UCA

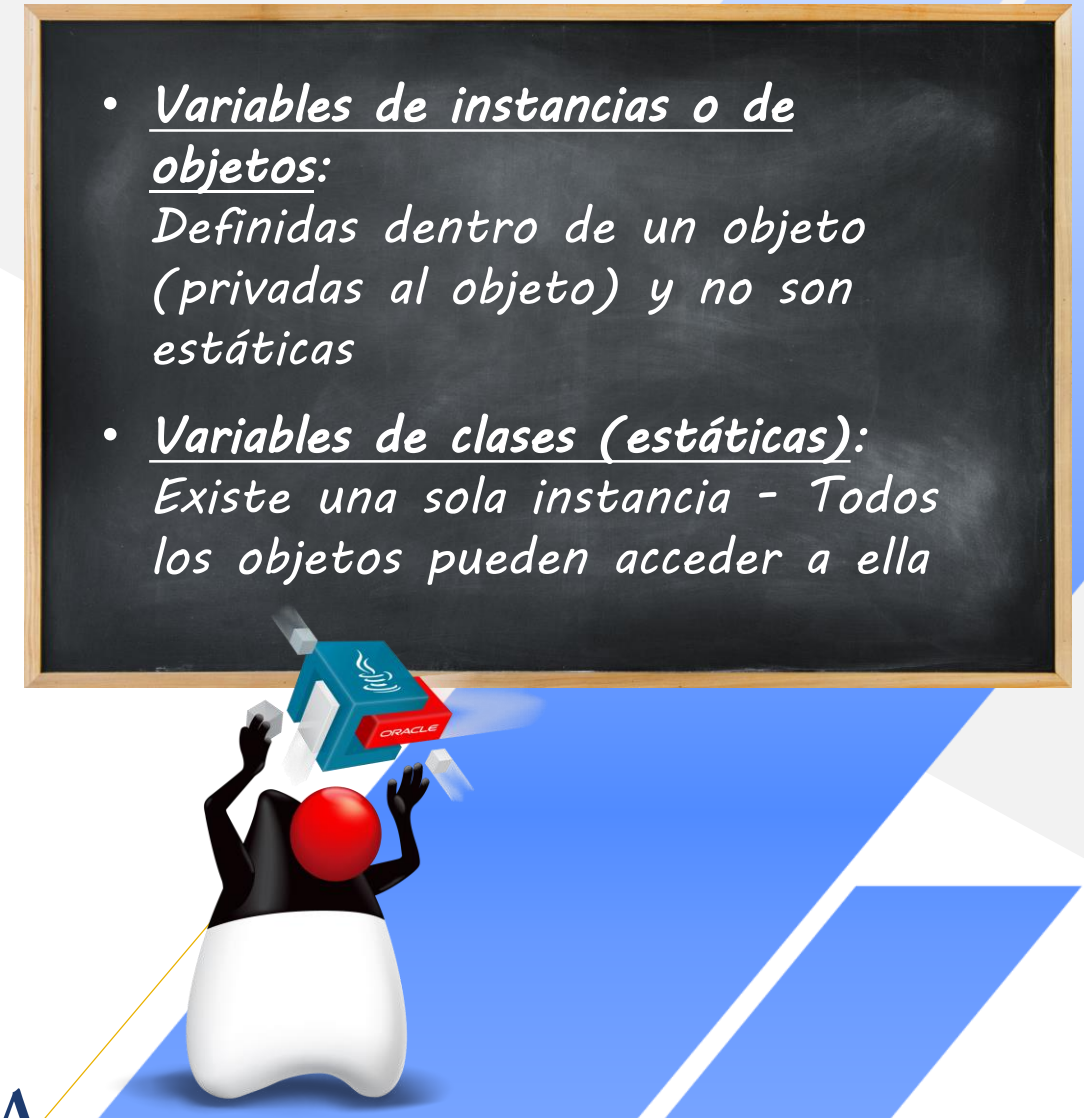
Programación Orientada a Objetos I

JAVA – ELEMENTOS BÁSICOS (II)

Constantes

- No existen las constantes locales en JAVA.
- Solo está permitido el uso de constantes de instancias u objetos y de variables de clases, anteponiendo el uso de la cláusula *final*.
- Por convención, se utilizan letras mayúsculas para definirlas.
- Ejemplo:

```
final int CONST_INT= 10;
```



Literales

- Es el conjunto de los ***valores posibles*** que puede manejar una variable de un programa JAVA.
- Existe un literal para cada tipo de dato:
  - literales de números
  - literales booleanas
  - literales de carácter
  - literales de cadenas

Literales

Literales de números - int

- Los literales de enteros son números de 32 bits con signo (tipo int).
- Aquellos que comienzan con 0 son octales, los que comienzan con 0x o 0X son hexadecimales y los que empiezan con 0b son binario (version 7.0)
- Ejemplos de literales enteras int:

```
255, 0377, 0xFF, 0xff, 0XFF, 0b11111111
                    // todos son el mismo valor
```

Literales de números - long

- Los literales de enteros más grandes se consideran como tipo long (64 bits con signo).
- Pero un literal int puede ser tratado como un literal long, agregándole la letra “L” después del valor.
- Ejemplo de literales enteras long:

```
3.000.000.000.000
255L          //L=forzado a ser literal long
255l          //l=forzado a ser literal long
```

Literales

Literales de números - double

- Los literales con un exponente o un punto decimal, resultan en un número de punto flotante de doble precisión (tipo double).
- Pero un literal double puede ser tratado como un literal float, agregándole la letra “F” después del valor.
- Ejemplo de literales float y double:

  double: 3.14, .1, .25

  float: 3.14F, .1f

  //F=forzado a ser literal float

Literales booleanos

- Los literales booleanos consisten en las palabras claves true o false (tipo boolean)

Literales de carácter

- Los literales de carácter están representados por un solo carácter encerrado entre comillas simples, almacenándose como UniCode de 16 bits (tipo char)
- Ejemplos de literales char:

  'a', '@', '6'

Caracteres especiales

Escape	Significado
\n	Nueva línea
\t	Tabulador
\b	Retroceso
\r	Retorno de carro

Escape	Significado
\\	Barra invertida
\'	Comilla simple
\"	Comillas dobles
\udddd	Carácter Unicode (d son dígitos hexadecimales)

Todos los caracteres Unicode:

http://www.hipenpal.com/tool/characters_to_unicode_charts_in_spanish.php

Literales

Literales de Cadenas

- Los literales de cadenas (String) son una serie de caracteres encerrados entre comillas dobles.
  Ejemplos:

```
"Hola"
""                    // String vacío
```

- Se puede incluir cualquier código especial (nueva línea, tabulación, etc.)
  Ejemplo:

```
"Un string con un \t tab incluido"
"O esto: \u255A\u2550\u2550\u2557"  // Se ve así:
```

Casting

- Cast o casting es la conversión de un tipo de dato a otro y se utiliza para asegurarse de que un valor se almacene o se interprete correctamente
- Cast Implícito: es automático
  La jerarquía en las conversiones de menor a mayor es:

  byte → short → int → long → float → double
- Cast Explícito: forzado por el programador, indicando el nuevo tipo entre paréntesis.
  Suponiendo:

```
int entero;
double decimal=10.8;
entero = decimal;     // Esto dará error de compilación:
                      // Type mismatch: cannot convert from double to int
entero = (int) decimal;      // Esto no. Al convertir el 10.8 en entero
                             // desaparecen los decimales
```

Operadores

Expresión: es cualquier enunciado que devuelve un valor.

Operador: es el símbolo que interviene en una expresión. Ejemplo:

```
C =  A + B
// Todo es una expresión. El signo + es un operador
```

Tipos de operadores:

- Matemáticos
- De Asignación
- Incremento/Decremento
- De Comparación
- Lógicos
- De Bits
- De Bits y de Asignación

Operadores

Operadores Matemáticos

+	suma
-	resta
*	multiplicación
/	división
%	Módulo

Operadores de Asignación

=	asignación
+=	suma y asignación
-=	resta y asignación
*=	multiplicación y asignación
/=	división y asignación
%=	módulo y asignación

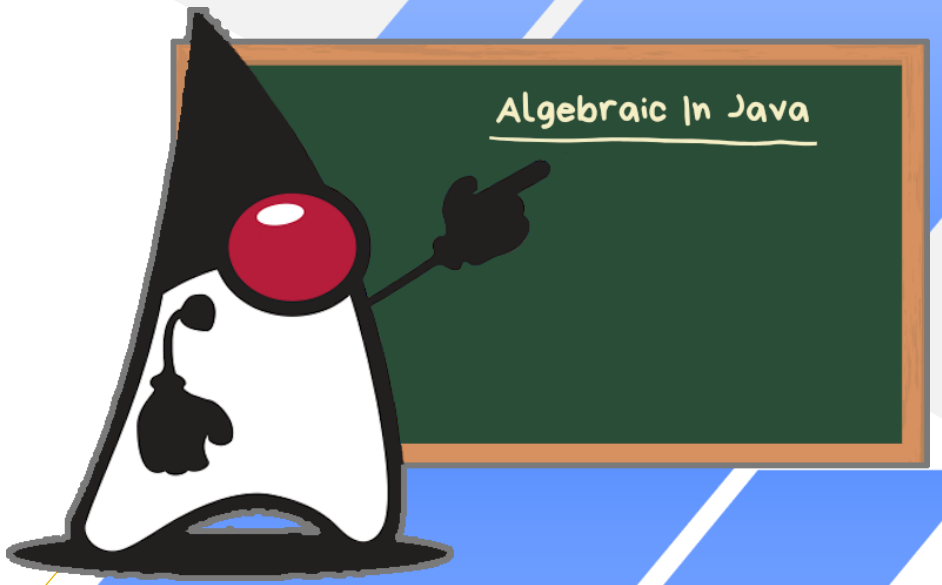
Operadores

Incremento y decremento (Unarios)

- i++ Post Incremento
- i-- Post Decremento
- ++i Pre Incremento
- --i Pre Decremento

Otros Operadores Unarios

- \- Negación
- ~ Complemento



Operadores

Operadores de Comparación

==	Igual
!=	Distinto
<	Menor
>	Mayor
<=	Menor e igual
>=	Mayor e igual

Operadores Lógicos

&	AND (completo)
&&	AND (cortocircuito)
|	OR (completo)
||	OR (cortocircuito)
^	XOR
!	NOT (unario)

Operadores

Operadores de bits

&	AND
|	OR
^	XOR
<<	Desplazamiento a la izquierda
>>	Desplazamiento a la derecha
>>>	Llenado de ceros a la derecha
~	Complemento de bits

Operadores de bits y asignación

&=	Asignación AND (x = x & y)
|=	Asignación OR (x = x | y)
^=	Asignación XOR (x = x ^ y)
<<=	Asignación del desplazamiento a la izquierda (x = x << y)
>>=	Asignación del desplazamiento a la derecha (x = x >> y)
>>>=	Asignación de llenado de ceros hacia la derecha (x = x >>> y)

Precedencia

1. Unarios:
   +, -, ++, --, !, ~
2. Multiplicación, división:
   *, /, %
3. Suma y resta:
   +, -
4. Desplazamiento:
   <<, >>, >>>
5. Relacionales:
   <, >, <=, >=
6. Igualdad:
   ==, !=
7. Lógicos completos y cortocircuito:
   &, ^, |, &&, ||
8. Condicionales:
   ? : (abreviación de *if/then/else*)
9. Asignación:
   =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=

Caracteres especiales

() paréntesis:

- Listas de parámetros en la definición y llamada a métodos
- Precedencia en expresiones
- Expresiones booleanas
- Conversión de tipos

{} llaves:

- Bloque de sentencias de código (clases, métodos)
- Valores de matrices inicializadas automáticamente

[] corchetes:

- Declaración y referencia a valores de vectores y matrices

; punto y coma:

- Separador de sentencias

, coma:

- Separador de: declaraciones de variables, lista de argumentos, elementos de arrays, incremento/decremento en una sola línea

. punto:

- Referencia a un método o propiedad de un objeto o de una clase (operador de ámbito).

Control de flujo

- Las sentencias o instrucciones en código JAVA definen y controlan el flujo del programa desarrollado.
- Si bien se ejecutan en el orden en que fueron escritas, existen muchas sentencias que alteran el orden del flujo del programa.
- Las sentencias que alteran el orden consecutivo de ejecución de un programa, se agrupan de acuerdo a sus características en los siguientes ítems:
  - Sentencias de ***salto condicionales***
  - Sentencias de ***bucles o ciclos***

Salto condicional: IF

```
if (condicion)
   sentencia;
```

o bien:

```
if (condicion)
   sentencia;
else
   sentencia;
```

```
if (condicion)
   {sentencias};
```

o bien:

```
if (condicion)
   {sentencias};
else
   {sentencias};
```

Salto condicional: SWITCH

```
switch (expresion) {
    case valor1:
    case valor2:
        sentencias;
        break;
    case valor3:
        sentencias;
        break;
    default:
        sentencias;
        break;
}
```

- ***expresion***: solo acepta los siguientes tipos de datos:
  - byte
  - char
  - short
  - int
  - String: a partir de la versión 7.0
- **break**: evita la ejecución de las sentencias indicadas en otros valores consecutivos.

Salto: TRY-CATCH

- Controla el flujo de ejecución como consecuencia de la generación de excepciones.
- Excepción: es un error grave que ocurre en tiempo de ejecución.
- Se genera en la JVM como consecuencia o respuesta a una condición inesperada, o bien en el código como resultado de la instrucción *throw*.
- Su propósito principal es permitir que un programa continúe ejecutándose sin interrupciones graves, incluso si se produce una excepción
- Incluye:
  - Un solo bloque try
  - Ninguno, uno o muchos bloques catch
  - Opcionalmente un bloque finally.

Salto: TRY-CATCH

- A partir de un bloque try, debe existir al menos un bloque catch o un bloque finally.
- De acuerdo, al tipo de excepción o error generado, se ejecuta el bloque catch correspondiente y luego, el bloque finally, si existiera.
- El bloque catch incluye todas las sentencias necesarias para el tratamiento de la excepción.
- El bloque finally incluye todas las sentencias necesarias se provoque o no la excepción.

Salto: TRY-CATCH

Bloque try: En este bloque se coloca el código que podría generar una excepción. Si ocurre una excepción en cualquier parte del bloque *try*, el flujo de control se desvía inmediatamente al bloque *catch*. El bloque *try* debe estar seguido por un bloque *catch* o un bloque *finally*.

```
try {
    // Código que podría generar una excepción
} catch (TipoDeExcepcion e) {
    // Manejo de la excepción
}
```

Salto: TRY-CATCH

Bloque catch: En este bloque se especifica el tipo de excepción que se desea manejar y el código que se ejecutará si se produce esa excepción. El parámetro e (puede tener cualquier nombre) se utiliza para hacer referencia a la excepción que se ha lanzado.

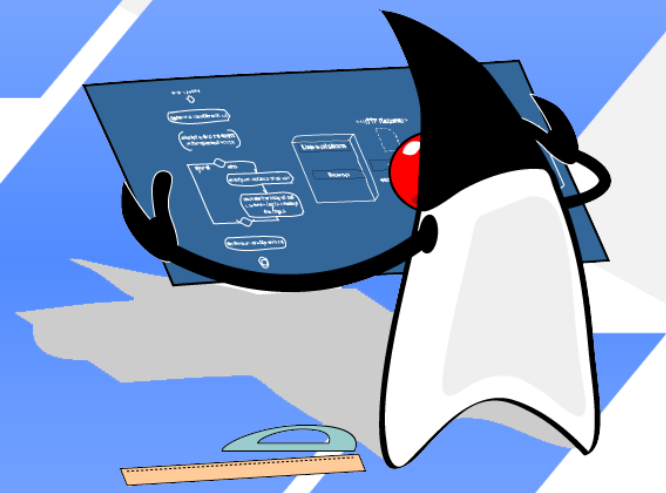
Se puede tener múltiples bloques *catch* después de un solo bloque *try*. Esto permite manejar diferentes tipos de excepciones de manera personalizada.

```
try {
    // Código que podría generar una excepción
} catch (TipoDeExcepcion1 e) {
    // Manejo de TipoDeExcepcion1
} catch (TipoDeExcepcion2 e) {
    // Manejo de TipoDeExcepcion2
}
```

Salto: TRY-CATCH

Bloque finally (opcional): Después de los bloques *try* y *catch*, se puede incluir un bloque *finally*. El código en este bloque siempre se ejecutará se haya producido una excepción o no.

```
try {
    // Código que podría generar una excepción
} catch (TipoDeExcepcion1 e) {
    // Manejo de TipoDeExcepcion1
} catch (TipoDeExcepcion2 e) {
    // Manejo de TipoDeExcepcion2
} finally {
    // Código que siempre se ejecutará
}
```



Ejemplo

```
try {
    // Intentamos dividir 10 por 0, lo que generará una excepción ArithmeticException
    int resultado = 10 / 0;
    System.out.println("El resultado es: " + resultado); // Esta línea nunca se ejecutará
} catch (ArithmeticException e) {
    // Capturamos la excepción y manejamos el error
    System.out.println("Se ha producido un error de división por cero.");
} finally {
    // Este bloque se ejecutará siempre, se produzca o no una excepción.
    System.out.println("Este bloque siempre se ejecuta.");
}
// El programa continúa ejecutándose después del manejo de la excepción
System.out.println("El programa continúa ejecutándose.");
```

Bucle: FOR

Se utiliza para repetir un bloque de código un número específico de veces o para iterar sobre una colección de elementos (como un arreglo o una lista). Es especialmente útil cuando se sabe exactamente cuántas veces se desea ejecutar el bloque de código.

```
for(inicialización; condición; actualización)
    { sentencias; }
```

El flujo típico de un bucle *for* es el siguiente:

1. Se ejecuta la inicialización **una vez al principio**.
2. Se evalúa la condición. **Si la condición es verdadera**, se ejecuta el bloque de código dentro del bucle.
3. Después de ejecutar el bloque de código, **se realiza la actualización**.
4. Se vuelve a evaluar la condición. Si es verdadera se repite el proceso, de lo contrario el bucle se detiene

Bucle: FOR

Ejemplos:

```
for (int i=0; i<10; i++)
      System.out.println("Hola!");     // 10 repeticiones (0 a 9)

for (int i = 1; i <= 5; i++) {
    System.out.println(i);             // Imprime del 1 al 5
                                       // es un menor o igual
}

for (int a=0, b=10; a<b; a++, b++);    // bucle infinito (¿Por qué?)
```

Bucle: FOR Avanzado (para colecciones)

Similar al ciclo *for*, pero utilizado para recorrer los objetos de una colección.

No necesita inicializar el contador, ni comprobar una condición ni actualizar el valor del índice.

```
for (objeto: colección)
    { sentencias; }
```

Ejemplo:

```
String[] nombres = {"Juan", "Pedro", "Dario", "Carlos"};
for(String nombre:nombres)
    System.out.println("Los nombres son: " + nombre);
```

Bucle: WHILE y DO/WHILE

Repiten el bloque de código mientras se cumpla una condición.

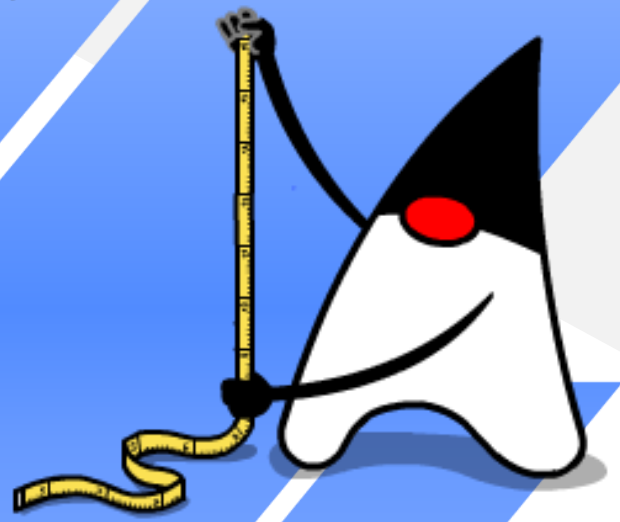
```
1) while (condición)        // Primero se verifica la condición
       { sentencias; }   // Si es verdadera se ejecuta el código

2) do
       { sentencias; }   // Primero se ejecuta el código
   while (condición);      // Y si es verdadera se ejecuta otro ciclo
```

- *break*: detiene la ejecución del ciclo actual, continuando en la sentencia siguiente del ciclo.
- *continue*: en lugar de detener la ejecución del ciclo, inicia otra vez con la siguiente iteración.

Array (Vectores)

- Un array o vector contiene una colección de elementos del mismo tipo que JAVA almacena y manipula usando un índice de tipo entero.
- Se pueden declarar vectores o array de cualquier tipo primitivo, de clases o de otro array (matrices).
- El tipo de dato especificado, determina el número de bytes reservado en memoria para cada posición del array
- Se dimensionan en tiempo de ejecución.
- Se crean (al igual que los objetos) con el operador *new*
- Los vectores se consideran *objetos*, que tienen métodos asociados, como por ejemplo *length* (longitud del array).



Array (Vectores)

Índice	Elementos
0	100
1	125
2	224
3	15
4	9

- El primer elemento de un array ocupa la posición cero del mismo, por lo tanto se dice que JAVA es zero-based array.
- Etapas necesarias para el uso de array:
  - Declarar una variable para almacenar el array
  - Crear un nuevo objeto de array y asignárselo a la variable
  - Almacenar y obtener los datos del array

Array (Vectores)

Para declarar un array se debe indicar el tipo de elementos, el nombre del array y ambos corchetes.

```
tipoDato nombreArray[];   o bien:  tipoDato[] nombreArray;
```

Ejemplos:

```
int duracion[];

short[] tamaño;
```

Para definir arrays de más de una dimensión, agregar más corchetes en la declaración y creación. Ejemplo:

```
double dosDArray[][] = new double[4][25];

// declara y crea un array bidimensional (4 x 25)
```

Array (Vectores)

Para crear instancias de la clase Array, se utiliza el operador new, el tipo de dato asignado y la longitud del array.

```
nomArray = new tipoDato [nPos];
```

Ejemplo:

```
int duracion[];
duracion[] = new int[4]; //4 posiciones (de 0 a 3)
```

No se pueden crear arrays estáticos en tiempo de compilación

```
int lista[50]; // Esto va a dar error!!!
```

Array (Vectores)

- Los límites del array se comprueban en tiempo de ejecución para evitar desbordamientos.
- No se pueden rellenar sin haber declarado el tamaño con el operador new.
- También, se puede combinar la declaración y la inserción de elementos en el vector, usando un inicializador de array.
- **Inicializador de array**: indica el conjunto de elementos que participa del mismo, separados por coma y encerrados entre llaves.
- No requiere el uso del operador new ni ser dimensionado previamente.
- Ejemplo:

```
int enteros [] = {15,34,67,31,59};  // 5 posiciones enteras
```

Array (Vectores)

Ejemplo para recorrido de vectores con ciclos FOR:

```
int[] notas = {4, 8, 7, 9};
    for(int i=0; i<4; i++){       // Ciclo FOR normal
        System.out.println("Nota: " + notas[i]);
    }
    for(int nEntero : notas) {   // Ciclo FOR avanzado
        System.out.println("Nota: " + nEntero);
    }
```

Array

- Podemos conocer el tamaño de un array con el método *length*. Eso significa que el índice de un array irá desde cero hasta *length* - 1

```
arreglo.length;
```

- Ejemplo de uso:

```
int a[][] = new int[10][3]; // matriz de enteros(10x3)
a.length;                   // Es 10 (la 1er dimensión)
a[0].length;                // igual a 3
```