Desarrollo de aplicaciones móviles I

Tema Nº2:POO en Java.

Indicador de logro Nº2:Aplica los conceptos de programación orientada a objetos con Java a través de casos prácticos.

**TEMA 01 Teoría de los**

Imagen que contiene Icono

Descripción generada automáticamente

**TEMA Nº2:**

POO en Java.

**Subtema 2.1:**

Funciones en Java

**Ejemplos:**

### La sentencia *if*

La sentencia **if**, actúa como cabría esperar. Si la condición es verdadera, la sentencia se ejecuta, de otro modo, se salta dicha sentencia, continuando la ejecución del programa con otras sentencias a continuación de ésta. La forma general de la sentencia **if** es:

if (condición){

sentencia;

}

Diagrama

Descripción generada automáticamente

Si el resultado del test es verdadero (**true**) se ejecuta la sentencia que sigue a continuación de **if**, en caso contrario, falso (**false**), se salta dicha sentencia, tal como se indica en la figura. La sentencia puede consistir a su vez, en un conjunto de sentencias agrupadas en un bloque.

if (condición){

sentencia1; sentencia2;

}

En el siguiente ejemplo, si el número del boleto que hemos adquirido coincide con el número aparecido en el sorteo, nos dicen que hemos obtenido un premio.

if(numeroBoleto==numeroSorteo) System.out.println("has obtenido un premio");

**La sentencia *if...else***

La sentencia **if...else** completa la sentencia **if**, para realizar una acción alternativa

if (condición)

sentencia1;

else

sentencia2;

Diagrama

Descripción generada automáticamente

Las dos primeras líneas indican que si la condición es verdadera se ejecuta la sentencia

1. La palabra clave **else**, significa que si la condición no es verdadera se ejecuta la sentencia 2, tal como se ve en la figura.

Dado que las sentencias pueden ser simples o compuestas la forma general de **if...else** es

if (condición){

sentencia1; sentencia2;

}

else{

sentencia3 sentencia4; sentencia5

}

Existe una forma abreviada de escribir una sentencia condicional **if...else** como la siguiente:

if (numeroBoleto==numeroSoreteo)

premio=1000;

else

premio=0;

en una sola línea

premio=(numeroBoleto==numeroSoreteo) ? 1000 : 0;

Un ejemplo significativo es el siguiente: el signo de un número elevado a una potencia par es positivo, y es negativo cuando está elevado a una potencia impar.

int signo=(exponente%2==0)?1:-1;

La condición entre paréntesis es la siguiente: un número es par, cuando el resto de la división entera de dicho número entre dos vale cero.

**La sentencia *switch***

Como podemos ver en la figura del apartado anterior, la sentencia **if...else** tiene dos ramas, el programa va por una u otra rama dependiendo del valor verdadero o falso de la expresión evaluada. A veces, es necesario, elegir entre varias alternativas, como se muestra en la siguiente figura

Diagrama

Descripción generada automáticamente

Por ejemplo, considérese las siguientes series de sentencias **if.** else

if(expresion==valor1)

sentencia1;

else if(expresion==valor2) sentencia2;

else if(expresion==valor3) sentencia3;

else

sentencia4;

El código resultante puede ser difícil de seguir y confuso incluso para el programador avanzado. El lenguaje Java proporciona una solución elegante a este problema mediante la sentencia condicional **switch** para agrupar a un conjunto de sentencias **if...else**.

switch(expresion){

case valor1:

sentencia1;

break; //sale de switch case valor2:

sentencia2;

break; //sale switch case valor3:

sentencia3;

break; //sale de switch default:

sentencia4;

}

En la sentencia **switch**, se compara el valor de una variable o el resultado de evaluar una expresión, con un conjunto de números enteros *valor1*, *valor2*, *valor3*, .. o con un conjunto de caracteres, cuando coinciden se ejecuta el bloque de sentencias que están asociadas con dicho número o carácter constante. Dicho bloque de sentencias no está entre llaves sino que empieza en la palabra reservada **case** y termina en su asociado **break**. Si el compilador no encuentra coincidencia, se ejecuta la sentencia **default**, si es que está presente en el código.

Veamos ahora un ejemplo sencillo: dado el número que identifica al mes (del 1 al 12) imprimir el nombre del mes.

public class SwitchApp1 {

public static void main(String[] args) {

int mes=3;

switch (mes) {

case 1: System.out.println("Enero"); break;

case 2: System.out.println("Febrero"); break;

case 3: System.out.println("Marzo"); break;

case 4: System.out.println("Abril"); break;

case 5: System.out.println("Mayo"); break;

case 6: System.out.println("Junio"); break;

case 7: System.out.println("Julio"); break;

case 8: System.out.println("Agosto"); break;

case 9: System.out.println("Septiembre"); break;

case 10: System.out.println("Octubre"); break;

case 11: System.out.println("Noviembre"); break;

case 12: System.out.println("Diciembre"); break;

default: System.out.println("Este mes no existe"); break;

}

}

}

Ahora un ejemplo más complicado, escribir un programa que calcule el número de días de un mes determinado cuando se da el año.

Anotar primero, los meses que tienen 31 días y los que tienen 30 días. El mes de Febrero (2º mes) es el más complicado ya que tiene 28 días excepto en los años que son bisiestos que tiene 29. Son bisiestos los años múltiplos de cuatro, que no sean múltiplos de 100, pero si son bisiestos los múltiplos de 400.

public class SwitchApp2 {

public static void main(String[] args) { int mes=2;

int año=1992; int numDias=30; switch (mes) {

case 1:

case 3:

case 5:

case 7:

case 8:

case 10:

case 12:

numDias = 31; break;

case 4:

case 6:

case 9:

case 11:

numDias = 30; break;

case 2:

if ( ((año % 4 == 0) && !(año % 100 == 0)) || (año % 400 == 0) ) numDias = 29;

else

numDias = 28;

break; default:

System.out.println("Este mes no existe"); break;

}

System.out.println("El mes "+mes+" del año "+año+" tiene "+numDias+" días");

}

}

**For, While, Do…. While, Break, Continue**

Tan importantes como las sentencias condiciones son las sentencias iterativas o repetitivas. Normalmente, las sentencias de un programa son ejecutadas en el orden en el que aparecen. Cada sentencia es ejecutada una y solamente una vez. El lenguaje Java, como la mayoría de los lenguajes, proporciona sentencias que permiten realizar una tarea una y otra vez hasta que se cumpla una determinada condición, dicha tarea viene definida por un conjunto de sentencias agrupadas en un bloque.

Las sentencias iterativas son **for, while** y **do** **while**

### La sentencia *for*

Esta sentencia se encuentra en la mayoría de los lenguajes de programación. El bucle **for** se empleará cuando conocemos el número de veces que se ejecutará una sentencia o un bloque de sentencias, tal como se indica en la figura. La forma general que adopta la sentencia **for** es

for(inicialización; condición; incremento) sentencia;

Diagrama

Descripción generada automáticamente

El primer término *inicialización*, se usa para inicializar una variable índice, que controla el número de veces que se ejecutará el bucle. La *condición* representa la condición que ha de ser satisfecha para que el bucle continúe su ejecución.

El *incremento* representa la cantidad que se incrementa la variable índice en cada repetición.

Ejemplo: Escribir un programa que imprima los primeros 10 primeros números enteros empezando por el cero

for (int i = 0; i < 10; i++)

{ System.out.println(i);

}

El resultado será: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

La variable índice *i* se declara y se inicializa en el término *inicialización*, la *condición* se expresa de modo que *i* se debe mantener estrictamente menor que 10; la variable *i* se incrementa una unidad en cada repetición del bucle. La variable *i* es local al bucle, por lo que deja de existir una vez que se sale del bucle.

Ejemplo: Escribir un programa que imprima los números pares positivos menores o iguales que 20

for (int i=2; i <=20; i += 2)

{ System.out.println(i);

}

Ejemplo: Escribir un programa que imprima los números pares positivos menores o iguales que 20 en orden decreciente

for (int i=20; i >= 2; i -= 2)

{ System.out.println(i);

}

Ejemplo: Escribir un programa que calcule el factorial de un número empleando la sentencia iterativa **for**. Guardar el resultado en un número entero de tipo **long** . Definición: el factorial de un número *n* es el resultado del producto *1\*2\*3\*………* *\*(n-1)\*n*.

public class FactorialApp {

public static void main(String[] args) { int numero=4;

long resultado=1;

for(int i=1; i<=numero; i++)

{ resultado\*=i;

}

System.out.println("El factorial es "+resultado);

}

}

**La sentencia *while***

A la palabra reservada **while** le sigue una condición encerrada entre paréntesis. El bloque de sentencias que le siguen se ejecuta siempre que la condición sea verdadera tal como se ve en la figura. La forma general que adopta la sentencia **while** es:

while (condición)

sentencia;

Diagrama

Descripción generada automáticamente

Ejemplo: Escribir un programa que imprima los primeros 10 primeros números enteros empezando por el cero, empleando la sentencia iterativa *while*.

int i=0;

while (i<10) {

System.out.println(i); i++;

}

El valor inicial de *i* es cero, se comprueba la condición *(i<10)*, la cual resulta verdadera. Dentro del bucle, se imprime *i*, y se incrementa la variable contador *i*, en una unidad. Cuando *i* vale 10, la condición *(i<10)* resulta falsa y el bucle ya no se ejecuta. Si el valor inicial de *i* fuese 10, no se ejecutaría el bucle. Por tanto, el bucle **while** no se ejecuta si la condición es falsa.

Ejemplo: escribir un programa que calcule el factorial de un número empleando la sentencia iterativa **while**

public class FactorialApp1 {

public static void main(String[] args) { int numero=4;

long resultado=1;

while(numero>0){

resultado\*=numero; numero--;

}

System.out.println("El factorial es "+resultado);

}

}

**La sentencia *do...while***

Como hemos podido apreciar las sentencias for y while la condición está al principio del bucle, sin embargo, do...while la condición está al final del bucle, por lo que el bucle se ejecuta por lo menos una vez tal como se ve en la figura. do marca el comienzo del bucle y while el final del mismo. La forma general es:

do{

sentencia;

}while(condición);

Diagrama

Descripción generada automáticamente

Ejemplo: Escribir un programa que imprima los primeros 10 primeros números enteros empezando por el cero, empleando la sentencia iterativa *do..while*.

int i=0;

do{

System.out.println(i); i++;

}while(i < 10);

El bucle **do...while**, se usa menos que el bucle **while**, ya que habitualmente evaluamos la expresión que controla el bucle al comienzo, no al final.

**La sentencia *break***

A veces es necesario interrumpir la ejecución de un bucle **for**, **while**, o **do...while**.

for(int i = 0; i < 10; i++){

if (i == 8)break;

System.out.println(i);

}

Consideremos de nuevo el ejemplo del bucle **for**, que imprime los 10 primeros números enteros, se interrumpe la ejecución del bucle cuando se cumple la condición de que la variable contador *i* valga 8. El código se leerá: "salir del bucle cuando la variable contador *i*, sea igual a 8".

Como podemos apreciar, la ejecución del bucle finaliza prematuramente. Quizás el lector pueda pensar que esto no es de gran utilidad pues, el código anterior es equivalente a

for(int i = 0; i <=8; i++)

System.out.println(i);

Sin embargo, podemos salir fuera del bucle prematuramente si se cumple alguna condición de finalización.

while(true){

if (condicionFinal) break;

//...otras sentencias

}

Como podemos apreciar en esta porción de código, la expresión en el bucle **while** es siempre verdadera, por tanto, tiene que haber algún mecanismo que nos permita salir del bucle. Si la condicion de finalización es verdadera, es decir la variable *condicionFinal* del tipo **boolean** toma el valor **true**, se sale del bucle, en caso contrario se continua el procesamiento de los datos.

**La sentencia *continue***

La sentencia **continue**, fuerza al bucle a comenzar la siguiente iteración desde el principio. En la siguiente porción de código, se imprimen todos los números del 0 al 9 excepto el número 8.

for(int i = 0; i < 10; i++){

if (i == 8)

continue;

System.out.println(i);

}

**Etiquetas**

Tanto **break** como **continue** pueden tener una etiqueta opcional que indica a Java hacia donde dirigirse cuando se cumple una determinada condición.

### salida:

for(int i=0; i<20; i++){

while(j<70){

if(i\*j==500)

break salida;

//...

}

//...

}

La etiqueta en este ejemplo se denomina *salida*, y se añade antes de la parte inicial del ciclo. La etiqueta debe terminar con el carácter dos puntos **:**. Si no disponemos de etiqueta, al cumplirse la condición *i\*j==500*, se saldría del bucle interno **while**, pero el proceso de cálculo continuaría en el bucle externo **for**.

### Ejemplo: los números primos

Escribir un programa que calcule los números primos comprendidos entre 3 y 100.

Los números primos tienen la siguiente característica: un número primo es solamente divisible por sí mismo y por la unidad, por tanto, un número primo no puede ser par excepto el 2. Para saber si un número impar es primo, dividimos dicho número por todos los números impares comprendidos entre 3 y la mitad de dicho número. Por ejemplo, para saber si 13 es un número primo basta dividirlo por 3, y 5. Para saber si 25 es número primo se divide entre 3, 5, 7, 9, y 11. Si el resto de la división (operación módulo %) es cero, el número no es primo.

public class PrimosApp {

public static void main(String[] args) {

boolean bPrimo;

System.out.println("Números primos comprendidos entre 3 y 100");

for(int numero=3; numero<100; numero+=2){

bPrimo=true;

for(int i=3; i<numero/2; i+=2){

if(numero%i==0){

bPrimo=false; break;

}

}

if(bPrimo){

System.out.print(numero+" - ");

}

}

}

}

En primer lugar, hacemos un bucle **for** para examinar los números impares comprendidos entre 3 y 100.

Hacemos la suposición de que *numero* es primo, es decir, de que la variable de control *bPrimo* toma el valor **true**. Para confirmarlo, se halla el resto de la división entera entre *numero*, y los números *i* impares comprendidos entre 3 y *numero*/2. (Se recordará que todo número es divisible por la unidad). Si el número *numero* es divisible por algún número *i* (el resto de la división entera *numero%i* es cero), entonces el número *numero* no es primo, se abandona el bucle (**break**) con la variable de control *bPrimo* tomando el valor **false**. En el caso de que *numero* sea un número primo, se completa el bucle interno, tomando la variable de control *bPrimo* el valor inicial **true**.

Por último, si el número es primo, *bPrimo* es **true**, se imprime en la ventana, uno a continuación del otro separados por un guión.

En este programa podemos observar la diferencia entre *print* y *println*. El sufijo *ln* en la segunda función indica que se imprime el argumento y a continuación se pasa a la línea siguiente.

**Subtema 2.2:**

Clases en Java.

Una clase es una plantilla que define la forma de un objeto. Especifica los datos y el código que operará en esos datos. Java usa una especificación de clase para construir objetos. Los objetos son instancias de una clase. Por lo tanto, **una clase es esencialmente un conjunto de planes que especifican cómo construir un objeto**.

Cuando defines una clase, declaras su forma y naturaleza exactas. Para ello, especifique las variables de instancia que contiene y los métodos que operan en ellas. Aunque las clases muy simples pueden contener solo métodos o solo variables de instancia, la mayoría de las clases del mundo real contienen ambas.

**Ejemplos:**

class NombreClase{

//Declarar variables de instancia

tipo var1;

tipo var2;

//..

//Declarar métodos

tipo metodo1(parámetros){

//Cuerpo del método

}

tipo metodo2(parámetros){

//Cuerpo del método

}

}

Aunque no existe una regla sintáctica que se deba cumplir, **una clase bien diseñada debería definir una y solo una entidad lógica**.

**Subtema 2.3:**

Métodos en Java

**Los métodos son subrutinas que manipulan los datos definidos por la clase** y, en muchos casos, brindan acceso a esos datos. En la mayoría de los casos, otras partes de tu programa interactuarán con una clase a través de sus métodos.

Un método contiene una o más declaraciones. En un código Java bien escrito, cada método **realiza solo una tarea**. Cada método tiene un nombre, y es este el que se usa para llamar al método. En general, puede dar el nombre que desee a un método cualquiera. Sin embargo, recuerde que main() está reservado para el método que comienza ejecución de su programa. Además, no use las palabras claves de Java para nombres de métodos.

**Ejemplos:**

Un método tendrá paréntesis después de su nombre. Por ejemplo, si el nombre de un método es *getval*, se escribirá *getval()* cuando su nombre se usa en una sentencia. Esta notación lo ayudará a distinguir los nombres de las variables de los nombres de los métodos.

La forma general de un método se muestra a continuación:

/\* Un programa que usa la clase Vehiculo

El archivo se llama DemoVehiculo.java

\*/

class Vehiculo {

int pasajeros; //números de pasajeros

int capacidad; //capacidad del combustible en galones

int mpg; //combustible consumido en millas por galon

}

//Esta clase declara un objeto de tipo Vehiculo

class DemoVehiculo {

public static void main(String[] args) {

Vehiculo minivan = new Vehiculo();

int rango;

//asignando valores a los campos de minivan

minivan.pasajeros = 9;

minivan.capacidad = 15;

minivan.mpg = 20;

//Calcular el rango asumiendo un tanque lleno

rango = minivan.capacidad \* minivan.mpg;

System.out.println("La Minivan puede llevar " + minivan.pasajeros + " pasajeros con un rango de " + rango + " millas");

}

}

**Subtema 2.4:**

Herencia en Java

Podemos definir la herencia como la capacidad de crear clases que adquieren de manera automática los miembros (atributos y métodos) de otras clases que ya existen, pudiendo al mismo tiempo añadir atributos y métodos propios.

**Ventajas de la Herencia**

Entre las principales ventajas que ofrece la herencia en el desarrollo de aplicaciones, están:

* **Reutilización del código**: En aquellos casos donde se necesita crear una clase que, además de otros propios, deba incluir los métodos definidos en otra, la herencia evita tener que reescribir todos esos métodos en la nueva clase.
* **Mantenimiento de aplicaciones existentes**: Utilizando la herencia, si tenemos una clase con una determinada funcionalidad y tenemos la necesidad de ampliar dicha funcionalidad, no necesitamos modificar la clase existente (la cual se puede seguir utilizando para el tipo de programa para la que fue diseñada) sino que podemos crear una clase que herede a la primera, adquiriendo toda su funcionalidad y añadiendo la suya propia.

**Ejemplos:**

//Clase para objetos de dos dimensiones

class DosDimensiones{

double base;

double altura;

void mostrarDimension(){

System.out.println("La base y altura es: "+base+" y "+altura);

}

}

//Una subclase de DosDimensiones para Triangulo

class Triangulo extends DosDimensiones{

String estilo;

double area(){

return base\*altura/2;

}

void mostrarEstilo(){

System.out.println("Triangulo es: "+estilo);

}

}

class Lados3{

public static void main(String[] args) {

Triangulo t1=new Triangulo();

Triangulo t2=new Triangulo();

t1.base=4.0;

t1.altura=4.0;

t1.estilo="Estilo 1";

t2.base=8.0;

t2.altura=12.0;

t2.estilo="Estilo 2";

System.out.println("Información para T1: ");

t1.mostrarEstilo();

t1.mostrarDimension();

System.out.println("Su área es: "+t1.area());

System.out.println();

System.out.println("Información para T2: ");

t2.mostrarEstilo();

t2.mostrarDimension();

System.out.println("Su área es: "+t2.area());

}

}

**Subtema 1.5:**

Sobrecarga de métodos en Java.

En general, para sobrecargar un método, simplemente declare versiones diferentes de él. El compilador se ocupa del resto. Debe observar una restricción importante: el **tipo** y/o el **número de parámetros** de cada método sobrecargado debe diferir. No es suficiente que dos métodos difieran solo en sus tipos de devolución.

**Ejemplos:**

// Programa de demostración de sobrecarga de métodos

class Sobrecarga{

void demoSobrec(){

System.out.println("Sin parámetros\n");

}

//Sobrecargando demoSobrec para un parámetro int

void demoSobrec(int a){

System.out.println("Un parámetro: " +a+"\n");

}

//Sobrecargando demoSobrec para dos parámetros int

int demoSobrec(int a, int b){

System.out.println("Dos parámetros: "+a+", "+b);

return a+b;

}

//Sobrecargando demoSobrec para dos parámetros double

double demoSobrec(double a, double b){

System.out.println("Dos parámetros tipo double: "+a+", "+b);

return a+b;

}

}

class DemoSobrecarga{

public static void main(String[] args) {

Sobrecarga sc= new Sobrecarga();

int sumaint;

double sumadouble;

//Llamando todas las versiones de demoSobrec

sc.demoSobrec();

sc.demoSobrec(2);

sumaint=sc.demoSobrec(4,6);

System.out.println("Resultado de demoSobrec(4,6) es: "+sumaint+"\n");

sumadouble=sc.demoSobrec(1.1,2.2);

System.out.println("Resultado de demoSobrec(1.1,2.2) es: "+sumadouble);

}

}

**Subtema 1.6:**

Polimorfismo en Java.

Polimorfismo es la capacidad que tienen los objetos de una clase en ofrecer respuesta distinta e independiente en función de los parámetros (diferentes implementaciones) utilizados durante su invocación. Dicho de otro modo el objeto como entidad puede contener valores de diferentes tipos durante la ejecución del programa.

En JAVA el término polimorfismo también suele definirse como ‘Sobrecarga de parámetros’, que así de pronto no suena tan divertido pero como veremos más adelante induce a cierta confusión. En realidad suele confundirse con el tipo de poliformismo más común, pero no es del todo exacto usar esta denominación.

**Ejemplos:**

class Animal {

public void makeSound() {

System.out.println("Grr...");

}

}

class Cat extends Animal {

public void makeSound() {

System.out.println("Meow");

}

}

class Dog extends Animal {

public void makeSound() {

System.out.println("Woof");

}

}

**Subtema 1.7:**

Interfaces de Java.

Una interfaz () es sintácticamente similar a una clase abstracta, en la que puede especificar uno o más métodos que no tienen cuerpo ({}). Esos métodos deben ser implementados por una clase para que se definan sus acciones.

Por lo tanto, una interfaz especifica qué se debe hacer, pero no cómo hacerlo. Una vez que se define una interfaz, cualquier cantidad de clases puede implementarla. Además, una clase puede implementar cualquier cantidad de interfaces.

Para implementar una interfaz, una clase debe proporcionar cuerpos (implementaciones) para los métodos descritos por la interfaz. Cada clase es libre de determinar los detalles de su propia implementación. Dos clases pueden implementar la misma interfaz de diferentes maneras, pero cada clase aún admite el mismo conjunto de métodos. Por lo tanto, el código que tiene conocimiento de la interfaz puede usar objetos de cualquier clase, ya que la interfaz con esos objetos es la misma.

**Ejemplos:**

public interface Series {

int getSiguiente(); //Retorna el siguiente número de la serie

void reiniciar(); //Reinicia

void setComenzar(int x); //Establece un valor inicial

}

**Actividad:**

1. CUESTIONARIO TÉCNICO

* Indique cuándo utilizar la estructura de control if…else y cuándo la estructura switch
* Desarrolle un caso práctico donde se aplique la sobrecarga de métodos
* Cite un ejemplo práctico de cuándo utilizar arreglos multidimensionales
* Mencione por qué se utiliza un arreglo en vez de utilizar muchas variables para almacenar datos
* Es conveniente gestionar las excepciones al desarrollar o sencillamente complejiza el proceso de implementación; desarrolle su argumentación

1. CONCLUSIONES DE LA EXPERIENCIA

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_