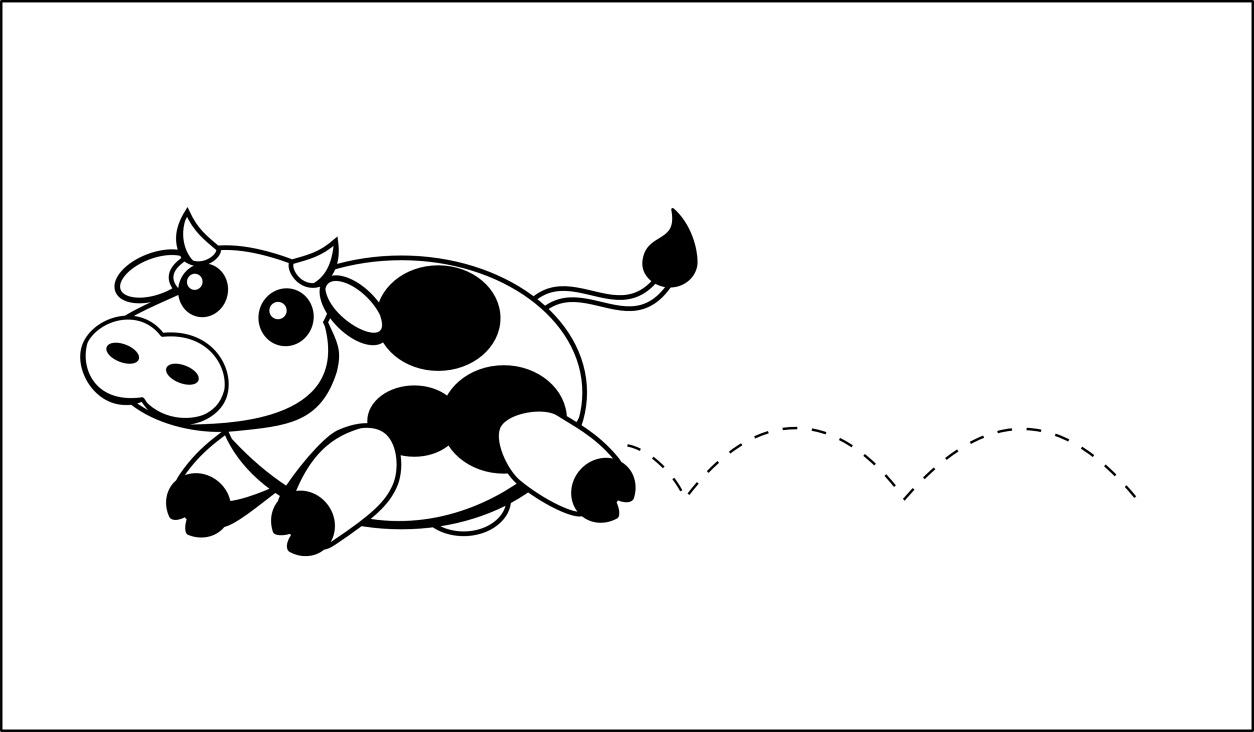
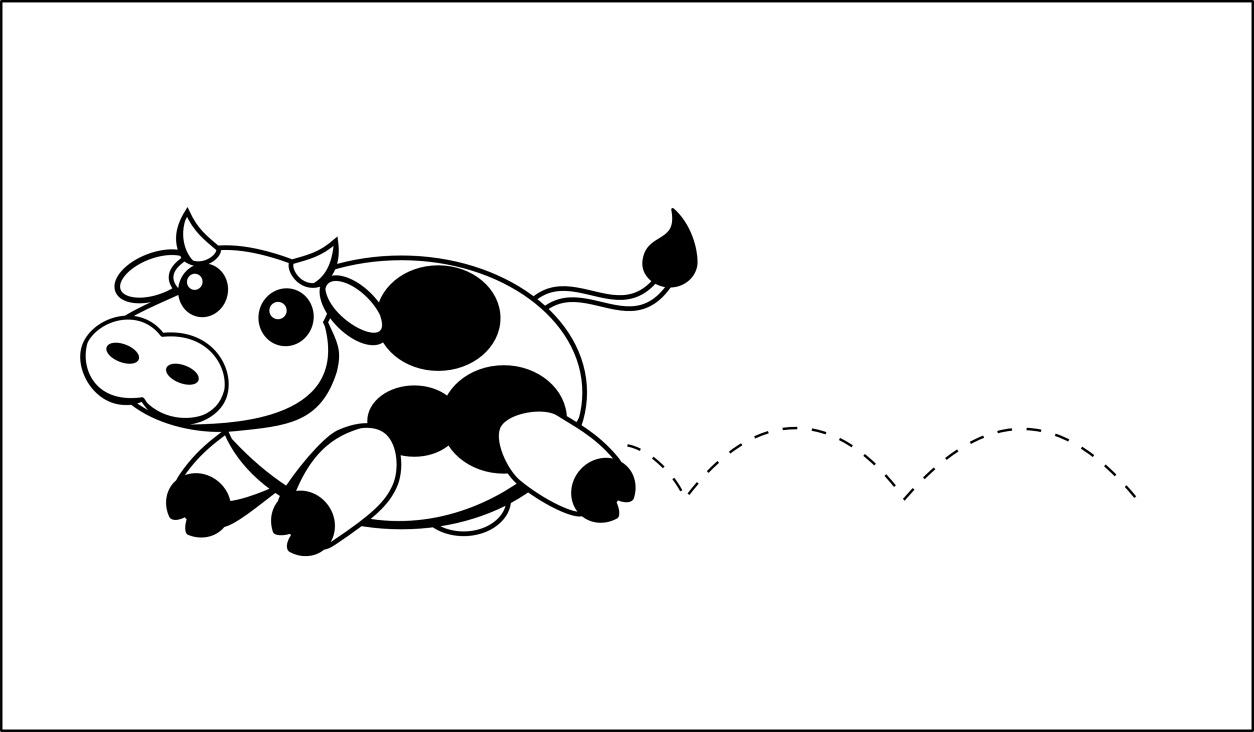
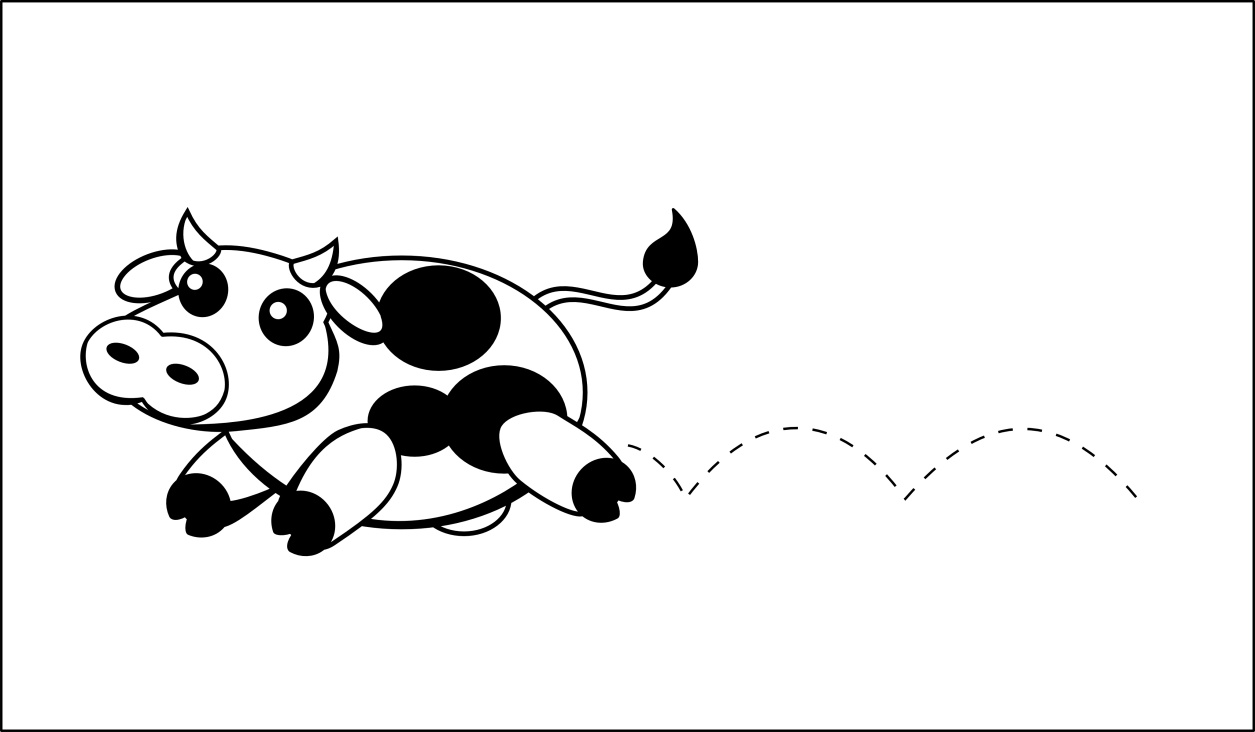
|  |
| --- |
| **Programación Orientada a Objetos con Java** |
| Para 2° año de EMT de Informática de UTU/CETP |
|  |
|  |
|  |
| **Luis Sebastián de los Angeles** |
|  |
|  |







# Programación Orientada a Objetos con Java

## Programando Objetos

La Programación Orientada a Objetos (POO) es un **paradigma de programación**, o sea, es una forma de entender y utilizar los lenguajes de programación y diseñar software. La POO surge como una respuesta a la necesidad de métodos y técnicas de programación más ágiles, más potentes y que permitieran describir mejor la realidad. El razonamiento detrás de la POO es que resulta mucho más natural programar si pensamos en los diferentes componentes de un programa como ***objetos que interactúan entre sí***.

De esta manera, obtenemos la definición de **Programa** en la POO, que es:

|  |  |
| --- | --- |
| vaca-habla-1.png | **DEFINICIÓN : Programa**  Conjunto de **OBJETOS** que **CUMPLEN DIVERSAS FUNCIONES** e **INTERACTÚAN ENTRE SÍ**, intercambiando **INFORMACIÓN.** |

**Esquema del funcionamiento de un programa diseñado mediante POO**

**Solicitud  
del  
Usuario**

**Respuesta  
del  
Programa**

**Programa**

De esta manera, para diseñar un programa lo que se hace es **diseñar y crear los objetos** que necesitamos para que cumplan las diversas tareas que componen el programa, y establecer las relaciones que debe tener entre sí.

¿Qué es un objeto?

En la POO, un objeto puede ser **cualquier cosa que yo le pueda describir a la computadora**. Puede basarse en un objeto de la realidad, o ser imaginario, o una abstracción.

|  |  |
| --- | --- |
| vaca-habla-1.png | **DEFINICIÓN : Objeto**  **Conjunto de recursos** (variables, algoritmos, funciones, etc.) que, juntos, **proporcionan una funcionalidad determinada dentro de un programa**. Un objeto puede representar una realidad física (los datos de una persona, por ejemplo) o un concepto completamente abstracto (como la conexión a una base de datos). |

Pero todos los objetos tienen una misma estructura: ***Clase,*** ***Nombre, Atributos y Métodos.***

La ***CLASE*** se puede definir como la **“familia” a la que pertenece un objeto**. Igual que una familia determina ciertas cualidades de una persona, la clase determina las características generales del objeto. De esta manera, todos **los objetos de una misma clase tienen las mismas características generales y las mismas capacidades**.

El ***NOMBRE*** es el **identificador de un objeto dentro del programa**. Un mismo programa puede, durante su ejecución, generar **varios objetos de la misma clase** (mediante un proceso llamado ***Creación de Instancias***, que veremos más adelante) pero para poder identificarlos y usar sus funciones, cada objeto debe tener **un nombre propio único** dentro del programa.

Los ***ATRIBUTOS*** son las **características del objeto**, lo que indica “cómo es” un objeto y en general, podemos decir que son ***variables*** tal y como las vimos antes. Algo importante a tener en cuenta es que, mientras que dos objetos de la misma clase tienen **el** **mismo conjunto de atributos**, **los valores** de esos atributos **pueden ser diferentes** (de la misma manera que dos hermanos pertenecen a la misma familia y comparten los mismos rasgos, pero de forma distinta).

Los ***MÉTODOS*** son las **capacidades del objeto**, indican “qué puede hacer” el objeto. Se trata de conjuntos de algoritmos o conjuntos de algoritmos que permiten que el objeto realice una tarea concreta. Es mediante la ejecución de estos métodos que el programa lleva a cabo sus funciones.

|  |  |
| --- | --- |
| vaca-habla-2.png | **CONCLUSIÓN:**   * **Todos los componentes** del programa **son objetos** que **se generan usando clases** . * Todo objeto del programa tiene **atributos y métodos** que se definen **en su clase**. * A los atributos **se les asigna valores** (semejante al uso de variables), los métodos **se ejecutan.** * El programa está formado por **objetos creados usando ese conjunto de clases**. |

***Ejemplo 1:***

Como se explicó más arriba, la POO se basa en el concepto de objeto que manejamos en el mundo real, y para este ejemplo usaremos **una vaca**. Primero **¿cómo está formada una vaca?**, podríamos entrar en muchos detalles, pero generalicemos lo más posible:

CUERNOS

COLA

MANCHAS

****

CABEZA

BOCA

OJOS

CUERPO

PATAS

UBRE

Lo que forma a nuestra vaca, sus características, serán sus **atributos**. Y podemos agruparlos de esta manera:

|  |
| --- |
| VACA |
| * Cabeza |
| * Cuernos |
| * Ojos |
| * Boca |
| * Cuerpo |
| * Manchas |
| * Ubre |
| * Cola |
| * Patas |

Ahora bien, una vaca también es capaz de **hacer** **cosas**. Por ejemplo:

|  |  |  |
| --- | --- | --- |
| vaca-muge.png | vaca-come.png | vaca-corre.png |
| **Mugir** | **Comer** | **Correr** |

Las cosas que nuestra vaca **puede hacer** son sus *habilidades o capacidades*, y las llamaremos **métodos.** Al igual que hicimos con los atributos, podemos agrupar los métodos en una tabla, de esta manera:

|  |
| --- |
| VACA |
| * Mugir() |
| * Comer() |
| * Correr() |

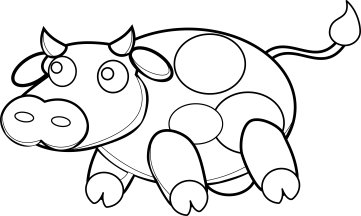
|  |  |
| --- | --- |
| vaca-habla-2.png | **IMPORTANTE:** Te habrás fijado que los métodos se escribieron con un paréntesis al final. Esto es algo importante a tener en cuenta, ya que los métodos **siempre** se deben anotar de esa manera, ya que **es la forma en que los diferenciamos** de los atributos.  Además, el paréntesis **tiene un propósito práctico: contener los datos de entrada** necesarios para la ejecución del método, un proceso que explicaremos en detalle más adelante en este libro cuando profundicemos en el desarrollo de métodos. |

¿Cómo se crea un objeto?

Para crear un objeto **primero debe existir su clase**. No podremos crear un objeto si su clase no ha sido definida aún.

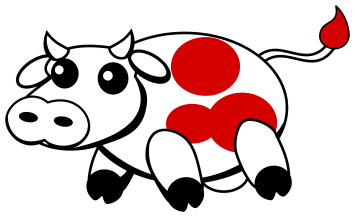
Las clases son los componentes básicos de todo proyecto de programación orientada a objetos. **Todo lo que existe y sucede dentro del programa, se define dentro de las clases, y se lleva a cabo mediante objetos que se generan a partir de esas clases.**

Podemos pensar en una clase como **en un molde**, con el cual **creamos nuevos objetos** que tendrán la “misma forma”, o sea las mismas características generales, pero que serán **objetos diferentes** durante la ejecución del programa.

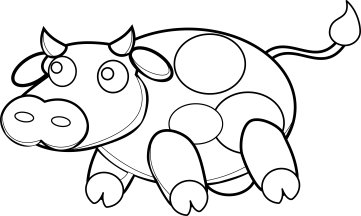
**Objeto 1**

**CLASE**



**Objeto 2**

Este proceso de creación de objetos se llama ***Creación de Instancias***o ***Instanciación*** , y cada objeto nuevo es una ***instancia*** de la clase a la que pertenece.

**Objeto  
(Instancia de Clase)**

**Pertenece** a una **clase** y recibe sus características, pero puede aplicarlas de forma particular

**Clase**

**Creación de Instancias**

**Genera** objetos que reciben sus características generales, pero **no suele ser utilizable por sí misma**.

|  |  |
| --- | --- |
| vaca-habla-1.png | **DEFINICIÓN : Instancia**  Es la creación de un **objeto** a partir de las características definidas en una **clase.** Los objetos sólo pueden crearse mediante instancias, y las instancias solo son posibles mediante clases.  Es mediante la creación de un objeto que la clase puede usarse, ya que en muy pocos casos es posible (o recomendable) el utilizar los métodos y/o atributos de una clase de forma directa.  En general: Una **clase es (casi) inútil sin objetos que la instancien**, y **es imposible crear un objeto mediante una instancia si este no pertenece a una clase.** |

¿Cómo se crea una clase?

La creación de una clase se llama ***declaración***. Al declarar una clase, se establece la descripción de los objetos que se crearán a partir de esa clase tal como lo hemos definido más arriba: mediante sus atributos y sus métodos.

Para la declaración de una clase, es importante que primero tengamos claro los atributos y los métodos que necesitaremos definir. Los diagramas que utilizamos en el ejemplo de más arriba nos servirán para eso, primero consideremos los datos que tenemos hasta el momento y vamos a plantearlos en un solo diagrama

|  |  |
| --- | --- |
| VACA | |
| Atributos | **Métodos** |
| * Cabeza | * **Mugir()** |
| * Cuernos | * **Comer()** |
| * Ojos | * **Correr()** |
| * Boca |  |
| * Cuerpo |  |
| * Manchas |  |
| * Ubre |  |
| * Cola |  |
| * Patas |  |

Para el desarrollo de clases en POO se utilizan diagramas muy similares a este mediante un lenguaje gráfico llamado **UML.** En este capítulo veremos **el primer paso** del diseño de clases en este lenguaje**: la creación de entidades**, pero para llegar a ese punto primero necesitamos **formalizar un poco** nuestro diagrama de atributos y métodos.   
En primer lugar formalicemos un poco nuestros atributos y nuestros métodos.

En este punto nos tenemos que preguntar acerca de la **visibilidad ó accesibilidad** de los atributos y métodos de la vaca.

|  |  |
| --- | --- |
| vaca-habla-1.png | **DEFINICIÓN : Visibilidad ó Accesibilidad**  Es una característica de los atributos y los métodos de un objeto que indica si el atributo o método puede ser modificado o utilizado por otro objeto. |

En principio, consideraremos dos valores de visibilidad: **público** y **privado**. La accesibilidad significa cosas diferentes para un atributo y para un método:

|  |  |
| --- | --- |
| vaca-habla-1.png | **Visibilidad o Accesibilidad**  **En un Atributo:**   * **Atributo Público:** *Cualquier objeto* puede acceder y modificar los valores del atributo. * **Atributo Privado:** *Solo el objeto propietario del atributo* puede acceder y modificar los valores del mismo.   **En un Método:**   * **Método Público:** El método puede ser ejecutado por *cualquier objeto* * **Método Privado:** El método solo puede ser ejecutado *por el objeto propietario del mismo* |

Para definir cuál característica del objeto será pública y cuál será privada, tenemos que considerar **la forma en que queremos que nuestro objeto interactúe** con los demás.

En general, para **los atributos**:

|  |  |
| --- | --- |
| vaca-habla-1.png | Un **atributo se definirá como público si:**   * Contiene o va a contener datos que deben ser **modificados por otros objetos**. * Sus datos pueden o deben **modificarse durante la ejecución** del programa.   De lo contrario, **el atributo será definido como privado**. |

Y para **los métodos**:

|  |  |
| --- | --- |
| vaca-habla-1.png | Un **método se definirá como público si:**   * El objeto lo va a necesitar para intercambiar información con otros objetos. * El resultado de su ejecución es necesaria para el funcionamiento de otros objetos.   De lo contrario, **el método será definido como privado**. |

Ahora solo debemos pensar cómo serán los atributos y los métodos de nuestra vaca. Como los atributos son sus características físicas, y estas no son fácilmente modificables, diremos que son atributos privados. Si fueran públicos, es como decir que podemos cambiar a voluntad las manchas de nuestra vaca, o cuántos ojos tiene.

En cuanto a los métodos, el razonamiento es parecido. El método **comer()** será privado (solo la vaca puede decidir cuándo comer). Pero los métodos **mugir()** y **correr()** pueden depender de estímulo externo (o sea, algo o alguien puede hacer que la vaca se ponga a correr o a mugir), así que los haremos públicos.

Para indicarlo en nuestro diagrama usaremos los símbolos "**+**" y "**-**" delante de cada característica:

|  |  |
| --- | --- |
| vaca-habla-1.png | **+** Característica 🡪 Indica que la característica es **pública**  **-** Característica 🡪 Indica que la característica es **privada** |

De esta manera nuestro diagrama, por ahora, se verá de esta manera:

|  |  |
| --- | --- |
| VACA | |
| Atributos | **Métodos** |
| - Cabeza | **+ Mugir()** |
| - Cuernos | **- Comer()** |
| - Ojos | **+ Correr()** |
| - Boca |  |
| - Cuerpo |  |
| - Manchas |  |
| - Ubre |  |
| - Cola |  |
| - Patas |  |

Un último elemento de UML que veremos ahora es la forma correcta en que una clase se describe en este lenguaje. En UML, una clase se describe indicando primero el nombre, luego los atributos y finalmente los métodos, con lo que el diagrama queda de esta manera:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| vaca-habla-1.png | **Representación de una Clase en UML:**   |  | | --- | | **NombreDeLaClase** | | **Atributos** | | **Métodos()** | |

Si aplicamos esta forma de definir objetos a la definición de nuestra vaca, obtendremos la clase **VACA** tal y como se vería en UML:

|  |
| --- |
| **VACA** |
| **- Cabeza**  **- Cuernos**  **- Ojos**  **- Boca**  **- Cuerpo**  **- Manchas**  **- Ubre**  **- Cola**  **- Patas** |
| **+ Mugir()**  **- Comer()**  **+ Correr()** |

Así que definimos nuestra primera clase y estamos en condiciones de empezar a crear el código de programación para poder utilizar esta clase en nuestro programa, pero para ello necesitamos conocer el lenguaje que usaremos: **Java**.  
En el próximo capítulo comenzaremos con los conceptos básicos de programación en Java y empezaremos a ver cómo transformar las clases que declaremos en código en un programa.

|  |  |
| --- | --- |
|  | **CONCLUSIÓN:**   * Un objeto necesita una clase para poder crearse. En esa clase se definen los **atributos** y los **métodos** que tendrá el objeto cuando sea creado. * La creación o definición de una clase se llama **declaración**, y la creación de un objeto a partir de una clase se llama **creación de instancias** o **instanciación**. * Todo objeto es **instancia** de alguna clase. * Los atributos y métodos de una clase tienen **visibilidad,** y pueden ser **públicos** o **privados.** |

## Introducción a la Programación con Java

En el capítulo anterior vimos algunos conceptos básicos sobre la POO. Antes de seguir profundizando en esos conceptos, veamos un poco sobre el lenguaje de programación con el que implementaremos nuestra POO en este curso: **Java.**

### ¿Qué es Java?

|  |  |
| --- | --- |
| Java logo.png | Java es un lenguaje de programación desarrollado en 1991 por James Gosling, Mike Sheridan y Patrick Naughton. Pasó por varios nombres, hasta que sus creadores lo nombraron en honor a una variedad de café (de ahí el logo de la taza). La primera versión pública fue liberada en 1995 por Sun Microsystems. Esta empresa llevó a cabo el desarrollo del lenguaje hasta 2010, cuando fue adquirida por Oracle.  Una de las principales características de Java es su **portabilidad:** un programa desarrollado en Java **no se ejecuta directamente** en el sistema operativo, sino que **es ejecutado por una máquina virtual** |
| (llamada “Máquina Virtual de Java”, **JVM,** por su sigla en inglés). De esta manera, un programa escrito en Java **puede ejecutarse en cualquier sistema operativo**, siempre y cuando la JVM esté instalada previamente en el mismo. Esta característica hace que sea un lenguaje muy usado en el mundo, manteniéndose en los primeros lugares del índice TIOBE[[1]](#footnote-1), y siendo el lenguaje de programación de aplicaciones para el sistema operativo **Android** de Google. | |

### ¿Cómo se Programa en Java?

Bueno, de eso se trata este libro, pero, para empezar, concentrémonos en la sintaxis de algunas **estructuras básicas.**

|  |  |
| --- | --- |
| vaca-habla-1.png | **Estructuras Básicas:**  Cuando en este libro hablamos de estructuras básicas, nos referimos a las **estructuras de datos y de control que resultan imprescindibles para programar.**  Estos son:   * Las **variables**, y las sentencias que permitan acceder a ellas y modificarlas. * La **expresión condicional** (el “**SI lógico”** ó **IF** como lo llamaremos de ahora en adelante) * El **bucle** **FOR** * El **bucle** **WHILE** |

Pero sin importar el objeto o estructura que estemos definiendo, **toda sentencia debe finalizar en punto y coma (;).**La ausencia del punto y coma al finalizar una sentencia es uno de los errores más comunes (y más frustrantes) que encuentra el programador que trabajo con Java.

|  |  |
| --- | --- |
| vaca-habla-2.png | **Finalizado de sentencias:**  ***Toda sentencia debe finalizarse con un punto y coma (;)*** ya que esto es lo que le indica al compilador de Java ***que la sentencia terminó*** ***y que puede evaluar la próxima línea del programa***, de hecho, puedo incluso incluir saltos de renglón en medio de una sentencia sin que esto afecte el resultado final, siempre y cuando use un punto y coma al final.  De esta manera la sintaxis de una sentencia cualquiera en Java sería:  **comando modificador1 modificador2;**  O también:  **comando**  **modificador1**  **modificador2;** |
|  |  |

#### Definir Variables en Java

Para definir variables, debemos tener en cuenta que una de las principales características de Java es que es un lenguaje de programación **fuertemente tipeado.** Esto significa ***que las variables y todos los objetos*** que definamos **deben declararse especificando su tipo de datos,** o de lo contrario el programa no podrá ejecutarse.

De esta manera, cuando debo declarar una variable en Java sigo estos pasos:

1. **Establezco su tipo de datos.**
2. **Establezco su nombre.**
3. **Establezco su valor inicial** (también llamado “inicializar la variable”)**.**

Esto nos deja la siguiente sintaxis general:

|  |  |
| --- | --- |
| vaca-habla-1.png | **Sintaxis para definir variables:**  **tipoDeDatos nombreDeVariable = valor;** |

Conviene acotar que esta forma de definir una variable en Java es lo que llamaremos **declaración directa** o **declaración con inicialización.** Más adelante veremos otras formas de declarar variables que pueden resultar más convenientes en algunos casos específicos.

Java es un lenguaje con muchos tipos de datos, pero en general usaremos unos pocos tipos básicos que cubrirán la mayoría de nuestras necesidades iniciales:

|  |  |
| --- | --- |
| vaca-habla-2.png | NOTA: Java es un lenguaje **muy estricto** con las minúsculas y las mayúsculas, y deben respetarse para evitar errores.  Esto significa que “string”, “String” y “STRING” son tres expresiones **totalmente** diferentes para el compilador de Java. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| vaca-habla-1.png | **Tipos de Datos básicos en Java y Ejemplos de declaración:**   |  |  | | --- | --- | | **Tipo de Dato** | **Descripción y Ejemplo** | | **int** | Para valores numéricos positivos o negativos enteros (sin decimales) entre -2.147.483.648 y 2.147.483.647 | | **int cantidad = 0;** | | | **long** | Igual que **int** pero para valores mayores que los soportados por **int** | | **long cantidad = 0;** | | | **float** | Para valores numéricos positivos o negativos con decimales | | **float resultado = 0.0;** | | | **double** | Igual que **float,** pero para valores decimales demasiado elevados o demasiado pequeños. | | **char** | Permite almacenar **un solo** carácter: una letra, número, espacio o símbolo.  Un valor tipo *char* **siempre** debe ir entre comillas simples. | | **char letra = 'a';** | | | **String** | Para cadenas de texto de uso general.  Un valor de tipo *String* **siempre** debe escribirse entre comillas. | | **String texto = "texto cualquiera";** | | | **boolean** | Permite almacenar solo dos posibles valores: **true** y **false** | | **boolean pruebaL = true;** | | |

Ahora que tenemos una sintaxis general para definir variables, y los tipos de datos necesarios para definirlas, podemos plantear cómo se **establecen, acceden, y modifican** los datos almacenados en una variable en Java.

Para todos los ejemplos que siguen a continuación, usaremos una variable de tipo *String* llamada “texto” y un par de variables de tipo *int* llamadas “num1” y “num2”.   
Esto implica que, en alguna parte del código, antes de los ejemplos, existen las siguientes sentencias, declarando las variables que usaremos:

|  |  |
| --- | --- |
| vaca-codigo.png | **int num1 = 0;**  **int num2 = 0;**  **String texto = "";** |

Por supuesto, esta aclaración se debe a que **es imposible trabajar con una variable que no ha sido declarada** o el compilador informará de un error**.**

|  |  |
| --- | --- |
| vaca-habla-1.png | **DEFINICIÓN: 1 – Estableciendo el valor de una variable ya existente**  **NombreDeVariable = valor;  texto = "nuevo valor de la variable";**  Cuando la variable ya existe**, no es necesario** indicar su tipo, simplemente se utiliza el nombre que le dimos, y mediante el símbolo “ **=** ” le asignamos su nuevo valor. |

O sea que basta con **igualar la variable a un valor compatible** para que la misma almacene dicho valor.

Para el siguiente ejemplo utilizaremos un **método,** que detallaremos con mayor precisión más adelante. Este método es **println()** y viene predefinido en el lenguaje. Permite mostrar en pantalla una línea de texto (que debe insertarse dentro de los paréntesis) y resulta útil para acceder al contenido de nuestra variable.

|  |  |
| --- | --- |
| vaca-habla-1.png | **DEFINICIÓN: 2.1 – Accediendo al valor de una variable ya existente (mediante un método)**  **método(NombreDeVariable);** |

Para mostrar en pantalla el contenido de la variable ***texto***, usaremos la siguiente línea de código:

|  |  |
| --- | --- |
| vaca-codigo.png | **System.out.println(texto);** |

Como podemos observar, simplemente **incluimos el nombre de la variable donde necesitemos los datos almacenados en ella**, y Java los reemplazará de forma automática.

Ahora bien, no siempre voy a utilizar mis variables por si solas o con un método.

En otros casos, **puedo utilizar los datos de una variable, o la combinación (mediante operaciones) de los datos de varias variables para establecer el valor de otra**.

Para explicarlo planteemos el siguiente caso: estableceremos el valor de la variable “num1” que definimos antes, luego usaremos a “num1” para establecer el valor de la variable “num2”, y crearemos una tercer variable de tipo *int*, llamada “total”, donde almacenaremos el valor de la suma de “num1” y “num2”.

El código resultante es el siguiente:

|  |  |
| --- | --- |
| vaca-codigo.png | **num1 = 2;**  **num2 = num1;**  **int total = num1 + num2;** |

En este ejemplo, la variable “num1” almacena un **2**. La variable “num2” esta igualada al valor de “num1”, va a almacenar el mismo valor que almacene la variable “num1”, así que también es un **2**. Finalmente, la variable “total” se define estableciendo su valor inicial como la suma de los valores de “num1” y “num2”, por lo que “total” almacena el resultado de 2 +2, o sea **4**.

|  |  |
| --- | --- |
| vaca-habla-1.png | **DEFINICIÓN: 2.2 – Accediendo al valor de una variable ya existente y almacenando el mismo en otra variable:**  **variable1 = variable2** |

En general, estas son las formas de manejar el contenido de las variables que definamos en Java. En caso de que nos encontremos con casos especiales, nos ocuparemos de ellos en el momento que los necesitemos.

### Estructuras de Control en Java

Las estructuras de control son el componente esencial de todo lenguaje de programación, ya que, como su nombre lo indica, son estructuras que permiten **controlar el flujo** de la ejecución del programa.

Antes de comenzar con las estructuras de control en sí, conviene notar que la declaración de las mismas varía un poco con relación a la de otras sentencias.

Las estructuras de control **no utilizan punto y coma (;)** **al finalizar**, en su lugar **utilizan llaves** (**"{"** y **"}"**) que establecen dónde empieza y dónde termina su contenido. A este espacio lo llamaremos **contexto** **de la estructura de control.**

|  |  |
| --- | --- |
| vaca-habla-1.png | **DEFINICIÓN: 3 – Sintaxis General de las Estructuras de Control**  **estructura(parámetros){**  **sentencias;**  **}** |

Otra particularidad es que las variables que declaremos ***dentro*** **del contexto** de una estructura de control, sólo existen **dentro de ese contexto**. Cuando termine la ejecución de la estructura de control, todas las variables declaradas dentro de la misma y su contenido **serán eliminadas** de la memoria. **Tampoco es posible acceder a las mismas de forma directa** desde afuera del contexto de la estructura de control, pero es posible recuperar la información de las mismas, como veremos más adelante.

#### La Condicional (IF/IF-ELSE)

La estructura de control más básica de todas es ***la condicional***, a la cual nos referiremos, de ahora en adelante, como ***if***.

Como repaso, recordemos que ***if*** es una estructura que permite que el programa decida **si ejecuta o no** **un conjunto de acciones, o decida** **entre *dos conjuntos* de acciones posibles basándose en el *valor de una condición***, la cual se evalúa al ejecutarse la estructura.

Esto implica que el ***if*** puede declararse de **dos formas diferentes**, que llamaremos **declaración simple** y **declaración completa**.

En cualquiera de los dos casos, el ***if*** evalúa el valor de **una condición** que por lo general tiene una de las siguientes formas:

* **Una comparación lógica entre valores** o **entre el contenido** **de dos variables**.
* **Una variable** de tipo **boolean*.***
* **El resultado de la ejecución de un método** que devuelve un valor de tipo **boolean*.***

La **declaración simple** se utiliza cuando queremos que el programa decida **si debe ejecutar o no** un conjunto determinado de sentencias. Si la condición resulta **verdadera**, las sentencias dentro del ***if*** se ejecutan, de lo contrario, el programa las ignora y no son ejecutadas.

|  |  |
| --- | --- |
| vaca-habla-1.png | **DEFINICIÓN: 4.1 – Declaración Simple del IF**  if( condiciónAEvaluar ){  //sentencias del bloque  } |

**Ejemplo 1**: la expresión dentro de las llaves del ***if*** se ejecutará (mostrará en pantalla “**El valor es 2**”), porque la condición expresada (“**num1 == 2**”) chequea si es verdadero o falso que la variable “**num1**“ almacena el valor “2”. En este caso la condición es verdadera, o sea, tiene el valor **true**.

|  |  |
| --- | --- |
| vaca-codigo.png | int num1 = 2;  if(num1 == 2){  System.out.println("El valor es 2");  } |

**Ejemplo 2:** la expresión dentro de las llaves del ***if*** **no se ejecutará** (no se mostrará en pantalla “**El valor es 3**”), porque la condición expresada (“**num1 == 3**”) chequea si es verdadero o falso que la variable “**num1**“ almacena el valor “3”. En este caso la condición es falsa, o sea, tiene el valor **false**.

|  |  |
| --- | --- |
| vaca-codigo.png | int num1 = 2;  if(num1 == 3){  System.out.println("El valor es 3");  } |

Ahora bien, si las condicionales utilizan comparaciones entre valores y variables, ¿cómo se realizan esas comparaciones? Para realizar comparaciones utilizaremos **operadores de comparación** y **operadores lógicos**, los cuales nos permiten expresar estas comparaciones.

Ya vimos, en los ejemplos anteriores, uno de estos comparadores: el **comparador de igualdad** ("**==**"). En la siguiente tala se detallan los otros comparadores existentes en el lenguaje:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| vaca-habla-1.png | **Operadores de Comparación:**  En todos los casos, es posible comparar entre sí el valor de variables o valores “sueltos” en cualquier combinación: variables con variables, variables con valores, o valores con valores.   |  |  | | --- | --- | | **Operador** | **Descripción** | | **var1 == var2**  **var1 == *valor*** | **Igualdad**  Devuelve **True** si los elementos comparados son **iguales.** | | **var1 != var2**  **var1 != *valor*** | **Desigualdad**  Devuelve **True** si los elementos comparados son **distintos.** | | **var1 > var2**  **var1 > *valor*** | **Mayor Que**  Devuelve **True** si el primero de los elementos comparados es **mayor que el otro.** | | **var1 < var2**  **var1 < *valor*** | **Menor Que**  Devuelve **True** si el primero de los elementos comparados es **menor que el otro.** | | **var1 >= var2**  **var1 >= *valor*** | **Mayor o Igual Que**  Devuelve **True** si el primero de los elementos comparados es **mayor o es igual que el otro.** | | **var1 <= var2**  **var1 <= *valor*** | **Menor o Igual Que**  Devuelve **True** si el primero de los elementos comparados es **menor o es igual que el otro.** |   **Operadores Lógicos:**  Permiten combinar o transformar comparaciones lógicas. Salvo la negación, que trabaja sobre una sola expresión, el resto Permite combinar dos expresiones o variables de tipo **boolean** en una sola.   |  |  | | --- | --- | | **Operador** | **Descripción** | | **!(comp1)** | **Negación Lógica**  Niega el valor de la expresión o variable de tipo **boolean** ubicada entre los paréntesis.  Equivalente a la negación de lógica proposicional. | | **comp1 && comp2** | **Conjunción Lógica**  Devuelve **true** **solo si** **las dos** tienen valor **true** al mismo tiempo.  Equivalente al operador “Y” de lógica proposicional. | | **comp1 || comp2** | **Disyunción Lógica Simple**  Devuelve **true** si **al menos una** tiene valor **true**.  Equivalente al operador “O” de lógica proposicional. | | **comp1 ^ comp2** | **Disyunción Lógica Exclusiva**  Devuelve **true** si **las dos expresiones** tienen valor lógico **opuesto entre sí**.  Equivalente al operador “XOR” de lógica proposicional. | |

Otra forma de trabajar con ***if*** es con la **declaración completa**. Esta forma permite que el algoritmo elija **un camino entre dos caminos posibles**, dependiendo del valor de la **condición** establecida. Esto se logra añadiendo un nuevo bloque de llaves encabezado por la palabra clave ***else.***

|  |  |
| --- | --- |
| vaca-habla-1.png | **DEFINICIÓN: 4.2 – Declaración Completa del IF**  if( condiciónAEvaluar ){  //sentencias\_si\_verdadero;  }  else {  //sentencias\_si\_falso;  } |

Si el chequeo de la condición resulta verdadero (**true**), se ejecuta el primer bloque de sentencias. Si resulta falso (**false**), se ejecuta el ***segundo boque de sentencias***, el que está dentro del contexto de **else**.

**Ejemplo 3**:

|  |  |  |
| --- | --- | --- |
| vaca-codigo.png | **Ejemplo: funcionamiento de un bloque IF**  int num1 = 2;  if(num1 == 3){  System.out.println("El valor es 3");  }  else{  System.out.println("El valor es 2");  }  **El resultado de este código al ejecutarse sería:**   |  | | --- | | **El valor es 2** | |

La expresión del primer bloque ***no se ejecutará*** (es decir, no se mostrará en pantalla el texto “**El valor es 3**”), porque la condición expresada (“**num1 == 3**”) chequea si es verdadero o falso que la variable “**num1**” almacena el valor “3”. En este caso la condición **es falsa** (o sea, tiene el valor **false**). Por este motivo, solo se ejecutará la expresión dentro del bloque **else**.

|  |  |
| --- | --- |
| vaca-habla-2.png | **Nota sobre la comparación de valores de tipo *String:***  Los comparadores **"==" y "!="** **no permiten** **comparar valores de tipo *String***, o sea, comparar si dos cadenas de texto son idénticas entre ellas.  En su lugar usaremos el método **.equals()**  **Ej:**  Supongamos dos variables de tipo String (“texto1” y “texto2”) y una cadena de texto cualquiera, la comparación entre ellas se realiza de la siguiente manera:  texto1.equals(texto2);  Otra manera, si quiero comparar texto1 a un texto cualquiera, sería:  texto1.equals("texto cualquiera"); |
|  |  |

#### El Bucle FOR

Otra forma de controlar el flujo de un algoritmo es mediante **bucles:** estructuras de control que ejecutan **de forma repetitiva** una serie de sentencias, mientras **se cumpla una condición**.

En el bucle FOR, esta condición toma la forma de una expresión que contiene un **contador** con un **tope** y una **regla de avance**. Cada vez que el código contenido en el bucle se ejecuta en su totalidad, ese contador **se modifica** de acuerdo a la regla de avance y se lo compara con el valor del tope. Si el contador **no iguala** el valor tope, el bucle **se ejecuta una vez más**. Por el contrario, si el contador **iguala al tope,** el bucle **termina su ejecución**.

En código, esto se expresa de esta manera:

|  |  |
| --- | --- |
| vaca-habla-1.png | **DEFINICIÓN: 4.3 – Declaración del Bucle FOR**  for(int contador = 0; contador < 10; contador++){  //sentencias del bucle  } |

Los componentes del bucle se declaran **entre los paréntesis** luego de la palabra clave ***for, separados por comas,*** y en este ejemplo son los siguientes:

**int contador = 0 🡪** Es la variableque cumple rol del **contador** del bucle. Como es natural en un contador, esta variable es de tipo **int,** ya que debe manejar ***solamente*** **valores numéricos enteros.** En este caso, la variable **contador** se inicia en 0, pero puede iniciarse en otro valor, según haga falta.

**contador < 10 🡪** Esta expresión cumple el rol de **tope** del bucle. Como puede observarse, es una comparación lógica, igual a las utilizadas en la estructura IF y que, por lo tanto, devuelve un valor de tipo **boolean.** Para que el bucle continúe su ejecución, esta comparación tiene que resultar en **true** (en este caso, para que eso suceda el valor de la variable **contador** debe ser menor a 10).

**contador++ 🡪** Esta expresión establece la **regla de avance** del bucle. En este caso, indica que tras cada ejecución completa del bucle, el valor de la variable **contador** debe aumentar en 1. También se pueden utilizar como reglas de avance expresiones como estas:

* **contador--** Indica que la variable **contador** debe disminuir en 1.
* **contador = contador + 2** Indica que la variable **contador** debe aumentar en 2

(NOTA: puede usarse **cualquier otro número**).

* **contador = contador - 2** Indica que la variable **contador** debe disminuir en 2

(NOTA: puede usarse **cualquier otro número**).

|  |  |  |
| --- | --- | --- |
| vaca-codigo.png | **Ejemplo: Funcionamiento de un Bucle FOR**  **for(int contador=0; contador<10; contador++){**  **System.out.println("El contador vale:" + contador);**  **}**    **Resultado:**   |  | | --- | | **El contador vale:0**  **El contador vale:1**  **El contador vale:2**  **El contador vale:3**  **El contador vale:4**  **El contador vale:5**  **El contador vale:6**  **El contador vale:7**  **El contador vale:8**  **El contador vale:9** | |

En la primera ejecución del bucle, se define la variable interna contador, como ya vimos, y se establece su valor en 0. Por supuesto, **0 es menor que 10**, por lo tanto, **el bucle puede continuar su ejecución**.

Así que se ejecuta **el código dentro del bloque del bucle**. En este caso, indica que se debe mostrar en pantalla el texto "El contador vale:", seguido del valor de la variable contador en ese momento, que, como ya dijimos, es 0. Por lo tanto, nos muestra en pantalla "El contador vale:0".

Como no hay más código dentro del bloque del bucle, pasamos al último comando del bucle: contador++, que le suma 1 al valor actual de la variable, o sea 0 + 1, por lo que contador ahora contiene el valor 1.

Este proceso se repite hasta que contador alcance el valor de 9. En ese punto, al ejecutarse el contenido del bloque y pasar al comando contador++, esta variable alcanza el valor de 10. Este valor rompe la condición, ya que **10 no es estrictamente menor que 10**. Por lo tanto, no se ejecuta el contenido del bloque y se sale del bucle.

Existe otra forma del bucle for, especialmente adaptada para trabajar con estructuras de datos multidimensionales como son las listas, pero la veremos más adelante, cuando estemos trabajando con ese tipo de estructuras.

#### **El Bucle WHILE**

A diferencia del bucle for, este bucle no establece un contador con un tope de ejecuciones, sino que chequea activamente una condición lógica como las usadas por el if. Esto implica que, dependiendo del valor de esa condición, el bucle while puede **no ejecutarse para nada,** o **ejecutarse indefinidamente**. Esta particularidad es muy útil en algunos casos, pero puede devenir en un ***bucle infinito*** si no tenemos cuidado.

El bucle while es considerablemente más sencillo de declarar que el bucle for:

|  |  |
| --- | --- |
| vaca-habla-1.png | **DEFINICIÓN: 4.3 – Declaración del Bucle WHILE**  while( condición ){  //sentencias del bucle  } |

Como se puede apreciar, su sintaxis es muy semejante a la que utiliza el if. En este caso, la condición no solo controla el acceso a la estructura, sino que también controla la ejecución de una nueva iteración[[2]](#footnote-2) del bucle. Mientras la condición establecida entre paréntesis devuelva un resultado True, el bucle va a continuar ejecutándose. Esto implica que el código no va a avanzar más allá del bucle mientras esto suceda. Por este motivo, el bloque de sentencias del while debe incluir alguna forma de control sobre el estado de esa condición, para evitar que el mismo quede sin control y caiga en un bucle infinito.

|  |  |
| --- | --- |
| vaca-habla-2.png | **Bucles Infinitos:**  Son bucles en los que no existe control sobre el valor de la condición que regula el funcionamiento del mismo, por lo que el bucle se ejecuta de forma repetitiva. Esto impide el avance de la ejecución del código, paraliza la aplicación, y consume recursos de cómputo en la computadora por encima de lo normal. |

Veamos un ejemplo:

|  |  |  |
| --- | --- | --- |
| vaca-codigo.png | **Ejemplo: Funcionamiento de un Bucle WHILE**  int contador=0;  **while( contador < 10 ){**  **System.out.println("El contador vale:" + contador);**  **contador++;**  **}**  **Resultado:**   |  | | --- | | **El contador vale:0**  **El contador vale:1**  **El contador vale:2**  **El contador vale:3**  **El contador vale:4**  **El contador vale:5**  **El contador vale:6**  **El contador vale:7**  **El contador vale:8**  **El contador vale:9** | |

Este ejemplo se diseñó para que entregue un resultado idéntico al que vimos con el bucle for. Nótese que definimos, necesariamente, la variable contador por fuera del bucle. La condición que establecimos en el bucle en este caso es idéntica que la usada para establecer el tope del for.

El funcionamiento sería de esta manera: al entrar en el bucle, se chequea el valor de la variable contador. Como el valor 0 es menor que 10, el bucle puede iniciarse y se ejecutan las sentencias del mismo: se muestra el mensaje en pantalla y se suma 1 al valor de la variable contador.

Este proceso se repite hasta que contador llega al valor 9. En esa iteración, luego de mostrar el mensaje, contador va alcanzar el valor 10. Eso significa que al chequear nuevamente la condición, esta va a devolver un valor False como ya vimos antes, y el bucle se cierra sin ejecutar su contenido.

1. TIOBE, una compañía dedicada a la evaluación de la calidad del software, mantiene un índice donde mide la popularidad de los diferentes lenguajes de programación basándose en la cantidad de búsquedas realizadas por los usuarios en Internet. Puede consultarse en <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html> [↑](#footnote-ref-1)
2. Iteración: una secuencia completa en una serie repetitiva de acciones, o sea, en este caso es una “vuelta” del bucle. [↑](#footnote-ref-2)