Introducción a Java: variables y control de flujo

Variables

Vimos que hay varios tipos de **variable** en Java:

- `int` para números enteros
- `double` para los números reales representables (si les interesa, luego les explico cuántos números podemos representar en la computadora y cómo se hace);
- `boolean` para variables booleanas (les recuerdo que usamos notación de lógico, $\top$ para verdadero y $\bot$ para falso)
- `String` para cadenas de texto (énfasis en la S mayúscula, les explico cuando veamos programación orientada a objetos).

Vimos que para inicializar una variable hay dos partes. El código

```
int x = 5;
```

realmente tiene dos partes. Podemos pensarlo como

```
int x;
x = 5;
```

La primera línea le dice a la computadora que reserve un espacio de memoria para un número entero que y lo llame `x`. La segunda le asigna un valor determinado (en este caso cinco).

Podemos inicializar muchas variables al mismo valor. Por ejemplo

```
int x = y = z = 0;
```

También vimos cómo usar **arreglos**, que pueden pensar como un rectángulo de cajitas donde en cada cajita pueden guardar una variable del mismo tipo. Es decir, no pueden guardar `int`s y `double`s en el mismo arreglo, pero sí `int`s diferentes.

Recuerden que los arreglos se indexan desde cero. Así, un arreglo de tamaño 5 tiene casillas 0, 1, 2, 3 y 4. Para acceder a la posición número `i` del arreglo `a` usamos `a[i]`; y esta es una variable `int` como cualquier otra. Recuerden que la primera casilla del arreglo es `a[0]` y la última es `a[n-1]`, donde `n` es el número de casillas del arreglo. (Convención de programación: los arreglos se van a llamar `a`, su tamaño `n` y el número de casillas será `i`, `j` o `k`).

¿Por qué tres letras? Podemos tener arreglos de más de una dimensión. En particular, nos van a ser útiles los arreglos de dos dimensiones, que matemáticamente se llaman **matrices** y son el campo de estudio del álgebra lineal (nota cultural, las matemáticas más útiles que pueden aprender en la vida como ingeniero, programador, actuario o matemático usualmente están en el lenguaje del

álgebra lineal). En caso de una matriz, vamos a usar letras mayúsculas como A, B (en el código no lo hagan) y denotamos la entrada en la fila i, columna j como $A_{i,j}$ en matemático y como `a[i][j]` en Java.

Instanciar un arreglo también se hace en dos pasos:

```
int[] a;
a = new int[n];
```

La primera línea guarda el espacio de memoria, la segunda (énfasis en el `new`) declara que el arreglo va a ser tamaño `n`. Este tamaño *no* puede cambiarse después (en Java).

Condicionales

Muchas veces al programar nos preguntamos por varias opciones. La pregunta más fácil que podemos preguntar tiene como respuesta sí o no. Por ejemplo, ¿es par 28? Sí. ¿Es mayor que cero -35? No. Claro que también podemos preguntar cosas con más de dos respuestas. Por ejemplo, ¿cuál es el residuo al dividir siete entre tres? (Si se acuerdan de la clase, eso es preguntar con qué es congruente siete, módulo tres o cuánto es siete módulo tres).

Para preguntas de sí o no, la estructura que usamos es `if-else`. La idea es sencilla: hacemos una pregunta de sí o no y según la respuesta hacemos algo distinto. Por ejemplo, supongan que queremos ver si un número `x` es negativo. Si es negativo, queremos multiplicarlo por dos, y si no, multiplicarlo por tres.

```
if(x < 0){
  x = x*2;
} else{
  x = x*3;
}
```

El código entre las llaves del if corre sólo en el caso de que `x`sea negativo, en cuyo caso corre la segunda línea y se salta hasta el final de este bloque. Si el número es no-negativo, no corre el código entre las llaves del `if` y corre únicamente el que está entre las llaves del `else`.

Ojo, cuando usamos un sólo signo de igualdad estamos *asignando* un valor a una variable. Así, `x = 5` dice "ahora `x` vale cinco" mientras que `x == 5` pregunta "¿x vale cinco?" y podríamos, por ejemplo, guardar una variable booleana `b` como `b = x==5`.

También podemos anidar `if`s. Supongamos que queremos extender el caso de arriba. Si `x`es negativo, queremos hacer lo mismo con los casos negativo y positivo, pero si es cero queremos sumarle uno. En esta situación, podemos usar dos bloques `if` como sigue:

```
if(x < 0){
  x = x*2;
} else{
  if(x == 0){
    x = x+1;
  } else{
    x = x*3;
  }
}
```

Aquí quiero hacer énfasis en dos cosas. Primero chequen que no fue necesario preguntar si `x>0`porque ya descartamos las únicas otras opciones posibles. Podríamos haber puesto el if, pero siempre evaluaría a verdadero. También pongan atención en cómo se alínean las llaves. En realidad no son necesarias cuando sólo están corriendo una línea de código, pero es buena práctica ponerlas siempre.

Hay preguntas que naturalmente tienen muchas respuestas pero saben todas las posibles opciones. Por ejemplo, ¿cuánto sobra al dividir `x` entre cuatro? Como vimos en clase, las únicas respuestas posibles son 0, 1, 2, y 3. Podrían programarlo con muchos bloques `if` anidados, pero siempre que tengan una pregunta de este estilo Java tiene una estructura para hacerlo: `swich`. En el ejemplo de arriba, el código se vería así:

```
switch(x % 4){
  case 0:
    //aquí hacemos algo;
    break;
  case 1:
    // aquí hacemos algo diferente;
    break;
  case 2:
    // otra cosa
    break;
  case 3:
    // una última cosa
    break;
}
```

Este bloque revisa el valor de `x%4`(lo que sobra al dividir `x` entre 4, o `x` módulo cuatro en el lenguaje de la clase) y especificamos qué hacer en cada caso. Noten que de antemano tenemos que saber todos los casos posibles para usar esta estructura, pues explícitamente programamos cada uno. Si puede haber opciones desconocidas, podemos poner un caso default, luego les enseño cómo.

Ciclos

Hay situaciones en las que queremos repetir una misma acción varias veces. Por ejemplo, pueden querer leer todos los números de un arreglo (cuando quieren hacer algo sobre todos los elementos de un arreglo se llama *list comprehension*).

Hay dos tipos de ciclo que vimos: `while` y `for`. `while`es la forma más general de un ciclo, y como su nombre lo dice, se repite mientras una condición sea cierta. Supongamos por ejemplo que les doy un entero `x`y quieren dividirlo entre dos hasta que el resultado sea un número impar. Como vimos en clase, la condición de ser par es lo mismo que pedir congruencia con cero módulo dos (o sea, que al dividir entre dos el residuo sea cero). El código para hacer esto es así (suponiendo que x es dado)

```
while(x % 2 == 0){
  x = x/2;
}
```

El código entre las llaves del while se va a repetir hasta que `x`sea un número impar. Noten que la evaluación del `while` se hace antes de ejecutar el código, y si de inicio les doy un número impar, jamás se va a correr el código entre las llaves. Si quieren correrlo al menos una vez, la estructura se llama `do-while`; la vemos después si les interesa.

Hay veces en las que sabemos cuántas veces necesitamos ejecutar el código entre las llaves. Por ejemplo, supongan que tenemos un arreglo `a` de tamaño `n` y queremos imprimir cada elemento del arreglo. Necesitamos imprimir *algo* exactamente `n` veces. El código en Java es así:

```
for(int i = 0; i < n; i++){
  System.out.println(a[i])
}
```

Analicemos con cuidado la sintaxis del `for`. Lo primero que hacemos es declarar un entero `i=0`. Este iterador se va a ir recorriendo (por eso el `i++`) y se ejecuta lo que está dentro de las llaves siempre que se satisfaga la condición `i < n`. Entonces, el bloque de arriba es equivalente al siguiente:

```
int i = 0;
while(i < n){
  System.out.println(a[i]);
  i++;
}
```

Ejercicios (parte 1)

1. Descarguen Eclipse y Java siguiendo estas instrucciones

2. Escriban el código de Fibonacci que no guarda todo el arreglo como vimos en clase.

3. Para hacer una matriz en Java, la sintaxis es similar a la de un arreglo, pero necesitan dos pares de corchetes, uno para cada dimensión. Por ejemplo,

```
int[][] a;
a = new int[5][10];
```

hace una matriz de 5×10. El ejercicio es que inicialicen una matriz `a` de 4×3 con los números que gusten y guarden su *transpuesta*. Es decir, guarden una matriz `b` de 4×3 tal que cada i-ésima fila de `b` sea la i-ésima columna de `a`.

Ejercicios (parte 2)

3. Escriban código que ordene de menor a mayor un arreglo `a` de tamaño `n`. Pista: sólo necesitan el arreglo y una variable más.

4. Supongan que les doy un arreglo ordenado `a`. Escriban código que busque de la manera más eficiente posible un elemento dado en `a`. Por ejemplo, si `a = [1, 2, 3, 5]` y les doy `x=3`, su programa tiene que imprimir `2`, la posición de 3 en el arreglo.