MÉTODOS, PACKAGES Y EXPRESIONES REGULARES

Métodos, Packages y Expresiones regulares

Índice:

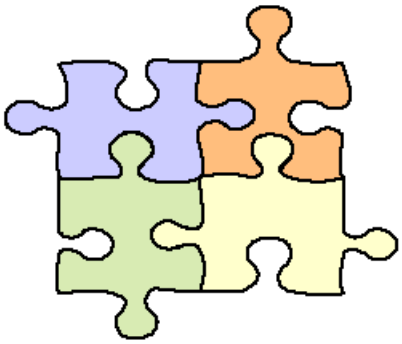
Métodos

Objetivos

La mejor forma de crear y mantener un programa grande, es construirlo a partir de piezas más pequeñas o módulos. Cada uno de los cuales es más manejable que el programa en su totalidad.

Los métodos (subprogramas) son utilizados principalmente con dos objetivos:

1. Para evitar la repetición de código en un programa al poder ejecutarlo desde varios puntos de un programa con sólo invocarlo.
2. un método puede ser llamado con diferentes argumentos, comportándose de la misma forma pero con valores y/o objetos distintos.



Definición.

El concepto de método consiste en encapsular un conjunto de instrucciones dentro de una declaración específica (llamada generalmente SUBPROGRAMA), permitiendo la descomposición funcional y la diferenciación de tareas.

Su utilidad principal es:

- Agrupar código que forma una entidad propia o una idea concreta.
- Agrupar código que se necesitará varias veces en un programa, con al misión de no repetir código.
- Se definen dándoles un nombre que agrupa una parte de código.
- Se les puede pasar valores para que los operen o procesen de alguna forma.
- Pueden devolver un resultado para ser usado en algún otro sitio.

La declaración de un método está formada por una **cabecera** y un **cuerpo**.

Codificación en Java.

La codificación de un método consiste en una **cabecera** para su identificación y de un **cuerpo** que contiene las sentencias que éste ejecuta.

La **cabecera** se compone de un **nombre** (identificador del método), el **tipo del resultado**(tipos primitivos o clases) y una **lista de parámetros, que puede contener cero o más variables**.

```
[Modif_de_método] Tipo_devuelto Nombre_de_método (lista_de_parámetr
      Conjunto de instrucciones
}
```

- La **lista de parámetros** consiste en cero o más parámetros formales (variables u objetos), cada uno de ellos con un tipo.
- En el caso de que el método tenga más de un parámetro, estos deben ir separados con una coma.

Por ejemplo:

```
public static void miMetodo(int v1, int v2, float v3, String v4, Cl

}
```

> Nótese que aunque existan varios parámetros pertenecientes al mismo tipo o clase, no pueden declararse abreviadamente, como ocurre con las variables locales y los atributos, indicando el tipo y a continuación la lista de parámetros separados por comas. Así, es ilegal la siguiente declaración del método anterior:

```
public static void miMetodo(int v1, v2, float v3, String v4, ClaseOl
```

> La declaración de un parámetro puede ir antecedida, como ocurre con las variables locales, por la palabra reservada final. En ese caso, el valor de dicho parámetro no podrá ser modificado en el cuerpo del método.

Atención

Un método es:

- Un bloque de código que tiene un nombre,
- recibe unos parámetros (opcionalmente), esos parámetros en la llamada se llaman argumentos.

- contiene sentencias o instrucciones para realizar algo (opcionalmente) y
- devuelve un valor de algún Tipo conocido (opcionalmente).

La sintaxis global es:

```
Tipo_Valor_devuelto  nombre_método ( lista_argumentos ) {
    bloque_de_codigo;
}
```

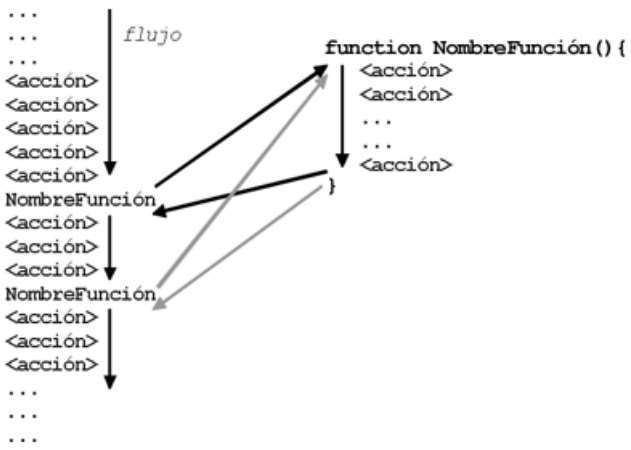
Llamada a un método

Los métodos pueden ser invocados o llamados desde cualquier método de la clase, incluido él mismo.

Además, cuando se invoca, se puede pasar un valor en cada parámetro, a través de una variable,un valor constante o un objeto.

En Java, la acción de pasar valores a parámetros de **tipo primitivo** (int, double, boolean, char..) se denomina **paso de parámetros por valor.** En éste caso, los argumentos que se pasan, no pueden ser modificados por la función, aunque se modificaran, al finalizar el método seguirían teniendo los mismos valores que en la llamada.
En el caso de que el parámetro sea **un Objeto**, lo que se está haciendo es un **paso de parámetros por referencia**, y en este caso, los argumentos si pueden ser modificados por el método.

El flujo de ejecución realiza un salto hasta el método y vuelve a la siguiente instrucción cuando el método haya terminado.



Los métodos se invocan con su nombre pasando la lista de argumentos entre paréntesis (si los hay).

Por ejemplo:

```
int x;
x = sumaEnteros(2,3);
```

Si el método no recibe ningún argumento, los paréntesis en la llamada son obligatorios. Por ejemplo para llamar a la función haceAlgo, simplemente se pondría:

```
haceAlgo();
```

ATENCIÓN: que como la función tampoco devuelve ningún valor no se asigna a ninguna variable. **(No hay nada que asignar).**

Ejemplo 01

Crearemos un método que realice la suma de dos enteros, y nos devuelva el resultado:

- cabecera del método : public static int sumar(int a,int b)

  tipo del resultado : int
  nombre del método : sumar
  lista de parámetros : int a, int b

- cuerpo del método {
- Instrucción : se retorna la suma de a y b return (a+b);
- fin del bloque }

Quedaría como sigue:

```
public static int suma(int a,int b){
       return a + b;
}
```

Y el programa completo:

```
import java.util.Scanner;

public class MetodoSumar {
    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        int x, y, s;

        System.out.println("Dame el primer valor :");
        x = teclado.nextInt();
        System.out.println("Dame el segundo valor :");
        y = teclado.nextInt();

        s = suma(x,y);

        System.out.println("Resultado de la suma = "+s);
    }
    public static int suma(int a,int b){
        return a + b;
    }
}
```

Dame el primer valor :

5
Dame el segundo valor :
-8
Resultado de la suma = -3

Process finished with exit code 0

Ejemplo 02

Programa que indica si un número es par o impar, utilizando un método al que se le pasa **como argumento un número entero** y devuelve **true** si es PAR y **false** si es IMPAR.

```
import java.util.Scanner;

public class MetodoParImpar {
    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        int x;
        boolean esPar;

        System.out.print("Dame el número :");
        x = teclado.nextInt();
        esPar = par(x);
        if (esPar)
            System.out.println("El número "+x+" es PAR");
        else
            System.out.println("El número "+x+" es IMPAR");
    }

    public static boolean par(int num){
        boolean resp = false;
        if (num%2==0){
            resp = true;
        }
        return resp;
    }
}
```

Dame el número :7
El número 7 es IMPAR

Process finished with exit code 0

Evitar escribir métodos demasiado largas.

Según estudios psicológicos, la mente humana no es capaz de procesar más de 6 ó 7 detalles diferentes a la vez. Las funciones demasiado largas suelen contener un número de detalles superior a este límite. Ello dificulta su legibilidad y comprensión y por tanto, su mantenimiento.

En general, en funciones demasiado largas hay trozos claramente diferenciados de código, débilmente acoplados. Cada uno de estos bloques suele realizar una tarea distinta. Por ejemplo, es habitual en muchas funciones que al principio se preparen los datos para realizar un cálculo, o se inicialice alguna estructura. A continuación se realizan una serie de cálculos y por último se presenten por la salida los resultados.

En ese caso, por ejemplo, se recomienda dividir la función en tres subfunciones inicializa(..), calcular(..) e imprimir(..). No importa que las expectativas iniciales de reutilización de estas funciones sean prácticamente nulas. En general, si dos trozos de código pueden aparecer juntos en una sola función o separados en dos subfunciones, la opción recomendada siempre es separarlos, salvo justificación irrefutable en contra.

Según Linus Torvalds, creador del núcleo de Linux, "las funciones o métodos deberían ser cortas y dulces, y servir para un único propósito. Deberían ocupar una o dos pantallas estándares de texto, es decir, una pantalla de 24 líneas por 80 caracteres de ancho.".

Evitar copiar y pegar trozos cuasi idénticos de código a lo largo de una aplicación software.

Si un trozo de código, por ejemplo, una serie de sentencias destinadas a imprimir una matriz por pantalla, se copia y pega cada vez que se necesite dicha funcionalidad, el resultado será una aplicación con bloques idénticos de código dispersos a través de múltiples lugares de una aplicación software.

El principal problema viene cuando hay que modificar este bloque de código. Por ejemplo, si simplemente queremos añadir un recuadro a la matriz, deberemos modificar de la misma forma múltiples puntos de nuestra aplicación, lo que es propenso a errores y puede provocar problemas de consistencia.

En lugar de copiar y pegar trozos de código a lo largo de una aplicación software, la práctica recomendada es encapsular dicho trozo de código en una función e invocarla cuando se necesite.

Los parámetros

Los parámetros de un método pueden ser de dos tipos:

- **variables de tipo simple de datos**: En este caso, el paso de parámetros se realiza siempre por valor. Es decir, el valor del argumento de llamada no puede ser modificado en el cuerpo del método (**El método trabaja con una copia del valor utilizado en la llamada).**
- **variables de tipo objeto (referencias)**: En este caso, lo que realmente se pasa al método es una referencia al objeto y, por lo tanto, el valor del argumento de llamada sí que puede ser modificado dentro del método (**El método trabaja directamente con el objeto utilizado en la llamada**), a no ser que se anteponga la palabra reservada final.

Tipo del parámetro	Método de pase de parámetro
Tipo simple de datos (ejemplo: int, char, boolean, double, etc.)	POR VALOR
Tipo referencial (Objetos de una determinada clase, vectores y Strings)	POR REFERENCIA

IMPORTANTE: Todas las variables declaradas en las definiciones de los métodos **son variables locales**; solo se conocen en el método en el que se definen.

¿ Qué es el paso por valor de una variable ?

El paso por valor de una variable se realiza cuando al llamar a una función con parámetros de entrada se crea una copia de esta variable y por lo tanto, la variable original no se modifica, solo se modifica la copia de la variable, vamos a ver un ejemplo de código fuente para entender mejor el concepto:

```
public static void main(String[] args) {
    //Creamos una variable de tipo entero y le damos un valor
    int miNumero = 5;
    //LLamamos a una función que modifica el valor de la variable
    modificarNumero(miNumero);
    //Imprimimos el valor de la variable
    System.out.println(miNumero);
}

private static void modificarNumero(int miNumero) {
    //Modificamos el valor de la variable a 7
    miNumero = 7;
}
```

¿ Cuál será el resultado? ¿5 o 7?

Veamos la salida de nuestro programa
5

¿ Por qué el resultado es «5» si en la función modificamos el valor a «7» ?

> La respuesta es porque se trata de un paso de variable por valor, es decir, cuando llamamos a la función no estamos pasando la variable "miNumero", sino una copia de la variable "miNumero", por eso, al salir de la función el valor de nuestra variable no se modificado.

Modificamos el método para que visualice el contenido de la variable local

```
private static void modificarNumero(int miNumero) {
    //Modificamos el valor de la variable a 7
    miNumero = 7;
    System.out.println(miNumero);
}
```

La salida de nuestro programa como es de esperar es:

7
5

IMPORTANTE: El paso por valor de variables en Java se realiza con todos los tipos primitivos de Java (int, float, double, boolean, char, etc... e incluso con algunos Objeto que posteriormente veremos que son especiales).

Existen algunas excepciones.

En el apartado anterior hemos dicho que todos los objetos se pasan por referencia en Java, pero esto no es 100 % cierto, existen algunas excepciones, sobre todo con los objetos que son inmutables.

¿ Qué es un objeto inmutable ?

Un objeto inmutable es aquel objeto en el que cada vez que se realiza una modificación sobre el mismo se crea una copia automáticamente del mismo, es decir, es la máquina virtual de Java la que crea la copia por nosotros. El típico ejemplo de un Objeto inmutable en Java es el objeto «String».

¿ Cómo se pasan los objetos inmutables como String ?

```
public static void main(String[] args) {
    String miCadena = "Hola Mundo";
    modificarCadena(miCadena);
    System.out.println(miCadena);
}
```

```
private static void modificarCadena(String miCadena) {
    miCadena = "Esto es una modificación de un String";
}
```

¿ Cuál será la salida de nuestro programa ?

Hola Mundo

Por lo tanto, queda demostrado que existen excepciones con algunos objetos, pero son los menos casos, y solo pasa con aquellos objetos que son inmutables en Java.

Devolución de un valor

Sentencia return.

Se utiliza para devolver un valor. La palabra clave ***return*** va seguida de una expresión que será evaluada para saber el valor de retorno.

> Esta **expresión** puede ser compleja o puede ser simplemente el nombre de un objeto, una variable de tipo primitivo o una constante.

Consideraciones:

- El tipo del valor de retorno debe coincidir con el tipoDevuelto que se ha indicado en la declaración del método.
- Si el método no devuelve nada (tipoDevuelto = void) la instrucción return es opcional.
- Un método puede devolver un tipo primitivo, un array, un String o un objeto. En el caso de arrays y objetos, devuelve una referencia a ese array u objeto.
- Un método tiene un único punto de inicio, representado por la llave de inicio {. La ejecución de un método termina cuando se llega a la llave final } o cuando se ejecuta la instrucción return.
- Usar un único return por método, y colocarlo como última sentencia del método se considera una muy buena práctica

Justificación: Este consejo es una consecuencia de uno de los principios de la programación estructurada. Dicho principio establece que los programas deberían tener un único punto de entrada y un sólo punto de salida. Además, un return puede considerarse como un GOTO hacia el final de la función. El uso de un único return por método, colocado consecuentemente al final de la misma, facilita tanto la depuración como la adaptabilidad de los programas.

Por ejemplo, si estamos depurando un programa y queremos saber que valor devuelve una función o método, en caso de existir un único return, bastará con imprimir el valor devuelto antes de invocar dicho return; o poner un punto de ruptura en tal sentencia. Si por el contrario tuviésemos, pongamos por ejemplo, 5 sentencias return en la función, tendríamos que colocar y gestionar 5 puntos de ruptura durante la depuración del programa.

Como ejemplo de mejora de la adaptabilidad, supongamos ahora que tenemos un método que devuelve una longitud en centímetros. Supongamos también que tenemos que adaptar nuestro programa para que trabaje en el sistema anglosajón, devolviendo ahora en lugar de una cantidad en centímetros, la misma cantidad pero expresada pulgadas. La adaptación requerida sería tan fácil como invocar una función cmApulgadas(..) sobre el valor devuelto por la función. Si hay un sólo return, sólo tendremos que modificar una línea de código. Si tenemos, pongamos por ejemplo, 5 sentencias return, tendremos que modificar 5 líneas de código, lo cual es más propenso a errores.

Ejemplo 01

Por ejemplo haremos un método al que se le pasarán dos números enteros y devolverá el mayor.

```
import java.util.Scanner;

public class EjemploMayor {
    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        int x, y, mayor;

        System.out.println("Dame el primer valor :");
        x = teclado.nextInt();
        System.out.println("Dame el segundo valor :");
        y = teclado.nextInt();

        mayor  = mayor(x, y);

        System.out.println("El número mayor es "+mayor);
    }

    public static int mayor(int a, int b){
        int resp = a;
        if (a<b){
            resp = b;
        }
        return resp;
    }
}
```

Una segunda versión del método:

```
public static int mayorV2(int a, int b){
        int resp;
        resp = Math.max(a, b);
        return resp;
    }
```

Ejemplo 02

Realiza un programa que tenga un método al que le pasamos un número, calcula el factorial y nos devuelve el resultado.

```java
import java.util.Scanner;

public class Factorial {
    public static void main(String[] args) {
        Scanner read = new Scanner(System.in);
        int num;
        long factorial;

        do {
            System.out.print("Introduce un número para saber su factorial (
            num = read.nextInt();
            if (num < -1)
                System.out.println("No se puede calcular el factorial de "
            else {
                if (num != -1 && num >= 0) {
                    factorial = factorial(num);
                    System.out.println("El factorial de " + num + " es " +
                }
            }
        } while (num != -1);
    }

    public static long factorial(int num) {
        long fact = 1;
        for (int i = num; i > 0; i--) {
            fact *= i;
        }
        return fact;
    }
}
```

```
Introduce un número para saber su factorial (Para salir: -1) :5
El factorial de 5 es 120
Introduce un número para saber su factorial (Para salir: -1) :0
El factorial de 0 es 1
Introduce un número para saber su factorial (Para salir: -1) :-8
No se puede calcular el factorial de -8
Introduce un número para saber su factorial (Para salir: -1) :-1

Process finished with exit code 0
```

El término Void

Que un método devuelva o no un valor es opcional. En caso de que devuelva un valor se declara el tipo que devuelve. Pero si no necesita devolver ningún valor, se declara como tipo del valor devuelto, la palabra reservada **`void`**. Por ejemplo:

```
public static void haceAlgo() {
      . . .
}
```

Cuando no se devuelve ningún valor, la instrucción **`return`** no es necesaria.

Ejemplos.

Ejemplo 1: Método void básico.

```
public class VoidEjemplo {
    public static void main(String[] args) {
        printMensaje();
    }

    public static void printMensaje() {
        System.out.println("Hola Mundo !!!");
    }
}
```

En este ejemplo, el método printMensaje se declara con un tipo de retorno void. Simplemente imprime "¡Hola, mundo!" en la consola y no devuelve ningún valor.

Ejemplo 2: Método void con parámetros, primer caso un literal como argumento.

```
public class Ejemplo2 {
    public static void main(String[] args) {
        saluda("Alicia");
    }

    public static void saluda(String name) {
        System.out.println("Hola, " + name + "!");
```

```
    }
}
```

Aquí, el método saluda toma un parámetro String name e imprime un mensaje de saludo. El método se declara con un tipo de retorno void, lo que indica que no devuelve ningún valor.

Ejemplo 3: Método void con parámetros.

```
public class Ejemplo3 {
    public static void main(String[] args) {
        String nombre="Antonio";
        saluda(nombre);
    }

    public static void saluda(String name) {
       System.out.println("Hola, " + name + "!");
    }
}
```

Aquí, el método saluda toma un parámetro String name, valor que recibe como argumento de la variable nombre, e imprime un mensaje de saludo. El método se declara con un tipo de retorno void, lo que indica que no devuelve ningún valor.

Consejos y buenas prácticas

Utiliza nombres descriptivos: Cuando definas los métodos de void, utiliza nombres descriptivos que indiquen claramente la acción que realiza el método.

```
public static void calculaSuma() {
    // body
}
```

Efectos secundarios: Asegúrate de que se utilizan los métodos void para las acciones que tienen efectos secundarios, como modificar los estados de los objetos, imprimir en la consola o escribir en un archivo.

Evita las Declaraciones de Retorno: No utilices declaraciones return en los métodos void

Documentación: Documenta siempre la finalidad y el comportamiento de los métodos utilizando comentarios o Javadoc para garantizar la claridad y su mantenimiento.

Sobrecarga de métodos

La **sobrecarga** de métodos (overload) es un mecanismo poderoso que nos permite asignar el mismo identificador a distintos métodos, la diferencia reside en el tipo o número de parámetros que utilicen. Esto resulta especialmente conveniente cuando se desea llevar a cabo la misma tarea con diferente número o tipos de variables.

Veamos un ejemplo simple:

Supongamos que hemos escrito una clase librería que implementa diferentes métodos para multiplicar dos números, tres números, y así sucesivamente.

Si hemos dado a los métodos nombres confusos o ambiguos, como multiplicar2(), multiplicar3(), multiplicar4(), entonces esa sería una API de clase mal diseñada. Aquí es donde entra en juego la sobrecarga de métodos.

Podemos implementar la sobrecarga de métodos de dos maneras diferentes:

- implementando dos o más métodos que tienen el mismo nombre pero toman un número diferente de argumentos
- implementando dos o más métodos que tienen el mismo nombre pero toman argumentos de diferentes tipos

Ejemplo 01 Número de Argumentos Diferentes

La clase Multiplicador muestra, en pocas palabras, cómo sobrecargar el método multiplicar() simplemente definiendo dos implementaciones que toman un número diferente de argumentos:

```
public class Multiplicador {

    public int multiplicar(int a, int b) {
        return a * b;
    }

    public int multiplicar(int a, int b, int c) {
        return a * b * c;
    }
}
```

Ejemplo 02: Argumentos de Tipos Diferentes

De manera similar, podemos sobrecargar el método multiplicar() haciendo que acepte argumentos de diferentes tipos:

```
public class Multiplicador {

    public int multiplicar(int a, int b) {
        return a * b;
    }

    public double multiplicar(double a, double b) {
        return a * b;
    }
}
```

Además, es legítimo definir la clase Multiplicador con ambos tipos de sobrecarga de métodos:

```
public class Multiplicador {

    public int multiplicar(int a, int b) {
        return a * b;
    }

    public int multiplicar(int a, int b, int c) {
        return a * b * c;
    }

    public double multiplicar(double a, double b) {
        return a * b;
    }
```

Es importante destacar, sin embargo, que **no es posible tener dos implementaciones de métodos que difieran solo en sus tipos de retorno.**

Para entender por qué, consideremos el siguiente ejemplo:

```
public int multiplicar(int a, int b) {
    return a * b;
}

public double multiplicar(int a, int b) {
    return a * b;
}
```

En este caso, el código simplemente no se compilaría debido a la ambigüedad en la llamada al método:

el compilador no sabría qué implementación de multiplicar() llamar.

Ejemplo 03

Programa con dos métodos para calcular el máximo de tres números

```
/** * Demostración de metodos sobrecargados */
public class PruebaSobrecarga {
    public static void main (String[] args) {
        int a=34, b=12, c=56;
        System.out.println("a = " + a + "; b = " + b + "; c = " + c);
        // El primer método
        System.out.println("El mayor de a y b es: " + mayor(a,b));
        // El segundo método
        System.out.println("El mayor de a, b y c es: " + mayor(a,b,c));
    }

// Definición de mayor de dos numeros enteros
public static int mayor (int x, int y) {
    return x>y ? x : y;
}

// Definición de mayor de tres numeros enteros
public static int mayor (int x, int y, int z) {
    return mayor(mayor(x,y),z);
}
```

Ejemplo de salida por pantalla:

```
$>java PruebaSobrecarga
a = 34; b = 12; c = 56
El mayor de a y b es: 34 El mayor de a, b y c es: 56
```

Ejercicios

TablaMult

Programa que escribe la tabla de multiplicar de un número introducido por teclado. El programa debe tener un método que recibe como parámetro un número entero y muestra la tabla de multiplicar de dicho número.

Millas

Programa que dado un valor en millas nos lo traduce a metros. El programa debe tener un método que reciba como parámetro una cantidad (en millas) y nos devuelva la cantidad en metros.

Descuento

Calcular el porcentaje de descuento que nos han hecho al comprar algo. Se debe solicitar la cantidad de tarifa y lo que realmente pagamos. Realizarlo creado un método al que le pasamos ambos parámetros y nos devuelve el resultado.

Angulo

Dado el valor de un ángulo, sería interesante saber su seno, coseno y tangente. Escribir una función que muestre en pantalla los datos anteriores.

Fecha

Proqrama que lee una fecha con el siguiente formato (dd-mm-aaaa) y nos dice si es correcta o no. El programa debe tener dos métodos, uno para saber si un año es bisiesto o no lo és, y otro al que le pasamos los datos necesarios y nos devuelve un boolean indicando si es correcta o no.

Primo

Programa que debe ir pidiendo números al usuario hasta que el usuario introduzca el cero. Para cada numero introducido, hay que decir si es primo o no. Hay que recordar que un numero es primo si solo es divisible por si mismo y por 1. El 1 no es primo por convenio.

El programa debe tener un método que permita saber si un número (que se le pasará como parámetro) es primo o no.

Comparar

Escribe un programa que sirva para probar un método que reciba dos números y que devuelva 0 si son iguales, 1 si el primero
es mayor que el segundo y -1 si el segundo es mayor que el primero.

NumerosAmigos

Dos números son amigos, si cada uno de ellos es igual a la suma de los divisores del otro.

Por ejemplo, 220 y 284 son amigos, ya que:

- Suma de divisores de 284: 1 + 2 + 4 + 71 + 142 = 220
- Suma de divisores de 220: 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284

Hacer un método que determine si dos números dados como argumentos son amigos o no.

A continuación realizar un programa que muestre todas las parejas de números amigos
menores o iguales a n, siendo n un número introducido por teclado.

El programa debe usar el método amigo previamente definido.

MuestraCalendario

Escribe un método que muestre el calendario para un mes en el siguiente formato:

Lunes	Martes	Miércoles	Jueves	Viernes	Sábado	Domingo
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30						

Los datos de entrada para el método serán el número de días en el mes y el día de la semana (1 a 7) en la que comienza ese mes.

En el ejemplo mostrado en la figura, habría que pasar como argumentos los siguientes datos de entrada mostrar_Calendario(30, 7)

Nota: Para escribir los números bien alineados, utilizar el carácter tabulador, '\t'.

Menú

Realizar un programa que muestre por pantalla el siguiente menú:

1.- Calcular el factorial de un número.
2.- Hallar si dos números son amigos
3.- Resolver ecuación de 2o grado
4.- Salir del programa

El usuario podrá elegir cada una de las opciones del menú.

Si la opción pulsada no es ninguna de las anteriores el programa debe mostrar un mensaje informando de tal error.

Para cada opción el programa mostrará en pantalla la solución y esperará a que se pulse una tecla. Una vez pulsada volverá a mostrar el menú anterior.

Notas

Programa que lee una calificación numérica entre 0 y 10 de cada alumno del aula, se acabará cuando a la pregunta ¿Otro alumno si/no? se le conteste que no.

Debes escribir un método para solicitar la nota (validándola) el método debe devolver una nota válida.

Otro método al que se le pase una nota numérica y devuelva una calificación alfabética:

- de 0 a <3 Muy Deficiente.
- de 3 a <5 Insuficiente.
- de 5 a <6 Suficiente.

- de 5 a <6 Bien.
- de 6 a <9 Notable
- de 9 a 10 Sobresaliente

Hora

Programa que pide como datos de entrada la hora, los minutos y los segundos.

Haz métodos para pedir estos valores (por separado) y que los valide.

Otro método que calcule la hora, minutos y segundos que serán, transcurrido un segundo y devuelva un String.

En el programa principal se mostrarán los 10 segundos siguientes a la hora introducida.

Dados

Escribir un programa que simule el lanzamiento de dos dados y muestre por pantalla la frecuencia de los resultados de mil lanzamientos. El programa debe pedir el número de lanzamientos a simular.

Escribe un método de devuelva un String con el resultado del lanzamiento.

Nota: debes utilizar el método predefinido de Java **Math.random()** que devuelve un valor aleatorio real entre 0 y 1.

Capicuas

Escribe un programa que solicite números enteros positivos e indique si son o no capicúas.

El programa solicitará números hasta que se introduzca uno negativo y usará un método para determinar si un número es capicúa.

PrimosMenoresN

Programa que pide un número n (validarlo) y nos muestra todos los números primos menores al valor n.

SumaDigitos

Define un método llamado sumaDeDigitos() que reciba como argumento un número entero y devuelva la suma de los dígitos que lo componen.

Escribe un programa donde se lea el número, se calcule esta suma y se presente en pantalla el resultado.

Unos ejemplos de ejecución son:

- para la entrada 102 el resultado es 3
- para 1800 el resultado es 9.

Almacen

En un almacén, cada producto es identificado por un número y un dígito adicional. Este segundo valor es un dígito de autoverificación, que se calcula a partir del primer número, de la siguiente forma:

a) Multiplicar la posición de las unidades y cada posición alternada por dos. Ejemplo: Si el número del producto es 543211 obtenemos 583412.
b) Sumar los dígitos no multiplicados y los resultados de los productos obtenidos en el apartado 1. En el ejemplo obtenemos 23.
c) Restar el número obtenido en el apartado 2 del número más próximo y superior a éste, que termine en cero. En el ejemplo sería 30-23=7. El resultado será el dígito de autoverificación.

Escribe una función que dado un número, devuelva su dígito de autoverificación.

Escribir un programa que vaya leyendo números de productos y compruebe mediante el dígito de autoverificación si el número introducido es correcto o no. El proceso se repetirá hasta que se introduzca un cero como número de producto.

Packages

Si el método esPrimo() va a ser usado en tres programas diferentes se puede copiar y pegar su código en cada uno de los programas, pero hay una solución mucho más elegante y práctica.

Los métodos de un determinado tipo (por ejemplo, funciones matemáticas) se pueden agrupar para crear un paquete (**package**) que luego se importará desde el programa que necesite esos métodos.

Cada paquete se corresponde con un directorio. Por tanto, si hay un paquete con nombre "matematicas" debe haber un directorio llamado también "matematicas" en la misma ubicación del programa que importa ese paquete (normalmente el programa principal).

Los métodos se pueden agrupar dentro de un paquete de dos maneras diferentes.

- Puede haber subpaquetes dentro de un paquete; por ejemplo, si quisiéramos dividir los métodos matemáticos en métodos relativos al cálculo de áreas y volúmenes de figuras geométricas y métodos relacionadas con cálculos estadísticos, podríamos crear dos directorios dentro de matemáticas con nombres geometría y estadística respectivamente.

  Estos subpaquetes se llamarían **matematicas.geometria** y **matematicas.estadistica**.

- Otra manera de agrupar los métodos dentro de un mismo paquete consiste en crear varios ficheros dentro de un mismo directorio. En este caso se podrían crear los ficheros Geometria.java y Estadistica.java.

Entenderemos mejor todos estos conceptos con un ejemplo completo. Vamos a crear un paquete con nombre **matematicas** que contenga dos clases: **Varias** (para métodos matemáticos de propósito general) y **Geometría**. Por tanto, en el disco duro, tendremos una carpeta con nombre **matematicas** que contiene los ficheros **Varias.java** y **Geometria.java**.

El contenido de estos ficheros se muestra a continuación:

```
package matematicas;
/** * Funciones matemáticas de propósito general
 *
 * */
public class Varias {
/**
* Comprueba si un número entero positivo es primo o no.
* Un número es primo cuando únicamente es divisible entre
* él mismo y la unidad.
*
* @param x un número entero positivo
* @return <code>true</code> si el número es primo
* @return <code>false</code> en caso contrario
*/
```

```
public static boolean esPrimo(int x) {
    boolean primo = true;
    for (int i = 2; i < x; i++) {
        if ((x % i) == 0) {
                primo = false;
        }
    }
    return primo;
}

 /**
* Devuelve el número de dígitos que contiene un número entero
*
* @param x un número entero
* @return la cantidad de dígitos que contiene el número
*/

public static int digitos(int x) {
    int n=0;
    if (x == 0) {
          n= 1;
    }
    else {
        n = 0;
        while (x > 0) {
            x = x / 10;
            n++;
        }
    }
    return n;
}
}
```

Se incluye a continuación **Geometria.java**. Recuerda que tanto Varias.java como Geometria.java se encuentran dentro del directorio matematicas.

```
package matematicas;

/**
* Funciones geométricas
* */

public class Geometria {

/**
* Calcula el volumen de un cilindro.
```

```
* Tanto el radio como la altura se deben proporcionar en las
* mismas unidades para que el resultado sea congruente.
*
* @param r radio del cilindro
* @param h altura del cilindro
* @return volumen del cilindro
*/

public static double volumenCilindro(float r, float h) {
    return Math.PI * r * r * h;
}

/** * Calcula la longitud de una circunferencia a partir del radio.
*
* @param r radio de la circunferencia
* @return longitud de la circunferencia
*/

public static double longitudCircunferencia(float r) {
    return 2 * Math.PI * r;
}
}
```

Observa que en ambos ficheros se especifica que las clases declaradas (y por tanto los métodos que se definen dentro) pertenecen al paquete matematicas mediante la línea **package matematicas**. Ahora ya podemos probar los métodos desde un programa externo. El programa PruebaFunciones.java está fuera de la carpeta matematicas, justo en un nivel superior en la estructura de directorios.

```
import matematicas.Varias;
import matematicas.Geometria;
import java.util.Scanner;

/** * Prueba varias funciones
*
* @author Luis José Sánchez
*/

public class PruebaFunciones {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n;
        float r, h;
        // Prueba esPrimo()
        System.out.print("Introduzca un número entero positivo: ");
        n = Integer.parseInt(sc.nextLine());
```

```
            if (matematicas.Varias.esPrimo(n)) {
                  System.out.println("El " + n + " es primo.");
             }
            else {
                 System.out.println("El " + n + " no es primo.");
            }
            // Prueba digitos() System.out.print("Introduzca un número entero p
            n = Integer.parseInt(sc.nextLine());
            System.out.println(n + " tiene " + matematicas.Varias.digitos(n) +
            // Prueba volumenCilindro()
            System.out.println("Cálculo del volumen de un cilindro");
            System.out.print("Introduzca el radio en metros: ");
            r = sc.nextFloat();
            System.out.print("Introduzca la altura en metros: ");
            h = sc.nextFloat();
            System.out.println("El volumen del cilindro es "
                           +matematicas.Geometria.volumenCilindro(r,h) + " m3"
      }
}
```

Las líneas:

```
import matematicas.Varias;
import matematicas.Geometria;
```

Cargan las clases contenidas en el paquete matemáticas y, por tanto, todos los métodos contenidos en ellos. No vamos a utilizar el comodín de la forma import matemáticas.*; ya que está desaconsejado su uso en el estándar de codificación en Java de Google.

Observa que se escriben por orden alfabético, se escribe antes Geometría y luego Varias. El uso de un método es muy sencillo; hay que indicar el nombre del paquete, el nombre de la clase y finalmente el nombre del método con los parámetros adecuados.

Por ejemplo, como vemos en el programa anterior, matematicas.Varias.digitos(n) devuelve el número de dígitos de n; siendo matemáticas el paquete, Varias la clase, dígitos el método y n el parámetro que se le pasa al método.

Ejercicios

EJERCICIO 1.

Crea la clase **Geometria** y la clase Varias dentro del paquete matematicas y úsalas como en el ejemplo.

EJERCICIO2.

Crea una clase **Varias** dentro del paquete matematicas que contenga los siguientes métodos. Recuerda que puedes usar unas dentro de otras si es necesario. Observa bien lo que hace cada método ya que, si los implementas en el orden adecuado, te puedes ahorrar mucho trabajo.

- **boolean esCapicua(int x)**: Devuelve verdadero si el número que se pasa como parámetro es capicúa y falso en caso contrario.
- **boolean esPrimo(int n):** Devuelve verdadero si el número que se pasa como parámetro es primo y falso en caso contrario.
- **int siguientePrimo(int x)**: Devuelve el menor primo que es mayor al número que se pasa como parámetro.
- **long potencia(int b, int potencia)**: Dada una base y un exponente devuelve la potencia.
- **int digitos(int num)**: Cuenta el número de dígitos de un número entero.
- **int voltea(int num)**: Le da la vuelta a un número.
- int digitoN(int num, int n): Devuelve el dígito que está en la posición n de un número entero. Se empieza contando por el 0 y de izquierda a derecha.
- **int posicionDeDigito(int digit)**: Da la posición de la primera ocurrencia de un dígito dentro de un número entero. Si no se encuentra, devuelve -1.
- **boolean quitaPorDetras(int num, int n)**: Le quita a un número n dígitos por detrás (por la derecha).
- **boolean quitaPorDelante(int num, int n)**: Le quita a un número n dígitos por delante (por la izquierda).
- **void pegaPorDetras(int num, int digit):** Añade un dígito a un número por detrás.
- **void pegaPorDelante(int num, int digit):** Añade un dígito a un número por delante.
- **int trozoDeNumero(int i, int j):** Toma como parámetros las posiciones inicial y final dentro de un número y devuelve el trozo correspondiente.
- **long juntaNumeros(int num1, int num2)**: Pega dos números para formar uno.

EJERCICIO3.

Crea una clase que usando el paquete matematicas pueda:

- Muestra los números primos que hay entre 1 y 1000.
- Muestra los números capicúa que hay entre 1 y 99999

Expresiones regulares

Una expresión regular define un patrón de búsqueda para cadenas de caracteres.

La podemos utilizar para comprobar si una cadena contiene o coincide con el patrón. El contenido de la cadena de caracteres puede coincidir con el patrón 0, 1 o más veces.

Algunos ejemplos de uso de expresiones regulares pueden ser:

- para comprobar que la fecha leída cumple el patrón dd/mm/aaaa.
- para comprobar que un NIF está formado por 8 cifras, un guión y una letra.
- para comprobar que una dirección de correo electrónico es una dirección válida.
- para comprobar que una contraseña cumple unas determinadas condiciones.
- Para comprobar que una URL es válida.
- Para comprobar cuántas veces se repite dentro de la cadena una secuencia de caracteres determinada.
- Etc.

El patrón se busca en el String de izquierda a derecha. Cuando se determina que un carácter cumple con el patrón este carácter ya no vuelve a intervenir en la comprobación.

Símbolos comunes en expresiones regulares

Expresión	**Descripción**
.	Un punto indica cualquier carácter
^expresión	El símbolo ^ indica el principio del String. En este caso el String debe contener la expresión al principio.
expresión$	El símbolo $ indica el final del String. En este caso el String debe contener la expresión al final.
[abc]	Los corchetes representan una definición de conjunto. En este ejemplo el String debe contener las letras a ó b ó c.
[abc][12]	El String debe contener las letras a ó b ó c seguidas de 1 ó 2
[^abc]	El símbolo ^ dentro de los corchetes indica negación. En este caso el String debe contener cualquier carácter excepto a ó b ó c.
[a-z1-9]	Rango. Indica las letras minúsculas desde la a hasta la z (ambas incluidas) y los dígitos desde el 1 hasta el 9 (ambos incluidos)
A|B	El carácter | es un OR. A ó B

Expresión	Descripción
AB	Concatenación. A seguida de B

Meta caracteres

Expresión	Descripción
\d	Dígito. Equivale a [0-9]
\D	No dígito. Equivale a [^0-9]
\s	Espacio en blanco. Equivale a [\t\n\x0b\r\f]
\S	No espacio en blanco. Equivale a [^\s]
\w	Una letra mayúscula o minúscula, un dígito o el carácter _ Equivale a [a-zA-Z0-9_]
\W	Equivale a [^\w]
\b	Límite de una palabra.

En Java debemos usar una doble barra invertida \\

Por ejemplo para utilizar \w tendremos que escribir \\w. Si queremos indicar que la barra invertida en un carácter de la expresión regular tendremos que escribir \\\\.

Cuantificadores

Expresión	Descripción
{X}	Indica que lo que va justo antes de las llaves se repite X veces
{X,Y}	Indica que lo que va justo antes de las llaves se repite mínimo X veces y máximo Y veces. También podemos poner {X,} indicando que se repite un mínimo de X veces sin límite máximo.
*	Indica 0 ó más veces. Equivale a {0,}
+	Indica 1 ó más veces. Equivale a {1,}
?	Indica 0 ó 1 veces. Equivale a {0,1}

Para usar expresiones regulares en Java se usa el package **java.util.regex**

Contiene las clases **Pattern** y **Matcher** y la excepción **PatternSyntaxException.**

Clase **Pattern:** Un objeto de esta clase representa la expresión regular. Contiene el método

compile(String regex) que recibe como parámetro la expresión regular y devuelve un objeto de la clase Pattern.

La clase **Matcher:** Esta clase compara el String y la expresión regular. Contienen el método matches(CharSequence input) que recibe como parámetro el String a validar y devuelve true si coincide con el patrón. El método find() indica si el String contienen el patrón.

Ejemplos de Expresiones Regulares en Java:

Comprobar si el String cadena contiene exactamente el patrón (matches) "abc"

```
Pattern pat = Pattern.compile("abc");
Matcher mat = pat.matcher(cadena);
if (mat.matches()) {
    System.out.println("SI");
} else {
    System.out.println("NO");
}
```

Comprobar si el String cadena contiene "abc"

```
Pattern pat = Pattern.compile(".*abc.*");
Matcher mat = pat.matcher(cadena);
if (mat.matches()) {
    System.out.println("SI");
} else {
    System.out.println("NO");
}
```

También lo podemos escribir usando el método find:

```
Pattern pat = Pattern.compile("abc");
Matcher mat = pat.matcher(cadena);
if (mat.find()) {
    System.out.println("Válido");
} else {
    System.out.println("No Válido");
}
```

Comprobar si el String cadena empieza por "abc"

```
Pattern pat = Pattern.compile("^abc.*");
Matcher mat = pat.matcher(cadena);
```

```
if (mat.matches()) {
    System.out.println("Válido");
} else {
    System.out.println("No Válido");
}
```

Comprobar si el String cadena empieza por “abc” ó “Abc”

```
Pattern pat = Pattern.compile("^[aA]bc.*");
Matcher mat = pat.matcher(cadena);
if (mat.matches()) {
    System.out.println("SI");
} else {
    System.out.println("NO");
}
```

Comprobar si el String cadena está formado por un mínimo de 5 letras mayúsculas o minúsculas y un máximo de 10.

```
Pattern pat = Pattern.compile("[a-zA-Z]{5,10}");
Matcher mat = pat.matcher(cadena);
if (mat.matches()) {
    System.out.println("SI");
} else {
    System.out.println("NO");
}
```

Comprobar si el String cadena no empieza por un dígito.

```
Pattern pat = Pattern.compile("^[^\\d].*");
Matcher mat = pat.matcher(cadena);
if (mat.matches()) {
    System.out.println("SI");
} else {
    System.out.println("NO");
}
```

Comprobar si el String cadena no acaba con un dígito.

```
Pattern pat = Pattern.compile(".*[^\\d]$");
Matcher mat = pat.matcher(cadena);
if (mat.matches()) {
```

```
            System.out.println("SI");
        } else {
            System.out.println("NO");
        }
```

Comprobar si el String cadena solo contienen los caracteres a ó b.

```
        Pattern pat = Pattern.compile("(a|b)+");
        Matcher mat = pat.matcher(cadena);
        if (mat.matches()) {
            System.out.println("SI");
        } else {
            System.out.println("NO");
        }
```

Comprobar si el String cadena contiene un 1 y ese 1 no está seguido por un 2.

```
        Pattern pat = Pattern.compile(".*1(?!2).*");
        Matcher mat = pat.matcher(cadena);
        if (mat.matches()) {
            System.out.println("SI");
        } else {
            System.out.println("NO");
        }
```

Expresión regular para comprobar si un email es válido.

```
package ejemplo1;
import java.util.Scanner;
import java.util.regex.*;

public class Ejemplo1 {
    public static void main(String[] args) {
       Scanner sc = new Scanner(System.in);
       String email;
       System.out.print("Introduce email: ");
       email = sc.nextLine();
       Pattern pat = Pattern.compile("^[\\w-]+(\\.[\\w-]+)*@[A-Za-z0-9]+(\\
       Matcher mat = pat.matcher(email);
       if(mat.find()){
          System.out.println("Correo Válido");
       }else{
          System.out.println("Correo No Válido");
```

```
        }
    }
}
```

Hemos usado la siguiente expresión regular para comprobar si un email es válido:

```
"^[\\w-]+(\\.[\\w-]+)*@[A-Za-z0-9]+(\\.[A-Za-z0-9]+)*(\\.[A-Za-z]{2,})$"
```

[\\w-]+	Inicio del email El signo + indica que debe aparecer uno o más de los caracteres entre corchetes: \\w indica caracteres de la A a la Z tanto mayúsculas como minúsculas, dígitos del 0 al 9 y el carácter _ Carácter – En lugar de usar \\w podemos escribir el rango de caracteres con lo que esta expresión quedaría así: [A-Za-z0-9-_]+
(\\.[\\w-]+)*	A continuación:El * indica que este grupo puede aparecer cero o más veces. El email puede contener de forma opcional un punto seguido de uno o más de los caracteres entre corchetes.
@	A continuación debe contener el carácter @
[A-Za-z0-9]+	Después de la @ el email debe contener uno o más de los caracteres que aparecen entre los corchetes
(\\.[A-Za-z0-9]+)*	Seguido (opcional, 0 ó más veces) de un punto y 1 ó más de los caracteres entre corchetes
(\\.[A-Za-z]{2,})	Seguido de un punto y al menos 2 de los caracteres que aparecen entre corchetes (final del email)

Antes de nada, no existe una expresión regular para email que sea 100% fiable, puesto que hay muchos formatos válidos de email <http://w3.unpocodetodo.info/utiles/regex-ejemplos.php?type=email> y muy complejos. Aquí vamos a usar una expresión regular más o menos sencilla extraída de ese enlace:

```
[^@]+@[^@]+\.[a-zA-Z]{2,}.
```

Significa lo siguiente, un email válido está compuesto de:

[^@]+	cualquier caracter que no sea @ una o más veces seguido de
@	una @ seguido de
[^@]+	cualquier caracter que no sea @ una o más veces seguido de
\.	un punto seguido de
[a-zA-Z]{2,}	dos o más letras minúsculas o mayúsculas

Un ejemplo en código java de esta expresión regular:

```
String emailRegexp = "[^@]+@[^@]+\\.[a-zA-Z]{2,}";

// Lo siguiente devuelve true
System.out.println(Pattern.matches(emailRegexp, "a@b.com"));
System.out.println(Pattern.matches(emailRegexp, "+++@+++.com"));

// Lo siguiente devuelve faslse
System.out.println(Pattern.matches(emailRegexp, "@b.com")); // Falta el nom
System.out.println(Pattern.matches(emailRegexp, "a@b.c")); // El dominio fi
```

Expresión regular para fecha

Imaginemos la fecha en formato dd/mm/yyyy. Son grupos de dos cifras separadas por barras. En una expresión regular \d representa una cifra. El día pueden ser una o dos cifras, es decir \d{1,2}, el mes igual y el año vamos a obligar que sean cuatro cifras exactamente \d{4}

Si queremos comprobar que una cadena leída por teclado cumple ese patrón, podemos usar la clase Pattern. A la clase Pattern le decimos el patrón que queremos que cumpla nuestra cadena y nos dice si la cumple o no.

El siguiente ejemplo comprueba si la cadena cumple con la expresión regular. Ten en cuenta que cuando en java metemos un caracter \ dentro de una cadena delimitada por "", debemos "escapar" esta \ con otra \, por ello todas nuestras \ en la expresión regular, se convierten en \\ en nuestro código java.

```
String regexp = "\\d{1,2}/\\d{1,2}/\\d{4}";

// Lo siguiente devuelve true
System.out.println(Pattern.matches(regexp, "11/12/2014"));
System.out.println(Pattern.matches(regexp, "1/12/2014"));
```

```
System.out.println(Pattern.matches(regexp, "11/2/2014"));


// Los siguientes devuelven false
System.out.println(Pattern.matches(regexp, "11/12/14")); // El año no tiene
System.out.println(Pattern.matches(regexp, "11//2014")); // el mes no tiene
System.out.println(Pattern.matches(regexp, "11/12/14perico")); // Sobra "pe
```

Supongamos que queremos que el mes se exprese como "ene", "feb", "mar", ... en lugar de como un número. Cuando hay varias posibles cadenas válidas, en la expresión regular se ponen entre paréntesis y separadas por |. Es decir, algo como esto (ene|feb|mar|abr|may|jun|jul|ago|sep|oct|nov|dic). Si además nos da igual mayúsculas o minúsculas, justo delante ponemos el flag de case insensitive (?i) (la 'i' es de ignore case)

El siguiente código muestra un ejemplo completo de esto.

```
String literalMonthRegexp = "\\d{1,2}/(?i)(ene|feb|mar|abr|may|jun|jul|ago|

// Lo siguiente devuelve true
System.out.println(Pattern.matches(literalMonthRegexp, "11/dic/2014"));
System.out.println(Pattern.matches(literalMonthRegexp, "1/nov/2014"));
System.out.println(Pattern.matches(literalMonthRegexp, "1/AGO/2014"));   //
System.out.println(Pattern.matches(literalMonthRegexp, "21/Oct/2014"));  //

// Los siguientes devuelven false
System.out.println(Pattern.matches(literalMonthRegexp, "11/abc/2014"));   /
System.out.println(Pattern.matches(literalMonthRegexp, "11//2014"));      /
System.out.println(Pattern.matches(literalMonthRegexp, "11/jul/2014perico")
```

Expresión regular para DNI

Este número son 8 cifras seguidas de una letra, que normalmente se escribe en mayúscula. Esta letra es una especie de checksum de las cifras anteriores, por lo que hay un algoritmo para validar que el número completo es correcto. Quedan excluidas las letras 'I', 'O' y 'U'.

Una expresión regular **no va a realizar este checksum**, pero sí nos puede ayudar a hacer una primera comprobación: 8 cifras y una letra mayúscula. La expresión regular puede ser así:

```
\d{8}[A-HJ-NP-TV-Z]
```

El siguiente código muestra un ejemplo completo de la expresión regular de DNI

```
String dniRegexp = "\\d{8}[A-HJ-NP-TV-Z]";

// Lo siguiente devuelve true
System.out.println(Pattern.matches(dniRegexp, "01234567C"));

// Lo siguiente devuelve faslse
System.out.println(Pattern.matches(dniRegexp, "01234567U")); // La U no es
System.out.println(Pattern.matches(dniRegexp, "0123567X")); // No tiene 8 c
```