|  |  |  |
| --- | --- | --- |
|  | **Carátula para entrega de prácticas** | |
| Facultad de Ingeniería | | Laboratorio de docencia |

Laboratorios de computación

salas A y B

|  |  |
| --- | --- |
| *Profesor:* | Ing. Guadalupe Lizeth Parrales Romay |
| *Asignatura:* | Programación Orientada a Objetos |
| *Grupo:* | 04 |
| *No de Práctica(s):* | 8 |
| *Integrante(s):* | Dávila Ortega Jesús Eduardo - No. Cuenta: 317199860  Díaz Hernández Marcos Bryan - No. Cuenta: 317027253  Pareja Ávila Emiliano - No. Cuenta: 317081345  Vázquez Zavala Oliver Alexis - No. Cuenta: 317202263 |
| *No. de Equipo de cómputo empleado:* |  |
| *No. de Lista o Brigada:* | 12,14, 30, 37 |
| *Semestre:* | 2021-1 |
| *Fecha de entrega:* | 28 de noviembre del 2020 |
| *Observaciones:* |  |
|  |  |

CALIFICACIÓN: \_\_\_\_\_\_\_\_\_

**Objetivo de la práctica.**

Implementar el concepto de polimorfismo en un lenguaje de programación orientado a objetos.

**Introducción.**

El polimorfismo se puede definir como la propiedad de que los objetos nunca dejan de reconocerse como miembros de una determinada clase, de tal manera que sin importar el tipo del cual sea la variable que los contenga, ante la invocación de cualquiera de sus métodos siempre se ejecutan de acuerdo con la clase que los define.

Esto permite enviar mensajes iguales a objetos de diferentes tipos por la posibilidad de la sobrecarga y la sobreescritura, que se manejan en la herencia y dentro de las propias clases.

Algo importante sobre el polimorfismo es su implementación por medio de las instancias, que dentro de este tema no es lo mismo a crear un objeto, ya que se puede dar la similitud entre los términos, porque cuando se crea un objeto se define el tipo de instancia y no es posible cambiarlo después de esto:

* Polígono polígono = new Polígono(); //tipo de dato polígono

Sin embargo cuando se define sólo la instancia de la clase, y no se le asigna un tipo de dato, está disponible para que le sea asignado otro tipo de dato. Donde es importante destacar que ese tipo de dato tendría que ser igual al de la instancia o ser de una subclase de esta.

* Polígono polígono = new Polígono(); //Se crea un objeto de la clase polígono
* Object objeto = polígono; //La referencia de object apunta hacia el objeto de polígono
* System.out.println(objeto); //Imprime el método toString del objeto de la clase polígono

El ejemplo muestra el cómo se puede hacer que una instancia de objeto pueda comportarse como la de polígono, esto es debido a que la clase padre sería Object y la clase hija sería polígono, debido a que todas las clases en Java descienden de Object.

Por medio de la creación de las instancias de clases padres es posible el ejecutar métodos de subclases, y esto es significa que se puede conseguir que un objeto tenga un comportamiento distinto al que se definió en su clase, y comportarse como el de su subclase, es necesario destacar que los métodos que se pueden ejecutar son los que se definen en la subclase y no se podrán utilizar otros.

Las clases abstractas son otro tema que se utiliza en la práctica, este es importante por el uso de la abstracción que envuelven en su implementación, ya que estas definen métodos sin describir su funcionamiento, esto significa que no pueden ser desarrollados por la falta de información, por ejemplo se tienen las figuras geométricas

donde existe la operación de calcular el área, pero no es posible definir el tipo de operación que se tendría que utilizar si no se sabe qué figura geométrica es, por ello el método sería considerado como abstracto, y una clase que tiene este tipo de métodos se conoce como una clase abstracta, sin embargo es posible el definir métodos concretos que puedan implementarse en cualquier caso.

public abstract class Polígono { // Sintaxis de la clase

public abstract área();// Método abstracto

public cantidadLados(){ //Método concreto

}

}

Para las clases abstractas es imposible el crear objetos de estas, porque no se tiene la información completa de las mismas, y solo es posible el crear instancias sin definir el tipo de datos, es decir crear referencias. En caso de que se herede una clase abstracta es importante el saber que es necesario el implementar todos los métodos que se definen en esta clase abstracta, de lo contrario se generan errores.

Otro tema importante que se utiliza son las interfaces, estas permiten la implementación el utilizar un concepto bastante interesante que es el mostrar que se hace, pero no definir cómo se hace. Esto significa que es una clase abstracta pura, donde no existen métodos concretos, todos los métodos son abstractos.

public interface NombreInterfaz {

tipoRetorno nombreMetodo([parámetros]);

}

Los métodos que se definen en un interfaz son públicos y abstractos, al mismo tiempo es posible definir atributos que serán: estáticos, públicos y finales, esto significa que pueden ser utilizados pero no modificados.

Para implementar una interfaz es necesario el colocar el término *implements,* una vez implementada la clase es necesario que se definan todos los métodos de esta, ya que son abstractos, es una condición para el uso de las interfaces, y una ventaja de estas es la posibilidad de poder implementar n interfaces, siempre y cuando se definan todos los métodos de estas.

* public class MiClase implements NombreInterfaz

**Análisis de los ejercicios.**

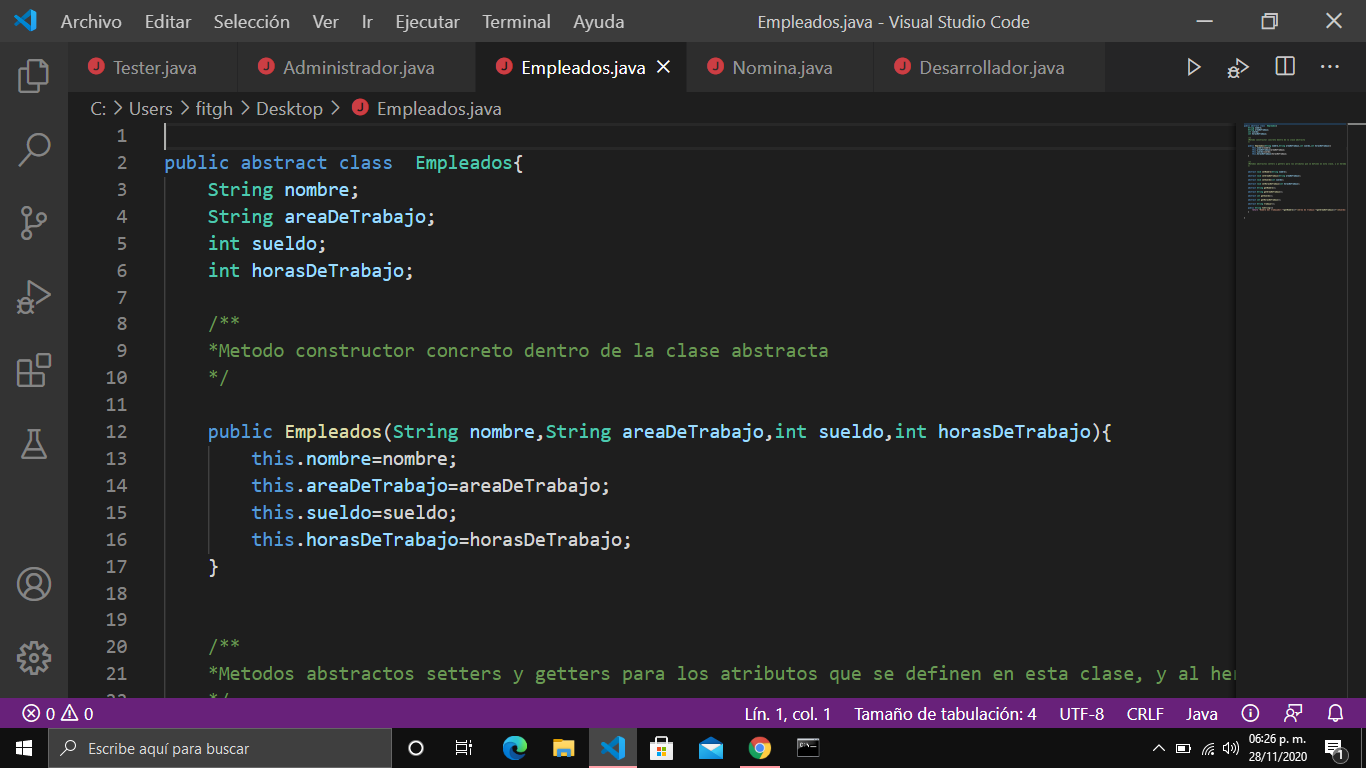
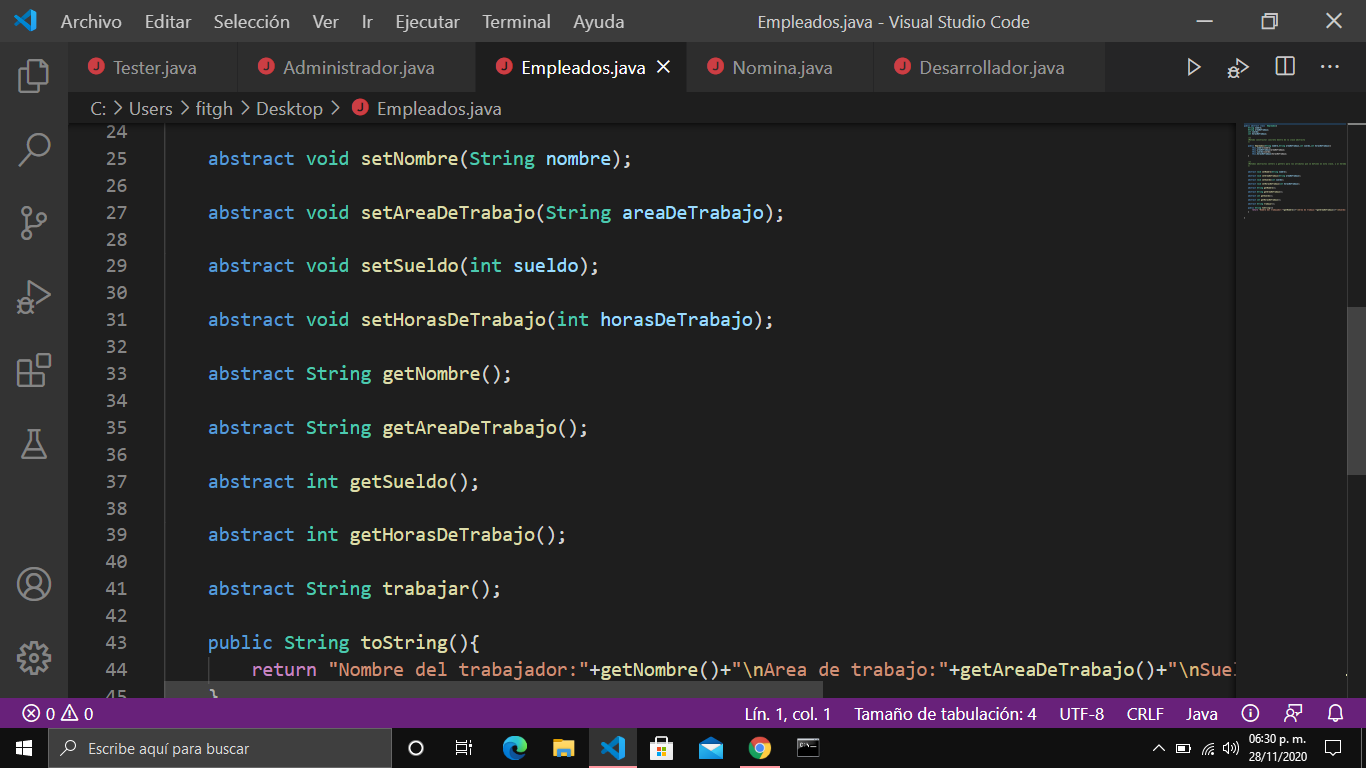
**Ejercicio 1**

* **Análisis previo**

Tomando como base la jerarquía de clases de empleados para una empresa de software desarrolladas en la práctica anterior y considerando los empleados: líder de proyecto, desarrollador, diseñador, tester y administrador, primero se debe crear una clase abstracta “Empleados”, que como su nombre lo indica es la encargada de proporcionar las bases para desarrollar la jerarquía de las clases que modelan a cada uno de los empleados de forma precisa, compartiendo algunas características heredadas de la clase abstracta, como lo son el nombre, el área de trabajo, el sueldo y las horas de trabajo, así como definir ciertos métodos abstractos como trabajar por ejemplo entre otros, además otra modificación requerida de la jerarquía previa es por la parte de la clase tester, la cual debe ser establecida como una clase abstracta con atributos respectivos a la posición en el proyecto, el tiempo de ejecución y el número de errores o problemas encontrados, esto debido a que a partir de ella se van a modelar otras dos clases concretas “Tester Desarrollador” y “Tester Usuario” en donde se requiere implementar ciertos métodos especializados para cada clase, además de sobrescribir otros como el caso de su método trabajar.

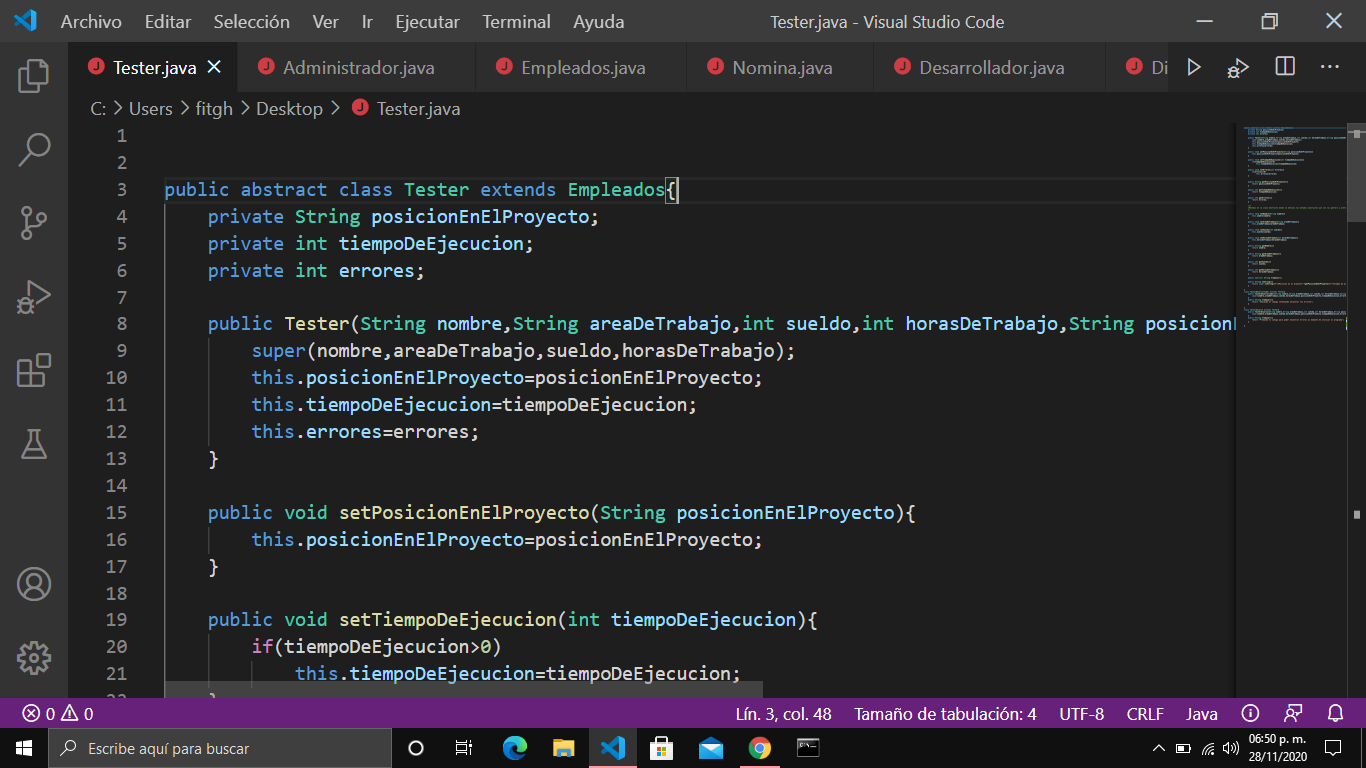
* **Desarrollo**

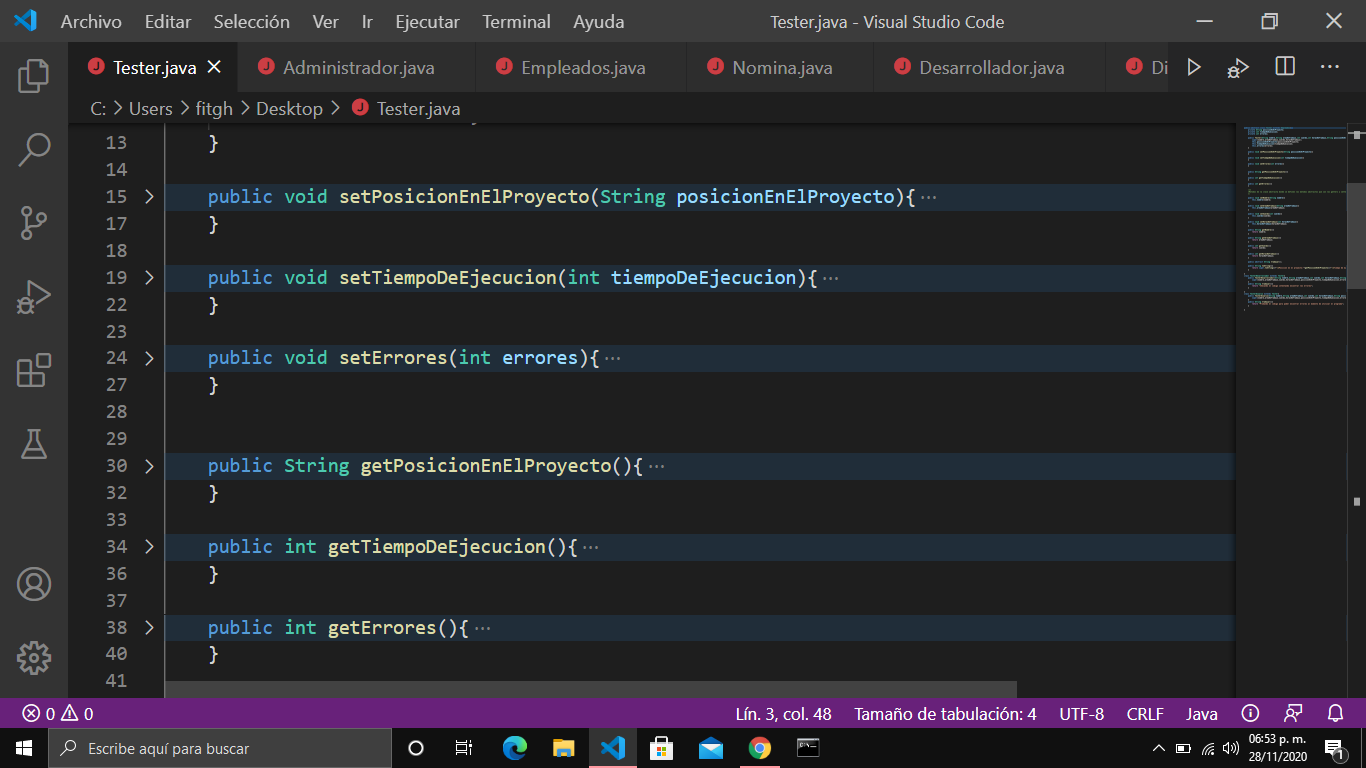
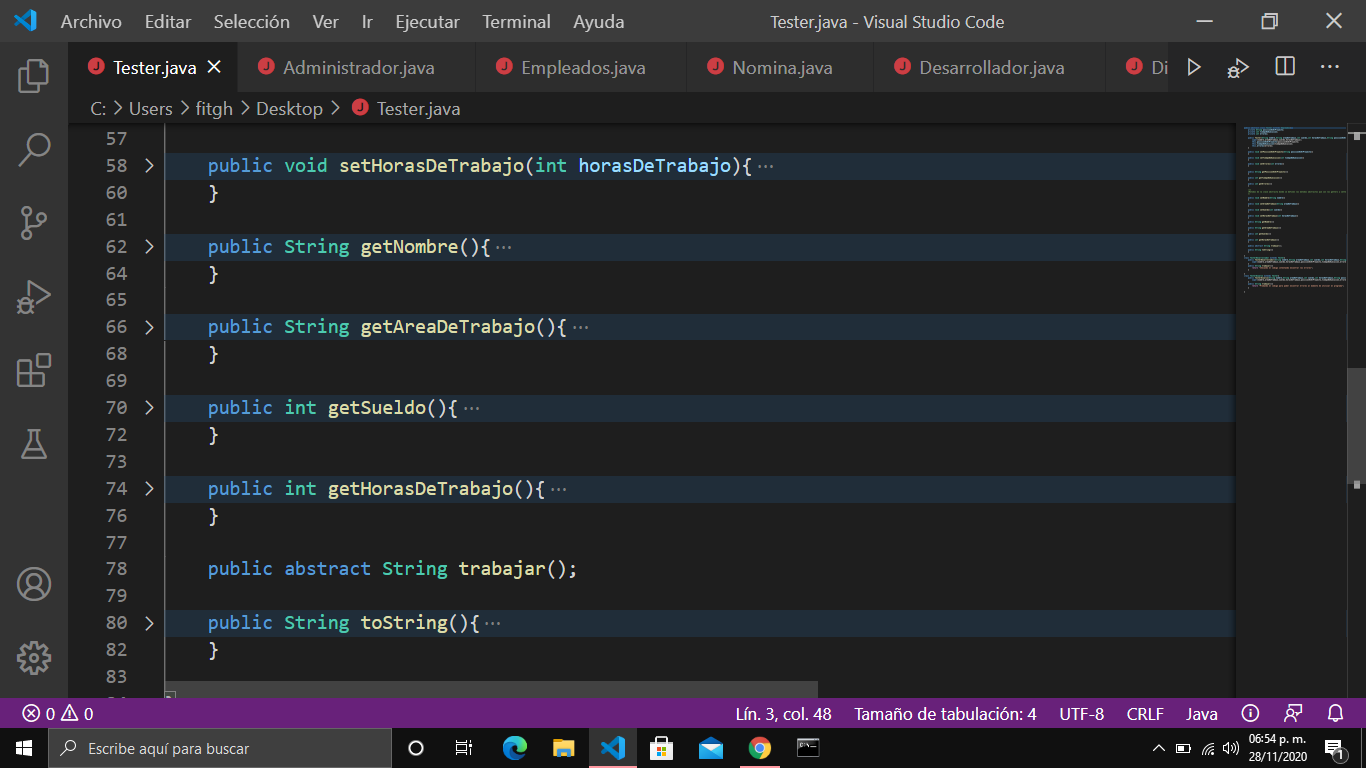
Inicialmente se estableció a la clase Empleado que anteriormente estaba definida como una clase genérica en una clase abstracta sirviendo como base para la jerarquía de clases de los empleados de la empresa de Software, permitiendo definir otras clases más especializadas correspondientes a cada uno de los empleados de la empresa, la clase abstracta Empleada dispone de los atributos del nombre, el área de trabajo, el sueldo y las horas de trabajo correspondientes a cada uno de los empleados y tiene la siguiente definición y cuenta con los siguientes métodos abstractos



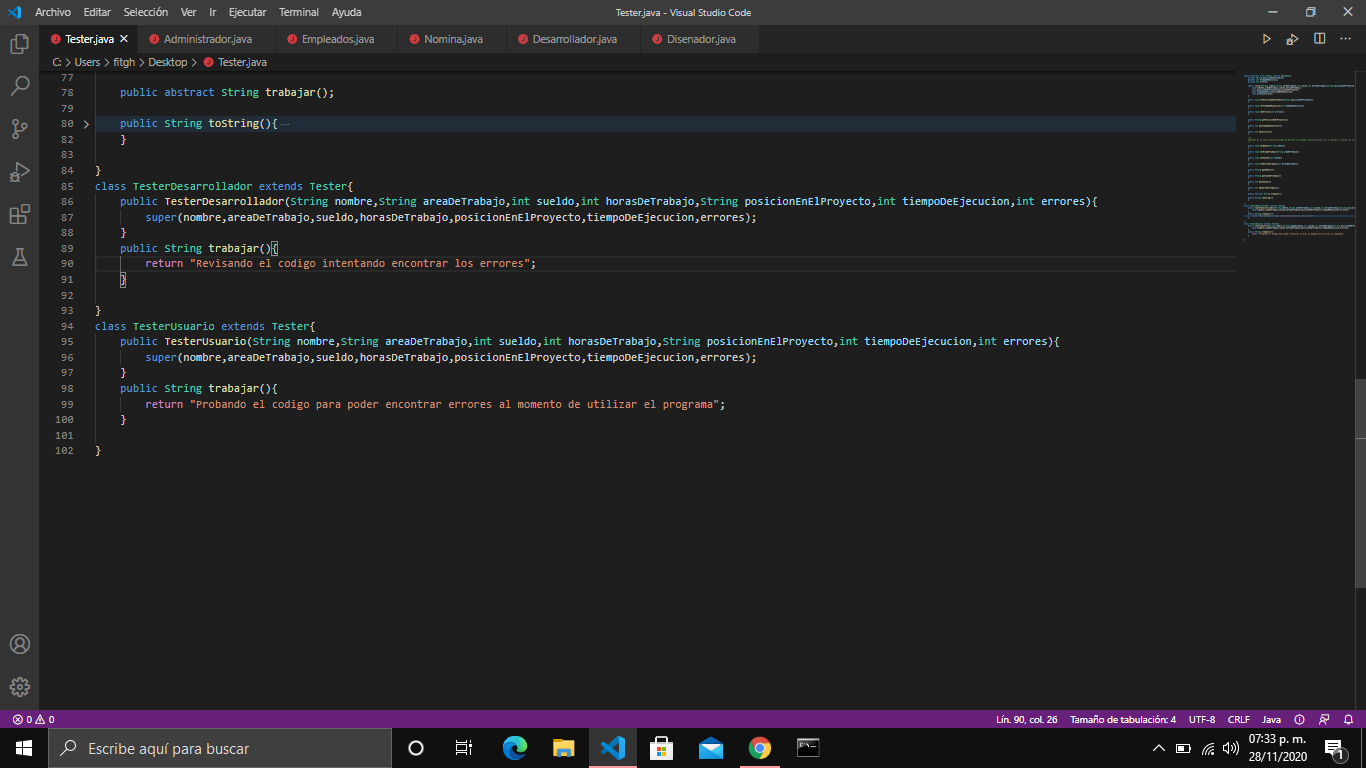
Definiendo a partir de esta clase y por relación de herencia las clases correspondientes a los diversos empleados de la empresa: líder de proyecto, desarrollador, diseñador, tester y administrador.

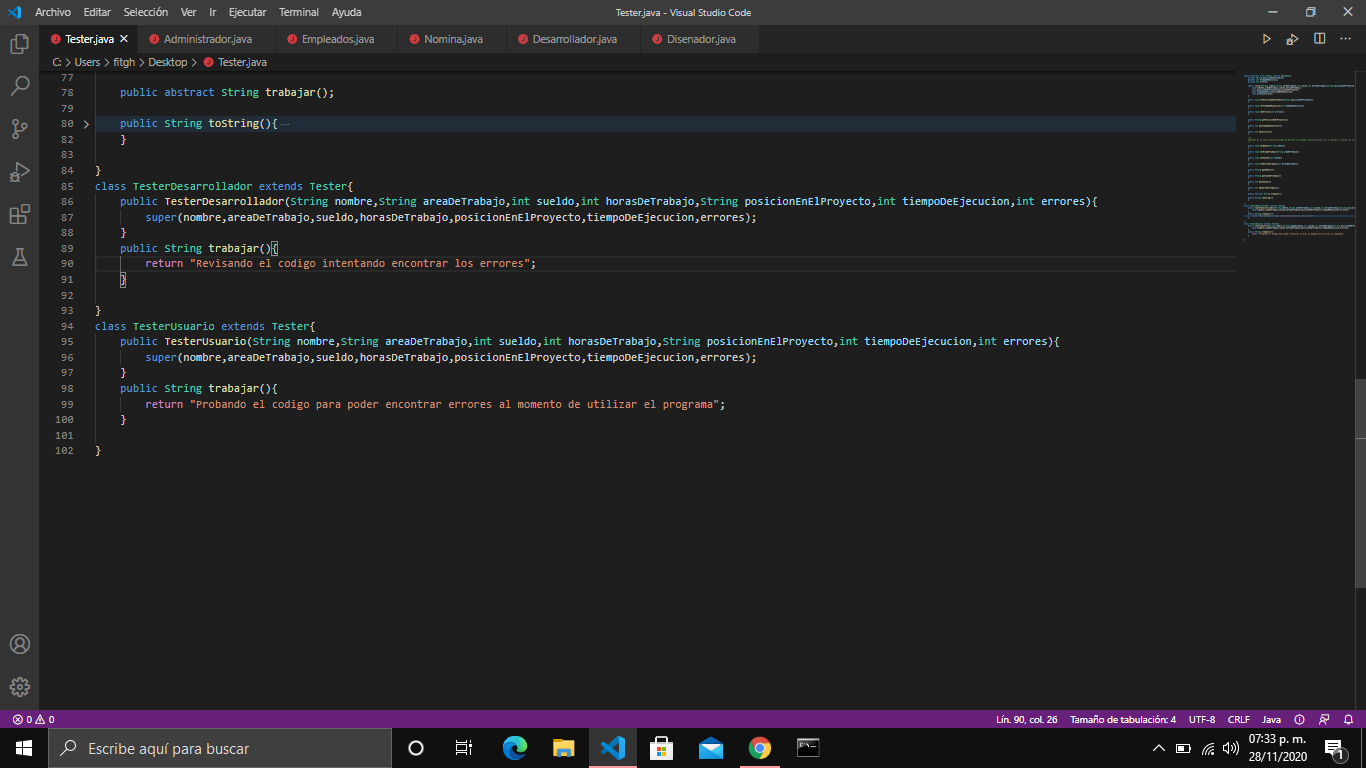
Por otro lado la clase abstracta definida tiene los atributos respectivos de la posición en el proyecto, el tiempo de ejecución y el número de errores o problemas encontrados, cuya definición y métodos son:



Posteriormente se modelaron las clases TesterDesarrollador y TesterUsuario donde la primera se encarga de modelar a un empleado del tipo Tester que analiza el código en busca de errores y el segundo a un trabajador que utiliza el software en busca de problemas al momento de la ejecución, debido a que ambas clases modelan a empleados que que realizan un trabajo en cierta parte distinto se requiere un método trabajar distinto en cada clase, por lo tanto la definición de las subclases es la siguiente:





* **Ejecución del programa**



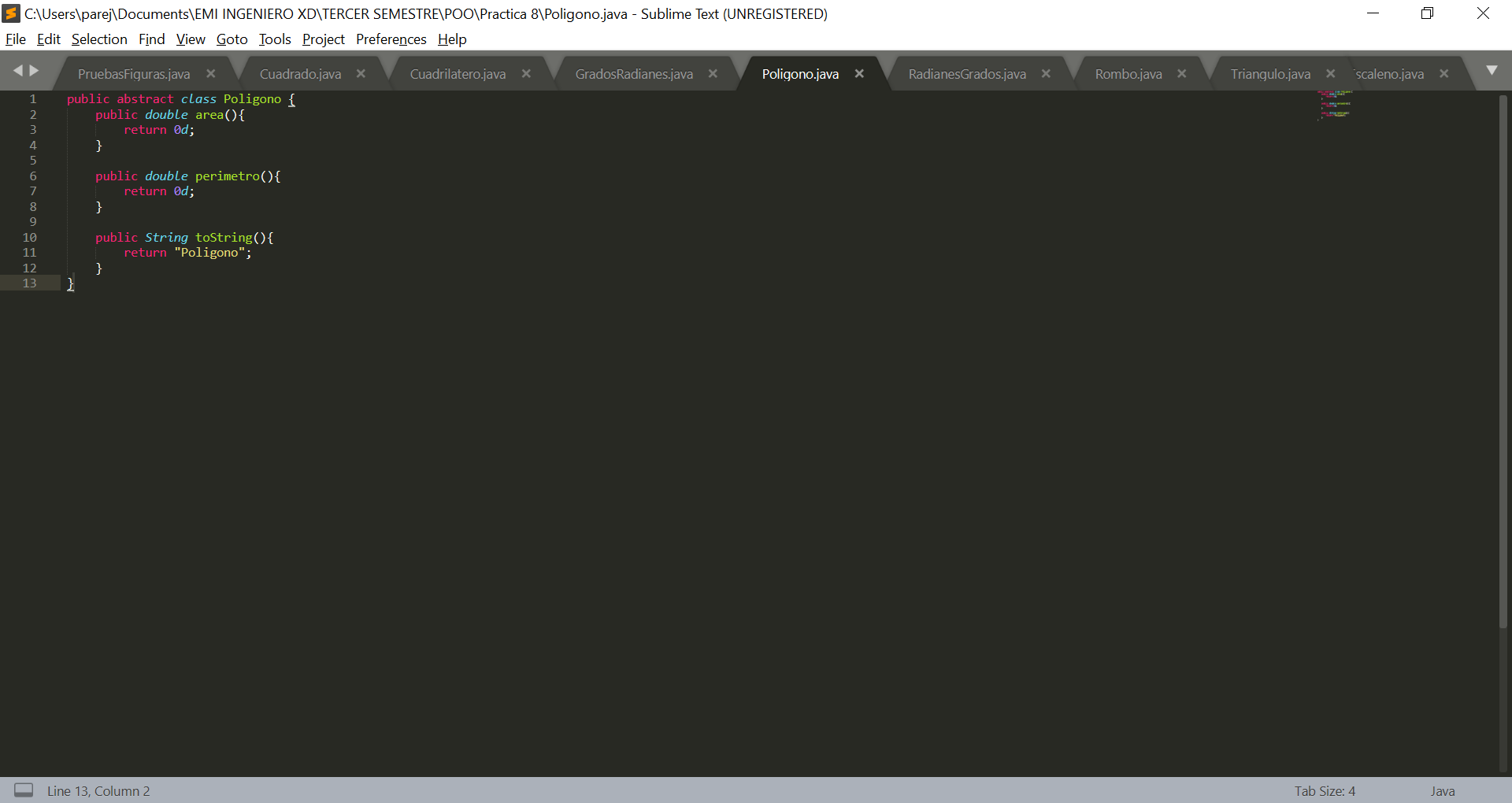
**Ejercicio 2**

* **Análisis previo**

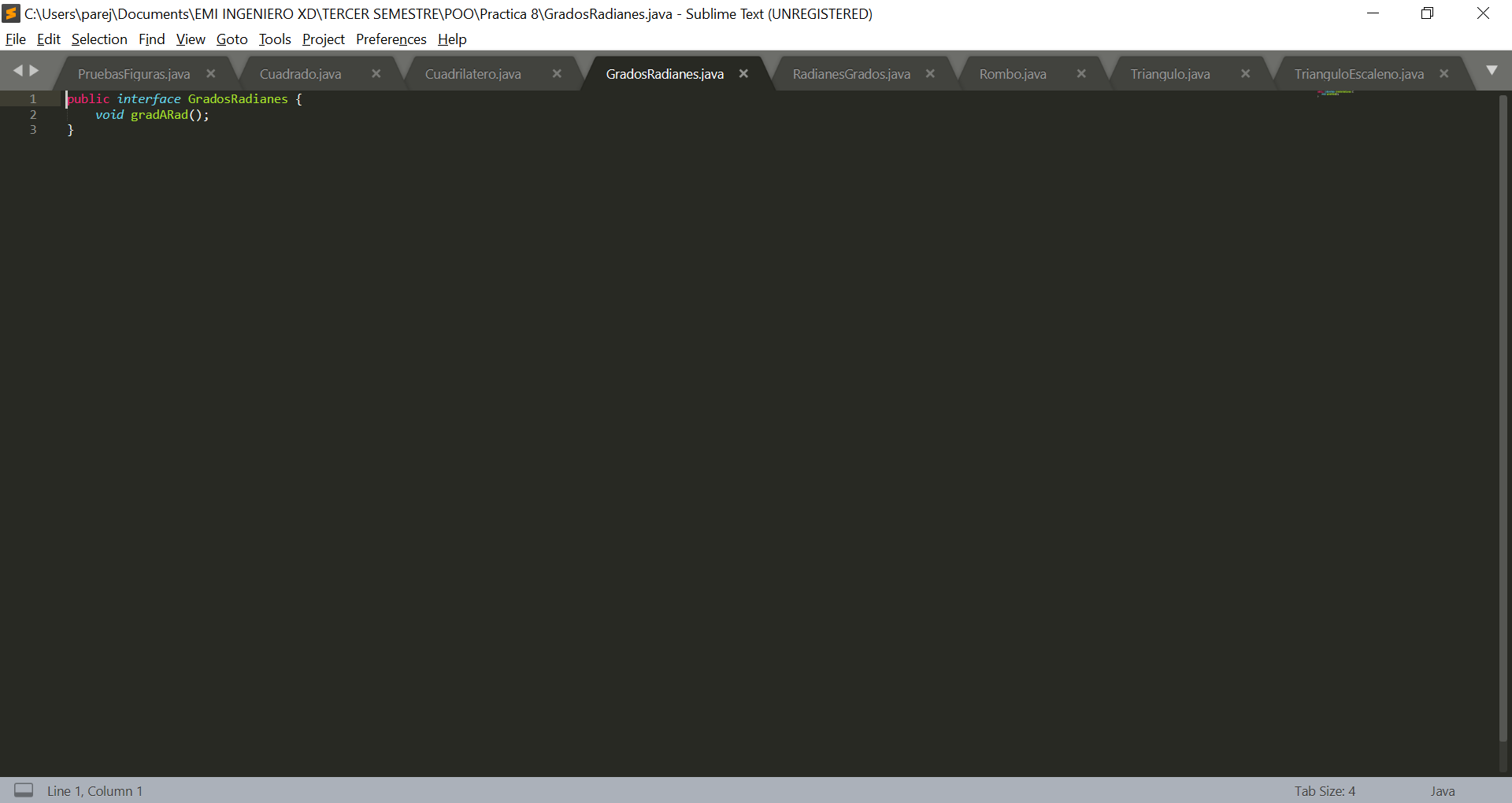
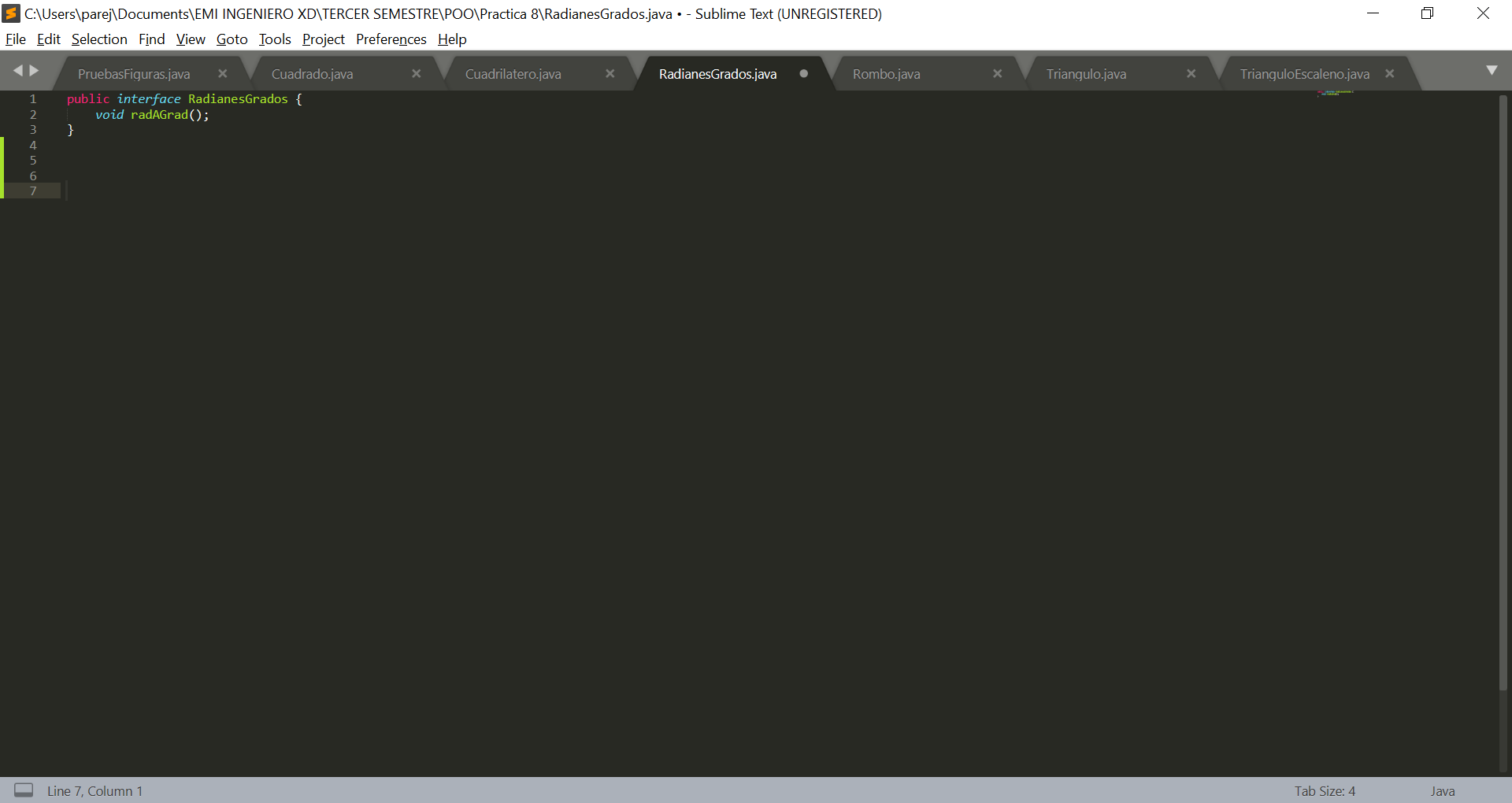
Para este ejercicio antes que nada, investigamos sobre cómo podríamos calcular los lados de cada una de las figuras que se nos pedían, investigamos sobre las leyes de senos para poder obtener los catetos en el caso del triángulo y rombo, debido a que en un cuadrado los ángulos son iguales estos por defecto ya se declararían de 90 grados. También para la clase Math.lang, en su método sin, tuvimos una confusión, ya que descubrimos que esta clase trabaja con radianes y no con ángulos como nosotros creíamos. También investigamos para las interfaces el cómo se convierten los grados a radianes y los radianes a grados, y descubrimos que para obtener un ángulo en radianes, multiplicamos a este ángulo por π y lo dividimos entre 180, y para obtener los radianes en ángulos, multiplicamos los radianes por 180 y los dividimos entre π. También para el desarrollo de las clases, idealizamos la abstracción de las clases Cuadrilátero y Triángulo y si es correcto que ambas sean abstractas, ya que un cuadrilátero puede ser desde un cuadrado, hasta un romboide o un trapecio, y sobre la clase Triángulo, existen 3 tipos diferentes de triángulos.

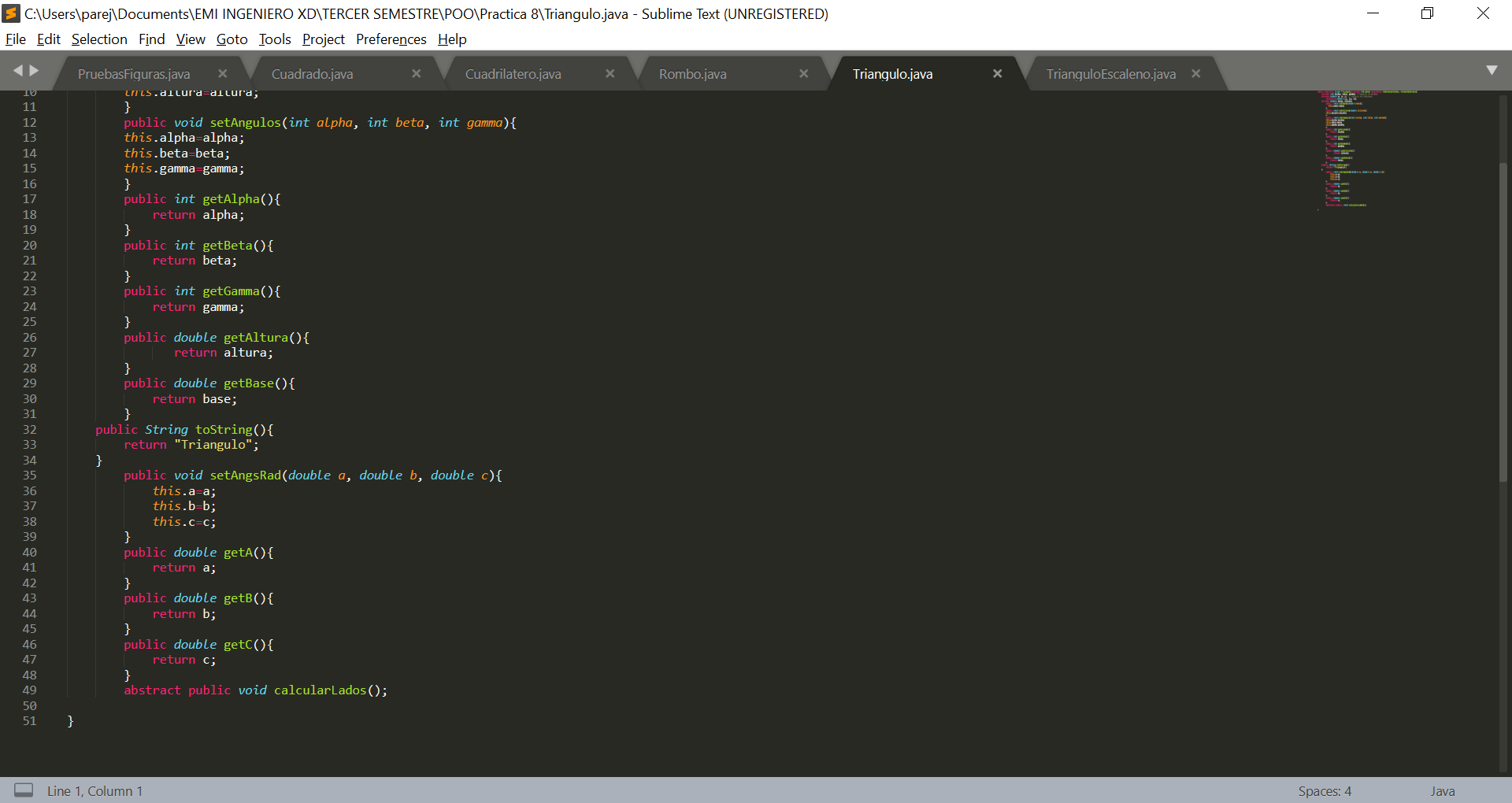
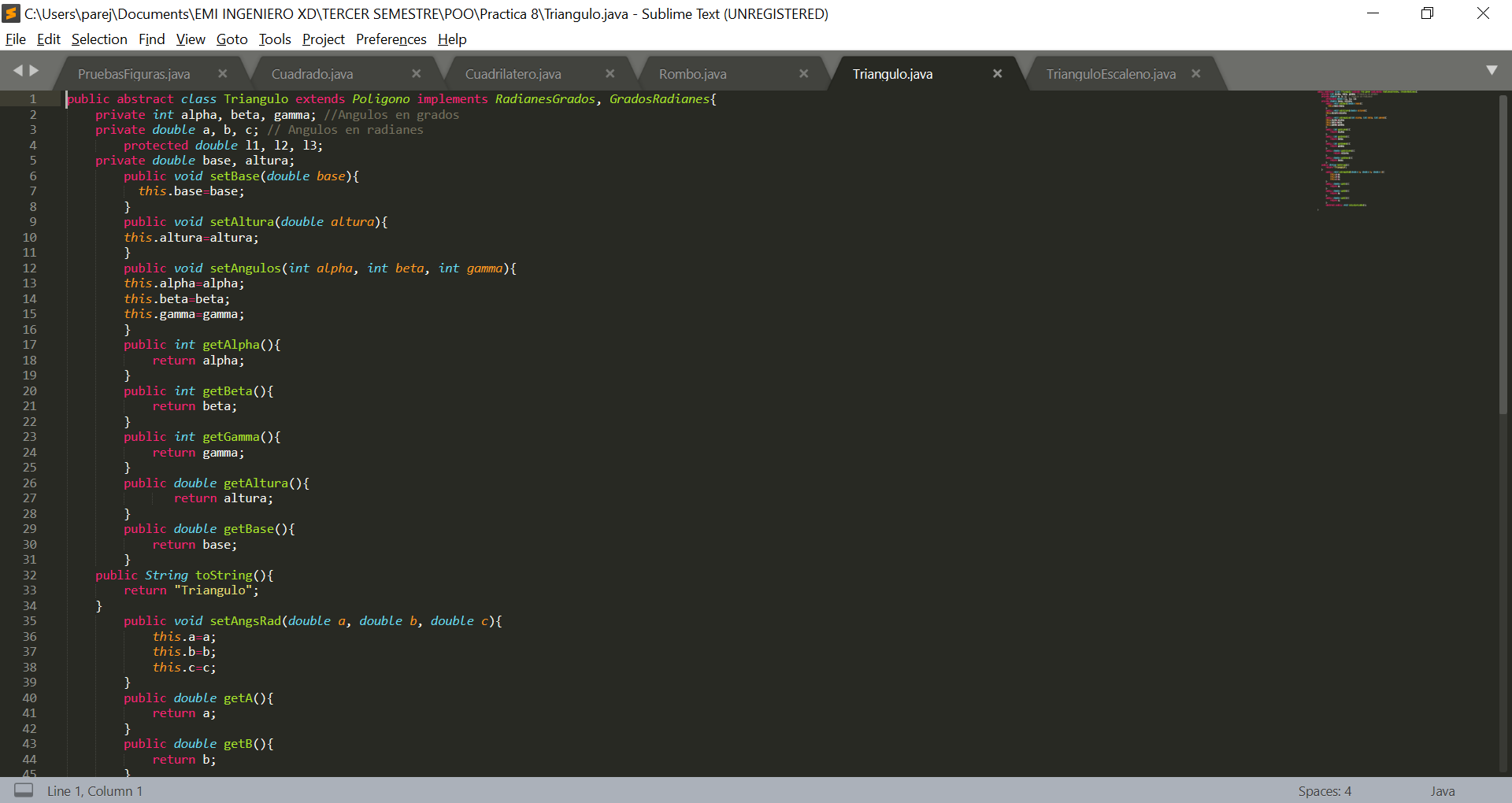
* **Desarrollo**

La clase padre de todas las demás subclases, Polígono, la dejamos igual, ya que sobrescribimos sus métodos en las demás clases, entonces no tenía caso escribir algo más en la clase, porque todas estas figuras geométricas tienen un perímetro y un área.



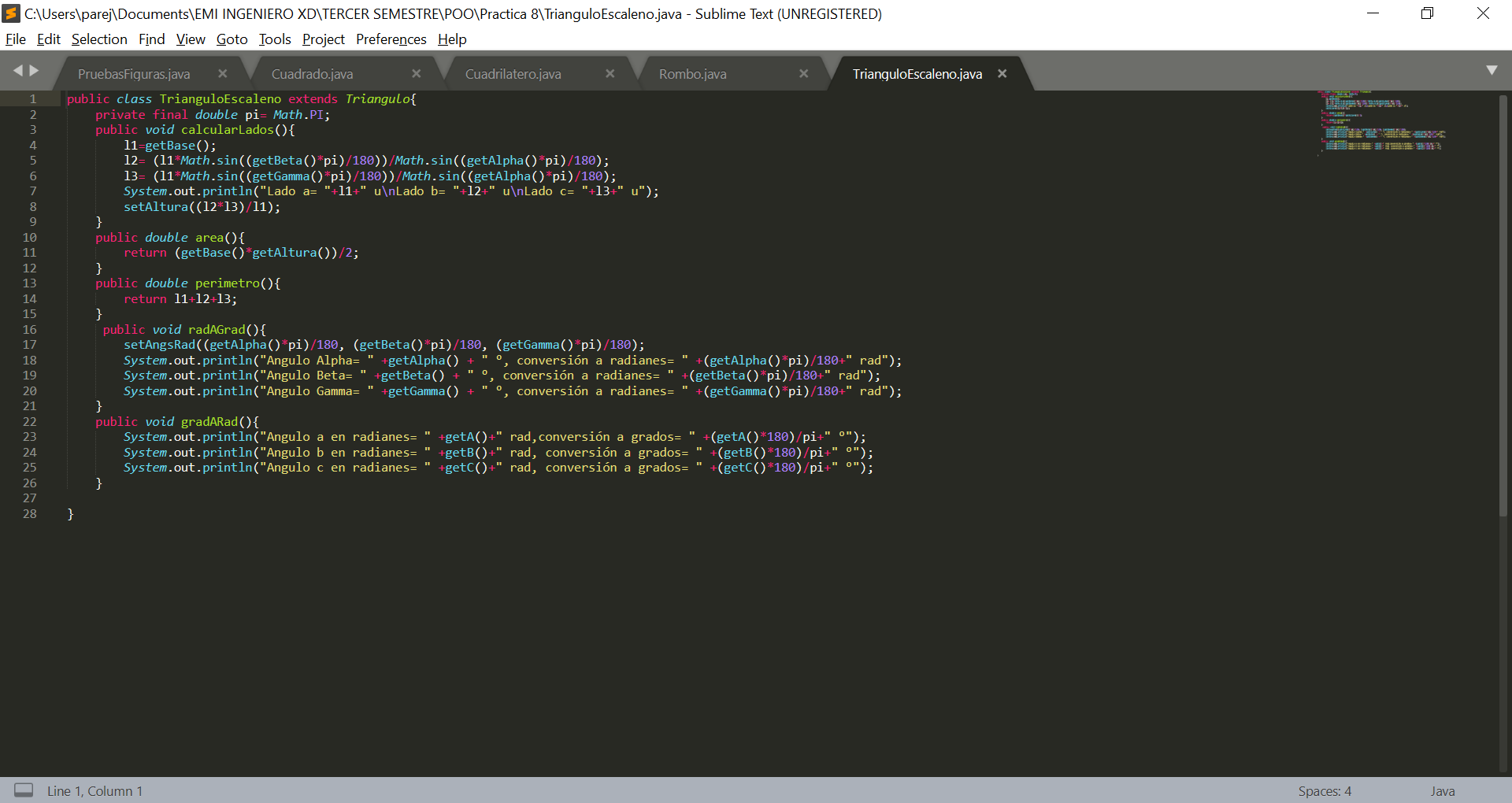
Luego decidimos crear las dos interfaces que nos ayudarían a convertir los ángulos a radianes y los radianes a ángulos, a estas interfaces las nombramos GradosRadianes y RadianesGrados y dentro de ellas instanciamos un método que definiríamos dentro de una clase concreta.

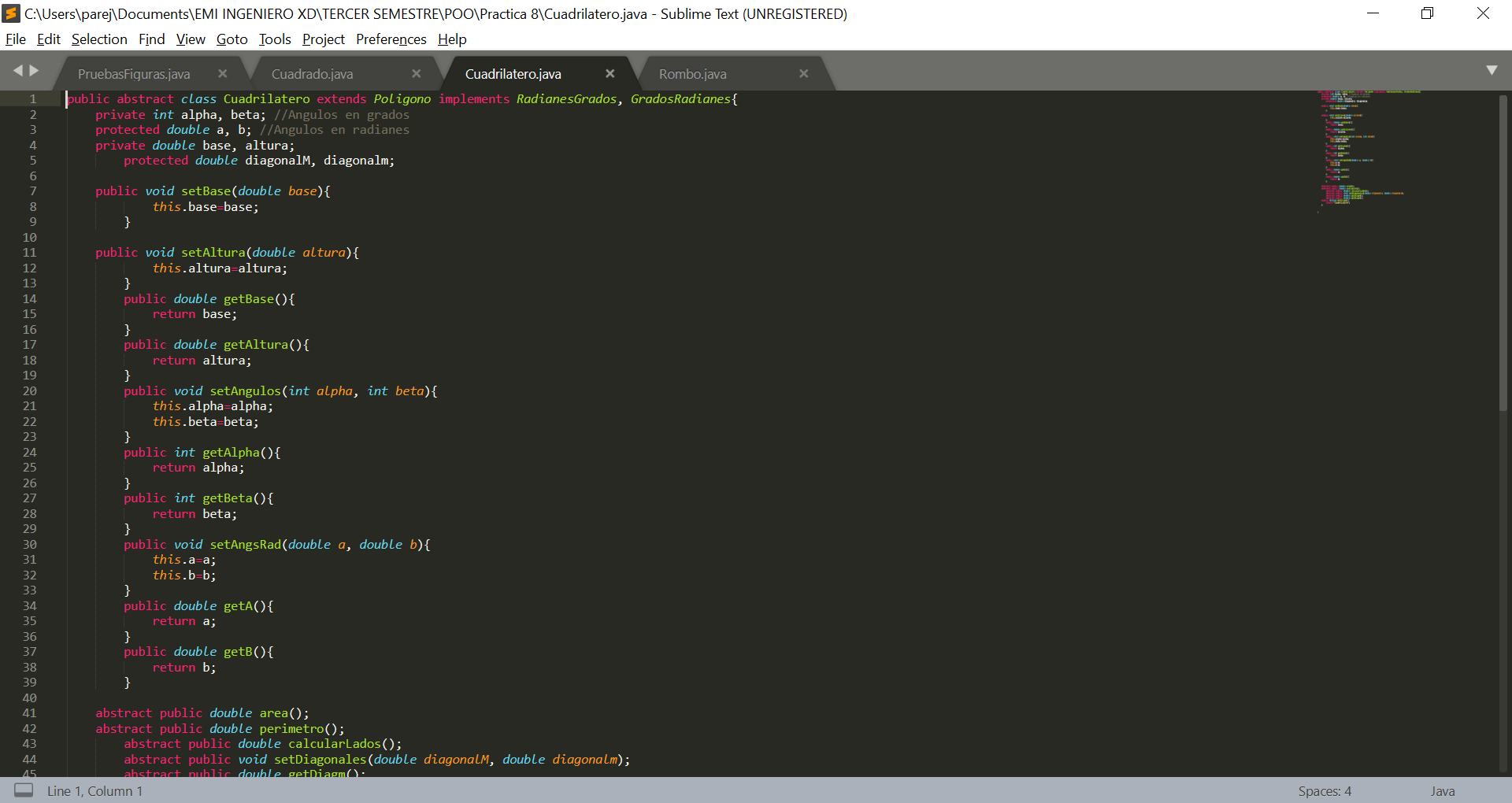
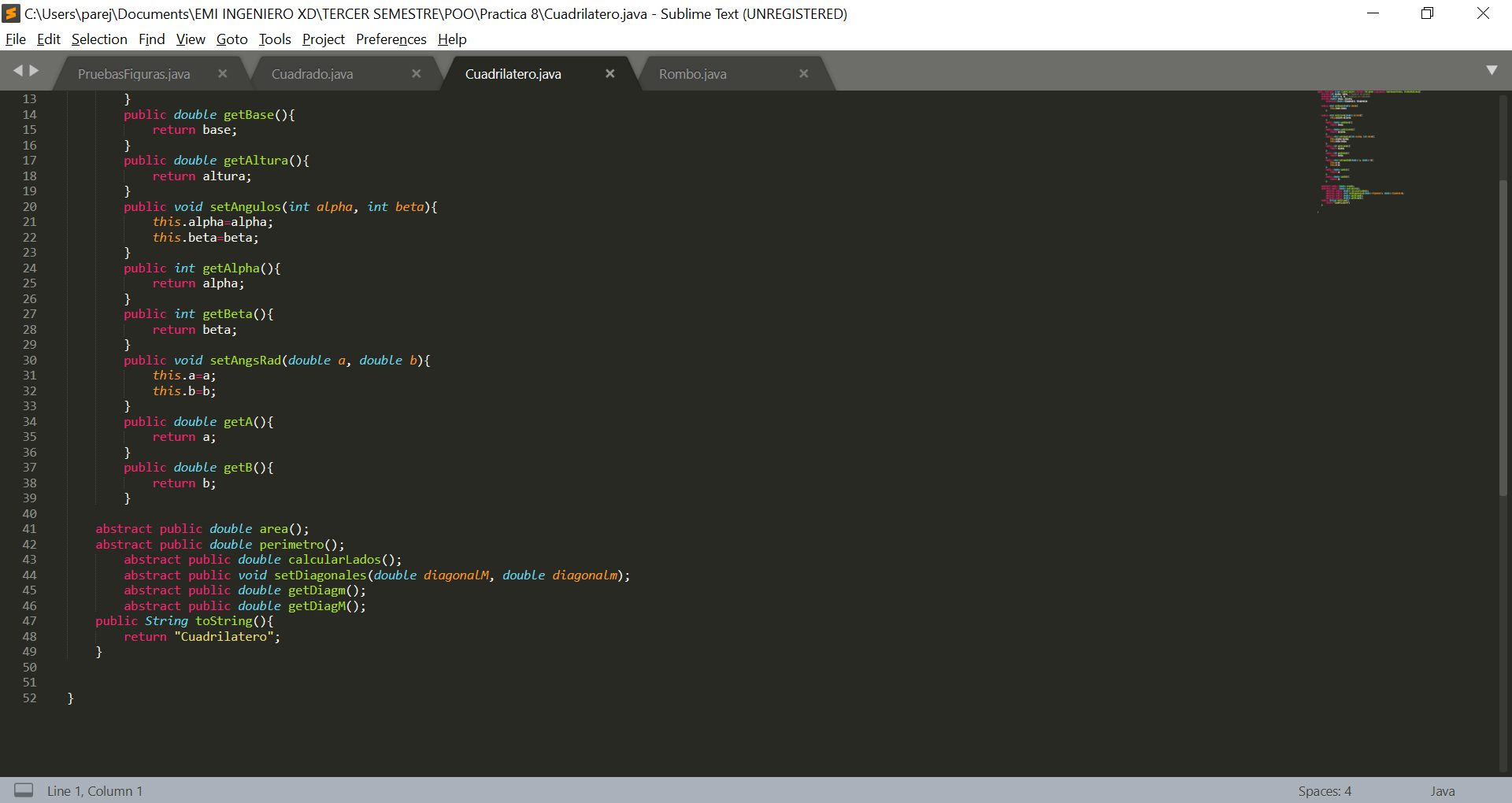


Después hicimos abstracta a la clase triángulo que es una subclase de polígono y por la propiedad de la múltiple herencia, hicimos que esta clase implementara a las interfaces RadianesGrados y GradosRadianes, sin embargo aún no podíamos declarar a los métodos de dichas interfaces en estas clases, porque al ser una clase abstracta no podemos instanciar una interfaz desde la referencia de una clase abstracta, dentro de la misma clase creamos otros atributos adicionales a los que ya se encontraban en los códigos proporcionados por la profesora, anexamos los 3 lados que componen a un triángulo, aunque uno de estos lados sería el atributo base que ya teníamos definido, ya que la base es un lado del triángulo. Hicimos métodos getters y setters de los diferentes atributos, como la altura, base, ángulos en grados y ángulos en radianes, y declaramos un método abstracto llamado CalcularLados, ya que los lados de un triángulo dependiendo su tipo, se pueden calcular de diferente manera a partir de sus ángulos. 

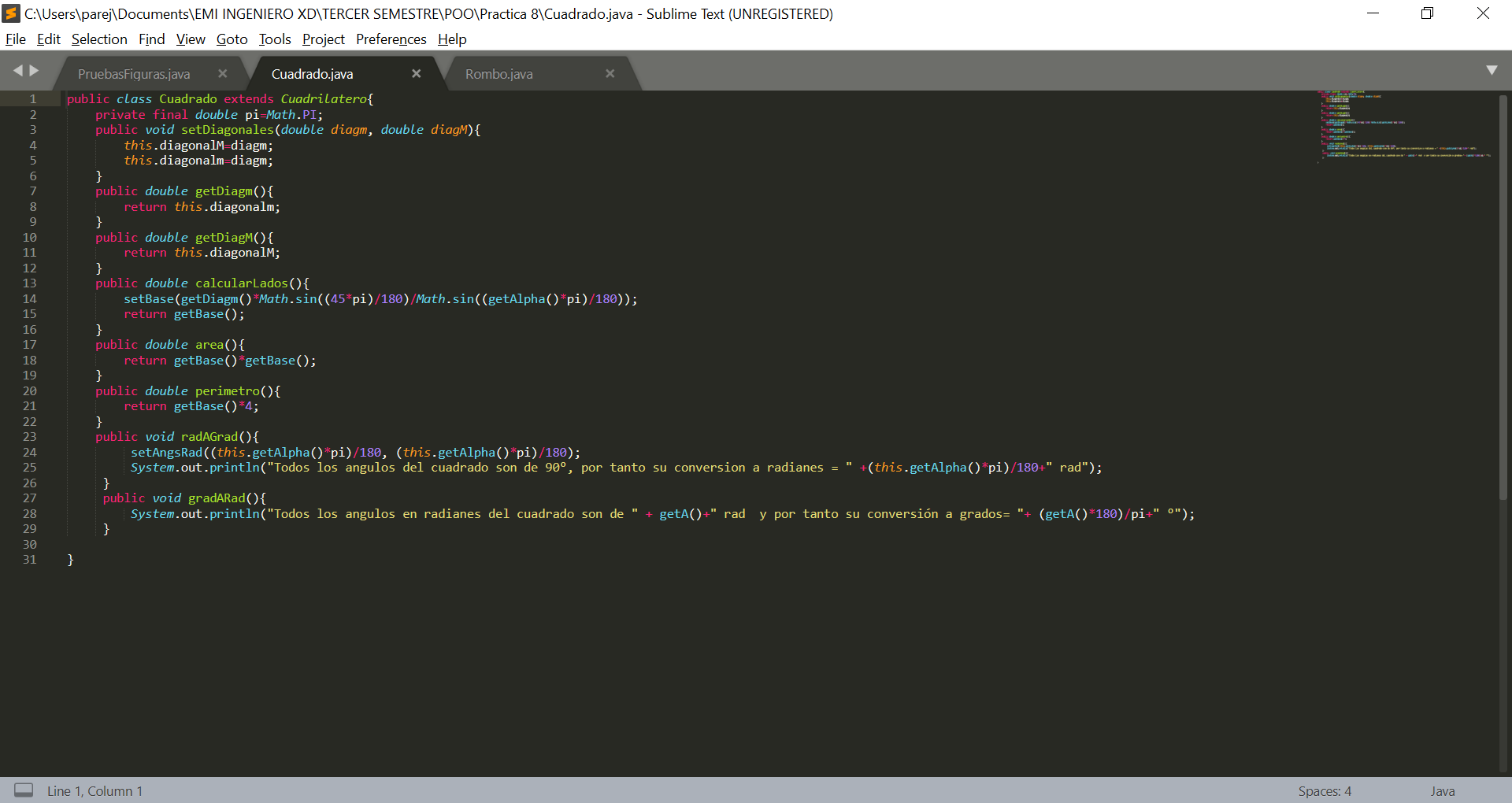
Después modelamos la clase Triángulo Escaleno y dentro de ella, definimos concretamente el método CalcularLados heredado de su clase padre Triángulo, y para calcular cada lado, ocupamos las leyes de senos y además al final de ella, utilizamos el método set altura que utilizaríamos para conocer el área del triángulo, después sobrescribimos los métodos área y perímetro heredados desde la clase polígono y definimos concretamente a los métodos de las interfaces implementadas por la clase abstracta padre de Triángulo Escaleno, que son radAGrad y gradARad, donde una convierte a los ángulos en radianes del triángulo a grados y el otro convierte a radianes a los ángulos en grados del triángulo. Cabe aclarar que para estos métodos utilizamos los métodos getters y setters heredados de la clase padre de Triángulo Escaleno para poder obtener a estos ángulos. Cabe aclarar que para el método CalcularLados, necesitábamos conocer la base del triángulo escaleno, puesto a que si no la conociéramos no podríamos calcular los catetos del triángulo porque sus ángulos permanecerían constantes pero sus dimensiones pueden ser desde un triángulo muy pequeño hasta uno gigante.

Para la clase Cuadrilátero hicimos algo bastante similar a la clase Triángulo, primero la hicimos abstracta gracias a la palabra reservada abstract y por medio de la multiherencia, hicimos que implementara a las interfaces RadianesGrados y GradosRadianes, añadimos 2 atributos adicionales que son las diagonales del cuadrilátero, esto lo añadimos, debido a que no podemos conocer las dimensiones de un cuadrilátero, sin conocer al menos la diagonal que lo compone, ya que en el ejemplo de un cuadrado, sabemos que todos sus ángulos son de 90 grados, pero necesitamos conocer al menos la diagonal que une a sus 2 extremos para poder conocer sus lados, entonces dentro de esta clase abstracta, creamos algunos métodos getters y setters para poder obtener algunos atributos del cuadrilátero, como la altura, la base, los ángulos en forma de grados y los ángulos en forma de radianes, y declaramos además otros métodos abstractos que son los setters y getters de las diagonales así como un método que nos ayudaría a calcular los lados de cada uno de los cuadriláteros modelados a partir de esta clase abstracta, como lo mencionamos en la clase Triángulo, en esta clase tampoco definimos los métodos de las interfaces, puesto a que no se puede declarar un método de una interfaz en una clase abstracta,

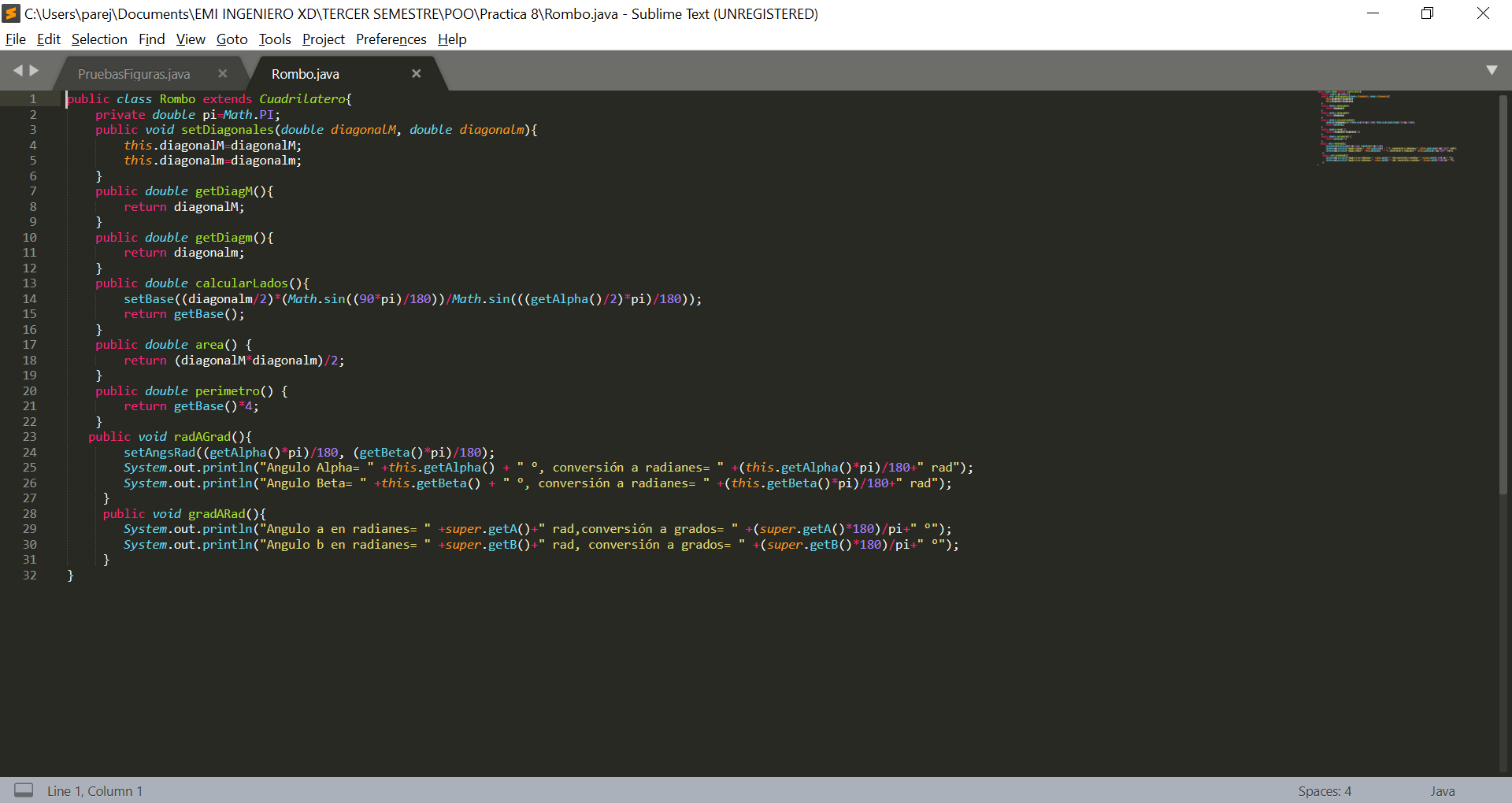
ya que ninguna es un objeto concreto.

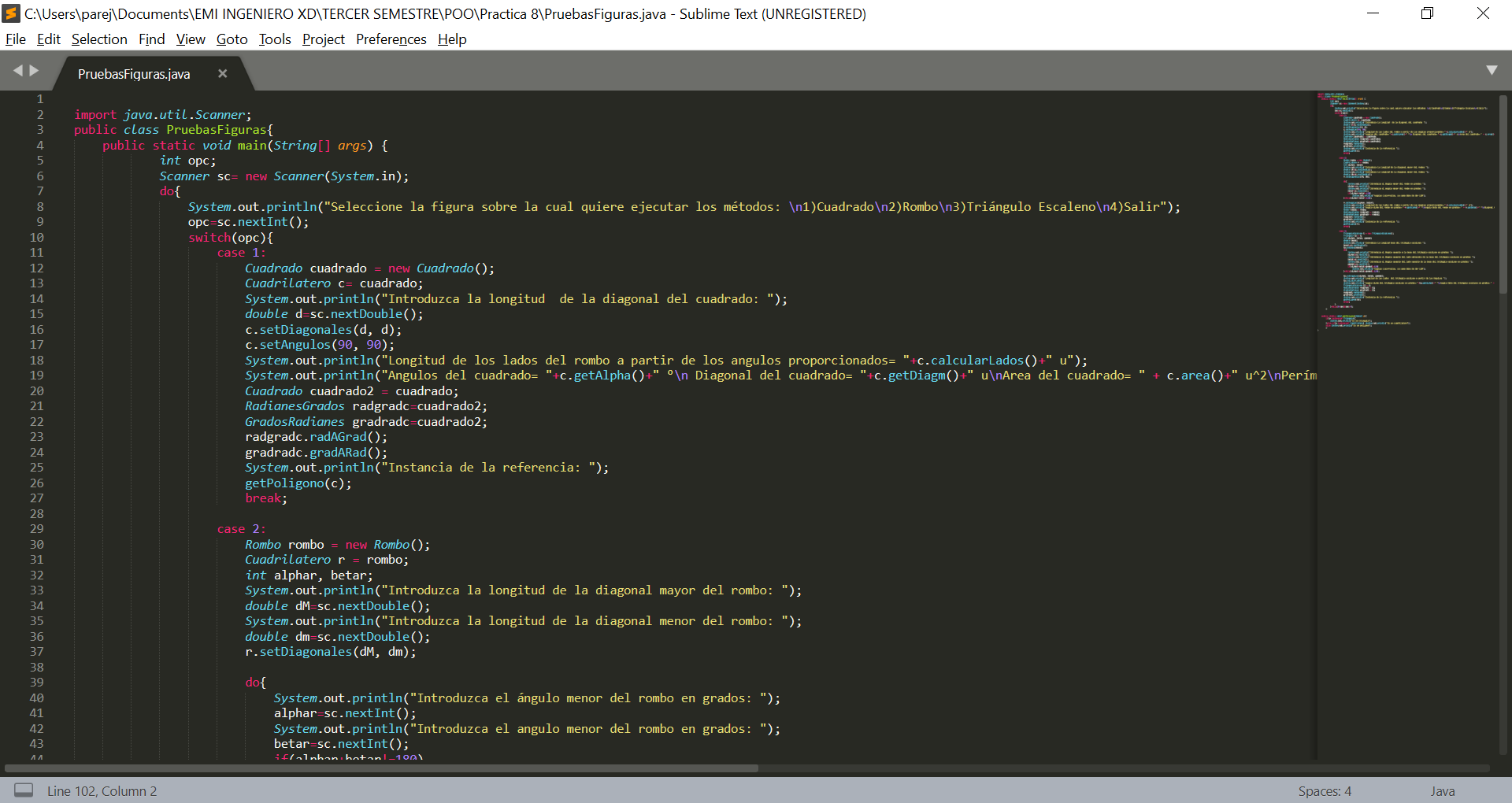


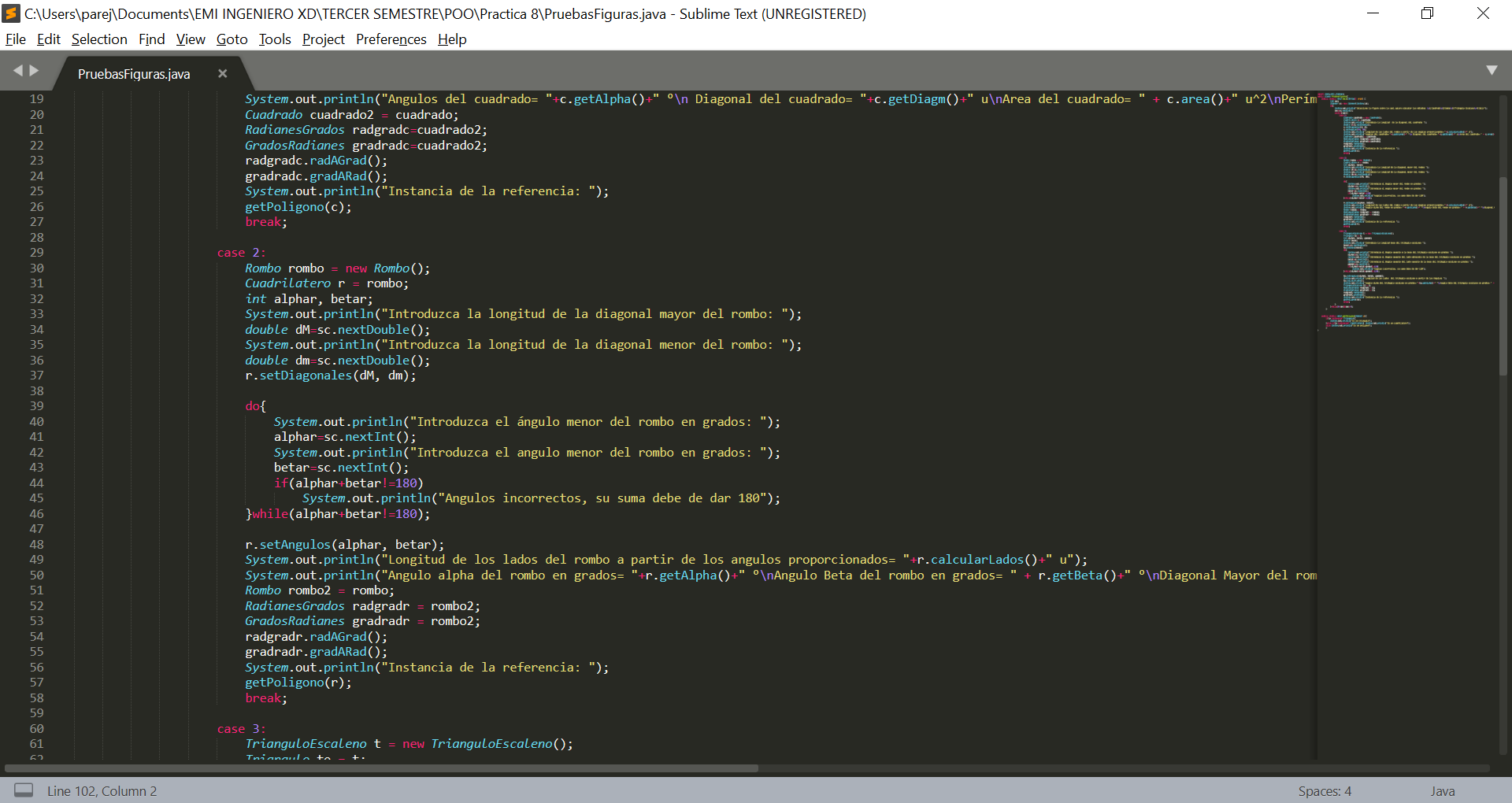
En la clase Cuadrado que es una herencia de la clase abstracta Cuadrilátero, en su método set diagonales, debido a que las diagonales del cuadrado son iguales, estas 2 diagonales las igualamos a uno de los dos parámetros que recibe el método después definimos los demás métodos getters que se definían como abstractos en la clase padre de Cuadrado, y en su método CalcularLados utilizamos las leyes de senos aunque en realidad no son mucho problema, ya que al ser todos ángulos de 90 grados en un cuadrado, al aplicar la ley de senos solo multiplicamos a la diagonal por el seno de 45 grados que es la mitad de 90 y lo dividimos entre el seno de 90 que es el ángulo opuesto a la diagonal del cuadrado. Realizamos una sobreescritura de los métodos perímetro y áreas aplicando las fórmulas que conocemos desde hace mucho tiempo para calcularlas y definimos dentro de esta clase a los métodos de las interfaces GradosRadianes y RadianesGrados de nuevo, y realmente no tenía mayor complejidad transformar los ángulos ya que todos son de 90 grados y sólo aplicamos las fórmulas explicadas en el análisis previo.



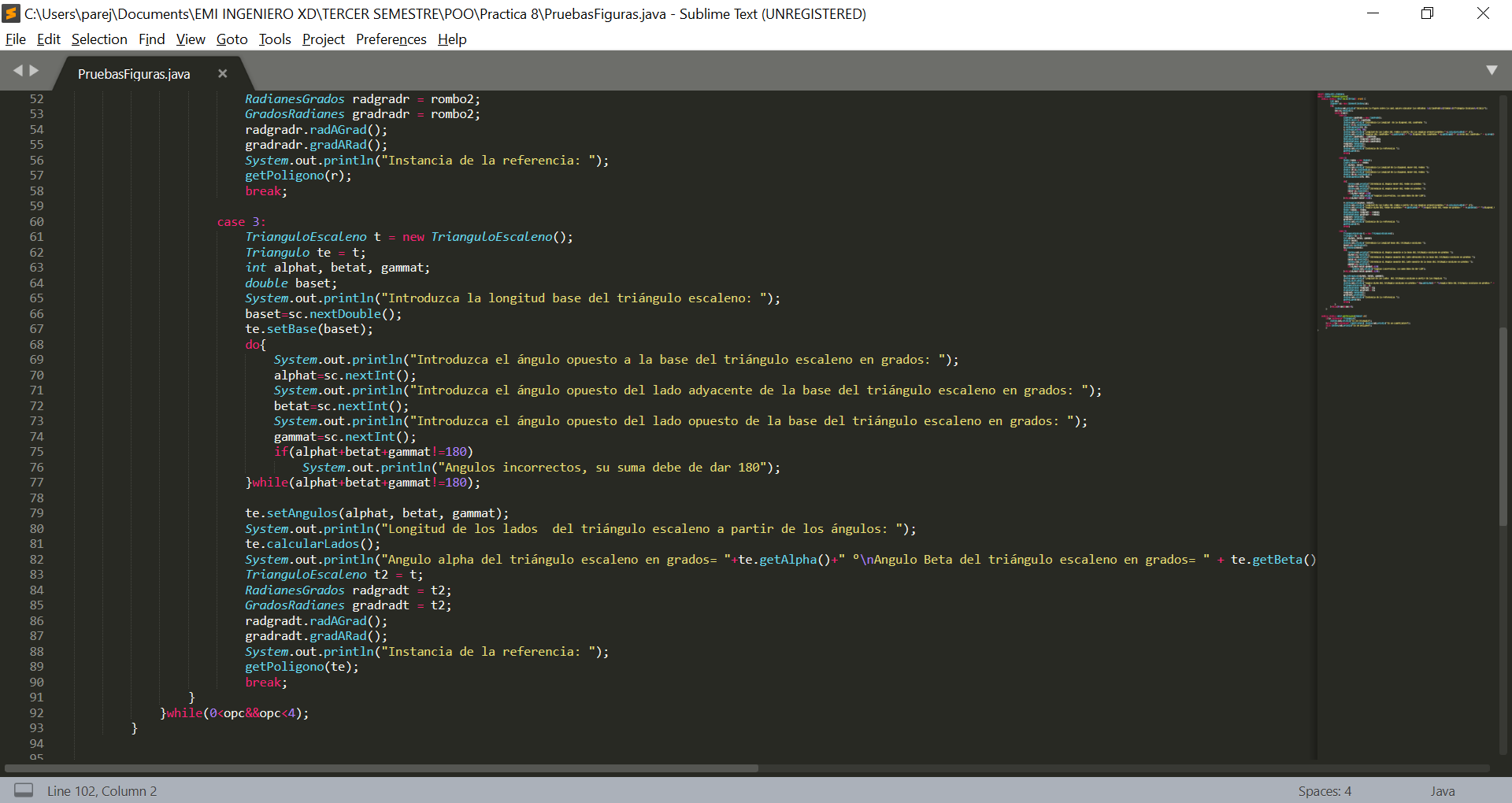
En la clase rombo hicimos la implementación similar a la clase Cuadrado, ya que definimos los métodos getters y setters que en la clase padre eran abstractos, para el cálculo de los lados ahora fue una implementación diferente, también utilizamos la ley de senos para poder obtener los lados del rombo, multiplicando a la diagonal menor por el seno de 90 grados que es su ángulo adyacente sobre el seno del ángulo opuesto a la diagonal menor dividido entre 2, para la definición del área y perímetro utilizamos las fórmulas que hemos visto de toda la vida, donde área de un rombo es igual a la diagonal mayor por la menor entre dos y su perímetro al ser todos sus lados iguales es igual a la longitud de unos de sus lados por cuatro. Y también dentro de esta clase definimos a los métodos de las interfaces GradosRadianes y RadianesGrados.

Dentro de la clase principal que es la clase de Pruebas, hicimos un menú para que el usuario pudiera utilizar los diferentes métodos que creamos en las múltiples clases a partir de las referencias concretas de las 3 figuras que definimos que son el triángulo escaleno, el cuadrado y el rombo, si el usuario selecciona la opción 1 del menú, accede a todas las características de un cuadrado, y creamos una referencia concreta de un cuadrado, pero después en base a las indicaciones de la práctica, creamos ahora una referencia de una clase abstracta cuadrilátero y la igualamos a la referencia concreta que teníamos del cuadrado, y después le preguntamos al usuario por la longitud de la diagonal del cuadrado, y en base a eso, con los diferentes métodos getters y setters, primero al método setdiagonales le pasamos como parámetros al valor ingresado por el usuario dos veces, ya que la diagonal mayor y menor de un cuadrado son iguales, también en el método setangulos, los dos ángulos que tenemos definidos como atributos en la clase Cuadrilátero los igualamos a 90, porque todos los ángulos de un cuadrado son iguales, después con los diferentes métodos getters definidos en las clases cuadrilátero y cuadrado, se imprimen a pantalla los diferentes atributos de un cuadrado, y para poder hacer uso de las interfaces creamos una nueva referencia concreta de un cuadrado y la igualamos a la referencia de la clase abstracta cuadrilátero, y a partir de esta nueva referencia, igualamos 2 interfaces GradosRadianes y RadianesGrados a esta y a partir de la referencia de las interfaces accedemos a sus métodos conviertiendo los ángulos del cuadrado en grados a radianes y viceversa.

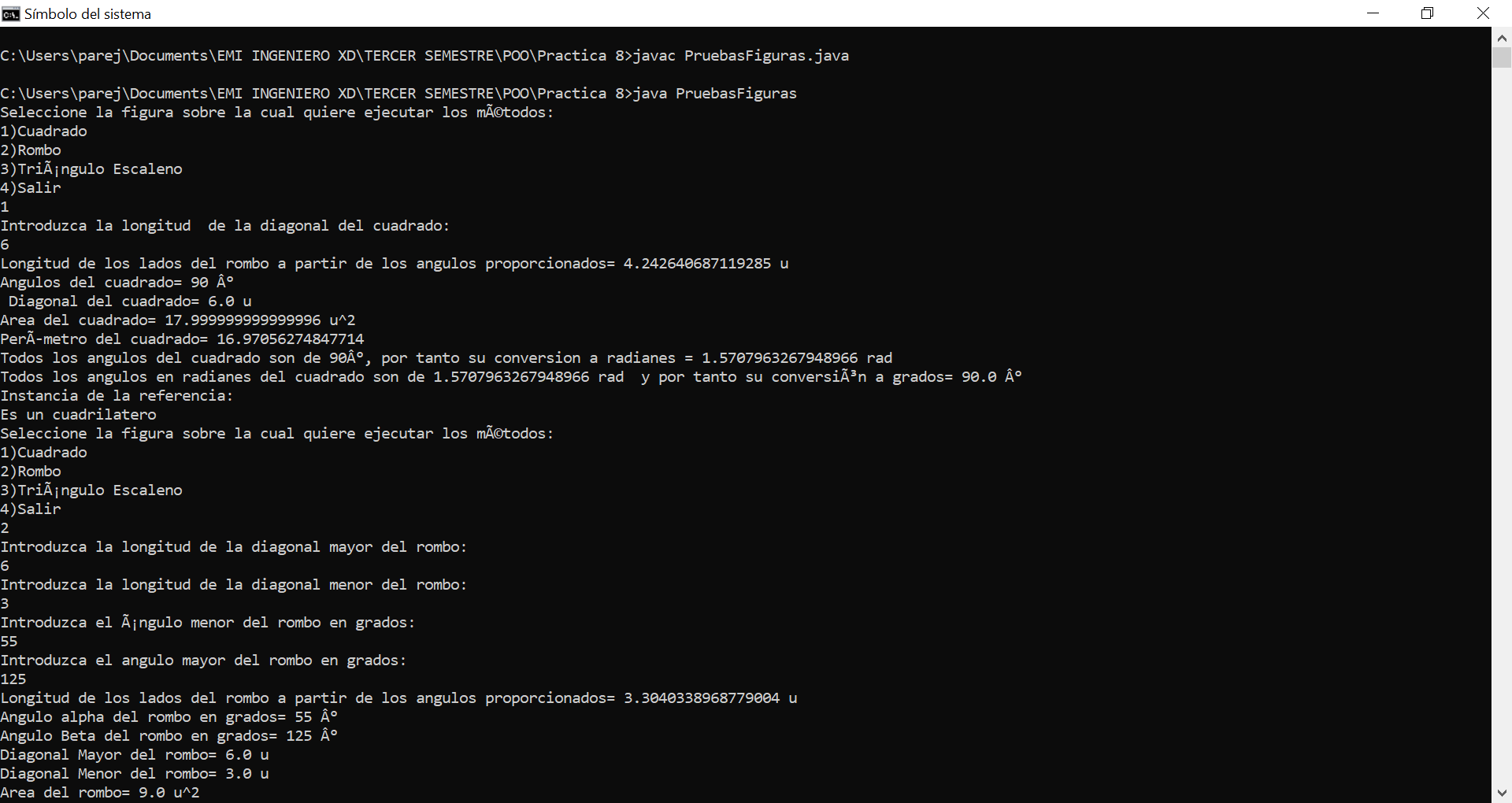


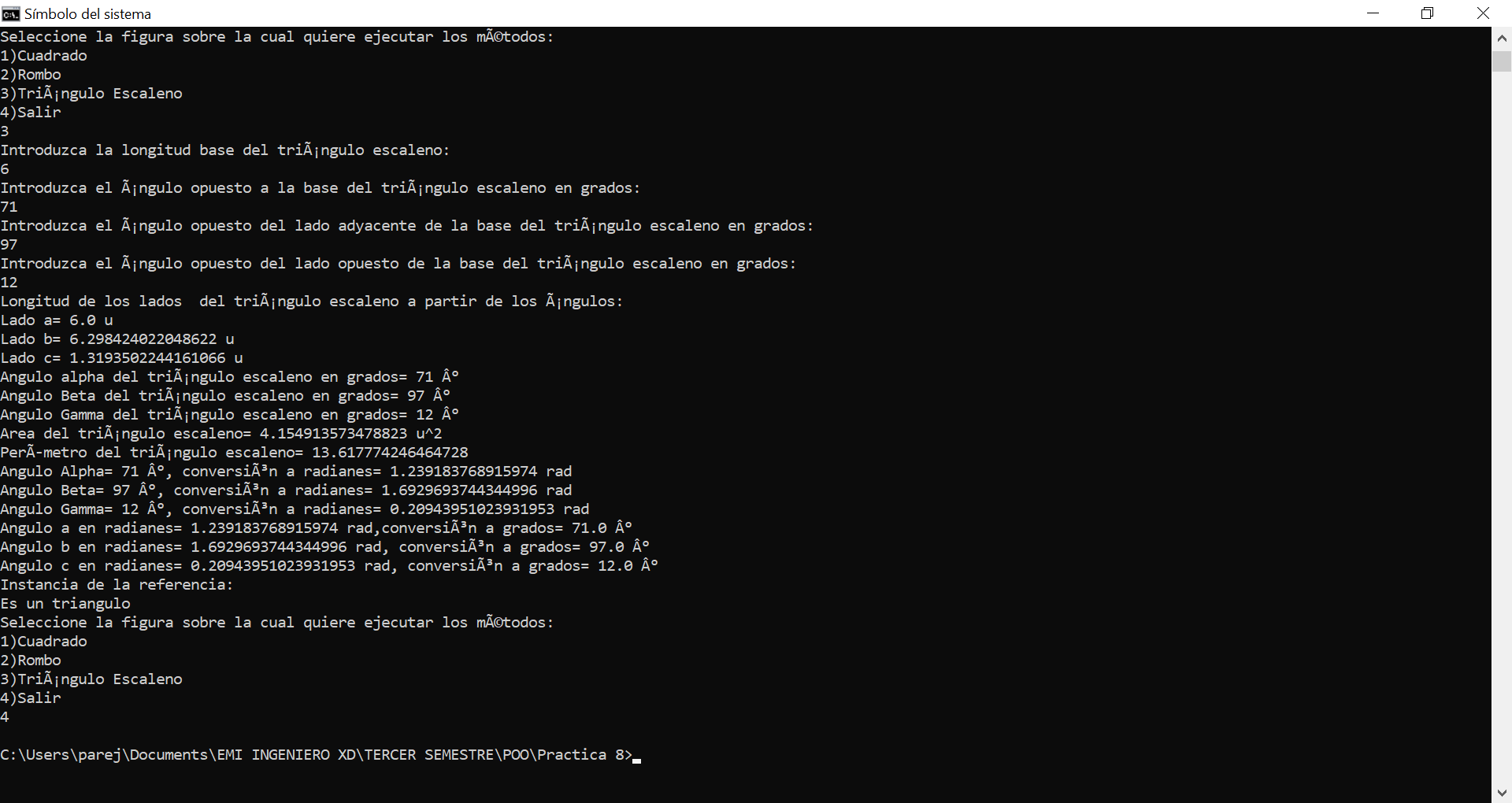
Si el usuario selecciona la opción 2 del menú, entonces accede a las características de un rombo, primero creamos la referencia concreta de un rombo, y después a partir de una clase Cuadrilátero, igualamos a esta a la referencia concreta del rombo, después hacemos las lecturas a partir de información proporcionado por el usuario de los ángulos del rombo y de las diagonales del rombo, pero para hacer las lecturas del rombo hicimos una condición y mientras la suma de los ángulos que ingrese el usuario no sean iguales a 180, el proceso de lectura se repetirá, después con los diferentes métodos getters, hacemos impresión a pantalla de los diferentes atributos del rombo, como su área, perímetro, diagonales y ángulos en grados, después creamos otra referencia concreta de un rombo y la igualamos a la que tenemos de la clase abstracta cuadrilátero y en base a esta referencia, instanciando dos interfaces GradosRadianes y RadianesGrados igualamos a estas a la nueva referencia concreta del rombo y accedemos a sus métodos que convierten a los ángulos expresados en grados a radianes y a los ángulos expresados en radianes a grados. 

Si el usuario selecciona la opción 3 del menú accede a las características de un triángulo escaleno, e hicimos algo bastante similar, que es crear primero una referencia concreta de un triángulo escaleno y después instanciamos a una clase abstracta Triángulo y la igualamos a la referencia concreta del triángulo escaleno, y a partir de esta referencia de la clase abstracta, hacemos las lecturas de sus tres ángulos y de su base, y también tenemos la condición de que si la suma de los 3 ángulos ingresados por el usuario no es igual a 180 el proceso de lectura se repetirá de nuevo. Luego con los diferentes métodos getters, imprimimos a pantalla las características de un triángulo escaleno, y luego creamos una referencia concreta de un nuevo triángulo escaleno y la igualamos a la referencia que teníamos a partir de un cuadrilátero, después instanciamos 2 interfaces (GradosRadianes y RadianesGrados), las referenciamos a la clase concreta de triángulo escaleno, y accedemos a sus métodos.



* **Ejecución del programa**





**Conclusiones**

* Dávila Ortega Jesús Eduardo:

Durante la práctica entiendo como implementar clases abstractas las cuales pueden ayudar a definir una herencia más especializada sin tener la necesidad de hacer una sobrecarga o sobreescritura de métodos, lo cual ayuda a que se vea de una manera más amigable y fácil el código, aunque a veces no lo sea, pero a su vez también vi los inconvenientes que tiene volver una clase que no era abstracta a una abstracta, ya que en nuestro caso empezaron a saltar errores los cuales no sabíamos que iban a suceder pero eran unos errores básico, que son fáciles de resolver, con lo anterior se logró entender como implementar correctamente una clase abstracta cumpliendo el objetivo de la práctica.

* Díaz Hernández Marcos Bryan:

En esta práctica aprendí a implementar el polimorfismo a través de las clases abstractas y del enlace dinámico, que permitió el poder ejecutar método de subclases por medio de clases padres, dando así la posibilidad de cambiar el comportamiento de una clase, así mismo la abstracción se volvió algo más entendible en el aspecto de la implementación y sobre todo que en la elaboración se realizó el análisis para poder comprender cómo utilizarla y resolver el ejercicio de la práctica, cumpliendo con el objetivo de la clase.

* Pareja Ávila Emiliano:

Esta práctica me fue muy útil, ya que revisamos varios de los conceptos de polimorfismo y me ayudó a repasar algunos temas como los de las interfaces que me parecen de mucha utilidad, ya que a partir de ellas puedes realizar métodos que no estén definidos en una clase. También se indagó sobre porqué es muy importante la abstracción en la programación orientada a objetos y por qué es bueno que la utilicemos para desarrollar algún proyecto, ya que a partir de una clase abstracta, podemos desarrollar muchas, como en el ejemplo de las figuras geométricas, a partir de la clase abstracta Triángulo, podemos modelar 3 clases concretas distintas como lo puede ser un triángulo escaleno, un isósceles o uno equilátero. Se cumplieron los objetivos de la práctica, porque pudimos implementar varios conceptos sobre el polimorfismo incluso en una clase que ya habíamos creado anteriormente.

* Vázquez Zavala Oliver Alexis:

Esta práctica me ha dejado de aprendizaje el polimorfismo en la programación orientada a objetos, y en particular como se presenta en Java, en donde mediante una jerarquía de clases es posible obtener un cierto resultado en un método que hereda una subclase y otro resultado distinto en otra subclase, ambas subclases heredando de la misma clase padre, tal es el caso de las áreas de las figuras desarrolladas en esta práctica, además con esta práctica pude conocer para qué sirven, sus diferencias y cómo implementar las clases abstractas y las interfaces en java para la creación de una jerarquía de clases.

**Bibliografía:**

* Augusto S. (2012). Algoritmos a fondo: con implementaciones en C y Java. Buenos Aires : Alfaomega.