EJERCICIO TELEFONO

INFORMACIÓN IMPORTANTE PREVIA A LA EVALUACIÓN

LA CLASE STRING

Cuando quiera comparar el contenido de un String, no utilice el operador == , en su defecto utilice la función equals, que devuelve true si el contenido del String es igual o false si no lo es. Para comparar valores primitivos si debe usar el operador ==, por ejemplo, para comparar los valores de un int.

Ejemplo:

```
String a=”abc”;

boolean resultado=a.equals(“abc”); //resultado es true

if(a.equals(“x”)){

        System.out.println(“son iguales”);//no imprime porque el resultado de la comparación dentro del if es false

}
```

Puede usar el operador == para comparar si el String es null

```
if(a==null) //es correcto
```

Para comparar si es diferente de null, puede usar el operador!=

```
if(a!=null) //es correcto
```

OPERADORES LÓGICOS

&& : (y) resulta true, si y solo si ambas proposiciones son verdaderas

||: (o) resulta true, si alguna de las dos proposiciones es verdadera

EJEMPLO:

```
int a=2;

int b=2;

if(a==b && b>0) // resultado es true
```

```
if(a==b && b<2) // resultado es false

if(a==b || b<2) // el resultado es true
```

EJEMPLO:

```
public void test(String a){
        if(a.equals(“123”){
                //algo si es igual a 123
        }else{
                //algo si es diferente a 123
        }
}
```

En el ejemplo planteado si es igual a 123 ejecuta un código, caso contrario ejecuta otro, pero si a llegar null, se produciría un NullPointerException, por lo cual antes de usar el equals se debe validar si la variable no es null

```
public void test(String a){
        if(a!=null && a.equals(“123”){
                //algo si es igual a 123
        }else{
                //algo si es diferente a 123
        }
}
```

El operador && funciona de la siguiente forma: Si la primera proposición es false, ya no valida la segunda porque ya todo el resultado es false.

Esto nos permite que como está planteado el ejemplo no va a ocurrir un NullPointerException jamás, ya que si el primer valor es null, la comparación resulta false y ya no valida la segunda parte a.equals(“123”), solo llegaría a la segunda proposición si la primera es verdadera, es decir si a es diferente de null.

EQUALSIGNORECASE

Al comparar el contenido con equals, el resultado es true, si la cadena coincide exactamente, distinguiendo mayúsculas de minúsculas.

Si se quiere que la comparación sea independiente de mayúsculas y minúsculas se usa equalsIgnoreCase, por ejemplo:

```
String a=”abc”;

String b=”Abc”;

if(a.equals(b)) //el resulado es false

if(a.equalsIgnoreCase(b)) //el resultado es true.
```

EVALUACIÓN

Crear un nuevo proyecto con el nombre **Java**<NombreApellido>, por ejemplo, **JavaSantiagoMosquera** dentro de su repositorio remoto.

Crear los paquetes:

com.krakedev.evaluacion

com.krekedev.test

Incluir en este paquete todas las clases Test que se colocan en recursos. Recuerde que las clases Test no deben modificarse en ningún momento.

REQUERIMIENTO 1 (10)

Crear una clase Teléfono, con la siguiente definición:

Atributos: **número**, **tipo** y **estado**, todos de tipo String

Agregar getters para los 3 atributos, no agregar setters

Agregar un constructor que reciba el **número** y el **tipo** y ejecute la siguiente lógica:

Asigna los valores de número y tipo a los atributos respectivos.

Dependiendo de las validaciones que realice, colocará el valor de estado:

C : Si es correcto

E : Si tiene error

Para determinar si es correcto o no el teléfono, se debe cumplir las siguientes condiciones:

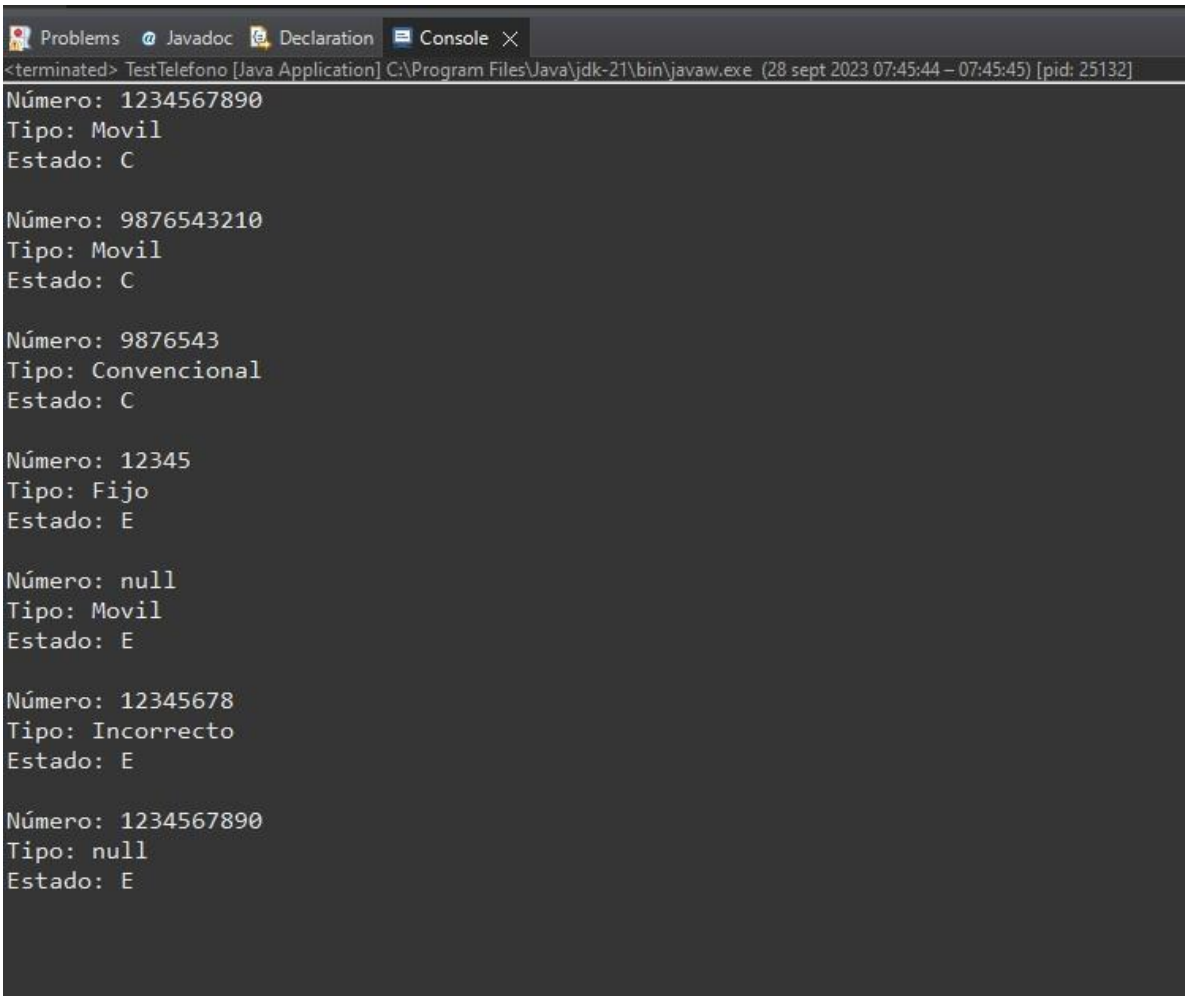
Tanto el teléfono como el tipo deben ser diferentes de null, si alguno es null, se considera con error

Los tipos pueden ser Movil o Convencional, si es un tipo diferente, se considera con error.

Si el tipo es Movil, el teléfono debe tener 10 caracteres, si es convencional, debe tener 7 caracteres. Validar únicamente la longitud, no el contenido de la cadena. La longitud de un String se obtiene con el método length()

TEST: TestTelefono.java

RESULTADO ESPERADO



REQUERIMIENTO 2 (5)

Crear una clase Direccion, con la siguiente definición:

Atributos callePrincipal y calleSecundaria, ambos tipo String

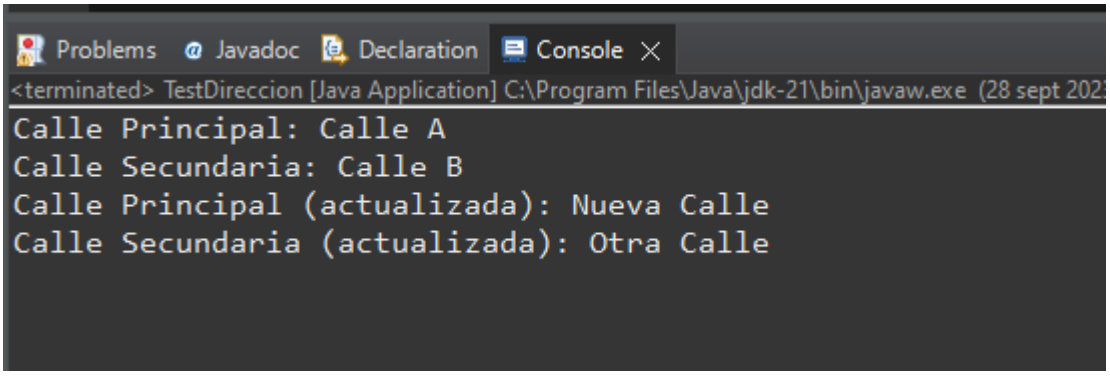
Getters y Setters para cada atributo

Constructor que recibe ambos parámetros y los setea

Hacer lo necesario en Dirección, para que la clase TestDireccion compile, y el ejecutar se obtenga los resultados esperados

TEST: `TestDireccion`.java

RESULTADO ESPERADO



REQUERIMIENTO 3 (5)

Crear la clase contacto con la siguiente definición:

Atributos: cedula, nombre y apellido, de tipo String

direccion de tipo Direccion

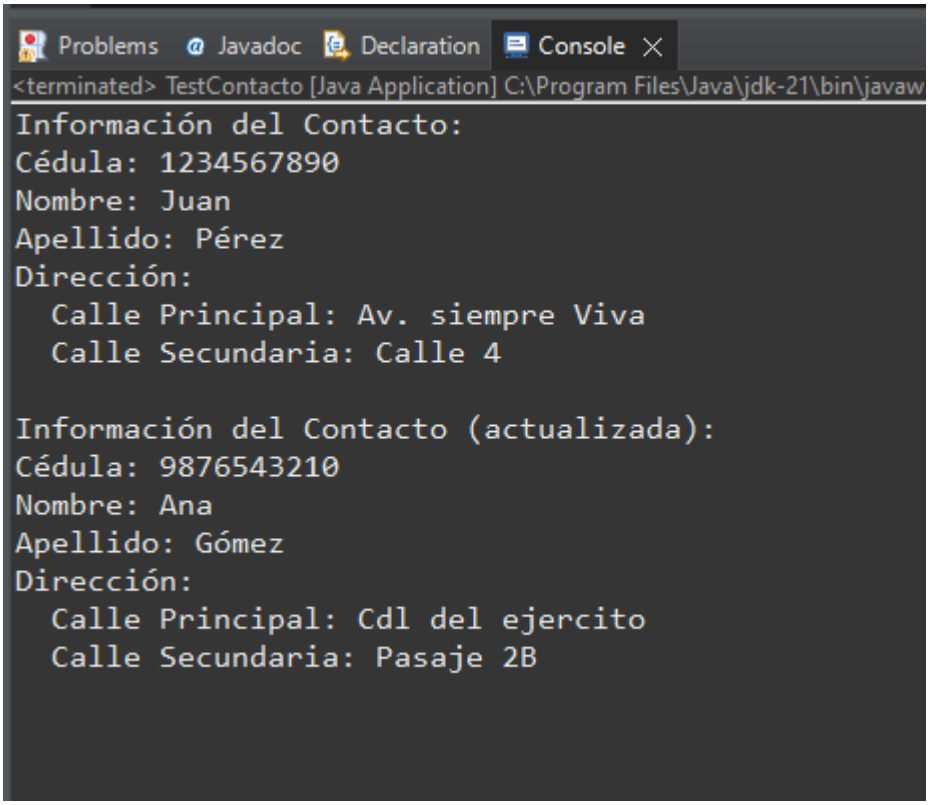
Constructor que recibe cedula,nombre y apellido y los asigna a los atributos correspondientes

Getters y setters para cada atributo

Método imprimir, que muestra la información de los clientes de manera que se muestre tal como en resultado esperado

TEST: TestContacto.java

RESULTADO ESPERADO

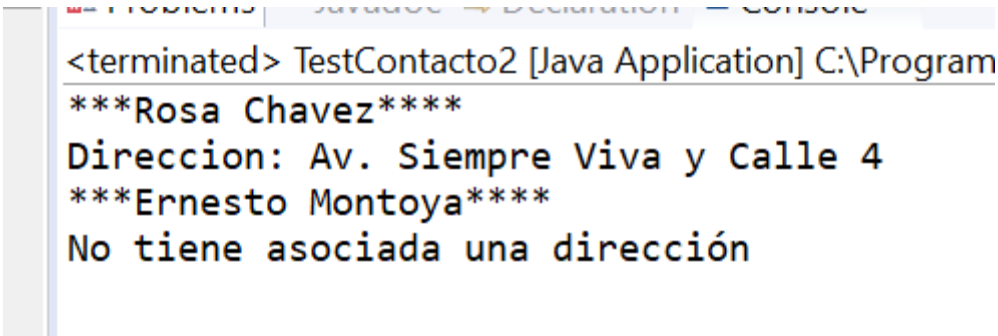


REQUERIMIENTO 4 (5)

Modificar el método imprimir para que al ejecutar TestContacto2, se obtenga los resultados esperados

TEST: TestContacto2.java

RESULTADO ESPERADO



REQUERIMIENTO 5 (10)

En la clase Contacto:

Agregar el atributo telefonos, del tipo ArrayList de Telefono, con los getters y setters respectivos.

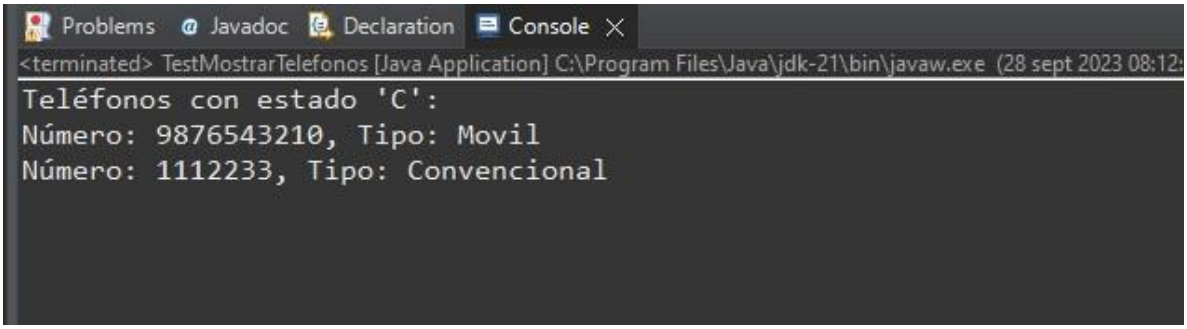
Crear un método agregarTelefono, que no retorna nada, recibe un Telefono y lo agrega a la lista de teléfonos.

Crear un método mostrarTelefonos, que no retorna nada, no recibe nada y muestra en consola todos los teléfonos ingresados con el formato que se indica en resultado esperado.

IMPORTANTE: El método mostrarTelefonos solo muestra los que tengan estado C

TEST: TestMostrarTelefonos

RESULTADO ESPERADO:

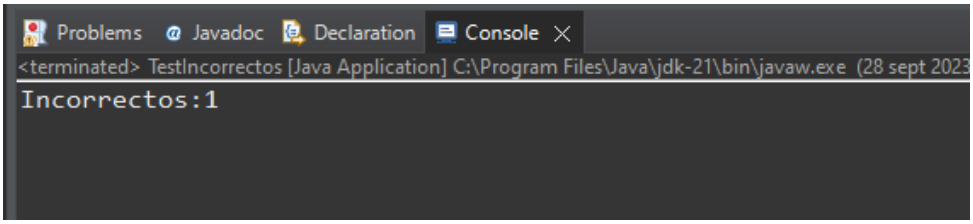


REQUERIMIENTO 6 (5)

Crear un método recuperarIncorrectos, no recibe nada y retorna un ArrayList de Telefono, con todos los teléfonos en estado E (Error)

Test: TestIncorrectos.java

RESULTADO ESPERADO:



REQUERIMIENTO 7 (5)

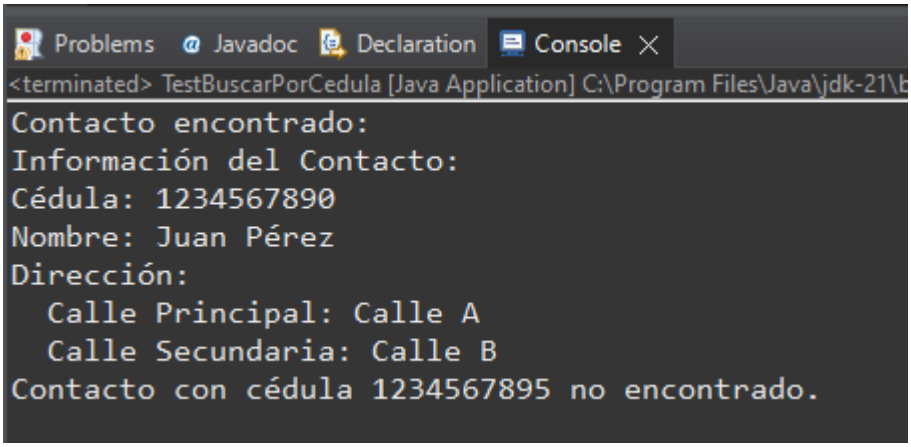
Crear una clase Directorio con un atributo contactos del tipo ArrayList de Contacto y un atributo fechaModificacion, de tipo Date (del paquete java.util)

Crear un método agregarContacto, que recibe un contacto, lo agrega a la lista y retorna true.

Agregar el método buscarPorCedula, que recibe la cédula de un contacto y lo busca en la lista, si existe retorna el Contacto, si no existe retorna null

TEST: TestBuscarPorCedula.java

RESULTADO ESPERADO:



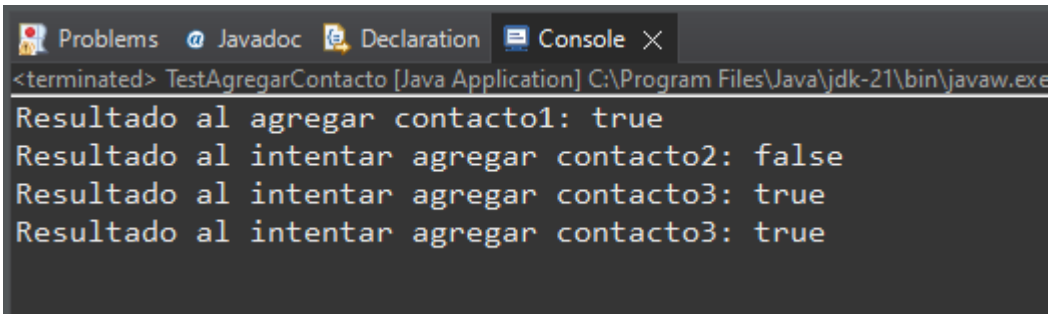
REQUERIMIENTO 8 (5)

Modificar el constructor para que no permita agregar Contactos con una cédula que ya exista en la lista de contactos. Reusar el método buscar por cédula para saber si existe o no en la lista

Si logra agregar retorna true, caso contrario retorna false

TEST: TestAgregarContacto.java

RESULTADO ESPERADO



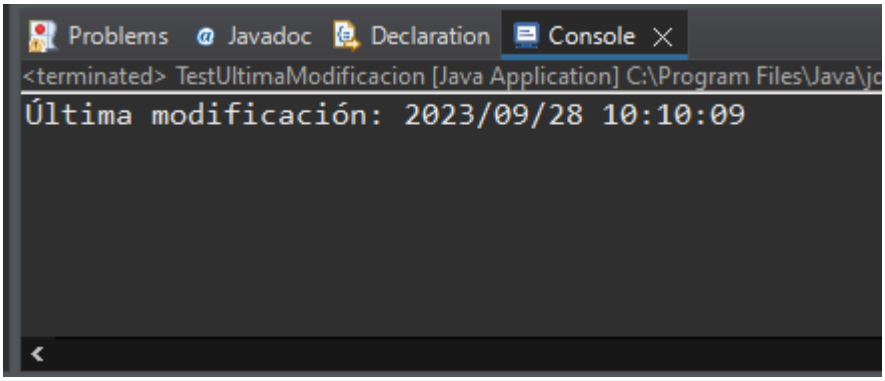
REQUERIMIENTO 9 (5)

Cada vez que logra agregar un contacto, se asigna un valor a ultimaModificacion, instanciando un objeto de tipo Date usando el constructor vacío.

Agregar un método consultarUltimaModificacion que retorna un String, con la fecha de última modficación en el formato: “yyyy/mm/dd HH:mm:ss”, para esto usar la clase SimpleDateFormat (buscar en google su uso)

TEST: TestUltimaModificacion

RESULTADO ESPERADO: Solo en este test, el resultado no va a ser igual ya que tendrá los datos de la fecha y hora en la que ejecute su prueba

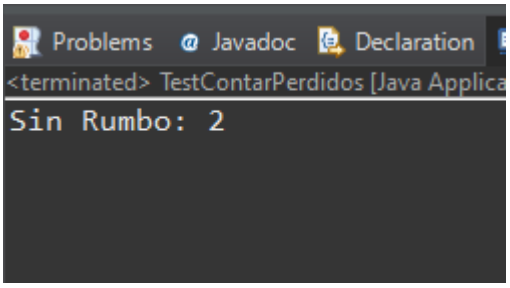


REQUERIMIENTO 10 (5)

En Directorio agregar un método contarPerdidos, que retorna cuantos contactos se tienen sin asignar dirección

TEST: TestContarPerdidos.java

RESULTADO ESPERADO

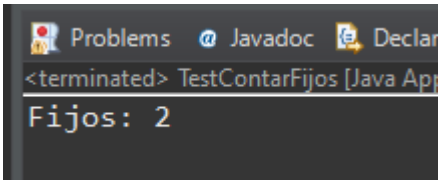


REQUERIMIENTO 11 (5)

En Directorio agregar un método contarFijos, que retorna cuantos contactos tienen teléfono Convencional y además están en estado C.

TEST: TestContarFijos.

RESULTADO ESPERADO



REQUERIMIENTO 12 (10)

En Directorio agregar 2 atributos: correctos e incorrectos, ambos del tipo ArrayList de Contacto

Agregar un método llamado depurar que no recibe ni retorna nada y ejecuta la siguiente lógica:

Coloca los contactos que tienen dirección asignada en la lista correctos y los que no tienen dirección en incorrectos.

Luego de repartirlos, vaciar la lista de contactos

TEST: TestDepurar

RESULTADO ESPERADO