilos creados por uno tienen un nivel de **prioridad**, este nivel de prioridad es 5, pero puede ir de 1 a 10. Para cambiar el nivel de prioridad de un hilo, se utiliza el método *Thread.setPriority()* (p. e. *Thread.setPriority(3)*: prioridad tres).
- Se puede **pausar** un hilo utilizando el método *Thread.sleep()*, dentro de los paréntesis, se escribe la cantidad de milisegundos que el hilo se detendrá (p. e. *Thread.sleep(2000)*: dos segundos). Pausar un hilo puede causar un error durante la ejecución, por lo que es recomendable trabajar con dicho método dentro de un bloque try-catch.

## 22 Trabajando con archivos

El paquete **java.io** contiene una clase llamada **File** que nos permite trabajar con archivos, una vez importada, declaramos un objeto de dicha clase y, en los paréntesis de *File* al final de la declaración, escribimos la ruta del archivo que buscamos trabajar, como se ve a continuación:

```
import java.io.File; // Paquete necesario para trabajar con archivos.
File <Nombre> = new File (<Ruta del archivo>); // Declaración objeto File.
```

El siguiente código nos ayuda a verificar la existencia del archivo que trabajaremos:

```
import java.io.File; // Paquete necesario para trabajar con archivos.

public class MyClass {
    public static void main(String[] args) {
        File archivo = new File("D:\\Escritorio\\test.txt"); // Declara objeto File con la
        ruta del archivo
        if(archivo.exists()) { //Si la ruta del objeto File creado existe.
```

```
        System.out.println(archivo.getName() + "exists!"); // Despliega mensaje.
    }
    else {
        System.out.println("The file does not exist");
    }
}
}
```

*Nota:* se utilizan doble barra invertida (\\) para escribir las rutas de un archivo, porque la diagonal invertida suele ser usada como **secuencia de escape**, además, es recomendable trabajar archivos alrededor de un bloque try-catch.

## 22.1 Lectura

La clase **Scanner** del paquete **java.util** nos va a ser útil para leer el contenido de un archivo en Java, el constructor de esa clase permite que le pasemos un objeto File en su constructor y declaración, como se ve a continuación:

```
try{
    File archivo = new File("D:\\Escritorio\\test.txt"); // Declara objeto File con la ruta
                                                         del archivo.
    Scanner arlectura = new Scanner(archivo); // Declara objeto Scanner para abrir un
                                                         archivo.
} catch(FileNotFoundException e){}
```

La clase *Scanner* es heredada de *Iterator*, por lo que esta primera se comporta como la última, podemos utilizar los métodos mencionados en el punto de Iterators para esta ocasión, el método *next()*, para Scanner, regresa el contenido del archivo palabra a palabra, por lo que el método regresará una línea del archivo hasta que encuentre un salto, por lo tanto, obtendremos línea a línea hasta que se acabe el fichero. Expandiremos el ejemplo anterior:

```
try{
    File archivo = new File("D:\\Escritorio\\test.txt"); // Declara objeto File con la ruta
                                                         del archivo.
    Scanner arlectura = new Scanner(archivo); // Declara objeto Scanner para abrir un
                                                         archivo.

    while(arlectura.hasNext()){ // Mientras encuentre más palabras.
        System.out.println(arlectura.next()); // Imprime una línea del archivo.
    }
    arlectura.close(); // Cierra el objeto Scanner.
} catch(FileNotFoundException e){ // Excepción por si no encuentra el archivo.

    System.out.println("No encontrado");
}
```

## 22.2 Creación y escritura

Utilizamos la clase **Formatter** del paquete **java.util** para crear y escribir archivos en una ruta especificada, en caso de que el archivo ya exista, el archivo se sobre escribirá. Un ejemplo:

```
import java.util.Formatter; // Paquete necesario para trabajar con Formatter.

public class MyClass {
    public static void main(String[] args) {
        try {
            Formatter f = new Formatter("D:\\Escritorio\\test.txt"); // Declara objeto
                           Formatter con la ruta donde se creará el archivo.
        } catch (Exception e) {
            System.out.println("Error");
        }
    }
}
```

Una vez declarado el objeto, podemos escribir en él con el método **format()**, el cual tiene una forma peculiar de trabajar: en él, escribimos un parámetros entre comillas con, por ejemplo, tres pares de caracteres **%s**, estos tres pares de caracteres en un solo parámetro representan tres palabras, separadas por espacios, después de este parámetro, le pasamos tres parámetros más, los cuales son las palabras en sí que va a contener el archivo, es decir, por cada par **%s**, va otro parámetro que es la palabra que queremos escribir. El carácter **%s** significa que lo que escribiremos es una cadena de texto. Podemos ver esto más claro con el siguiente ejemplo:

```
import java.util.Formatter; // Paquete necesario para trabajar con Formatter.

public class MyClass {
    public static void main(String[] args) {
        try {
            Formatter f = new Formatter("D:\\Escritorio\\test.txt"); // Declara objeto
                           Formatter con la ruta donde se creará el archivo.
            f.format("%s %s", "esta es la primer cadena a escribir", "esta es la segunda
                           cadena, ambas están separadas por un espacio\n"); // Escribe en el archivo.
            f.format("%s %s", "soy el segundo parámetro de format", "soy el tercer parámetro
                           de format\n"); // Escribe en el archivo.
            f.close(); // Cierra el objeto.
        } catch (Exception e) {
            System.out.println("Error");
        }
    }
}
```