TRABAJO PRÁCTICO Nº3

Resolvé los siguientes ejercicios

1 Crear una clase llamada **Persona**, la cual estará compuesta de la siguiente manera:

- Atributos privados:
  - nombre (String),
  - apellido (String),
  - direccion (String).
- Métodos públicos:
  - *Persona()*, que es su *constructor*.
  - Los *setters* y *getters* de cada uno de sus atributos.
  - El método *toString()*.

El *constructor* Persona() se encargará de inicializar cada uno de los atributos de la clase con su valor por defecto; como en este caso todos son Strings los inicializará con “” (string vacío).

Cada uno de los *setters* recibirá como parámetro un valor con tipo de dato acorde al atributo con el que está relacionado y lo asignará al mismo. Los *setters* no devuelven nada, por lo que su tipo siempre es ***void***. Los *getters*, en cambio, devuelven el valor del atributo asociado, son del mismo tipo de dato que el atributo, y nunca reciben parámetros.

El método *toString()* debe implementarse tal como se muestra a continuación:

```
@Override
public String toString() {
	return "Persona [nombre=" + nombre + ", apellido=" + apellido + ",
direccion=" + direccion + "]";
}
```

Crear la clase **Test** con su función *main()*. En esta función se crearán dos instancias de **Persona** asignando valores a sus atributos mediante sus *setters*.

Finalmente deberás mostrar por pantalla los objetos completos utilizando *System.out.println(objeto)* y también cada uno de sus atributos por separado usando los *getters*.

Observación: Para cargar los objetos y luego mostrarlos debés implementar los métodos *completarDatos(...)*, que recibe una instancia de **Persona** y los valores a asignarle, y el método *mostrarPersona(...)*, que sólo recibe una instancia de **Persona**.

Pära los siguientes valores:

	persona1	**persona2**
nombre	Diego	Pablo
apellido	Díaz	Gómez
dirección	Yatay 240	Av. del Libertador 6796

… la salida del programa deberá ser:

```
Persona [nombre=Diego, apellido=Diaz, domicilio=Yatay 240]
nombre: Diego
apellido: Diaz
domicilio: Yatay 240
Persona [nombre=Pablo, apellido=Gomez, domicilio=Av. del Libertador 6796]
nombre: Pablo
apellido: Gomez
domicilio: Av. del Libertador 6796
```

2 Escribir la clase **Cuadrado**, la que en su único constructor recibirá la medida de un lado. Crear el setter y el getter del atributo lado y los métodos *perimetro()* (lado * 4) y *superficie()* (lado * lado). Por último implementar el método toString() que muestre no solo el tamaño del lado sino también sus campos calculados. Para eso escribiremos este método de la siguiente manera:

```
@Override
	public String toString() {
		return "Cuadrado [lado=" + lado + ", perimetro()=" +
perimetro() + ", superficie()=" + superficie() + "]";
	}
```

Escribir la clase Test donde se cree un cuadrado de 10 (diez) unidades de lado; luego cambiar su medida a 50 (cincuenta) unidades. Después de cada operación mostrar el estado del cuadrado. La ejecución debería mostrar los siguiente:

```
Cuadrado [lado=10, perimetro()=40, superficie()=100]
Cambio el tamaño del cuadrado a 50 por lado.
Cuadrado [lado=50, perimetro()=200, superficie()=2500]
```

3 Crear la clase **TarjetaDeCredito** con la siguiente estructura:

- Atributos privados:
  - numero (String),
  - titular (String),
  - limiteDeCompra (double),
  - acumuladoActual (double).
- Método:
  - *Constructor parametrizado y público* que reciba número, titular y límite de compra por parámetros y los asigne al atributo correspondiente. El atributo *acumuladoActual* se inicializará con 0 (cero).
  - Los *getters* de cada uno de sus atributos, públicos, y los *setters*, todos privados, menos el método *setAcumuladoActual()* que no existe.
  - El método *toString()* (público), el cual además de los atributos debe incluir el monto disponible para comprar.
  - El método público *montoDisponible()* que devuelve la diferencia entre el límite de compras y el acumulado actual de gastos, pero si por alguna razón este valor es inferior a cero devuelve cero. Por ejemplo, si gastaste determinado monto y luego cambiaron el límite a un valor menor a éste, el monto disponible debe ser 0 (cero).
  - El método privado *compraPosible()* que según el monto recibido por parámetro devuelve si se puede o no hacer la compra. Para saber si la compra es posible el monto de la misma no debe superar al monto disponible para compras.
  - El método público *actualizarLimite()*, que recibe un nuevo límite de compra.
  - El método privado *acumularGastoActual()*, que recibe el importe de la compra y lo suma al acumulado actual.
  - El método público *realizarCompra()*, el cual dado un monto comprueba si esta se puede realizar (si con la compra no se supera el límite), y si es posible la procesa actualizando los atributos que deba actualizar siempre usando los métodos que corresponda. Este método devuelve un booleano que indica si la compra se pudo realizar o no.

En la clase **Test**, se creará un objeto tarjeta con la siguiente información:

- Numero 4145414122221111
- Titular Juan Perez
- Limite 10000

Luego:

- Hacer una compra de $4000

- Mostrar el estado de la instancia (aprovechando el método *toString()*). Verás que el disponible debería ser de $6000.
- Bajar el límite a $3000.
- Intentar otra compra de $4000 (no debería poder).
- Volver a mostrar el estado de la clase; ahora el disponible debería ser $0.

4 La clase **Cafetera** cuenta con los siguientes atributos

- capacidadMaxima (entero, la cantidad máxima de café que puede contener la cafetera en mililitros),
- cantidadActual (entero, la cantidad actual de café que hay en la cafetera en mililitros).

Implementar, al menos, los siguientes métodos:

a. Constructor predeterminado o *por defecto*: establece la capacidad máxima en 1000 y la actual en cero (cafetera vacía).
b. Constructor con la capacidad máxima de la cafetera: inicializa la capacidad máxima con lo recibido y la cantidad actual en cero (vacía).
c. Constructor con la capacidad máxima y la cantidad actual. Si la cantidad actual es mayor que la capacidad máxima de la cafetera, la ajustará al máximo.
d. *Setters* privados y *getters* públicos. El *setter* de la capacidad nunca debe permitir un valor menor a 250 (si es menor lo fuerza a 250); la cantidad actual debe controlar que nunca sea menor a cero ni mayor a la capacidad de la cafetera.
e. *llenar()*: iguala la cantidad actual de la cafetera con la capacidad máxima.
f. *servirTaza(int)*: simula la acción de servir una taza con la capacidad indicada por parámetro. Si la cantidad actual de café no alcanza para llenar la taza, se sirve lo que haya.
g. *vaciar()*: setea la cantidad de café actual en cero.
h. *agregarCafe(int)*: añade a la cafetera la cantidad de café indicada, en el caso de ser posible. Devuelve la cantidad sobrante.

Realizar la clase **Test** para probar el correcto funcionamiento de todos los métodos de la clase previamente realizada. Crear una cafetera por defecto, otra con medio litro de capacidad y una tercera con tres cuartos litros de capacidad y una cantidad inicial de medio litro de café. Usar un único método *testearCafetera(Cafetera)* para probar las tres cafeteras por separado, una cada vez.

5 Crear la clase **Hotel** con los siguientes atributos:

- nombre (String)
- localidad (String)
- habitacionesTotales (int)
- habitacionesOcupadas (int)

- habitacionesReservadas (int)

La clase tendrá dos constructores parametrizados: el primero recibirá el nombre, la localidad y el total de habitaciones; el segundo será completo, para setear cada uno de sus atributos.

Además contará con los siguientes métodos públicos:

- *reservarHabitaciones()*: recibe por parámetro la cantidad de habitaciones a reservar. Valida que sea posible realizar por completo la acción y devuelve *true* o *false* según corresponda.
- *ocuparHabitaciones()*: recibe por parámetro la cantidad de habitaciones a ocupar y un booleano que indica si son de la reserva o no. Valida que sea posible realizar por completo la acción (comparando con las reservadas si son de la reserva y con las libres si no lo son) y devuelve *true* o *false* según corresponda. Para ocupar las habitaciones usará los métodos privados *ocuparConReserva()* y *ocuparSinReserva()*.
- *desocuparHabitaciones()*: recibe por parámetro la cantidad de habitaciones a desocupar. Valida que sea posible realizar la acción y devuelve *true* o *false* según corresponda.
- cantidadHabitacionesLibres(): devuelve la cantidad de habitaciones no ocupadas.
- *disponibilidad()*: devuelve la cantidad de habitaciones disponibles descontando las ocupadas y las reservadas.
- Los *getters()* de todos sus atributos, públicos, y los *setters()* privados, salvo el nombre que sí es público (se le puede cambiar el nombre). El hotel debe tener por lo menos una habitación, y las reservas no pueden ser negativas ni superar la cantidad de habitaciones del hotel (en todos los casos se ajustará al límite superado).
- El método toString() que mostrará todos los atributos más los miembros calculados *cantidadDeHabitacionesLibres()* y *disponibilidad()*. Esta vez modificaremos el formato de salida y escribiremos lo siguiente:

```
@Override
public String toString() {
	return String.format("Hotel %s %s [habitaciones=%d, ocupadas=%d,
reservadas=%d, libres=%s, disponibles=%d]", nombre, localidad,
totalHabitaciones, habitacionesOcupadas, habitacionesReservadas,
cantidadHabitacionesLibres(), disponibilidad());
}
```

En la clase **Test** crear una instancia de **Hotel** con los valores "Hilton", "CABA", 200. Luego realizar las siguientes operaciones mostrando el resultado de la operación y mostrando al comienzo y luego de cada una el estado del hotel:

1. Ocupar 201 habitaciones sin reserva (no debe dejarnos)
2. Ocupar 30 habitaciones sin reserva. Debe dejarnos y quedar 170 disponibles y libres.
3. Reservar 50 habitaciones (deben dejarnos y quedar 120 disponibles y 170 libres).
4. Liberar 200 habitaciones (no debe dejarnos y seguir todo igual).
5. Ocupar 30 habitaciones reservadas (debe dejarnos).
6. Ocupar 30 habitaciones reservadas (no debe dejarnos porque quedaban 20 entre las reservadas).
7. Liberar 40 habitaciones (debe dejarnos).

Luego crear el hotel “Astoria”, “MDQ”, 150, 100, 10 e intentar exactamente las mismas operaciones. El resultado de la ejecución del programa para este segundo hotel debería mostrar lo siguiente:

```
Estado inicial del hotel Astoria (MDQ)
Hotel Astoria MDQ [habitaciones=150, ocupadas=100, reservadas=10,
libres=50, disponibles=40]
Intento ocupar 201 habitaciones sin reserva.
No pude ocupar las habitaciones requeridas
Hotel Astoria MDQ [habitaciones=150, ocupadas=100, reservadas=10,
libres=50, disponibles=40]
Intento ocupar 30 habitaciones sin reserva.
Pude ocupar las habitaciones requeridas
Hotel Astoria MDQ [habitaciones=150, ocupadas=130, reservadas=10,
libres=20, disponibles=10]
Intento reservar 50 habitaciones.
No pude reservar las habitaciones requeridas
Hotel Astoria MDQ [habitaciones=150, ocupadas=130, reservadas=10,
libres=20, disponibles=10]
Intento liberar 200 habitaciones.
No pude liberar las habitaciones requeridas
Hotel Astoria MDQ [habitaciones=150, ocupadas=130, reservadas=10,
libres=20, disponibles=10]
Intento ocupar 30 habitaciones con reserva.
No pude ocupar las habitaciones requeridas
Hotel Astoria MDQ [habitaciones=150, ocupadas=130, reservadas=10,
libres=20, disponibles=10]
Intento ocupar 30 habitaciones con reserva.
No pude ocupar las habitaciones requeridas
Hotel Astoria MDQ [habitaciones=150, ocupadas=130, reservadas=10,
libres=20, disponibles=10]
```

```
Intento liberar 40 habitaciones.
Pude liberar las habitaciones requeridas
Hotel Astoria MDQ [habitaciones=150, ocupadas=90, reservadas=10,
libres=60, disponibles=50]
```

6 Crear la clase **Automovil** con los siguientes atributos: marca (String), modelo (String), patente (String), capacidadTanque (double), cantidadDeCombustible (double) y rendimientoPorLitro (double, indica cuantos kilometros recorre con un litro de combustible).

Implementar los siguientes métodos y constructores:

a. Constructor parametrizado: recibe marca, modelo, patente y la capacidad del tanque de combustible.
b. Método público *viajar()*: recibe la cantidad de kilómetros. Actualiza la cantidad de combustible consumido invocando al método *consumirCombustible()* y devuelve la cantidad de kilómetros que efectivamente se realizaron con el combustible existente en el tanque. Debe descartar valores negativos (no puede “desviajar” kilómetros).
c. Método privado *consumirCombustible()*: recibe la cantidad de kilómetros que se quiere recorrer y actualiza la cantidad combustible en el tanque según los kilómetros requeridos y el rendimiento por litro. Devuelve un double indicando los litros consumidos.
d. Método público *cargarCombustible()*: recibe por parámetro la cantidad que se quiere cargar, que nunca debe ser menor o igual que cero o mayor al espacio disponible. Si puede, actualiza el atributo correspondiente. Devuelve true o false dependiendo del éxito de la acción.
e. Método privado *espacioDisponible()*, que devuelve la diferencia entre el tamaño del tanque y los litros de combustible almacenados.
f. Método público *pocoCombustible()*, booleano, que devuelve verdadero siempre que la cantidad de combustible sea menor al 15% de la capacidad del tanque.
g. Todos los *getters* públicos, todos los *setters* privados salvo *setPatente()* y *setRendimientoPorLitro()*.
h. El método toString() que mostrará todos los valores de estado de la clase incluyendo *espacioDisponible()* y *pocoCombustible()*.

En la clase test, crear el objeto a través del constructor. "Ford","Fiesta", "ABCD123", capacidad tanque:40. Setear el rendimiento por litro en 10 y llenar el tanque con 20 (veinte) litros de combustible.

Probar todos los métodos recorriendo 180 kilómetros primero e intentando recorrer 50 después, mostrando siempre la cantidad de kilómetros que devuelve *viajar()*. La salida sería la siguiente:

```
Automovil [cantidadDeCombustible=0.0, capacidadTanque=40.0, marca=Ford,
modelo=Fiesta, patente=ABCD123, rendimientoPorLitro=0.0,
pocoCombustible()=true]
Seteo el rendimiento por litro en 10 (km)
Automovil [cantidadDeCombustible=0.0, capacidadTanque=40.0, marca=Ford,
modelo=Fiesta, patente=ABCD123, rendimientoPorLitro=10.0,
pocoCombustible()=true]
Cargo 20 litros de combustible
Automovil [cantidadDeCombustible=20.0, capacidadTanque=40.0, marca=Ford,
modelo=Fiesta, patente=ABCD123, rendimientoPorLitro=10.0,
pocoCombustible()=false]
Intente recorrer 180.0 kilometros y recorri 180.0.
Automovil [cantidadDeCombustible=2.0, capacidadTanque=40.0, marca=Ford,
modelo=Fiesta, patente=ABCD123, rendimientoPorLitro=10.0,
pocoCombustible()=true]
Intente recorrer 50.0 kilometros y recorri 20.0.
Automovil [cantidadDeCombustible=0.0, capacidadTanque=40.0, marca=Ford,
modelo=Fiesta, patente=ABCD123, rendimientoPorLitro=10.0,
pocoCombustible()=true]
```

7 Crear una clase **Gato** y una clase **Raton**. Ambos tienen como atributo energía (int) y un constructor que recibe un valor que completa este dato.

El gato y el ratón recorren la cantidad de metros que permita la energía que tengan. Un gato puede correr mientras le dé la energía, descontando un punto de energía por metro recorrido. Al ratón se le descuentan 2 puntos de energía por cada metro recorrido.

Implementar en **Gato** el método *alcanzar()* que reciba como parámetro una instancia de **Raton** y la distancia en metros que debe recorrer para alcanzarlo. Devolverá *true* o *false* dependiendo de si lo alcanza o no.

Ejemplo: un gato con 100 de energía puede correr 100 metros, pero un ratón con los mismos 100 de energía puede correr sólo 50 metros. O sea que si el ratón está a más de 50 metros del gato y ambos tienen 100 de energía, no lo va a alcanzar.

Crear un gato y un ratón en la clase **Test** con y probar con 10, 50 y 80 metros, ambos con 100 de energía.

8 Crear la clase **Superheroe** con los atributos *nombre* (String), *fuerza* (int), *resistencia* (int) y *superpoderes* (int). Todos los atributos numéricos deberán aceptar valores entre 0 y 100; en caso de que el *setter* correspondiente reciba un valor fuera de rango

deberá setear el valor límite correspondiente (si recibe un valor negativo asignar cero y si recibe un valor superior a cien asignar cien).

El constructor de la clase recibirá todos los valores de sus atributos por parámetro y usará los setters (todos privados) para validar el rango correcto de los poderes del superhéroe.

También habrá el método *toString()* para mostrar el estado completo de la instancia y un método *competir()*, ambos públicos. Este último recibirá otro superhéroe como parámetro y, comparando los poderes de uno con otro, devolverá "TRIUNFO", "EMPATE" o "DERROTA", dependiendo del resultado. Para triunfar un superhéroe debe superar al otro en al menos 2 de los 3 ítems.

Escribir la clase **Test** que contenga el método *main()* para probar el correcto funcionamiento de la clase previamente realizada con el siguiente ejemplo:

superHeroe1: Nombre: "Batman", Fuerza: 90, Resistencia: 70, Superpoderes: 0

superHeroe2: Nombre: "Superman", Fuerza: 95, Resistencia: 60, Superpoderes: 70.

Hacer jugar al superheroe1 pasándole el objeto superheroe2 y mostrar el resultado por pantalla. Chequear el resultado (debería ser "DERROTA").

Hacer jugar al superheroe2 contra el superheroe1 y mostrar el resultado por pantalla. Chequear el resultado (debería ser "TRIUNFO").

Crear más superhéroes con distintos valores y jugar.