

Anuncios

- Clase práctica del viernes, 1-2:30 y 3-4:30 en el aula 26-152.
- Si ha traído su portátil, inícielo y arranque Forte.
 - Cree un archivo vacío llamado 'Lecture4' (clase 4) y un método *main()* vacío dentro de una clase:

```
public class Lecture4 {  
    public static void main(String[] args) {  
        // Escriba aquí su código  
    }  
}
```

1.00 Clase 4

**Más información sobre los tipos de datos en Java
Estructuras de control**

Operadores aritméticos en Java

Tabla por orden de prioridad, mayor prioridad al comienzo

Operadores	Significado	Asociatividad
++	incremento	De Derecha a izquierda
--	decremento	
+ (unario)	unario + (x = +a)	De Izquierda a derecha
- (unario)	unario - (x = -a)	
*	multiplicación	
/	división	
%	resto	
+	suma	De Izquierda a derecha
-	resta	

Prioridad y asociatividad

- **La prioridad entre operadores es como indica el orden de la tabla anterior:**
 - Los operadores de la misma fila poseen la misma prioridad:
int i=5, j= 7, k= 9, m=11, n;
n= i + j * k - m; // n= ?
- **La asociatividad determina el orden en el que se deben aplicar los operadores de la misma prioridad:**

int i=5, j= 7, k= 9, m=11, n;
n= i + j * k / m - k; // n= ?
- **Los paréntesis anulan el orden de prioridad:**
int i=5, j= 7, k= 9, m=11, n;
n= (i + j) * (k - m)/k; // n= ?

Prioridad y asociatividad, 2ª parte

- La prioridad del operador es como indica el orden de la tabla anterior.

– Los operadores de la misma fila poseen la misma precedencia.

```
int i=5, j= 7, k= 9, m=11, n;  
n= i +j * k - m; // n= 57
```

- La asociatividad determina el orden en el que se deben aplicar los operadores de la misma precedencia.

```
int i=5, j= 7, k= 9, m=11, n;  
n= i+j * k / m - k; // n= 1
```

- Los paréntesis anulan el orden de precedencia.

```
int i=5, j= 7, k= 9, m=11, n;  
n= (i+j) * (k - m)/k; // n=-2
```

Ejercicios sobre el operador

- **Cuál es el valor de int n:**

- n= 1 + 2 - 3 / 4 * 5 % 6;

- n= 6 + 5 - 4 / 3 * 2 % 1;

- i= 5; j= 7; k= 9;

- n= 6 + 5 - ++i / 3 * -j % k --;

- i= 5;

- n= i + ++i;

Ejercicios sobre el operador

- **Cuál es el valor de int n:**

- `n= 1 + 2 - 3 / 4 * 5 % 6;` //n=3
- `n= 6 + 5 - 4 / 3 * 2 % 1;` //n=11
- `i= 5; j= 7; k= 9;`
- `n= 6 + 5 - ++i / 3 * --j % k--;` // n=8
- `i= 5;`
- `n= i + ++i;` // n=11 en Java
// n=12 en C++
- // ¡Nunca haga nada de esto!

Aritmética mixta

- **Promoción:**

- Cuando dos operandos poseen tipos distintos, Java convierte el tipo de menor capacidad en el de mayor capacidad:

```
int i; // Máximo 2147483647
short j; // Máximo 32767
i= j; // ok
```

- **Conversión explícita:**

- Cuando desee convertir un tipo de mayor capacidad en uno de menor capacidad, debe indicarlo de forma explícita.

```
int i; short j;
j= i; // Ilegal. No compilará
j= (short) i; // convierte ((a)) i en un short int (entero corto)
```

- **Operadores binarios (+, -, *, /):**

- Si cualquiera de los operandos es (*double*, *float* o *long*), el otro se convertirá en (*double*, *float* o *long*).
- De lo contrario, ambos se convertirán en *int*.

Ejemplo de aritmética mixta

```
public class TypePromotion {
    public static void main(String[] args) {
        byte b= 5, bval;
        short s= 1000, sval;
        int i= 85323, ival;
        long l= 999999999999L, lval;
        float f= 35.7F, fval;
        double d= 9E40, dval;

        bval= b + s;           // no compilará
        bval= (byte) (b + s);  // conversión ok, desbordamiento
        sval= (short) (b + s); // ¡conversión requerida!
        fval= b + s + i + 1 + f; // float ok
        lval= b + s + i + 1;    // long ok
        ival= (int) f;         // truncación ok
    }
}
```

Propiedades de la aritmética entera

- El desbordamiento por exceso se da a partir de:
 - La división por cero, incluyendo 0/0 (no definido).
 - El programador es responsable de comprobar y prevenir esto.
 - Java le advertirá (mediante el lanzamiento de una excepción) si no es posible realizar una operación aritmética entera (esto se tratará próximamente).
 - La acumulación de resultados que excedan la capacidad del tipo entero que se está utilizando.
 - Al igual que en las divisiones por cero, el programador es responsable de comprobar y prevenir esto.
 - En este caso, Java no hace ninguna advertencia.

Ejemplo de desbordamiento por exceso de un entero

```
public class Intoverflow {
    public static void main(String[] args) {
        int bigval= 2000000000;
        System.out.println("bigval: " + bigval);
        bigval += bigval;
        System.out.println("bigval: " + bigval); } }
```

```
// salida
~~~~~: 2000000000
~~~~~: -294967296
```

Es necesario analizar el rango de los resultados bajo las peores circunstancias. A menudo se utiliza un entero largo (*long*) para almacenar sumas de enteros (*int*), etc.

Propiedades de números de coma flotante

- **Valores de coma flotante anómalos:**
 - **No definidos, como 0.0/0.0:**
 - $\pm 0.0/0.0$ produce un resultado NaN (*Not a Number*) (no numérico).
 - Cualquier operación que implique un NaN produce un resultado NaN.
 - Dos valores NaN no pueden ser iguales.
 - Compruebe si el número es NaN utilizando los métodos:
 - `Double.isNaN(double d)` o `Float.isNaN(int i)`
 - Devuelve un booleano verdadero si el argumento es NaN.
 - **Desbordamiento por exceso, como 1.0/0.0:**
 - 1.0/0.0 produce el resultado INFINITO-POSITIVO.
 - -1.0/0.0 produce el resultado INFINITO-NEGATIVO.
 - Mismas reglas, resultados como para NaN (`Double.isInfinite`)
 - **Desbordamiento por defecto, cuando el resultado es menor que el número más pequeño que podamos representar.**
 - Complejo, no tratado muy bien (representado como cero).
- **Errores de redondeo: observe los siguientes ejemplos**

Ejemplo

```
public class NaNTest {
    public static void main(String[] args) {
        double a=0.0, b=0.0, c, d; c= ab;
        System.out.println("c: " + c);
        if (Double.isNaN(c))
            System.out.println(" c es NaN");
        d= c + 1.0;
        System.out.println("d: " + d);
        if (Double.isNaN(d))
            System.out.println(" d es NaN");
        if (c == d)
            System.out.println("Oops");
        else
            System.out.println("NaN != NaN"); double
        e= 1.0, f;
        f= ea;
        System.out.println("f: " + f);
        if (Double.isInfinite(f))
            System.out.println(" f es infinito");
    } }
}
```

Programa para redondear un *Float*

```
public class Rounding {
    public static void main(String[] args) {
        System.out.println("Número de veces en las que el inverso != 1");
        for (int test=2; test < 100; test++) {
            float top= test;
            float bottom= 1.0F/test;
            if (top*bottom != 1.0F)
                System.out.println(test + " " + top*bottom);
        } }
}
```

//Salida:

```
41 0.99999994
47 0.99999994
55 0.99999994
61 0.99999994
82 0.99999994
83 0.99999994
94 0.99999994
97 0.99999994
```

**Se da también con los *double*.
Es prácticamente lo mismo,
dado que 1.0 es más
exacto que un *double*.**

Doubles, malos contadores de bucles

```
public class counter {
    public static void main(String[] args) {
        int i= 0;
        double c= 0.0;
        while (c!= 10.0 && i < 52) {
            c += 0.2;
            i++;
            if ( i % 10 == 0 || i >= 50)
                System.out.println("c: " + c + " i: " + i); } } }

//Salida
c: 1.9999999999999998 i: 10
c: 4.0000000000000001 i: 20
c: 6.0000000000000003 i: 30
c: 8.0000000000000004 i: 40
c: 9.9999999999999996 i: 50
c: 10.1999999999999996 i: 51
c: 10.3999999999999995 i: 52
```

Observe el error creciente y acumulativo. Nunca utilice floats o doubles como contadores de bucles.

Problemas numéricos

Problema	Entero	Float, double
División cero (excede)	Excepción lanzada El programa se bloquea a menos que se detecte	INFINITO_POSITIVO, INFINITO_NEGATIVO
0/0	Excepción lanzada El programa se bloquea a menos que se detecte	NaN (no numérico)
Desbordamiento	Sin aviso. El programa da resultados erróneos.	INFINITO_POSITIVO, INFINITO_NEGATIVO
Por defecto	No es posible	Sin aviso. Establecido a 0
Redondeo, errores de acumulación	No es posible	Sin aviso. El programa da resultados erróneos.

Habitual. Son malas noticias.

Más información sobre estructuras de control

- Tres estructuras de control:
 - Secuencia: ejecuta el próximo enunciado.
 - Este es el comportamiento por defecto.
 - Ramificación: enunciados *if*, *else*.
 - *If*, *else* son las principales construcciones que se utilizan.
 - Se utiliza el enunciado *switch* si se dan muchas opciones.
 - Iteración: bucles *while*, *do* y *for*
 - Existen construcciones adicionales para interrumpir los bucles “antes de tiempo”.

Enunciado *switch*

- Se utiliza como sustituto de largas cadenas *if-else*.
 - La condición de ramificación debe ser un tipo entero, no un *String*, *float*, etc.
 - En una sentencia *switch* no se pueden emplear rangos, sólo valores o expresiones simples.
 - C# permite *strings* (cadenas) como condición de ramificación. Esto no se permite en Java o C++.
- ```
int velocidad;
switch (velocidad/10) { // Límite= 9 millas por
 // hora (bicicleta)

 case 3:
 case 2:
 system.out.println("Arrestar"); // Continúa leyendo y
 // pasa al siguiente
 // caso

 case 1:
 system.out.println("Multa");
 break; // finaliza el enunciado switch
 //evitando que continúe leyendo el
 //siguiente caso

 case 0:
 system.out.println("velocidad legal");
 break;
 default:
 system.out.println("Falsa lectura del radar");
}
```

## Interrupción de la iteración: *Break*

- El enunciado *break* en bucles *for*, *while* o *do-while* transfiere el control al enunciado inmediatamente después del final del bucle.

```
int bajo= 8;
int alto= 12;
for (i=bajo; i < alto; i++) {
 System.out.println("i= " + i);
 if (i >= 9) {
 System.out.println("Demasiado alto");
 break;}
}
System.out.println("Próximo enunciado");

// la salida es
i= 8
i= 9

Demasiado alto
Próximo enunciado
```

## Interrupción de la iteración: *Continue*

- El enunciado *continue* salta al final del bucle pero continúa el bucle.

```
int sum= 0;
int bajo= -1;
int alto= 2;
for (i=bajo; i < alto; i++) {
 if (i == 0) {
 System.out.println("División entre 0 ");
 continue; }
 int q= 1/i;
 sum += q;
}
System.out.println("suma= " + sum); // suma = 0
```

## Ejercicio de iteración

- Recuerde la definición de factorial:

$$n! = n * (n-1) * (n-2) * \dots * 1$$

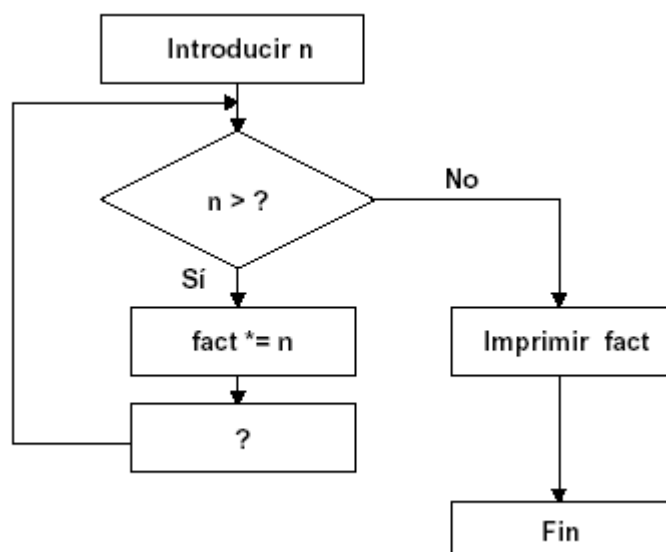
Por ejemplo:  $4! = 4 * 3 * 2 * 1 = 24$

- Un factorial posee las siguientes propiedades:
  - $0! = 1$
  - $n$  es un número entero positivo.
- Escriba un método *main()* que calcule el valor de  $n!$  para un  $n$  determinado. Utilice por ejemplo:

```
int n= 6;
```

```
// suponga n >=0; no compruebe
```

## Calculando el resultado



# Bucle controlado por centinela (Opcional)

Imagine que el usuario no quiere que el programa se ejecute sólo una vez. En vez de esto, desea que le pidan de nuevo introducir un número, y una vez hecho, que le pidan que introduzca otro.

Para ello, se utilizará un bucle controlado por centinela.

La idea del bucle controlado por centinela se basa en la existencia de un valor especial (“centinela”) que se utiliza para indicar cuándo se ha acabado el bucle.

En este ejemplo, el usuario introducirá “-1” para avisar al programa de que acabe.

Suponga que el usuario introduce el número mediante `JOptionPane`. Si escribe el código a mano, no se preocupe por la sintaxis exacta de `JOptionPane`; suponga simplemente que el usuario introduce un número válido.

Corrija su programa.