Programación en Java

Ejercicios

IES Fernando Wirtz Suárez. A Coruña.
Módulo: Programación.
Profesor: Fernando Rodríguez Diéguez

XUNTA DE GALICIA
CONSELLERÍA DE EDUCACIÓN
E ORDENACIÓN UNIVERSITARIA

Unión Europea
Fondo Social Europeo

Índice

Material complementario y videos:
wirtzJava.blogspot.com

Fernando Rodríguez Diéguez
rdf@fernandowirtz.com
Versión 2024-10-27

INTRODUCCIÓN

En este documento hay cuestiones y ejercicios. Para resolver los ejercicios crea en Netbeans una carpeta por cada tema, que se llame como tus iniciales en mayúscula (solo 3 letras) + "t" +*nn* (siendo *nn* número de tema: de 01 a 19). Si tu nombre es José Luís Pérez de Miranda, la carpeta de cada tema sería: JPM*t01,* JPM*t02, etc.*

Dentro de cada carpeta de cada tema, crearemos un PDF (con el mismo nombre que la carpeta) que incluya:

- Portada con los datos del curso, módulo y tu nombre.
- Índice con una entrada para cada ejercicio, con el número de ejercicio y la página en la que se encuentra (en el índice no va el enunciado del ejercicio)
- Cuestiones de cada tema con su respuesta.
- Para cada ejercicio: el número de ejercicio, el enunciado, una captura del terminal en el que se vea funcionando y un pequeño comentario con los errores más importantes a la hora de resolverlo y cualquier otro comentario que creas interesante sobre cada ejercicio.
- El PDF debe tener numeradas las páginas y buena presentación.

Además del PDF, tendrás una carpeta por cada ejercicio, que será un proyecto en NetBeans con la solución del mismo. Esta carpeta se nombrará así:

- Iniciales de usuario (3 caracteres: ej: JMP)
- Tema: ("t" + 2 dígitos, ejemplo t01)
- Número de ejercicio ("e"+ 2 dígitos ejercicio del tema)

Así pues, el primer ejercicio del usuario *JPM* sería el proyecto y carpeta *JPMt01e01* y el ejercicio 7 del tema 3 sería *JPMt03e07*.

Para los ejercicios sencillos, puedes agrupar varios de ellos en un solo proyecto, en el mismo paquete o en distintos paquetes. El proyecto tendría varios *main,* algo no habitual en un caso real, pero puede ser cómodo para los ejercicios de los primeros temas, que apenas tienen un archivo *.java* o dos.

En el documento de apuntes tienes un anexo con instrucciones para instalación y manejo básico de Netbeans, y también en este video: https://www.youtube.com/watch?v=INxJTvbgTiw

Si el ejercicio tiene varios archivos (por ejemplo, distintas formas de hacerlo o pide distintos apartados) se le añadirá letra minúscula al final: a, b, c, etc… por ejemplo *JPMt1e01a.* Si un programa tiene alguna pregunta sobre el mismo, contéstala en el PDF.

A lo largo del curso se realizará un proyecto, que será el juego del *MasterMind*, paso a paso, de forma que en cada capítulo se le va añadiendo más funcionalidades sobre la versión del capítulo anterior. Este proyecto se tratará en el PDF como cualquier otro ejercicio del tema e irá en una carpeta (proyecto) aparte. Se nombrará igual en todos los capítulos, con las iniciales de tu nombre y el texto 'proy' por ejemplo: *JPMproy*.

En el aula virtual se entregará en las fechas indicadas un archivo *zip* con el mismo nombre que la carpeta del tema (ej: JPMt01) que incluirá: el PDF de soluciones, las carpetas con el código de cada ejercicio descritas previamente y la carpeta del proyecto *MasterMind*.

No entregar ningún ejercicio con errores de compilación o ejecución, o que no respondan a lo solicitado. En caso de dudas, preguntar siempre al profesor antes de subirlos.

1. Conceptos Básicos

Cuestiones previas

a) Crea un programa con estas instrucciones e interpreta el resultado, obteniendo conclusiones sobre precedencia de operadores, qué hacer el operador %, división entre enteros, etc.

```
int   a = 2 * 3 + 4;   System.out.println(a);
int   b = 4 + 2 * 3;   System.out.println(b);
int   c = 5 - 10 % 2;  System.out.println(c);
float d = 5 / 2;       System.out.println(d);
float e = 5f / 2;      System.out.println(e);
```

b) Resuelve a mano, sin programa ni calculadora, las siguientes expresiones:

```
-4 * 7 + 20 % 3 / 2 - 5
( ( 3 + 2 ) % 2 - 15 ) / 2 * 5
3 + 6 * 14 % 3;
d =8 + 7 * 3 + 4 * 6 / 2 % 4
```

c) ¿Qué hace este programa? ¿Puede producirse algún error?

```
int a,b,c;
Scanner teclado = new Scanner(System.in);
System.out.print("Introduce valor a: ");
a = teclado.nextInt();
System.out.print("Introduce valor b: ");
b = teclado.nextInt();
c = a * b;
System.out.println("Resultado: " + c);
```

d) ¿Qué hace este programa? Corrígelo sus errores. ¿Qué ocurre para la variable 'b' introducimos cero?

```
int a,b,c;
Scanner teclado = new Scanner(System.in);
System.out.print("Introduce valor a: ");
a = teclado.nextInt();
System.out.print("Introduce valor b: ");
b = teclado.nextInt();
c = a / b;
System.out.println("Resultado: " + c);
```

e) Expresar, utilizando operadores aritméticos, (en una sola línea, como en la cuestión 'b' las siguientes expresiones matemáticas:

$$\frac{m+n}{n} \qquad \frac{\frac{m+n}{p}}{\frac{p-r}{s}} \qquad \frac{\frac{c*r*t}{100}}{\frac{m^2+n^2}{p+q}}$$

f) Indicar el valor de las variables enteras 'a', 'b' y 'c', 'd' sobre las que se efectúan consecutivamente las operaciones:

```
a=3; b=2; c=a*b-b; d=b*b; a=b-a; b++; d+=2;
```

Ejercicios

1.1. Realiza un programa que lea por consola un valor en euros y lo convierta a dólares (suponer que 1 euro es igual a 1,14 dólares).

1.2. Realiza un programa que lea por consola un valor en dólares y lo convierta a euros (suponer que 1 euro es igual a=1,14 dólares).

1.3. Realiza un programa que lea por consola dos números enteros (sin decimales) y nos muestre los resultados de sumar, restar y dividir dichos números. Comprueba que la división responde con decimales

1.4. Queremos conocer los datos estadísticos de una asignatura, por lo tanto, necesitamos un programa al que se le introduzcan por consola el número de suspensos, suficientes, notables y sobresalientes de una asignatura, y nos calcule:

- El tanto por ciento de alumnos que han superado la asignatura.
- El tanto por ciento de notables y sobresalientes de la asignatura.

Trata de minimizar el número de operaciones realizadas. Los datos se introducen en variables sin decimales, pero los porcentajes sí tienen decimales. ¿Tendría algún sentido pedirle al usuario que introdujese el total de alumnos?

1.5. Un departamento de climatología ha realizado recientemente su conversión al sistema métrico. Diseñar un algoritmo para realizar las siguientes conversiones:

- Leer por consola la temperatura dada en la escala Celsius y mostrar su equivalente Fahrenheit (la fórmula de conversión es “F=9/5 ºC+32”). Resultado redondeado a dos decimales.
- Leer la cantidad de agua del pluviómetro en pulgadas y mostrar su equivalente en centímetros (25.5 mm = 1 pulgada). Resultado redondeado a un decimal.

Verifica que has hecho bien la operación de conversión de temperaturas:

Cuando se encuentra 9/5 interpreta que son dos enteros, hace división entera y el resultado es 1 (y no 1,8 que es lo correcto) En cambio si hacemos celsius *9/5, primero hace celsius *9, y como celsius es float, el resultado celsius *9 también lo será, y al dividir entre 5, el numerador sigue siendo float y por eso lo hace bien. Para evitar problemas, cuando queramos decimales, la solución es ponerle: 5.0, o bien 5f, o bien 5d. (5f le dice que es de tipo float, 5d le dice que es de tipo double).

En el redondeo ocurre lo mismo, Math.round genera un entero, al dividir entre 100, seguimos con enteros, y perdemos los decimales. A ese 100 final hay que ponerle 100.0 ó bien 100f.

Así pues, soluciones correctas serían:

```
fahre = Math.round((9f / 5 * celsius + 32)*100)/100f;
fahre = Math.round((9 / 5f * celsius + 32)*100)/100.0;
fahre = Math.round((9.0 / 5 * celsius + 32)*100)/100f;
fahre = Math.round((9 / 5.0 * celsius + 32)*100)/100.0;
fahre = Math.round((celsius * 9 / 5 + 32)*100)/100.0;
fahre = Math.round((celsius * 9 / 5 + 32)*100)/100f;
```

y serían incorrectas:

```
fahre = Math.round((9 / 5 * celsius + 32)*100)/100;
fahre = Math.round((9 / 5f * celsius + 32)*100)/100;
fahre = Math.round((9 / 5 * celsius + 32)*100)/100f;
```

1.6. El coste de un automóvil nuevo para un comprador es la suma total del coste de fábrica del vehículo, más el porcentaje de la ganancia de la tienda (que se aplica sobre el coste de fábrica) y añadiéndole finalmente los impuestos estatales aplicables (sobre el precio de venta calculado ya con beneficio de la tienda).

Suponiendo una ganancia de tienda de 10% y un impuesto del 20%, realiza un programa que lea por consola el coste inicial del automóvil y calcule el coste para el consumidor.

> Nota:
>
> Como, a lo mejor, en algún momento esos porcentajes podrían ser otros, sería buena idea definirlas como constantes al principio de programa, así, si en un futuro nos informan que es otro valor solo tenemos que cambiarlas ahí, y no recorrer todo el código para ver donde hay que cambiarla.
>
> Esta es una filosofía que puedes emplear para todos los programas. Estas constantes tienen el atributo 'final' para evitar que a lo largo del código le cambiemos su valor por error. Una variable *final* no se puede modificar su valor una vez que se le crea. Se suele poner el nombre de la variable en mayúsculas:

1.7. Queremos realizar un pequeño programa para introducirlo en el ordenador de a bordo de nuestro coche y que nos informe del consumo medio del coche cada 100 km. Diseña un programa al que le introduzcamos el kilometraje de la última vez que se repostó, el kilometraje actual, los litros de gasolina que tenía al finalizar la última vez que repostó y la cantidad de gasolina actual.

1.8. Diseñar un programa al que se le introduzcan las edades de cuatro personas y nos calcule la media de edad de los mismos.

(Opcional) Hacer una segunda versión del ejercicio anterior (1.8b) que solo utilice dos variables. ¿Sería posible hacerlo con una sola variable?

1.9. El siguiente programa pretende intercambiar dos variables enteras introducidas previamente, y luego mostrarlas por pantalla. Corrige los errores que encuentres en el código.

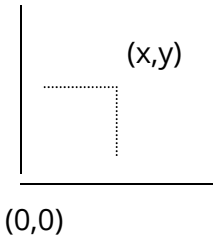
```
package ejercicios;
import java.util.Scanner;
public class Ejercicios {
    public static void main(String[] args) {

        int var1, var2;

        System.out.print("Introduce var1: ");
        var1 = teclado.nextInt();
        System.out.print("Introduce var2: ");
        var2 = teclado.nextInt();
        var1 = var2;
        var2 = var1;
        System.out.println("Ahora var1 es igual a var1");
        System.out.println("Ahora var2 es igual a var2");
  }}
```

¿Sería posible intercambiar los valores de dos variables sin usar ninguna variable adicional? Piensa como hacerlo o busca en internet una solución.

1.10. Programa al que se le introduzcan por consola las coordenadas X e Y de un punto del plano y calcule el área del rectángulo que forma ese punto con el origen de los ejes de coordenadas (supón que solo pueden ser positivos):



1.11. Diseñar un programa al que se le introduzca la longitud de dos catetos de un ángulo recto y nos devuelva el valor de la hipotenusa. Busca en internet como calcular en Java potencias y raíces cuadradas para aplicar el teorema de Pitágoras. Investiga también si existe en Java alguna utilidad que nos haga este cálculo automáticamente.

Nota para expertos:

Aunque el tema de las excepciones se estudiará más adelante, para la resolución de estos ejercicios, cuando tengamos un código problemático que pueda producir un error de ejecución (por ejemplo, una división por cero), podemos incluir las instrucciones problemáticas dentro de un bloque *try...catch* sencillo. Así el programa no se interrumpirá. Ejemplo:

```
try { x=y/z;} catch( Exception e ) {x=0;}
```

Proyecto:

A lo largo del curso vamos a desarrollar un proyecto completo, será el juego del MasterMind. En el este juego, al comenzar, se genera un número al azar de 4 dígitos sin dígitos repetidos. A continuación, el jugador tendrá 10 intentos para adivinarlo.

En cada intento se informa al jugador de cuantos dígitos acertó en su posición correcta y cuantos acertó pero en posición errónea. Esto le irá sirviendo de pista hasta llegar a adivinar el número.

En este video tienes más información sobre la mecánica del juego, en su versión tradicional en la que se juega con colores en vez de con dígitos, pero el funcionamiento es idéntico: https://youtu.be/Eh6fiuNlE5I?si=jlNft3W20DKd6-PP

Ese será el proyecto completo, además le añadiremos funcionalidad adicional, como poder configurar cuantos dígitos tiene el número a adivinar, cuantos intentos tiene disponible el jugador.

También permitiremos guardar información en disco como por ejemplo un ranking de los mejores jugadores.

Todo esto lo iremos desarrollando poco a poco, en cada tema. La tarea para este primer tema es sencilla, se trata de lo siguiente:

Mostrar un mensaje de bienvenida y decirle al usuario que introduzca un número, qué será el número que deberá adivinar el jugador. Suponemos que el número introducido está comprendido entre 100 y 999. A continuación, mostrar cada uno de los dígitos de ese número por separado, cada uno en una variable distinta: unidades, decenas, centenas.

```
Juego del Mastermind!
Introduce el número a adivinar (de 3 dígitos, sin dígitos repetidos):
123
Unidades:3
Decenas:2
Centenas:1
```

2. Estructuras de Control: Condiciones

Cuestiones previas

Evaluar las siguientes expresiones (*verdadero* o *falso*) , teniendo en cuenta los valores de las variables y razona la respuesta:

```
a) i=1                    i >= 10
b) i=1                    i > 0 && i < 10
c) i=1                    i > 0 && i > 10                    //¿tiene sentido?
d) i=-1                   i > 0 || i < 10
e) i=1, j=10              i > 0 && i < 10 || j==1
f) i=1                    !(i==1)
g) i=1, j=10              !(i==10) && j==10
h) i=1, j=10              i > 2 || i < 10 && j==10
i) i=1, j=0, k= -1        i+k <= j-k*3 && k>=2
j) i=3, j=2, k=-1         i==3 || j <=2 && k>0
k) i=3, j=2, k=-1         (i==3 || j <= 2 ) && k > 0
l) año = 2020             año % 4 ==0
m) año = 2000             año % 4 ==0 && año %100 !=0 || año %400==0  //bisiesto
```

Ejercicios

2.1. Hacer programas sencillos que hagan lo siguiente.

a) Leer un número entero y determinar si es mayor de 10.

b) Leer dos números enteros y muestre si el primero es mayor que el segundo.

c) Leer un número entero y de determinar si se trata de un número par o impar.

d) Leer dos números enteros y diga si los dos son mayores de 10 o no lo son.

e) Leer dos números enteros y diga si al menos uno de los dos es mayor de 10.

f) Leer un número entero y decir si está comprendido entre 10 y 20 (ambos incluidos).

g) Leer dos números enteros y diga si uno y solo uno es mayor de 10.

h) Leer dos números y decir si el primero es mayor que el segundo, si es menor o si los dos números son iguales.

i) Leer dos números enteros y diga si el segundo es divisor del primero (prevenir divisiones por cero, que causan error) *Repasar operadores en cortocircuito para hacer un solo if.*

j) Leer dos números enteros y diga si el menor de ellos es divisor del mayor (prevenir divisiones por cero, que causan error) *Repasar operadores en cortocircuito para hacer un solo if.*

k) Leer un número y decir si es múltiplo de 2, de 3 y/o de 10. *Si no es múltiplo de 2 ya no hay que mirar si lo es de 10.*

l) Leer tres números enteros y diga si hay alguno mayor que cero.

m) Leer tres números enteros y diga si hay alguno mayor que cero, pero no todos.

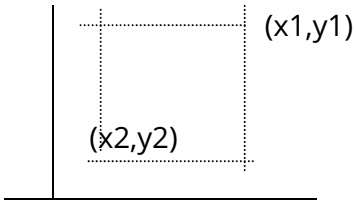
2.2. Diseñar un programa al que se le introduzcan por consola tres números enteros. Si todos son negativos, mostrar el producto de los tres. Si alguno es negativo, pero no todos, mostrar la suma de los tres. En caso de que todos sean positivos sumar los dos primeros y multiplicar dicha suma por el tercero. ¿existe alguna combinación de los valores leídos no contemplado en el algoritmo? (Suponer que el cero es un número positivo). Hacer el programa lo más sencillo posible.

2.3. Para aprobar el curso se valorará la nota del examen, la valoración del trabajo en clase y la nota de un trabajo práctico. Aprobarán los alumnos que estén en alguna de las siguientes situaciones:

- Nota examen mayor o igual a 5
- Nota examen entre 4 y 5, trabajo en clase mayor que cinco y trab. práctico mayor que 5.
- Nota examen entre 4 y 5, y una nota mayor que 8 o en el trabajo práctico o en la valoración del trabajo en clase.

Diseñar un algoritmo al que se le introduzcan por consola la nota del examen, la valoración del trabajo en clase y la nota del trabajo práctico y saque por pantalla si está aprobado o no, todo en con una sola sentencia condicional.

2.4. Programa al que se le introduzcan las coordenadas X e Y de dos puntos del plano, nos diga si lo que forman es un cuadrado o un rectángulo y calcule el área del mismo.



Math.abs(x) obtiene el valor absoluto de x, sin signo.

2.5. Diseñar un algoritmo al que se le introduzca la cantidad de horas, minutos y segundo mostrados en un reloj digital, que verifique que los valores sean correctos y calcule el total de segundos transcurridos desde el comienzo del día. No emplear las clases de fecha de Java.

23:59:57 será una hora correcta y 25:61:88 será una hora incorrecta

2.6. Un sistema de ecuaciones lineales de la forma:

$$ax + by = c$$
$$dx + ey = f$$

puede resolverse utilizando las siguientes fórmulas:

$$x = \frac{ce-bf}{ae-bd} \qquad y = \frac{af-cd}{ae-bd}$$

Realizar un programa que lea por teclado los dos conjuntos de coeficientes (a, b y c, y d, e y f) y calcule los valores para 'x' e 'y' según la fórmula descrita ¿Existen algunos casos para los cuales este algoritmo no funcione? *Nota: cuando en matemáticas se escriben dos variables juntas, por ejemplo 'ce' quiere decir 'c por e'.*

Solo hay que verificar que el denominador es distinto de cero

2.7. Realizar un programa que informe si un año introducido previamente es bisiesto o no. Son bisiestos los años múltiplos de 4 que no sean múltiplos de 100. Como excepción los múltiplos de 400 también son bisiestos. Se puede hacer una primera versión con varias sentencias condicionales y otra más sofisticada con una sola. No usar las clases de fechas de Java.

2.8. La tabla siguiente representa las horas de salida de un bus. Diseña un algoritmo al que se le introduzca el día (1-7) y la hora (9-14), verifique la entrada y nos informe si hay bus o no. Hacer una primera condición que haga la verificación de la entrada de datos y otra única condición para ver si hay bus.

	Lunes	Martes	Miérc.	Jueves	Viernes	Sábado	Domingo
9:00							Si
10:00	Si	Si	Si	Si	Si	Si	Si
11:00							Si
12:00				Si			Si
13:00							Si
14:00	Si	Si	Si	Si	Si		Si

Hay que tratar de hacer un "if" que agrupe todas las condiciones. Piensa como lo dirías si alguien te preguntase a qué horas hay bus y luego trata de pasarlo a una sola condición en Java.

2.9. Realizar un programa al que se le introduzca la hora del día (0 –23) y nos diga que días de la semana hay salida de bus (utilizar la tabla anterior).

2.10. Realizar un programa al que se le introduzcan DIA, MES, AÑO, que verifique que los valores introducidos sean correctos, sin emplear las clases de fechas de Java.

Si primero calculamos los días que tiene un mes/año, luego será muy fácil la verificación de la fecha.

2.11. Realizar un programa al que se le introduzcan DIA, MES, AÑO, que calcule el día siguiente. Suponemos que los valores son correctos, sin emplear las clases de fechas de Java.

Hay que hacer un tratamiento diferente según sea fin de año, fin de mes (no fin de año) o un día no fin de mes.

2.12. Introduciendo dos fechas (día, mes, año), hacer un programa que nos diga cuál de las dos es mayor (agrupa las condiciones de día, mes y año en una sola sentencia condicional). Suponemos que se introducen fechas válidas y no empleamos las clases de fecha de Java.

2.13. Realizar un programa que se le introduzca una nota (un valor entero entre 0 y 10) y nos muestre por pantalla la nota final en texto, con la siguiente equivalencia: Muy deficiente (0,1,2), Insuficiente (3,4), Aprobado (5,6), Notable (7,8) y Sobresaliente (9,10). Haz una versión con if anidados, otra con switch + "case :" y otra con switch + "case ->"

2.14. Diseña un algoritmo capaz de obtener la letra del DNI a partir del número de DNI. Consiste en dividir dicho número entre 23 y tomar el resto de la división asignándole la letra correspondiente según la siguiente tabla (Hacer con *switch*, no empleando ni arrays ni *String*)

RESTO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LETRA	T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

2.15. Haz 3 versiones de un mismo programa, que pida que se introduzcan por teclado 2 números *int* sobre sendas variables llamadas 'a' y 'b' y que incluyan las instrucciones que se muestran abajo. Para cada una de las versiones, ejecútalo varias veces, introduciendo cero para la variable 'b' y finalmente explica la diferencia de comportamiento entre cada una de las versiones:

- Versión a) `if (b !=0 && a%b==0) {r = a/b; System.out.println(r);}`
- Versión b) `if (a%b==0 && b !=0) {r = a/b; System.out.println(r);}`
- Versión c) `if (b !=0 & a%b==0) {r = a/b; System.out.println(r);}`

  Puede ser el mismo código con las tres sentencias juntas, teniendo una sola vigente y las otras dos como comentarios. Comentando unas y descomentando otra vas probando cada caso. Si quieres, no hagas programas y contesta directamente en el PDF de soluciones.

Proyecto:

Sobre el proyecto del tema anterior, añadir las siguientes funcionalidades:

- Verificar que él número introducido está comprendido entre 100 y 999 y que no tiene dígitos repetidos. Si no se cumple eso, se informará por pantalla del error y terminará el programa.

- En caso contrario, se hará desaparecer el número de la pantalla - por ejemplo, con: *System.out.print ("\n\n\n\n\n\n");* - y el jugador introducirá un número para intentar adivinar el número introducido previamente.

- Sobre este intento, se hará la misma verificación que sobre el número a adivinar, esto es, comprendido entre 100 y 999 y que no tiene dígitos repetidos. En caso de que no se cumpla, también terminará el programa informando del error.

- En caso contrario, se informará de cuantos dígitos del intento coinciden en la misma posición que en el número a adivinar y cuantos dígitos coinciden, pero en posiciones equivocadas. La siguiente figura muestra un ejemplo:

```
Juego del Mastermind!
Introduce el número a adivinar (de 3 dígitos, sin dígitos repetidos):
123
========= Aquí los saltos de línea==================================
Bienvenido jugador:
Intenta adivinar el número (3 dígitos, sin dígitos repetidos):
134
Bien colocados:1
Mal colocados:1
```

3. Estructuras de Control: Bucles

Cuestiones previas

Contestar con una frase que es lo que muestran estos programas (intenta contestar sin ejecutarlo):

```
a) for (int i=1; i<=10; i++)      System.out.println(i);
b) for (int i=1; i<=10; i+=2)     System.out.println(i);
c) for (int i=1; i<=10; i--)      System.out.println(i);
d) for (int i=10; i>=1; i--)      System.out.println(i);
   Los siguientes suponiendo int j=10, k=20
e) for (int i=j; i<=k; i++)       System.out.println(i);
f) for (int i=j+1; i<=j+5; i++)   System.out.println(i);
g) for (int i=k; i>=j; i--)       System.out.println(i);
h) i=10; while (i>=j && i<=k)     System.out.println(i);
i) i=10; while (i>=j && i<=k)     { System.out.println(i); i++;}
j) i=10; while (i>=j || i<=k)     { System.out.println(i); i+=3;}
```

Ejercicios

3.1. Mostrar por pantalla la frase “esto es un bucle” 10 veces.

3.2. Mostrar por pantalla los n primeros números naturales, siendo n el valor tecleado previamente.

3.3. Diseñar un algoritmo que permita introducir números positivos y nos vaya informando uno a uno si es par o impar. El programa finalizará cuando se introduzca un número negativo. Hacer dos versiones, una primera con una lectura de teclado adelantada (antes de entrar en el bucle) y otra versión en la que la lectura de datos se haga únicamente al principio del bucle.

3.4. Diseñar un programa capaz de leer un valor entero, que verifique que esté comprendido entre 1 y 10, y mostrar su tabla de multiplicar.

3.5. Sumar una cantidad de números que se van introduciendo, siempre que sean positivos hasta que se teclee el -1.

> Para sumar valores e ir acumulando esa suma se suele emplear una estructura con la siguiente forma. Antes del bucle: **acumulador = 0**. Y dentro del bucle **acumulador = acumulador + nuevo valor**.

3.6. Diseñar un programa que muestre los números pares comprendidos entre 100 y 1 en orden descendente.

3.7. Indica la funcionalidad del siguiente programa. ¿Tiene algún error?¿Falta alguna llave? ¿Podrías reducir el recorrido del *for* para que el programa sea más rápido, manteniendo su funcionalidad?

```
import java.util.Scanner;
public class Ejercicios {
    public static void main(String[] args) {
        int num;
        Scanner teclado = new Scanner(System.in);
        System.out.print("Introduce un número natural entre 1 y 1000: ");
        num = teclado.nextInt();
        if (num <0 && num > 1000)
           System.out.println("Numero incorrecto.");
        else
           for (int i=num;num>=1;i--)
               if (num % i == 0) System.out.println(i);
  }     }
```

3.8. Calcular la suma de los 100 números siguientes a uno tecleado previamente.

3.9. Sumar una cantidad de números que se van introduciendo por teclado hasta que la suma de los mismos valga más de 100.

3.10. Programa al que se le introduzcan las edades de los alumnos (-1 para finalizar) y nos informe si hay alguno menor de edad.

> En cuanto encuentre un menor de edad, la respuesta va a ser "Sí hay alguno", aunque luego sigamos introduciendo edades, esta situación ya no va a cambiar, ojo al hacer el "if".

3.11. Diseñar un algoritmo capaz de leer dos valores enteros, m y n, y nos muestre los números comprendidos entre ellos ordenados ascendentemente, junto con el valor de elevar cada uno de esos números al cuadrado. No se sabe si m > n, pero se desea utilizar una sola sentencia repetitiva tanto si m > n como si m < n.

3.12. Diseñar un algoritmo al que se le introduzcan las notas de los exámenes de una clase con decimales. Se introducirá –1 para indicar que no hay más notas. El algoritmo informará del total de notas introducidas, calculará la nota mínima, máxima , la media, si hay algún 5.0 exacto, y el porcentaje de aprobados. Hay que validar que estén entre 0 y 10.

- Hay que pensar primero el bucle: el código ser repite mientras..... Una vez aclarado esto, puedes pensar qué es lo que se hace cada vez (en este caso, qué hago con cada nota introducida)
- Para calcular la nota máxima hay que tener guardada en una variable la máxima hasta ese momento y al leer una nueva nota, compararla la nueva con la guardada, y si la nueva es mayor que la guardada, guardamos la nueva, ya que ahora es la máxima y así para todas las iteraciones del bucle (para empezar puedes decir que la máxima nota es -1)
- Para calcular la media hay que ir *acumulando* los valores y la cantidad de elementos, y al final dividir lo acumulado entre la cantidad de elementos.

3.13. Programa al que le introduzcas un número y que garantice que es positivo, es decir si lo introducimos menor o igual que cero nos obligará a volver a introducirlo las veces que sea necesario hasta que se introduzca uno positivo.

> Cuando tenemos que hacer algo una o más veces, pero por lo menos una, el do...while puede ser mejor opción que el while.

3.14. Diseñar un algoritmo que informe de los números perfectos que hay entre 1 y 10000. Un número perfecto si es igual a la suma de todos los divisores menores que él.

3.15. Sin ejecutarlo, intenta explicar que hace este programa.

```
public class Ejercicios {
    public static void main(String[] args)  {
        int hora=0;
        for (int i=1;i<=7*24;i++) {
            System.out.println (hora);
            if (++hora > 23) hora=0;
        }
    } //fin main
} //fin clase
```

3.16. Programar el siguiente juego. El ordenador genera un número al azar entre 0 y 99 que habrá que adivinar. El jugador intentará adivinar el número y el programa le responderá si es mayor, menor o si ha acertado. Se dispone de un máximo de 5 intentos.

3.17. Diseñar un algoritmo al que se le introduzcan las notas de un alumno en los nueve módulos de un ciclo y nos informe si puede ir a la FCT (todos aprobados). ¿Habría que solicitarle al usuario siempre las notas de los nueve módulos?

3.18. Mostrar las tablas de multiplicar del 1 al 9.

3.19. Programa que se le introduzcan números por consola y que informe si están ordenados ascendentemente o no. El programa finaliza si se introduce cero o en cuanto se detecte que los números no están ordenados.

> Para cada número, puedo ver si es mayor que el anterior. En ese caso estarán ordenados ascendentemente, en caso contrario no.

3.20. Programa que muestre los número primos comprendidos entre 2 y 1000. Hacer los bucles con un número mínimo de iteraciones (se permite el uso de *break*) sabiendo que:

- Un número primo es solo divisible por 1 y el mismo.
- Los números primos son todos impares salvo el 2.
- Los divisores de un número siempre son menores que la mitad de dicho número (salvo el mismo).
- Si un número no es divisible por 2 ya no va a serlo por ningún otro número par.

3.21. La serie de Fibonacci se compone a partir de números naturales, empezando por 0 y 1, y construyendo cada nuevo número de la serie como la suma de los dos anteriores elementos de la serie, así pues, el tercero sería 1 (es 0 +1), el siguiente 2 (1+1), luego 3 (2+1), luego 5 (3+2), 8 (5+3) y así sucesivamente. El siguiente programa pretende mostrar 12 primeros números de esta serie ¿Funciona correctamente? Es decir, muestra: 0,1,1,2,3,5,8,13,21,34,55,89. Si no es así, corrígelo.

```
public class Ejercicios {
    public static void main(String[] args) {
      int ant=1, ant2=0, num;
      for (int i=1; i<=12; i++){
           num = ant + ant2;
           System.out.println(num);
           ant = num; ant2 = ant;
      }
    }
}
```

3.22. Hacer un programa que permita calcular el sueldo final de los vendedores de un concesionario de coches. Para cada uno se introducirá: sueldo base, cantidad de horas extras realizadas y cantidad de ventas. El sueldo final será el sueldo base más las horas extras (se pagan a 100€ la hora) y un plus en función de las ventas: entre 10 y 20 ventas →500€, entre 21 y 30 ventas → 1000€ y si supera las 30 ventas → 1300 €. Después de tratar cada vendedor se preguntará al usuario y hay más vendedores a tratar, finalizando el programa si contesta negativamente. Al final el programa mostrará el total de ventas del concesionario y el salario final del vendedor que más ventas ha realizado (supón que al menos introducen los datos de un empleado).

> En este tipo de ejercicios hay que distinguir el bucle (repito el proceso mientas.....) y luego las operaciones que hago con cada uno.

3.23. Programa que permita introducir números y nos muestre el resultado de sumarlos, hasta que introduzca un número mayor que 1000. Si no se introduce un número mayor que 1000 el programa finalizará después de introducidos 15 números. *(Ojo: es fácil equivocarse y hacer que solicite un numero de más o bien que no sume correctamente)*

3.24. Realizar un programa que permita introducir el sexo ('H','M') y la edad de los 30 trabajadores de una fábrica. El programa debe obligar a que las edades que se introduzcan estén comprendidas entre 16 y 70 años. El programa nos mostrará la edad y el sexo del más joven y también mostrará la media de edad de las mujeres. Finalmente nos informará si hay alguno con más de 60 años. Si se introduce cero como edad el programa terminará en ese momento sin esperar a que introduzcan los 30 trabajadores.

Usar tipo char para almacenar sexo.

3.25. Mostrar por pantalla alternativamente "hola" y "adiós" hasta completar x líneas, siendo x un número entero positivo tecleado previamente (ojo: el número x puede ser par o impar)

3.26. Hacer cuatro programas que pinten 9 líneas según los siguientes gráficos:

a.)	b.)	c.)	d.)
0000000000	1	1111111111	9
1111111111	22	222222222	98
2222222222	333	33333333	987
3333333333	4444	4444444	9876
……	……	…….	…….

Cada apartado se resuelve con un for dentro de otro for

3.27. La suma de los dígitos de los números múltiplos de tres es también un múltiplo de tres. Realizar un programa que compruebe si esta afirmación es correcta o no para los primeros 10000 números enteros positivos.

3.28. Sacar por pantalla la suma, la media y el producto de los números pares comprendidos entre dos números enteros tecleados previamente. No sabemos si el que introduce primero es menor que el segundo. No incluir en los cálculos los números introducidos, solo los intermedios.

3.29. Hacer un programa para llevar el control de venta de entradas. Lo primero que ha de hacer es pedir la cantidad de entradas que se pondrán a la venta, después irá solicitando la cantidad de entradas que quiere comprar, estando limitado a un máximo de 10 por cliente. El programa finalizará cuando se agoten las entradas mostrando la cantidad de entradas que se ha llevado el que más ha comprado. Pensar primero el bucle y luego lo que hay que hacer en cada iteración

3.30. Modificar el programa anterior para que si un cliente introduce un número negativo o mayor que 10 le informe de su error y le obligue a meter correctamente las entradas que desea, las veces que sea necesario hasta que lo haga bien.

3.31. Programa al que se le introduzcan 30 números y si la suma de los mismos es par nos muestre los 3 primeros que introducimos, y si es impar nos muestre los tres últimos que introducimos. Hay que garantizar que cada uno de dichos números está comprendido entre 0 y 1000.

3.32. (Opcional) Programa al que se le pase un número entero y que lo muestre en binario. Deberá hacer divisiones sucesivas e ir componiendo un número (*long*) con los restos obtenidos hasta que el cociente será cero. Pista: cada cero y uno obtenido como resto hay que situarlo en la posición adecuada del resultado: unidades, decenas, centenas, etc.

3.33. (Opcional) Amplia el programa anterior para que el usuario pueda introducir previamente la base destino y el programa pueda pasar a binario (base 2), ternario (base 4) u octal (base 8).

Proyecto:

Sobre el proyecto del tema anterior, añadir las siguientes funcionalidades:

- La petición de intento para el jugador se convierte ahora en un bucle, para tener 5 intentos para adivinarlo. El programa finalizará o bien si acierta o bien si se le acaban los intentos. Se informará al usuario de dicha situación: *"Enhorabuena, lo has adivinado"* o por el contrario *"Lo sentimos, te has quedado sin intentos"*.

- En caso que un intento sea erróneo (bien por cantidad de dígitos, bien por repetidos), ese intento no se contará de cara al límite de intentos.

- En la siguiente figura se muestra la apariencia que debe tener el programa:

```
Juego del Mastermind!
Introduce el número a adivinar (de 3 dígitos, sin dígitos repetidos):
123
========= Aquí los saltos de línea================================
Bienvenido jugador:
Intenta adivinar el número (3 dígitos, sin dígitos repetidos):
Es tu intento número: 1
567
Bien colocados:0
Mal colocados:0

Intenta adivinar el número (3 dígitos, sin dígitos repetidos):
Es tu intento número: 2
567
Bien colocados:0
Mal colocados:0

Intenta adivinar el número (3 dígitos, sin dígitos repetidos):
Es tu intento número: 3
789
Bien colocados:0
Mal colocados:0

Intenta adivinar el número (3 dígitos, sin dígitos repetidos):
Es tu intento número: 4
456
Bien colocados:0
Mal colocados:0

Intenta adivinar el número (3 dígitos, sin dígitos repetidos):
Es tu intento número: 5
455
El intento no puede contener dígitos repetidos
Intenta adivinar el número (3 dígitos, sin dígitos repetidos):
Es tu intento número: 5
124
Bien colocados:2
Mal colocados:0

Lo sentimos, te has quedado sin intentos
```

4. Cadenas

Como vimos en el primer capítulo, cuando se solicita al usuario un número con los métodos de la clase *Scanner: nextInt*(), *nextFloat*(), etc. y luego un String con *nextLine*() se produce un error, ya que interpreta el *<ENTER>* que introducimos al final del número como el contenido de la cadena, por lo que la cadena toma valor vacío. Para evitar esto, a partir de este momento, se recomienda dejar de usar *nextInt*(), *nextInt*(), etc. y usar solo *nextLine*() con la conversión correspondiente, por ejemplo:

```
int i = Integer.parseInt (teclado.nextLine());
```

Cuestiones previas

Primero haz en el ordenador el ejercicio 4.1. para tomar contacto con los String y luego explica que hacen los siguientes fragmentos de código, suponiendo que las variables utilizadas están definidas previamente y están inicializadas con un valor cualquiera:

a)
```
if (cad.substring(0,2).equals(cad.substring(cad.length()-2,cad.length())))
```

b)
```
b=false;
for (int i=0;i<= cad.length()-2;i++)
    if(cad.charAt(i)==cad.charAt(i+1)) b = true;
```

¿Por qué el bucle va hasta *length()-2* en vez de hasta *length()*, o *length()-1* ?

c)
```
res = nom.toUpperCase().charAt(0) +"." + ape1.toUpperCase().charAt(0)+"." +
      ape2.toUpperCase().charAt(0) +".";
```

Ejercicios

Por defecto usar clase String. Se especificará cuando se deba usar la clase StringBuilder.

4.1. Realizar un programa al que se le introduzca una cadena por teclado y haga lo siguiente:

a. Mostrar por pantalla el contenido de la cadena pasada a mayúsculas y minúsculas.
b. Decir si en la cadena aparece el carácter 'x'.
c. Decir si la cadena tiene más de 10 posiciones.
d. Decir si la cadena contiene el carácter 'x' a partir de la cuarta posición.
e. Crear una cadena formada por las 5 primeras posiciones de la cadena.
f. Crear una cadena formada por las 5 últimas posiciones de la cadena.
g. Decir si la cadena es igual a la cadena "`hola`".
h. Convertir la cadena de entrada a una variable de tipo int, suponiendo que dicha cadena es un número, esto es, contiene solo dígitos decimales (del 0 a 9)
i. Convertir la cadena de entrada a una variable de tipo int, suponiendo que dicha cadena es un número hexadecimal, esto es, contiene solo dígitos decimales (del 0 a 9) y letras de la A a F.
j. Si se encuentra con en su interior con "prueva" sustituir por "prueba".
k. Decir si la primera posición de la cadena es igual a la última.
l. Decir cuántos dígitos numéricos hay en la cadena.
m. Decir si la cadena es un palíndromo (se lee igual hacia adelante como hacia atrás)
n. Crear una cadena que sea igual a la introducida, pero con la primera y última posiciones intercambiadas. *Ejemplo: si introducen: "abcde", obtendría "ebcda".*

4.2. Diseña un algoritmo capaz de obtener la letra del NIF a partir del número de DNI. Consiste en dividir dicho número entre 23 y tomar el resto de la división asignándole la letra correspondiente según la siguiente tabla. Almacena las letras del NIF en una cadena.

RESTO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LETRA	T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

4.3. Realiza un programa que solicite que se le introduzcan una cadena y un carácter y nos muestre cuantas veces está contenido el carácter en la cadena.

4.4. Realiza un programa que muestre una contraseña generada aleatoriamente teniendo las siguientes limitaciones: Tendrá entre 5 y 10 posiciones que solo pueden ser letras entre la 'a' y la 'j'.

4.5. (Opcional) Realiza un programa que muestre una contraseña generada aleatoriamente teniendo las siguientes limitaciones: Tendrá entre 5 y 10 posiciones que solo pueden ser letras entre la 'a' y la 'j' pero sin letras repetidas.

4.6. Realiza un programa al que se le introduzca un email y nos devuelva el nombre del dominio es decir, lo que está entre la arroba y el punto (Ejemplo: *info@mundo-r.com* mostraría *mundo-r*). Opcionalmente, hacer una segunda versión que contemple que pueda haber varios puntos en el dominio (Ejemplo: *info@clientes.mundo-r.com* mostraría *clientes.mundo-r*)

4.7. Un algoritmo de encriptado monoalfabético consiste en la sustitución de una letra por otra a lo largo de todo el mensaje, por ejemplo las A por F, las B por X, las C por M. Obviamente si la A pasa a ser F, ninguna otra letra pasará F. Hacer un programa que le introduzca una cadena en mayúsculas y muestre la cadena encriptada (todo lo que no sean letras mayúsculas se deja intacto: números, espacios en blanco, etc.). Hacer una segunda versión que funcione con mayúsculas y minúsculas.

4.8. Realizar un programa que solicite una cadena, un número que indica una posición de la cadena y una letra. El programa reemplazará sobre la misma cadena, el carácter que hubiera en la posición indicada por la letra introducida. Hacer dos versiones, la primera con *String* y otra con *StringBuilder*.

4.9. Escribe un programa que solicite que se introduzca por teclado un nombre completo y una edad y muestre el siguiente mensaje. Hacer una versión utilizando la clase *Format* y otra versión con el método *System.out.printf.*

```
Hola, me llamo Pepe Pérez López y tengo 20 años
```

suponiendo que se introduce *Pepe Pérez López* como nombre completo y *20* como edad.

4.10. Realizar un programa que solicite la entrada de 10 cadenas de caracteres y construya una cadena con el primer carácter de cada cadena. Finalmente mostrará dicha cadena por pantalla.

4.11. Describe que realiza el código siguiente e indica si contiene algún error.

```
java.util.Scanner teclado = new java.util.Scanner(System.in);
 System.out.println("Introduce una cadena:");
 String cadena = teclado.nextLine();
 StringBuilder cadenaSB = new StringBuilder(cadena);
 int pos;
 do {
     pos = cadenaSB.indexOf(" ");
     if (pos!=-1) cadenaSB.deleteCharAt(pos);
 } while (pos != -1);
 System.out.println(cadenaSB);
```

4.12. Realizar un programa que solicite la entrada de una cadena de 6 posiciones, que todas sean dígitos y sin repetidos. Si no cumple esas condiciones, el usuario deberá introducirla de nuevo hasta que lo haga correctamente.

4.13. (Opcional) Realizar un programa al que se le introduzca un número decimal y lo convierta a un String con su representación hexadecimal. Hay que hacerlo codificando el proceso sin usar las clases de Java y luego compararlo con el resultado ofrecido por las clases de Java para convertir de decimal a hexadecimal.

Para convertir a hexadecimal hay que hacer divisiones sucesivas entre 16 de los cocientes obtenidos hasta que el cociente sea cero. El número será la concatenación de restos, convirtiendo los mayores de 9 a la letra correspondiente: 10-> A, 11->B...15->F

4.14. Realizar un programa al que se le introduzca una cadena por teclado, que la convierta a *StringBuilder*, y aplicando métodos de esa clase, haga lo siguiente:

a. Borrar el carácter que haya en la posición 3.
b. Insertar una 'x' en la posición 5.
c. Sustituir el carácter de la posición 1 por una 'z'.
d. Borrar los caracteres entre la posición 5 y 10.
e. Darles la vuelta a todos los caracteres del *StringBuilder*.
f. Convertir el *StringBuilder* en cadena.

Habrá que verificar en algunos casos que la cadena tiene una longitud mayor que la de la posición indicada, sino producirá errores.

4.15. Realizar un programa en el que el usuario introduzca un texto y sustituya sus posiciones impares por asteriscos, por ejemplo: "abcdefg" cambie las posiciones impares pasaría a: "a*c*e*g"

Proyecto:

Sobre el proyecto del tema anterior, añadir las siguientes funcionalidades:

- Convertir tanto el número a adivinar como los intentos, a cadenas de caracteres. Seguimos con las mismas limitaciones, es decir, cada posición de la cadena debe de ser un dígito numérico y no puede haber repetidos.

- La longitud del número a adivinar, y también de los intentos, pasan a ser de 4 dígitos, pero en el programa deben tratarse esos dos parámetros como variables. A esas variables se les asignan valores al comenzar el programa y que luego no se modifican. Así, si en un futuro quisiésemos cambiar esa longitud o el número de intentos, solo cambiando el valor asignado a esas variables en un único sitio del programa, todo seguiría funcionando sin más cambios.

5. Funciones

5.1. Realizar funciones que realicen los siguientes supuestos, y un *main*() desde el que se llame a dichas funciones:

- Función llamada *par()* que se le pasa un entero y devuelve *true* si es par, *false* si no lo es.
- Función llamada *mayor()* que se le pasan 3 *double* y devuelve el mayor de ellos.
- Función llamada *sumaIntervalo()* que le pasan dos *long* y devuelve otro *long* con la suma de los números comprendidos entre los números pasados (el primero es menor que el segundo, y ambos mayores que cero, en caso contrario devuelve -1)
- Función llamada *contarCeros()* que se le pasa una cadena y devuelve la cantidad de ceros que tiene.
- Función llamada *aleatorio()* que se le pasan dos valores enteros devuelve un entero al azar comprendido entre esos dos valores (el primero es menor que el segundo, y ambos mayores que cero, en caso contrario devuelve -1)

5.2. Programa que presente un menú y permita calcular repetidas veces a) el área de círculo –necesitará el radio- , b) el área de cuadrado –necesitará el lado - , c) el área de triángulo –necesitará base y altura – d) Salir. Usar funciones para cada una de las 3 opciones.

5.3. Hacer una función llamada *CalcularDiasMes* que se le pase como parámetro un año y un mes y devuelva los días que tiene ese mes, teniendo en cuenta los años bisiestos. A continuación, realizar un programa al que se le introduzca una fecha y nos informe de los días pasados desde el 1 de enero de ese año (no usar clases Java de fechas).

5.4. Partiendo de la función del programa anterior, hacer un programa al que se le introduzcan dos fechas del mismo año y nos informe de los días comprendidos entre ellas (no usar clases Java de fechas)

5.5. Programa que calcule el factorial de números menores de 20. El programa permite al usuario meter los números que desee y finalizará cuando meta un número menor que 1 o mayor que 20. Crea las funciones que consideres útiles y que puedas reutilizar en otros programas.

5.6. Realiza una función llamada *cantidadDivisores* al que se le pase como parámetro un número entero y devuelva el número de divisores o bien cero si el número es negativo. Usa dicha función en un *main.*

5.7. Un número primo es aquel que solo tiene como divisores el número 1 y a él mismo. Usando la función del programa anterior, haz un programa que muestre la suma de los números primos comprendidos entre 1 y 1000.

5.8. Realizar un programa al que se le introduzcan dos números enteros positivos y nos diga cuál de los dos tiene más divisores (usar función previa).

5.9. El siguiente programa tiene una variable global llamada *saldo* sobre la que se ejecutan las funciones *ingresar()* y *retirar()* que aumentan y reducen el saldo respectivamente, no pudiendo quedar el *saldo* por debajo de cero. ¿Hay algún error? ¿Cuánto vale la variable *saldo* al finalizar la ejecución del programa?

```
public class Ejercicio {
    public static  int saldo=0;
    public static void main(String[] args){
      boolean ok = false;
      ingresar(30);
      retirar(10);
      if (retirar(10)==false) System.out.println("No se puede retirar tanto");
        System.out.println("Saldo final: "+ saldo);
    }
    static void ingresar(int i){
        saldo+=i;
    }
    static boolean retirar(int i){
        if (i>saldo) return false;
            saldo-=i;
            return true;
     }
}
```

5.10. Este código es erróneo ¿Qué mostraría por pantalla? ¿Por qué?

```
public static void main(String[] args)  {
   float saldo = 500f;
   saldo = ingresar(saldo, 400f);
   if (ingresar(saldo, 400f)>1000f)
            System.out.println("Ya tienes más de 1000 euros: " + saldo);
       else System.out.println("No tienes más de 1000 euros: " + saldo);
}

static float ingresar (float saldo, float increm){ return saldo+increm;}
```

5.11. ¿Qué mostraría este código por pantalla? ¿Por qué?

```
public static void main(String[] args) {
  float saldo = 500f;
  ingresar(saldo, 600f);
  if (saldo > 1000f) System.out.println("Ya tienes más de 1000 euros: " + saldo);
  else               System.out.println("No tienes más de 1000 euros: "+ saldo);
}

static void ingresar(float saldo, float increm) {saldo += increm;    }
```

Proyecto:

El código realizado hasta el momento es poco mantenible, ya que tiene un método *main* muy extenso y tiene mucho código duplicado. Debemos refactorizar este método para pasar todo a funciones. Haremos los siguientes cambios:

- Desarrollar un método al que le pasa una cadena y verifique si todos valores son dígitos numéricos y otro método al que se le pasa una cadena y verifique si tiene alguna posición duplicada. Serán llamados cuando se crea el número a adivinar al principio y también para validar cada intento del jugador.

- La longitud del número a adivinar y la cantidad de intentos son unos parámetros globales de la aplicación así que conviértelos en variables globales. También el objeto Scanner.

- Aun así, el *main* sigue siendo muy grande. Podemos hacer un método que valide cada intento del jugador. Se le podría pasar el intento y el número a adivinar y mostraría por pantalla cuantos hay bien colocados y cuantos mal colocados. Puede devolver un *boolean* indicando si el jugador ha ganado o no.

6. Clases y Objetos

Cuestiones:

Examina la clase *Producto* y contesta a las siguientes cuestiones:

```
public class Producto {
    public String nombre;
    public double precio;
    public double IVA;
    private double descuento;

    public Producto (String nom, double p, double IVA) {
        this.nombre = nom;
        this.precio = p;
        this.IVA = IVA;
        this.descuento=0;
    }

    public double calcularPrecioFinal () {
        double prFin= this.precio + (this.precio * this.IVA);
        double prFinDesc = prFin - (prFin * this.descuento);
        return prFinDesc;
    }
    public void setDescuento (double desc) {
        this.descuento = desc;
    }
}
```

a) ¿Cuántos atributos tiene?
b) ¿Cuántos constructores tiene?
c) ¿Cuántos métodos tiene?
d) ¿Cómo harías para crear en el *main()* de un programa dos productos? Por ejemplo, '*p1*' que sea un ordenador, que tiene un impuesto del 21% y '*p2*' una barra de pan, que tiene el 4%
e) ¿Con qué descuento se crean?
f) ¿Cómo mostrarías por pantalla el precio final de ambos productos creados?
g) Aplícale un descuento del 10% al ordenador y muestra de nuevo el precio final.
h) Cambia el nombre a la barra de pan.
i) ¿Qué ocurre si hago en el main: `p1.IVA = p2.IVA;` ?
j) ¿Qué ocurre si hago en el main: `p1.descuento = p2.descuento;` ?
k) ¿Qué ocurre si hago en el main: `p1 = p2;` ?
l) ¿Cómo harías para que el objeto `p1` tuviese el mismo contenido que `p2` pero fuesen variables independientes que puedan tomar valores diferentes más adelante?
m) ¿Qué ocurre si hago `if (p1 ==p2) System.out.println("Iguales");` ?
n) Haz un main con el siguiente código:

```
Producto p3 = new Producto ("impresora",50d, 0.21);
Producto p4 = p3;
p3.precio = 60;
System.out.println(p4.precio);
```

¿Qué mostrará por pantalla?¿Por qué?

Ejercicios:

6.1. A partir de la clase *MovilPrepago* que te proporcionará el profesor, haz un programa en el que se cree un teléfono con unas características determinadas y luego ejecute una serie de actividades como recargar 10 euros, llamar 30 segundos, navegar 10MB, etc. Viendo cómo se va modificando el saldo.

Las propiedades de un móvil prepago son:

- int numeroMovil (9 dígitos)
- float costeMinutoLlamada (euros, con dos decimales)
- float costeConsumoMB (euros, con dos decimales)
- float saldo (euros, con dos decimales)

Tiene un único constructor con la siguiente firma:

MovilPrepago(long nM, float cML, float cMB, float s)

Los métodos son:

- void **efectuarLlamada** (int segundos): reduce el saldo. Si el saldo no es suficiente, se corta la llamada
- void **navegar**(int MB) análogo a efectuar una llamada
- boolean **recargar** (int importe): aumenta el saldo, debe ser múltiplo de 5 euros, sino devuelve *false*
- float **consultarSaldo** ()

Lo primero de debemos hacer es crear una instancia de un móvil (con el constructor) con unos valores cualesquiera, por ejemplo:

```
MovilPrepago miMovil = new MovilPrepago(900900900L, 0.1f, 0.5f, 10f);
```

y después llamar a los métodos y mostrar por pantalla el atributo saldo del móvil.

6.2. A partir de la clase *MovilPrepago* del ejercicio anterior, crea un programa en el que el usuario dé de alta un teléfono con unas características determinadas y luego le permita mediante un menú hacer operaciones como consultar saldo, recarga, hacer llamada, navegar. Se proporciona un modelo (*modeloMenu)* a modo de esqueleto, con el menú y las funciones necesarias, en las que solo tienes rellenar los huecos.

Es similar al anterior, primero creamos un móvil:

```
MovilPrepago miMovil = new MovilPrepago(900900900L, 0.1f, 0.5f, 10f);
```

O pidiéndole los datos al usuario. Luego ya el menú sobre ese móvil.

6.3. A partir de la clase *CuentaCorriente* que te proporcionará el profesor, crea un programa en el que se creen una cuenta corriente (siempre se crean con saldo inicial cero) y mediante un menú le permita al usuario hacer operaciones como ingresar, retirar y consultar saldo en la cuenta. El único atributo público de la cuenta es el IBAN (String de dígitos/letras). *Otros atributos privados son saldo, contadorIngresos, porcentajeComision, pero como son privados, no tenemos acceso a ellos.* Los métodos son:

- **constructor** (String Iban, float porcComision): se le pasa el nombre de la cuenta y el porcentaje de comisión para las retiradas de dinero. También fija saldo inicial a cero
- void **ingresar** (float importe): aumenta el saldo. Si se hacen 3 ingresos consecutivos, sin ninguna retirada en medio, se regala 0,7 euros al usuario *(pero de esto no tenemos que preocuparnos, es el método de la clase que nos proporciona el que lo hace).*
- boolean **retirar** (float importe): reduce el saldo solo si es posible ya no puede quedar negativo (devuelve si se ha podido efectuar la retirada o no). Cada retirada tiene una comisión asociada.
- float **getSaldo** (): devuelve el saldo actual en la cuenta.
- **setComision** (float porcentaje). Cambia la comisión de retirada.
- **getPorcentajeComision**() devuelve un float con el porcentaje de comisión a aplicar en retiradas.

6.4. Haz un programa como el anterior, pero creando dos cuentas en vez de una. Habrá una opción de menú nueva que será *"Cambiar cuenta activa"* para pasar de una cuenta a otra. Tendremos una variable de tipo *CuentaCorriente* que le puedes llamar *cuentaActiva* que unas veces apuntará a una cuenta y otras veces a la otra

Recuerda que las variables de tipo Objeto, a diferencia de los tipos primitivos, son apuntadores a los objetos, si tenemos los objetos 'cuenta1' y 'cuenta2' creados con su constructor, podemos hacer luego una tercera variable 'cuentaActiva' sin constructor y hacer 'cuentaActiva=cuenta1' o bien 'cuentaActiva=cuenta2' cuando nos interese.

El programa al principio hará algo así:

```
CuentaCorriente cuenta1 = new CuentaCorriente ("ES010001",1.5f);
CuentaCorriente cuenta2 = new CuentaCorriente ("ES020002",0.5f);
CuentaCorriente cuentaActiva = cuenta1;  //sin constructor.
```

En alguna opción de menú permitirá cambiar cuentaActiva de cuenta1 a cuenta2 y viceversa.

Haz una segunda versión de la clase CuentaCorriente (puedes llamarle CuentaCorriente2) , en la que la comisión de retirada sea común para todas las cuentas. Haz una versión del programa (puedes llamarle 6.4b) que use esta nueva clase. El constructor ya no tendrá como parámetro el porcentaje de comisión ya que es común para todas las cuentas.

Se puede fijar al comienzo del programa el porcentaje de comisión y luego el programa es similar al anterior, salvo que el método de setComision y getComision serán estáticos:

```
CuentaCorriente2.setComision(0.1f);
cuenta1 = new CuentaCorriente2 ("ES010001");
cuenta2 = new CuentaCorriente2 ("ES020002");
cuentaActiva = cuenta1;
```

6.5. A partir de la clase *Ahorcado* que te proporcionará el profesor, crea un programa que permita al usuario jugar al ahorcado. Desconocemos los atributos de la clase *Ahorcado* ya que son privados, pero sus métodos públicos son:

- **Constructor**(String txtAdivinar, String txtPista). Crea el juego con la frase a adivinar y con un texto de pista para el jugador.
- **Constructor**(String txtAdivinar). Crea el juego con la frase a adivinar, sin pistas.
- boolean **probarLetra** (char x). Comprueba si la letra pasada como parámetro está en la frase a adivinar, en caso afirmativo, devuelve *true*, sino *false.*
- boolean **adivinada**(). Devuelve *true* si se ha adivinado la frase, *false* en caso contrario.
- boolean **perdio**(). Devuelve *true* si se ha llegado al número máximo de intentos sin adivinar la frase, *false* en caso contrario.
- char **leerLetra** (). Pide al usuario por consola una letra y devuelve dicha letra.
- void **pintar** (). Dibuja un "tablero" en la consola, con el estado del juego.

Como en ejercicios anteriores empezaremos creando una instancia de Ahorcado con uno de los constructores que tiene. Luego habrá un bucle para jugar mientras no adivine la palabra secreta y no pierda. Dentro del bucle leeremos la letra por teclado, probará si es correcta o no y pintará el tablero.

6.6. A partir de las clases *Nim* y *Consola*, crea un programa que juegue al NIM contra la máquina. Se trata de un tablero con 3 filas de palillos, con 3, 5 y 7 palillos respectivamente. Dos jugadores por turnos alternos (el usuario y la máquina) han de retirar palillos del tablero perdiendo el que se quede el último palillo. En cada turno se pueden retirar uno o más palillos, pero solo de una misma fila.

Por lo tanto, la estructura del programa será la siguiente: primero se crea el juego. Luego habrá un bucle mientras no gane ninguno de los dos y, dentro del bucle, primero piensa la máquina, luego hace su jugada, y se comprueba si ha ganado. Si no ha ganado, introduces los valores de tu jugada (fila y cantidad de palillos) y juegas. Se comprueba si has ganado y así sucesivamente hasta que gane uno de los dos.

Desconocemos los atributos de la clase *Nim* ya que son privados, pero sus métodos públicos son:

- **Constructor** () por defecto. Se crea una instancia del juego.
- boolean **juega** (int fila, int cant). Valida los datos y retira la cantidad de palillos indicada de la fila indicada devolviendo true. Devuelve false si los parámetros no son válidos. Las filas van de 0 a 2.
- boolean **fin**() devuelve *true* si se alcanzó el final del juego (un solo palillo en el tablero).
- int **piensa** (). Se invoca para que la máquina piensa su jugada y devuelve un entero cuyas decenas son la fila (0,1,2) y las unidades son los palillos que retira (entre 1 y 7), pero no implica que haga dicha jugada. Lo normal será llamar luego al método *juega* con los parámetros obtenidos con *piensa().*

La clase Consola, tiene los siguientes métodos públicos estáticos:

- **pintarTablero** (Nim tablero). Se le pasa un tablero de Nim y lo pinta por consola.
- Int **leerEntero**(texto). Pide un valor por consola y valida que sea un entero de forma que si mete letras o valores incorrectos no "casca" y vuelve a pedir que se introduzcan correctamente. El texto que se pasa como parámetro es el que le dice al usuario lo que tiene que introducir, así no hace falta meter antes un *System.out.println*. Ejemplo: *x=leerEntero("introduzca fila");* hará internamente ya *System.out.println ("introduzca fila")*

Hay que tener en cuenta que los usuarios hablamos de filas 1,2 y 3 mientras que la clase Nim habla de filas 0,1 y 2.
Haz un programa que empiece siempre la máquina (ganará siempre) y, si quieres, haz otro en que empieces tú, a ver si eres capaz de ganarle.
Por último, si no quieres, no es necesario usar el método *leerEntero(),* puedes usar la lectura de teclado habitual: *nextInt(), nextLine(),* etc.

En general, procuraremos siempre de separar la parte de interface de usuario (*println*, *nextInt*, etc...) de la lógica del programa. Así podremos transportar fácilmente esa lógica a otro tipo de programas, con otra interface (consola, interface gráfica, página web, etc.)

En este programa, la clase *Nim* no depende de la interface de usuario, para cambiar a otro tipo de entorno simplemente sustituiremos la clase Consola por otro tipo de sistema de interacción con el usuario.

Trata de seguir esta filosofía en las clases que desarrolles.

Crearemos una instancia del juego y luego un bucle mientras no sea fin de juego. Alternaremos movimientos de la máquina: piensa() y juega() con movimientos del jugador: leerEntero() y juega().
Los métodos de la clase Consola son estáticos, no hay que crear instancia.

6.7. Implementar una clase llamada *Circulo*, que tiene solo la propiedad radio y los métodos *setRadio, getRadio, calcularCircunferencia, calcularSuperficie, calcularDiametro* además de un constructor. Crea un programa que use esta clase creando un par de círculos y usando los métodos creados. ¿El atributo *radio* puede ser privado?

6.8. Implementar una clase llamada *EjemplarLibro* que se va a usar en una biblioteca y que tiene los siguientes atributos:

- Título de libro (se le pone en el momento del alta)
- Fecha de adquisición (es la fecha del sistema en el momento del alta)
- Número de ejemplar: 1, 2, 3, etc. (de un mismo libro, la biblioteca tiene varios ejemplares)
- Prestado (sí /no). Inicialmente no está prestado.

Además, tiene los siguientes métodos:

- **Constructor** 1: cuando es el primer ejemplar de un determinado título, se le pasa como parámetro solo el título del libro. El resto de datos los puede calcular él.
- **Constructor** 2: se le pasa como parámetro un libro y copia todos sus atributos salvo el número de ejemplar que será 1 más el del libro pasado.
- **prestar ()**: si no está prestado lo marca como prestado y devuelve true¸ si está prestado no hace nada y devuelve false.
- **devolver ()**: si está prestado lo marca como no prestado y devuelve true¸ si no está prestado no hace nada y devuelve false.
- **toString** (): Genera un String con el nombre, la fecha entre paréntesis y el número de ejemplar entre corchetes. Este método se usará para sacar por pantalla de forma cómoda los datos de un libro.

Haz un main() que cree 4 libros (probando ambos constructores), que realice algún préstamo y devolución, y finalmente muestre los libros -con *toString()*-.

6.9. Ejercicios de fechas. Usando las clases de Java para el manejo de fechas, realiza programas con la siguiente funcionalidad:

a) Introducir tu fecha de nacimiento y muestre cuantos días han pasado hasta ahora mismo.
b) Introducir una fecha y un número de días y calcule la fecha que se obtiene al incrementar dichos días a la fecha.
c) Introducir dos horas de reloj y nos dé la diferencia entre ambas en segundos.
d) ¿Cuántos años bisiestos ha habido desde el año 1 dC?
e) Introducir una fecha y mostrar el día de la semana que le corresponde.
f) Introducir un tipo de producto (1- perecedero, 2-electrónica, 3-ropa) y la fecha de compra, y que informe si se puede devolver a día de hoy o no (los plazos de devolución, son respectivamente 5 horas, 6 meses, 15 días)
g) Introducir un año y decir cuántos domingos tiene.
h) Indica el día de la semana (en texto, en gallego) del 31 de diciembre de los últimos 5 años.

6.10. Desarrolla la clase *MovilPrepago* que te proporcionó el profesor para ejercicios anteriores y compárala con la proporcionada. ¿Has añadido modificadores de acceso: public /private?

6.11. Desarrolla la clase *CuentaCorriente* que te proporcionó el profesor para ejercicios anteriores y compárala con la proporcionada.

Este ejercicio es opcional, simplemente puedes ojear el código de la clase proporcionada y ver si lo harías igual.

6.12. Diseña una clase llamada *Alumno* para gestionar los alumnos de una escuela. Contiene los atributos: nombre completo, DNI, fecha de nacimiento y nombre de la escuela (común para todos los alumnos). Además del constructor, los métodos *set* y *get* para cada atributo, tendrá los siguientes métodos:

- Método que nos dice si es mayor de edad o no.
- Método al que se le pase como parámetro otro alumno y nos devuelva *true* si el alumno pasado es menor que él mismo. *False* en caso contrario.
- Método al que se le pase como parámetro otro alumno y nos devuelva *true* si el alumno pasado es exactamente igual en todos los campos (un duplicado). *False* en caso contrario.

Realiza a continuación un programa sencillo que use la clase definida.

6.13. Realiza un juego de la ruleta rusa para dos jugadores. La pistola tiene 6 huecos en la recámara pero solo una bala. Al empezar la partida se genera una posición al azar de la recámara para la bala de forma que puede salir en el primer disparo, en el segundo, etc. hasta el sexto. Los jugadores irán disparando sucesivamente hasta que uno de los dos se muera. Crea una clase *Pistola* con los atributos y métodos que la definen.

6.14. Diseña una clase llamada *Ruleta* y un programa que la use, con la siguiente funcionalidad.

- El jugador puede apostar lo que quiera a par o a impar .
- En cada juego obtiene un número al azar entre 0 y 36
- Si sale 0 pierde tanto par como impar, y en caso contrario si el jugador acierta (par o impar) gana tanto como lo apostado.

Define los métodos necesarios para que los usuarios puedan apostar a par o impar. Hacer un programa en el que un jugador lleve 10.000 euros y juegue 10.000 veces 1 euro cada vez siempre apostando a par ¿Con cuánto dinero acaba? El resultado final puede ir entre 0 y 20.000 euros. Ejecútalo varias veces. ¿Conclusiones sobre jugar a la ruleta? ;)

6.15. Un profesor ha desarrollado un examen tipo test de 20 preguntas y cada una de ellas tiene cuatro opciones: *a), b), c) ó d)*, siendo solo una de ellas la correcta. Cada pregunta correcta suma 0.5 puntos y cada una incorrecta resta 0.2 puntos, no pudiendo llevar un alumno ninguna nota inferior a cero. Desarrolla una clase llamada *Examen* que almacene las respuestas correctas de las 20 preguntas. La clase debe disponer de algún mecanismo mediante el que se le pase las respuestas de un alumno para las 20 preguntas (a,b,c,d o bien z, siendo respuesta en blanco) y nos informe de la puntuación en el examen. Hacer un programa que permita primero informar de las respuestas correctas a un examen y después introducir las respuestas de los alumnos, mostrándonos la nota obtenida por los mismos.

6.16. (Opcional) Realiza mediante objetos un juego de Gato y Ratón: sobre un tablero que por el momento es de 20 filas x 20 columnas (aunque podría cambiar el tamaño), el gato y ratón se sitúan inicialmente en una posición al azar. En cada turno el gato y el ratón se mueven alternativamente. El ratón es rápido pero ciego así que se mueve de 2 en 2 casillas pero en una dirección aleatoria (si el movimiento provocase que saliese del tablero, vuelve a generarlo hasta que sea un movimiento válido). El gato se mueve de una en una casilla pero siempre en dirección hacia el ratón. El programa termina cuando se encuentren. El programa mostrará la situación del tablero después de cada movimiento (el usuario pulsará <ENTER> para que se produzca cada movimiento). Pistas:

- Hacer una clase *TableroGatoRaton*, con un constructor en el que se crea las dimensiones *cantidad de filas* y *cantidad de columnas.* Esa clase tiene un método para pintar el tablero, por ejemplo, con un punto para casillas vacías, una "G" para la posición del gato y una "R" para la posición del Ratón.
- Hacer una clase *Ratón* y una clase *Gato* cuyas propiedades son su posición, con un constructor que sitúa al gato o ratón en una posición inicial, y un método *mover* que cambia de posición, según las reglas descritas.
- Hacer un *main()* que cree un tablero, un ratón y un gato y alterne los movimientos de los dos últimos.

6.17. Sobre el siguiente código, modifica la clase *Contacto* para que funcione el main() mostrado:

```
import java.time.*;
public class ejercicio {
public static void main(String[] args) {

Contacto contacto;
contacto = new Contacto ("Marta", 6661111222L, LocalDate.parse("2019-11-25"));
contacto = new Contacto ("Miguel", 1111111L, LocalDate.now());
contacto = new Contacto ("Ana", 3333333L,"2019-11-20");
contacto = new Contacto ("Daniel", 444444L);
}}

class Contacto {
    public String nombre;
    public long numero;
    public LocalDate fechaAltaAgenda;

    Contacto (String no, long nu,LocalDate fe){
      this.nombre = no;
      this.numero = nu;
      this.fechaAltaAgenda = fe;
    }
}
```

6.18. Define una enumeración llamada *DiaSemana* en un archivo *.java*, que contenga los días de la semana (LUNES, MARTES, MIÉRCOLES, JUEVES, VIERNES, SÁBADO, DOMINGO). A continuación, realiza un programa que solicite al usuario que ingrese el texto de un día de la semana (puede hacerlo en mayúsculas o minúsculas). El programa responderá si el día ingresado es un día laborable (lunes a viernes) o un fin de semana (sábado y domingo).

6.19. Define una enumeración llamada *NivelEstudios* con valores: PRIMARIA, SECUNDARIA, CICLOS, UNIVERSIDAD. Define también una clase llamada *Estudiante* que tenga como atributos: nombre, edad y nivel de estudios. Haz los atributos privados e incluye *getters* y *setters.* Haz un programa que solicite al usuario los datos de un alumno, y si todos son correctos, cree una instancia de Alumno y muestre el método *toString* del mismo, que permitirá visualizar todos sus atributos en una línea. Los datos son correctos si el nombre ocupa más de 3 caracteres, la edad está entre 0 y 120 y el nivel de estudios introducido se corresponde con un valor de la enumeración definida.

Proyecto:

Sobre el proyecto del tema anterior, añadir las siguientes funcionalidades:

- El número secreto se genera al azar. Puedes hacerlo generando en bucle un dígito entre 0 y 9 al azar y lo vaya concatenando en la cadena, siempre que no genere duplicados. Lo hará repetidamente hasta llegar al tamaño de número deseado. Al hacer esto, ya no es necesario "limpiar" la pantalla para que comience el jugador, ya que al no introducir a mano el número secreto, no hay nada que ocultar.

- Crear una clase llamada "Juego" que guarde toda la información sobre el juego en curso. Contendrá como atributos todas las variables del juego, a saber: el número secreto a adivinar, la cantidad de intentos que lleva el jugador, el intento actual con sus dígitos bien colocados y mal colocados, etc. También incluirá los dos parámetros del juego: el tamaño del número a adivinar y cantidad de intentos permitidos. ¿Alguno de ellos puede ser estático? ¿y final?

- Haz que los atributos no estáticos de la clase anterior sean privados e incluye setters y getters públicos para ellos.

- Puede ser interesante que la clase incluya una enumeración con el estado en el que está una partida: jugando, has ganado, has perdido…

- Crea uno o varios constructores sobre la clase Juego.

- Pasa todos los métodos de la lógica que estaban en el main a métodos de la clase Juego:
  - método que genera el número secreto (también puede ser en el constructor de la clase).
  - Método que valide que el intento introducido por el usuario cumple los requisitos del juego.
  - Método que calcula el número de aciertos de un intento; tanto los bien colocados como los mal colocados.
  - Método que incremente el contador de intentos.

- En los métodos anteriores, en determinadas circunstancias, se cambiará el estado de juego a la situación de haber ganado o haber perdido.

- Una vez completada la clase anterior, el *main* de la aplicación solo deberá crear una instancia de la clase "Juego" e invocará a sus métodos en bucle hasta que termine el juego. Las tareas propias del *main* serán tan solo las de entrada/salida por teclado/pantalla.

Aunque ahora mismo no le veas mucha utilidad a estructurar la aplicación de esta forma, es fundamental cara a tener un código optimizado. Ahora solo tienes un jugador jugando una sola partida cada vez, pero imagina que tuvieses 3 jugadores simultáneamente. Con este planteamiento solo necesitarías crear tres instancias de la clase "Juego" y cambiar ligeramente la estructura del main para establecer los turnos, pero nada más. Sin objetos, tendrías que triplicar muchas variables y sería todo muy confuso.

7. Arrays y ArrayList

Bloque A: Array

Cuestiones:
Ejecuta estas porciones de código y responde a las cuestiones:

a) Indica los errores del siguiente código y muestre lo que saca por pantalla:

```
int [] a = new int [] {10,21,37,40,51};
int p=0,i=0;
for (int x=0;x<a.length;x++)
    if (x%2==0) p+=a[x]; else i+=a[x];
System.out.printf("%d-%d\n",p,i);
```

Añade la línea: `System.out.println(Arrays.toString(a));` ¿Qué muestra?

b) Indica los errores de cada línea:

a) `int [] arr = new arr [] {10,20,30,40,50};`
b) `int [5] arr = new int [] {10,20,30,40,50};`
c) `int [] arr = new int [5] {10,20,30,40,50};`
d) `int [] arr = new int [] {10,20,30,40,50};`
e) `int [] arr = new int [5];`

c) Corrige los errores de este código e indica la salida del mismo:

```
int [] arr = new int [5] {10,20,30,40,50};
for (int i=0; i<= arr.length()-1;i=+1)
    arr[i]=arr[i+1];
System.out.println(Arrays.toString(arr));
```

d) Indica que muestra este código y corrige los errores que encuentres:

```
public static void main(String[] args)  {
   int [] arr = new int [] {10,20,30,40,50};
   System.out.println(Arrays.toString(fun (arr)));
}
static int [] fun (int [] x ){
    int [] y = new int[x.length*2];
    int cont=0;
    for (int z : x){y[cont++]=z; y[cont++]=z+10;}
    return y;
}
```

e) Dado el siguiente programa en Java:

```
public class ej {
 public static void main(String[] args)  {
   boolean result=true;
   if (args.length != 2) result=false;
   else if (!args[0].equals(args[1])) result= false;
   System.out.println (result); } }
```

¿Qué mostrarían las siguientes ejecuciones?:

```
java ej      abc   abc   abc
java ej      abc   abc
java ej      ABc   abc
```

f) Indica qué hace este programa.

```
boolean result=true;
   if (args.length != 2) result=false;
   else if (!args[0].equals(args[1])) result= false;
   else { int up=0, low=0;
          for (int i=0;i<args[0].length();i++){
            if (Character.isUpperCase(args[0].charAt(i))) up++;
            else if (Character.isLowerCase(args[0].charAt(i))) low++;
      }
      if (low <= 1 ||  up <= 1 ) result = false;
   }
   System.out.println (result);
```

Ejercicios:

7.1. Realiza un programa que tenga definido un *Array* como variable global llamado *temperaturaMeses,* de 12 enteros, con las temperaturas medias de un lugar en cada uno de los meses del año. Serán valores enteros negativos o positivos. El programa tendrá diferentes funciones que realicen las siguientes tareas:

a) Llenar el *Array* con valores al azar para todos los meses (entre 0 y 40).
b) Mostrar por pantalla el contenido del *Array*.
c) Mostrar por pantalla el contenido del *Array* en orden inverso.

7.2. Realiza un programa similar al anterior, con las siguientes tareas:

a. Llenar el *Array* con valores al azar para todos los meses (entre 0 y 40).
b. Mostrar la temperatura media del año.
c. Mostrar los meses en los que hizo más de 30 grados.
d. Decir si hay algún mes con más de 30 grados, recorriendo lo imprescindible del array, es decir, en cuanto sepa que sí hay alguno, que pare de recorrer el array. Puedes usar break o return.
e. Decir si hay algún mes con más de 30 grados, recorriendo lo imprescindible del array pero ahora sin usar ni *break* ni *return.*
f. Decir si hay alguna temperatura repetida en dos o más meses.

7.3. Realiza un programa similar al anterior, pero que en la parte 'a' garantice que no hay temperaturas repetidas.

7.4. Realiza un programa similar a los anteriores pero que realice las siguientes funciones:

a. Llenar el *Array* con valores al azar para todos los meses. Para enero, febrero, marzo, noviembre y diciembre los valores estarán comprendidos entre -10 y +20 y para el resto de meses entre +10 y +40.
b. Mostrar por pantalla el contenido del array con el nombre de los meses (usar clases Java para obtener el nombre del mes)
c. Mostrar la temperatura máxima del año, usando *printf* especificando en el formato que incluya el signo "+" en temperaturas positivas y que solo muestre 1 decimal.
d. Mostrar el nombre del mes más frío del año.
e. Para hacer pruebas estadísticas, desplazar los valores a la derecha, pasando el último al primero.

7.5. Realizar una clase *Primitiva* que tenga definido un *Array* privado de 6 elementos con el resultado de un sorteo de la primitiva (serán 6 enteros con valores comprendidos entre 1 y 49 y sin repetidos). La clase dispondrá de un constructor en el que se generan y almacenen esos números al azar, también tendrá un método al que se le pase una combinación jugada como parámetro (no necesariamente ordenada) y devuelva el número de aciertos. Realiza tres programas distintos que usen esa clase:

- Programa que el usuario introduzca los 6 números de su boleto y le diga cuantos acertó.
- Programa en el que se generen 1000 boletos al azar y nos informe de cuantos boletos hay con 3, 4, 5 y 6 aciertos *(a lo mejor es necesario crear un nuevo método en la clase Primitiva para generar boletos al azar).*
- Programa con un bucle que genere boletos hasta que acierte los 6 números ¿Cuántos boletos has tenido que crear?

7.6. Realizar una clase llamada *Parking* que gestione los coches aparcados en un parking mediante un *Array* que almacene las matrículas. El parking es un poco raro, mide solo 3 metros de ancho y caben 20 coches, pero uno detrás de otro por lo que el último en entrar debe ser el primero en salir *(esta estructura se llama pila LIFO – Last Input, First Output)*. La clase debe tener métodos siguientes:

a. Mostrar el estado del parking, esto es las matrículas de las plazas ocupadas.
b. Aparcar: se le pasará el número de matrícula. Devuelve número de plaza o cero si lleno. La plaza del fondo es la número 1 y la que está más cerca de la entrada es la 20.
c. Desaparcar: (¿hace falta pasarle algún parámetro?). Devuelve la matrícula del coche a desaparcar o bien una cadena vacía si el parking está vacío.
d. Mostrar la cantidad de plazas libres en ese momento.

Hacer un programa con un menú que permita al usuario usar los métodos creados mostrando la información del parking correspondiente a los métodos creados (los métodos no deben escribir nada por consola, eso lo hace el programa que usa la clase)

Además del array, la clase Parking necesita una variable adicional, que le indique la posición en la que aparcará el siguiente coche. Inicialmente vale cero, al aparcar un coche se incrementa y al desaparcar se decrementa. Si vale 20 no cabrán más coches.

Para implementar este tipo de estructuras LIFO existe una Colecciones que resuelven esto de forma más sencilla, se verá en el tercer trimestre.

7.7. Realizar una clase llamada *Parking2* que gestione los coches aparcados en un parking mediante un *Array*, que almacene las matrículas y la hora de entrada en el parking. El parking es un poco raro, mide solo 3 metros de ancho y caben 20 coches, pero uno detrás de otro por lo que el último en entrar debe ser el primero en salir *(esta estructura se llama pila LIFO – Last Input, First Output).* Cada minuto en el parking cuesta 2 céntimos (máx 10 euros). La clase debe tener métodos siguientes:

- Mostrar el estado del parking, esto es las matrículas/hora de entrada de las plazas ocupadas.
- Aparcar: se le pasará el número de matrícula. Devuelve número de plaza o cero si lleno. La plaza del fondo es la número 1 y la que está más cerca de la entrada es la 20.
- Desaparcar: (¿hace falta pasarle algún parámetro?). Al desaparcar se calcula el importe a pagar según los segundos de estancia. Devuelve la matrícula del coche y el importe a pagar. Si es parking está vacío devuelve cadena vacía y cero euros.
- Mostrar lo recaudado hasta ese momento.

Hacer un programa con un menú que permita al usuario usar los métodos creados. No hay que introducir a mano los valores de hora de entrada y salida en el parking, se harán por el reloj del sistema.

7.8. (Opcional) Realizar una clase *carritoCompra* que mantenga las compras realizadas por un cliente en un *Array*. Cada posición del *Array* contendrá el código del producto, la descripción, el precio unitario y la cantidad de unidades compradas y el precio total de ese producto (cantidad x precio unitario). Se permite comprar como máximo de 100 artículos diferentes. Además de la lista de productos, la clase tiene el importe total de la compra, que debe estar siempre actualizado. Necesitamos los métodos:

a. Mostrar por consola el estado actual del carrito.
b. Añadir producto.

Hacer un programa con un menú que permita al usuario operar con el carrito de la compra.
Nota: Pensar en crear un método .toString() para facilitar el mostrar por pantalla cada producto comprado.

7.9. Realizar un programa que tenga una función a la que se le pasa un Array con las edades de los alumnos de una clase y nos devuelva la edad media. Añadir una función a la que se le pasa un Array con las edades de los alumnos y nos devuelva un Array solo con los mayores de edad.

7.10. Realizar un programa que tenga definido un *Array* de 12 filas y 30 columnas llamado *temperaturaDia* con las temperaturas medias de un lugar en cada uno de los días del año (suponemos meses de 30 días). Serán valores enteros negativos o positivos. El programa debe tener funciones para realizar las siguientes tareas:

a. Llenar de valores el Array: para evitar tener que introducir los valores a mano, genera valores al azar para todos los días. Para enero, febrero, marzo, noviembre y diciembre los valores estarán comprendidos entre -10 y +20 y para el resto de meses entre +10 y +35.
b. Mostrar por pantalla el contenido del array en forma de tabla: todos los días de un mes en la misma fila, ocupando 3 posiciones (una para el signo y dos dígitos para la temperatura).
c. Mostrar la temperatura media del año.
d. Calcular la temperatura media de cada mes.
e. Crear un Array unidimensional con la temperatura mínima de cada mes.

7.11. (Opcional) Realizar un programa que tenga definido un *Array* de 12 filas y 30 columnas llamado *temperaturaDia* con las temperaturas medias de un lugar en cada uno de los días del año (suponemos meses de 30 días). Serán valores enteros negativos o positivos. El programa debe tener funciones para realizar las siguientes tareas:

a. Llenar de valores el Array: para evitar tener que introducir los valores a mano, genera valores al azar para todos los días. Para enero, febrero, marzo, noviembre y diciembre los valores estarán comprendidos entre -10 y +20 y para el resto de meses entre +10 y +35.
b. Mostrar el día en el que más calor hizo del año.
c. Decir si hay algún día con más de 30 grados, recorriendo lo imprescindible del array, es decir, en cuanto sepa que sí hay alguno, que pare de recorrer el array.

7.12. Realizar un programa llamado hipotenusa al que se le pasen como parámetros dos valores, verifique que son enteros positivos y que suponiendo que son los catetos de un ángulo recto, calcule el valor de la hipotenusa. (se refiere a introducirlos en la llamada al programa, `java prog param1 param2`, sería algo así: `java hipotenusa 4 2`.

Para asignar parámetros a programas en Netbeans, clicamos botón derecho sobre el proyecto y, en *Propiedades del Proyecto*, en la sección *Ejecutar*: comprobamos que la entrada *'clase main'* contiene el nombre del paquete+programa de este ejercicio en concreto, y en *'argumentos'* ponemos separados por espacios en blanco los argumentos. Para ejecutar pulsamos *F6*, o bien *Ejecutar proyecto*, pero no *May+F6 (Ejecutar archivo actual)* como sí podemos hacer en otros casos.

7.13. Realizar un programa que tenga un Array con una agenda de teléfonos simple con unos valores cualquiera. Por ejemplo:

Marta	666111222
Miguel	981981981
Ana	900900900
Daniel	+34881000001

Al programa se le pasará como parámetro un nombre. Si el nombre está en la agenda, mostrará su teléfono, y en caso contrario informará que no existe dicha persona. `java prog Ana`.

En el tercer trimestre veremos que este tipo de estructuras, como la agenda, que no van orientadas a una numeración específica, se almacenan mejor en estructuras llamadas "Map".

7.14. (Opcional) Realizar un programa que defina un *Array* con 10 elementos e introduzca valores a azar entre 1 y 20 pero garantizando que no se introducen repetidos y una vez introducidos, ordénalos ascendentemente programando un algoritmo de ordenación.

En el apartado siguiente veremos que mediante ArrayList disponemos de métodos que ordenan por nosotros.

Bloque B: ArrayList

Cuestiones:

a) Muestra el contenido del ArrayList después de ejecutar este código:

```
ArrayList <Long> a = new ArrayList <>();
for (int i=0;i<=9;i++)
    a.add((long)Math.pow(2,i+1));
```

b) Si sobre el ArrayList anterior ejecutamos el siguiente código ¿Cómo quedaría el ArrayList?

```
for (int i=0;i<a.size();i+=2)
    a.set(i,999);
```

c) Inventa dos ejemplos de lo que podría ser la salida de este código:

```
ArrayList <Integer> a = new ArrayList <>();
int lon=(int) (Math.random()*6)+5;
for (int i=0;i<lon;i++)
    a.add((int)(Math.random()*100)+1);
Collections.sort(a);
for (int i=0; i<a.size();i++)
    System.out.println(a.get(i));
```

d) Inventa dos ejemplos de lo que podría ser la salida de este código:

```
ArrayList <Integer> a = new ArrayList <>();
int n=0;
for (int i=0;i<6;i++) {
    do { n = (int)(Math.random()*49)+1;
    } while (a.contains(n));
    a.add(n);
}
Collections.sort(a);
for (int i=0; i<a.size();i++)
    System.out.println(a.get(i));
```

e) ¿Qué hace este código con los valores contenidos en el ArrayList anterior? ¿Tiene errores?

```
 for (int i=0;i<a.size();i++) {
    a.set(i,a.get(i)+1);
```

f) ¿Qué hace este código con los valores contenidos en el ArrayList? ¿Tendrá valores mayores de 30?

```
ArrayList <Integer> a = new ArrayList <>();
llenarArrayList (a);   //llena el arrayList con valores positivos y negativos
for (int i=0;i<a.size();i++) {
    if (a.get(i)>30) a.set(i,30);
    else a.set(i,Math.abs(a.get(i)));
}
```

g) Indica que hace el siguiente código ¿Estadísticamente hablando, cuántos elementos tendrá más o menos el ArrayList?

```
ArrayList <Integer> a = new ArrayList <>();
for (int i=0;i<10000;i++) {
    a.add((int)(Math.random()*1000)+1);
}
for (int i=0; i<a.size();i++)
    if (a.get(i)>= 250 & a.get(i) <= 750) a.remove(i);
```

h) Este código quiere ordenar un Array convirtiéndolo en ArrayList, aplicándole el método estático *sort* y volviendo a convertir el ArrayList al array inicial. Qué falta en la línea en blanco.

```
Integer [] arr = {10, 3, 7, 2, 9, 5}; //no fuciona con int
. . . . . .
Collections.sort(lista);
arr = lista.toArray(new Integer[lista.size()]);;
System.out.println (Arrays.toString(arr));
```

i) ¿Es correcto este código? ¿Hace algo?

```
static void fun (ArrayList <String> a, String x) {while (a.remove(x)){}}
```

j) ¿Es correcto este código? ¿Hace algo?

```
public static void main(String[] args) {
        ArrayList <Integer> x = new ArrayList <>();
        for (int i=0;i<5;i++) x.add((int)(Math.random()*100));
        Collections.sort(x);
        fun (x, 20); fun (x, 200); fun (x, -1);
}

static void fun (ArrayList <Integer> a, int n) {
        for (int i=0; i<a.size();i++) {
            if (a.get(i)>n) {a.add(i,n);return;}
        }
        a.add(n);
}
```

Ejercicios:

7.15. Realiza un programa que tenga un *ArrayList* llamado *alturaAlumnos* que mantenga una lista con las alturas de los alumnos de un centro. Serán valores positivos entre 0,50 y 2,50 redondeados a dos decimales. El programa tendrá las siguientes funciones (accesibles mediante un menú):

a. Añadir altura.
b. Mostrar lista actual con el número de posición
c. Eliminar por posición. Se le pasa como parámetro una posición y elimina la altura en dicha posición.
d. Eliminar por valor. Se le pasa como parámetro una altura y elimina todas las posiciones en las que se encuentre dicha altura. Devuelve la cantidad de eliminaciones.
e. Ordenar la lista.
f. Vaciar la lista.

7.16. Realizar un programa al que se le vayan introduciendo por teclado números enteros. El programa dispone de dos *ArrayList*, uno llamado **`positivos`** y otro **`negativos`**. Se trata de meter los números introducidos en uno u otro según su signo hasta que finalice el programa (esto ocurrirá cuando se introduzca cero). Al finalizar, mostrará la media aritmética de cada *ArrayList*.

7.17. Realiza una clase llamada *Primitiva2* similar a la clase *Primitiva,* pero empleando ahora una *ArrayList*, y aprovechando los métodos de los que dispone para simplificar la generación de números sin repetidos. Además, los números premiados se mantendrán ordenados y, por último, la búsqueda de los números jugados en el *ArrayList* de los premiados se requiere que se haga de forma dicotómica. Rehacer los 3 programas del ejercicio anterior de la *Primitiva* pero usando esta nueva clase.

- Programa que el usuario introduzca los 6 números de su boleto y le diga cuantos acertó.
- Programa en el que se generen 1000 boletos al azar y nos informe de cuantos boletos hay con 3, 4, 5 y 6 aciertos *(a lo mejor es necesario crear un nuevo método en la clase Primitiva para generar boletos al azar).*
- Programa con un bucle que genere boletos hasta que acierte los 6 números ¿Cuántos boletos has tenido que crear?

Opcionalmente, puedes probar a hacer una nueva versión de la clase, sería Primitiva3, que sea igual que Primitiva2, pero con Array en vez de ArrayList. Como no dispones de métodos para ordenar ni para búsqueda dicotómica deberás construirlos tú mismo. Así compruebas las ventajas de usar ArrayList y sus métodos ya creados en vez de Arrays.

7.18. Realizar un programa que tenga una función a la que se le pasa un entero y devuelva un ArrayList con todos sus divisores.

7.19. Realizar una clase *carritoCompra2* que mantenga las compras realizadas por un cliente en un *ArrayList*. Cada posición del *ArrayList* contendrá el código del producto, la descripción, el precio unitario y la cantidad de unidades compradas y el precio total de ese producto (cantidad x precio unitario). Se permite comprar un número indeterminado de artículos. Además de la lista de productos, la clase tiene el importe total de la compra, que debe estar siempre actualizado. Necesitamos los métodos:

a) Mostrar por consola el estado actual del carrito.
b) Vaciar carrito.
c) Añadir producto.
d) Eliminar producto (se le pasa el código de producto) y lo elimina físicamente del *ArrayList.*

Hacer un programa con un menú que permita al usuario operar con el carrito de la compra.
Pensar en crear un método .toString() para facilitar el mostrar por pantalla cada producto comprado.

7.20. Diseñar una clase *Factura* que consta de:

- Número identificador: lo proporciona el usuario en el alta de la factura.
- Fecha de la factura: la toma del sistema en el momento del alta.
- Número de cliente: : lo proporciona el usuario en el alta de la factura.
- Porcentaje de IVA: 21% para todas las facturas.
- Un número indeterminado de *lineaFactura* que contienen:
  - ✓ Descripción del producto
  - ✓ Precio unitario
  - ✓ Cantidad de unidades
  - ✓ Porcentaje de descuento: 5% para líneas con más de 10 unidades.
  - ✓ Importe total de la línea.
- Importe total: inicialmente cero, y se va actualizando siempre que se añadan/eliminen líneas.

Implementar la clase con su constructor y métodos para añadir línea de factura, eliminar línea de factura y mostrar la factura por consola. Te hará falta una clase *lineaFactura* con su constructor.
Para añadir línea de factura, habrá que solicitar al usuario los campos necesarios para añadirlo (descripción, precio unitario y cantidad de unidades). Para eliminar una línea, solicitaremos el número de línea.
Hacer también un programa con un menú para gestionar una factura (alta, añadir/eliminar líneas, mostrar factura) Nota: pensar en método toString() para *líneaFactura.*

7.21. (Opcional) Realizar una versión del juego de cartas de las 7 y media para 3 jugadores. Los jugadores van solicitando cartas 1 a 1 para llegar a las 7 y media sin pasarse, pudiendo plantarse cuando lo desee. Las figuras valen medio punto y el resto su valor numérico. La baraja tiene 40 cartas. El juego se acaba cuando todos los jugadores se planten, se hayan pasado, o se acaben las cartas.

Hacer una clase Baraja paso a paso, primero una versión que solo obtenga las cartas de forma aleatoria eliminando la carta del ArrayList para que no vuelva a salir, luego que juegue un solo jugador y finalmente, la versión completa.

Pensar en crear las clases: Carta, Jugador y una clase Baraja que incluya 40 cartas y hasta 3 jugadores.

7.22. Realizar un programa que cree un ArrayList con 10.000 números aleatorios entre 1 y 6 (como si fuese lanzar un dado). Utilizando los métodos estáticos de la clase *Collections* guarda en otro ArrayList la distribución de resultados obtenidos (cuantas veces ha salido el uno, cuantas veces ha salido el dos, etc...) y muestra su contenido. Finalmente, también mediante métodos de *Collections*, mostrar la diferencia de veces entre el número que más ha salido y el que menos ha salido.

7.23. Realizar el programa que simule el comportamiento de una "cola" FIFO con los nombres de pacientes que esperan en la consulta del médico. Tendrá un menú con las siguientes opciones a) Llega un paciente (esto es, Introducir elemento al final de la cola) b) Llamar al paciente para ser atendido (esto es, sacar el primer elemento de la cola) mostrándolo por pantalla y c) Mostrar el estado de la cola y d) salir de programa.

7.24. Realizar un programa con una función a la que se le pasan dos ArrayList de enteros como parámetros y nos devuelva *true* si los dos ArrayList tienen los mismos elementos, aunque sean en otro orden, y devuelva *false* en caso contrario (suponemos que no tienen valores repetidos).

7.25. Realizar una clase llamada *Parking3* que gestione los coches aparcados en un parking mediante un *ArrayList*, que almacene las matrículas y la hora de entrada en el parking. El parking es

un poco raro, mide solo 3 metros de ancho y caben 20 coches, pero uno detrás de otro por lo que el último en entrar debe ser el primero en salir *(esta estructura se llama pila LIFO – Last Input, First Output).* Cada minuto en el parking cuesta 2 céntimos (máx 10 euros). La clase debe tener métodos siguientes:

- Mostrar el estado del parking, esto es las matrículas/hora de entrada de las plazas ocupadas.
- Aparcar: se le pasará el número de matrícula. Devuelve número de plaza o cero si lleno. La plaza del fondo es la número 1 y la que está más cerca de la entrada es la 20.
- Desaparcar: (¿hace falta pasarle algún parámetro?). Al desaparcar se calcula el importe a pagar según los segundos de estancia. Devuelve el importe a pagar. Si es parking está vacío devuelve -1.
- Mostrar lo recaudado hasta ese momento.

Hacer un programa con un menú que permita al usuario usar los métodos creados. No hay que introducir a mano los valores de hora de entrada y salida en el parking, se harán por el reloj del sistema.

Proyecto:

Sobre el proyecto del tema anterior, añadir las siguientes funcionalidades:

- Añadir a la clase "Juego" un atributo de tipo Arraylist que mantenga los datos de cada intento realizado por el usuario, en vez de tener solo la información del último intento, que era lo que teníamos hasta el momento. De cada intento mantendremos el número, la cantidad de dígitos bien colocados y los mal colocados (te hará falta crear una clase, a la que puedes llamar "Jugada" o "Intento").

- En la consola, cuando se le pide a un jugador un intento, mostrar la lista de los intentos previos con su resultado tal y como muestra la imagen siguiente:

```
Bienvenido al Juego del Mastermind!
Para debug: numero generado=9214
Intenta adivinar el número (4 dígitos, sin repetidos):
Dispones de 10 intentos.
Intento número 1 : 9213
==========================================
1       9213    Bien colocados:3        Mal colocados:0
Intento número 2 : 9213
Error: Intento número 2 : 9222
Error: Intento número 2 : 92w1
Error: Intento número 2 : 92834
Error: Intento número 2 : 9278
==========================================
1       9213    Bien colocados:3        Mal colocados:0
2       9278    Bien colocados:2        Mal colocados:0
Intento número 3 : 9573
==========================================
1       9213    Bien colocados:3        Mal colocados:0
2       9278    Bien colocados:2        Mal colocados:0
3       9573    Bien colocados:1        Mal colocados:0
Intento número 4 :
```

- A la hora de validar un nuevo intento, ahora sí que podemos validar si ese intento ya se realizó previamente o no.

- Evalúa los getters y setters creados en el proyecto previo y elimina aquellos que creas que no son necesarios, ya que son de gestión interna de la clase "Juego" y no deben ser accesibles desde el exterior.

- Este podría ser un ejemplo del *main.* Fíjate que es muy sencillo, ya que toda la lógica está en la clase juego. La clase que lo contiene solo debería tener un método adicional para mostrar por consola los intentos realizados hasta el momento (también podría estar en una clase aparte)

```
public static void main(String[] args) {

      String cifraIntento;
      Juego juego = new Juego();
      System.out.println("Bienvenido al Juego del Mastermind!");
      System.out.println("Intenta adivinar el número ("+Juego.TAM_NUMERO+" dígitos, sin repetidos):");
      System.out.println("Dispones de " + Juego.MAX_INTENTOS + " intentos.");
      while (juego.getEstadoJuego() == EstadoJuego.JUGANDO) {
          boolean intentoValido=true;
          do {
               if (!intentoValido)System.out.print("Error: ");
               System.out.print("Intento número " + juego.getOrdinalIntento() + " : ");
               cifraIntento = scanner.nextLine();
               intentoValido=juego.validarIntento(cifraIntento);
          } while (!intentoValido);
          juego.procesarIntento(cifraIntento);
          mostrarSituacionActual(juego);
          if (juego.getEstadoJuego() != EstadoJuego.GANADOR)    juego.siguienteIntento();
      }
  }
```

8. Clases y Herencia

Cuestiones:

Examina la clase *Bicho* y su clase hija *BichoDormilon:*

```
class Bicho {
    public int hambre;
    private int peso;
    Bicho () { hambre =50; peso=50; }
    Bicho (int h, int p) { hambre =h; peso=p;}
    public void come () { hambre -=5; peso++; }
}

class BichoDormilon extends Bicho {
    public int sueño;
    BichoDormilon () { sueño = hambre * 2; }
    BichoDormilon (int i) { super(i,0); sueño = i+20; }

    @Override
    public void come () {hambre -=10;   sueño +=5;   }
    public void aDormir () {super.come();   sueño =0;   }
}
```

Y trata de responder las siguientes cuestiones, justificando tu respuesta. Luego comprueba los resultados, ejecutando un programa con las sentencias propuestas:

a) ¿Qué mostraría el siguiente código?

```
Bicho bi = new Bicho();
BichoDormilon  bd = new BichoDormilon();

System.out.println("bi -> hambre: " + bi.hambre);
System.out.println("bd -> hambre: " + bd.hambre);
System.out.println("bd -> sueño: " + bd.sueño);
System.out.println("bd -> peso: " + bd.peso);
```

b) Si a continuación se ejecutan estas instrucciones ¿qué mostraría?

```
bi.come();
bd.come();
System.out.println("bi -> hambre: " + bi.hambre);
System.out.println("bd -> hambre: " + bd.hambre);
bd.aDormir();
System.out.println("bd -> hambre: " + bd.hambre);
System.out.println("bd -> sueño: " + bd.sueño);
```

c) Sobre las mismas clases, vamos a crear un nuevo *BichoDormilon* ¿qué mostrarían estas sentencias?

```
BichoDormilon bd2 = new BichoDormilon (5);
bd2.come(10);
System.out.println("bd2 -> hambre: " + bd2.hambre);
```

d) Vamos a hacer algo parecido otra vez, vamos a crear otro *BichoDormilon* ¿qué mostrarían estas sentencias?

```
BichoDormilon bd2 = new BichoDormilon (5);
bd2.come();
System.out.println("bd2 -> hambre: " + bd2.hambre);
System.out.println("bd2 -> sueño: " + bd2.sueño);
```

Ejercicios:

8.1. Realiza las siguientes operaciones:

- Crea una clase llamada *Figura2D* con atributos numéricos con decimales y públicos: *ancho* y *alto* y un método *verDim* que muestre por consola el alto y el ancho en una sola línea, con dos decimales.
- Define una clase llamada *Triángulo* que es hija de *Figura2D* y añádele una cadena llamada *estilo* (vale: isósceles, rectángulo, equilátero, etc.), un método llamado *área* que devuelva la superficie del triángulo y un último método llamado *verEstilo* que muestre por consola el valor del atributo estilo.
- Finalmente hacer un programa que cree un triángulo, asigne valores a sus atributos y use los métodos para ver sus dimensiones, estilo y área.

8.2. Copia las clases anteriores como una nueva versión de las mismas (añade sufijo _v2) y realiza los siguientes cambios:

- Ahora los atributos *ancho* y *alto* serán privados.
- Añade los métodos *getter* y *setter* para ambos atributos.
- Modifica la clase *Triangulo* obligados por los cambios en su clase padre.
- Haz una copia del programa anterior, sobre las nuevas clases creadas, y comprueba que el programa creado en el ejercicio anterior sigue funcionando.

Si creas estas clases mediante copiar y pegar del ejercicio anterior, cuidado con la cláusula extends ya que no la actualiza, puedes tener el error: `public class Triangulo_v2 extends Figura2D` en vez de: `public class Triangulo_v2 extends Figura2D_v2`.

8.3. Copia las clases anteriores como una nueva versión de las mismas (añade sufijo _v3) y realiza los siguientes cambios:

- Añade un constructor a la clase *Triangulo* al que se le pasan tres parámetros: estilo, alto y ancho.
- Se invocará al constructor por defecto de la clase base.
- Modificar el programa de los ejercicios anteriores para que los triángulos sean creados mediante este nuevo constructor.

8.4. Copia las clases anteriores como una nueva versión de las mismas (añade sufijo _v4) y realiza los siguientes cambios:

- Añade un constructor a la clase *Figura2D* al que se le pasan dos parámetros: alto y ancho. ¿Funciona ahora el constructor de *Triangulo* (creado en el ejercicio anterior)? ¿Por qué?
- Reescribe el constructor de *Triangulo* creado en el ejercicio anterior para que haga uso del nuevo constructor de la clase base.
- Comprueba que el programa creado en el ejercicio anterior sigue funcionando.
- Haz una copia del programa anterior, sobre las nuevas clases creadas, y comprueba que sigue funcionando.

8.5. Copia las clases anteriores como una nueva versión de las mismas (añade sufijo _v5) y realiza los siguientes cambios:

- Añade un constructor más a la clase *Figura2D,* a este se le pasa solo un parámetro que se le asigna tanto al alto como al ancho (figura igual alto que ancho).
- Hacer el constructor sin parámetros (ya no se crea por defecto) en este caso tanto alto como ancho igual a cero.
- Añade a la clase *Triangulo* también un constructor sin parámetros, que invoque al constructor sin parámetros de la clase base (aunque se hace por defecto) y que ponga el estilo a *null*.
- Añade otro constructor a la clase *Triangulo*, con un solo parámetro, para aquellos que tienen igual alto que ancho (en este caso el estilo será *'igualBaseAltura').*
- Haz un programa similar a los de ejercicios anteriores pero que use las nuevas características incorporadas en este ejercicio.

8.6. Copia las clases anteriores como una nueva versión de las mismas (añade sufijo _v6) y realiza los siguientes cambios:

- Crea una nueva clase *TrianColor_v6*, hija de la clase *Triángulo_v6* que incluye un nuevo atributo privado de tipo *String* llamado *color.*
- Esta nueva clase tiene un constructor de 4 parámetros (alto, ancho, estilo, color), además del *getter* y *setter* de *color*.
- Haz un nuevo programa que cree un triángulo de color y llame a métodos definidos en sus clases base.

8.7. Copia las clases anteriores como una nueva versión de las mismas (añade sufijo _v7) y realiza los siguientes cambios:

- *Figura2D_v7* tendrá un nuevo constructor, que recibe como parámetro otro objeto de tipo *Figura2D_v7* y que copiará todos sus datos.
- Hacer otro constructor análogo para la clase *Triangulo_v7.*
- Hay un programa que cree un *Triangulo_v7* con el constructor de 3 parámetros creado previamente, y otro triángulo utilizando el nuevo constructor. Mostrar el área de ambos.

8.8. Copia la última versión de las clases *Figura2D*, *Triangulo*, *Triancolor* como una nueva versión de las mismas (añade sufijo _v8) y realiza los siguientes cambios:

- Añadir a la clase *Figura2D_v8* un atributo privado llamado *nombre* de tipo *String.*
- Añadir el *getter/setter* de ese campo y modificar los constructores de dicha clase para incorporar como parámetro el nombre de la figura (en el constructor por defecto se le asignará valor *null*).
- Modificar las clase hija *(Triangulo_v8)* y nieta *(Triancolor_v8)* para incluir el *nombre* en constructores.
- Crear una nueva clase hija de *Figura2D_v8* llamada *Rectangulo_v8* con constructor con tres parámetros (alto, ancho, nombre), constructor con un dos parámetros (alto igual a ancho y nombre) y un método que devuelve *boolean* llamado *esCuadrado()*.
- Hacer un programa que cree un ArrayList con 4 rectángulos, algunos cuadrados y otros no, y cuente cuantos hay cuadrados. En el mismo main() crear un triancolor a los que le asignes también el nuevo atributo: *nombre* y muestres su área.

8.9. A partir de la clase *CuentaCorriente* que te proporcionará el profesor realiza las siguientes operaciones:

- Estudia los métodos: 'ingresar' y 'retirar' y añádeles un comentario a cada método explicando su funcionamiento (bonificaciones y comisiones).
- Crea una clase hija llamada *CuentaPlazo* igual que la anterior, pero sin comisiones. Además, tiene un nuevo atributo, que es la fecha de creación.
- Esta nueva clase ya no permitirá que haya subclases de ella.
- Crea constructor para la nueva clase que incorpore la inicialización de la fecha de creación al día en curso.
- Sobrescribe el método *retirar* () para que no que no calcule comisiones. Revisa los modificadores de acceso de atributos y quizás tengas que crear algún getter/setter.
- Haz un programa que cree inicialmente una cuenta a plazo y luego, mediante un menú, permita ingresar, retirar y consultar el saldo.

8.10. A partir de la clase *MovilPrepago* que te proporcionará el profesor realiza las siguientes operaciones:

- Crea una subclase llamada *MovilPlus* igual que la anterior pero que tiene una nueva funcionalidad llamada *videollamada* a la que se le pasa los segundos de la videollamada y reduce el saldo de forma similar a 'navegar'. En este caso, cada segundo de llamada son 2 MB de datos (recuerda que si los atributos de *MovilPrepago* son privados, esto puede ser un problema a resolver).
- Crea constructor para la nueva clase, que llame al constructor de la clase padre.
- Sobrescribe el método *toString ()* en la superclase para que muestre el número y el saldo del móvil.
- Añade una nueva subclase de *MovilPrepago* llamada *MovilTarifaPlana*, en la que se redefine el método navegar para no reducir el saldo y además en el constructor fija el coste de navegación a cero.
- Haz un programa que cree una instancia de *MovilPrepago,* otra de *MovilPlus* y otra de *MovilTarifaPlana,* y realicen diversas operaciones sobre los mismos: llamar, navegar, videollamar, etc. mostrando como se reduce su saldo con el nuevo método *toString ().*

8.11. Crear una clase llamada *Trabajador* con los atributos privados: id, nombre, fecha de nacimiento y salario base.

- Dispondrá también de un constructor que inicialice todos sus campos, getters, setters, método *toString ()* y *equals ()*, sabiendo que dos trabajadores son iguales si tienen el mismo 'id'.
- Crear una subclase de *Trabajador* llamada *Asalariado* que añade un nuevo atributo llamado *salarioFinal* y otro llamado *horasExtra.*
- El constructor de esta nueva clase *Asalariado* incorpora la inicialización a cero de las horas extra y el salario final igual al salario base.
- La clase *Asalariado* también incorpora setter y getter para las horas extra y un método llamado *calcularSalarioFinal()* al que se le pasa a cuanto se paga la hora extra en ese momento y calcula el salario final del empleado siendo su salario base más el importe de las horas extras trabajadas.
- Crear una subclase de *Trabajador* llamada *ConsultorExterno* que añade un nuevo atributo llamado *horasTrabajadas* y *salarioFinal.*
- El constructor de esta nueva clase *ConsultorExterno* incorpora la inicialización a cero de las horas trabajadas, salario base y salario final.

- La clase *ConsultorExterno* también incorpora setter y getter para las horas trabajadas y un método llamado *calcularSalarioFinal()* al que se le pasa a cuanto se paga la hora a los consultores en ese momento y calcula el salario final del consultor solo en función de las horas trabajadas (el salario base de estos trabajadores es cero).
- Haz un programa que cree un ArrayList de *Asalariados* y otro de *ConsultoresExternos* e introduce "a mano" dos o tres empleados en ambos ArrayList.
    - Después modifica el contenido de ambos ArrayList añadiendo también "a mano" las horas extra/horas trabajadas respectivamente para todos los empleados.
    - Fijar el importe de hora extra a 20 euros y la hora de consultor a 100 euros y modificar de nuevo los ArrayList calculando el salario final de cada trabajador.
    - Finalmente, recorriendo con un *for-each* ambos ArrayList, obtener el total que gastará la empresa en salarios.
    - ¿Sería útil tener un ArrayList que pudiese contener instancias de ambas clases?

Proyecto:

Sobre el proyecto del tema anterior no vamos a añadir herencia, que es lo que se corresponde con este capítulo, ya que no tiene sentido en este contexto, pero vamos a seguir optimizando la aplicación. En este caso vamos a añadir un menú principal para que el jugador pueda elegir:

1) Nueva partida

2) Configurar cantidad dígitos del número (entre 3 y 8)

3) configurar cantidad de intentos (entre 5 y 12)

4) Salir del juego

9. POLIMORFISMO

Cuestiones:

Examina de nuevo la clase *Bicho* y su clase hija *BichoDormilon* del apartado anterior y trata de responder las siguientes cuestiones, justificando tu respuesta. Luego comprueba los resultados, ejecutando un programa con las sentencias propuestas:

a) Vamos a definir una variable *Bicho* pero llamar sobre ella al constructor de *BichoDormilon.* ¿Qué mostraría el siguiente código?

```
Bicho bi2 = new BichoDormilon(10);
System.out.println("bi2 -> hambre: " + bi2.hambre);
System.out.println("bi2 -> sueño: " + bi2.sueño);
```

b) ¿Solucionaríamos el problema del *println* anterior así?

```
System.out.println("bi2 -> sueño: " + ((BichoDormilon) bi2).sueño);
```

c) ¿Y así?

```
BichoDormilon bd4 = (BichoDormilon) bi2;
System.out.println("bd2 -> sueño : " + bd4.sueño);
```

Ejercicios:

9.1. Realiza un programa con una variable de tipo *Figura2D_v8*, y sobre ella llama a uno de los constructores de *Triangulo_v8.* Muestra sus dimensiones y el cálculo del área.

9.2. Realiza un programa que permita al usuario seleccionar un tipo de figura (*Triangulo* o *Rectangulo),* luego llame al constructor adecuado solicitando al usuario los parámetros necesarios, dependiendo del tipo de figura.

- Después de crear la figura, mostrará las dimensiones de la misma (sea cual sea su tipo).
- Finalmente, y utilizando el operador *instanceof,* mostrará unos datos adicionales dependiendo del tipo de figura que sea:
  - Si es de tipo *Triangulo*, mostrará el área.
  - Si es de tipo *Rectangulo,* mostrará si es de forma cuadrada o no.
- Usar una única variable para almacenar la figura, sea del tipo que sea.

9.3. Modificar el ejercicio anterior para añadir a las figuras posibles el *Triancolor.* Así pues, el usuario seleccionará entre: *Triangulo, Triancolor, Rectangulo.*

- Después de crear la figura, mostrará las dimensiones de la misma (sea cual sea su tipo).
- Finalmente, y utilizando el operador *instanceof,* mostrará unos datos adicionales dependiendo del tipo de figura que sea:
  - Si es de tipo *Triangulo*, mostrará el área (sea *Triancolor* o no).
  - Si es de tipo *Rectangulo,* mostrará si es de forma cuadrada o no.
  - Si es de tipo *Triancolor*, mostrará el color.

Ojo con el comportamiento de **instanceof**: si x es una instancia de TrianColor, x instanceof Triangulo será true, es decir un Triancolor también es un Triangulo.

9.4. Realiza un programa que contenga un ArrayList de figuras2D de cualquiera de sus tipos e introduce valores "a mano", por ejemplo, un par de instancias de cada tipo. A continuación, el programa:

- Sumar el área de todas ellas. ¿Tienen implementado el método *área ()* todas ellas?
- Contar cuantos triángulos (sean de color o no) y cuantos rectángulos.

9.5. Empleando las clases previas: *MovilPrepago, MovilTarifaPlana, MovilPlus,* realizar un programa que inicialmente permita seleccionar al usuario qué tipo de tarifa tiene entre las tres posibles, luego configurará el móvil solicitando al usuario los parámetros necesarios y finalmente presentará al usuario un menú para realizar las operaciones permitidas (navegar, llamar, recargar, videollamar, ver saldo y salir) según el tipo de tarifa.

9.6. Modificar el programa anterior para que el usuario tenga un Array de 5 teléfonos, y que después de elegir la operación a realizar pueda elegir con cuál de los 5 teléfonos desea hacerla.

9.7. Volviendo a la parte final del último ejercicio del capítulo anterior, haz un programa que cree un ArrayList que pueda contener tanto *Asalariados* como *ConsultoresExternos* e introduzca "a mano" trabajadores de ambos tipos en el ArrayList.

- Después modificar el contenido del ArrayList añadiendo 1 hora extra/horas trabajada a *Asalariados y ConsultoresExternos* respectivamente.
- Fijar el importe de hora extra a 20 euros y la hora de consultor a 100 euros y modificar de nuevo el ArrayList calculando el salario final de cada trabajador.
- Finalmente, recorriendo con un *for-each* el ArrayList, obtener el total que gastará la empresa en salarios.

9.8. Crea una clase llamada *Consola* con un método estático sobrecargado llamado *leerEntero()* que solicite al usuario que teclee un valor entero, cumpliendo las siguientes características:

- Si no se le pasa ningún parámetro, no tiene requisitos, es simplemente un nextInt().
- Si se le pasa un parámetro de tipo texto, escribe ese texto antes de solicitar el valor. Ejemplo: *leerEntero ("Introduzca su edad");*
- Si se le pasa un parámetro tipo texto y dos enteros, garantizará que el valor tecleado esté comprendido entre ambos. Ejemplo:
  leerEntero ("Introduzca su edad", 0, 120);
- Si se le pasan dos enteros, garantizará que el valor tecleado esté comprendido entre ambos, pero no muestra texto de instrucciones previo. Ejemplo:
  System.out.println ("Introduzca su edad"); leerEntero (0, 120);

Finalmente, haz un programa que pruebe todas las variantes del método.

9.9. Haz un programa con un menú que permita gestionar la cola de espera de un médico. Hay tres tipos de pacientes: los que vienen a consulta (se le pide al usuario nombre, fecha de nacimiento, motivo de la consulta), los que viene por recetas (se le pide: nombre, fecha de nacimiento, lista de medicamentos) y el que viene a revisión (se le pide nombre, fecha de nacimiento y fecha de la visita anterior).

- Las tarifas del médico son: Consulta: 100 eur. Recetas: 5 eur por cada unidad. Revisión: 30 eur para mayores de 65 años, 50 eur para resto.
- Crear una clase para cada tipo de paciente en el propio archivo del programa con los constructores necesarios y el método de *facturar*() en cada una de las clases. Implementa herencia si lo crees necesario.
- El programa tendrá un menú para:
    a) Registrar la llegada del paciente: se le preguntará por qué viene al médico y se le piden sus datos.
    b) Llamar a consulta (por orden de llegada). Se le cobra la tarifa correspondiente.
    c) Consultar total facturado hasta ese momento.

Puedes crear una clase Clínica, que tendrá un ArrayList de Pacientes, o bien definir ese ArrayList en el programa como variable global y no tener la clase Clínica.

9.10. Haz un programa con un menú que permita gestionar un parking.

- El parking tiene 100 plazas y pueden aparcar cualquier tipo de 3 tipos de vehículos distintos: Vehículos en general, Furgonetas y Autobuses.
- Todos los vehículos pagan 4 céntimos por minuto, pero las furgonetas pagan además un suplemento de 20 céntimos por cada metro de su longitud y los autobuses pagan un suplemento de 25 céntimos por asiento.
- El menú del programa deberá permitir:
    a) Entrada de un vehículo. Se le pide al usuario la matrícula, tipo de vehículo y datos adicionales para el cálculo de la estancia (longitud, número de asientos…).
    b) Salida del vehículo. Se le pide al usuario la matrícula, se calcula el importe a pagar y libera la plaza.
    c) Mostrar la lista de vehículos en el parking con la matrícula, tipo de vehículo y fecha/hora de llegada (piensa en el método *toString*). Al final número total de plazas ocupadas.
    d) Salir
- Puedes hacer coste 4 céntimos por segundo (en vez de por minuto) para probarlo.
- No hay el concepto de número de plaza, los coches van aparcando donde quieren.
- Mantén la mayor cantidad de información (datos y cálculos) en las clases, no en el programa. Puedes crear las clases en el mismo archivo que el programa (por comodidad) con el modificador de acceso por defecto.
- El parking será un ArrayList. Para localizar un vehículo (en la opción de menú de *Salida del vehículo)* emplea *ArrayList.indexOf* y ello te puede implicar definir *equals* en alguna clase.

Al igual que en el ejercicio anterior, puedes crear una clase Parking, que tendrá un ArrayList de Coches, o bien definir ese ArrayList en el programa como variable global y no tener la clase Parking.

9.11. (Opcional) Partiendo de clases del ejercicio de cuentas bancarias del capítulo anterior *(CuentaCorriente, CuentaPlazo)*, haz un programa con un menú que permita gestionar unas cuentas bancarias que se almacenan en un ArrayList (añadir cuenta, eliminar cuenta, ingresos y, retiradas) siempre accediendo por un identificador que tendrá cada cuenta.

9.12. Crea una clase *Bicicleta* de la que deseamos mantener los siguientes datos: marca, modelo, precio y descuento. Se pide crear el constructor, getters y setters, método toString (), equals (), un método que devuelva el precio con el descuento aplicado y finalmente un método que fije el descuento a aplicar. Este último método estará sobrecargado de la siguiente forma:

- *fjarDescuento* () → (se le hace 10% y dura ese descuento 1 mes)
- *fjarDescuento* (double d) → (se le hace d %, 1 mes)
- *fjarDescuento* (double d , int n) → (se le hace d %, n meses)

Haz un programa sencillo que defina una o dos instancias de bicicletas y use los métodos creados

Notas:

- Dos bicicletas son iguales si tienen la misma marca y modelo.
- Si se fija un descuento, se elimina el descuento que pudiera haber anteriormente.
- Piensa si es necesario incorporar algún atributo adicional, para que, cuando ejecutemos el método de ver el precio final (con descuento aplicado), sepa si tiene que aplicar algún descuento o no.

Proyecto:

Sobre el proyecto del tema anterior no vamos a añadir polimorfismo de clases ya que no tenemos herencia, pero vamos a seguir optimizando la aplicación.

Piensa que en un futuro quisieses pasar esta aplicación a un entorno web. La lógica del juego valdría perfectamente tal cual está, pero toda la entrada salida no (habría que hacer una interfaz html o similar). Asegúrate de tener el código bien separado. El programa (la clase que contiene el Main) debería tener solo con los métodos de entrada/salida y el proceso repetitivo del juego hasta que termine el juego y nada más.

La clase Juego debería mantener toda la lógica del juego: comprobar los aciertos, ver si ha terminado el juego, etc.

También podría ser interesante tener una clase con los parámetros de configuración, que no "ensucie" la clase Juego, pero que tampoco esté en la clase *main*. Los valores de esa clase podrían ser empleados por la clase Juego y por la clase Main. Ejemplo:

```
public class Config {
    public static final int TAM_NUMERO_LIMINF = 3;
    public static final int TAM_NUMERO_LIMSUP = 8;
    private int tamNumero;
    public static final int MAX_INTENTOS_LIMINF = 5;
    public static final int MAX_INTENTOS_LIMSUP = 12;
    private int maxIntentos;

    public Config() {
        this.tamNumero = 4;
        this.maxIntentos = 10;
    }

    public boolean setTamNumero(int tamNumero) {
        if (tamNumero < Config.TAM_NUMERO_LIMINF ||
            tamNumero > Config.TAM_NUMERO_LIMSUP)
            return false;
        this.tamNumero = tamNumero;
        return true;
    }
```

```
    public boolean setMaxIntentos(int maxIntentos) {
        if (maxIntentos < Config.MAX_INTENTOS_LIMINF ||
            maxIntentos > Config.MAX_INTENTOS_LIMSUP)
            return false;
        this.maxIntentos = maxIntentos;
        return true;
    }

    public int getTamNumero() { return tamNumero;    }

    public int getMaxIntentos() { return maxIntentos;  }
}
```

10. Clases Abstractas e Interfaces

Cuestiones:

A partir de las clases siguientes:

```
public abstract class PoligonoRegular {
    double tamañoLado;
    abstract double area();
    abstract int getCantidadLados();
    double perimetro() { return getCantidadLados() * this.tamañoLado;    }

    PoligonoRegular (double tl) { this.tamañoLado = tl;    }
}

class Pentagono extends PoligonoRegular {
    Pentagono (double t ) {super (t); }

    @Override
    int getCantidadLados () {return 5;}

    @Override
    double area () { return 1.72048d * Math.pow(this.tamañoLado,2);
    }
}
class Hexagono extends PoligonoRegular {
    Hexagono (double t ) {super (t); }

    @Override
    int getCantidadLados () {return 6;}

    @Override
    double area () {
        double lado = this.tamañoLado;
        double apotema = Math.sqrt((lado *lado)-((lado/2)*(lado/2)));
        return lado  * apotema * 3;
    }
}
```

a) Prueba a crear instancias de hexágonos y pentágonos sobre variables de tipo *PoligonoRegular* y comprueba mediante alguna calculadora online que calcula correctamente su área y perímetro.
b) ¿Por qué *PoligonoRegular* es abstracta?
c) ¿El método *área()* podría ser no abstracto?
d) ¿Es posible crear una clase hija de *PoligonoRegular* sin desarrollar el método *área()*?
e) ¿Puedo crear instancias de *PoligonoRegular*?
f) ¿Por qué el método *perímetro()* no es abstracto si no se calcula igual para cada tipo de polígono regular? (pentágono es lado por 5, hexágono es lado por 6, etc.)
g) ¿El siguiente código es correcto? Explica por qué funciona.

```
ArrayList <PoligonoRegular>  listaPoligonos = new ArrayList <>();
for (PoligonoRegular p : listaPoligonos)
      System.out.println(p.area());
```

h) Si no existiesen las clases abstractas y el método *área()* lo definiésemos en las clases hijas, ¿Funcionaría el código anterior? ¿Por qué?

Ejercicios:

10.1. Copia la última versión de las clases de los primeros ejercicios: *Figura2D*, *Triangulo*, *TrianColor* y *Rectangulo* (añade sufijo _v9) y realiza los siguientes cambios:

- Crea un método abstracto llamado *area ()* en Figura2D que ha de implementarse en las clases hijas.
- Crea un método *precio (float precioMetroCuadrado)* en la clase Figura2D, que use el método abstracto anterior.
- Verificar que las clases hijas implementan el método a*rea ().* ¿Qué ocurriría si no lo tuviesen implementado?
- Haz un programa que almacene figuras de los tres tipos en un ArrayList y finalmente se recorra el ArrayList con un *for-each* mostrando el precio de cada figura, suponiendo un precio de 10 euros el metro cuadrado.

10.2. Diseña una clase abstracta llamada *Figura3D_v1* con método abstracto *volumen* (). Crea subclases: *Esfera_v1* y *PrismaRectangular_v1* que implementen el método de la superclase. Incorpora los atributos que creas necesarios a las tres clases. Haz un programa que cree una instancia de cada una de las 2 figuras y muestre cuál de ellas ocupa más.

10.3. Haz una nueva versión de las clases del ejercicio anterior (añádele el sufijo _v2). Añádele a la clase base el método abstracto *superficie* (). Crea una nueva clase Cilindro y haz que implemente los métodos de la superclase. Haz un programa que permita crear una instancia de cada una de las 3 figuras y nos muestre cuál tiene más superficie.

10.4. Copia el ejercicio del capítulo anterior de la lista de espera del médico y redefine la superclase *Paciente* como abstracta haciendo el método *facturar()* abstracto, ya que se implementa en las tres clases hijas.

10.5. Pensando que en el futuro implementemos el juego de ajedrez para dos jugadores, se desea crear una clase abstracta llama *PiezaAjedrez,* con dos atributos enteros llamados *fila* y *columna* que representan sus coordenadas en el tablero (valores entre 0 y 7) y un método abstracto llamado *mover ()* al que se le pasan como parámetro la *fila* y *columna* destino de un movimiento. El método devolver *true* si el movimiento se puede realizar o *false* si es un movimiento erróneo. Implementa esa clase y sus subclases *AlfilAjedrez* y *TorreAjedrez*. Para simplificarlo, vamos a pensar en movimientos en un tablero vacío, es decir solo con una pieza, la que se está movimiento.

- Haz un programa que permita al usuario mover una sola pieza (al comenzar elegirá *Alfil* o *Torre*) por el tablero, partiendo de la posición 0,0, indicando las coordenadas destino del movimiento que quiere hacer cada vez, terminando el programa cuando introduzca fila -1.
- El programa tendrá una función que presente por pantalla la situación del tablero.
- Por comodidad, puedes hacer las clases dentro del mismo fichero que el programa.

10.6. Modifica la clase *PiezaAjedrez* (versión _v2) para incluir métodos ¿estáticos? para que el usuario introduzca la *columna* como letra (a-h) y la fila (entre 1 y 8) y los convierta a los valores usados previamente (entre 0 y 7). Esto obligará a generar una nueva versión del programa anterior, para que el usuario introduzca a-h y 1-8 como valores destino del movimiento.

10.7. Crea una interfaz llamada *Perimetrable* con un solo método llamado *calcularPerimetro* al que no se le pasa ningún parámetro y devuelve un *double* con el perímetro de la clase que la implemente. Copia las clases *Figura2D*, *Triangulo*, *TrianColor* y *Rectangulo* en su última versión (añade sufijo _v10) y modifica la nueva *Figura2D* para que implemente *Perimetrable*.

- ¿Qué ocurre al añadir esta implementación? ¿Se produce algún error de compilación?
- ¿Por qué? ¿Hay alguna forma rápida de arreglar ese error?
- Modifica las clases necesarias para que calculen el perímetro correctamente *(supón que los triángulos son equiláteros para simplificar el cálculo)*

Haz un programa sencillo que cree instancias de *Triangulo*, *TrianColor* y *Rectangulo* y muestre su perímetro.

10.8. Crea una nueva clase *PoligonoIrregular* (no es hija de Figura2D ni de ninguna otra clase) que tiene como único atributo un ArrayList con la longitud de todos los lados del mismo. Haz que esta clase implemente *Perimetrable.* Crea un programa sencillo con una instancia de *PoligonoIrregular* y que calcule su perímetro. Añade al programa un ArrayList de figuras (*PoligonoIrregular, Rectangulo_v10, Triangulo_v10, Triancolo_v10*) y que muestre el perímetro de todos ellos.

- ¿Tiene sentido que *Perimetrable* sea interfaz y no clase abstracta? ¿Por qué?

10.9. Desarrolla una interfaz llamada *Ciclista* con un método llamado *recorrer()* al que se le pasa un número de kilómetros y una cadena con tipo de terreno y devuelve los segundos que tarda en recorrerlo. Una interfaz llamada *Nadador* con un método llamado *nadar* a la que se le pasan metros y devuelve los segundos en recorrerlo, y por último otra interfaz llamada *Saltador* con un método *saltarAltura* que no recibe parámetros y que devuelve los centímetros saltados.

- Desarrolla una clase *Delfin* que implemente la interfaz *Nadador*. El tiempo en recorrer una distancia es aleatorio entre 40km/h y 70km/hora
- Desarrolla una clase *BallenaAzul* que implemente la interfaz *Nadador*. El tiempo en recorrer una distancia es de 10km/h para las mayores de 5 años y 13km/h para las menores.
- Desarrolla una clase *TriAtleta* que implemente las tres interfaces, con los criterios que tu consideres (pueden devolver valores aleatorios entre unos mínimos y máximos que tú decidas o tener en cuenta otros parámetros como edad, sexo, etc.).
- Realiza un programa sencillo que cree instancias de delfines, ballenas azules y triatletas y use los métodos desarrollados.

10.10. Empleando las tres clases definidas en el ejercicio anterior (*Delfín*, *BallenaAzul* y *TriAtleta*) y la interfaz *Nadador*, realiza un programa que contenga un ArrayList de nadadores (polimorfismo de interfaces) con tres elementos, uno de cada tipo, esto es un *Delfín*, una *BallenaAzul* y un *TriAtleta.* A continuación, recorre el ArrayList con un bucle for...each, mostrando cuánto tarda cada uno de ellos en recorrer un kilómetro.

10.11. Duplica las interfaces y las clases del ejercicio anterior (añadiéndoles el sufijo _v2) Modifica la interfaz *Saltador_v2* añadiéndole el método *saltaPertiga* que recibe como parámetro una altura en centímetros y devuelve *true* si ha logrado el salto y *false* si no lo ha logrado ¿Qué ocurre con la clase *TriAtleta_v2?* Desarrolla *saltaPertiga* como método *default* en la interfaz de forma que por defecto devuelva *false.*

- Haz una nueva versión de *TriAtleta* (sufijo _v2b) que implemente *saltaPertiga* con este criterio: para saltos de más de 6 metros devuelve *false,* entre 5 y 6 metros devuelve *true* la mitad de las veces y por debajo de 5 metros devuelve siempre *true.*
- Haz un programa que cree una instancia de *Triatleta_v2* y otra de *Triatleta_v2b* y que muestre el resultado de ambos atletas saltando: 100cm, 550cm, 560cm, 580cm, 700cm.

10.12. Añade el sufijo _v2 a las clases de capítulos previos: *movilPrepago, movilTarifaPlana, movilPlus* y crea una interfaz llamada *PrePagable* que estas clases deberían implementar. Incluye en la interfaz todos los métodos que creas oportuno y, si es necesario, alguno puede ser método por defecto.

10.13. Se desea desarrollar un programa gestione los dispositivos domóticos de un edificio. Para ello tendremos un ArrayList que contenga en principio 3 elementos, uno para el termostato de la calefacción, otro para el ascensor y otro para el dial de la radio del hilo musical, pero en el futuro podríamos tener más elementos.

El termostato tiene una fecha de última revisión, un valor entero en grados centígrados: mínimo 15, máximo 80 y la temperatura inicial es 20. El ascensor tiene una planta en la que se encuentra, pudiendo ser desde 0 a 8. La planta inicial es la cero. El dial de radio va desde 88.0 a 104.0 avanzando de décima en décima, siendo el valor inicial 88.0.

De cada elemento (y los futuros que aparezcan) deben ser capaces de realizar las siguientes funciones:

- **subir**(), incrementa una unidad el elemento domótico. Devuelve true si la operación se realiza correctamente o false si no se puede hacer por estar al máximo.
- **bajar**(): decrementa una unidad el elemento domótico. Devuelve true si la operación se realiza correctamente o false si no se puede hacer por estar al mínimo.
- **reset**(): pone en la situación inicial el elemento domótico. No devuelve nada.
- **verEstado**(): Devuelve un String con el tipo de dispositivo y su estado actual.

Además, el termostato debe incluir un nuevo método:

- **revisar**(). Fija a la fecha de revisión a la fecha actual. No devuelve nada.

Una vez definido el sistema, crea un programa que inicie un ArrayList con una instancia de cada uno de los 3 dispositivos y luego mediante un menú nos permita hacer operaciones, primero qué operación queremos realizar (0:Salir, 1:subir un dispositivo, 2:bajar un dispositivo, 3: resetear un dispositivo, 4:revisar termostato) y luego seleccionar sobre que elemento queremos trabajar (verificando que sea un valor entre 0 y el tamaño del ArrayList -1)

- El menú, además de las opciones nos mostrará siempre el estado de todos los dispositivos.

10.14. Se desea hacer la gestión de las habitaciones de un hotel. Todas las habitaciones tienen un número de habitación y un proceso de check-in y check-out. En el hotel hay tres tipos de habitaciones, aunque podría haber más en el futuro:

- ✓ 3 habitaciones Lowcost (cuesta 50 euros/dia todo el año).
- ✓ 10 habitaciones dobles. Tienen una tarifa normal de 100 euros/dia y un incremento del 20% si el día de salida es abril, julio o agosto.
- ✓ 5 habitaciones Suite. 200 euros/día con 20% de descuento para estancias de 10 o más días.

- El programa inicialmente meterá las 18 habitaciones en un ArrayList y las marcará como "no ocupadas".
- El programa tendrá un menú para hacer check-in entre las habitaciones libres, check-out entre las ocupadas y listar habitaciones libres y ocupadas.
- El check-in es común para todas las habitaciones, consiste en marcar la habitación como ocupada. El check-out consiste en marcar la habitación como libre y calcular el importe a pagar que se calculará en función del tipo de habitación y de los días de estancia (quizás sea necesario almacenar la fecha de llegada en el momento del check-in)

Cuestión 1: ¿*Habitacion* debería ser una clase abstracta o una interfaz? ¿Por qué?
Cuestión 2: ¿Es útil que el método *toString()* en la clase *Habitacion*?

- Mantener toda la información en las clases más que en el programa que las utiliza.
- Puedes crear una clase Hotel, que tendrá un ArrayList de Habitaciones, o bien definir ese ArrayList en el programa como variable global y no tener la clase Hotel.
- Sugerencia: Para probar el programa, al hacer el check-out, puedes considerar cada día como un segundo, así, si han pasado 3 segundos, considerar 3 días.

10.15. Haz una versión del ejercicio anterior pensando que el programa podría usarse para alquilar otro tipo de elementos nuevos (no habitaciones, por ejemplo, coches) con unos atributos totalmente distintos y con una forma de facturar distinta, pero el ArrayList debe poder contener cualquier tipo de objeto siempre que tengan implementados los métodos *checkIn* y *checkOut*.

Proyecto:

Sobre el proyecto del tema anterior, vamos a comprobar el polimorfismo de interfaces cambiando el atributo de tipo ArrayList que contiene los intentos del juego, por la interfaz List.

Adicionalmente, añade una nueva opción al menú principal llamado "Ayuda" que explique la mecánica del juego.

11. PAQUETES

Las preguntas que se formulan en los ejercicios se pueden contestar en el propio código.

11.1. Crea un paquete llamado *pClasesApoyo*. Dentro de *pClasesApoyo* crea un subpaquete llamado *pMates*. Haz las siguientes pruebas, comentando los problemas que te encuentres.

- En el paquete *pClasesApoyo* crea una clase **pública** *Dado* con un método **público estático** *lanzar* () que devuelva el resultado aleatorio de lanzar un dado.
- En el paquete *pClasesApoyo* crea una clase con acceso ***default*** *DadoFalso* con un método **público** *lanzar (int n)* que devuelva el resultado aleatorio de lanzar un dado, pero trucado, de forma que el número *'n'* pasado como parámetro tiene el doble de posibilidades de salir que el resto.
- En el paquete *pMates*, crea una clase pública *Calculadora* que tiene definido un ArrayList **público** de enteros, con métodos: **private** *sumar* (), **protected** *menor* () y **public** *media* () que operan sobre el ArrayList de la clase devolviendo la suma, el menor y la media respectivamente.

  Para calcular la media puedes usar el método sumar();
- En el paquete *pMates*, crea una clase hija de *Calculadora*, llamada *CalculadoraPro,* acceso **default**.
  - Tiene un método similar a *sumar*() de la clase base pero **público** y que devuelve el valor en hexadecimal ¿Podemos hacer *super.sumar*() para calcular la suma? ¿Hay @Override?
  - Tiene un método similar a *menor*() de la clase base pero **público** y que devuelve cero si el menor del ArrayList es menor que cero ¿Podemos hacer *super.menor*() para calcular la suma? ¿Hay @Override?
- En el paquete *pClasesApoyo* crear una clase *CalculadoraSuper* hija de *Calculadora*, pero no es necesario que le añadas ningún método. ¿Necesita algún constructor? ¿Por qué? ¿Podría ser hija de *CalculadoraPro*?
- Realiza un programa (en el paquete en el que trabajas habitualmente) que emplee los métodos de *Calculadora* (primero creando una instancia con un ArrayList con unos valores cualquiera) y luego intentando llamar a sus tres métodos: *sumar(), menor(),media()*. Comenta los problemas de acceso que te encuentres y sus causas.

11.2. Realiza un programa (en el paquete en el que trabajas habitualmente) que emplee los métodos de *CalculadoraPro* (primero creando una instancia con un ArrayList con unos valores cualquiera) y luego llamando a sus tres métodos. Comenta los problemas de acceso y sus causas.

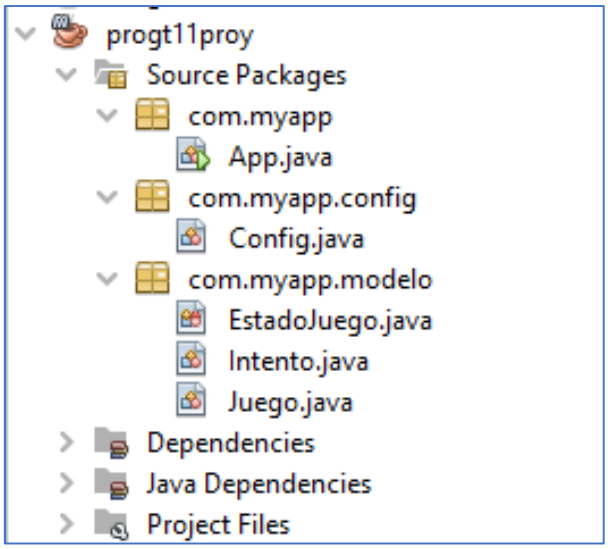
11.3. Realiza un programa (en el paquete en el que trabajas habitualmente) que emplee los métodos *lanzar*() de la clase *Dado* y *DadoFalso*. ¿Algún problema de acceso? Enumera distintas formas de solucionarlo: ¿cambiando el acceso a las clases? ¿cambiando de paquete el ejercicio?

11.4. Realiza un programa que utilizando el método *lanzar()* de la clase *Dado* anterior y la constante *PI* de la clase *Math*, simule que lanzamos un dado y muestre el área del círculo que tuviera por radio el valor obtenido en el dado. Ejemplo: si sale en el dado un 3, el área sería 3* 3 * PI. Se pide que no haga falta añadir el prefijo con la clase ni al método *lanzar*() ni a la constante PI.

Proyecto:

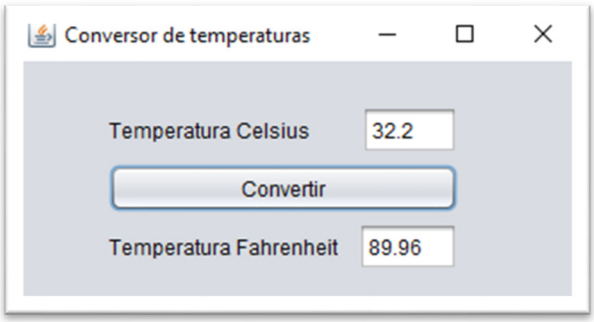
Sobre el proyecto del tema anterior, podemos estructurar mejor la aplicación en paquetes. La clase con el main podemos dejarla en la carpeta que está, pero podemos crear un paquete llamado "modelo" y meter en él: la clase "Juego", la clase "Intento" y la enumeración con el estado de Juego.

Podemos crear otro paquete llamado "config" para la clase "Config" y todo lo que pueda surgir en un futuro referente a la configuración del programa.

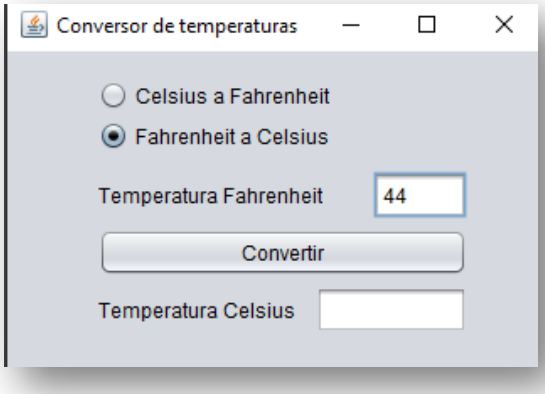


12. Interfaz Gráfica de Usuario

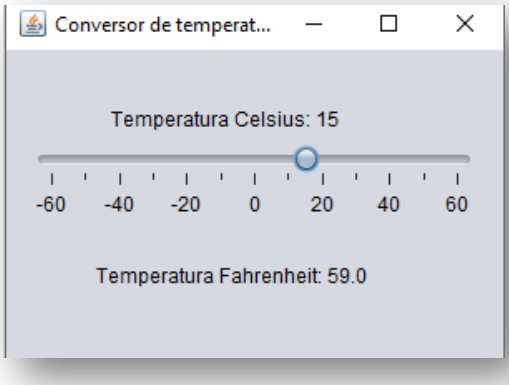
12.1. Realiza un programa como el de la imagen que permita convertir entre grados Celsius a Fahrenheit. Implementa una clase *Conversor* con métodos estáticos para pasar de Celsius a Fahrenheit y viceversa. Ponle título a la ventana.



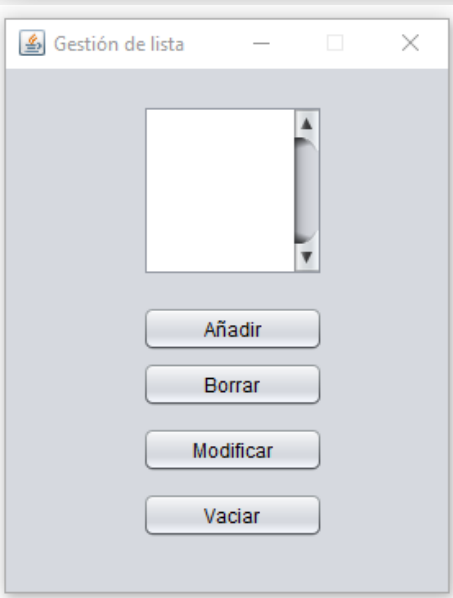
12.2. Realizar una nueva versión del programa anterior, pero con unos botones de opción para que sea bidireccional, de Celsius a Fahrenheit y también de Fahrenheit a Celsius. Cada vez que cambiemos la selección de los botones de opción cambiarán los textos de las etiquetas y se vaciarán las cajas de texto. El programa informará mediante sendas ventanas emergentes tanto si no se ha seleccionado ningún botón de opción, como si la caja de texto de entrada está vacía.



12.3. Realizar una nueva versión del primer programa, pero sustituyendo las cajas de texto y el botón por un *slider* como muestra la figura:

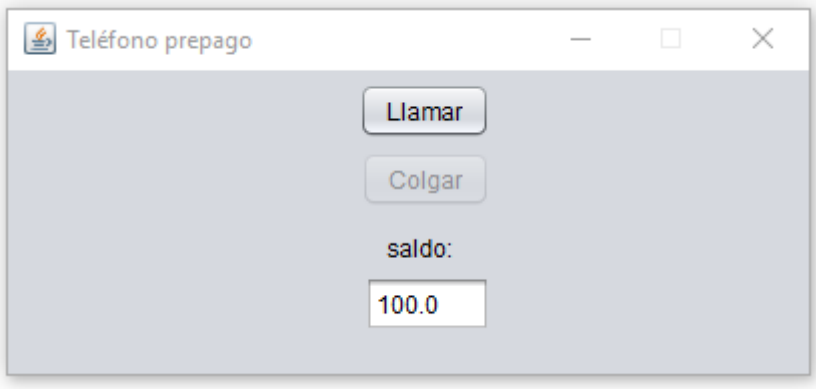


12.4. Realizar un programa con una lista con los nombres de los alumnos de una clase (inicialmente vacía). El programa dispondrá de un botón para añadir a la lista (mediante un diálogo pedirá el nombre a usuario), un botón para eliminar el elemento seleccionado de la lista (si no hay ninguno muestra un mensaje en un diálogo), un botón para modificar un elemento seleccionado (solicitará el nuevo nombre con un dialogo para introducirlo) y finalmente un botón para vaciar la lista (este botón pedirá confirmación mediante un diálogo).

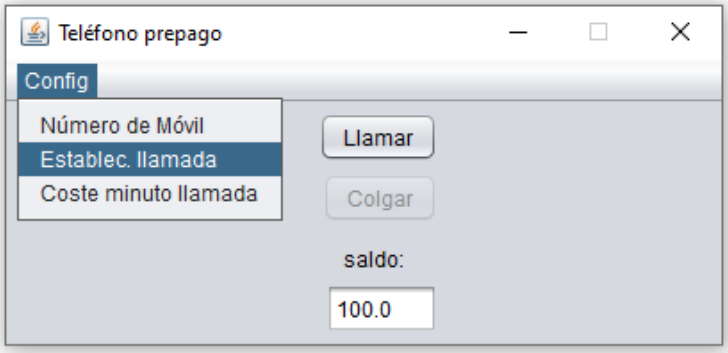


12.5. Copia la clase *MovilPrepago* como una nueva versión *(MovilPrepago_v3)* y haz una aplicación con botones para llamar y colgar que permita efectuar llamadas.

- Cuando se pulsa el botón de llamar comienza una llamada y se habilita el botón de colgar (y se deshabilita el de llamar).
- Cuando se pulsa el botón de colgar se calculan los segundos de la llamada y se actualiza el saldo, se habilita de nuevo el botón de llamar y se deshabilita el de colgar.
- Opcional: Añádele un botón para recargar.

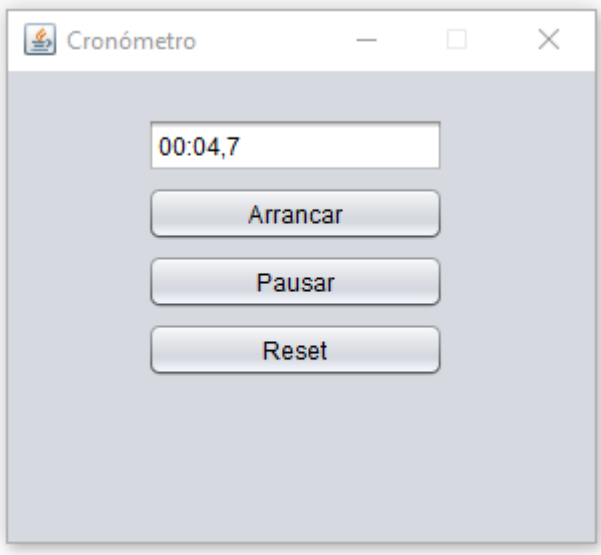


12.6. Haz una nueva versión del programa anterior con un menú superior llamado *Config* con 3 opciones, una para establecer el número de móvil, otra para el coste de establecimiento de llamada si tu clase dispone de ese atributo y una última para establecer el coste por minuto de llamada. Pueden ser necesarios *setters* en la clase *MovilPrepago_v3*.



12.7. A partir de la clase *Cronometro* que te proporciona el profesor, realiza un programa como el de la figura, con un contador de minutos, segundo y décimas de segundo que se puede arrancar (inicialmente a cero, esto es 00:00,0)

- Emplea la clase *swing.Timer.*
- Añade un método *toString* a la clase Cronómetro, que prepare el texto formateado a mostrar en el campo de texto (MM:SS,D)



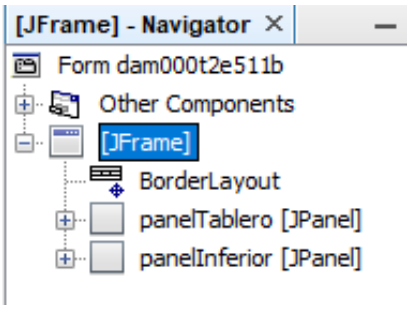
12.8. (Opcional) Realizar un programa que incluya la imagen proporcionada por el profesor (*bola.gif)* y que, mediante las teclas de cursor arriba/abajo sea capaz de moverla hasta los límites superior e inferior respectivamente de la ventana de la aplicación.

12.9. (Opcional) Realizar una nueva versión del programa anterior, pero que la imagen se mueva de arriba abajo automáticamente, rebotando en sentido contrario al llegar al final límite de la ventana.

12.10. Realizar un programa que dibuje un tablero de ajedrez con un array de 8 x 8 botones, en colores alternos. Al pulsar un botón, mostrará una ventana de diálogo con la fila y columna del botón pulsado. Pista: la fila y columna pueden ser reflejadas en el nombre del botón manipulable con setName() y getName().
Para la colocación de los botones en forma de tablero hay dos opciones: especificarlo "a mano", de forma que le decimos donde situarlo con el método setbounds() o bien mediante el uso del tipo de panel GridLayout de 8 filas y 8 columnas de forma que al añadirle los Jbutton, se van colocando por filas y columnas, uno al lado del otro.

12.11. Ejercicio guiado: Buscaminas.
A partir de la clase *BuscaMinas* que te ha entregado el profesor (puedes construirla desde cero si quieres), elabora una aplicación gráfica para jugar al clásico juego del Buscaminas.

- Crea un JFrame. A continuación, desde la ventana navegador, botón derecho sobre ella > *Set Layout > Border Layout.*
- Arrastramos un JPanel desde la paleta a la parte inferior del JFrame, y lo soltamos cuando la línea de puntos amarilla nos muestre la zona inferior del *frame.* Esto indica que estamos situando el panel en el área *South.* Desde la ventana Navigator, botón derecho sobre este panel: lo llamamos *panelInferior* cambiándole el nombre de la variable Luego veremos que contendrá el botón para configurar el número de bombas del tablero.
- Arrastramos otro JPanel desde la paleta a la parte central del *frame,* y le llamaremos *panelBotones.* Desde la ventana *Navigator,* botón derecho sobre este panel: *Set Layout > GridLayout*. Luego le asignaremos todos los botones.



- En las propiedades de ambos paneles podemos fijar su *PreferedSize* para establecer su tamaño, 500 x 500 para el primero y 500 x 50 para el segundo.
- El JFrame deberá tener también como atributo una instancia de la clase *BuscaMinas* para almacenar la lógica y el estado del juego.

```
private BuscaMinas juego;
```

- También deberá tener un atributo que será un Array de 8 x 8 botones.

```
JButton tablero [ ] [ ];
```

  Para cada uno de los botones deberemos, en el constructor, asignarle un nombre (fila+columna, será útil luego), color de fondo, etc... y añadirlo al panel. También habrá que añadirle un evento (*ActionEvent*)

```
tablero = new JButton [8][8];
for (int f=0;f<8;f++){
   for (int c = 0; c < 8; c++) {
      tablero [f][c] = new JButton();
      tablero [f][c].setFont(new java.awt.Font("Tahoma", 0, 36));
      tablero [f][c].addActionListener(new java.awt.ActionListener() {
               public void actionPerformed(java.awt.event.ActionEvent evt)
                                 {FActionPerformed(evt);}});
      tablero [f][c].setName(Integer.toString(f)+Integer.toString(c));
      tablero [f][c].setBackground(Color.white);
      tablero [f][c].setText("");
      panelTablero.add(tablero[f][c]);
   }
}
```

- En el constructor del JFrame hay que llamar al constructor de la clase *BuscaMinas* para crear el juego en sí, que tiene la lógica del juego: coloca las minas, los valores de cada celda, etc.

```
this.juego = new BuscaMinas (8,8,8);
```

- Implementar el código asociado al evento, con las siguientes funciones:
  - Hay que detectar el botón que lo ha causado, el que se ha pulsado. En el constructor le dimos a cada botón como nombre la fila y columna en la que se encontraba, así que para saber el botón pulsado:

```
String nn = ((JButton)evt.getSource()).getName();
int fila = Integer.parseInt(nn.substring(0,1));
int col = Integer.parseInt(nn.substring(1,2));
```

  - Luego llamamos a la lógica del juego con esa posición, a ver que hay en esa casilla. Si hay bomba se acaba el juego, cambiamos el color de fondo a rojo y acabaría el juego:

```
int x = juego.elegir (fila,col);
 if (x == -1 )  {  //perdiste
          ((JButton)evt.getSource()).setBackground(Color.red);
          JOptionPane.showMessageDialog(this,"Has perdido");
          System.exit(0);
} else ...
```

  - Si no es bomba, hay que cambiarle el color de fondo a amarillo y mostrar su valor, es decir, el número de bombas colindantes *(ese valor lo calcula la clase BuscaMinas)*. Por las normas del juego, si ese valor es cero, es decir sin bombas colindantes, se descubren automáticamente las 8 celdas que la rodean (y así recursivamente). Eso lo hace la clase *BuscaMinas*, aquí lo que hay que hacer es poner el fondo amarillo y el texto con el valor a todo lo que se haya descubierto (la casilla pulsada y las posibles descubiertas automáticamente).

```
... else {
        for (int f=0;f<8;f++){
          for (int c = 0; c < 8; c++) {
            if (juego.casillas[f][c].descubierta) {
              tablero [f][c].setText(Integer.toString(juego.casillas[f][c].valor));
              tablero [f][c].setBackground(Color.yellow);
            }
          }
        }
... if
```

  - Finalmente, en este caso, habrá que ver si ya ganó el jugador.

```
... if (juego.ganaste()) {
     JOptionPane.showMessageDialog(this,"Has ganado");
     System.exit(0);
  }
```

- Una vez que funcione todo lo anterior y podamos jugar, podemos implementar alguna mejora:
  - Al acabar una partida, preguntar al usuario si quiere jugar una partida nueva. Para ello, habrá que llamar de nuevo al constructor de la clase *BuscaMinas* para que genere un nuevo tablero y volver los botones a su situación inicial.

```
this.juego = new BuscaMinas (8,8,8);
for (int f=0;f<8;f++){
   for (int c = 0; c < 8; c++) {
      tablero [f][c].setBackground(Color.white);
      tablero [f][c].setText("");
   }
}
```

  - Cuando acaba una partida (haya ganado o no), se podría mostrar al usuario la solución, es decir todas las casillas descubiertas:

```
for (int f=0;f<8;f++){
        for (int c = 0; c < 8; c++) {
           if (juego.casillas[f][c].valor!=-1) {
               tablero [f][c].setText(Integer.toString(juego.casillas[f][c].valor));
               tablero [f][c].setBackground(Color.yellow);
           }
           else {
               tablero [f][c].setBackground(Color.red);
           }
        }
     }
```

  - Se podría incorporar un botón que permitiese cambiar el número de bombas, lo que implicaría empezar la partida de nuevo.

  - Se podría incorporar una utilidad, que el usuario pudiese marcar donde cree que hay una bomba. Clicaría con el botón derecho y cambiaríamos el color de fondo a naranja, pero no tendría ningún efecto sobre el juego.

```
//hay que cambiar el evento de ActionListener a MouseAdapter:
tablero [f][c].addMouseListener(new java.awt.event.MouseAdapter() {
          public void mouseClicked(java.awt.event.MouseEvent evt) {FActionPerformed(evt);}});
//y el código del evento:
private void FActionPerformed(java.awt.event.MouseEvent evt) {
    if (((java.awt.event.MouseEvent) evt).getButton() == java.awt.event.MouseEvent.BUTTON3  &&
       ((JButton)evt.getSource()).getBackground()!= Color.yellow                    ){
        if (((JButton)evt.getSource()).getBackground()!=Color.orange)
           ((JButton)evt.getSource()).setBackground(Color.orange);
        else
        ((JButton)evt.getSource()).setBackground(Color.white);
       return;
    }
    //...si no es click derecho sigue la ejecución con el código ya escrito
```

12.12. Ejercicio guiado: Juego de Nim.

A partir de la clase *Nim* que te ha entregado el profesor, elabora una aplicación gráfica para jugar al clásico juego del Nim.

- Crea un JFrame.
- El JFrame deberá tener también como atributo una instancia de la clase *Nim* para almacenar la lógica y el estado del juego. También deberá tener un atributo que será un Array de 3 x 7 botones (aunque no los instanciaremos todos) y por último un array para almacenar cuantos botones hay en cada fila.

```
Nim juego;
JButton tablero [ ] [ ];
int [] maxFila  = {3,5,7};
```

En el constructor del JFrame, instanciaremos los dos atributos anteriores *(juego y tablero)* y crearemos 3 botones para la primera fila, 5 para la segunda y 7 para la tercera, por lo que los crearemos con un bucle. Para cada uno de los botones deberemos, en el constructor, asignarle un nombre (fila+columna, será útil luego), color de fondo, etc… añadirlo al jFrame y asígnale una posición con *setBounds* en función de su fila y columna. También habrá que añadirle un evento (*ActionEvent*). En este caso no usamos GridLayout, los colocamos "a mano". Para hacer todo esto, en el constructor, después de *initComponents()* añadiremos:

```
juego = new Nim();
tablero = new JButton [3][7];
for (int f=0;f<3;f++){
    for (int c = 0; c < maxFila[f]; c++) {
          tablero [f][c] = new JButton();
          tablero [f][c].addMouseListener(new java.awt.event.MouseAdapter() {
                @Override
                public void mouseEntered(java.awt.event.MouseEvent evt) {FMouseEntered(evt);}
                @Override
                public void mouseExited(java.awt.event.MouseEvent evt)  {FMouseExited(evt); } });
                      tablero [f][c].addActionListener(new java.awt.event.ActionListener() {
                      public void actionPerformed(java.awt.event.ActionEvent evt)
                            {FActionPerformed(evt);}});
               tablero [f][c].setName(Integer.toString(f)+Integer.toString(c));
               tablero [f][c].setBackground(Color.white);
               add(tablero[f][c]);
               tablero [f][c].setBounds(c*70,f*70,60,60);
  }  }
```

Se usa el evento seleccionar (click) pero también los eventos de ratón de entrada y salida en el botón para mostrar al usuario los botones (palillos del juego) que tiene seleccionados y listos para retirar cuando haga clic.

- Cuando entra el ratón en un botón (sin clicar) el cambia el color de fondo y marca todos los botones a la derecha como se indicó en el paso anterior.

```
private void FMouseEntered(java.awt.event.MouseEvent evt) {
    String nn = evt.getComponent().getName();
    int fila = Integer.parseInt(nn.substring(0,1));
    int col = Integer.parseInt(nn.substring(1,2));
    for (int i=col; i< maxFila[fila]; i++) {
       tablero[fila][i].setBackground(Color.gray);
    }
  }
```

  Cuando sale el ratón de un botón deshacemos el paso anterior.

```
private void FMouseExited (java.awt.event.MouseEvent evt) {
    evt.getComponent().setBackground(Color.red);
    String nn = evt.getComponent().getName();
    int fila = Integer.parseInt(nn.substring(0,1));
    int col = Integer.parseInt(nn.substring(1,2));
    for (int i=col; i < maxFila[fila]; i++) {
      tablero[fila][i].setBackground(Color.white);
    }
}
```

- Al clicar en un botón, obtenemos su fila y columna, haremos invisibles ese botón y los que están a su derecha y llamamos a la lógica del juego con la jugada del usuario, si no gana jugará también la máquina.

```
private void FActionPerformed(java.awt.event.ActionEvent evt) {
    evt.getSource();
    String nn = ((JButton)evt.getSource()).getName();
    int fila = Integer.parseInt(nn.substring(0,1));
    int col = Integer.parseInt(nn.substring(1,2));
    for (int i=col; i < maxFila[fila]; i++) {
      tablero[fila][i].setVisible(false);
    }
   jugadaUsuario (fila,col);
    jugadaMaquina();
}
 public void jugadaUsuario (int f, int c) {
    int cant = maxFila[f]-c;
    boolean ok = juego.Juega(f, cant);
    if (juego.Fin()) FinPartida("Has ganado");
    else  maxFila[f]=c;
 }
 public void jugadaMaquina () {
    int jugada = juego.Piensa();
    int f = (int) (jugada /10);
    int cant = jugada % 10;
    JOptionPane.showMessageDialog(this,"Quito " + cant + " de la fila " + (f+1));
    for (int i=maxFila[f]-cant; i < maxFila[f]; i++) {
      tablero[f][i].setVisible(false);
    }
    boolean ok = juego.Juega(f, cant);
    if (juego.Fin()) FinPartida ("He ganado");
    else  maxFila[f]-=cant;
 }
 public void FinPartida(String mensaje){
     JOptionPane.showMessageDialog(this,mensaje);
     System.exit(0);
 }
```

- Una vez que funcione todo lo anterior y podamos jugar, podemos implementar alguna mejora:
  - Al acabar una partida, preguntar al usuario si quiere jugar una partida nueva. Para ello, habrá que llamar de nuevo al constructor de la clase Nim y restaurar los valores del array maxFila[] ya que se han modificado durante la ejecución de la partida. También habrá que hacer visibles de nuevo todos los botones.

```
public void FinPartida(String mensaje){
      JOptionPane.showMessageDialog(this,mensaje);
      int respuesta = JOptionPane.showConfirmDialog(this, "¿Otra partida?");
      if (respuesta == JOptionPane.OK_OPTION ) NuevaPartida();
      else System.exit(0);
}
public void NuevaPartida(){
   maxFila[0]=3; maxFila[1]=5; maxFila[2]=7;
   juego = new _Nim();
   for (int f=0;f<3;f++){
      for (int c = 0; c < maxFila[f]; c++) {
         tablero[f][c].setBackground(Color.white);
         tablero [f][c].setVisible (true);
      }
   }
   this.setVisible(true);
}
```

  - Preguntar al usuario al principio de cada partida si quiere que empiece la máquina o el usuario.

```
//al final de NuevaPartida():
int respuesta = JOptionPane.showConfirmDialog(this, "¿Empiezo yo?");
 if (respuesta == JOptionPane.OK_OPTION ) jugadaMaquina();
```

  - Añadirle a los botones un icono para hacer el programa más vistoso. Llevaría un icono para su estado normal (el gif animado *gema.gif* que te entrega el profesor) y otro para su estado cuando el ratón está encima (*gemaOver.gif)*

```
tablero [f][c].setIcon(new javax.swing.ImageIcon(getClass().getResource("/gema.gif")));
```

(*) los gif tienen que estar src\main\resources

En el método *FMouseEntered()* añadiremos ahora:

```
tablero[fila][i].setIcon(new javax.swing.ImageIcon(getClass().getResource("/gemaOver.gif")));
```

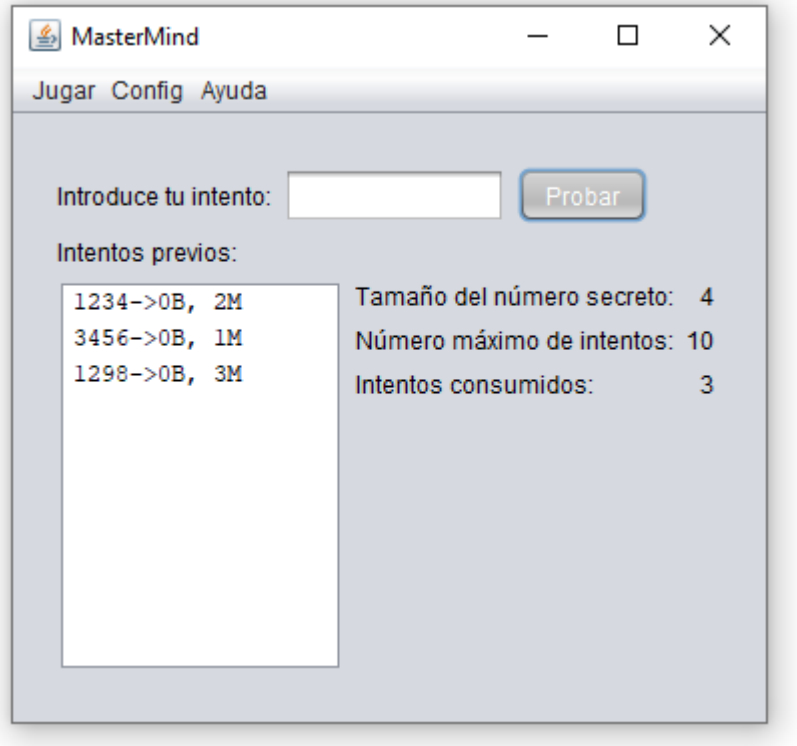
y en *FMouseExited()*:

```
tablero[fila][i].setIcon(new javax.swing.ImageIcon(getClass().getResource("/gema.gif")));
```

Proyecto:

En este capítulo debes convertir la aplicación de consola en una aplicación de entorno gráfico. No va a ser a hacer algo muy sofisticado, será algo sencillo para comprender la programación guiada por eventos.

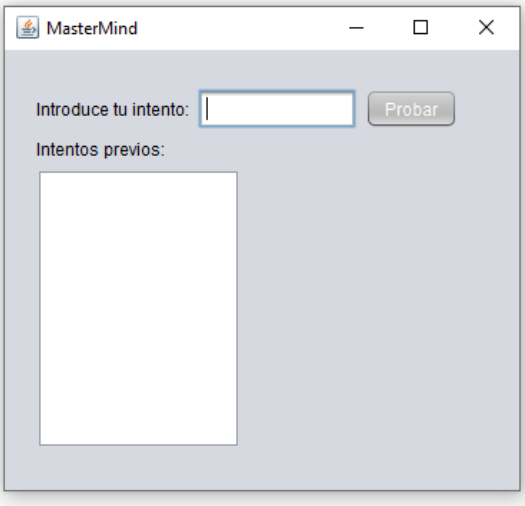
La siguiente imagen muestra la apariencia que podría tener la aplicación:



Aprovecha la utilidad de Netbeans para crear interfaces gráficas Swing arrastrando componentes y debes hacerlo poco a poco siguiendo estos pasos:

- Las clases con las reglas del juego son válidas tal cual sin modificación alguna. Aquí puedes comprobar las bondades de la separación de código en clases con distintas funciones y la diferencia entre la vista y la lógica de negocio.

- El primer paso es crear un JFrame con los elementos gráficos mínimos para jugar una sola partida, sin preocuparnos de la configuración (tamaño del número, cantidad de intentos) ni de la ayuda, etc. Sería algo así:



- La captura de datos y la información de situaciones de error la puedes hacer mediante ventanas emergentes (JOptionPane).

- Una vez que funcione correctamente puedes avanzar hasta obtener la aplicación completa como se muestra en la primera figura, mostrando más información en pantalla y con un menú superior con la funcionalidad de la aplicación:

  - Jugar > Nueva partida
  - Config > Tamaño del número a adivinar
  - Config > Número máximo de intentos
  - Ayuda > Mostrar instrucciones

- No borres la aplicación de consola, así conservas las dos opciones en el mismo proyecto (Desde Netbeans, con MAY+F6 decides cual ejecutar)

13. Excepciones

Cuestiones previas *(entregar en PDF)*

a) ¿Puede finalizar abruptamente esta porción de código?

```
Scanner teclado = new Scanner(System.in);
System.out.print("Num 1: ");int num1 = teclado.nextInt();
System.out.print("Num 2: ");int num2 = teclado.nextInt();
System.out.println("div=" + num1 / num2);
```

b) ¿Qué muestra este código? ¿Finaliza el programa abruptamente?

```
int [] a = new int [] {1,2,3,4,5};
int p = -1;
try { System.out.print(a[p]);  }
catch (ArrayIndexOutOfBoundsException e ) {System.out.print("0"); }
```

Antes de resolver correctamente cada ejercicio, provoca la excepción para así saber el nombre de la misma, para luego reflejarla en el *catch*, por ejemplo:

```
catch (ArrayIndexOutOfBoundsException e )
catch (InputMismatchException e )
```

13.1. Haz un programa que solicite al usuario dos números enteros y los intente dividir (división entera, sin decimales). Si se produce algún error el programa no finalizará de forma abrupta si no que capturará la excepción ocurrida: o bien entrada de valores incorrectos o bien por división por cero, informando al usuario de lo que ha acontecido. Finalmente, mostrará el resultado de la división, que será cero en caso de cualquiera de los dos errores posibles.

Nota: para provocar la excepción, hacemos la división de enteros, sin casting: double res=num1 /num2; con num2=0. Si hiciésemos el casting: double res = (double) num1 / num2; no se produce la excepción, informa que el resultado es 'Infinity'.

13.2. Haz un programa que tenga definido un Array de 7 posiciones de *double* que representa la temperatura media en una ciudad durante una semana (puedes generar valores al azar entre 0 y 40 para llenarlo). Se le solicitará al usuario que introduzca dos posiciones (entre 0 y 6), y calculará la temperatura media entre esos días de la semana, ambos inclusive. Si los valores introducidos no son válidos, por estar fuera de los límites del array se capturará la excepción y la media será cero.

13.3. Haz un programa que, desde el *main()*, solicite al usuario que introduzca tres enteros, correspondientes a día, mes y año de una fecha. Realiza una función llamada *validarFecha()* que reciba esos tres valores y construya una fecha con *LocalDate.of* . Si los valores no se corresponden con una fecha válida se producirá una excepción *DateTimeException* que deberá ser capturada en el *main.* Previamente, en caso de que los valores introducidos no sean enteros, el *main()* avisará de tal situación capturando la excepción en tipos de datos incorrectos al hacer la lectura de teclado.

13.4. Crear una función *leerEntero(int max, int min)* que solicite al usuario la entrada de un número entero y que valide que es un valor entero y que está comprendido entre los parámetros *max* y *min* (esto es, mayor o igual que *min* y menor o igual que *max).* El usuario deberá repetir la entrada de datos hasta que lo haga bien, devolviendo finalmente la función el valor introducido y validado. La función controlará las posibles excepciones (por ejemplo caracteres no numéricos).

13.5. Añadir a la función anterior un parámetro de texto que represente el típico mensaje que se le muestra al usuario antes de introducir un valor. La función hará el *System.out.print* de ese parámetro y evitamos que lo tenga que hacer el programa que utiliza la función. Ejemplo:

```
int edad = leerEntero ("Introduzca su edad", 0, 120);
```

ya no hay que hacer antes de la llamada *System.out.print ("Introduzca su edad");*

13.6. Realiza una función/método que se le pase por parámetro una cadena que representa una dirección de correo electrónico. En caso de que la cadena no cumpla las condiciones sintácticas de un email, se generará una excepción llamada *EmailInvalidoException* con mensaje: "Formato email inválido". Los requisitos que tiene que tener una dirección de email podrían ser: número mínimo de caracteres=5, una arroba y al menos un punto después de la arroba. Hacer también un programa que llame a la función anterior y capture la excepción generada.
En el tercer trimestre veremos las expresiones regulares que facilitan estas tareas de validación

Proyecto:

Sobre el proyecto anterior, haz que cuando el usuario configure erróneamente el número de intentos máximo o el tamaño del número a adivinar, se produzca una excepción. Las excepciones pueden ser, tanto si lo que introduce el usuario no es un número, como si el valor introducido no está dentro del rango permitido.

Puedes crear excepciones personalizadas o bien usar *RuntimeExcecption* con el mensaje adecuado.

Las excepciones serán capturadas tanto en la aplicación de consola como en la aplicación gráfica para que los mensajes de error lleguen al usuario.

14. Ficheros

Crea en el proyecto una carpeta llamada "archivos" para ubicar en ella los ficheros que se generen en este apartado.

14.1. Realizar un programa que escriba distintas líneas con un texto cualquiera (con eñes y acentos) en un fichero llamado *fich01.txt* especificando la codificación UFT-8. El fichero estará almacenado en la carpeta *archivos* creada previamente. Una vez que funcione realiza los siguientes cambios:

a) Incorpora *File.separator* para componer la ruta del archivo.
b) Modifica la codificación, pasándola a ISO-8859-1.

14.2. Basándote en el ejercicio anterior, hay un programa en el usuario introduzca palabras o frases y se vayan escribiendo como líneas separadas en un fichero llamado *fich02.txt*. El programa finaliza cuando el usuario teclee "fin". Para definir el objeto teclado para introducir texto especifica la codificación de la consola mediante: `Scanner teclado = new Scanner(System.in, "ISO-8859-1");`

14.3. (Opcional) Haz una versión del programa anterior, que compruebe previamente si el fichero *fich02.txt* existe o no. En caso afirmativo hará una copia del archivo como *fich02.bak* y seguirá añadiendo las líneas sobre *fich02.txt* .

14.4. Realizar un programa que lea secuencialmente el fichero *fich02.txt* línea a línea y lo muestre por pantalla, con el texto en mayúscula.

14.5. Haz una versión del ejercicio anterior que lea todas las líneas en una sola operación de lectura.

14.6. (Opcional) Realizar un programa con un método al que se le pase una cadena con la ruta y nombre de un fichero. El método lo leerá y devolverá un array cuyo primer elemento contendrá el número de caracteres del fichero. El segundo elemento del array debe contar el número de líneas del fichero. El método devolverá *null* si no existe el fichero. El programa debe tener un main que llame al método creado pasándole como parámetro *fich04.txt.* Mejora opcional: añade una tercera posición al array que contenga el número de palabras del fichero (puedes usar el método split).

14.7. Realizar un programa que escriba distintas líneas de texto en un fichero llamado *alumnos.csv*, que contenga en cada línea: nombre del alumno, fecha de nacimiento, y notas de las tres evaluaciones (con dos decimales). Cada uno de los campos estará separado por un punto y coma. El programa finalizará cuando se introduzca 'Z' como nombre de alumno. Este puede ser un caso donde puede ser muy cómodo usar la clase PrintWriter.

14.8. Realizar un programa que lea el fichero *alumnos.csv* línea a línea y finalmente muestre la cantidad de alumnos que tienen una nota final >=5 y el nombre del alumno con mejor nota. (La nota final se calcula redondeando un 20% de la primera evaluación más un 30% de la segunda más 50% de la tercera). Prueba a cargar el archivo *alumnos.csv* en una hoja de cálculo con LibreOffice Calc. Si el archivo tiene coma como separador decimal, tenemos que reemplazarlo por punto antes de convertir a número, si no fallará: Double.parseDouble(textoNota)

14.9. Realizar un programa que cree un ArrayList de 5 Trabajadores (pueden ser Asalariados o ConsultoresExternos). Inserta objetos de ambas clases con unos valores cualesquiera "a mano". A continuación, guarda los objetos serializados en un archivo llamado *fich16.dat.* Copia ambas clases de ejercicios anteriores y su superclase (añadiendo _v2 a su nombre) para adaptarlas a este ejercicio. Pista: La superclase debe implementar Serializable.

14.10. Realizar un programa que lea un archivo como el del ejercicio anterior y muestre por pantalla su contenido. Además, introducirá los objetos leídos en un ArrayList de tipo *Trabajadores_v2.*

14.11. Realizar un programa que solicite al usuario dos valores: el coste por MB de consumo de datos y el coste por minuto de llamada. A continuación, se construirá un fichero de tipo *Properties* con los parámetros introducidos. Un ejemplo de ese fichero podría ser:

```
#Fichero de Configuración.
#Mon Jun 07 19:55:07 CEST 2021
CosteConsumoDatos=0.3
CosteMinutoLlamada=0.2
```

Este fichero se usará en el ejercicio siguiente para dar de alta móviles: el usuario solo tendrá que introducir el número ya que el resto de parámetros los tomará de los parámetros leídos del fichero. Eso sí, todos los móviles tendrán las mismas tarifas.

14.12. Realizar un programa que inserte en un ArrayList 5 elementos de tipo *MovilPrepago* pero solo solicitando al usuario el número de móvil, ya que el resto de parámetros los tomará del fichero obtenido en el ejercicio anterior. Finalmente mostrará el contenido del ArrayList para comprobar que se han leído los datos correctamente.

Proyecto:

Realiza dos mejoras sobre el proyecto anterior:

El texto de instrucciones de la aplicación está escrito en el propio código, esto no es una buena práctica. Crea un archivo de texto con esas instrucciones. Cuando el usuario seleccione la opción "Ayuda", se leerán todas las líneas de ese fichero y se mostrarán por pantalla.

Hasta ahora, si un usuario cambiaba un parámetro (tamaño del número secreto o máximo de intentos) este cambio solo tenía vigencia para esa ejecución de la aplicación en concreto, pero no para siguientes ejecuciones. Si se guardan en un fichero de configuración y ese fichero se lee al iniciar la aplicación, corregimos este problema. Por hacerlo coherente, también puedes guardar en ese fichero los límites de ambos valores, aunque estos no se pueden modificar; esto permitiría cambiar los límites inferiores y superiores sin recompilar el programa, solo modificando ese archivo.

Para esta tarea puedes crea este archivo con formato "properties" con una estructura así:

```
maxintentos.valorActual=10
maxintentos.limiteInferior=5
maxintentos.limiteSuperior=12
tamNumeroSecreto.valorActual=4
tamNumeroSecreto.limiteInferior=3
tamNumeroSecreto.limiteSuperior=8
```

Siguiendo con la metodología de separación de clases por su funcionalidad, puedes crear una clase llamada FileManager (o similar) que se encargue de estas operaciones.

15. Colecciones

15.1. Realizar un programa que contenga una *LinkedList* para almacenar las matrículas de los coches aparcados en un parking. El parking es un poco raro, mide solo 3 metros de ancho y caben 10 coches, pero uno detrás de otro por lo que el último en entrar debe ser el primero en salir *(esta estructura se llama pila LIFO – Last Input, First Output)*. El programa tendrá un menú para:

a. Aparcar: se le pasará el número de matrícula y lo almacenará a no ser que esté lleno.
b. Desaparcar: Muestra la matrícula del coche a desaparcar o bien un mensaje informando cadena vacía si el parking está vacío.
c. Mostrar la lista de las matrículas de los coches que hay en el parking, por orden inverso al de llegada, es decir primero el último en llegar

Nota: Usa los métodos que meten y sacan por el principio de la lista: *addFirst*, *removeFirst*

15.2. Haz una versión del ejercicio anterior, pero con los métodos que tiene *LinkedList* referidos específicamente a pilas (*peek*, *pool*,*pop*, *push*, etc.)

15.3. Haz un programa que cree un conjunto *HashSet* que almacene la lista de personas que van a una fiesta (de una persona sabemos su nombre, teléfono, email y fecha de nacimiento). Crea en un archivo aparte la clase Persona con los atributos y métodos que necesites. En el programa introduce "a mano" unas cuantas personas, y luego muestre la edad media y el nombre del mayor *(también puedes crear un menú con la opción de añadir persona y mostrar lista)*

- Intenta hacer una inserción de una persona repetida ¿Qué ocurre? *(Dos personas son iguales si tienen exactamente el mismo nombre).*
- Muestra todos los valores almacenados en el *HashSet* ¿Tienen algún orden?

15.4. Repite el ejercicio anterior con un *LinkedHashSet* ¿Qué ha cambiado?

15.5. Modifica la clase Persona (llámale Persona_v2) para que dos personas sean iguales si tienen el mismo nombre sin tener en cuenta mayúsculas y minúsculas. Repite el ejercicio anterior intentando incluir en el conjunto valores que no sean permitidos por esta nueva condición (por ejemplo: Ana y ANA generarían un duplicado).

15.6. Realizar un programa que tenga un *HashMap* que almacene una plantilla de jugadores de un equipo (Nombre, salario). El programa dispondrá de un menú que permita: añadir jugador, eliminar jugador, consultar el salario de un jugador e incrementar el salario un 10% a un empleado.

15.7. Realizar un programa que tenga un *HashMap* que almacene una plantilla de jugadores de un equipo, cuya clave es el nombre de cada jugador. De cada uno de ellos, además del nombre, tendremos su salario y su edad. La plantilla tiene un máximo de 25 jugadores. El programa dispondrá de menú que permita: añadir jugador, eliminar jugador, lista jugadores con su salario y que tenga también opción que permita introducir un salario y muestre lo datos de los jugadores que tiene un salario parecido al introducido *(por "parecido" entendemos que el salario del jugador esté en un rango de 6000 euros arriba o abajo respecto al introducido)*.

15.8. Realizar un programa al que se le introduzca un año y genera un array con las temperaturas medias de una ciudad para todos los días de ese año (365 ó 366). La temperatura será un entero entre 10 y 30 grados. Almacena en un Mapa la distribución de temperaturas, es decir, para cada temperatura, cuantos días del año la hubo y muestra dicha distribución.

15.9. Realizar un programa que genera un array con las temperaturas medias de una ciudad para todos los días de un año no bisiesto. La temperatura será un entero entre 10 y 30 grados. Ayudándote de un *TreeMap*, muestra la temperatura mínima y cuantas ocurrencias tuvo y la temperatura máxima y cuantas ocurrencias tuvo.

15.10. (Opcional) El siguiente programa simula una cola de la caja de una tienda para atender a sus clientes. Ejecútalo varias veces cambiando los parámetros de tiempo hasta que entiendas bien su funcionamiento. El cajero tarda entre 2 y 10 minutos en atender a cada cliente. Los clientes llegan a la cola entre 3 y 7 minutos.

```
import java.util.LinkedList;
import java.util.Random;
import java.util.Scanner;

public class dam000t15e10 {
    static Scanner teclado;
    static Random random;
    static final int HORAS = 8;
    static LinkedList <Integer> cola;
    static int instanteSiguCliente;
    static int t;

    //la cola contiene los minutos que va a requerir la atención del cajero
    // cuando le toque. Al primero de la cola se le va reduciendo en cada instante.
    public static void main(String[] args) {
        teclado = new Scanner(System.in);
        random = new Random();
        cola = new LinkedList<>();
        instanteSiguCliente = random.nextInt(5) + 3;

        //un bucle simula los minutos que van pasando
        for (t=1; t<=HORAS*60;t++){
            if (!cola.isEmpty()) {
                //reducir el tiempo de atencion del primero de la lista
                //que es el que está siendo atendido
                cola.set(0,cola.getFirst()-1);
                //si ese tiempo es cero, acabó de atenderlo. Sale de la cola
                if (cola.getFirst()==0) cola.removeFirst();
            }
            //se añade un nuevo cliente a a la lista (instante aleatorio)
            if (t == instanteSiguCliente) {
               cola.addLast(random.nextInt(9)+2);
               //se genera el instante de llegada del próximo cliente
               instanteSiguCliente += random.nextInt(5) + 3;
            }
            mostrarCola();
        }
    }
    static void mostrarCola(){
        System.out.print("Minuto: " + t);
        System.out.println("(Sig.cli llegará en minuto:" + instanteSiguCliente +")");
        for (int i=0;i<cola.size();i++) {
            System.out.printf("%02d",cola.get(i));
            if (i==0) System.out.println(" <<< Atendiendo (tiempo para terminar)");
            else
              System.out.println(" <<< Esperando (tiempo en caja cuando le toque)");
        }
        System.out.println("---------------\nPulsa");
        teclado.nextLine();
    }
}
```

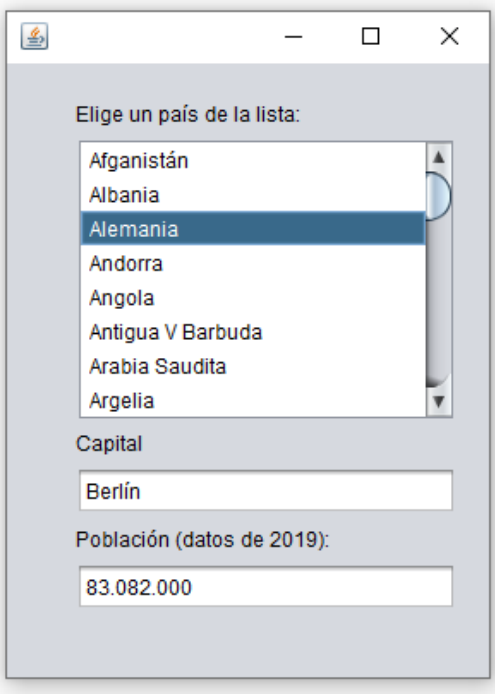
15.11. (Opcional) Modifica el programa anterior sabiendo que el cajero tarde entre 3 y 12 minutos en atender a un cliente y además, si un cliente tiene 5 o más clientes delante de él en la cola (incluido el que están atendiendo), se marchará sin comprar nada.

15.12. (Opcional) Modifica el programa anterior para que no muestre el estado de la caja cada instante si no que haga la simulación completa de las 8 horas y finalmente muestre cuantos clientes fueron atendidos y cuantos se marcharon sin comprar nada.

15.13. (Opcional) Modifica el programa anterior añadiendo una segunda caja, es decir tener dos colas (el cliente se pondrá siempre en la cola que menos gente tenga). Nos interesa mostrar si ya no perdemos ningún cliente o al menos es una cifra pequeña.

Nota: Este tipo de programas se utilizan para optimizar procesos llamados "teorías de colas" y conocer el impacto de posibles cambios: tardar menos tiempo en atender al cliente, poner otro cajero, etc.

15.14. Crea una aplicación gráfica con Swing, con una colección que se cargue desde el archivo *paises.csv* (contiene una línea por país, con nombre de país, capital y población, separados por punto y coma). El programa tendrá una lista desplegada con los países, ordenada alfabéticamente. Cuando se elija un país, en una caja de texto se mostrará su capital y en otra su población. *Elige la colección más adecuada pensando que la interfaz con el usuario podría ser otra (consola, web...), que queremos que la lista de países esté ordenada y que el texto de capital y población se actualiza a partir del nombre seleccionado en la lista.*



15.15. (Opcional) Haz una versión del ejercicio 1 pero muestre los coches por orden de llegada. Recordemos que en ese ejercicio se añadían los elementos en la lista por el principio, en la primera posición. Por lo tanto, al mostrar el ArrayList con un *for...each* los mostraba por el orden inverso al de llegada. Ahora queremos lo contrario, que muestre la lista de final a principio. Pista: iterador descendente.

15.16. Haz un programa con un *TreeSet* de Persona_v2 (es un conjunto ordenado), que introduzca "a mano" unas cuantas personas en el conjunto y a continuación las muestre con un *for...each*. ¿Qué ocurre? Crea una nueva versión de Persona_v3 que resuelva los problemas encontrados y que ordene y no permita duplicados por nombre sin distinguir mayúsculas de minúsculas. Si eliminases de la clase Persona_v3 los métodos equals() y hashCode(). ¿Seguiría funcionando correctamente?

15.17. Realiza un programa que defina un *Arraylist* de Persona_v3. Añade 5 personas distintas al mismo y luego muéstralas por pantalla ordenadas por email. A continuación, vuelve a mostrarlas, pero esta vez ordenadas por nombre. ¿Debes usar un Comparator en ambos casos?

15.18. Realiza un programa que defina un *Arraylist* de Persona_v3. Añade 5 personas al mismo, pero una de ellas que esté repetida. Utilizando conversiones entre colecciones (sin iterar sobre el ArrayList) elimina los repetidos, preservando el orden de los elementos del ArrayList.

15.19. Realiza un programa que defina una variable de tipo interfaz *Set* de enteros instanciada con *HashSet.* Crea un bucle de cien mil de iteraciones que añada al conjunto números aleatorios entre 1 y un millón. Con *LocalDateTime* y *ChronoUnit.MILLIS.between* mide los milisegundos que le lleva ejecutar el bucle. A continuación, sobre la misma variable de tipo interfaz, instánciala con *TreeSet* y realiza el mismo proceso. Mide de nuevo el tiempo que le lleva con la nueva clase. Compara los resultados y razona a qué es debido.

15.20. En el tema 7 hicimos varios ejercicios sobre el sorteo de la Primitiva (con array y arraylist). Ahora que conoces estas nuevas colecciones. ¿Cuál elegirías para obtener los números premiados de un sorteo de Primitiva? Recuerda que eran 6 números aleatorios sin repetidos y ordenados ascendentemente. Haz el código para ver lo sencillo que sería obtener esos números en comparación a realizarlo mediante ArrayList.

Proyecto:

Sobre el proyecto anterior, realiza una pequeña modificación:

Ya que los intentos no pueden repetirse, sustituye en la clase Juego la interfaz List de intentos por Set. Esto cambia la forma de comprobar si el intento es repetido y por lo tanto inválido ya que al hacer *add()* lo comprueba automáticamente-.

¿Qué tipo de Set es el más adecuado?

16. Orientación a Objetos Avanzada

16.1. Un punto en el espacio bidimensional se define por sus coordenadas X e Y que pueden incluir decimales. Realiza un programa que solicite al usuario las coordenadas de dos puntos y que calcule y muestre su distancia. Emplea *records* para almacenar los datos de cada punto.

16.2. Queremos realizar una aplicación que gestione nuestras tarjetas de crédito. De cada tarjeta queremos saber su número (16 dígitos), el nombre del titular, fecha de caducidad y código CVV (3 dígitos). El programa tendrá un menú para añadir una nueva tarjeta, borrar tarjeta, listar todas las tarjetas no caducadas y listar tarjetas caducadas. Una vez que se crea una tarjeta ya no se puede modificar ninguno de sus atributos. ¿Tiene sentido emplear *record* en vez de clases?

16.3. Realizar, a partir de los atributos estáticos de las wrapper class, un programa que muestre el valor máximo y mínimo de los Integer y los Long y también cuantos bytes ocupan en memoria.

16.4. Realizar un programa al que se le introduzca un número entero. Mostrar dicho número en su representación hexadecimal. Evitar que se produzca una excepción si el usuario no introduce un valor correcto (por ejemplo, si introduce texto o un número con decimales o un número demasiado grande).

Pista: La clase wrapper Integer dispone de un método estático para convertir un número en un String con la representación hexadecimal de dicho número.

16.5. Escribe un programa que solicite al usuario que introduzca una cadena y verifique, mediante expresiones regulares, si la cadena se corresponde con:

a) Una matrícula de coche (4 dígitos y 3 letras mayúsculas).
b) Un número binario de una o más posiciones.
c) Un número hexadecimal de entre 5 y 8 posiciones.
d) Una fecha en formato YYYY-MM-DD (solo formato, no valida días del mes).
e) Como el anterior, pero admite también que el día o el mes esté en un solo dígito.
f) Un nombre de usuario en twitter, empieza por @ y puede contener letras mayúsculas y minúsculas, números, guiones y guiones bajos (entre 2 y 15 caracteres)

16.6. Escribe un programa que solicite al usuario que introduzca un email y que extraiga el nombre del usuario (lo que hay antes de la arroba) y el TLD (el dominio de nivel superior, lo que va después del punto).

16.7. Crea una clase de tipo genérica llamada *ListaPequeña* que tendrá como atributos privados un ArrayList de tipo genérico llamado *valores* y un entero (qué será *final*, piensa el porqué) con el tamaño de la lista, llamado *tamañoMax.*

- Al constructor de esta clase le pasaremos un entero positivo que se asignará a *tamañoMax*. El constructor instanciará también el ArrayList *valores*.
- Tendrá un método llamado *añadir* que se le pasa un objeto de tipo genérico y si el arraylist aún tiene un tamaño menor que *tamañoMax,* lo añadirá, pero si el arraylist ha llegado a ese tamaño, añadirá objeto pasado en la última posición del ArrayList, sustituyendo el elemento que allí hubiera (así conseguimos que el ArrayList no crezca por encima del tamaño máximo)
- Tiene un método llamado *clear(),* que simplemente hace un clear() del ArrayList *valores.*
- Tiene un método llamado *getValores()* que devuelve el ArrayList privado *valores.*

Haz un programa que cree una ListaPequeña de tipo String de 5 elementos. Añádele 6 ó 7 String a la lista y muestra su contenido, para ver si ha crecido por encima de esos 5 elementos.

Crea otro programa similar al anterior, pero en vez de trabajar con String trabaje con una clase que hayas creado previamente, por ejemplo: *Persona_v3.*

Una forma de abordar los ejercicios de genéricos es hacerlo como si no fuesen genéricos, por ejemplo, pensando que son String, y luego hacer el paso a genéricos. En este caso podríamos pensar en una lista pequeña de String y luego hacer la "generalización" sustituyendo todos los sitios donde aparezca String por "<T>".

16.8. Crea una clase de tipo genérica (cualquier tipo que herede de *Number*) que describa un cuadrado.

- Tiene un único atributo llamado "lado" de tipo genérico.
- Tiene un constructor al que se le pasa también un valor genérico y lo asigna a *lado*.
- Tiene un método que calcule el área del cuadrado, multiplicando *lado.doubleValue()* * *lado.doubleValue()* y devolviendo un tipo de dato *Number.*

Haz un programa que instancie cuadrados de tipo Integer, Long y Double de distintos tamaños y muestra el área de todos ellos.

16.9. Define una interfaz llamada *Operable*, que sea genérica de tipo 'T'. Tiene que declarar los métodos llamados: *suma* y *resta.* Ambos métodos recibirán dos parámetros de tipo genérico 'T' y devolverán un genérico del mismo tipo.

- Crea una clase (no genérica) llamada *OperadorEntero* que implemente la interfaz anterior con Integer (ya sabemos cómo es la suma y resta de enteros)
- Crea una clase (no genérica) llamada *OperadorString* que implemente la interfaz anterior con Strings. La suma de Strings es la concatenación de la primera cadena pasada como parámetro con la segunda. La resta de String será eliminar todas las ocurrencias de la cadena segundo operando que se encuentren en la cadena primer operando, por ejemplo, *resta ("AbcDDxxDD","DD") devolvería "Abcxx".*
- Finalmente, haz un programa que instancie ambas clases creadas y opere con ellas.

16.10. (Opcional) Diseña una clase llamada *NotasAlumnos*, que contenga un ArrayList de Float con las notas de los alumnos de una escuela y además unos atributos que almacenen el valor mínimo permitido, valor máximo permitido y otro atributo de tipo *char* llamado nivel. Los dos primeros se determinan en el constructor de la clase y el tercero se actualiza desde una clase interna llamada *NivelEscuela.* Esta clase interna tiene un método llamado *calcularNivel()* que, en función de una serie de parámetros, calcula el nivel de la escuela. Actualmente se calcula así:

- *LimiteAprobado* es un atributo de la clase interna que se define en el constructor de la misma como la media aritmética entre el valor máximo y mínimo permitido. Será el valor que defina si una nota es un aprobado o no.
- Una escuela es de nivel 'A' si aprueba más del 80% de la escuela.
- Una escuela es de nivel 'C' si suspenden más del 50% de la escuela.
- Una escuela es de nivel 'B' en cualquier otro caso.

Finalmente, la clase *NotasAlumnos* dispone de un método llamado *añadirNota* () al que se le pasa como parámetro una nota y, si está entre los valores mínimo y máximo permitido, la añade al ArrayList.

Haz un programa que cree una instancia de la clase *NotasAlumnos* , le añada unos valores cualquiera a las notas de los alumnos y calcule y muestre el nivel de la escuela.

16.11. (Opcional) Contesta a las siguientes preguntas referentes al ejercicio anterior *(puedes crear un archivo .txt con las respuestas con la misma nomenclatura que el resto de ejercicios)*:

- ¿Se podría resolver sin una clase interna?
- ¿Qué ventaja representa el tener los cálculos de nivel en una clase distinta a *NotasAlumnos*?
- ¿Podría ser la clase *NivelClase* una clase externa?

16.12. Haz una nueva versión del programa 17 del capítulo anterior (Implementación de Comparator) empleando una clase anónima en *Collections.sort ()* para ordenar por nombre la lista de Persona_v3.

16.13. (Opcional) Supón que vas a implementar un juego de mesa online basado en un tablero de 20 x 20 casillas en el que pueden jugar muchos jugadores a la vez.

- Habrá una clase *Tablero* que tendrá un array de enteros de 20 x 20 donde mantendrá en cada casilla el *id* del jugador que hay en esa casilla en ese momento (o bien cero, si no hay ningún jugador en esa casilla).
- Habrá una clase *Jugador* que mantendrá para cada jugador: su *id* y la fila y columna del tablero en la que esté. Al empezar la fila y la columna se generan al azar entre las casillas no ocupadas del tablero. El *id* es un número correlativo que identifica a cada jugador en el tablero y por tanto e en la partida. Para el primer jugador de la partida es 1 y se va autoincrementando cada vez que se crea un nuevo jugador.

Implementa las clases *Jugador* y *Tablero*. Como no sabemos cómo se programará el juego online (no sabemos cuándo empieza y acaba una partida), haremos la clase *Tablero* con el patrón ***Singleton*** de forma que, en el alta de un jugador, si no existe tablero (porque es el primer jugador de la partida) se cree el tablero y para los siguientes jugadores de la partida usen ese mismo tablero.

No sabemos el resto de reglas del juego por ahora. Solo haremos un programa que permita dar de alta 10 jugadores y nos muestre para cada uno de ellos su *id* y posición en el tablero.

Ojo: Recuerda que en el programa no creamos ninguna instancia del tablero, el tablero se crea cuando se crea el primer jugador.

16.14. (Opcional) Crear "a mano" un fichero de tipo *properties* que mantenga el nombre del sitio web de una empresa, la IP del mismo y el puerto de conexión. Realizar un programa que lea ese fichero y compruebe mediante expresiones regulares que la IP tiene 4 bloques de entre 1 y 3 dígitos separados por puntos. Luego extrae de la expresión regular esos 4 bloques y verifica que todos tienen un valor menor que 256. Finalmente crea una clase **inmutable** llamada *WebSite* con los valores leídos.

```
1  # Fichero de configuración de la web.
2  dominio = www.miweb.com
3  ip = 100.200.22.0
4  puerto = 80
5
```

Proyecto:

En este tema, no hay que hacer ningún cambio en el proyecto.

De manera opcional, puedes investigar que es Javadoc y documentar las clases con esta herramienta.

17. Tratamiento XML

Crea una carpeta llamada 'archivos' en la carpeta raíz del proyecto Netbeans y copia el archivo XML 'librería.xml' proporcionado por el profesor. (El archivo tiene codificación UTF-8).

17.1. Realiza un programa que muestre el nombre del documento raíz del archivo 'librería.xml'.

17.2. Realiza un programa que muestre el título de todos los libros presentes 'librería.xml'.

17.3. Realiza un programa que muestre el título de todos los libros presentes en 'librería.xml' con su precio. *(Todos los libros tienen precio).*

17.4. Realiza un programa que muestre el título de todos los libros presentes en 'librería.xml' con su temática. *(Puede que algún libro no tenga temática, en ese caso mostrará "Temática desconocida").*

17.5. Realiza un programa que muestre el título de todos los libros presentes en 'librería.xml' con su precio de aquellos libros que cuesten menos de 10 euros.

17.6. Realiza un programa que muestre el nombre de todos los libros y su autor o autores.

17.7. Realiza un programa que muestre el nombre de todos los libros con su alto, ancho y número de páginas. *(Algunos puede que no tengan toda o parte de esa información, mostrar una interrogación en sus valores, por ejemplo: "El perfume → Dimensiones ? cm x ? cm. ? páginas).*

17.8. Diseña una clase llamada Libro, que sea capaz de mantener para un libro su ISBN, nombre, precio y autores. Crea un programa que cargue en un ArrayList de 'Libro' la información correspondiente que viene en el archivo 'libreria.xml'. A continuación, ordenará el ArrayList por título de libro y mostrará por pantalla el contenido completo de ese ArrayList.

Pista: Implementar en la clase Libro el método toString() para facilitar el programa.

17.9. (Opcional) Haz una función llamada *buscarAtributosEnHijos ()* a la que se le pase como parámetro un *Element* y una cadena. La función buscará si ese *Element* tiene algún hijo con un atributo con nombre igual a la cadena pasada. En caso afirmativo devolverá el valor de ese atributo y en caso negativo devolverá la cadena vacía " ". Por ejemplo, si le pasásemos el primer libro del archivo 'librería.xml' *(es el que tiene por título 'Follas novas')*, y el atributo '*paginas*' devolvería *"496"*. La explicación es que ese *Element,* entre todos sus hijos, tiene uno con un atributo llamado '*paginas*' con valor *496*.

Luego haz un programa que muestre todos los libros de 'librería.xml' con el número de páginas, que obtenga ese número de páginas con la función realizada.

17.10. Modifica el archivo XML 'librería.xml' pasando el precio a dólares (1 dólar= 0,91eur). Guárdalo con el nombre 'libreriaDolar.xml'

17.11. Modifica el archivo XML 'librería.xml' pasando el alto y el ancho a pulgadas. Guárdalo con el nombre 'libreriaPulgadas.xml'

17.12. Modifica el archivo XML 'librería.xml' añadiendo a cada libro una etiqueta *<editorial>*. Al usuario se le dirá el título del libro y él introducirá el nombre de la editorial. Guarda el archivo con el nombre 'libreriaConEditorial.xml'

17.13. Modifica el archivo XML 'librería.xml' eliminando las etiquetas 'isbn', 'dimensiones', 'tematica' y 'precio'. Guárdalo con el nombre 'libreriaResumen.xml'.

17.14. Crea un archivo XML desde cero que solo contenga un elemento raíz llamado `<agenda>` y elementos hijo de tipo texto como los que muestra la siguiente figura. No hace falta que el usuario introduzca los valores, puedes meterlos en el código "a mano".

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<agenda>
    <contacto>Pepe Perez</contacto>
    <contacto>Ana López</contacto>
</agenda>
```

Proyecto:

En este tema, no hay que hacer ningún cambio en el proyecto.

De manera opcional, puedes investigar que es JUnit y realizar algún test unitario sobre los métodos de la clase Juego.

18. Acceso a Base de Datos

Instala XAMPP y mediante phpmyadmin crea una base de datos llamada 'empresa'. A continuación, ejecuta el script cargaEmpresa.sql proporcionado por el profesor para crear la tabla 'Empleado' e llenarla con datos. Verifica la carga y observa la estructura de la tabla

18.1. Realizar un programa que se conecte a la base de datos 'empresa' y muestre el nombre de los empleados de la empresa.

18.2. Realizar un programa que solicite al usuario una fecha y obtenga de la base de datos los empleados nacidos después de esa fecha y muestre su nombre y edad.

Pista: hay que convertir de LocalDate a java.sql.Date.

18.3. Realizar un programa que solicite al usuario una categoría laboral, y que muestre la cantidad de empleados que hay con esa categoría.

Pista: función agregada en SQL.

18.4. Crear una clase llamada Empleado cuyos atributos privados son los campos de la tabla 'Empleado', añádele un constructor con todos los campos, getters, setters y método *toString()*. Realizar un programa que cargue un ArrayList de clase Empleado con todos los empleados de la tabla. Mostrar a continuación todo el contenido del ArrayList.

Pista: Implementa el método toString() en la clase Empleado .

18.5. Realizar un programa que solicite al usuario todos los datos de un empleado y que inserte una nueva fila en la tabla Empleado. Después de introducir un empleado, preguntará si se desea insertar más, repitiendo el proceso mientras el usuario lo desee.

18.6. Realizar una versión del programa anterior en la que el usuario no introduce el 'id'. El sistema la calcula como el máximo almacenado en la tabla + 1. Pista: Habrá dos PreparedStatement, uno para obtener el max (i) y otro para el insert.

18.7. Realizar un programa que actualice el salario de los empleados para que no haya ningún empleado que cobre menos de 1000 euros. (Hacerlo mediante UPDATE). El programa informará de cuantos empleados se han visto afectados.

18.8. Partiendo de la clase Empleado definida previamente, realiza un programa con un main que tenga dos métodos o funciones: el primer método leerá el fichero csv llamado *empleados.csv* proporcionado por el profesor y que contiene una línea por cada empleado con la siguiente estructura: *id ; nombre ; fechaNacimiento ; categoria ; salario*

Para cada línea del fichero se deberá crear una instancia de empleado e introducirla en un *hashSet* de empleados. El segundo método del main leerá el hashset anterior y, para cada empleado que cobre más de 2000 euros, lo insertará en la tabla de empleados.

Pistas:

- El hashset será una variable global para poder acceder a ella desde ambos métodos.
- Para insertar en la tabla puedes basarte en el ejercicio 5, pero recorriendo el HashSet en vez pedirle a usuario los datos de los empleados.

18.9. Realizar un programa que elimine de la tabla a los mayores de 60 años. (Hacerlo mediante DELETE). El programa informará de cuantos empleados se han visto afectados.

18.10. Realizar un programa que actualice el salario de los empleados para que no haya ningún empleado que cobre menos de 1100 euros. Hacerlo actualizando un *ResultSet*. A continuación, volver al principio ese *ResultSet* y mostrar nombre y salario de todos los empleados.

18.11. Partiendo de la clase Empleado definida previamente, sigue el patrón *Repository* para crear una clase con las operaciones de:

- Buscar un empleado: se le pasará 'id' y devolverá un objeto Empleado.
- Borrar empleado: se le pasará 'id' y devolverá *boolean (* encontrado y borrado o no).
- Insertar empleado: se le pasarán todos los datos y y devolverá *boolean* según lo haya encontrado e insertado o no.
- Listar empleados: devuelve un ArrayList de Empleados, con todos los registros de la tabla.

Finalmente, hacer un programa con un menú, que permita: buscar, borrar, insertar y listar, empleando la clase anterior.

Puedes hacer dos versiones de este ejercicio, una utilizando una conexión única creada en el constructor mediante el patrón Singleton y otra versión en la que para cada operación se establece una nueva conexión.

18.12. Repite el primer ejercicio de este bloque sobre la base de datos SQLite *empresa.db* proporcionada por el profesor. Obviamente, no hará falta el MySQL para este ejercicio.

Proyecto:

Sobre el proyecto anterior, queremos guardar en base de datos información sobre todas las partidas jugadas para poder ser utilizada en un futuro para distintas tareas: estadísticas, entrenar una IA, etc.

Para ello, al finalizar una partida, haya ganado o haya perdido, se guardarán todos los datos básicos de ese juego junto con la fecha/hora del mismo, serían: número secreto, límite de intentos, resultado final (ganó o perdió) y la lista de intentos realizados.

No es necesario guardar el atributo con el tamaño del número ya que se puede obtener como la longitud del número secreto. De cada intento solo guardaremos el número, ya que la cantidad de bien colocados y mal colocados del intento se puede calcular fácilmente.

Siendo estrictos, deberíamos tener 2 tablas ya que hay una relación "uno a n" entre la partida y los intentos, pero para hacerlo más simple, puedes guardar todos los datos en una única tabla, con una fila por partida. Los intentos los puedes guardar todos en un único atributo, todos juntos, separados por un punto y coma (como se hace en los archivos csv).

En este capítulo se cierra el proyecto.

Consejos técnicos:

- Usa SQLite en vez de otro gestor de BD. Será más simple.
- Crea una clase con la estructura exacta de la tabla y una clase DAO sobre esa clase. En este caso, solo te hace falta el método para guardar. Puedes hacer un método para convertir la instancia de la clase 'Juego' con sus intentos en esa clase que empleará el DAO.

19. Programación Funcional

19.1. Definir una interfaz funcional llamada *IImpuesto* con un único método abstracto llamado *aplicar()* al que se le pasan dos parámetros, un importe *(double)* y un impuesto *(float)* y devuelve un *double.* A continuación, crear una clase llamada *Iva* que implemente dicha interfaz de forma que el cálculo sea *importe + importe * impuesto/100.*

Finalmente, hacer un programa que cree una instancia de la clase anterior sobre una variable de tipo *IImpuesto* y que muestre por pantalla el resultado de aplicarle un Iva del 10% a un importe de 1000 euros.

19.2. A partir de la interfaz anterior, realizar un programa que cree una clase anónima sobre una variable de tipo *IImpuesto.* La clase anónima implementará el método *aplicar()* como *importe = importe + impuesto (sería como una tasa fija, por ejemplo 1000 eur + tasa 125 eur = 1125).*

El programa mostrará el resultado de aplicarle una tasa de 10 eur a un importe de 50 eur.

19.3. Realizar una nueva versión del primer ejercicio, sustituyendo la clase definida por una expresión Lambda.

19.4. Realizar una nueva versión del segundo ejercicio, sustituyendo la clase anónima por una expresión Lambda.

19.5. Realizar un programa que cree una clase anónima que implemente la interfaz *Predicate* verificando si una cadena está toda en mayúsculas o no. Prueba la implementación mediante su método *test()* para un caso que se cumpla y otro que no.

19.6. Realizar una nueva versión del ejercicio anterior, sustituyendo la clase anónima por una expresión Lambda.

19.7. Crear una clase llamada *Libro* con atributos privados *String titulo, float precio e int meGusta*, con su constructor, toString (), getters y setters. El atributo *meGusta* se inicia en 0 y su setter solo permite incrementos de uno en uno.

Crear un programa con un ArrayList de libros y realiza las siguientes operaciones (no hace falta un menú de interacción con el usuario, puedes hacerlas "a mano").

- Añadir varios libros.
- Ordenar la lista por precio mediante referencia a métodos.
- Mostrar el contenido de ArrayList mediante referencia a métodos.

19.8. A partir el ArrayList del ejercicio anterior y mediante las funciones de API Stream realizar las siguientes operaciones:

a) Mostrar los libros de más de 10 euros.
b) Mostrar los títulos de los libros de más de 10 euros.
c) Crear un conjunto Set llamado *favoritos* con los libros que tienen más de 2 "*me gusta*".
d) Contar los libros que hay con cero "*me gusta*".
e) Obtener la cantidad de *"me gusta"* de todos los libros en total.
f) Crear una lista igual a la inicial, pero con los precios en dólares (1 dólar = 0,92 eur).
g) Mostrar el libro con más *"me gusta"*.

19.9. Crea un fichero de texto con varias líneas y haz un programa que, mediante API Stream, muestre por pantalla aquellas líneas que empiezan por "A".

Práctica Final Temas 1-7

Juego de la Oca

Objetivo

Realizar un programa orientado a objetos que permita jugar al Juego de la Oca. Podrán jugar hasta un máximo de 4 jugadores. Cada jugador parte de la casilla 1 y debe llegar a la casilla tirando un dado de forma consecutiva un jugador detrás de otro. El juego finaliza cuando uno de los jugadores llega a la casilla final.

Casillas con características especiales:

Oca: Casillas 5, 9, 14, 18, 23, 27, 32, 36, 41, 45, 50, 54 y 59. Si se cae en una de estas casillas, se avanza hasta la siguiente casilla en la que hay una oca y volver a tirar. Si se cae en la 59 ya no hay a donde saltar pero se vuelve a tirar.
Puente: Casilla 6 y 12. Si se cae en estas casillas se salta a la casilla 19 (la Posada) y se pierde un turno.
Posada: Casilla 19. Si se cae en esta casilla se pierde un turno.
Laberinto: Casilla 42. Si se cae en esta casilla, se está obligado a retroceder a la casilla 30.
Cárcel: Casilla 56. Si se cae en esta casilla, hay que permanecer dos turnos sin jugar.
Calavera: Casilla 58. Si se cae en esta casilla, hay que volver a la Casilla 1.
Ganador: Es necesario llegar a la 63 con los puntos justos, en caso de exceso se rebota hacia atrás.

Desarrollo del juego

- Primero se solicitará el número de jugadores (entre 2 y 4).
- A continuación es una sucesión de turnos entre los jugadores, teniendo en cuenta si hay jugadores que están "sancionados" sin tirar, hasta que uno de ellos llegue a la meta.
- En cada tirada, se indicará el jugador y, o bien se tirará el dado si no está sancionado indicándole la casilla a la que llega, o bien se le indica que está sancionado y no puede tirar aún.

Se entregará

- - Proyecto Netbeans exportado a zip, con código fuente.
- - PDF con instrucciones de uso.
- - PDF con un resumen de las mayores dificultades para desarrollar el proyecto. Como están estructuradas las clases, y cómo se podría ampliar en un futuro.

Pistas

- Habrá que almacenar para cada casilla: su nombre, la casilla a la que "salta" (al menos para las especiales) y también los turnos que le quita al jugador que caiga.
- Para cada jugador, habrá que mantener los turnos que tiene de sanción para que, si tiene alguno, cada vez que le pase el turno, se vaya reduciendo hasta poder volver a jugar.
- Probablemente la clase *TableroOca* debería tener un Array/ArrayList de *Casillas* y un Array/ArrayList de *Jugadores*.

Práctica Final Temas 8-14

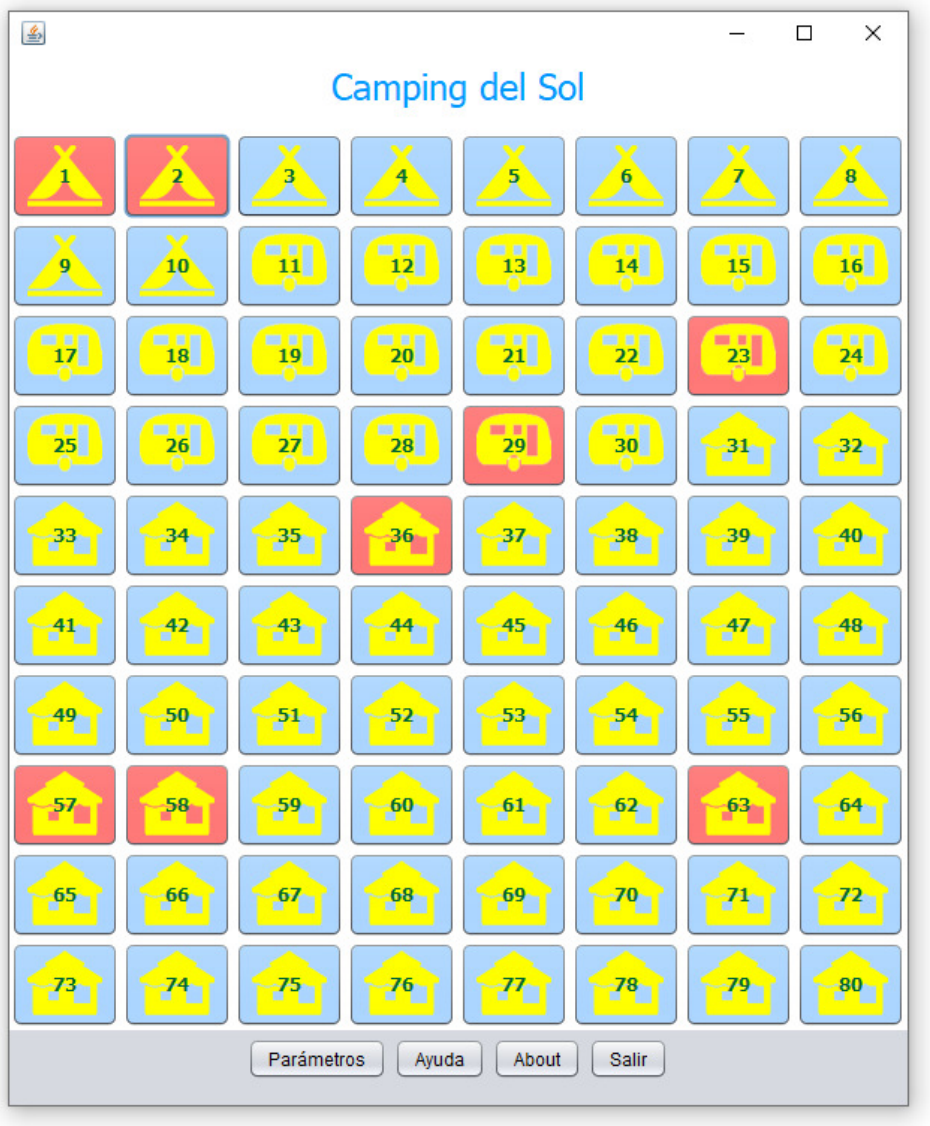
Gestión de un camping

Objetivo

Realizar un programa que, mediante una interfaz gráfica, permita gestionar las entradas y salidas en las parcelas de un camping calculando los importes a pagar por los campistas según el tipo de alojamiento y los días de estancia.

Requisitos

El programa presentará de forma gráfica un "mapa" de las parcelas (pueden ser botones) en el que se verá cuales están libres u ocupadas por su color de fondo (por ejemplo, azul: libre y rojo: ocupada) y se identificarán con distintos iconos los distintos tipos de parcela.



Dispondrá de una opción de "Check-in", clicando sobre una parcela 'desocupada' solicitando mediante un cuadro de diálogo obligatoriamente el DNI del huésped principal. A continuación, puede solicitar algún otro dato de acuerdo al tipo de parcela, según el cuadro que se muestra a continuación.

Dispondrá de una opción de "Check-out", clicando sobre una parcela 'ocupada' para realizar la salida de sus ocupantes, calculando y mostrando el importe a pagar por los mismos.

También se trabajará con ficheros, como se describirá en un apartado posterior.

Tipos de parcelas y precio:

Tipo de parcela	Cálculo de importe por día	Observaciones
Tiendas de campaña	20 eur por tienda 1 eur por electricidad	10% descuento para estancias de más de 7 días.
Caravana/Autocaravana	40 eur por día, si día de salida en agosto. 30 eur por día, resto de meses.	Mínimo 10 días. No se permite checkout si menos días.
Bungalow	20 eur por adulto. Los niños no pagan.	Si solo 1 ó 2 noches, paga 20% recargo.

Ficheros:

- Los parámetros del camping que se ven en la tabla anterior estarán guardados como variables en una clase llamada *Param.* Estos valores se tomarán de un fichero gestionado por la clase *Properties* y se cargarán esa clase llamada *Param* al principio del programa y cuando haya modificaciones.

- El botón [Parámetros] de la pantalla principal indicará al usuario mediante un diálogo que debe modificar ("a mano") el fichero comentado en el punto anterior y se actualizará la clase *Param* cuando cierre dicho diálogo.

- Guardará en un fichero con formato .csv todo lo facturado, añadiendo una línea cada vez que se produce un checkout con: dni huésped principal, número de parcela, tipo de parcela, fecha de entrada, fecha de salida, importe pagado.

- Guardará en un fichero la situación actual del camping, de forma que cada día por la mañana (o por ejemplo si se apaga el ordenador inesperadamente), al arrancar la aplicación cargue la situación actual. (Si el camping se mantiene mediante un ArrayList de parcelas, será un fichero con el ArrayList serializado).

- No se puede modificar el número de parcelas, ni cuantas hay de cada tipo, serán unos valores constantes almacenados en la clase *Param*. (Por comodidad, podemos considerar que a nivel de colocación en el ArrayList que primero van todas las tiendas, luego las caravanas y luego los bungalows)

- Los archivos estarán en una carpeta llamada */data* creada en la raíz del proyecto (no en /src). En el entorno real de producción, esta carpeta estará en la misma ruta en el *.jar*.

Estructura del proyecto:

- Por una parte, tendrá las clases que mantienen la lógica del camping. Un posible modelo podría incluir:
  - Superclase (¿abstracta?) **Parcela** con subclases **Tienda**, **Caravana**, **Bungalow**. La clase Parcela debe implementar la interface *iAlquilable* que se muestra al final.
  - Clase **Camping** con el ArrayList polimórfico de Parcelas.
  - Clase **Param** de Parámetros (con importes y condiciones de las parcelas)
  - Clase **Ficheros** con métodos estáticos que se encarguen de todas las operaciones de lectura y escritura de ficheros:
    i. Guardar estado actual del camping.
    ii. Cargar estado actual del camping.
    iii. Guardar factura al hacer checkout.
    iv. Cargar parámetros de camping.

- Tendrá un *JFrame* con los paneles necesarios: al menos habrá uno de tipo *Grid* para presentar los botones que representan las parcelas. También tendrá otro para el título en la parte superior y otro para los botones que se ven en la parte inferior gráfico anterior. Esos botones contienen:
  - *Ayuda: JOptionPane* con descripción/manual del programa
  - *About: JOptionPane* con autor, email de contacto, fecha, versión, etc.
  - *Parámetros: JDialog* indicando el archivo de parámetros de debe modificar.
  - *Salir.*

- El proyecto tendrá una carpeta para los iconos de los botones y otra carpeta para los archivos. Se pueden mantener todas las clases en el mismo paquete ya que no son muchas, aunque una buena política sería tener un paquete para las clases que forman la lógica del programa, otro para la parte gráfica y otro para la gestión de ficheros.

- Para el cálculo del importe a pagar en el checkout se pueden tratar los segundos como días, para hacer pruebas.

Se aconseja hacer la aplicación en modo de prototipos sucesivos siguiendo los pasos que se mencionan a continuación, pero sin pasar de uno al siguiente hasta que se pruebe y funcione bien.

1. Hacer las clases necesarias.
2. Hacer un programa sencillo de consola con un menú para checkin, checkout y mostrar estado del camping. Usar valores constantes para cantidad de parcelas, precios, etc. También la introducción de datos se puede hacer "a mano", sin menús…
3. Hacer la parte gráfica. Crea un jFrame con sus paneles para la gestión del camping que realice las funciones de checkin y checkout similar al programa de consola anterior, también metiendo parámetros constantes para los valores del checkin.
4. Añadir ventanas de diálogo para pedir al usuario la información de ckeckin y mostrar la información de checkout.
5. Incorporar los ficheros solicitados: primero el de la gestión de los parámetros y luego el resto.
6. Finalmente pulir aspectos gráficos, control de excepciones, validación de datos de entrada, mensajes de ayuda, botón de Ayuda y About, etc.

A entregar: Se entregará un archivo .zip con el siguiente contenido.

1. Archivo .zip con la exportación del proyecto Netbeans.
2. Archivo .jar con el ejecutable
3. Carpeta de ficheros de datos (*properties*, *csv de facturas*, *fichero con estado del camping*).
4. PDF con el alcance del proyecto: esto es, que cosas de las solicitadas se implementaron, cuáles no, otros cambios realizados, etc. También se puede hacer un resumen de las mayores dificultades para desarrollar el proyecto. Como están estructuradas las clases, y cómo se podría ampliar en un futuro.

Nota: Si queréis distribuir el ejecutable al cliente sería necesario un .zip con el contenido del punto 2 y del punto 3. Si el proyecto es Maven hay que incluir la clase main en el archivo pom.xml y recordar que la ruta de las imágenes también depende de si el proyecto es Maven o Ant (ver apuntes)

Interface a implementar:

```
public interface iAlquilable {
    //checkin marca la parcela como ocupada
    boolean checkIn(String dniHuesped);

    //checkout marca la parcela como libre y
    //calcula el importe a pagar en función de los parámetros del camping
     double checkOut(Param param);
}
```

Práctica Final Temas 9-19

Juego de Emparejar items

Objetivo

Realizar un programa que, mediante una interfaz gráfica, represente un juego de emparejamiento de "ítems". Por defecto esos ítems serán textos y se le presentarán al usuario de dos columnas (por ejemplo: columna de países y columna de capitales, etc). El usuario deberá seleccionar las parejas correctas en un determinado tiempo.

El juego permitirá cargar los "ítems" desde un archivo XML de forma que las temáticas pueden ser variadas: idiomas (palabra en castellano vs. palabra en otro idioma), operaciones matemáticas, cualquier tipo de pregunta - respuesta, etc.

Requisitos

El proceso del juego será el siguiente:

1.- Solicitar al usuario su nombre (alias) y la selección de temática (habrá un fichero XML por cada temática que incluirá las parejas y otra información que se describe más abajo) El programa leerá todos los XML de la carpeta por defecto para estos archivos y mostrará las temáticas disponibles.

2.- Leer el archivo XML completo correspondiente a la temática seleccionadas y almacenar las parejas en un ArrayList además del resto de información del XML.

3.- Seleccionar al azar un número de parejas determinado por el XML para el juego, almacenarlas en un Map ordenado y presentarlas por pantalla junto con un cronómetro descendente con el tiempo restante. La presentación consistirá en dos listas a emparejar (dos columnas de botones, o como dos listbox, etc). Hay que tratar de hacerlo atractivo para el usuario (colorido, música y/o sonidos).

4.- El juego finaliza cuando resuelve todas las parejas o se le acaba el tiempo.

5.- Al finalizar guarda en una tabla de una base de datos SQLite el alias del usuario, la temática, fecha, cantidad de aciertos y tiempo en resolverlo y muestra el "top users" para esa temática (los 5 jugadores que acertaron todas en menor tiempo). Luego ofrecerá repetir partida con misma temática, seleccionar nueva temática o salir.

Estructura del archivo XML

```
<?xml version="1.0" encoding="UTF-8"?>
<juegoParejas>                        <|--nombre del juego(el programa) común para todos -->
   <name>Países y Capitales</name>    <!--temática-->
   <number>10</number>                <!--cantidad de preguntas de cada juego-->
   <time>8</time>                     <!--segundos para cada juego-->
<pair>
    <item1 type="text">España</item1>
    <item2 type="text">Madrid</item2>
</pair>
<pair>
    <item1 type="text">Portugal</item1>
    <item2 type="text">Lisboa</item2>
</pair>
</juegoParejas>
```

Consideraciones

- Separar acceso a datos, parte visual, lógica del juego en clases/paquetes separadas.
- Crear 3 archivos XML para distintos tipos de juego (por ejemplo, conceptos en español-inglés, películas y directores, etc.)

Mejoras opcionales

- Almacenar en un archivo de tipo *Properties* la configuración del juego: carpeta en la que se encuentran los archivos XML de temáticas, última temática jugada, último alias, etc.

- Permitir que los valores de las parejas (uno o los dos) puedan ser imágenes; podrían tener en el archivo xml el atributo *type="imagen"*. Ejemplo:

```
<pair>
    <item1 type="text">Portugal</item1>
    <item2 type="text">portu.jpg</item2>
</pair>
```

- Login de usuario con nombre y contraseña almacenado en la base de datos encriptado.