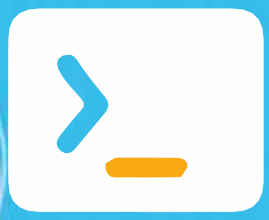


Argentina programa

Programación Orientada a Objetos con JAVA

Guía I –Parte 2

Introducción a JAVA

VALORES POR DEFECTO

En Java las variables atributos asumen valores por defecto si no son inicializadas. A grandes rasgos el valor por defecto será cero o null dependiendo del tipo de dato. La siguiente tabla resume los valores por defecto dependiendo del tipo de dato.

byte	0
short	0
int	0
long	0
float	0.0
double	0.0
boolean	false
char	'\u0000'
Objetos	null

Las variables locales son ligeramente diferentes; el compilador no asigna un valor predeterminado a una variable local no inicializada. Las variables locales son aquellas que se declaran dentro de un método. Si una variable local no se inicializa al momento de declararla, se debe asignar un valor antes de intentar usarla. El acceso a una variable local no inicializada dará lugar a un error en tiempo de compilación.

OPERADORES

Los operadores son símbolos especiales de la plataforma que permiten especificar operaciones en uno, dos o tres operandos y retornar un resultado. También aprenderemos qué operadores poseen mayor orden de precedencia. Los operadores con mayor orden de precedencia se evalúan siempre primero.

Primeramente, proceden los operadores unarios, luego los aritméticos, después los de bits, posteriormente los relacionales, detrás vienen los booleanos y por último el operador de asignación. La regla de precedencia establece que los operadores de mayor nivel se ejecuten primero. Cuando dos operadores poseen el mismo nivel de prioridad los mismos se evalúan de izquierda a derecha.

OPERADOR DE ASIGNACIÓN	
=	Operador de Asignación Simple
OPERADORES ARITMÉTICOS	
+	Operador de Suma
-	Operador de Resta
*	Operador de Multiplicación
/	Operador de División
%	Operador de Módulo
OPERADORES UNARIOS	
+	Operador Unario de Suma; indica que el valor es positivo.
-	Operador Unario de Resta; indica que el valor es negativo.
++	Operador de Incremento.
--	Operador de Decremento.
OPERADORES DE IGUALDAD Y RELACIÓN	
==	Igual
!=	Distinto
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
OPERADORES CONDICIONALES	
&&	AND
||	OR
!	Operador Lógico de Negación.

ESTRUCTURAS DE CONTROL

Las estructuras de control son construcciones hechas a partir de palabras reservadas del lenguaje que permiten modificar el flujo de ejecución de un programa. De este modo, pueden crearse construcciones de alternativas mediante sentencias condicionales y bucles de repetición de bloques de instrucciones. Hay que señalar que un bloque de instrucciones se encontrará encerrado mediante llaves {……..} si existe más de una instrucción.

ESTRUCTURAS CONDICIONALES

Los condicionales son estructuras de control que cambian el flujo de ejecución de un programa de acuerdo a si se cumple o no una condición. Cuando el flujo de control del programa llega al condicional, el programa evalúa la condición y determina el camino a seguir. Existen dos tipos de estructuras condicionales, las estructuras if / else y la estructura switch.

IF/ELSE

La estructura if es la más básica de las estructuras de control de flujo. Esta estructura le indica al programa que ejecute cierta parte del código sólo si la condición evaluada es verdadera («true»). La forma más simple de esta estructura es la siguiente:

```
if(<condición>){
  <sentencias>
}
```

En donde, <condición> es una expresión condicional cuyo resultado luego de la evaluación es un dato booleano(lógico) verdadero o falso. El bloque de instrucciones <sentencias> se ejecuta si, y sólo si, la expresión (que debe ser lógica) se evalúa a true, es decir, se cumple la condición.

Luego, en caso de que la condición no se cumpla y se quiera ejecutar otro bloque de código, otra forma de expresar esta estructura es la siguiente:

```
if(<condición>){
<sentencias A>
} else {
   <sentencias B>
}
```

El flujo de control del programa funciona de la misma manera, cuando se ejecuta la estructura if, se evalúa la expresión condicional, si el resultado de la condición es verdadero se ejecutan las sentencias que se encuentran contenidas dentro del bloque de código if (<sentencias A>). Contrariamente, se ejecutan las sentencias contenidas dentro del bloque else (<sentencias B>).

En muchas ocasiones, se anidan estructuras alternativas if-else, de forma que se pregunte por una condición si anteriormente no se ha cumplido otra y así sucesivamente.

```
if (<condicion1>) {
<sentencias A>
} else if(<condicion2>){
<sentencias B>
} else {
<sentencias C>
}
```

Al contrario de la estructura if / else, la estructura switch permite cualquier cantidad de rutas de ejecución posibles. Un switch funciona con los datos primitivos byte, short, char e int. También funciona con Enumeraciones, tema que se verá más adelante en el curso, y con unas cuantas clases especiales que «envuelven» a ciertos tipos primitivos: Character, Byte, Short, e Integer (tema que trataremos cuando se profundice en Orientación a Objetos).

SWITCH

El bloque switch evalúa qué valor tiene la variable, y de acuerdo al valor que posee ejecuta las sentencias del bloque case correspondiente, es decir, del bloque case que cumpla con el valor de la variable que se está evaluando dentro del switch.

```
switch(<variable>) {
case <valor1>:
<sentencias1> break;
case <valor2>:
<sentencias2> break;
default:
<sentencias3>
}
```

El uso de la sentencia break que va detrás de cada case termina la sentencia switch que la envuelve, es decir que el control de flujo del programa continúa con la primera sentencia que se encuentra a continuación del cierre del bloque switch. Si el programa comprueba que se cumple el primer valor (valor1) se ejecutará el bloque de instrucciones <sentencias1>, pero si no se encuentra inmediatamente la sentencia break también se ejecutarían las instrucciones <sentencias2>, y así sucesivamente hasta encontrarse con la palabra reservada break o llegar al final de la estructura.

Las instrucciones dentro del bloque default se ejecutan cuando la variable que se está evaluando no coincide con ninguno de los valores case. Esta sentencia equivale al else de la estructura if-else.

Nota: pueden encontrar un ejemplo de estructuras condicionales en Moodle.

ESTRUCTURAS REPETITIVAS

Durante el proceso de creación de programas, es muy común, encontrarse con que una operación o conjunto de operaciones deben repetirse muchas veces. Para ello es importante conocer las estructuras de algoritmos que permiten repetir una o varias acciones, un número determinado de veces.

Las estructuras que repiten una secuencia de instrucciones un número determinado de veces se denominan bucles, y se denomina iteración al hecho de repetir la ejecución de una secuencia de acciones.

Todo bucle tiene que llevar asociada una condición, que es la que va a determinar cuándo se repite el bucle y cuando deja de repetirse.

WHILE

La estructura while ejecuta un bloque de instrucciones mientras se cumple una condición. La condición se comprueba antes de empezar a ejecutar por primera vez el bucle, por lo tanto, si la condición se evalúa a «false» en la primera iteración, entonces el bloque de instrucciones no se ejecutará ninguna vez.

```
while (<condición>) {
<sentencias>
}
```

DO / WHILE

En este tipo de bucle, el bloque instrucciones se ejecuta siempre al menos una vez. El bloque de instrucciones se ejecutará mientras la condición se evalúe a «true». Por lo tanto, entre las instrucciones que se repiten deberá existir alguna que, en algún momento, haga que la condición se evalúe a «false», de lo contrario el bucle será infinito.

```
do {
<sentencias>
} while (<condición>);
```

La diferencia entre do-while y while es que do-while evalúa su condición al final del bloque en lugar de hacerlo al inicio. Por lo tanto, el bloque de sentencia después del "do" siempre se ejecutan al menos una vez.

FOR

La estructura for proporciona una forma compacta de recorrer un rango de valores cuando la cantidad de veces que se deber iterar un bloque de código es conocida. La forma general de la estructura for se puede expresar del siguiente modo:

```
for (<inicialización>; <terminación>; <incremento>) {
<sentencias>
}
```

La expresión <inicialización> inicializa el bucle y se ejecuta una sola vez al iniciar el bucle. El bucle termina cuando al evaluar la expresión <terminación> el resultado que se obtiene es false. La expresión <incremento> se invoca después de cada iteración que realiza el bucle; esta expresión incrementa o decrementa un valor hasta que se cumpla la condición de <terminación> del bucle.

La estructura for también ha sido mejorada para iterar de manera más compacta las colecciones y los arreglos, tema que se verá más adelante en este curso. Esta versión mejorada se conoce como *for-each*.

Como regla general puede decirse que se utilizará el bucle for cuando se conozca de antemano el número exacto de veces que ha de repetirse un determinado bloque de instrucciones. Se utilizará el bucle do-while cuando no se conoce exactamente el número de veces que se ejecutará el bucle, pero se sabe que por lo menos se ha de ejecutar una.

Se utilizará el bucle while cuando es posible que no deba ejecutarse ninguna vez.

Nota: pueden encontrar un ejemplo de estructuras repetitivas en Moodle.

SENTENCIAS DE SALTO

En Java existen dos formas de realizar un salto incondicional en el flujo "normal" de un programa. A saber, las instrucciones break y continue.

BREAK

La instrucción break sirve para abandonar una estructura de control, tanto de las condicionales (*if-else y switch*) como de las repetitivas (*for, do-while y while*). En el momento que se ejecuta la instrucción break, el control del programa sale de la estructura en la que se encuentra contenida y continua con el programa.

CONTINUE

La sentencia *continue* corta la iteración en donde se encuentra el *continue*, pero en lugar de salir del bucle, continúa con la siguiente iteración. La instrucción *continue* transfiere el control del programa desde la instrucción *continue* directamente a la cabecera del bucle (*for, do-while o while*) donde se encuentra.

Como un ejemplo vale más que mil palabras nos pondremos manos a la obra.

Supongamos que nos piden implementar para el videojuego "Carrera Mortal" un Auto con las siguientes características y comportamientos para ser anexado al proyecto:

Sus características (atributos) son: color, patente y combustible con una carga inicial de 50lts.

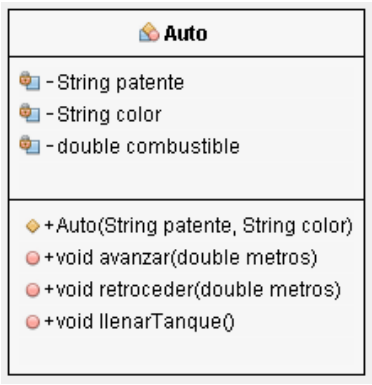
El auto sólo tendrá como comportamiento:

• Avanzar: Al que le indicaremos la cantidad de metros que deseamos avance y deberemos tener en cuenta que por cada 10 mts consume 1lt de combustible y solo podrá avanzar si hay combustible suficiente.

• Retroceder: Al que le indicaremos la cantidad de metros que deseamos retroceda y deberemos tener en cuenta que por cada 10 mts consume 1lt de combustible y solo podrá retroceder si hay combustible suficiente.

• LlenarTanque: Llenará el tanque de este auto nuevamente con 50lts de combustible.

Si representamos con diagrama de clases UML a nuestro Auto, tendríamos lo siguiente:

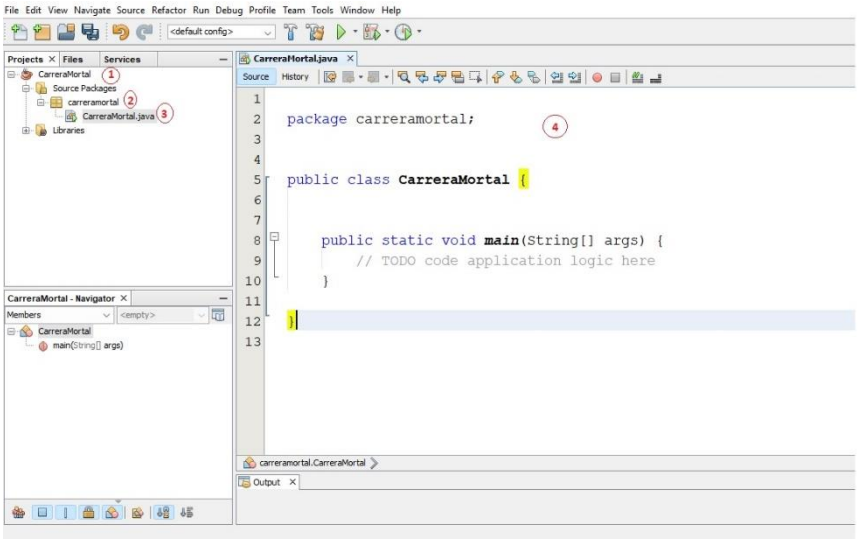


Como podemos observar, las características de Auto (sus atributos), no son ni más ni menos que variables de un tipo determinado que almacenaran un estado; y su comportamiento (métodos) no son otra cosa que en otros lenguajes conocemos como procedimientos o funciones (si devuelven algo); y es allí en donde utilizaremos las estructuras de programación aprendidas en el curso anterior como iteraciones, condicionales o simples estructuras secuenciales.

En la figura, también se observa que la clase Auto también posee un constructor a través del cual podremos crear instancias (objetos) de tipo Auto inicializando sus atributos patente y color.

Una vez analizado el problema y habiendo modelado que es lo queremos hacer, ahora procederemos a implementar en código java como lo vamos a hacer.

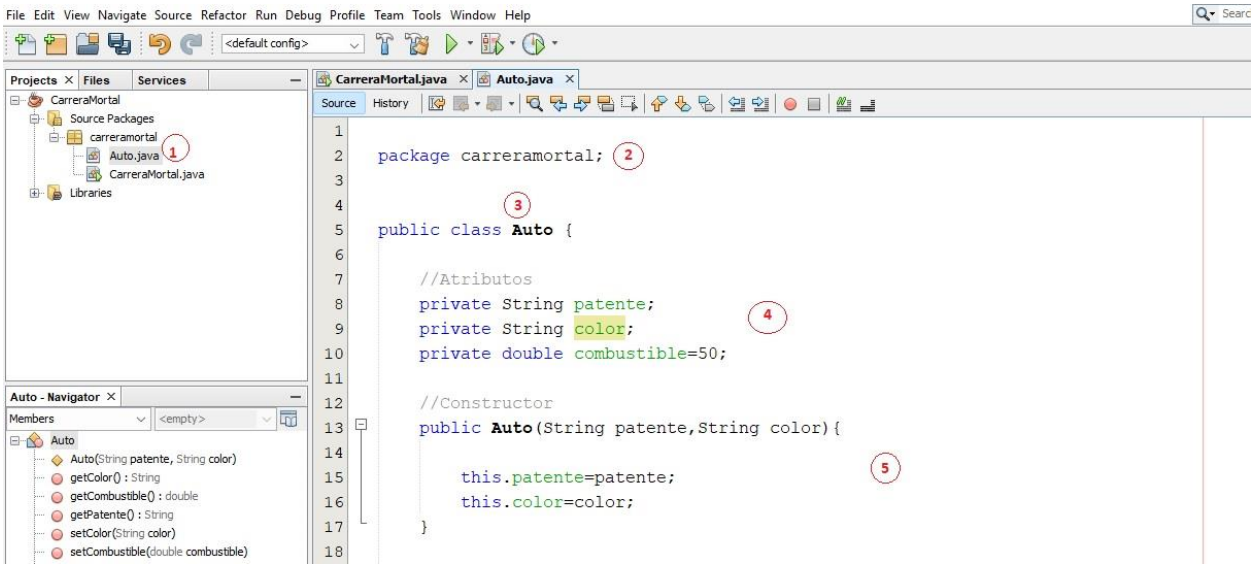
Para ello, en nuestro IDE (Entorno de Desarrollo Integrado) Netbeans crearemos un nuevo proyecto con el nombre por ejemplo “CarreraMortal”



Como se puede observar, Netbeans representa en el panel izquierdo a nuestro proyecto con una taza de café (1) y dentro de este creó automáticamente con el mismo un paquete (2) y una clase (3) dentro de la cual está el método main ya explicado al principio de este documento, dentro del cual haremos las pruebas.

En el panel derecho pueden observar la estructura del archivo java que contienen clase "CarreraMortal" (4).

A continuación dentro del paquete "carreraMortal" (2) crearemos nuestra plantilla (clase) Auto.



En esta figura podemos observar que dentro del paquete "carreraMortal" se ha agregado un archivo java "Auto.java" (1) y en el panel derecho observamos el contenido de dicho archivo, es decir nuestra clase Auto. Allí en esta primer vista se alcanzan a visualizar en que paquete se encuentra la clase (2), la definición de la misma (3) y la declaración de las variables atributos (4), además del constructor encargado no solo de permitirnos crear instancias de esta clase sino además de poder inicializar sus variables atributos: patente y color.-

Pero, como pueden ver, nuestros atributos tienen el modificador de acceso "private" y eso los hace inaccesibles desde el exterior, por ello crearemos los métodos públicos que nos permitirían desde afuera conocer el estado de esos atributos y poder modificarlos (métodos getter y setter).

Un método "get" tiene como función devolver al exterior el estado de un atributo, es decir, se comporta como una función; y el nombre comienza generalmente con la palabra get seguido del nombre del atributo cuyo estado va a devolver. Por ejemplo el "get" del atributo patente sería así:

```
public String getPatente() {
    return patente;
}
```

Y por otro lado, un método “set” tiene por tarea permitir que el estado de un atributo sea modificado desde el exterior; es decir, recibe un nuevo dato que reemplazará el estado actual de un atributo; no devolviendo nada al exterior, comportándose como un procedimiento. Por ejemplo el “set” del atributo patente sería así:

```
public void setPatente(String patente) {
    this.patente = patente;
}
```

Como se habrán dado cuenta, los modificadores de acceso de estos atributos son “public” es decir, pueden ser vistos desde el exterior. Ahora la pregunta es: Porque debo hacer esto…? Ocultar el estado de los atributos y tener métodos públicos para poder acceder a ellos?, bien a este concepto se lo conoce como “encapsulamiento”.

Ahora, una vez definidos los miembros principales de la clase (atributos, constructor, métodos getter y setters) podemos proceder a dar implementación a los métodos avanzar, retroceder y llenarTanque.

Con respecto al método *avanzar*, según el requerimiento, recibe los metros que va a avanzar el auto; como precondición tenemos que el auto solo avanzará si hay combustible disponible en su tanque ya que por cada 10 metros consumirá 1 litro de combustible.

Por lo tanto nuestro método podría comportarse como un procedimiento, es decir no devolver nada y recibir por parámetro, la cantidad de metros que va a avanzar; siendo este un dato numérico de tipo real. Siendo su declaración la siguiente:

```
public void avanzar (double metros) { }
```

y dentro del cuerpo del método podríamos por ejemplo analizar con esos metros que recibe como parámetro cuando combustible consumirá.

```
double consume=metros/10;
```

Luego en base a esa información verificaríamos si el tanque del auto tiene combustible suficiente, podríamos indicar a través de un mensaje por consola que el auto pude avanzar y restar lo consumido al combustible, caso contrario no puede avanzar porque no hay combustible suficiente. Es decir, aquí utilizaríamos una estructura condicional (if..else)

```
if(this.combustible >= consume) {
        System.out.println("Avanzando: "+metros +"metros");
        this.combustible=this.combustible-consume;
}else {
        System.out.println("No hay suficiente combustible");
}
```

Con respecto al método *retroceder*, su lógica sería similar a la de avanzar.

Por último, el método *llenarTanque*, tendrá como responsabilidad volver a asignar a la variable atributo combustible el valor 50. Es decir, este método también será un procedimiento (void), pero no necesita recibir ningún dato del exterior; ya que literalmente debe asignar a combustible valor 50.

```
public void llenarTanque() {

        this.combustible=50;
}
```

A continuación el código completo de la clase Auto.

C:/Users/Usuario/Documents/NetBeansProjects/CarreraMortal/src/carreramortal/Auto.java

```
 1 package carreramortal;
 2
 3 public class Auto {
 4
 5     //Atributos
 6     private String patente;
 7     private String color;
 8     private double combustible=50;
 9
10     //Constructor
11     public Auto(String patente,String color){
12
13         this.patente=patente;
14         this.color=color;
15     }
16
17     //Getters and Setters
18
19     public String getPatente() {
20         return patente;
21     }
22
23     public void setPatente(String patente) {
24         this.patente = patente;
25     }
26
27     public String getColor() {
28         return color;
29     }
30
31     public void setColor(String color) {
32         this.color = color;
33     }
34
35     public double getCombustible() {
36         return combustible;
37     }
38
39     public void setCombustible(double combustible) {
40         this.combustible = combustible;
41     }
42
43     public void avanzar(double metros){
44
```

C:/Users/Usuario/Documents/NetBeansProjects/CarreraMortal/src/carreramortal/Auto.java

```
45          double consume=metros/10;
46          if(this.combustible>=consume){
47
48              System.out.println("Avanzando: "+metros +"metros");
49              this.combustible-=consume;
50          }else{
51
52              System.out.println("No hay suficiente combustible");
53          }
54      }
55
56       public void retroceder(double metros){
57
58          double consume=metros/10;
59          if(this.combustible>=consume){
60
61              System.out.println("Retrocediendo: "+metros +"metros");
62              this.combustible-=consume;
63          }else{
64
65              System.out.println("No hay suficiente combustible");
66          }
67      }
68
69   public void llenarTanque(){
70
71        this.combustible=50;
72   }
73
74 }
```

Ahora desde el método main de la clase “CarreraMortal”, crearemos por ejemplo un Auto de color “azul” y patente “AA999BB” y lo haremos avanzar 20 metros y retroceder 50; al finalizar visualizaremos el estado del combustible por consola.

Es importante entender que al crear una clase Auto, disponemos de un nuevo tipo de dato, y por lo tanto podremos declarar variables de tipo Auto; por lo tanto para resolver lo que debemos hacer en el main, lo primero que haremos será declarar una variable de tipo Auto y guardar allí una instancia del mismo con los datos propuestos.

Auto miAuto=new Auto(“AA999BB”,”azul”);

En este caso, el identificador que utilizamos para la variable de tipo auto es “miAuto” y para crear una instancia del mismo utilizamos la palabra clave “new” seguida del nombre del constructor de Auto que lo preparamos para recibir primero la patente y luego el color.

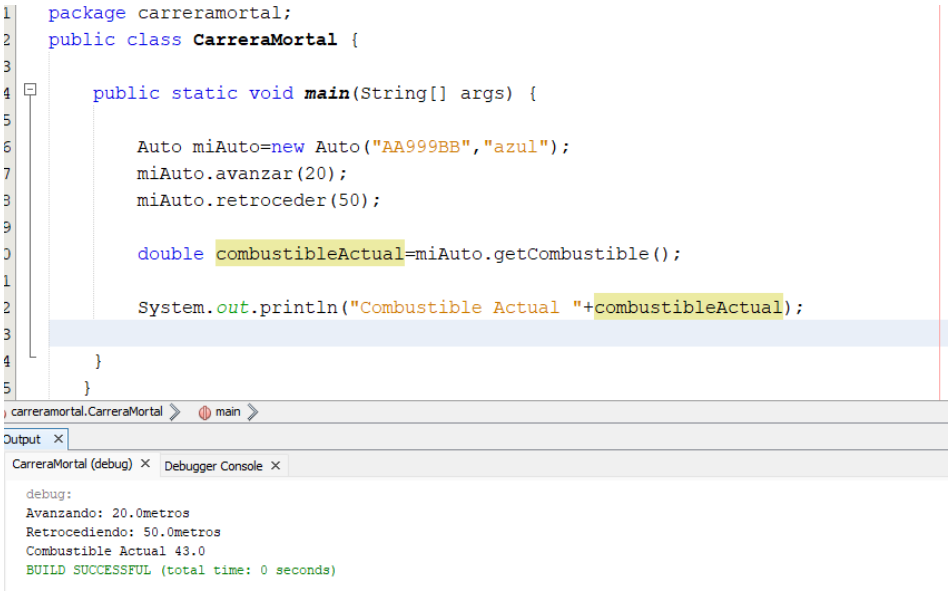
A partir de allí, a través de esa variable de referencia “miAuto” podremos acceder a sus métodos; entonces por si lo queremos hacer avanzar 20 metros, bastará que escribamos la siguiente línea de código:

```
miAuto.avanzar(20);
```

Si queremos mostrar por consola el estado del combustible del auto en ese momento:

```
System.out.println(miAuto.getCombustible());
```

Se muestra a continuación como podría ser la prueba desde el main y la salida por consola obtenida al ejecutar el código:



BIBLIOGRAFIA:

Christopher Alexander, “A Pattern Language”, 1978

Erich Gamma, “Design Patterns: Elements of Reusable OO Software”

Martin Fowler, “Analysis Patterns: Reusable Object Models”, Addison Wesley, 1997