

INTRODUCCIÓN A LA PROGRAMACIÓN

Semana 2

ING. MARTIN POLIOTTO

Docente a cargo del modulo

Julio 2020



DIPLOMATURA EN

**NUEVAS
TECNOLOGÍAS**

SECRETARÍA DE
EXTENSIÓN
UNIVERSITARIA

SEU

UTN
Facultad Regional Córdoba

Ministerio de
PROMOCIÓN DEL EMPLEO
Y DE LA ECONOMÍA FAMILIAR

Ministerio de
CIENCIA Y
TECNOLOGÍA

CBA
ENTRE TODOS

Oficina de
Montevideo
Organización
de las Naciones Unidas
para la Educación,
la Ciencia y la Cultura

Semana 02

1. Sintaxis del lenguaje

1.1 Elementos básicos: variables

Cuando se desea ejecutar un programa, el mismo debe cargarse en la memoria del computador. Un programa, entonces, ocupará normalmente varios cientos o miles de bytes. Sin embargo, con sólo cargar el programa no basta: deben cargarse también en la memoria los datos que el programa necesitará procesar. A partir de aquí, el programa puede ser ejecutado e irá generando ciertos resultados que de igual modo serán almacenados en la memoria. Estos valores (los datos y los resultados) ocuparán en memoria un cierto número de bytes, que depende del tipo de valor del que se. En todo lenguaje de programación, el acceso a esos datos y resultados ubicados en memoria se logra a través de ciertos elementos llamados **variables**.

Una variable es un grupo de bytes asociado a un nombre o identificador, pero de tal forma que a través de dicho nombre se puede usar o modificar el contenido de los bytes asociados a esa variable.

1.1.1 Reglas para nombres

Si bien, los nombres de las variables son definidos por el programador, existen ciertas restricciones a considerar al momento de nombrarlas:

- Los nombres de variables deben empezar por letra (mayúscula o minúscula), un carácter de subrayado (_) o un signo pesos (\$).
- No pueden empezar con un número
- No pueden contener puntuación, espacios o guiones (medios)
- No pueden coincidir con palabras reservadas (como class o static) del lenguaje
- Por convención se utiliza notación de camello. En general la primer letra de cada palabra se indica con mayúsculas excepto la primera. Por ejemplo:

boolean estaVacio = true;

- Algunos ejemplos no válidos:

3xy // Comienza con un número.

dir ant //Hay un blanco en el medio (no es un caracter válido).

nombre-2 // Usa el guión alto y no el bajo.

int //Es palabra reservada.

1.1.2 Asignaciones

Para cambiar el valor de una variable se usa la **instrucción de asignación**, que consiste en escribir el nombre de la variable, seguido del signo igual y luego el valor que se quiere asignar. El

signo igual (=) se designa como operador de asignación. Notar que en el lenguaje Java toda instrucción finaliza con punto y coma (;). Ejemplo:

x1 = 7;

Cabe notar que también se puede asignar el resultado de una expresión. Por ejemplo:

- `Int suma = sum1 + sum2;`
- `boolean esMayor = 10 > 90;`
- `String cad = "Hola" + "Mundo";`

1.1.3 Tipos de datos en Java

En el lenguaje Java existen dos grupos principales: las variables simples (o tipos primitivos) y las referenciales (de objeto). Los primeros son capaces de almacenar números, caracteres o valores lógicos, mientras que los segundos permiten almacenar la dirección de memoria de objetos creados en los programas.

Existe una relación entre el tipo de datos, la memoria ocupada y la cantidad de información que es posible almacenar dentro una variable. La siguiente tabla muestra esta relación para los 8 tipos primitivos manejados en Java:

Nombre	Tipo	Byte	Rango de valores
boolean	Lógico	1	true ó false
char	Carácter simple	2	Un caracter
byte	Entero	1	-128 a 127
short	Entero	2	-32768 a 32767
int	Entero	4	-2.147.483.648 a 2.147.483.649
long	Entero	8	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.808
float	Numérico con coma flotante	4	34×10^{-38} a 34×10^{38}
double	Numérico con coma flotante	8	1.8×10^{-308} a 1.8×10^{308}

Si por ejemplo definimos un variable de tipo **byte**:

byte unByte;

Java utiliza exactamente 1 byte de memoria, lo que equivale a 8 bits (valores 0 o 1) consecutivos. Entonces:

+	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

Donde el primer bit se utiliza para el signo (positivo o negativo) y el resto para representar un valor numérico. Como cada uno de los 7 bits restantes pueden asumir valores 0 o 1, la cantidad de combinaciones viene dada por **2^7** con lo cual tenemos:

- **2^7** = 128 valores negativos
- **0**
- **$2^7 - 1$** = 127 valores positivos (sin contar el cero)

1.1.4 Consideraciones adicionales

- Un tipo especial de variable es aquella que puede contener un valor inicial que no cambiara durante la

ejecución de un programa. En este caso hablamos de **constates**. En Java se utiliza el modificar **final** como se muestra a continuación:

```
final int CURRENT_YEAR = 2020;
```

Notar que por convención el identificador de una constante se indica con letras mayúsculas. Si tiene más de una palabra, se usa como separador al signo (_)

- Siempre es posible asignar una variable de un tipo de menor rango a una de mayor rango.
 - Un **char**, **byte** o **short** dentro de un **int**
 - Un **int** dentro de un **long**
 - Un **long** dentro de un **float**
 - Un **float** dentro de un **double**
- Si la relación anterior es inversa, entonces se obtiene un error de compilación
- Las constantes de precisión flotante son manejadas como **double**. Por ejemplo:

```
float num = 27.9; // error de compilación
```

Una posible solución es indicar con una letra F para que el compilador asuma que es **float**:

```
float num = 27.9F;
```

- Algo especialmente curioso es la forma en la que el lenguaje almacena y gestiona los caracteres. Si un carácter se almacena en una variable entera, este almacena directamente su valor ASCII. Por ejemplo:

```
char car = 'a';
```

```
int ascii = car;
```

En **ascii** se almacena el valor 97.

También es posible incrementar **car**:

```
car++;
```

quedado con un valor 'b' . Finalmente es posible, de manera inversa, obtener el carácter a partir de su valor `ascii`:

```
char car = 98;
```

1.2 Operadores aritmético

Una expresión es una fórmula en la cual se usan operadores (como suma o resta) sobre diversas variables y constantes (que reciben el nombre de operandos de la expresión). Si el resultado de la expresión es un número, entonces la expresión se dice expresión aritmética.

Por ejemplo:

```
int suma, num1 = 10, num2 = 7;
suma = num1 + num2;
```

Note que en una asignación primero se evalúa cualquier expresión que se encuentre a la derecha del signo =, y luego se asigna el resultado obtenido en la variable que esté a la izquierda del signo =.

La siguiente tabla muestra los principales operadores aritméticos del lenguaje Java:

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
-	operador unario de cambio de signo	-4	-4
+	Suma	2.5 + 7.1	9.6
-	Resta	235.6 - 103.5	132.1
*	Producto	1.2 * 1.1	1.32
/	División (tanto entera como real)	0.050 / 0.2 7 / 2	0.25 3
%	Resto de la división entera	20 % 7	6

Algunas consideraciones a tener en cuenta con los operadores aritméticos:

- Las operaciones entre integrales: **byte, short o int** se resuelven como **int**. Por ejemplo:

short a, b, c;

a = 5;

b = 9;

c = a + b; // error de compilación.

Una posible solución sería:

Int c = a + b;

o bien;

c = (short) (a + b);

En este segundo caso, se realiza un casting o conversión explícita del resultado del operador +.

- La división entre enteros se resuelve como entero. Por ejemplo:

int *div, a, b;*

a = 6;

b = 10,

div = a/b;

El resultado de la división en este caso es 0.

Siempre que uno de los dividendos sea un valor de punto flotante, la división se resolverá como flotante.

- Existen también operadores de incremento o decremento que permiten aumentar o disminuir en

una unidad el valor de una variable numérica, como se muestra a continuación

```
int c = 0;
```

```
c++;
```

```
--C;
```

- Un último aspecto a considerar son **las prioridades de los operadores**. La regla es la que se indica en el siguiente recuadro:

Operators	Precedence
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	? :
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

Mayor



Menor

1.3 Estructura mínima de un programa Java

En el caso más simple un programa Java cuenta de una única clase. Esta clase contiene el proceso, rutina o método principal: **main()** y en éste se incluyen las sentencias del programa principal. Estas sentencias se separan entre sí por caracteres de punto y coma.

La estructura de un programa simple en Java es la siguiente:

```
public class ClasePrincipal {  
    public static void main(String[] args) {  
        sentencia_1;  
        sentencia_2;  
        // ...  
        sentencia_N;  
    }  
}
```

- En un programa Java todo se organiza dentro de las **clases**. Todos los programas o aplicaciones independientes escritas en Java tienen un **método main** o principal que, a su vez, contiene un conjunto de sentencias. En Java los conjuntos o bloques de sentencias se indican entre llaves **{ }**. Dentro de una clase también encontraremos atributos y métodos. Éstos serán abordados en detalle cuando se comience con POO.
- Cuando escribimos código en general es útil realizar comentarios explicativos. Los comentarios no tienen efecto como instrucciones para el lenguaje, simplemente sirven para que cuando un programador lea el código pueda comprender mejor lo que lee. En Java se pueden usar dos tipos de comentarios:

- **Comentario en una línea o al final de una línea:** se introduce con el símbolo //
- **Comentario multilínea:** se abre con el símbolo /* y se cierra con el símbolo */

1.4 Entrada y Salida de datos

Para comenzar a escribir los primeros programas Java es necesario revisar de qué forma el lenguaje toma datos desde la entrada estándar del equipo y a su vez permite mostrar resultados de sus procesos.

1.4.1 Visualización de resultados por Consola

Al ejecutar un programa, lo normal es que antes de finalizar el mismo muestre por pantalla los resultados obtenidos. En el lenguaje Java la instrucción más básica para hacer eso es **System.out.print()**. Esta instrucción permite mostrar en pantalla tanto el contenido de una variable como también mensajes formados por cadenas de caracteres.

- Algunos ejemplos:

Caso	Descripción
<code>System.out.print("Hola Java");</code>	Muestra la cadena de texto
<code>int a = 8;</code> <code>System.out.print(a);</code>	Muestra el contenido de una variable
<code>System.out.println("Hola Java" + a)</code>	Muestra la concatenación En este último caso el método println() permite mostrar el texto y luego agrega automáticamente un

	salto de línea .
--	------------------

- Java también proporciona secuencias de escape, es decir, una combinación de la barra invertida \ y un carácter que nos permite insertar caracteres especiales dentro de un texto. A continuación se muestran las secuencias más usadas:

Secuencia de Escape	Descripción
\n	Salto de línea
\t	Tabulador
\\	Diagonal Inversa \
\"	Comillas Dobles
\'	Comilla Simple
\r	Retorno de Carro (Solo en modo Administrador)
\b	Borrado a la Izquierda (Solo en modo Administrador)

1.4.2 Entrada de datos por Teclado

La entrada o lectura de datos en Java es uno de los conceptos más importantes y fundamentales al momento de interactuar con el usuario de nuestro programa. La entrada de datos en Java, a diferencia de otros lenguajes es un poco complicada (no demasiado) y existen diferentes formas de hacerlo.

1.4.2.1 Utilizando clases de **java.io**

La entrada de datos con estas clases se puede hacer en al menos tres líneas según lo que necesitemos hacer. El código siguiente muestra esta variante:

```
BufferedReader lector = new BufferedReader(new  
InputStreamReader(System.in));  
  
String cadena = lector.readLine();
```

De esta forma creamos un objeto lector que nos permite leer toda una línea de texto hasta que se ingrese un <Enter>.

Si por ejemplo quisiéramos leer un número necesitamos incluir:

```
BufferedReader lector = new BufferedReader(new  
InputStreamReader(System.in));  
  
int num = Integer.parseInt(lector.readLine()); // convertimos la //cadena en un  
número entero.
```

- Es importante notar que este tipo de lectura nos obliga a declarar a nivel de método `main()` una excepción de tipo `IOException`:

```
public static void main(String[] args) throws IOException
```

Esto se verá con mayor profundidad en las últimas clases.

- El mecanismo es sencillo, leer cadenas mediante el método **`readLine()`** y luego convertir al tipo de variable que necesitemos trabajar.

1.4.2.2 Utilizando un **Scanner** de **java.util**

La clase *Scanner* de la librería *java.util* es también muy sencilla para obtener datos de entrada del usuario, a diferencia de *BufferedReader*, *Scanner* si posee un método para la lectura de números y para la lectura de texto que nos ayudarán a facilitar un poco las cosas, veamos:

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

```
String cadena = lector.nextLine();
```

```
int num = lector.nextInt();
```

- A diferencia el primer mecanismo estas instrucciones no obligan a declarar ninguna excepción y nos permiten leer tanto cadena como variables de tipos primitivos (int, float, boolean, etc). Por este motivo será el mecanismo elegido para la resolución de los ejercicios propuestos.

2. Estructuras de Control

2.1 Estructuras Secuenciales

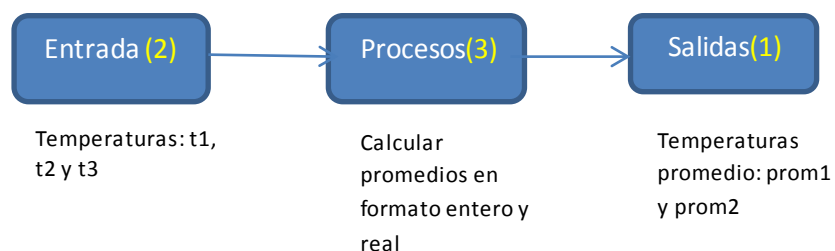
Cuando comenzamos un curso de programación lo primero que abordamos son problemas simples de lógica lineal que pueden ser resueltos mediante la aplicación de secuencias de instrucciones simples (de asignación, de visualización o de carga por teclado) una debajo de la otra, de tal manera que **cada instrucción se ejecuta una después de la otra**. Un bloque de instrucciones lineal de esta forma, se conoce en programación como un bloque secuencial de instrucciones o también como una estructura secuencial de instrucciones.

2.2.1 Caso de estudio

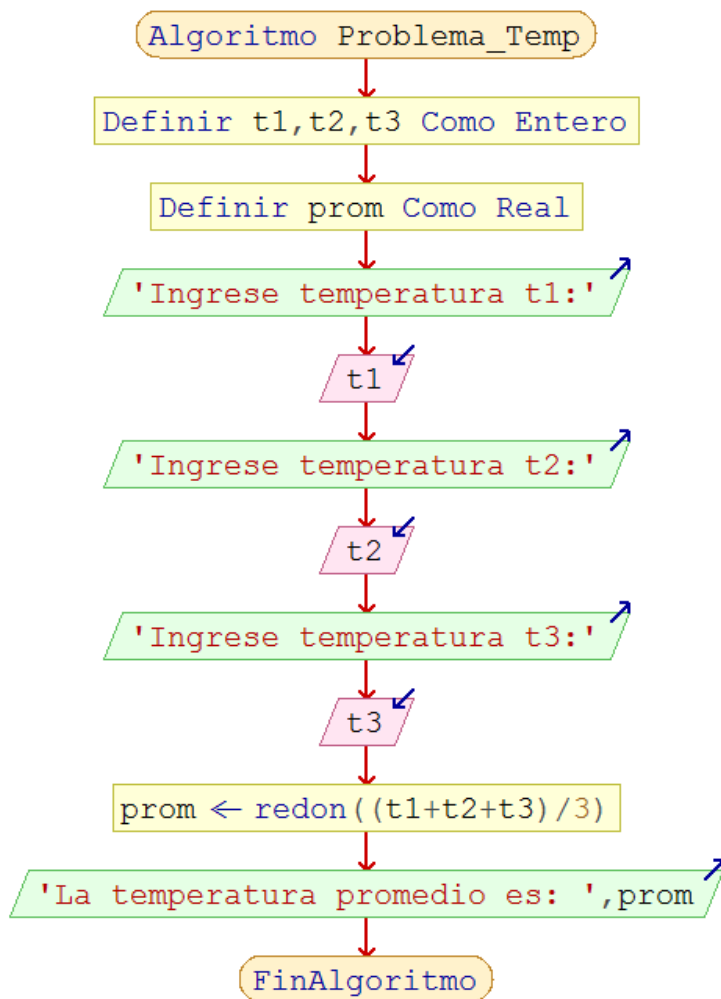
Se tiene registrada la temperatura ambiente medida en tres momentos diferentes en un depósito de químicos y se necesita calcular el valor promedio entre las temperaturas medidas en formato entero (sin decimales).

Los pasos a seguir son los siguientes:

1) Realizamos un análisis del problema:



- 2) Diagramamos la lógica de resolución mediante un diagrama de flujo:



En pseudocódigo:

Algoritmo Problema_Temp

Definir t1,t2,t3 Como Entero

Definir prom Como Real

Escribir 'Ingrese temperatura t1:'

Leer t1

Escribir 'Ingrese temperatura t2:'

Leer t2

Escribir 'Ingrese temperatura t3:'

Leer t3

prom <- redon((t1+t2+t3)/3)

Escribir 'La temperatura promedio es: ',prom

FinAlgoritmo

- 3) Traducimos el diagrama a líneas de código Java: (Opción Archivo>>Exportar de PSeInt)

```
import java.io.*;
```

```
import java.math.*; //quitar esta línea...
```

```
public class Problema_temp {
    public static void main(String args[]) throws IOException {
```



```
        BufferedReader bufEntrada = new BufferedReader(new
        InputStreamReader(System.in));
        int prom;
        int t1;
        int t2;
        int t3;
        System.out.println("Ingrese temperatura t1:");
        t1 = Integer.parseInt(bufEntrada.readLine());
        System.out.println("Ingrese temperatura t2:");
        t2 = Integer.parseInt(bufEntrada.readLine());
        System.out.println("Ingrese temperatura t3:");
        t3 = Integer.parseInt(bufEntrada.readLine());
        prom = Math.round((t1+t2+t3)/3);
        System.out.println("La temperatura promedio es: "+prom);
    }
}
```

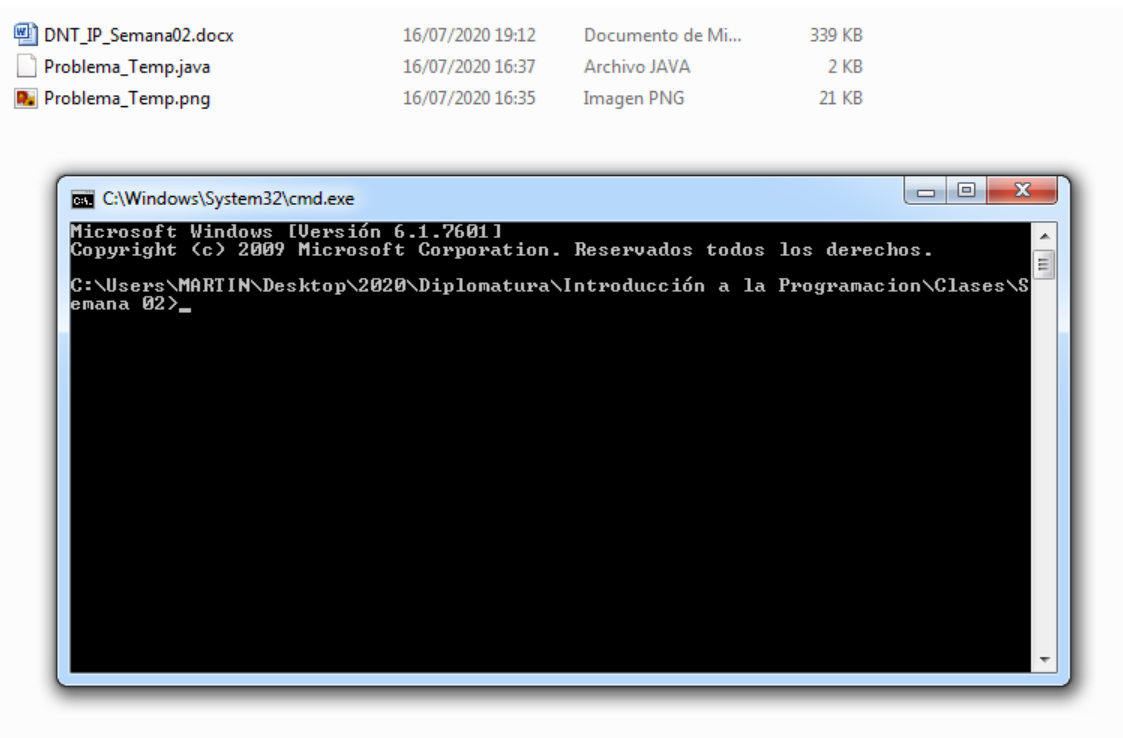
- Si bien el código generado por el diagramador es correcto, observar que la segunda línea corresponde con importar el paquete `java.math.*`. Este no es necesario para el caso planteado.
- El nombre del archivo `.java` generado deberá coincidir con el nombre de la clase pública que contiene, porque de otra forma no compilará. En este caso, la clase debería llamarse **Problema_Temp**.

2.2.2 Cómo ejecutar el programa generado por PSeInt

Con la facilidad prevista por el diagramador, nuestro primer desafío para con el lenguaje es compilar y ejecutar el programa generado.

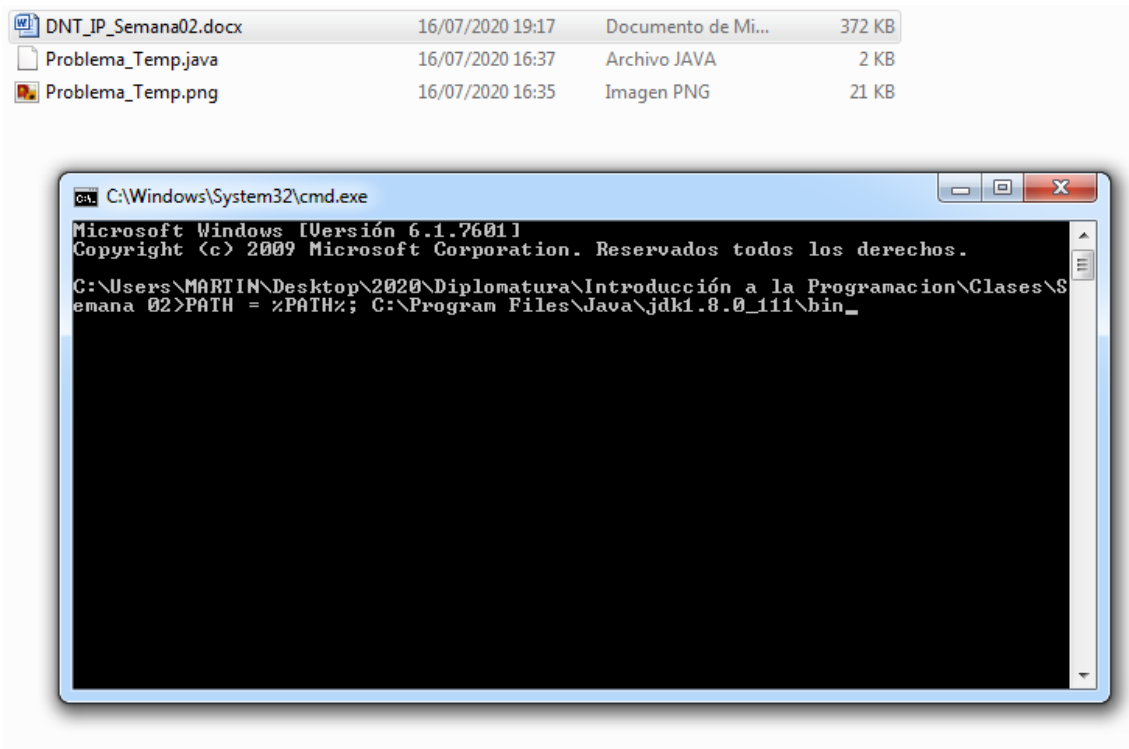
Pasos para compilar y ejecutar:

- Ejecutar un ventana de comandos en la carpeta donde PSeInt guardó los archivos. Para ello posicionarse en la barra de dirección del explorador y escribir `cmd + <Enter>`.

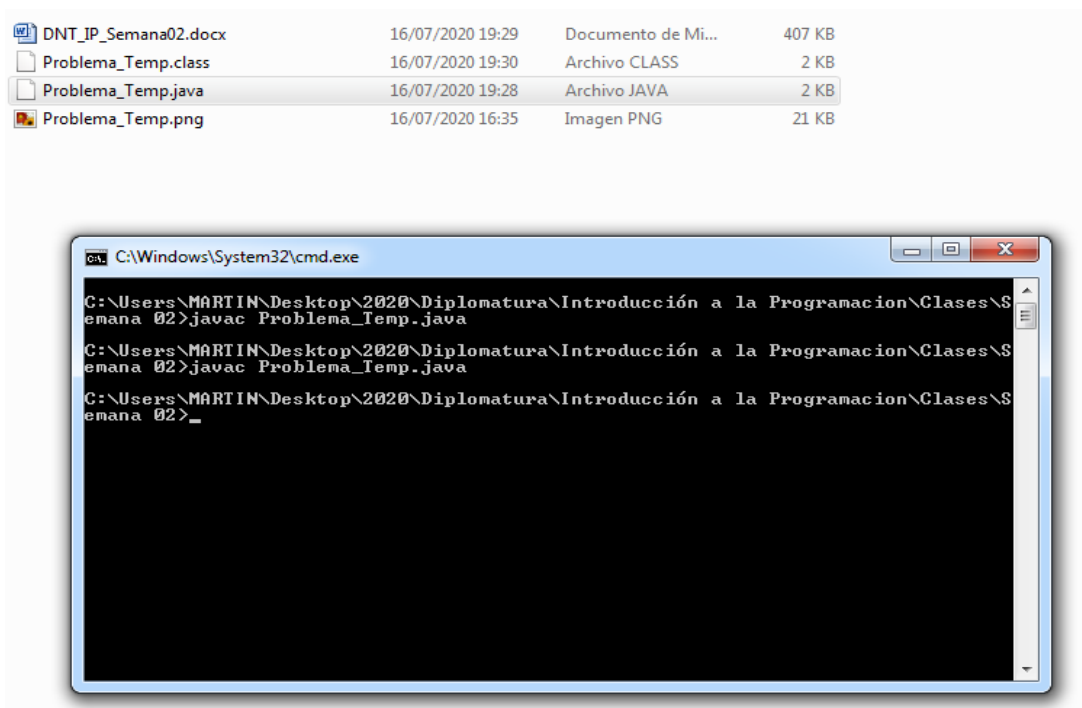


- En la ventana de comandos validar el contenido de la variable PATH. Si la variable no tiene la ruta a las herramientas de desarrollo del JDK que tiene disponible en su equipo, modificar el contenido de la variable mediante:

PATH = %PATH%; "Ruta a la carpeta \bin del JDK instalado"

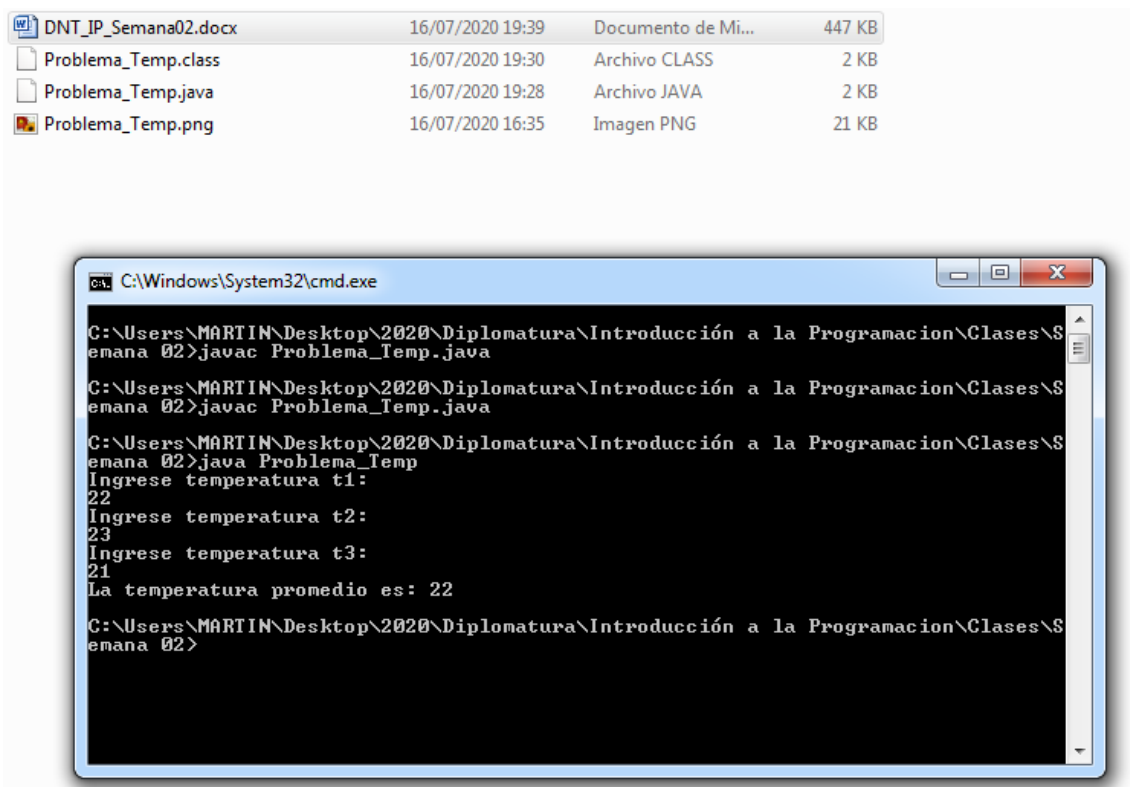


- Por último utilice las herramientas: javac.exe y java.exe para compilar y ejecutar respectivamente el programa. Para compilar el programa ejecutar >> javac Problema_Temp.java. Notar que es necesario indicar la extensión .java.



Notar que son hay problemas de sintaxis, el problema compilará y en la carpeta de trabajo se genera un archivo **Problema_Temp.class**. Este achivo es nuestro programa comilado (en formato Bycode) que podrá ser interpretado por cualquier máquina virtual (JVM).

Por último utilizamos la herramienta java.exe para ejecutar el programa.



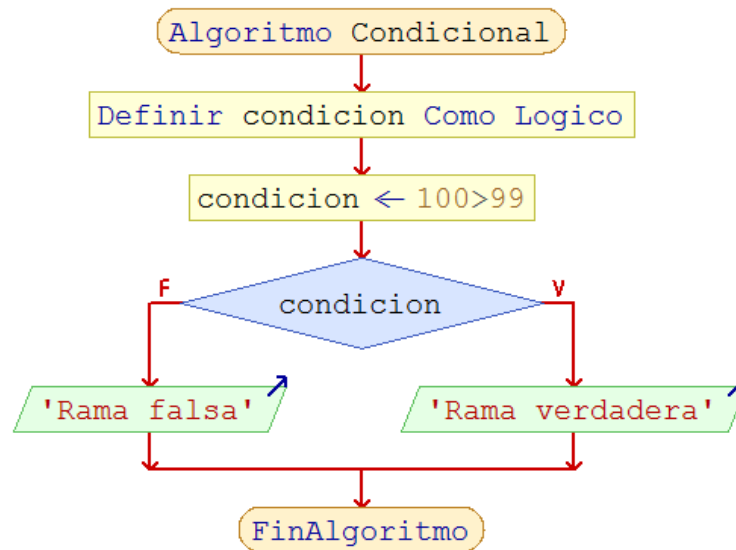
Notar que para ejecutarlo no es necesario indicar la extensión del archivo, solo ingresar >>**java Problema_Temp.java**.

3. Estructuras condicionales

En problemas que no sean absolutamente triviales, es muy común que en algún punto se requiera comprobar el valor de alguna condición, y en función de ello proceder a **dividir la lógica del algoritmo en dos o más ramas o caminos de ejecución**. Para casos así, los lenguajes de programación proveen instrucciones específicamente diseñadas para el chequeo de una o más condiciones, permitiendo que el programador indique con sencillez lo que debe hacer el programa en cada caso. Esas instrucciones se denominan estructuras condicionales, o bien, instrucciones condicionales.

En general, una instrucción condicional contiene una expresión lógica que puede ser evaluada por verdadera o por falsa, y dos bloques de instrucciones adicionales designados en general como la salida o rama verdadera y la salida o rama falsa. Si un programa alcanza una instrucción condicional y en ese momento la expresión lógica es verdadera, el programa ejecutará las instrucciones de la rama verdadera (y sólo esas). Pero si la expresión es falsa, el programa ejecutará las instrucciones de la rama falsa (y sólo esas).

3.1 Condicional simple y doble



En el lenguaje **Java**, una instrucción condicional típica como la que se muestra en la figura anterior, se escribe (esquemáticamente) así:

```
if (expresión lógica) {  
    instrucciones de la rama verdadera  
}  
else {  
    instrucciones de la rama falsa  
}
```

- La expresión lógica que se quiere evaluar es siempre una expresión o fórmula cuyo resultado es un valor lógico (o valor de verdad).
- La “rama verdadera” se escribe entre llaves, inmediatamente después de cerrar el paréntesis de la condición, y la “rama falsa” va después de la rama verdadera, también envuelta entre llaves, pero precedida de la palabra reservada **else**.

- La rama else es opcional, es decir podría no estar. En tal caso se dice que la estructura es **condicional simple**. Su forma completa (con ambas ramas) se denomina **condicional doble**.
- Dentro de la rama verdadera como en la falsa, es posible incluir nuevamente condicionales simples o dobles, en tal caso hablamos de anidamientos de condicionales.

3.1.1 Operadores relacionales

Para el planteo de expresiones lógicas, todo lenguaje de programación provee operadores que implican la obtención de un valor de verdad como resultado. La tabla a continuación muestra los operadores de relacionales o de comparación utilizados en Java:

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
==	igual que	7 == 38	false
!=	distinto que	'a' != 'k'	true
<	menor que	'G' < 'B'	false
>	mayor que	'b' > 'a'	true
<=	menor o igual que	7.5 <= 7.38	false
>=	mayor o igual que	38 >= 7	true

3.1.2 Conectores lógicos

Realizan operaciones sobre datos booleanos y tienen como resultado un valor booleano. En la siguiente tabla se resumen los diferentes operadores de esta categoría.

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
!	Negación - NOT (unario)	<code>!false</code> <code>!(5==5)</code>	true false
 	Suma lógica con cortocircuito: si el primer operando es true entonces el segundo se salta y el resultado es true	<code>true false</code> <code>(5==5) (5<4)</code>	true true
&&	Producto lógico con cortocircuito: si el primer operando es false entonces el segundo se salta y el resultado es false	<code>false && true</code> <code>(5==5) && (5<4)</code>	false false

- Los operadores lógicos permiten plantear condiciones lógicas compuestas, reduciendo en ciertas ocasiones la complejidad de la lógica diseñada para ciertos problemas
- Los operadores **OR** (||) y **AND** (&&) en Java, como en la mayoría de los lenguajes, son cortocircuitados. Esto implica que:
 - `exp1 && exp2 =>` si **exp1** es **falsa** entonces **exp2** no se evalúa
 - `exp1 || exp2 =>` si **exp1** es **verdadera** entonces **exp2** no se evalúa.

3.1.3 Operador Ternario

Existe un operador adicional que permite evaluar una expresión lógica y devolver un valor en caso de ser verdadero y uno diferente en caso contrario. Lo importante es que ambos valores son del mismo tipo. Este operador recibe el nombre de **ternario**, y puede ejemplificarse:

```
int entrada = -1;
```

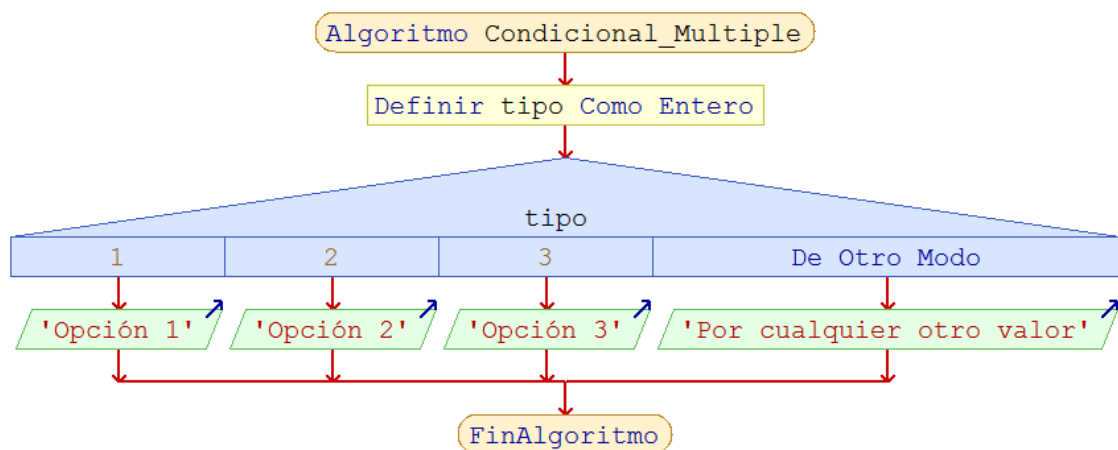
```
int valor = entrada > 0 ? entrada : 0;
```

```
Sistem.out.println(valor);
```


La salida por consola sería 0, ya que la condición planteada es falsa y el operador devuelve el valor indicado luego de los dos puntos (:).

3.2 Condicional múltiple

Cuando es necesario evaluar un valor con más de dos elecciones posibles, se puede resolver con estructuras alternativas anidadas o en cascada, o si se quiere con estructuras simples en secuencia. Sin embargo, si se tiene un problema donde el número de alternativas es grande, usar estos métodos, puede plantear serios problemas de escritura del algoritmo y naturalmente de legibilidad. Es por eso que existe una estructura llamada **condicional múltiple** que considera estos casos en particular y que podemos representar como:



En **Java**, la sintaxis para esta estructura es la siguiente:

```
switch (expresión_entera) {

    case (valor1) : instrucciones_1; [break;]

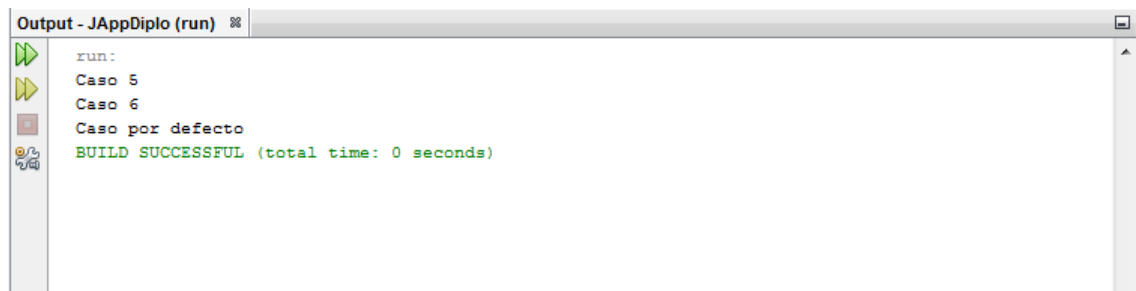
    case (valor2) : instrucciones_2; [break;]
```

```
...  
  
case (valorN) : instrucciones_N; [break;]  
  
default: instrucciones_por_defecto;  
  
}
```

- La expresión entera puede ser de tipo byte, short o int. Si colocamos una variable de tipo **long** no compilará.
- A partir de la versión Java 7, es posible incluir como expresión de selección una variable de tipo String.
- Notar que si bien la palabra reservada **break** se indica como opcional para cada caso, si se omite y un case se evalúa como verdadero, en resto de casos se evaluarán secuencialmente independientemente si son coincidentes o no. Por ejemplo:

```
public class TestSwicth {  
    public static void main(String[] args) {  
        int valor = 5;  
        switch (valor) {  
            case 5:  
                System.out.println("Caso 5");  
            case 6:  
                System.out.println("Caso 6");  
            default:  
                System.out.println("Caso por defecto");  
        }  
    }  
}
```

Mostrará como resultado:



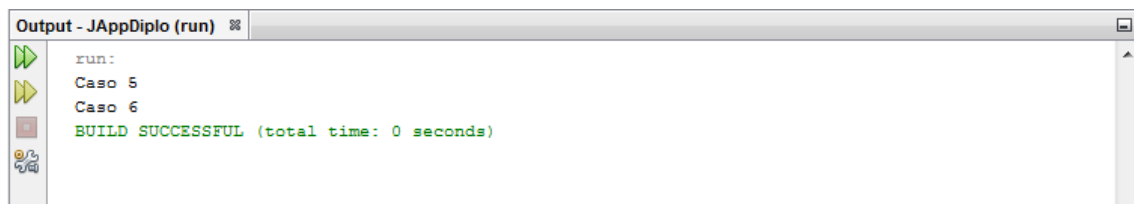
Al estar ausente la sentencia break; la estructura no corta luego de encontrar un caso exitoso y continúa evaluando secuencialmente el resto de los casos.

- Generalmente, y por buena práctica, el caso **default** se indica al final de todos los casos y sin una sentencia break, lo que no quita que puede ubicarse entre los casos. Por ejemplo:

```
public class TestSwicth {  
  
    public static void main(String[] args) {  
  
        int valor = 5;  
  
        switch (valor) {  
  
            default:  
  
                System.out.println("Caso por defecto");  
  
            case 5:  
  
                System.out.println("Caso 5");  
  
            case 6:  
  
                System.out.println("Caso 6");  
  
        }  
  
    }  
}
```

}

Mostrará como resultado:

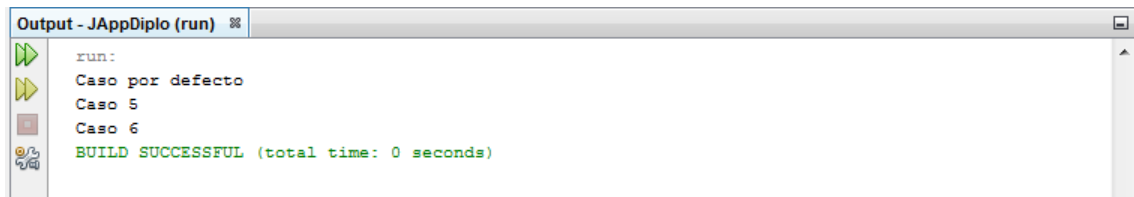


Notar que en esta oportunidad se alcanza el caso 5 y al no tener break, el caso 6 también se evaluará. El caso por defecto, al no estar luego del primer caso evaluado exitosamente, simplemente no se evalúa. Si ningún caso se evalúa exitosamente, entonces el caso por defecto si evaluará.

Por último analicemos el siguiente ejemplo:

```
public static void main(String[] args) {  
  
    int valor = 7;  
  
    switch (valor) {  
  
        default:  
  
            System.out.println("Caso por defecto");  
  
        case 5:  
  
            System.out.println("Caso 5");  
  
        case 6:  
  
            System.out.println("Caso 6");  
  
    }  
  
}  
  
}
```

La salida será:



```
run:
Caso por defecto
Caso 5
Caso 6
BUILD SUCCESSFUL (total time: 0 seconds)
```

En este caso como ningún caso se evalúa como verdadero, la variable **valor** se inicializa en 7, entonces se evalúa el caso por defecto como exitoso y a partir de allí la lógica es la misma que se indicó en los ejemplos anteriores.