DESARROLLO DE APLICACIONES WEB

Anexo II - Unidad 5

P r o g r a m a c i ó n M o d u l a r

1r DAW

IES La Mola de Novelda

Departament d'informàtica

Professor: Joan Carbonell Picó

Índice

Anexo II Unidad 5: Programación Modular

1.- INTRODUCCIÓN

La **programación estructurada** es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de computadora recurriendo únicamente a subrutinas y tres estructuras básicas: **secuencia**, **selección** (if y switch) e **iteración** (bucles for y while).

En cambio, la **programación modular** es un paradigma de programación que consiste en **dividir un programa en módulos o subprogramas** con el fin de hacerlo más legible y manejable. Se presenta históricamente como una **evolución de la programación estructurada** para solucionar problemas de programación más grandes y complejos del que ésta puede resolver.

Esta es una metodología bastante popular y útil en la hora de diseñar algoritmos complejos puesto que consiste en ir obteniendo subproblemas lo más simples posibles puesto que éstos son más fáciles de resolver. Esta técnica se denomina **divide y vencerás** o **diseño descendente** o **análisis descendente** (***top-down***). En pocas palabras, éste se basa en la estrategia, aplicable en cualquier campo y no solo en la programación, de dividir problemas complejos en un conjunto de problemas más simples y fáciles de atacar y entender.

La cuestión es considerar un problema complejo como un conjunto de problemas simples que serán más fáciles de resolver. Por lo tanto, la resolución de estos problemas más simples nos ayuda a resolver tareas más complejas. Además, conforme aumenta el tamaño de un programa (líneas de código fuente) se hace más complicado hacer un mantenimiento del programa y muchas veces tendremos código redundante.

Por lo tanto, el primer paso que hay que hacer para resolver cualquier problema será intentar encontrar su descomposición en problemas más pequeños.

2.- VENTAJAS DE LA PROGRAMACIÓN MODULAR

- Facilita la comprensión del problema y su resolución escalonada.
- Aumenta la claridad y legibilidad de los programas.
- Permite que varios programadores trabajan en el mismo problema a la vez, puesto que cada uno puede trabajar en uno o varios módulos de manera bastante independiente.
- Reduce el tiempo de desarrollo, reutilizando módulos previamente desarrollados.

- Mejora la fiabilidad de los programas, porque es más sencillo diseñar y depurar módulos pequeños que programas enormes.
- Facilita el mantenimiento de los programas.
- Se consigue la reutilización de código. En lugar de escribir el mismo código repetido cuando se necesite, se hace una llamada al método que lo realiza.

3.- FUNCIONES

Una **función** es cada uno de los subproblemas en que se ha dividido el problema principal. Por lo tanto, por cada subproblema a resolver, dentro de vuestro código fuente se tendrá que definir una función diferente. En el lenguaje Java, estos conjuntos de instrucciones se los denomina **métodos**, en lugar de funciones, pero a efectos prácticos, los podéis considerar como lo mismo. Los identificadores de los métodos siguen las mismas convenciones de código que las variables (*lowerCamelCase*).

Cuando se habla de **método principal**, se trata de un conjunto de instrucciones que, etiquetadas bajo un identificador llamado *main*, resuelven el problema general (o sea, todo el programa). Dado que hasta el momento no se había aplicado diseño descendente, no había subproblemas, y por lo tanto en vuestro código fuente solo se había definido este único método. No hacía falta otro.

Cuando se habla de la **invocación de un método** sobre valores de ciertos tipos de datos complejos, como las cadenas de texto (*String*), se trata de ejecutar un conjunto de instrucciones con un objetivo común: transformar la cadena de texto u obtener datos contenidos

La **declaración básica de un método** se hace usando la sintaxis que se muestra a continuación. Cómo podéis ver, su formato es muy parecido a cómo se declara el método principal (pero no exactamente igual, alerta!):

```
public static void nombreMétodo() {
        //Aquí dentro irán sus instrucciones
        //...
}
```

Esta declaración se puede llevar a cabo en cualquier lugar del fichero de código fuente, siempre que sea entre las claves que identifican el inicio y fin de fichero (*public class NombreClasse { … }*) y fuera del bloque de instrucciones método principal, o cualquier otro método. Normalmente, se suele hacer inmediatamente a continuación del método principal

Ejemplo: Problema que lee una lista, la ordena y la visualiza por pantalla

```
public class OrdenarDescendente {
	public static void main(String[] args) {
		//Instrucciones del método principal (problema general)
		//...
	}
	//Método que resuelve el subproblema de leer la lista.
	public void leerLista() {
		//Instrucciones del método
		//...
	}
	// Método que resuelve el subproblema de ordenar la lista.
	public void ordenarLista() {
		//Instrucciones del método
		//...
	}
	//Método que resuelve el subproblema de mostrar la lista por pantalla.
	public void mostrarLista() {
		//Instrucciones del método
		//...
	}
}
```

Por las características del lenguaje Java, hace falta que el método principal tenga un formato muy concreto. En caso contrario, habrá un error de compilación en futuros pasos del proceso. Todo el código que iría normalmente dentro del bloque de instrucciones del método principal se ubica en un nuevo método auxiliar, y dentro del método principal simplemente se invoca este nuevo método. De hecho, no es imprescindible que conozcáis los detalles de los motivos por los cuales es necesario hacer este cambio. Simplemente podéis usar el código siguiente de ejemplo como plantilla para generar vuestros programas, teniendo en cuenta que todo el código que pondríais normalmente al método principal, ahora irá al método inicio.

```
public class OrdenarDescendente {
	public static void main (String[] args) {
		//Aquí es necesario usar el nombre de la clase que estáis creando.
		OrdenarDescendente programa = new OrdenarDescendente();
		programa.inicio();
	}
	public void inicio() {
		//Instrucciones del método principal (problema general)
		//...
	}
	//Resto de métodos
	//...
}
```

4.- EJERCICIOS

- Programa que solicita un número positivo diferente a cero por teclado y nos dice si es par o impar.
- Programa que solicita un número positivo diferente a cero y nos dice si es primero o no

5.- INSTRUCCIÓN RETURN

Tienes que tener en cuenta dos cosas importantes con la sentencia ***return***:

- Cualquier instrucción que se encuentre después de la ejecución de *return* NO será ejecutada. Es común encontrar funciones con múltiples sentencias *return* en el interior de los condicionales, pero una vez que el código ejecuta una sentencia *return* todo el código que venga después no se ejecutará.
- El tipo del valor que se devuelve en una función tiene que coincidir con el del tipo declarado en la función, es decir si se declara *int*, el valor devuelto tiene que ser un número entero.
- En el caso de los procedimientos (void) podemos usar la sentencia *return* pero sin ningún tipo de valor, solo la usaríamos como una manera de acabar la ejecución del procedimiento.

6.- DOCUMENTACIÓN

Cómo dijimos en una unidad anterior, si los comentarios de los programas en general son importantes, tendremos que prestar mucha atención a los comentarios de las funciones puesto que con toda probabilidad la utilizarán otros programadores o programadoras que querrán saber exactamente que hace ese método.

Las etiquetas utilizadas en la documentación de los métodos son:

- ***@param*** seguida de un nombre de parámetro indica qué parámetros espera como entrada al método. Si un método acepta varios parámetros, cada uno de ellos se especificarán con diferentes etiquetas ***@param*** a los comentarios.
- ***@return*** especificamos qué devuelve exactamente el método. A pesar de que el tipo de dato que se devuelve ya se especifica a la cabecera del método, mediante esta etiqueta podemos explicarlo con más detalles.

7.- CREACIÓN DE BIBLIOTECAS DE RUTINAS MEDIANTE PAQUETES.

Si cualquiera de las funciones que hemos implementado anteriormente queremos utilizarlas en otro programa, el que haríamos es copiar y pegar. Esta solución no es nada práctica ni elegante.

Una forma de poder utilizar funciones o métodos que ya hemos creado anteriormente es con la agrupación de **paquetes** (*package*). Las funciones de un determinado tipo (por ejemplo funciones matemáticas) se pueden agrupar para crear un paquete que después se podrá importar desde el programa que necesita esas funciones.

Para declarar un paquete en Java se hace uso de la palabra reservada "***package***" seguido de la "**ruta**" del paquete, como se muestra a continuación:

package ruta.del.paquete;

Cosas a tener en cuenta en la declaración de un paquete:

- Se hace al principio del archivo Java. Primero declaramos el paquete y después pondremos los ***imports*** y las clases, interfaces, métodos, etc.
- Cada punto (.) en la ruta del paquete es una nueva carpeta. Por ejemplo:

  package otro_paquete.mi_paquete equivale a la ruta **otro_paquete/mi_paquete**.
- En general, los nombres de los paquetes en Java se declaran en minúscula y si es necesario las palabras se separan por medio del subrayado (_) o guión bajo.
- Si no declaramos ningún paquete para la clase que estamos haciendo (al principio lo hacemos así) esta quedaría en un paquete por defecto (*default package*). En él estarán todas las clases declaradas que no tengan un paquete declarado. A pesar de que no genera ningún error de compilación, siempre será recomendable declarar un paquete a cada componente de nuestro programa en Java. De esa manera podemos dar diferentes niveles de seguridad a los componentes que hemos creado y lo pueden mantener organizado

Cómo hemos dicho anteriormente, cada paquete corresponde a un directorio. Por lo tanto, si hay un paquete de nombre **matemáticas** también tendrá que haber una carpeta con el nombre **matemáticas** en la misma ubicación del programa que importa ese paquete (normalmente el programa principal).

Las funciones se pueden agrupar dentro de un paquete de dos maneras diferentes:

1. Pueden haber subcarpetas dentro de un paquete. Por ejemplo, imaginamos que queremos dividir las funciones matemáticas en:

   Funciones relativas al cálculo de áreas y volúmenes de figuras geométricas

   Funciones relacionadas con cálculos estadísticos

   En estos casos, crearíamos dentro de la carpeta **matemáticas** las carpetas **geometría** y **estadística**. Estas subcarpetas se llamarían **matemáticas.geometría** y **matemáticas.estadística**.

2. Otra manera de agrupar las funciones dentro de un mismo paquete consiste en crear diferentes archivos dentro del mismo directorio. En este caso, podríamos crear los archivos **Geometría.java** y **Estadística.java**.

7.1.-EJERCICIO

Tenéis que crear un paquete de nombre **matemátiques** que contenga dos clases: **Varies** (funciones matemáticas de propósito general) y **Geometria**. Por lo tanto en el disco duro, tendremos que tener una carpeta de nombre **matemátiques** que contendrá los archivos **Varies.java** y **Geometria.java**. Tendréis que implementar el contenido de estos archivos a partir de sus firmas:

- Descárgate el archivo **Archivo.zip**.
- Implementa los métodos de las dos clases.
- Observa que en los dos archivos se especifica que las clases declaradas pertenecen al paquete **matematiques** mediante la línea **package matematiques**.
- Ahora probaremos las funciones desde un programa externo. Descárgate el programa **ProvaFuncions.java**. Este estará fuera de la carpeta matemáticas, justamente en un nivel superior en la estructura de directorios.
- Fíjate que las líneas **import matematiques.Varies;** e **import matematiques.Geometría;** cargan las clases contenidas en esos paquetes. Se recomienda no utilizar la forma **import matematiques.*;** puesto que está desaconsejado en el uso estándar de codificación en Java de Google.
- Si compilamos desde el directorio donde está **ProvaFuncions.java** veremos que también se

generan los **byte-code** de las dos clases importadas del paquete **matematiques**.

8.- ÁMBITO DE LAS VARIABLES

El ámbito de las variables es el espacio donde existe esa variable, es decir, el lugar donde esa variable es válida. Las variables utilizadas como parámetros (por valor) o las variables que se definen dentro de la función o método son locales, es decir, su ámbito es la función y fuera de ella esas variables ya no existen.

9.- PASO DE PARÁMETROS POR VALOR Y POR REFERENCIA.

En Java, como el mayoría de lenguajes de programación existen dos formas de pasar parámetros a una función o método:

- Por **valor**: Cuando se pasa un parámetro por valor, en realidad se pasa una copia de la variable, sólo importa el valor. Cualquier modificación que se le haga a la variable que se le pasa como parámetro dentro de la función no tendrá ningún efecto fuera de la misma. Fíjate en el siguiente ejemplo y comprueba que a pesar de que la variable que se le pasa como parámetro se modifica dentro de la función, este cambio no tiene ningún efecto en el programa principal:

```
public class PerValor {

    public static void main(String[] args){

        int n = 10;
        System.out.println(n);
        calcula(n);
        System.out.println(n);
    }

    public static void calcula(int x){

        x = x + 24;
        System.out.println(x);
    }
}
```

- Por **referencia**: Cuando se pasa un parámetro por referencia, al contrario que por valor, si se modifica su valor dentro de la función.

  En la mayoría de los lenguajes de programación es el programador o programadora quien decide cuando un parámetro se pasa por valor y cuando se pasa por referencia. En Java no podemos elegir. Todos los parámetros que son del tipo *int*, *double*, *float*, *char* o *String* se pasan siempre por valor, mientras que los *arrays* se pasan siempre por referencia. Por lo

tanto, si pasamos un *array* como parámetro, cualquier cambio que hagamos a la *array* dentro de la función, permanecerá una vez finalice la ejecución de la función. Atentos con el paso de parámetros con los *arrays*. Fíjate en el siguiente ejemplo:

```
public class PerReferenciaArrays {

    public static void main(String[] args){

        int n[]  = {8,33,200,150,11};
        int m[] = new int[5];

        muestraArray(n);
        incrementaArray(n);
        muestraArray(n);
    }

    public static void muestraArray(int x[]){
        for(int i = 0;i < x.length;i++){
            System.out.printl(x[i] + " ");
        }
        System.out.println();
    }

    public static void incrementa(int x[]){
        for(int i = 0;i < x.length;i++){
            x[i]++;
        }
    }
}
```

10.- SOBRECARGA DE FUNCIONES

Java permite tener dos o más funciones con el mismo nombre dentro del mismo programa. Esto es conoce como **sobrecarga de funciones**. La manera de poder distinguir una de las otras es mediante su listado de parámetros, que tendrán que ser diferentes en cantidad o en tipo. Si vamos a la API de Java, veremos muchas funciones o métodos sobrecargados

10.1.- EJERCICIO

Diseñar un programa que contenga una función que suma dos valores enteros, otra que sume dos valores decimales, otra que sume dos caracteres (consiste en concatenarlos), otra que sume dos cadenas de texto (consiste en concatenarlas) y otra que haga una suma ponderada entre dos valores enteros. Todas ellas podrían estar en un paquete llamado **matematiques.numeros** (las sumas de números) y en el paquete **matematiques.text** (las sumas con caracteres y cadenas de texto).

11.- RECURSIVIDAD

Un función puede ser invocada desde cualquier lugar, desde el programa principal, desde otra función e incluso desde dentro de su cuerpo de instrucciones. En este últimocaso, cuando una función se invoca a sí misma, decimos que es una función recursiva. El esquema general es:

```
static int funciónRecursiva() {
   ...
   funciónRecursiva(); //llamada recursiva
   ...
}
```

Podemos ver que dentro de la función recursiva se invoca a la misma función recursiva, la cual volverá a invocar a la misma función, y así sucesivamente. Esta situación nos lleva a un bucle infinito de llamafas a la función. Para evitarlo, tendremos que habilitar un mecanismo que pare, en un momento dado, las series de llamadas recursivas. Un sentencia condicional ***if*** impedirá que la llamada recursiva sea infinita. La condición que para la llamada recursiva a la función se llama **caso base**. Este es el esquema general (el que devuelva un *int* es un ejemplo, puede devolver cualquier tipo):

```
int funciónRecursiva(datos) {
   int resultado;
   if (caso base) {
      resultado = valorBase;
   } else {
      resultado = funciónRecursiva(nuevos datos); //llamada recursiva
       . . .
   }
return (resultado);
}
```

Cuando caso base sea falso, se volverá a llamar a ella misma. El parámetro nuevos datos tiene que ser más pequeño que datos. De esa manera se garantiza que en algún momento se llega al caso base.

11.1.- EJERCICIOS

1. Crea una función recursiva para calcular la potencia de un número elevado a otro (x^n).
2. Crea una función recursiva para calcular el factorial de un número.

12.- ACTIVIDADES

Ejercicios 1-14

Crea una biblioteca de funciones matemáticas que contenga las siguientes funciones. Recuerda que puedes usar unas dentro de otras si es necesario.

1. **esCapicua**: Devuelve verdadero si el número que se pasa como parámetro es capicúa y falso en caso contrario.
2. **esPrimo**: Devuelve verdadero si el número que se pasa como parámetro es primo y falso en caso contrario.
3. **siguientePrimo**: Devuelve el menor primo que es mayor al número que se pasa como parámetro.
4. **potencia**: Dada una base y un exponente devuelve la potencia.
5. **digitos**: Cuenta el número de dígitos de un número entero.
6. **voltea**: Le da la vuelta a un número.
7. **digitoN**: Devuelve el dígito que está en la posición n de un número entero. Se empieza contando por el 0 y de izquierda a derecha.
8. **posicionDeDigito**: Da la posición de la primera ocurrencia de un dígito dentro de un número entero. Si no se encuentra, devuelve -1.
9. **quitaPorDetras**: Le quita a un número n dígitos por detrás (por la derecha).
10. **quitaPorDelante**: Le quita a un número n dígitos por delante (por la izquierda).
11. **pegaPorDetras**: Añade un dígito a un número por detrás.
12. **pegaPorDelante**: Añade un dígito a un número por delante.
13. **trozoDeNumero**: Toma como parámetros las posiciones inicial y final dentro de un número y devuelve el trozo correspondiente.
14. **juntaNumeros**: Pega dos números para formar uno.

Ejercicio 15

Muestra los números primos que hay entre 1 y 1000.

Ejercicio 16

Muestra los números capicúa que hay entre 1 y 99999.

Ejercicio 17

Escribe un programa que pase de binario a decimal.

Ejercicio 18

Escribe un programa que pase de decimal a binario.

Ejercicio 19

Une y amplía los dos programas anteriores de tal forma que se permita convertir un número entre cualquiera de las siguientes bases: decimal, binario, hexadecimal y octal.

Ejercicios 20-28

Crea una biblioteca de funciones para arrays (de una dimensión) de números enteros que contenga las siguientes funciones:

1. **generaArrayInt**: Genera un array de tamaño n con números aleatorios cuyo intervalo (mínimo y máximo) se indica como parámetro.
2. **minimoArrayInt**: Devuelve el mínimo del array que se pasa como parámetro.
3. **maximoArrayInt**: Devuelve el máximo del array que se pasa como parámetro.
4. **mediaArrayInt**: Devuelve la media del array que se pasa como parámetro.
5. **estaEnArrayInt**: Dice si un número está o no dentro de un array.
6. **posicionEnArray**: Busca un número en un array y devuelve la posición (el índice) en la que se encuentra.
7. **volteaArrayInt**: Le da la vuelta a un array.
8. **rotaDerechaArrayInt**: Rota n posiciones a la derecha los números de un array.
9. **rotaIzquierdaArrayInt**: Rota n posiciones a la izquierda los números de un array.

Ejercicio 29-34

Crea una biblioteca de funciones para arrays bidimensionales (de dos dimensiones) de números enteros que contenga las siguientes funciones:

1. **generaArrayBiInt**: Genera un array de tamaño n x m con números aleatorios cuyo intervalo (mínimo y máximo) se indica como parámetro.
2. **filaDeArrayBiInt**: Devuelve la fila i-ésima del array que se pasa como parámetro.
3. **columnaDeArrayBiInt**: Devuelve la columna j-ésima del array que se pasa como parámetro.
4. **coordenadasEnArrayBiInt**: Devuelve la fila y la columna (en un array con dos elementos) de la primera ocurrencia de un número dentro de un array bidimensional. Si el número no se encuentra en el array, la función devuelve el array {-1, -1}.
5. **esPuntoDeSilla**: Dice si un número es o no punto de silla, es decir, mínimo en su fila y máximo en su columna.
6. **diagonal**: Devuelve un array que contiene una de las diagonales del array bidimensional que se pasa como parámetro. Se pasan como parámetros fila, columna y dirección. La fila y la columna determinan el número que marcará las dos posibles diagonales dentro del array. La dirección es una cadena de caracteres que puede ser “nose” o “neso”. La cadena “nose” indica que se elige la diagonal que va del noroeste hacia el sureste, mientras que la cadena “neso” indica que se elige la diagonal que va del noreste hacia el suroeste.