# Estructuras de programación

## 1. Estructuras de Programación
### 1.1.Introducción

En la estructura de un programa java destacan algunos elementos, como son: *Comentarios, Palabras reservadas, Sentencias, Bloques, Clases, Métodos, el método main, entre otras que se detallan a continuación.*

***Comentarios***

Los *comentarios* sirven para documentar los programas y en ellos se escriben anotaciones sobre cómo funciona el programa o sobre cómo se ha construido. Los comentarios ayudan a los programadores actuales y futuros o a los usuarios de los mismos a comprender el programa. En Java, como en todos los lenguajes de programación, los comentarios no son sentencias de programación y son, por consiguiente, ignorados por el compilador.

En Java, los comentarios que constan de una única línea están precedidos por dos barras inclinadas (`//`), si se extienden sobre varias líneas están encerrados entre `/*` y `*/`.

Ejemplos de comentarios:

```
// Esto es un comentario relativo
/* Comentario escrito en
   2 líneas */
```

Existen también otra clase de comentarios, denominados comentarios de documentación, que pueden ser extraídos a archivos HTML utilizando javadoc. Es necesario introducirlos entre los símbolos /** . . . */.

***Palabras reservadas***

Las palabras reservadas o palabras clave (Keywords) son palabras que tienen un determinado significado para el compilador y no pueden ser utilizadas para otros fines. Por ejemplo, la palabra `while` significa que se habrá de evaluar la expresión que viene a continuación y, en función del valor de la misma, se ejecutarán o no se ejecutarán las sentencias siguientes. Otras palabras reservadas son `public`, `static` y `private`, que representan modificadores. Otro ejemplo es `class`, una palabra reservada muy utilizada, que significa que la palabra que viene a continuación es el nombre de la estructura clase.

***Sentencias***

Una sentencia representa una acción o una secuencia de acciones. Cada sentencia termina con un punto y coma (;). Algunos ejemplos de sentencias son:

```
z = 15;                        //esta sentencia asigna 15 a la
                               //variable z
z = z+100;                     //esta sentencia añade 100
                               //al valor de z
println("Bienvenido Sr.");     //sentencia de visualización
```

***Bloques***

Un *bloque* es una estructura que agrupa sentencias. Los bloques comienzan con una llave de apertura ( { ) y terminan con una llave se cierre ( } ). Un bloque puede estar dentro de otro bloque y se dice que el bloque interior está anidado dentro del exterior o que ambos bloques están *anidados:*

```
z = 15;
z = 2+100;
if (z > 225)
{
      z = 2-5;
}
```

***Clases***

La clase es la construcción fundamental de Java y constituye una plantilla o modelo para fabricar objetos. Un programa consta de una o más clases y cada una de ellas puede contener declaraciones de datos y métodos.

***Métodos***

Un *método* es una colección de sentencias que realizan una serie de operaciones determinadas. Por ejemplo:

```
System.out.println("Bienvenido a Java");
```

es un método que visualiza un mensaje en el monitor o consola.

***Método main ()***

Cada aplicación Java debe tener un método main declarado por el programador que define dónde comienza el flujo del programa. El método main tendrá siempre una sintaxis similar a ésta:

```
public static void main (Stringq[] args)
{
      //sentencias
}
```

## 1.2.Estructuras de Decisión

### 1.2.1. Sentencia if-else simple y compuesta

La sintaxis de la sentencia *if-else* es:

```
if ( condición )
  Bloque de código a ejecutar si la condición es cierta
else
  Bloque de código a ejecutar si la condición es falsa
```

Ejemplo:

```
// La respuesta es Aceptar o Cancelar
if (respuesta == Aceptar) {
  // código para realizar la acción Aceptar
  System.out.println( "Su petición está siendo atendida" );
}
else {
  // código para realizar la acción Cancelar
  System.out.println( "Cancelando acción" );
}
```

Utilización de else-if:

```
int valor;
char clasificacion;
if (valor > 90)
{
   clasificacion='A';
}else
  if (valor > 80)
  {
     clasificacion='B';
  }else
    if (valor > 70)
    {
       clasificacion='C';
    }else
     {
       clasificacion='F';
     }
```

### 1.2.2. Sentencia switch

Mediante la sentencia *switch* se puede seleccionar entre varias sentencias según el valor de cierta expresión.

La forma general de *switch* es la siguiente:

```
switch ( expresionMultivalor ) {
  case valor1 : conjuntoDeSentencias; break;
  case valor2 : conjuntoDeSentencias; break;
  case valor3: conjuntoDeSentencias; break;
  default: conjuntoDeSentencias; break;
}
```

Un ejemplo sencillo:

```
int meses;
switch ( meses ){
  case 1: System.out.println( "Enero" ); break;
  case 2: System.out.println( "Febrero" ); break;
  case 3: System.out.println( "Marzo" ); break;
  //Demas meses
  // . . .
  case 12: System.out.println( "Diciembre" ); break;
  default: System.out.println( "Mes no valido" ); break;
}
```

## 1.3.Estructuras de Repetición

### 1.3.1. Iteraciones con for, while, do – while

**El bucle while.-** es el bucle básico de iteración. Sirve para realizar una acción sucesivamente mientras se cumpla una determinada condición.

La forma general del bucle *while* es la siguiente:

```
while ( expresiónBooleana )
{
  sentencias;
}
```

Por ejemplo, multiplicar un número por 2 hasta que sea mayor que 100:

```
int i = 1;
while ( i <= 100 ) {
  i = i * 2;
}
```

***El bucle do-while.-*** es similar al bucle *while*, pero en el bucle *while* la expresión se evalúa al principio del bucle y en el bucle *do-while* la evaluación se realiza al final.

La forma general del bucle *do-while* es la siguiente:

```
do {
  sentencias;
} while ( expresiónBooleana );
```

La sentencia *do-while* es el constructor de bucles menos utilizado en la programación, pero tiene sus usos, cuando el bucle deba ser ejecutado por lo menos una vez.

Por ejemplo, cuando se lee información de un archivo, se sabe que siempre se debe leer por lo menos un carácter:

```
int c;
do {
  c = System.in.read( );
  // Sentencias para tratar el carácter c
} while ( c != -1 ); // No se puede leer más (Fin archivo)
```

***El bucle for.-*** Mediante la sentencia *for* se resume un bucle *do-while* con una iniciación previa. Es muy común que en los bucles *while* y *do-while* se inicien las variables de control de número de pasadas por el bucle, inmediatamente antes de comenzar los bucles. Por eso el bucle *for* está tan extendido.

Sintaxis:

```
for ( iniciación ; terminación ; incremento )
  sentencias;
```

Ejemplo:

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

Algunos (o todos) estos componentes pueden omitirse, pero los puntos y coma siempre deben aparecer (aunque sea sin nada entre sí).

Ejemplo común de uso:

```
// cad es una cadena (String)
for (int i = 0; i < cad.length() ; i++){
  // hacer algo con el elemento i-ésimo de cad
}
```

### 1.3.2. Sentencias break, continue, return

***Sentencia break***

La sentencia *break* provoca que el flujo de control salte a la sentencia inmediatamente posterior al bloque en curso. Ya se ha visto anteriormente la sentencia *break* dentro de la sentencia *switch*.

El uso de la sentencia break con sentencias etiquetadas es una alternativa al uso de la sentencia *goto*, que no es soportada por el lenguaje Java.

Se puede etiquetar una sentencia poniendo una identificador Java válido seguido por dos puntos antes de la sentencia:

```
nombreSentencia: sentenciaEtiquetada
```

La sentencia *break* se utiliza para salir de una sentencia etiquetada, llevando el flujo del programa al final de la sentencia de programa que indique:

```
break nombreSentencia2;
```

Un ejemplo de esto sería el programa:

```
void gotoBreak() {
    System.out.println("Ejemplo de break como 'goto' ");
a:  for( int i=1; i<10; i++ ){
      System.out.print(" i="+i);
      for( int j=1; j<10; j++ ){
        if ( j==5 )
          break a; //Sale de los dos bucles!!!
        System.out.print(" j="+j);
      }
      System.out.print("No llega aquí");
    }
}
```

Al interpretar *break a*, no solo se rompe la ejecución del bucle interior (el de *j*), sino que se salta al final del bucle *i*, obteniéndose:

i=1 j=1 j=2 j=3

***Sentencia continue***

Del mismo modo que en un bucle se puede desear romper la iteración, también se puede desear continuar con el bucle, pero dejando pasar una determinada iteración. Se puede usar la sentencia ***continue*** dentro de los bucles para saltar a otra sentencia, aunque no puede ser llamada fuera de un bucle.

Tras la invocación a una sentencia *continue* se transfiere el control a la condición de terminación del bucle, que vuelve a ser evaluada en ese momento, y el bucle continúa o no dependiendo del resultado de la evaluación. En los bucles *for* además en ese momento se ejecuta la cláusula de incremento (antes de la evaluación).

Por ejemplo el siguiente fragmento de código imprime los números del 0 al 9 no divisibles por 3:

```
for ( int i = 0 ; i < 10 ; i++ ) {
  if ( ( i % 3 ) == 0 )
    continue;
  System.out.print( " " + i );
}
```

Ejemplo utilizando etiquetas:

```
void gotoContinue( ) {
f: for ( int i=1; i<5; i++ ) {
     for ( int j=1; j<5; j++ ) {
       if ( j>i ) {
         System.out.println(" ");
         continue f;
       }
       System.out.print( " " + (i*j) );
     }
   }
}
```

En este código la sentencia *continue* termina el bucle de *j* y continua el flujo en la siguiente iteración de i. Ese método imprimiría:

1
2 4
3 6 9
4 8 12 16

***Sentencia return***

Se puede usar para salir del método en curso y retornar a la sentencia dentro de la cual se realizó la llamada. Para devolver un valor, simplemente se debe poner el valor (o una expresión que calcule el valor) a continuación de la palabra ***return.*** El valor devuelto por ***return*** debe coincidir con el tipo declarado como valor de retorno del método.

Cuando un método se declara como ***void*** se debe usar la forma de ***return*** sin indicarle ningún valor. Esto se hace para no ejecutar todo el código del programa:

```
int contador;
boolean condicion;
int devuelveContadorIncrementado(){
  return ++contador;
}

void metodoReturn(){
  //Sentencias
  if ( condicion == true )
    return;
  //Más sentencias a ejecutar si condición no vale true
}
```

### 1.3.3. Bloques try, catch, finally

Las excepciones son otra forma más avanzada de controlar el flujo de un programa. Con ellas se podrán realizar acciones especiales si se dan determinadas condiciones, justo en el momento en que esas condiciones se den.

Esas condiciones se conocen como *excepciones* que ayudan a la construcción de código robusto. Cuando ocurre un error en un programa, el código que encuentra el error lanza una excepción, que se puede capturar y recuperarse de ella. Java proporciona muchas excepciones predefinidas.

**Sintaxis:**

```
try {
    sentencias;
} catch( Exception ) {
    sentencias;
}
```

El manejo de excepciones se analizará con más detalle en la séptima unidad.

# 2. Funciones de entrada/salida estándar

## 2.1.Introducción

- En Java la E/S de información se realiza mediante *flujos;* es decir, secuencias de datos que provienen de una fuente.
- Estos flujos son objetos que actúan de intermediarios entre el programa y el origen o destino de la información, de forma que éste lee o escribe en el flujo y puede hacer abstracción sobre la naturaleza de la fuente.
- Como las clases relacionadas con flujos pertenecen a un paquete denominado java.io, los programas que utilizan flujos necesitan la inclusión en los mismos de la instrucción:

```
import java.io.*;
```

## 2.2.Salida de texto y variables por pantalla

***System.out***

La salida básica por pantalla se lleva a cabo mediante los métodos **print** y **println**, pertenecientes a la clase `PrintStream`. A estos métodos se les puede llamar mediante `System.out`, que hace referencia a la salida estándar. El mecanismo básico para salida con formato utiliza el tipo `String`. En la salida, el signo + combina dos valores de tipo `String` y si el segundo argumento no es un valor de tipo `String`, pero es un tipo primitivo, se crea un valor temporal para él de tipo `String`.

Estas conversiones al tipo `String` se pueden definir también para objetos.

Sintaxis:

- *`print(a):`* Imprime *a* en la salida, donde *a* puede ser cualquier tipo básico Java ya que Java hace su conversión automática a cadena.
- *`println(a):`* Es idéntico a *`print(a)`* salvo que con *`println()`* se imprime un salto de línea al final de la impresión de *a*.

## 2.3.Lectura desde teclado

***System.in***

Este objeto implementa la entrada estándar (normalmente el teclado). Los métodos que nos proporciona para controlar la entrada son:

- *read():* Devuelve el carácter que se ha introducido por el teclado leyéndolo del buffer de entrada y lo elimina del buffer para que en la siguiente lectura sea leído el siguiente carácter. Si no se ha introducido ningún carácter por el teclado devuelve el valor -1.
- *skip(n):* Ignora los *n* caracteres siguientes de la entrada.

*Ejemplo:*

```
/*
Ejemplo 1 sobre operaciones básicas de entrada y salida.
Ejecute el siguiente programa y analice los resultados
*/
import java.io.*;
public class EntradaSalida1
{
	public static void main (String[] args)
	{
		int i;
		char c;
		try{
			System.out.println ("Escriba un número natural con "
			+"un único dígito y pulse RETURN");

			i=System.in.read();
			System.in.skip(2) ; //Evitamos los caracteres \r\n
			//System.in.skip(System.in.available());

			System.out.println(i);
			System.out.println ("Escriba un número natural con "
			+"un único dígito y pulse RETURN");
			c=(char)System.in.read();
			System.out.println(c);

		}catch ( IOException e){
			System.out.println ("Error");
		}
	}
}
```

**StringBuffer**

Podemos utilizar un StringBuffer para almacenar una cadena de caracteres, ejemplo:

```
import java.io.*;
public class EntradaSalida2
{
	public static void main(String[] args)
	{
		String cadena;
		System.out.print("Escribe un texto: ");
		// Almacenamos el texto en buffer
		StringBuffer buffer = new StringBuffer();
		char c;
		try {
			//Mientras no presione enter
			while ( (c = (char) System.in.read() )!='\n' )
				buffer.append(c);
		} catch(IOException ex){ }
		//Quitamos los espacios antes y despues de...
		//cadena = buffer.toString().trim();
		cadena=buffer.toString();
		System.out.println("la cadena es: " + cadena);
	}
}
```

**readLine()**

No obstante, la entrada básica en Java suele realizarse mediante el método `readLine()` de la clase `BufferedReader`, que lee una secuencia de caracteres de un flujo de entrada y devuelve una cadena.

Para efectuar este tipo de entrada debe construirse un objeto de la clase `BufferedReader` sobre otro de la clase `InputStreamReader` asociado a `System.in`, que se encarga de convertir en caracteres los bytes leídos desde teclado.

Según este planteamiento, la lectura de valores numéricos con llevaría dos fases:

1. *lectura de una cadena*
2. *conversión de la cadena en número.*
   a. ***Valores tipo entero (`int` o `long`)***
      *Métodos `Integer.parseInt` e `Integer.parseLong`*

   b. ***Valores de tipo real (`Float` o `Double`)***
      *Métodos `floatValue` o `doubleValue`*

Los tipos de datos `Integer`, `Long`, `Float` y `Doble` son clases pertenecientes al paquete `java.lang` que representan como objetos los tipos simples de su mismo nombre.

Ejemplo:

```
import java.io.* ;
public class EntradaSalida3
{
      public static void main (String[] args)
      {
            InputStreamReader isr = new InputStreamReader (System.in);
            BufferedReader br = new BufferedReader(isr);
            String cadena;
            try{
                  System.out .println ("Escribe tu nombre y presiona
                                        ENTER");
                  cadena=br.readLine();
                  System.out.println("Hola " +cadena+", escribe un
                                    número entero y presiona ENTER") ;
                  int entero=Integer.parseInt(br.readLine());
                  System.out.println("El número introducido fue el"
                                        +entero);
                  System.out.println(cadena+", introduce ahora un real
                                        y presiona ENTER") ;
                  System.out.println("(utilice el punto como separador
                                        ej 3.45)");
                  cadena=br.readLine();
                  Double d=new Double (cadena);
                  double real=d.doubleValue();
                  System.out.println("El real es "+real) ;
            }catch (IOException e){
                  System.out .println ("Error");
            }
      }
}
```

**Scanner()**

Otro recurso para leer información del teclado es la utilización de Scanner.

```
import java.io.*;
import java.util.Scanner;
public class EntradaSalida4
{
	public static void main(String [] arg)
	{
		Scanner s = new Scanner(System.in);
		System.out.println("ingresa un texto ");
		String cadena = s.next();
		System.out.println("el texto es "+ cadena);
		System.out.println("ingresa un numero real");
		float valor = s.nextFloat();
		System.out.println("el numero es "+ valor);
		//Otra forma de utilizar Scanner
		System.out.println();
		System.out.println("Otra forma de utilizar Scanner");
		System.out.println("ingresa una secuencia de 5 números
				separadas por ,");
		String texto  = s.next();
		//s.close();
		//String texto = "1, 2, 3, 4, 5, 6";
		s = new Scanner(texto).useDelimiter("\\s*,\\s*");
		System.out.println(s.nextInt());
		System.out.println(s.nextInt());
		System.out.println(s.nextInt());
		System.out.println(s.nextInt());
		s.close();
	}
}
```