ISBN 978-9942-8734-4-6

PROGRAMACION ORIENTADA A OBJETOS PARA PRINCIPIANTES

Autores:
Ing. Esparza Cruz, Nelly Karina

DATOS DEL AUTOR

Ing. Sist. Nelly Karina Esparza Cruz, MIE

Ingeniera en Sistemas de la Universidad Técnica de Babahoyo

Magister en Administración de Empresas de la Universidad Técnica de Babahoyo

Magister en Informática Empresarial de UNIANDES

Catedrática de la Universidad Técnica de Babahoyo desde el 2004. Siempre quiso ser Ingeniera en Sistemas, su principal desarrollo profesional ha sido en el campo de la programación. Desarrollador Independiente de Software, creadora del Software Integrado para Microempresas (SIM), sistema de facturación e inventario que se encuentra funcionando en muchos negocios a nivel nacional.

Dedicatoria

A Dios, por darme una vida llena de bendiciones.

A mis padres, mi hermano, mi esposo y mis princesas, son la razón de mí existir.

A mis amigos por ser siempre esa luz brillante que alegra la vida.

INTRODUCCION

El presente libro está inspirado en entregar a los estudiantes de programación una obra con conceptos claros y sencillos de la programación orientada a objetos, el contenido se estructura en 4 capítulos:

Capítulo 1. Fundamentos del paradigma orientado a objetos y lenguaje Java. Este capítulo introduce al lector a conocer los pilares de la POO(Programación Orientada a Objetos) y del lenguaje Java, también se familiarizará con los conceptos de clases y objetos.

Capítulo 2. Herencia y clases abstractas. El lector estará apto para crear clases basadas en otras clases existentes, se impulsa la reutilización del código y la creación de jerarquías de clases que sirvan para la creación de otras clases más especializadas.

Capítulo 3. Polimorfismo e Interfaces de Clases. En este capítulo se refuerzan los conceptos de herencia y se incorporan nuevos conceptos orientados a potenciar el funcionamiento de las clases hijas usando polimorfismo e interfaces.

Capítulo 4. Manejo de excepciones. El lector adquirirá técnicas de programación para crear soluciones informáticas robustas con un adecuado manejo de errores en tiempo de ejecución

En cada tema se expone la teoría necesaria para entender los ejemplos incluidos en el presente libro.

1. CAPITULO: Fundamentos del Paradigma Orientado a Objetos y Lenguaje Java .6
1.1. INTRODUCCIÓN 7
1.1.1. PARADIGMA ORIENTADO A OBJETOS 8
1.1.2. PILARES DE LA PROGRAMACIÓN ORIENTADA A OBJETOS ... 9
1.2. LENGUAJE DE PROGRAMACIÓN JAVA 13
1.2.1. VARIABLES Y TIPO DE DATOS 13
1.2.2. SENTENCIAS DE CONTROL 16
1.2.3. MÉTODOS Y CLASES 32
1.3. CLASES Y OBJETOS 34
1.3.3. MÉTODOS 36
1.3.4. PROPIEDADES 39
2. CAPÍTULO: Herencia y Clases Abstractas 48
2.1. HERENCIA 49
2.2.1. MÉTODOS VIRTUALES 54
2.2. HERENCIA EN JAVA 54
2.2.1. CLASE PADRE O SUPERCLASE 54
2.2.2. CLASE HIJA O SUBCLASE 55
3. CAPÍTULO: Polimorfismo e Interfaces de clases 60
3.1. CONCEPTO DE POLIMORFISMO 61
3.1.1. DIFERENCIA ENTRE CLASES CONCRETAS Y ABSTRACTAS .64
3.1.2. ANÁLISIS DE MÉTODOS ABSTRACTOS E INTERFACES 65
4. CAPITULO : Manejo de Excepciones 70
4.1. INTRODUCCIÓN 71
4.1.1. DEFINICIÓN DE BLOQUE TRY, CATCH Y FINALLY 72
4.2. MANEJO DE EXCEPCIONES 73

1. CAPÍTULO: Fundamentos del Paradigma Orientado a Objetos y Lenguaje Java

1.1. INTRODUCCIÓN

Los lenguajes de programación han variado a través de la historia y se han adaptado a las soluciones requeridas en cada etapa del tiempo, según Llavori(2000) la evolución histórica es posible realizar el siguiente resumen:

- Primera generación : lenguaje de máquina
- Segunda generación : lenguajes simbólicos o ensambladores
- Tercera generación : primeros lenguajes de alto nivel en donde el programa ya no depende del hardware en los que funcionan
- Cuarta generación : el programador ya se enfoca en indicar "qué" debe hacer el programa y no "cómo"

La programación orientada a objetos es una extensión natural de la actual tecnología de programación, apareció aproximadamente por los años 60 en lenguajes Smalltalk, Simula, Ada este introduce características que aumentan la probabilidad de que un sistema se implemente y se mantenga con efectividad. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones y las estructuras de datos que esos procedimientos manejan. La programación modular, en cambio, trata de descomponer el problema en módulos con el fin de hacerlo más legible y manejable.

La programación orientada a objetos es un distanciamiento de la programación estructurada o procedural, en donde se pone énfasis en la estructura de los datos y en el encapsulamiento de los procedimientos que operan sobre estos datos, los principales objetivos de la POO son uniformidad, portabilidad, reusabilidad, mantenibilidad y flexibilidad.

Según Fernández (2012), el elemento principal de la POO son las clases, a través de las cuales se logra definir cómo funciona un determinado tipo de objeto. Luego de esto, podemos crear objetos de esa clase.

Estos objetos son entidades que contienen un determinado "estado", "comportamiento (método)" e "identidad".

Un objeto mantiene la información para que este le permita definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase, al poder tener valores diferenciados en sus atributos.

Los métodos definen el comportamiento de la clase, y los atributos el estado del objeto.

1.1.1. PARADIGMA ORIENTADO A OBJETOS

Los paradigmas de programación han evolucionado a través del tiempo, cuando se habla de paradigmas se refiere a formas de pensar, un enfoque particular propuesto por un autor o una comunidad de programadores para la solución a un problema y en el caso de la programación se refiere al estilo de la programación, sin embargo, es frecuente que los lenguajes de programación implementen varios paradigmas. Fundamentalmente es posible dividir a los paradigmas en: imperativos "el cómo se debe calcular" y los declarativos "el qué se debe calcular". El enfoque declarativo tiene algunas formas: el paradigma funcional, el lógico y los lenguajes descriptivos, mientras que el enfoque imperativo puede ser expresado en forma modular, procedimental y estructurado.

Sin embargo también es posible dividir a los paradigmas en cómo se encuentran organizados los programas, y aquí podemos encontrar programación estructurada, orientada a objetos, orientada a eventos, programación genérica. A continuación se muestra una tabla con los lenguajes de programación que se destacaron en la historia, se muestra que paradigma usó cada uno de ellos.

Lenguaje de programación	***Paradigma***
Fortran	Imperativo, procedural, no estructurado
Pascal	Imperativo, procedural, estructurado
Prolog	Funcional y lógicos
Smalltalk	Orientado a objetos
Delphi	Imperativo – orientado a objetos
VB	Imperativo – orientado a objetos
PHP, Perl, Ruby, Javascript	Scripting

Tabla 1. Paradigmas de lenguajes de programación

1.1.1. PILARES DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

La programación orientada a objetos propone una forma distinta de programar que incorpora nuevas concepciones y que apareció con el objetivo de encontrar solución a los problemas detectados en la programación estructurada o modular, entre ellos, según Goytia(2014) destacan los siguientes:

- ✓ **Clase:** describe a un grupo de objetos que comparten atributos y comportamientos comunes. También se puede decir que una clase es el molde, una plantilla qué define un tipo de variable o datos que el programador crea para un objetivo específico.

- ✓ **Objeto:** es la instancia de una clase, un ejemplar de una clase, el cual tiene datos y todos aquellos procedimientos que pueden manipular aquellos datos definidos en la clase; el acceso a los datos de un objeto se debe realizar únicamente a través de estos procedimientos. Un objeto se identifica por su nombre, posee estados, tiene un conjunto de métodos y un conjunto de atributos asociados a él, soportan encapsulamiento.

- ✓ **Método:** son las acciones que puede realizar un objeto, los mecanismos mediante los cuales las clases acceden a los datos, los métodos se ejecutan cuando son llamados o invocados por un objeto. Cada método tiene un nombre y puede tener parámetros de entrada, se ejecutan cuando el objeto recibe un mensaje enviado por un objeto. El método más importante es el constructor, el cual se activa en el momento que se instancia el objeto con la palabra new.

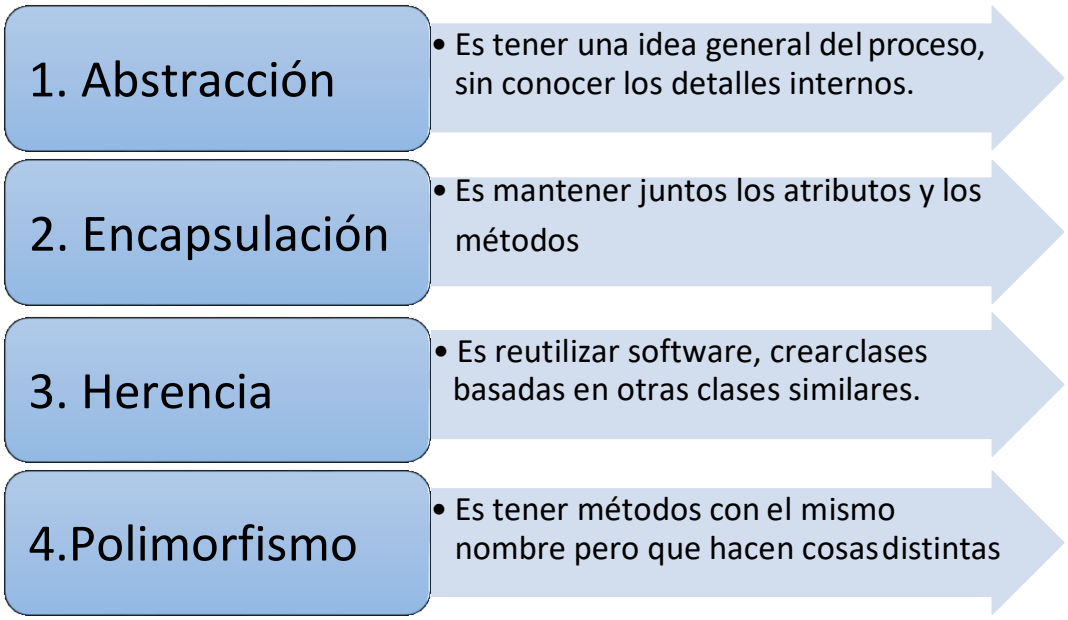
- ✓ **Propiedad o Atributo:** son valores o características de los objetos, cuyo valor puede alterarse al momento de ejecutar algún método. Mediante los atributos se define información oculta dentro de un objeto, la cual es solamente manipulada por los métodos definidos sobre dicho objeto.

- ✓ **Estado Interno:** es una variable declarada privada, la cual únicamente puede ser accedida y alterada por un método del objeto, se lo utiliza para indicar distintas situaciones posibles para el objeto.

- ✓ **Evento:** es un suceso que el sistema envía para que el objeto realice algo según corresponda al evento suscitado.

- ✓ **Mensaje:** es una comunicación dirigida a un objeto, la cual este le ordena para que se ejecuten uno de sus métodos con ciertos parámetros asociados al evento que lo generó.

La programación orientada a objetos está basada en cuatro pilares fundamentales:



1. Abstracción

La abstracción es un principio para ignorar aspectos que no son relevantes para poder concentrarse en aquellos que lo son, según Moreno (2015) de manera general es posible observar este principio en:

a. Generalización.- Se aplica cuando dos clases tienen características comunes y se crea una clase padre que pueda agrupar a todas las que comparten estas características. Por ejemplo: tenemos clases vendedor y clases cliente que comparten los atributos de datos personales, en este caso podemos crear una clase persona que sea el padre de las clases antes mencionadas.
b. Agregación .- Esta se produce cuando una clase contiene a otra, es posible determinar que existe una relación de agregación cuando es posible decir que la CLASE_A es parte de la CLASE_B. Por ejemplo la clase Vehículo, puede tener como atributos a otras clases como la clase MOTOR, RUEDAS, etc.
c. Especialización.- Cuando es posible crear una clase que es un caso especial de otra. Por ejemplo la clase empleado y la clase empleado_contratado.

Para comprender la abstracción, pensemos en el siguiente ejemplo:

Un sensor de temperatura:

- Mide la temperatura
- Muestra su valor

No sabemos:

- Cómo hace para medir la temperatura
- Cuáles son los materiales de su fabricación
- Cuánto tiempo dura

Es común que en la programación orientada a objetos se hable de generalización, agregación y especialización, sin embargo, cada uno es diferente del otro.

2. Encapsulación

El encapsulamiento oculta detalles de la implementación de un objeto, complementa a la abstracción. Esto permite la inclusión de una sola entidad de información y operaciones que mantienen controlada la clase, está compuesto por:

- Componentes públicos (Accesibles, Visibles).- Pueden ser accesibles fuera de la clase.
- Componentes privados (No accesibles, Ocultos).- Estos componentes sólo están disponibles dentro de la implementación de la clase.
- Restricción de accesos indebidos.- También denominados protegidos, son accesibles dentro de la clase y para las clases heredadas.

Los objetos encapsulan lo que hacen. Ocultan la funcionalidad interna de sus operaciones, de otros objetos y del mundo, en clase se distingue el "qué hacer" del "cómo hacer".

Un ejemplo de encapsulamiento es cuando presionamos el botón "encender" de un computador, no necesitamos saber cómo lo hace, entonces se ocultan los detalles del "cómo" solamente vemos la opción Encender.

3. Herencia

Según Regino(2015), consiste en crear clases a partir de otras de las cuales pueden heredar sus atributos (datos) y comportamientos (métodos), es decir permite re-utilizar código previamente desarrollado. La clase que hereda es llamada superclase o clase padre y la clase heredada se denomina subclase o clase hija.

La herencia proporciona las siguientes ventajas:

- Las clases derivadas o subclases proporcionan comportamientos de la cual pueden heredar de la clase base, es decir, los programadores van a reusar códigos previamente desarrollados y probados en la creación de aplicaciones más grandes.

- También se crean superclases abstractas, que definen comportamientos genéricos, que son implementados parcialmente en las clases padres para que su implementación total se dé en las clases hijos definiendo comportamientos en las subclases especializadas. El propósito de una clase abstracta es servir de modelo base para la creación de otras clases derivadas, pero cuya implementación depende de las características

particulares de cada una de ellas. La clase base sirve como guía para la definición de las subclases.

4. Polimorfismo

Es la capacidad que tienen las clases por medio de la herencia, de tener comportamientos diferentes para un mismo método, son métodos con nombres comunes pero implementados de forma diferente dependiendo de la funcionalidad que tendrán en cada clase.

Por ejemplo, el área de un cuadrado, rectángulo y círculo, son calculados de forma distinta usando diferentes formas pero con datos similares; sin embargo, en sus clases respectivas se puede realizar la implementación del área bajo el nombre común de método Área y dependiendo del objeto que invoque al método este ejecutará el que le corresponde.

Ejemplos,
Superclase: Clase Animal → Subclases: Perro, Tiburón

Se puede definir un método morder en cada subclase, cuya implementación cambia de acuerdo a la clase invocada, sin embargo el nombre del método es el mismo.

perro.morder ≠ tiburón.morder ≠ Animal.morder

1.2. LENGUAJE DE PROGRAMACIÓN JAVA

1.2.1. VARIABLES Y TIPO DE DATOS

DEFINICIÓN DE VARIABLES

Según Llinas(2010), una variable almacena datos de un determinado tipo y a la cual se la diferencia por nombres, este dato puede cambiar en el transcurso del programa. Por tanto, para definir una variable sólo se puede decir cuál será el nombre que se le dará y cuál será el tipo de datos que podrá almacenar, en Java es

obligatorio declarar las variables para poder utilizarlas y se debe seguir la siguiente sintaxis:

[private|public|protected] <tipoVariable> <nombreVariable> [= valor][, nombre = [valor]….] ;

Una variable puede declararse en una clase, en cuyo caso se correspondería con el tipo de miembro que también se denomina atributo de la clase y puede ser usada en toda las clase donde fue definida, también pueden ser locales las cuales sólo son accesibles dentro del método que la declara, o como parámetros de entrada donde serían accesibles dentro del método que las está declarando.

Por ejemplo para declarar una variable entera de nombre total, debe hacerse de la siguiente manera:

```
int total;
int total=0;
int total1=0, total2=0;
```

Los nombres de variables pueden llevar caracteres alfanuméricos, pero no pueden tener caracteres especiales, tampoco pueden ser palabras reservadas. A continuación se muestran palabras reservadas del lenguaje Java:

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
int	interface	long	native	new	null
package	private	protected	public	return	short
static	super	switch	syncronized	this	throw
throws	transient	try	void	volatile	while

Tabla 2. Palabras reservadas lenguaje Java

TIPOS DE DATOS

Dentro del lenguaje Java existen tipos sencillos o también llamados primitivos y tipos compuestos (tipo de datos de referencia, que corresponde a un objeto de una clase).

A continuación se detallan los tipos sencillos de datos:

Tipos de dato	***Subtipos***	***Descripción***	
Entero	Int, short, byte	Int	4 bytes. Valores enteros entre -2.147.483.648 y 2.147.483.647
		Short	2 bytes. Valor entero entre -32768 y 32767
		Byte	1 byte. Valor entero entre -128 y 127
Flotantes	Float, double	Float	4 bytes. Números reales hasta 7 decimales desde -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38
		double	8 bytes. Números reales hasta 15 decimales desde -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308
Caracteres	Char	2 byte. Unicode caracteres del código ASCII	
Booleano	Boolean	1 byte. Valores true(verdadero) y false(falso)	

Tabla 3. Tipos sencillos o primitivos en Java

Java es muy restrictivo con los tipos de datos, para poder asignar una variable de un tipo a otro se puede hacer conversión de tipos de datos de forma sencilla, por medio de la conversión automática y usando el casting, lo cual realiza una conversión forzada de un tipo de dato a otro.

Según Garrido(2015), las conversiones automáticas sólo son posibles si el tipo de dato al que se necesita convertir pertenece de manera general al mismo tipo de dato (son compatibles entre sí) y la variable destino tiene mayor rango de valores que la de origen, es decir, es más grande. A continuación se muestra un ejemplo de asignar una variable byte a una entera, la cual es válida ya que byte es más pequeño que int :

```
byte varbyte=1;
int entero;
entero=varbyte;
```

Sin embargo, para asignar variables de diferentes datos o de tamaños menores al de origen es posible hacerlo mediante el comando cast, en estos casos el valor de mayor tamaño es truncado, es decir es posible perder datos por la poca capacidad de la variable receptora, se debe usar usando el siguiente formato:

(tipo_de_dato_destino)valor;

En el caso inverso de que sea necesario asignar un entero a un byte, debemos usar el cast de la siguiente forma:

```
byte varbyte=1;
int entero;
varbyte=(byte)entero;
```

Los tipos de datos compuestos nos permiten agrupar datos simples en estructuras más complejas, un ejemplo de esto son los arrays, los cuales nos permiten trabajar con mayores cantidades de datos en forma sencilla, únicamente accediendo a los mismos a través de índices que indican el orden que ocupa el dato a buscar. Para declarar un array debe usar los corchetes para determinar el tamaño del mismo, por ejemplo: int codigo_cliente [100] en donde se crea espacio para 100 variables de tipo entero que serán accesibles por medio del nombre codigo_cliente y su índice correspondiente.

Los arrays pueden ser undimensionales (llamados también vectores) o bidimensionales (llamados también matrices), a continuación se muestra un ejemplo de cada uno:

int codigo_cliente[100] → arreglo undimensional
int codigo_cliente[100][50] → arreglo bidimensional

1.2.2. SENTENCIAS DE CONTROL

Según Deitel(2004), un lenguaje de programación utiliza sentencias de control para que el programa realice las repeticiones requeridas o se bifurque en función

de los cambios de estado en el programa. Las sentencias de control se clasifican en dos grupos: de selección (if,switch) y de repetición(while, do while, for).

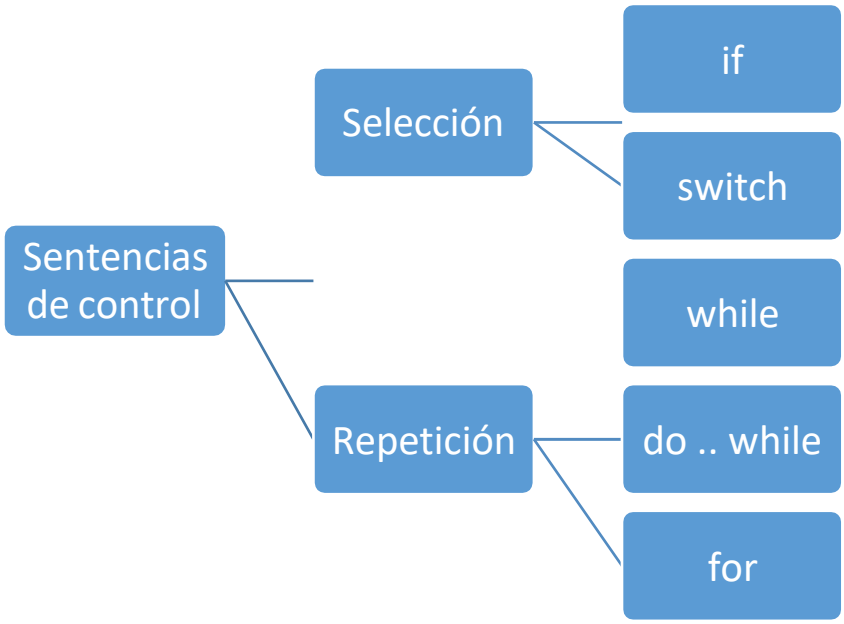


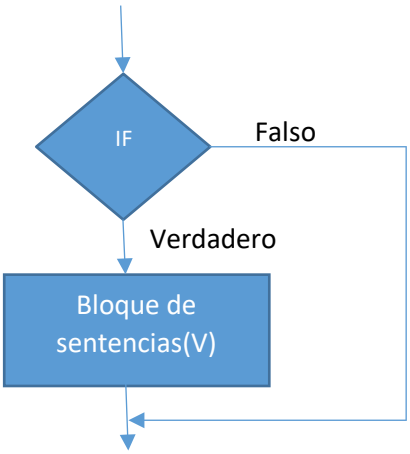
Figura 1. Tipos de sentencias de control

SELECCIÓN

Las sentencias de selección o de bifurcación ejecutan un bloque de sentencia u otro según el resultado devuelto de la expresión lógica evaluada. Dentro de las sentencias de selección tenemos: if, if else, switch

Estructura selectiva simple IF

Bifurcación if, ejecuta un bloque de sentencias solo cuando se cumple la condición del IF, es decir cuando al evaluar la condición esta devuelve true. Si la condición es verdadera se ejecuta el bloque de sentencias, si la condición es falsa el flujo del programa continua en la sentencia inmediatamente posterior al final del IF, es decir no se ejecuta el bloque if se ignora.



If (condición_a_evaluar)

{

bloque de sentencias a ejecutar si es verdadera

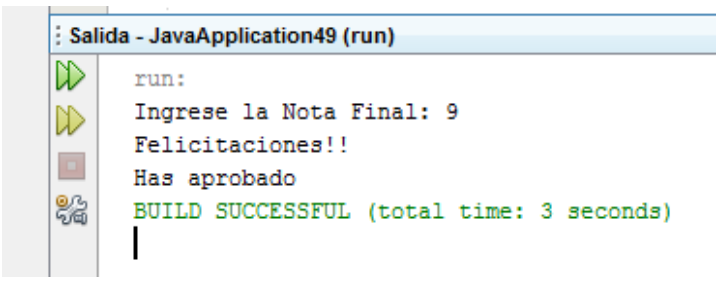
}

```
if (d==0)
System.out.print(" el numero es nulo ");
```

Ejemplo:

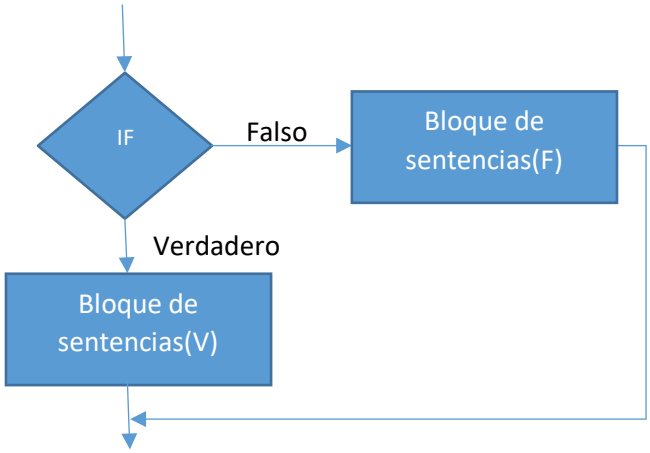
```
import java.util.*;
public class JavaApplication49 {
    public static void main(String[] args) {
/*
 * Programa que pide una nota por teclado y muestra un mensaje si la nota es
 * mayor o igual que 7
 */
        Scanner sc = new Scanner( System.in );
        System.out.print("Ingrese la Nota Final: ");
        int nota = sc.nextInt();
        if (nota >= 7 ){
            System.out.println("Felicitaciones!!");
            System.out.println("Has aprobado");
        }
    }
```

El programa lee por teclado el valor de la Nota Final y muestra Aprobado si esta es mayor a 7 puntos, como podemos observar a continuación:



Estructura selectiva doble IF ELSE

Birfurcación if else, esta sentencia ejecuta un bloque por verdadero o un bloque por falso, es decir, entra obligatoriamente al if evalúa la condición lógica y ejecuta un bloques de sentencias mutuamente excluyentes. Si se cumple la condición, se ejecuta el bloque de sentencias asociado al IF(verdadero), caso contrario si la condición no se cumple, entonces se ejecuta el bloque de sentencias asociado al ELSE(falso).




```
If (condición_a_evaluar)
{
bloque de sentencias a ejecutar si es verdadera
}
Else
{
bloque de sentencias a ejecutar si es falsa
}
```

```
if(a%2==0)
System.out.println("El numero es par");
else
System.out.println("El numero es impar");
```

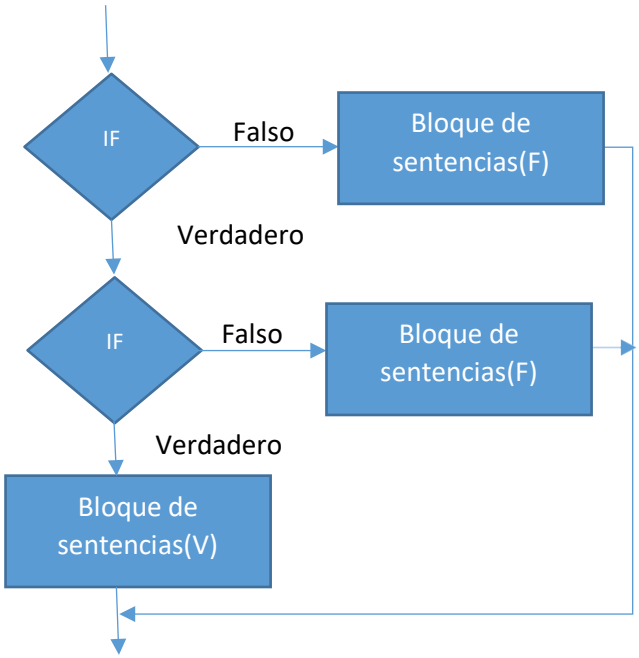
En los casos que el programa requiera que por verdadero se ejecuten unas instrucciones y por falso otras se debe usar la estructura if else como vemos en el ejemplo siguiente:

```java
/*
 * Programa que pide una nota por teclado y muestra si se ha aprobado o no
 */
        Scanner sc = new Scanner( System.in );
        System.out.print("Ingrese la Nota final : ");
        int nota = sc.nextInt();
        if (nota >= 7){
            System.out.println("Felicitaciones!!");
            System.out.println("Has aprobado");
        }
        else
            System.out.println("Lo Siento, has reprobado");
    }
}
```

```
a - JavaApplication49 (run)
run:
Ingrese la Nota final : 7
Felicitaciones!!
Has aprobado
BUILD SUCCESSFUL (total time: 8 seconds)
```

Estructura anidada IF ELSE IF

En ciertos algoritmos es necesario evaluar una condición dentro de otra, en esos casos se utiliza un IF ELSE IF anidado donde el bloque de sentencias a ejecutar incluye otro IF, el programador debe tener un control adecuado de donde terminan los resultados por verdadero o por falso para evitar resultados no deseados en tiempo de ejecución. Un ejemplo puede ser el siguiente:




```
If (condición_a_evaluar)
{
   If (condición_2)
    { bloque verdadero_c2}
  Else
    { bloque_falso_c2}
}
Else
{
bloque de sentencias a ejecutar si es falsa
}
```

```
BufferedReader in;
in = new BufferedReader (new InputStreamReader (System.in));
System.out.println("Ingrese un numero entero: ");
int d = Integer.parseInt(in.readLine());
if (d==0)
System.out.print(" el numero es nulo ");
else
{if (d<0)
System.out.print("El numero es negativo");
else
System.out.print("El numero es positivo");}
}}
```

Es posible que el programa requiera estructuras más complejas como IF anidados, a continuación veremos un programa con este caso:

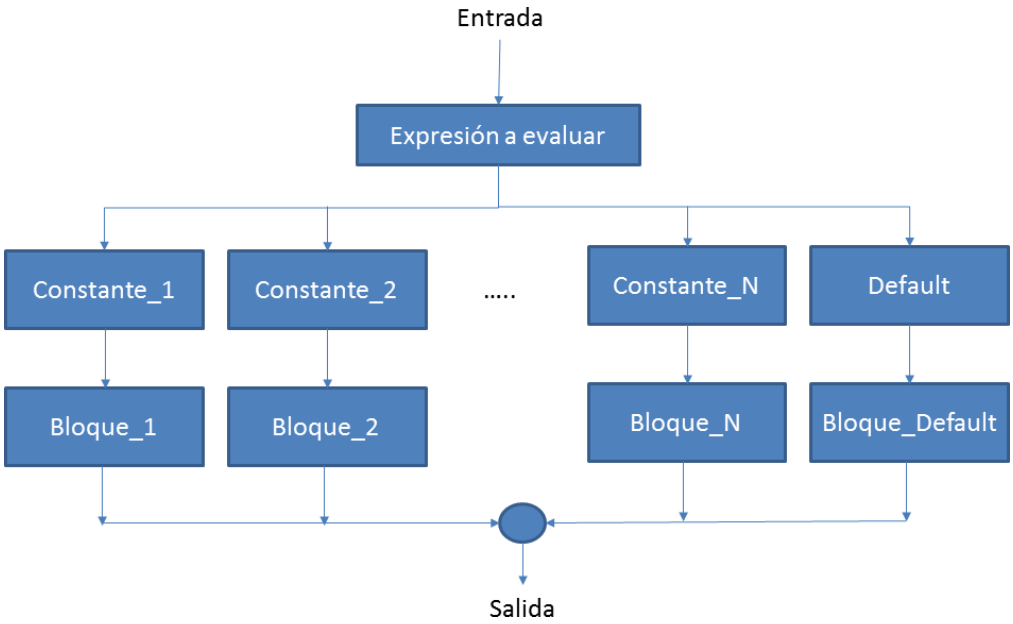
```
import java.util.*;
public class JavaApplication49 {
    public static void main(String[] args) {
/*
 * Programa que muestra un saludo distinto según la hora introducida
 */
        Scanner sc = new Scanner(System.in);
        int hora;
        System.out.println("Introduzca una hora (un valor entero): ");
        hora = sc.nextInt();
        if (hora >= 0 && hora < 12)
            System.out.println("Buenos días");
        else if (hora >= 12 && hora < 21)
             System.out.println("Buenas tardes");
        else if (hora >= 21 && hora < 24)
              System.out.println("Buenas noches");
        else
              System.out.println("Hora no válida");
    }
}
```

```
la - JavaApplication49 (run)
run:
Introduzca una hora (un valor entero):
9
Buenos días
BUILD SUCCESSFUL (total time: 5 seconds)
```

Cuando se ejecuta una sola instrucción por verdadero o falso no son necesarias las llaves, sin embargo, cuando es necesario más de una instrucción se debe tener precaución en abrir llaves y cerrar llaves para evitar comportamientos no deseados del programa, el orden en la programación asegura el éxito de la misma.

Estructura SWITCH

Cuando en un programa existen más de dos posibles bifurcaciones del código la solución es implementar una sentencia selectiva como SWITCH que permita seleccionar un bloque de sentencias entre varios casos. Es parecido a una estructura de IF ELSE, pero simplifica la lectura y comprensión del código. La estructura SWITCH consta de una expresión lógica que se evalúa al inicio y una serie de etiquetas CASE que representan cada posible valor a tomar y una opción DEFAULT opcional que se ejecutará en caso de que no exista ninguna sentencia case válida. El uso del BREAK es muy importante en la sentencia SWITCH puesto que indica el final de la ejecución del bloque case.




```
Switch(expresión_a_evaluar) {
        Case constante_1: bloque_1; break;
        Case constante_2: bloque_2; break;
        .......
        Case constante_n: bloque_n; break;
        Default: bloque_default;
}
```

La sentencia switch es muy usada en los menús de opciones o cuando el programa necesita ejecutar bloques de sentencias diferentes según el valor de la opción.

```
int opc;
System.out.println("--------------MENU DE OPCIONES --------------");
System.out.println("1. Seno de un numero");
System.out.println("2. Coseno de un numero");
System.out.println("3. Logaritmo de un numero");
System.out.println("4. Salir");
System.out.println("Ingrese una opcion: ");
BufferedReader in;
in=new BufferedReader(new InputStreamReader(System.in));
opc=Integer.parseInt(in.readLine());
switch(opc)
{
case 1:System.out.println("1. Calcular Seno de un numero");break;
case 2:System.out.println("1. Calcular Coseno de un numero");break;
case 3:System.out.println("1. Calcular Logaritmo de un numero");break;
   }

}
}
```

Las sentencias siempre deben finalizar con un break, ya que de esta manera se asegura que sólo se ejecute uno de los casos del swtich.

```
/*
 * Programa que asigna habitaciones según el nivel
 */
                int nivel = 5;

                switch(nivel) {
                        case 5:
                                System.out.println("Habitación 1");
                                break;
                        case 10:
                                System.out.println("Habitación 2");
                                break;
                        case 15:
                                System.out.println("Habitación 3");
                                break;
                        case 20:
                                System.out.println("Habitación 4");
                                break;
                        default:
                                System.out.println("Habitación 5");
                }
}}
```

```
a - JavaApplication49 (run)
run:
Habitación 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Es posible también que existan case alineados, es decir que el programa tome un mismo camino por más de un valor, en ese caso se puede usar un switch como el siguiente:

```
/*
 * Programa que asigna habitaciones según el nivel
 */
        int nivel = 5;

        switch(nivel) {
                case 5:case 10:case 15:
                        System.out.println("Habitación 3");
                        break;
                case 20:
                        System.out.println("Habitación 4");
                        break;
                default:
                        System.out.println("Habitación 5");
        }

        System.out.println("Fuera del switch (y de las habitaciones)");
    }
}
```

```
a - JavaApplication49 (run)
run:
Habitación 3
Fuera del switch (y de las habitaciones)
BUILD SUCCESSFUL (total time: 1 second)
```

SENTENCIAS REPETITIVAS

Según Aguilar(2011), también llamados bucles o estructuras iterativas, en los programas siempre se necesitan repetir muchas veces un bloque de sentencias mientras se cumpla una condición, en Java encontramos while, do while, for.

Las estructuras repetitivas tienen una variable que lleva el control del bucle y la cual participa en todas las partes del mismo, es posible encontrar las siguientes partes en un ciclo repetitivo

- Inicio del ciclo: Consiste en asignar en valor inicial a la variable de control del bucle.

- Condición: Es la parte más importante del ciclo, aquí se válida hasta cuando se repiten las sentencias, es necesario que esta condición cambie su estado en algún momento del programa ya que si no lo hace es posible que se cree un ciclo infinito.

- Bloque de Sentencias: Conjunto de sentencias que repetirán cuando la condición sea verdadera.

- Actualización: En cada repetición, es necesaria actualizar la variable de control del bucle para controlar su posible cambio de estado.

Al trabajar con estructuras repetitivas se utilizan contadores, acumuladores, forzar la salida del bucle y continuar al inicio del bloque.

Contador

Son variables de tipo entero que incrementan o decrementan en un valor constante (en cada iteración del bucle), y que pueden utilizarse para contar el número de iteraciones del bucle. Deben ser inicializadas antes del bucle para evitar valores errados, con el valor de inicio que requieren según el problema.

Las variables tipo contador tienen el siguiente formato:

contador = contador + 1 ó contador++

Por ejemplo:

```
c = c + 1;
a += 2;
b--;
```

Para el siguiente ejemplo en Java se inicializa el contador en 0 y se incrementa en uno hasta que llega a 5, imprimiendo en cada iteración el valor del contador, a continuación el código y la ejecución del mismo:

```
package javaapplication49;
public class JavaApplication49 {
    public static void main(String[] args) {
                int contador = 0;
                while (contador < 5) {
                        contador = contador + 1;
                        System.out.println(contador);
                        }
            }
    }
```

```
a - JavaApplication49 (run)
run:
1
2
3
4
5
BUILD SUCCESSFUL (total time: 0 seconds)
```

Una variable de tipo contador se puede usar por ejemplo para:

1. Número de veces que se hace clic en un botón
2. Número de notas ingresadas
3. Número de estudiantes aprobados
4. Cuántos números positivos se han ingresado

Acumulador

Son variables del mismo tipo que almacena valores en forma acumulativa. Al igual que el contador debe tener un valor de inicio, suma valores en cada iteración del bucle. La inicialización asegura que la variable no tenga un valor basura que arrastre desde la memoria o de corridas anteriores del programa.

Una instrucción de acumulación tiene la siguiente forma:

acumulador = acumulador + cantidad

ó

acumulador += cantidad

Es posible usar una variable de tipo acumulador en los casos siguientes:

1. La suma total de los sueldos de los empleados
2. La suma de las notas de un estudiante
3. La edad total de una lista de estudiantes
4. La suma total de los depósitos de una cuenta

Para el siguiente ejemplo se calcula el factorial del número n, en este caso el factorial de 5. El factorial de 5 es: resultado=5*4*3*2*1, dando como total 120. La variable acumuladora es resultado y almacena el resultado de las multiplicaciones sucesivas para encontrar el factorial.

```
package javaapplication49;
public class JavaApplication49 {
    public static void main(String[] args) {

        int resultado = 1, n=5;
        for (int i = 1; i <= n; i++) {
            resultado *= i;
        }
        System.out.println("Factorial de "+n+" es "+resultado);
        }
    }
```

```
- JavaApplication49 (run)
run:
Factorial de 5 es 120
BUILD SUCCESSFUL (total time: 0 seconds)
```

Las características de una variable acumuladora son:

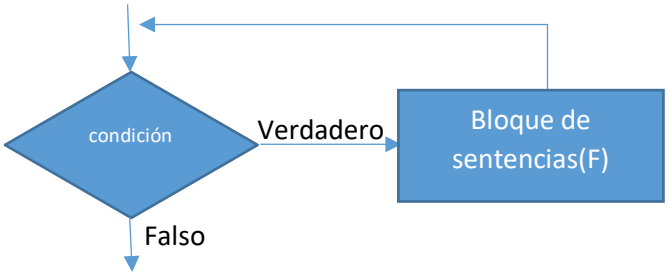
1. La variable acumuladora debe aparecer a la derecha y a la izquierda de la asignación.
2. Se realiza con operadores aritméticos +,-,*,/
3. En una expresión de acumulación no se usan operadores booleanos ni relacionales.

Estructura WHILE

La estructura de repetición WHILE repite el bloque de sentencias mientras la condición WHILE es verdadera, la condición se evalúa al inicio del bloque se utiliza generalmente cuando no se conoce exactamente el número de repeticiones a ejecutarse.

La sintaxis de while es:

```
While (condición_a_evaluar)
{
   Bloque de sentencias a repetir si es verdadera la condición
}
```




```
while (opc<1||opc>5)
{System.out.println("1. Volumen de un Paralelepipedo Rectangular");
System.out.println("2. Volumen de una Esfera");
System.out.println("3. Volumen de un Cilindro Recto");
System.out.println("4. Volumen de un Cono Circular Recto");
System.out.println("5. Salir");
System.out.println("Seleccionar su opcion: ");
opc = Integer.parseInt(in.readLine( ));
}
```

Estructura DO WHILE

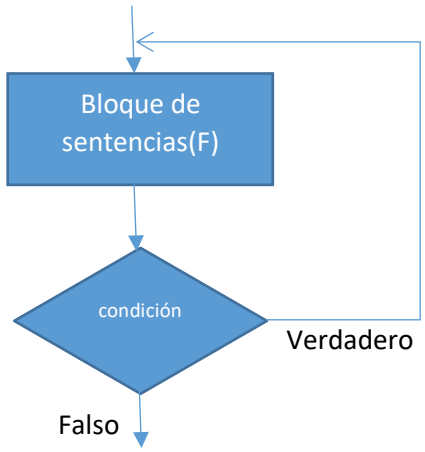
La sentencia repetitiva do while evalúa la condición al final, por lo cual ejecuta el bloque de sentencias al menos una vez después comprueba la condición, repite el bloque de sentencias mientras la condición es verdadera.

La sintaxis de do while es:

Do {

Bloque de sentencias a repetir si es verdadera la condición

} while (condición_a_evaluar);




```
do
{
System.out.println("MENU PARA CALCULAR FUNCIONES MATEMATICAS");
System.out.println("--------------------------");
System.out.println("1. RAIZ CUADRADA");
System.out.println("2. LOGARITMO NEPERIANO");
System.out.println("3. LOGARITMO DECIMAL");
System.out.println("4. VALOR DE e ELEVADO A LA X");
System.out.println("5. INGRESAR EL VALOR");
System.out.println("6. SALIR");
System.out.println("INGRESE LA OPCION POR FAVOR");
resp=Integer.parseInt(in.readLine());
switch (resp){
case 1:raiz(x);break;
case 2:lognep(x);break;
case 3:logdec(x);break;
case 4:ealax(x);break;
case 5:
x = (Double.valueOf(in.readLine())).doubleValue();break;
case 6:
break;
}
}
while(resp!=6);
```

Estructura FOR

La sentencia repetitiva FOR sólo se utiliza cuando se sabe el número de veces que se debe repetir el bloque de sentencias. Repite el bloque de sentencias mientras la

condición del FOR es verdadera. Un FOR es un caso particular de la estructura WHILE, la variable de control del bucle tiene un valor de inicio, un valor de incremento/decremento y uno de fin.

La sentencia de FOR es:

For (variable=valor_inicio;condición_a_evaluar;incremento/decremento)

{

Bloque de sentencias a repetir si es verdadera la condición

}

```
for (opc=1;opc<10;opc++)
{
    System.out.println("Opc = "+opc);

}
```

Salida:

```
Opc = 1
Opc = 2
Opc = 3
Opc = 4
Opc = 5
Opc = 6
Opc = 7
Opc = 8
Opc = 9
BUILD SUCCESSFUL (total time: 6 seconds)
```

SALTO DE FLUJO NORMAL DEL PROGRAMA

Un programa puede ver cambiado el transcurso normal lineal del código debido al uso de dos comandos que se han diseñado para tener un mayor control justamente dentro de los bucles, permiten modificar el flujo secuencial de un programa y provocan un salto de ejecución. Estas sentencias son BREAK y CONTINUE, las cuales se utilizan frecuentemente con las sentencias repetitivas, sin embargo también pueden ser usadas dentro de las sentencias selectivas.

Sentencia BREAK

Se utiliza para interrumpir la ejecución de una estructura de repetición o de un SWITCH. Cuando se ejecuta el BREAK, el flujo del programa continúa en la sentencia inmediatamente posterior a la estructura de repetición o al SWITCH.

```
for(int i=0;i<10;i++)
{
System.out.println(i);
if(i==5)
break;
}
```

```
0
1
2
3
4
5
BUILD SUCCESSFUL (total time: 3 seconds)
```

Aunque el for está diseñado para llegar hasta 10 este se detiene en 5 por el break y termina el bucle

Sentencia CONTINUE

Esta aparece únicamente en una estructura de repetición. Cuando se ejecuta un CONTINUE, deja de ejecutar el resto del bloque de sentencias de la estructura iterativa para volver al inicio de esta.

```
for(int i=0;i<10;i++)
{
if(i==5)
continue;
System.out.println(i);
}
```

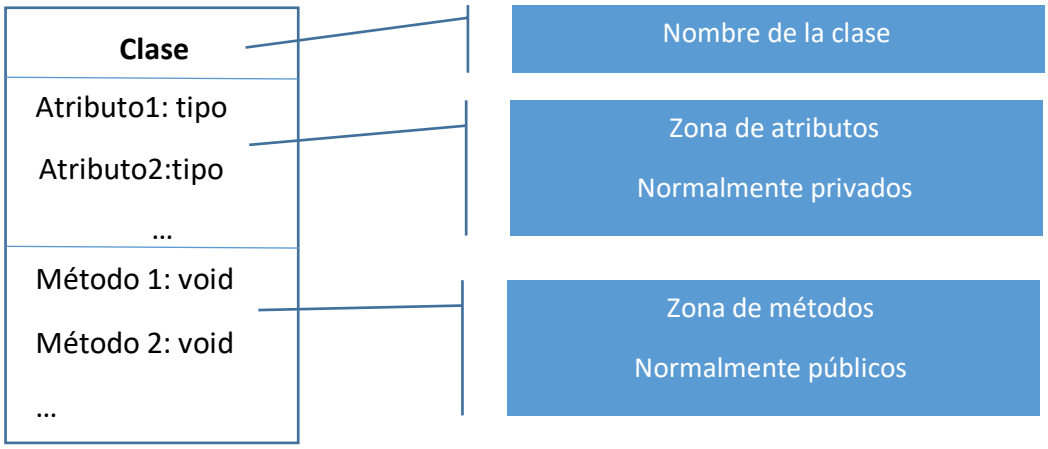
Salida:

```
0
1
2
3
4
6
7
8
9
BUILD SUCCESSFUL (total time: 7 seconds)
```

Aunque el for está diseñado para llegar hasta 10 el número 5 no se imprime porque tiene un continue, al llegar a 5 vuelve al inicio del bucle.

1.2.3. MÉTODOS Y CLASES

Según Pérez(2015) para representar gráficamente los ejemplos se utilizará el Lenguaje Unificado de Modelado (UML), el cual es un lenguaje gráfico para visualizar, documentar y construir las partes de un sistema de software orientado a objetos. El diagrama de clases permite visualizar claramente cuáles son los elementos de las mismas: atributos y métodos, los cuales se muestran a continuación:



Atributos

Son las características o los datos de las clases, en los objetos son las características individuales que diferencia un objeto de otro.

Por ejemplo en la clase carros, se tienen atributos como placa y marca.

Objeto de clase carro: carro1

Placa = “GRY-8855”

Marca = “Mazda”

Objeto de clase carro: carro2

Placa = “GRY-9755”

Marca = “Chevrolet”

```
public class carros {
    private String placa;
    private String marca;

}
```

Métodos

Es el comportamiento de los objetos de una clase dada, son los procedimientos que interaccionan con los atributos o datos. Constituyen la lógica de la clase, ya que contienen el código que manipula el estado del objeto.

```
public class carros {
    private String placa;
    private String marca;
    private int velocidad;

    public void detener()
    {
        velocidad=0;
    }

    public void avanzar()
    {
        velocidad=velocidad+10;
    }
```

Método detener()

Establece el valor del atributo velocidad en 0

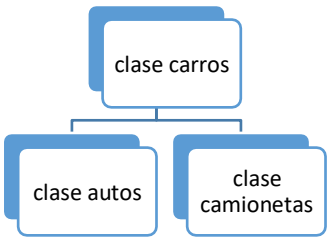
Método avanzar()

Incrementa el valor del atributo velocidad en 10

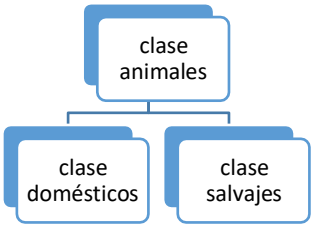
Clases

Es una colección de objetos que poseen características y operaciones comunes. Ya que esta contiene toda la información necesaria para crear nuevos objetos.

Clase carros



Clase Animales



1.3. CLASES Y OBJETOS

1.3.1. DEFINICIÓN DE UNA CLASE

Las clases son el conjunto de atributos y métodos, los cuales pueden ser públicos o privados, generalmente se declara a los atributos privados(sólo accesibles desde miembros de la clase) y los métodos públicos. Los atributos definen los datos que debe almacenar la clase y los métodos la funcionalidad de la misma, las clases constituyen la plantilla a partir de la cual se crean los objetos. Para crear las clases en Java se utiliza la palabra reservada class.

La sintaxis para definir una clase es:

class Nombre_de_la_clase {

datos y funciones privados

……

public:

datos y funciones públicas

……

} lista de objetos;

Ejemplo: Definir la clase punto en un espacio con dos dimensiones, en donde los atributos de la clase serían x e y.

Punto
Int x Int y

```
Public class Punto
{
  Private int x,y;
}
```

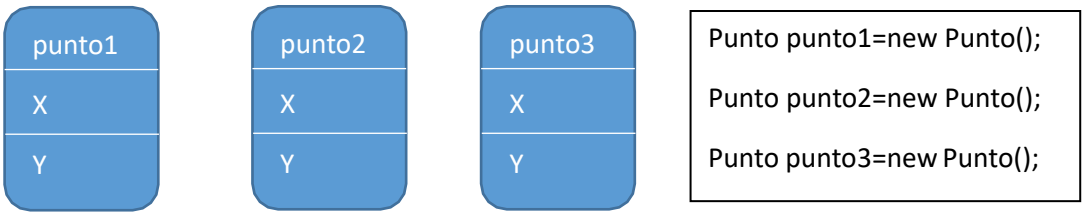
Las reglas para los identificadores de clase son:

- Tener una longitud máxima de 32

- Que la primera letra del nombre sea mayúscula
- No debe tener caracteres especiales ni espacios

1.3.2. OBJETOS

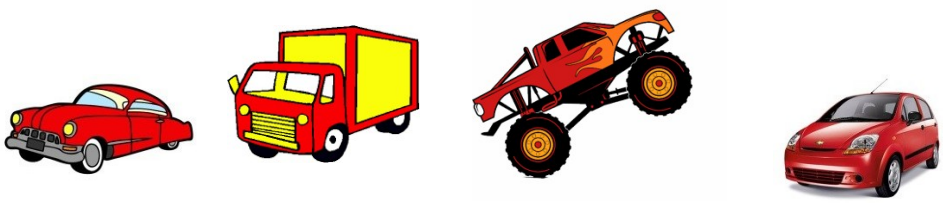
Los objetos son instancias de la clase, es a través de los objetos que podemos trabajar en el programa, ocupan un espacio en memoria y podemos crear cuantos objetos soporte la memoria del computador. Cada objeto tendrá un estado, es decir valores individuales el uno del otro y podrán ejecutar los métodos de la clase en forma independiente. Para crear un objeto se utiliza la palabra reservada new y visualmente si creamos objetos de la clase punto se vería de la siguiente forma:



Los objetos se crean con los atributos y métodos definidos en la clase, internamente la creación de un objeto se realiza de la siguiente manera:

1. Al crear el primer objeto de la clase, se localiza la clase y se carga en memoria.
2. Se inicializan las variables estáticas de la clase
3. Se reserva la memoria para el nuevo objeto
4. Se inicializan las variables con valores por defecto
5. Se ejecutan el método constructor invocado

Los objetos representan cosas, pueden ser reales o imaginarios, a continuación se muestran algunos ejemplos, objetos de la clase carros:



1.3.3. MÉTODOS

Los métodos son las funciones que permitirán acceder a los atributos de la clase, pueden ser de varios tipos:

- Modificadores, cuando cambian los valores de los atributos
- Selectores, cuando devuelven el valor de los atributos
- Iteradores, cuando cambian algún atributo y devuelven datos a la clase
- Constructores, son un método especial que se ejecuta cuando se crean los objetos en forma automática, generalmente se utiliza para inicializar variables de las clases. Este método lleva el mismo nombre de la clase, pero no es de ningún tipo, la clase puede tener más de un constructor.

La declaración de los métodos en el lenguaje Java se realiza de la siguiente forma:

Modo_de_acceso modificadores tipo_de_retorno nombre(argumentos)

{

Cuerpo del método

}

Modo_de_acceso: public, private, protected

El modo de acceso que se defina tanto para atributos como para métodos define los procedimientos para acceder a los mismos, en la siguiente tabla se observa el nivel de visibilidad según el modo de acceso seleccionado:

Modo_de_acceso	***Public***	***Protected***	***Private***
Desde la propia clase	Si	Si	Si
Desde otra clase en el mismo paquete	Si	Si	No
Desde otra clase fuera del paquete	Si	No	No
Desde una subclase del mismo paquete	Si	Si	No
Desde una subclase fuera del paquete	Si	Si	No

Modificadores: static, abstract, final, native, synchronized

Las variables estáticas son variables de la clase no de los objetos, se crean con el primer objeto de la clase.

Las variables final no pueden cambiar el valor durante la ejecución del programa.

Las clases abstractas se crean para servir como molde de otras clases, de las clases abstractas no se pueden crear objetos.

Tipo de retorno: Si el método no devuelve ningún valor debe ser void, caso contrario debe ser del tipo devuelto puede ser int, float, char, de otro tipo de objetos, entre otros.

Argumentos: Son los parámetros de entrada que recibe el método, si no recibe ninguno entonces debe quedar vacío.

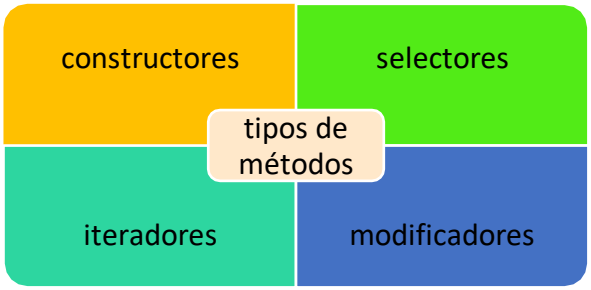


Figura 2. Tipos de métodos

En el siguiente ejemplo veremos los diferentes tipos de métodos:

Modo_de_acceso

```
public class fechas {
    public int dia,mes,anio;

    public fechas()
    {
        dia=01;
        mes=05;
        anio=2016;
    }

    public void cambiar_dia(int nuevo_dia)
    {
        dia=nuevo_dia;
    }

    public int devuelve_dia()
    {
        return dia;
    }
```

Método constructor

Método modificador

Método selector

Otro ejemplo: Métodos de la clase carros

Avanzar ()

Detener ()

Acelerar ()

Sobrecarga de métodos

El lenguaje Java permite la sobrecarga de métodos, es decir, métodos con el mismo nombre pero que se diferencian por el número de argumentos o por el tipo de datos que reciben o devuelven, es posible tener sobrecarga de cualquier método, en el siguiente ejemplo se sobrecarga al método constructor, la diferencia radica en el número de argumentos del método. La sobrecarga de métodos es una de las maneras en las que Java implementa el polimorfismo(Ver Capítulo 3).

```
public class fechas {
    public int dia,mes,anio;

    public fechas()
    {
        dia=01;
        mes=05;
        anio=2016;
    }


    public fechas(int d, int m, int a)
    {
        dia=d;
        mes=m;
        anio=a;
    }
```

CREACION DEL OBJETO

fechas f1=new fechas();

fechas f2=new fechas(31,12,2016);

Veamos otro ejemplo en la clase carros:

```
public class carros {
    private String placa;
    private String marca;
    private int velocidad;

    public void detener()
    {
        velocidad=0;
    }


    public void avanzar()
    {
        velocidad=velocidad+10; // Método que avanza 10 km
    }

    public void avanzar(int valor_avanzar)
    {
        velocidad=velocidad+valor_avanzar; // Método que avanza un valor
                                           //que se envía como parámetro
    }

}
```

Método que avanza 10 km

Método que avanza valor ingresado por parámetro

1.3.4. PROPIEDADES

Con el objetivo de realizar la implementación de las clases en una forma óptima, se recomienda la utilización de las propiedades en Java, las cuales se pueden definir como métodos especiales que obtienen o envían datos a los atributos de la clase, son de tipo público.

La sintaxis para declarar un método que tenga la funcionalidad de las propiedades seria de la siguiente forma:

```
public tipo_dato_atributo getAtributo (){
    return atributo;
}
```

Lee atributo de la clase

```
public void setAtributo (tipo_dato_atributo variable){
   this.atributo = variable;
}
```

Modifica atributo de la clase

En el siguiente ejemplo se muestran métodos set y get para el atributo nombre:

```
public class Persona {
  private String nombre;

  public String getNombre() {
    return this.nombre;
  }
```

```
  public void setNombre(String nombre) {
    this.name = nombre;
  }

}
```

Lineamientos Generales y Estructura de un programa en Java

Todo en Java es una clase y los programas no son archivos individuales sino proyectos. Para el desarrollo de los ejemplos incluidos en este libro se utilizará el entorno de desarrollo libre NetBeans en donde los proyectos en Java están organizados en paquetes "package", las mismas que permiten agrupar las clases comunes que son parte del proyecto, en Java hay unos 59 paquetes lo cual nos dá una idea de la amplia gama de soluciones que se han desarrollado a través del tiempo y que al programador crear soluciones informáticas de alta calidad. Sin embargo, es posible crear nuevos paquetes, de hecho al crear el proyecto este crea automáticamente un paquete con el mismo nombre y todas las clases que pertenecen a la solución deberían estar agrupadas en el mismo paquete, es decir en la primera línea de cada clase debe constar a q `package nombre_proyecto;` quete puede contener:

- Clases
- Interfaces
- Tipos enumerados
- Anotaciones

Es posible que se importen paquetes para utilizar las clases públicas del mismo en la solución que se está desarrollado, para esto se utiliza el comando import de la siguiente manera: import java.awt.Graphics

En los programas en Java es importante tener en cuenta si la sentencia está escrito en mayúsculas o minúsculas, o si la primera letra es mayúscula y el resto minúsculas, es decir, son sensibles entre letras mayúsculas y minúsculas, por ejemplo: nombre, NOMBRE y Nombre son variables distintas, es decir aquí tenemos tres variables, de allí la importancia de tener en cuenta exactamente como se declaran los atributos y los métodos. Sin embargo es conveniente seguir ciertos

lineamientos en las escritura del código para que la programación se realice en forma ordenada y sistemática, sugiero las siguientes normas:

1. Los nombres de clases deben tener la primera letra mayúsculas, por ejemplo: Clase Persona, Clase Animales, Clase Figuras.
2. Los métodos deben ir en minúsculas, por ejemplo area(), calcular_sueldo()
3. Los nombres de variables de tipo final, en mayúsculas por ejemplo PI

La extensión de los ficheros fuentes es *.java y la de los ficheros compilados *.class, el nombre del fichero físico debe ser igual al nombre de la clase, por ejemplo si vamos a crear la clase Persona, el fichero al crearse debe llamarse Persona, ya que persona no es igual a Persona en el lenguaje Java.

Los comentarios se escriben de dos maneras:

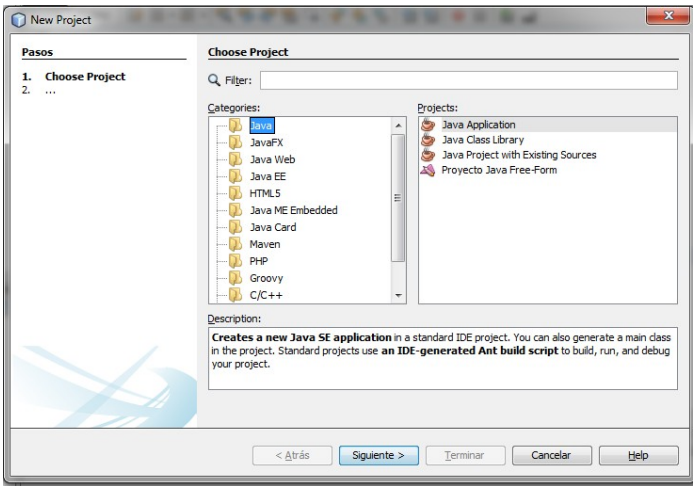
1. Una sola línea usando doble barra, // comentario
2. Varias líneas, entre asterisco /* …. */

Las aplicaciones siempre deberán tener un método main para poder ejecutar las clases creadas, y si el proyecto tiene varias clases estas deberán pertenecer al mismo paquete.

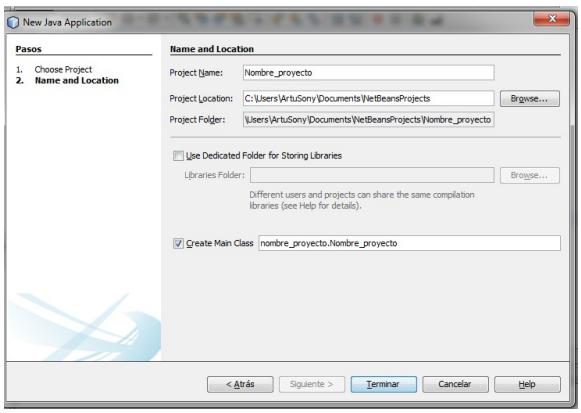
Existen muchos entornos de desarrollo para programar en Java, entre los cuales podemos mencionar Eclipse, IntelliJ IDEA, Netbeans, he incluso no es necesario tener un IDE extenso con muchas funcionalidades, es posible hacerlo en un editor sencillo como jEdit. Sin embargo, la elección del entorno de desarrollo dependerá de las funcionalidades que como desarrollador te hacen sentir más cómodo o de las características del equipo en el cual será instalado, lo cierto es que existe una gran cantidad de entornos de desarrollo para Java.

Para los ejercicios desarrollados en este libro se utilizó NetBeans IDE 8.0.2 y los pasos para crear una aplicación son los siguientes.

1. Abrir el programa NetBeans IDE 8.0.2
2. Clic en Archivo. Nuevo proyecto
3. Seleccionar la Categoría Java y el proyecto Java Application



4. Ingrese el nombre del proyecto



5. Se crea automáticamente un paquete con el mismo nombre del proyecto.

```
package nombre_proyecto;

/**
 *
 * @author ArtuSony
 */
public class Nombre_proyecto {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

}
```

Nombre de la clase

Cuerpo del método

ENTRADA Y SALIDA DE DATOS

Es fundamental interactuar con el usuario a través de las funciones de entrada/salida de datos, en Java para lectura de datos por teclado se utiliza la clase Scanner del paquete java.util ó BufferedReader/InputStreamReader del paquete java.io

Usando la clase Scanner

Según Marín(2016), en Java para obtener información de usuario se utiliza System.in sin embargo este sólo lee la información en bytes razón por la cual es necesario utilizar otras clases, en este caso se usa para leer datos la clase Scanner

Para obtener información ingresada por el usuario desde el teclado, es necesario importar el paquete java.util para poder utilizar el objeto Scanner, el cual permite leer todo tipo de dato desde el teclado del computador. Los pasos para usar la clase scanner son:

1. Importar el paquete java.util.scanner
2. Crear el objeto de la clase scanner y conectarlo con System.in
    a. Scanner teclado=new Scanner(System.in);

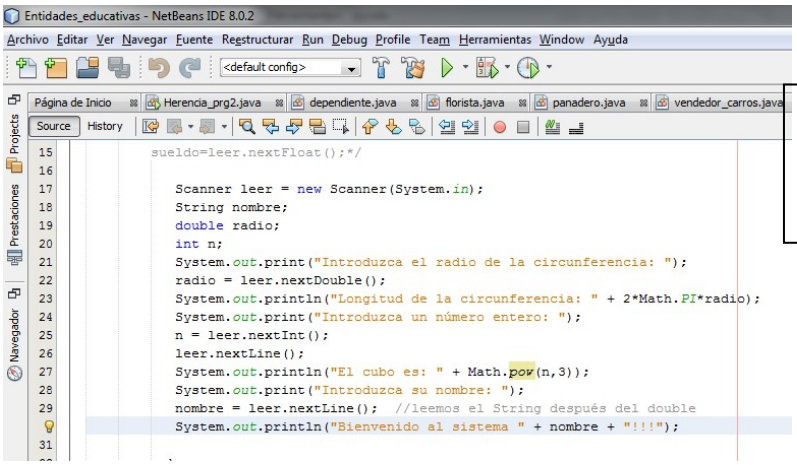
3. Dependiendo del tipo de dato a leer usaremos los siguientes comandos:

Método	Ejemplo
nextByte()	byte b=teclado.nextByte();
nextDouble()	double d=teclado.netxDouble();
nextFloat()	float f=teclado.nextFloat();
nextInt()	int i=teclado.nextInt();
nextLong()	long l=teclado.nextLong();
nextShort()	short s=teclado.nextShort();
next()	String p=teclado.next();
nextLine()	String o=teclado.nextLine();

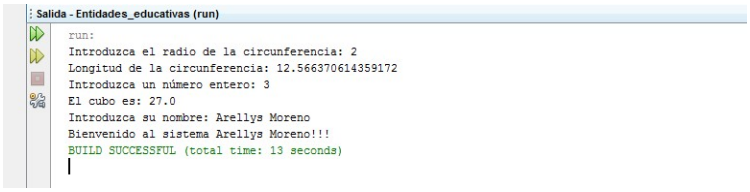
Existen tantos métodos next como tipos de datos en Java, use dependiendo del dato a leer el next____() correspondiente.

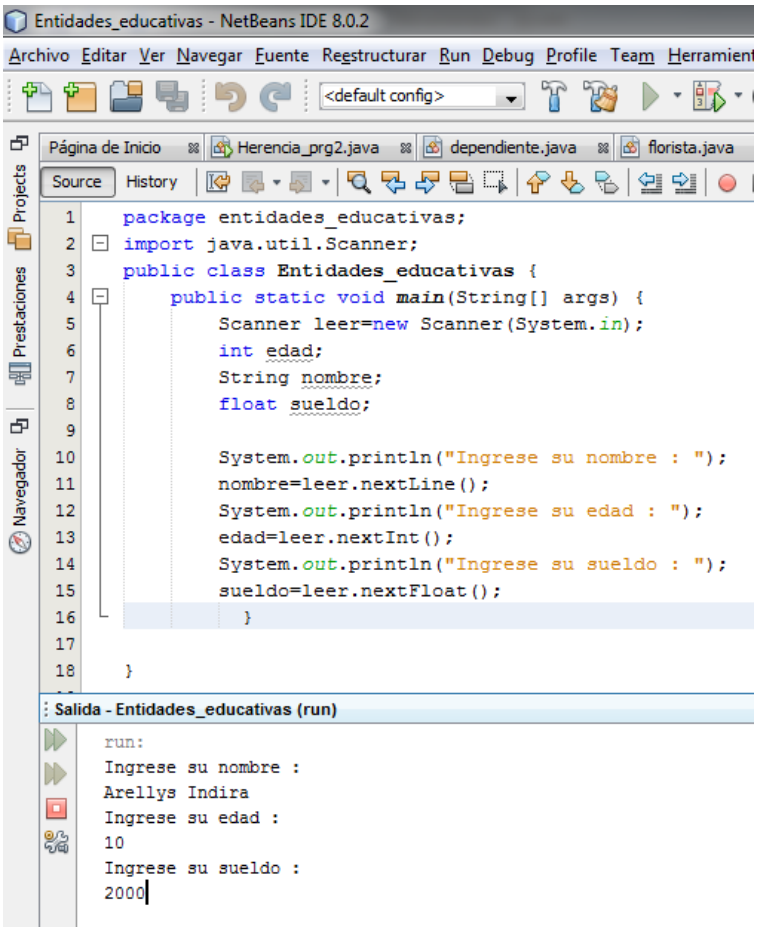
Es importante limpiar el buffer de entrada cuando se leen datos de tipo carácter junto a datos de tipo numérico, para esto se usa el comando nextLine() antes de leer el dato cadena.

A continuación se muestra un ejemplo de la lectura de datos con Scanner.



PROGRAMA DONDE SE LEEN VARIABLES DE TIPO INT, DOUBLE Y STRING.





PROGRAMA DONDE SE LEEN VARIABLES DE TIPO INT, FLOAT Y STRING.

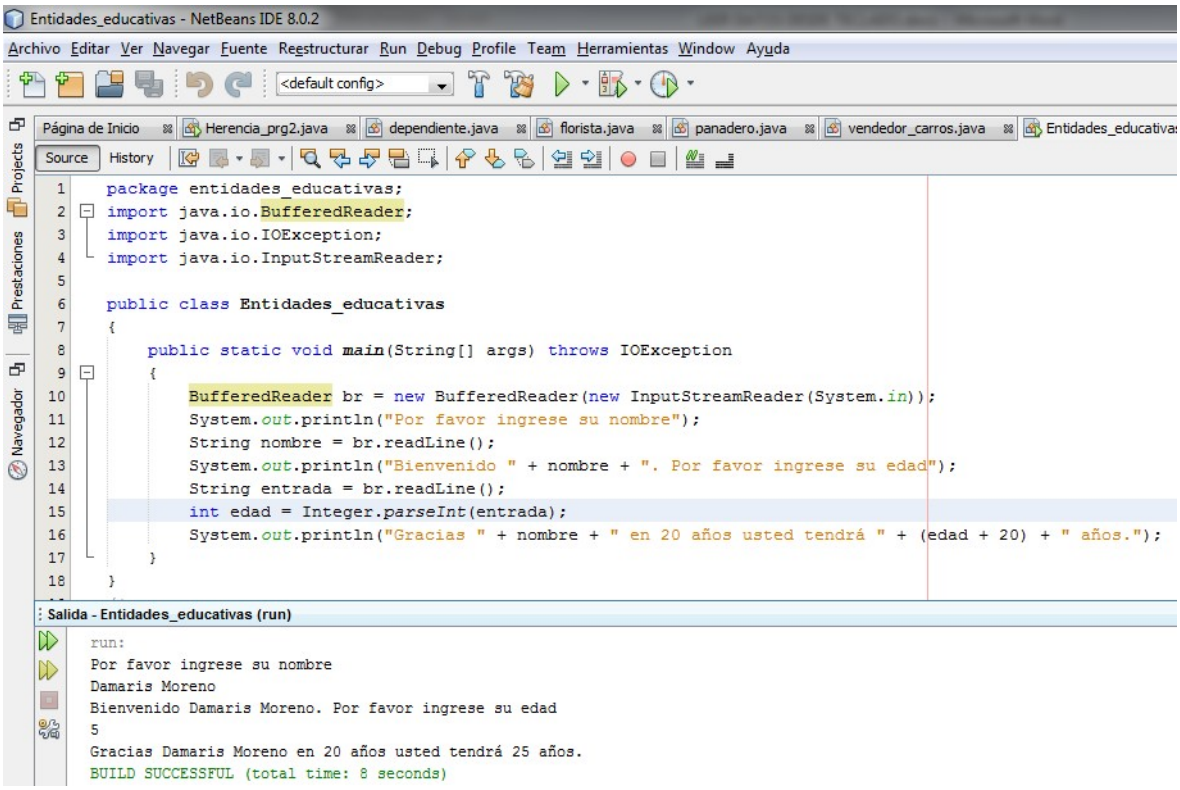
BufferedReader

La segunda manera de leer datos desde teclado es usando la clase BufferedReader, es cual únicamente tiene el método readline() para la entrada y siempre retorna string, por lo cual siempre debemos convertir al tipo de dato numérico a la vez que se debe controlar a través de la IOException. Para realizar el proceso debe seguir los siguientes pasos:

1. El primer paso es importar los paquetes BufferedReader, InputStreamReader, IOException
    a. import java.io.*;

2. Luego declarar un objeto de la clase BufferedReader
    a. BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

3. Y realizar las lecturas de datos usando el método readLine().
    a. String nombre = br.readLine();



Salida de datos en pantalla

Para imprimir por consola en Java se utiliza el comando System.out.println de la siguiente manera:

System.out.println ("Mensaje a mostrar en pantalla");

Es posible concatenar los resultados a mostrar en una sola instrucción println usando el símbolo (+), como se muestra a continuación:

int precio=10;

System.out.println ("El precio es de " + precio + " dólares");

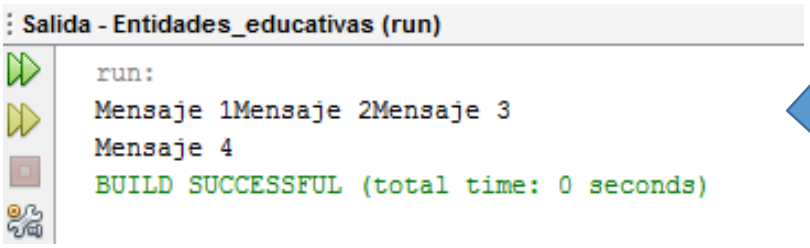
En el siguiente ejemplo se leen datos de varios tipos y se imprimen los resultados:

```
public static void main(String[] args)
{
   Scanner sc = new Scanner(System.in);
   int cont=0,ap=0,de=0;
   double promap=0,promde=0,sumap=0,sumde=0;
   String nombre;
   System.out.println("Ingrese su nombre :");
   nombre=sc.nextLine();
   System.out.println("Cuantas notas va a ingresar?");
   int c = sc.nextInt();
   while(cont!=c){
   System.out.println("Ingresa nota #"+(cont+1));
   int nota=sc.nextInt();
   if(nota>=7){
        sumap=sumap+nota;
        ap=ap+1;
   }else{
        sumde=sumde+nota;
        de=de+1;
   }
   cont++;
   }
   if (ap>0)
        promap=sumap/ap;
   if (de>0)
        promde=sumde/de;
   System.out.println("Resultados de : "+nombre);
   System.out.println("Total de notas aprobadas "+ap+".  Total de notas reprobadas "+de);
   System.out.println("Promedio de Notas superiores a 7 : "+
   (promap)+"  Promedio  de Notas inferiores a 7 : " +  (promde));
}
```

Diferencia entre print y println

El comando print imprime el contenido y no hace salto de línea, mientras que el println imprime el contenido y automáticamente realiza el salto de línea, es como si tuviera un \n al final de la línea de impresión, como veremos en el siguiente ejemplo:

```
System.out.print("Mensaje 1");
System.out.print("Mensaje 2");
System.out.println("Mensaje 3");
System.out.println("Mensaje 4");
```



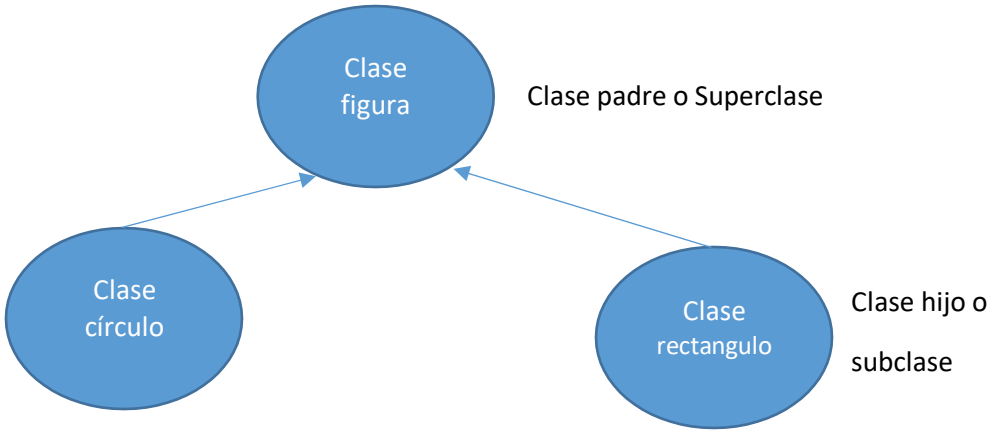
Aquí se realiza el salto de línea

2.CAPÍTULO: Herencia y Clases Abstractas

2.1. HERENCIA

Según Deitel(2004), la herencia es uno de los pilares de la POO es la reutilización del software en la que se crean nuevas clases basadas en otras ya existentes que mantienen cierta similitud en su estructura de forma general. Los programadores heredan código ya depurado de la clase padre lo cual significa un gran ahorro de tiempo además de otros beneficios como trabajar con programación ya optimizada, la subclase o clase hija se crea heredando atributos y métodos de la superclase o clase padre, y a la vez se agregan nuevos atributos y nuevos métodos a la clase especializada, a la vez se puede crear nuevas clases a partir también de la clase heredada haciendo más profundo el nivel de especialización. Las superclases tienden a ser "más generales" y las subclases "más específicas".

En el siguiente ejemplo se explica el concepto de herencia en forma gráfica:



En Java la herencia se usa con la palabra reservada EXTEND en la clase hija.

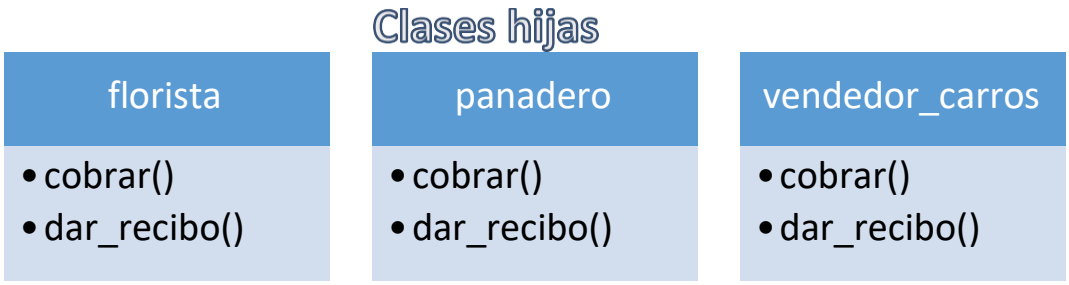
A continuación se muestran ejemplos de herencia:

Superclase o clase padre	**Subclase o clase hijo**
Estudiante	Estudiantes_Colegio, Estudiante_universitario
Figura	Círculo, Triangulo, Rectángulo, Esfera, Cubo
Préstamo	PrestamoCasa, PrestamoCarro, PrestamoSalud
Empleado	Docentes, Obreros, Administrativos
CuentaBancaria	CuentaCorriente, CuentaDeAhorros

En la clase derivada es posible:

- Añadir nuevos atributos y métodos propios de la clase hija
- Modificar los métodos heredados de la clase padre:
    - Refinar: agregar nuevas funcionalidades al método heredado
    - Reemplazar: se cambia totalmente el método heredado con nuevas funcionalidades.

La herencia se aplica entre clases que tienen comportamientos familiares de tal forma que puedan ser agrupadas en una clase padre y sus clases hijas lo hereden, cómo se muestra en el siguiente ejemplo:

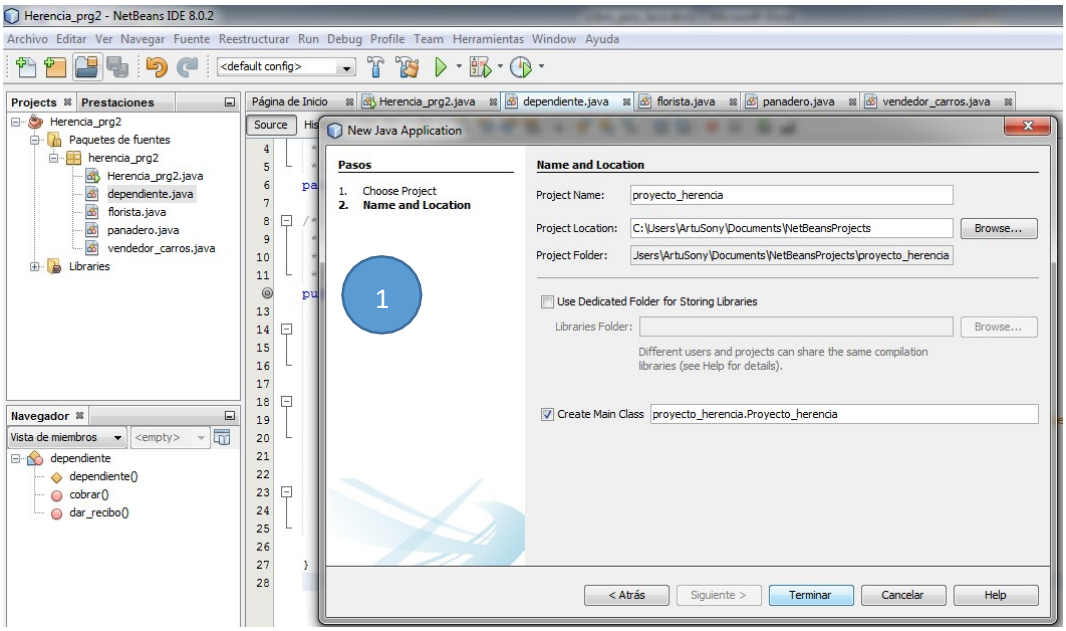


Todas estas clases tienen comportamientos similares, entonces creamos una clase padre que agrupe a todas.

Para crear esta solución debe realizar los siguientes pasos:

1. Crear el proyecto
2. Crear la clase padre : dependiente
3. Crear las clases hijas: florista
4. Crear las clases hijas: panadero
5. Crear las clases hijas: vendedor_carros
6. Crear los objetos en el main y llamar a los métodos

A continuación se detallan los pasos en imágenes:




```java
package herencia_prg2;

/**
 *
 * @author ArtuSony
 */
public class dependiente {
    public dependiente()
    {
        System.out.println("Objeto creado clase dependiente");
    }
    public void cobrar()
    {
        System.out.println("Estimado cliente, por favor acercarse a cancelar sus valores pendientes");
    }

    public void dar_recibo()
    {
        System.out.println("Gracias por su pago");
    }

}
```

2

Creamos los métodos en la clase padre y un constructor

```java
package herencia_prg2;

/**
 *
 * @author ArtuSony
 */
public class florista extends dependiente {
    public florista()
    {
        System.out.println("Objeto creado clase florista");
    }
```

3

Cada clase hija también tiene constructores

La clase A se debe relacionar con la clase B, si "A es un B" se cumple entonces si existe herencia

4

es un

```
package herencia_prg2;

/**
 *
 * @author ArtuSony
 */
public class panadero extends dependiente{
    public panadero()
    {
        System.out.println("Objeto creado clase panadero");
    }
```

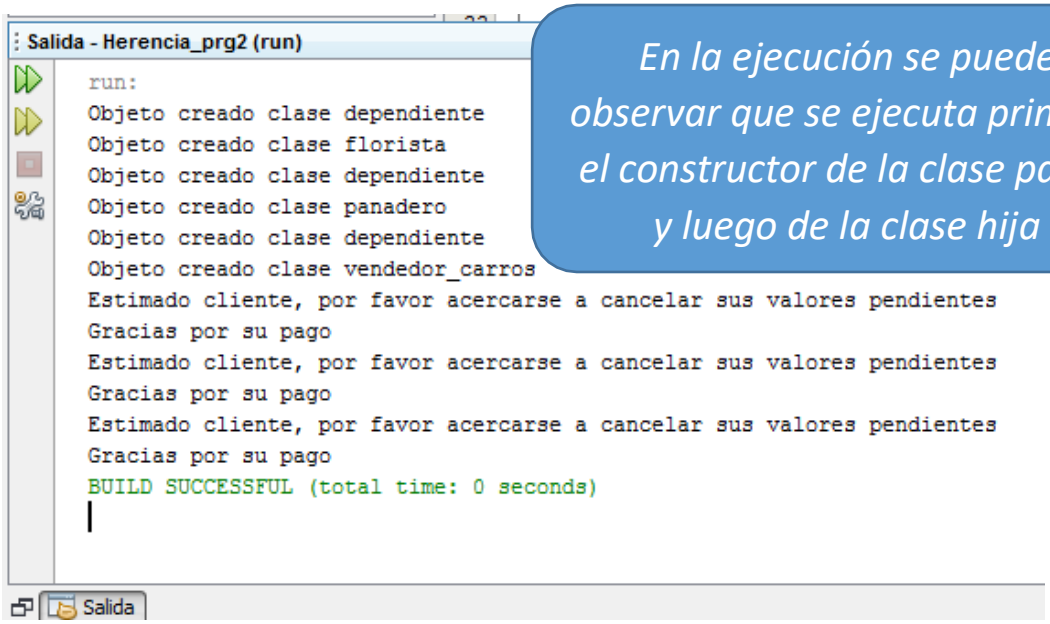
5

```
package herencia_prg2;

/**
 *
 * @author ArtuSony
 */
public class vendedor_carros  extends dependiente{
     public vendedor_carros()
    {
        System.out.println("Objeto creado clase vendedor_carros");
    }

}
```

6

```
public class Herencia_prg2 {


    public static void main(String[] args) {
        florista f=new florista();
        panadero p=new panadero();
        vendedor_carros v=new vendedor_carros();
        f.cobrar();
        f.dar_recibo();
        p.cobrar();
        p.dar_recibo();
        v.cobrar();
        v.dar_recibo();
    }
```

Se crea un objeto de cada clase y se llama a los métodos

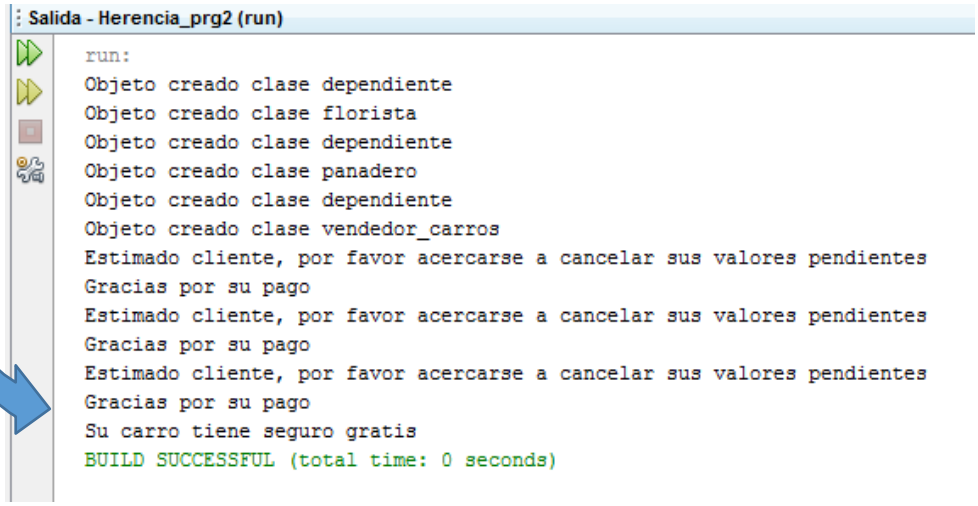


En la ejecución se puede observar que se ejecuta primero el constructor de la clase padre y luego de la clase hija

También es posible redefinir los métodos heredados y utilizar los métodos de la clase padre usando la palabra SUPER, como se muestra a continuación:

```
public class vendedor_carros  extends dependiente{
    public vendedor_carros()
    {
        System.out.println("Objeto creado clase vendedor_carros");
    }

    public void dar_recibo()
    {
        super.dar_recibo();
        System.out.println("Su carro tiene seguro gratis");
    }
```

La salida del método se redefine y el resultado es:



2.2.1. MÉTODOS VIRTUALES

Existen métodos especiales llamados virtuales, estos métodos se utilizan frecuentemente cuando se implementa la herencia y sirven para redefinir estos métodos en las clases hijas, en Java se utiliza la palabra virtual para crear estos métodos. En las subclases los métodos virtuales deben redefinirse con la palabra override, y es aquí donde se implementará la funcionalidad del método.

2.2. HERENCIA EN JAVA

En algunos lenguajes de programación se puede realizar herencia simple y herencia múltiple, pero en Java sólo se admite la herencia simple, es decir un hijo tiene un solo padre. La relación de herencia significa que ambas están relacionadas mediante la forma "es un", por ejemplo tenemos la clase derivada circulo y la clase padre figura, es posible afirmar que un circulo es una figura, entonces si existe entre ambas clases una relación de herencia.

Para indicar que una clase hereda de otra se utiliza la palabra reservada extends, siguiendo el ejemplo anterior seria: class circulo extends figura { … }

Al realizar la herencia entre clases, la clase hija hereda de la clase padre sus variables y métodos, en donde los métodos virtuales(virtual) pueden ser redefinidos(override) en la subclase, asi mismo pueden agregarse nuevos métodos y atributos aparte de los heredados, se puede crear tantas clases derivadas como se requiera.

Los métodos constructores de las clases padres pueden invocarse en las clases derivadas con la palabra super().

Ejemplo de herencia:

2.2.1. CLASE PADRE O SUPERCLASE

```
public class Persona {
   private String nombre;
   private String apellidos;
   private int edad;

   public Persona (String nombre, String apellidos, int edad) {
```

Atributos = datos de la clase

```
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;                }
```

Método constructor, se ejecuta automáticamente al crear el objeto. Lleva el mismo nombre de la clase sin ningún tipo

```
    public String getNombre () { return nombre; }
    public String getApellidos () { return apellidos; }
    public int getEdad () { return edad; }
}
```

Métodos que devuelven datos de la clase o modifican los atributos, también llamados métodos set y get

2.2.2. CLASE HIJA O SUBCLASE

Clase heredada. Agrega un atributo denominado IdDocente

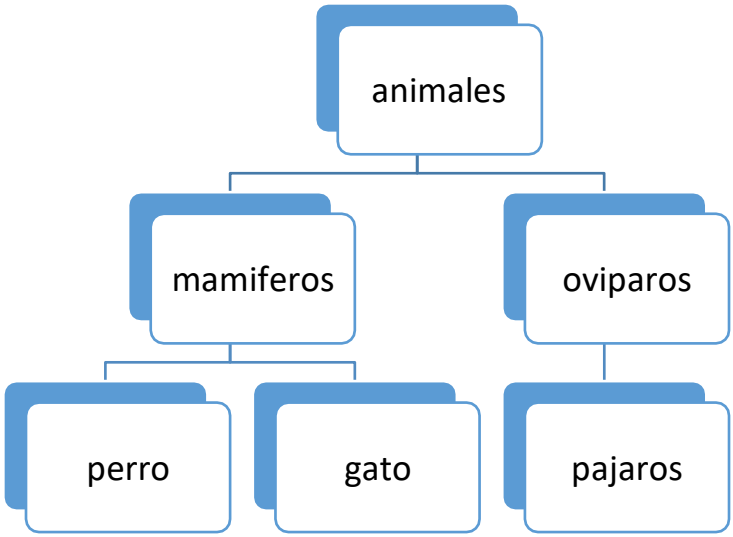
```
public class Docente extends Persona {
    private String IdDocente;
    public Docente (String nombre, String apellidos, int edad)
      {
        super(nombre, apellidos, edad);
        IdDocente = "No"; }
```

Constructor de clase derivada, envía datos al constructor de la clase padre usando la palabra super

```
    public void setId Docente (String IdDocente) { this.Id Docente = IdDocente; }
    public String getIdDocente () { return IdDocente;  }
    public void mostrarNombreApellidosYCarnet() {
          System.out.println ("Docente : " + getNombre() + " " + getApellidos() +
       " con Id de profesor: " + getIdDocente() ); }
}
```

Método de clase hija que usa los métodos creados en la clase padre

En Java solo se permite herencia simple, sin embargo, es posible crear jerarquías de clases muy extensas como se muestra en el siguiente ejemplo:

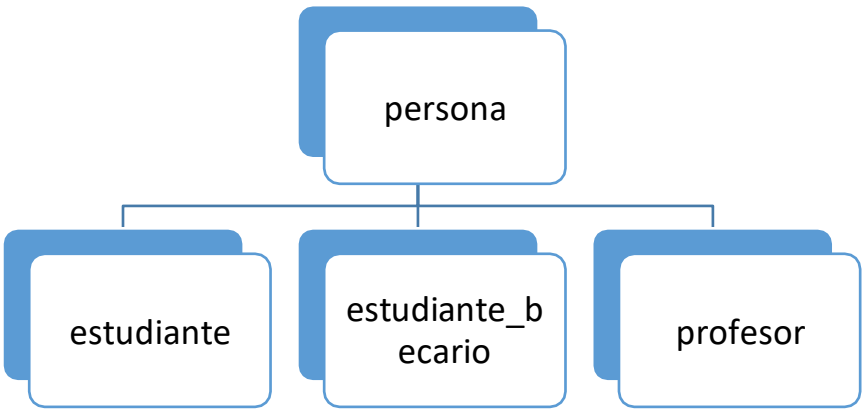


Public class Animales { … }

Public class Mamiferos extends Animales { … }

Public class Perro extends Mamiferos { … }

En el siguiente ejemplo de herencia se implementa la siguiente estructura de clases:



```java
package ejercicio;

public class persona {

    private String nombre;
    private String direccion;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getDireccion() {
        return direccion;
    }

    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }
}
```

Clase Persona

Atributos

Propiedad getNombre()
devuelve atributo nombre

Propiedades

```java
package ejercicio;
public class estudiante extends persona {

    private String escuela;
    private double grado;

    public String getEscuela() {
        return escuela;
    }

    public void setEscuela(String escuela) {
        this.escuela = escuela;
    }

    public double getGrado() {
        return grado;
    }

    public void setGrado(double grado) {
        this.grado = grado;
    }
}
```

Clase Heredada
Estudiante

```java
package ejercicio;
public class estudiante_becario extends estudiante {

    private String ciudad;

    public String getCiudad() {
        return ciudad;
    }

    public void setCiudad(String ciudad) {
        this.ciudad = ciudad;
    }
```

Clase Heredada Estudiante_becario

```java
package ejercicio;
public class profesor extends persona {

    private String asignatura;

    public String getAsignatura() {
        return asignatura;
    }

    public void setAsignatura(String asignatura) {
        this.asignatura = asignatura;
    }
}
```

Clase Heredada Profesor

Ejecución en main de las clases planteadas:

```
package ejercicio;
public class Ejercicio {
    public static void main(String[] args) {
        persona person1 = new persona();
        person1.setNombre("Nelly Esparza");

        estudiante estudiante1 = new estudiante();
        estudiante1.setNombre("Vicente Esparza");
        estudiante1.setEscuela("Astudillo");

        estudiante_becario estudiante_becario1 =
         new estudiante_becario();
        estudiante_becario1.setNombre("Arellys Moreno");
        estudiante_becario1.setEscuela("Ecomundo");
        estudiante_becario1.setCiudad("EEUU");

        profesor profesor1 = new profesor();
        profesor1.setNombre("Damaris Moreno");
        profesor1.setAsignatura("Arte");


        System.out.println("Datos de los objetos ...");
        System.out.println("  person1.getNombre() = " + person1.getNombre());
        System.out.println("  estudiante1.getNombre() = " + estudiante1.getNombre());
        System.out.println("  estudiante_becario1.getNombre() = " +
                estudiante_becario1.getNombre());
        System.out.println("  profesor1.getNombre() = " + profesor1.getNombre());
    }

}
```

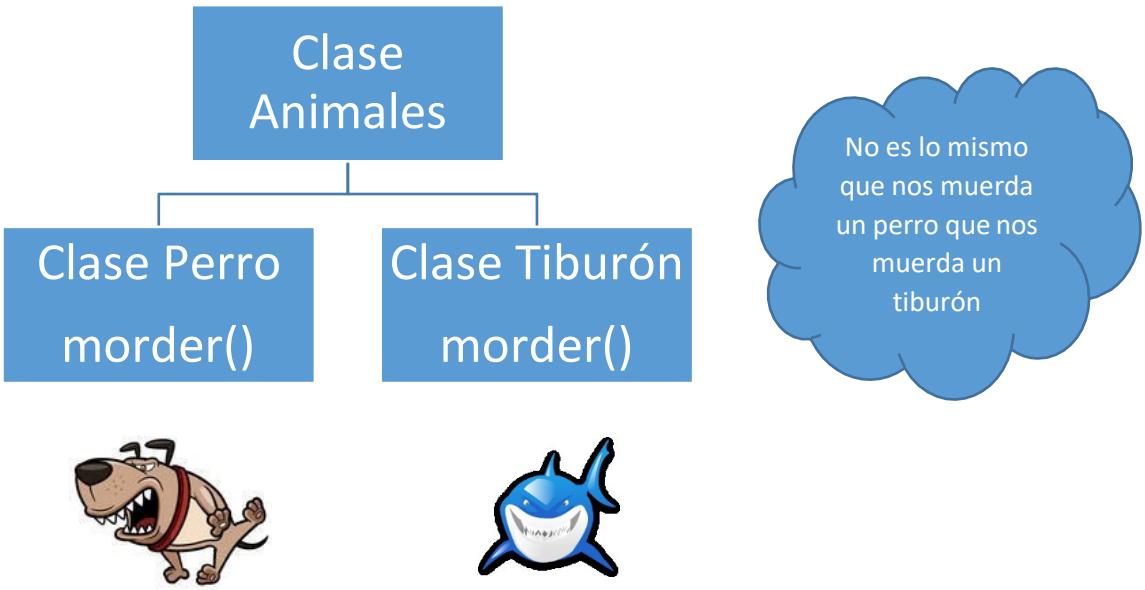
Las clases heredadas pueden usar atributos y métodos de clase padre

3.CAPÍTULO: Polimorfismo e Interfaces de clases.

3.1. CONCEPTO DE POLIMORFISMO

Según Bell(2003), el polimorfismo es uno de los cuatro pilares de la programación orientada a objetos, junto a la abstracción, encapsulación y herencia. Polimorfismo significa que varias clases pueden tener un método con el mismo nombre, pero este método tiene una funcionalidad distinta implementada internamente, de allí que "polimorfismo" significa múltiples formas, muchas veces también este concepto confunde con el de sobrecarga de métodos, pero no es lo mismo.

Para ejemplificar el concepto de polimorfismo, pensemos en la siguiente estructura de clases:

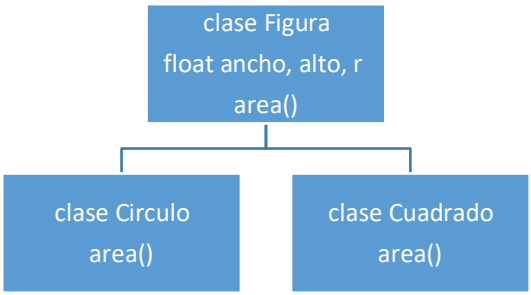


Se debe tener en cuenta lo siguiente para poder establecer que estamos usando polimorfismo:

1. El polimorfismo sólo es posible cuando se utiliza la relación de herencia entre dos clases, es decir, debe existir una clase padre y una clase hijo.
2. Las clases padres son abstractas y tienen métodos abstractos que serán implementados en las clases hijas
3. Las clases hijas redefinen los métodos usando la palabra reservada @override

4. Las clases padres e hijas pueden tener otros métodos que no son polimórficos, es decir, no se redefinen.

Para el ejemplo de la clase Figura se usará el siguiente diagrama de clases:



La clase figura es abstracta, es decir, no es posible instanciar objetos de la clase padre, sólo de las clases derivadas, el método polimórfico es área(), en donde se calculará el área de la figura dependiendo de la clase que lo invoque. A continuación se explicará la implementación de esta estructura de clases en lenguaje Java.

1. Crear la clase padre

```
/*
 * Ing. Nelly Esparza Cruz
 */
package ejercicio_polimorfismo;
public abstract class Figura {
    protected float alto,ancho, r;
    public abstract double area();
    }
```

Clase Figura de tipo abstracta con la implementación del método polimórfico área().

2. Crear la clase derivada Circulo

```
package ejercicio_polimorfismo;
public class Circulo extends Figura {
   private final double pi=3.1415;
   public Circulo(float radio)
    {
        r=radio;
    }

   @Override
    public double area()
    {
        return (pi)*(r*r);
    }
```

herencia

Constructor de clase

Redefinición de método polimórfico

3. Crear la clase derivada Cuadrado

```
package ejercicio_polimorfismo;
public class Cuadrado extends Figura {
    public Cuadrado(float alto, float ancho)
    {
        this.alto=alto;
        this.ancho=ancho;
    }
    @Override
    public double area() {
        return (ancho*alto);
    }

}
```

herencia

Constructor de clase

Redefinición de método polimórfico

4. Crear los objetos en el Main y llamar a los métodos

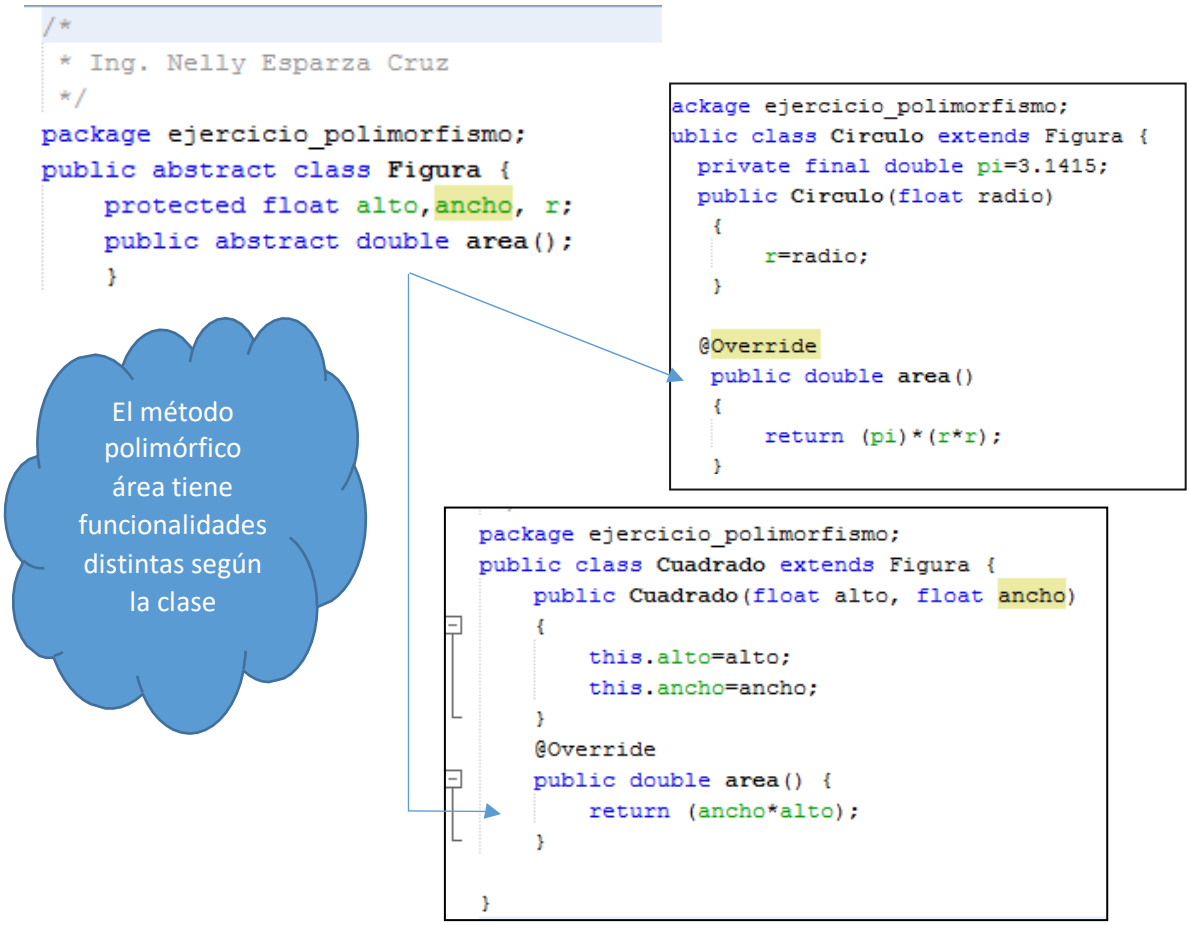
```
package ejercicio_polimorfismo;
public class Ejercicio_polimorfismo {

    public static void main(String[] args) {
        Figura cir=new Circulo(5);
        Figura cua=new Cuadrado(5,5);

        System.out.println("El area del circulo es : ");
        System.out.println(cir.area());
        System.out.println("El area del cuadrado es : ");
        System.out.println(cua.area());

    }

}
```

Clase abstracta. Clase padre

Clase derivada

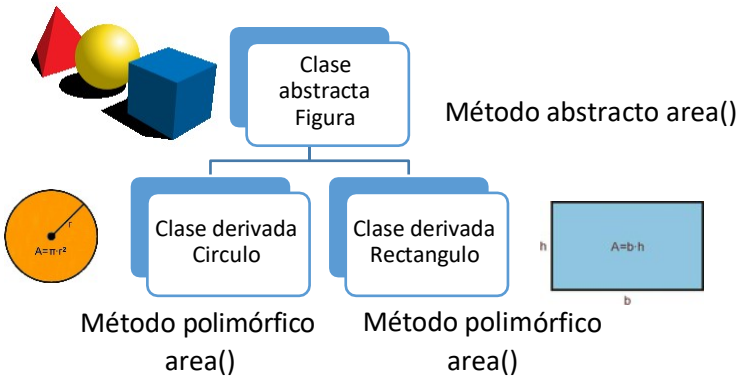
Ejecución de métodos

De manera general los códigos están relacionados de la siguiente manera:



3.1.1. DIFERENCIA ENTRE CLASES CONCRETAS Y ABSTRACTAS

Las clases abstractas son las que se crearon para ser heredadas por otras clases ya que no pueden ser instanciadas por si mismas, mientras que las clases concretas si pueden instanciarse para crear objetos. Un método abstracto debe pertenecer a una clase abstracta. En la siguiente imagen se muestra como ejemplo una clase abstracta figura y las clases derivadas círculo y rectángulo:



3.1.2. ANÁLISIS DE MÉTODOS ABSTRACTOS E INTERFACES

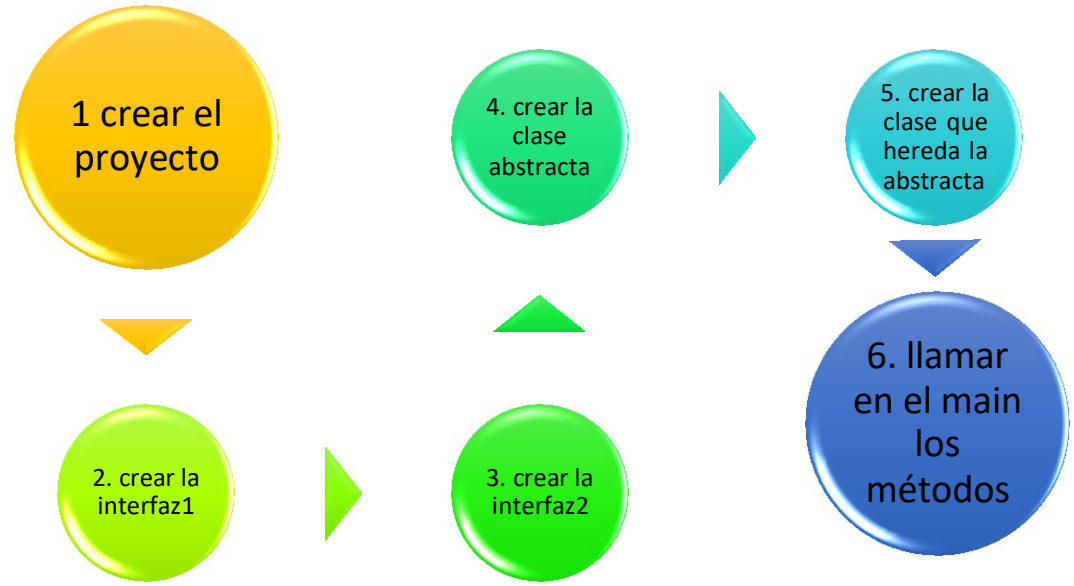
Según Sierra(2000), las interfaces son conjuntos de métodos declarados sin definiciones establecidas pueden incluir constantes, sin embargo, estas sólo pueden ser public, static y final. Las clases que implementan las interfaces tienen la obligación de crear la definición del método para que adquiera una conducta o funcionalidad. Las interfaces permiten que las clases puedan implementar varios comportamientos, es decir al tener más de una interfaz puede heredar más de un comportamiento.

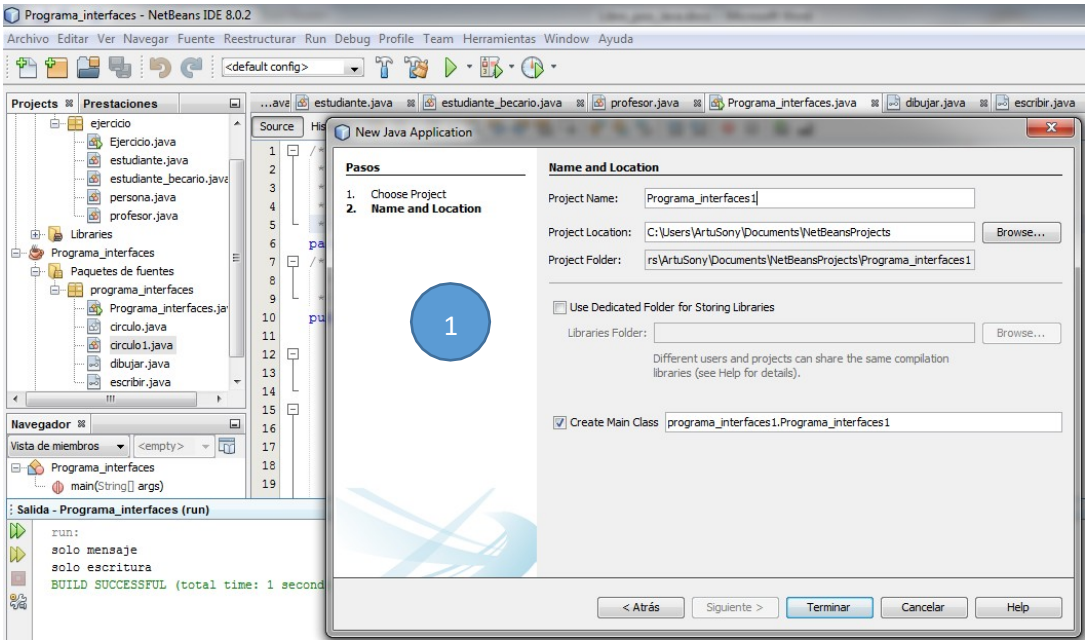
Las interfaces se asemejan de las clases abstractas en que ambas definen métodos que obligatoriamente se redefinen en las subclases, sin embargo, tienen diferencias entre ellas puesto que las interfaces simulan la posibilidad de herencia múltiple. Para declarar una interfaz se utiliza la palabra interface de la siguiente manera:

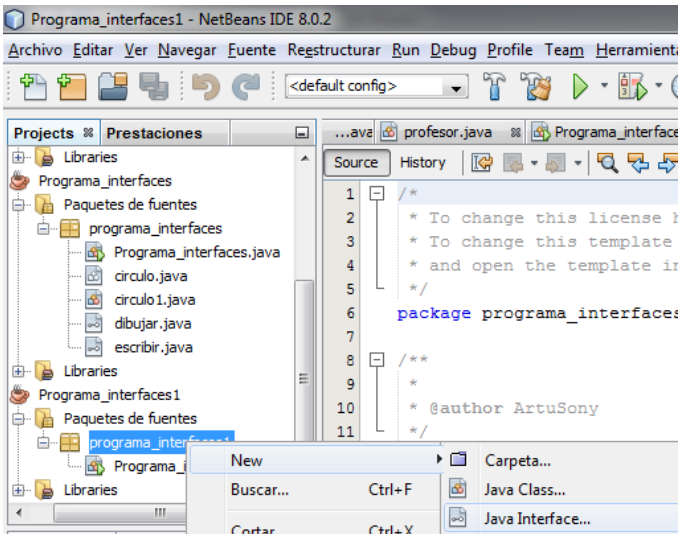
```
public interfaz Dibujar{
        public void setValorx(int x);
        public void setValory(int y);
}
```

A continuación desarrollaremos la implementación de una interfaz, aquí podremos observar como las interfaces nos permiten heredar de más de una clase, simulando la herencia múltiple.

Los pasos a seguir para crear este ejemplo son:








```java
package programa_interfaces;

/**
 *
 * @author ArtuSony
 */
public interface dibujar {
    public void mensaje();

}
```

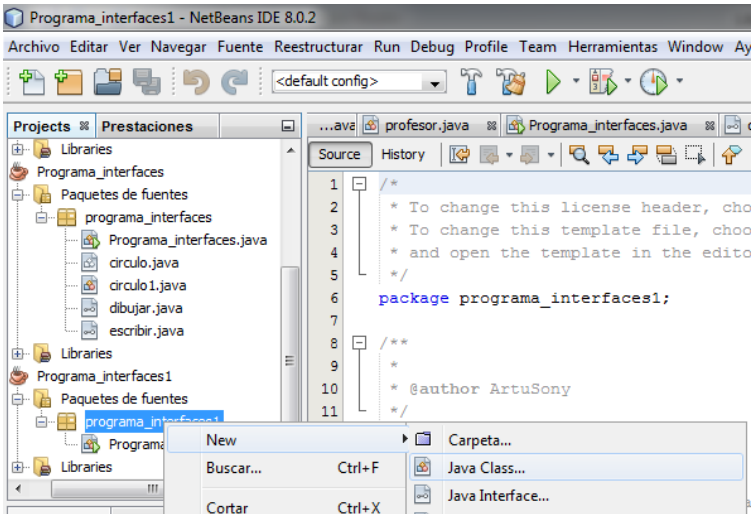
2

Crear la primera interfaz llamada dibujar con el método mensaje()

```java
package programa_interfaces;

/**
 *
 * @author Ing. Nelly Esparza
 */
public interface escribir {
    public void escritura();

}
```

3

Crear la segunda interfaz llamada escribir con el método escritura()



4

Crear la clase abstracta para implementar las dos interfaces

```java
package programa_interfaces;

/**
 *
 * @author Ing. Nelly Esparza
 */
abstract class circulo implements dibujar,escribir {
    public void mensaje()
    {
        System.out.println("solo mensaje");
    }
    public void escritura()
    {
        System.out.println("solo escritura");
    }

}
```

5

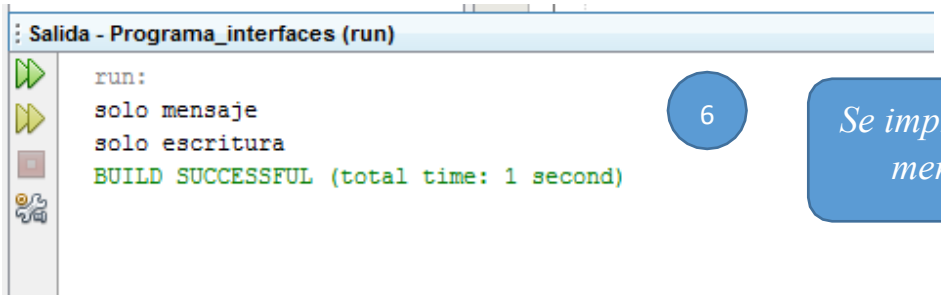
Crear la clase circulo para poder instanciar la clase abstracta

```java
package programa_interfaces;

/**
 *
 * @author Ing. Nelly Esparza
 */
public class circulo1 extends circulo {

}
```

```
package programa_interfaces;
/**
 * @author Ing. Nelly Esparza
 */
public class Programa_interfaces {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        circulo1 c=new circulo1();
        c.mensaje();
        c.escritura();
    }

}
```

6

Crear en el main el objeto de la clase circulo1 y llamar a los métodos



6

Se imprimen los mensajes

4.CAPÍTULO : Manejo de Excepciones

4.1. INTRODUCCIÓN

Una excepción es un evento que se desencadena cuando ocurre algo no inesperado durante la ejecución del programa, es necesario controlar que las aplicaciones puedan manejar estos errores en una forma óptima y esto es posible implementando excepciones en los fragmentos de códigos de las aplicaciones.

Un ejemplo muy usado para demostrar el uso de las excepciones es el controlar la división para 0 en las aplicaciones matemáticas, de tal forma que al ocurrir este evento el programa no tenga un comportamiento indeseado, sino que realice un procedimiento definido por el programador y la aplicación siga funcionando con total normalidad como si no hubiera pasado nada.

Las excepciones pueden ser de tres tipos:

1. Error.- Se producen cuando ocurren eventos graves como fallas de hardware, generalmente el programa no puede seguir funcionando después de ocurrir una excepción de esta categoría y debe cerrarse.
2. Exception.- Se usan para capturar errores controlables de programación.
3. RuntimeException.- Generalmente ocurren por problemas en la programación y deben ser controladas.

Los métodos relacionados a las excepciones son:

- string getMessage(), envía un mensaje asociado a la excepción
- string toString(), devuelve un string que describe la excepción
- void printStackTrace(), contiene el método donde ocurrió la excepción

La gestión de excepciones se realiza de la siguiente manera:

1. Manejando las excepciones en los fragmentos de códigos conflictivos
    a. Usando los bloques try { …. } catch { … }
2. Lanzamiento manual de excepciones, cuando el programador considera que debido a una situación específica debe lanzarse una excepción.
    a. Usando la instrucción throws

Para implementar las excepciones se deben implementar los bloques try..catch de la siguiente manera:

```
try {
  // código que será vigilado de posibles errores
}
catch (Exception e) {
  // código que se ejecuta si ocurre un error en el bloque try
}
finally  {
 // bloque de sentencias que se ejecutan siempre, ocurra o no el error
}
```

4.1.1. DEFINICIÓN DE BLOQUE TRY, CATCH Y FINALLY

TRY

El bloque try contiene las instrucciones que se quieran controlar y donde pude ocurrir un error que provoque una excepción, en el caso de que esto suceda se interrumpe la ejecución del bloque try y se pasa el control al bloque catch.

CATCH

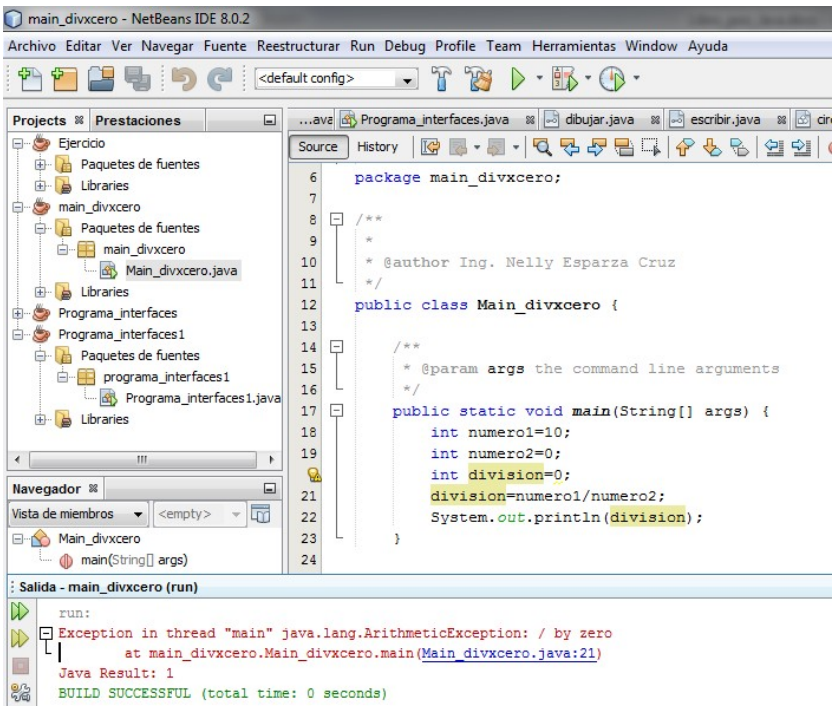
En el bloque catch se escriben las instrucciones que se ejecutarán si en el bloque try ocurre una excepción, en caso de no ocurrir ningún problema este bloque nunca se ejecuta.

FINALLY

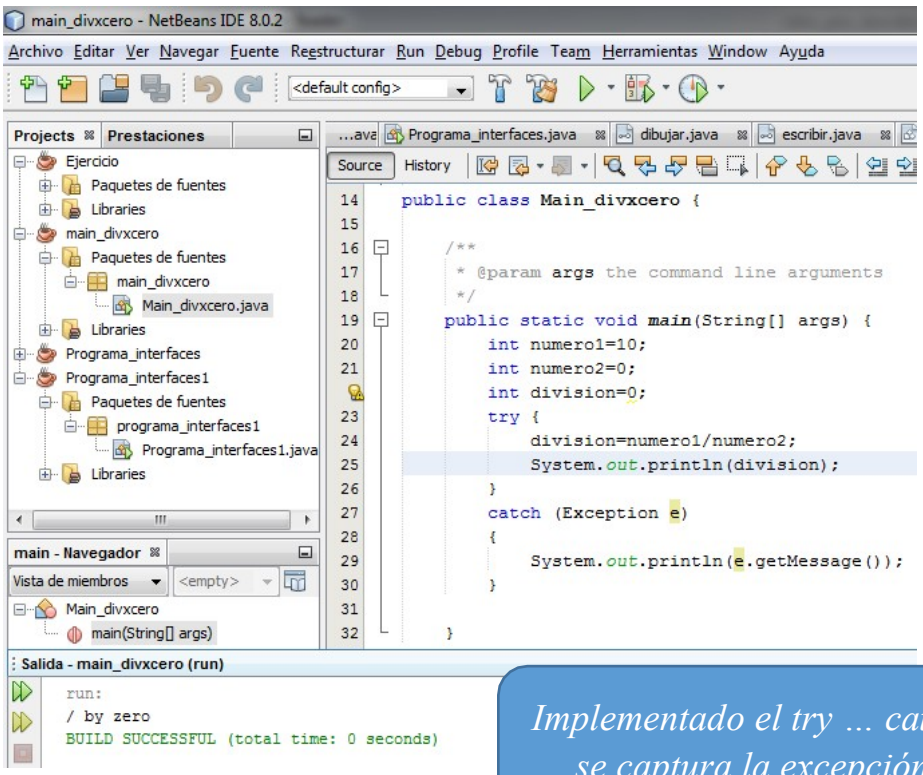
El bloque finally es opcional y se ejecuta siempre ocurra o no una excepción, sin embargo es importante puesto que los programadores lo usan para cerrar archivos o inicializar los valores de ciertos componentes, optimizando de esta forma las aplicaciones.

4.2. MANEJO DE EXCEPCIONES

En el ejemplo siguiente podemos observar un programa sin excepciones:



Una vez implementada las excepciones quedaría de la siguiente manera:

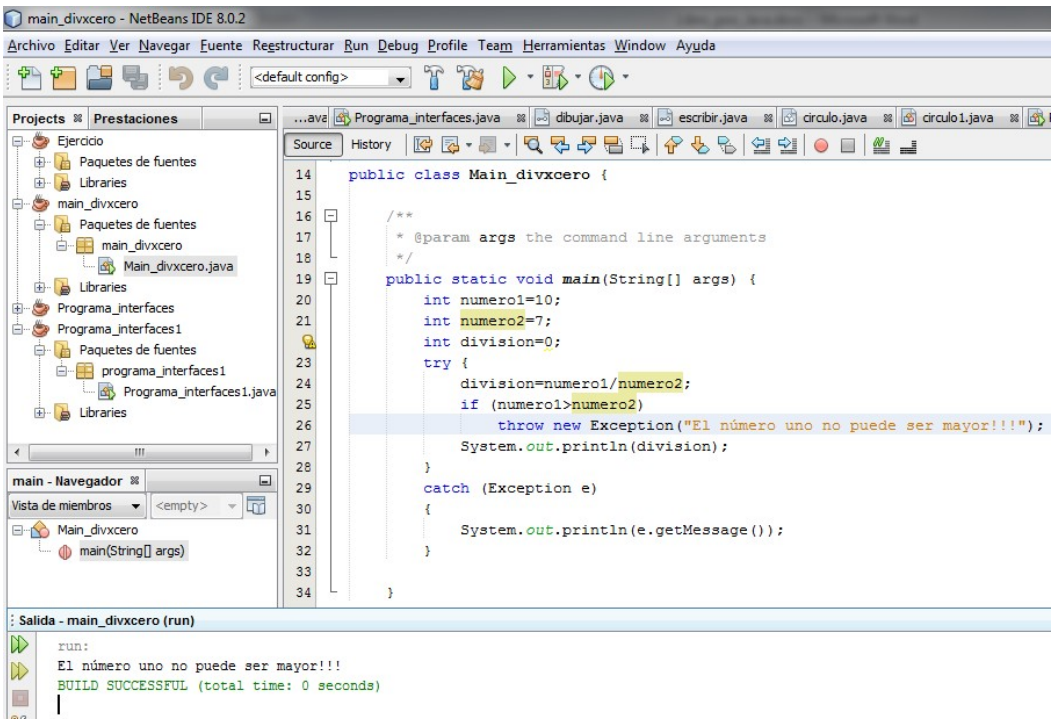


Implementado el try … catch, se captura la excepción

En el mismo ejemplo se lanzará una excepción creada por el programador cuando el número uno sea mayor que el número dos.

```
try {
    division=numero1/numero2;
    if (numero1>numero2)
        throw new Exception("El número uno no puede ser mayor!!!");
    System.out.println(division);
}
catch (Exception e)
{
    System.out.println(e.getMessage());
}
```

En el caso de que sea verdadero el if se lanza la excepción



El bloque finally es opcional, puede ir como no puede ir, veamos cómo se implementa:

```
try {
    division=numero1/numero2;
    if (numero1>numero2)
        throw new Exception("El número uno no puede ser mayor!!!");
    System.out.println(division);
}
catch (Exception e)
{
    System.out.println(e.getMessage());
}
finally {
    System.out.println("Gracias por usar el programa ;) ");
}
```

```
run:
El número uno no puede ser mayor!!!
Gracias por usar el programa ;)
BUILD SUCCESSFUL (total time: 0 seconds)
```

Bibliografía

- Aguilar, L. & Martínez, I. (2011). *Programación en Java : algoritmos, programación orientada a objetos e interfaz gráfica de usuario*. México: McGraw-Hill.
- Bell, D., Parr, M. & Elizondo, A. (2003). Java para estudiantes. México, D.F: Pearson Education.
- Deitel, H., Deitel, P. & Elizondo, A. (2004). *Cómo programar en Java*. México, D.F: Pearson/Educación.
- Domínguez, J. (2005). *El riesgo de mercado : su medición y control*. Madrid: Delta.
- Fernández, H. (2012). *Programación orientada a objetos usando java*. Bogota: Ecoe Ediciones.
- Garrido P.(2015). *Comenzando a programar con JAVA*. España: UNIVERSITAS.
- Goytia, J. & González, A. (2014). *Programación orientada a objetos C++ y Java : un acercamiento interdisciplinario*. México D.F: Larousse - Grupo Editorial Patria.
- Llavori, R. (2000). *Introducción a la programación con Pascal*. Castelló de la Plana: Universitat Jaume I.
- Llinas, L. (2010). *Todo lo básico que debería saber sobre programación orientada a objetos en Java*. Bogota: Ediciones de la U.
- Marín, A. & Montes, F. (2016). Aprende a programar con Java : un enfoque práctico partiendo de cero. Madrid: Paraninfo.
- Moreno, J. (2015). *Programación orientada a objetos*. Paracuellos de Jarama, Madrid: Ra-Ma.
- Pérez, J. (2015). *Programación orientada a objetos*. Madrid: RA-MA Editorial.
- Savitch, W., Elizondo, A., Carillo, A., Botello, F. & Luna. (2007). *Resolución de problemas con C*. México: Pearson Educación.
- Regino, E. (2015). *Lógica de programación orientada a objetos*. Bogota: Ecoe Ediciones.

- Pérez, J. (2015). Programación orientada a objetos. Madrid: RA-MA Editorial.
- Sierra, F. (2000). Java 2 : curso de programación (4a. ed. City: RA-MA Editorial.

Ing. Sist. Nelly Karina Esparza Cruz, MIE Ingeniera en Sistemas de la Universidad Técnica de Babahoyo, Magister en Administración de Empresas de la Universidad Técnica de Babahoyo, Magister en Informática Empresarial de UNIANDES.

Catedrática de la Universidad Técnica de Babahoyo desde el 2004. Siempre quiso ser Ingeniera en Sistemas, su principal desarrollo profesional ha sido en el campo de la programación. Desarrollador Independiente de Software, creadora del Software Integrado para Microempresas (SIM), sistema de facturación e inventario que se encuentra funcionando en muchos negocios a nivel nacional.

ISBN: 978-9942-8734-4-6

Universidad-Técnica-de-Babahoyo-Oficial
www.utb.edu.ec
km. 2.5 via Montalvo