Examen parcial

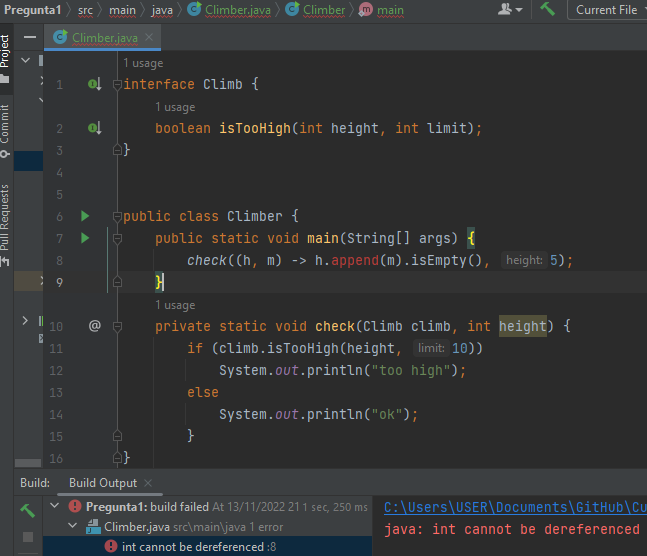
# Pregunta 1

### Parte 1

### 

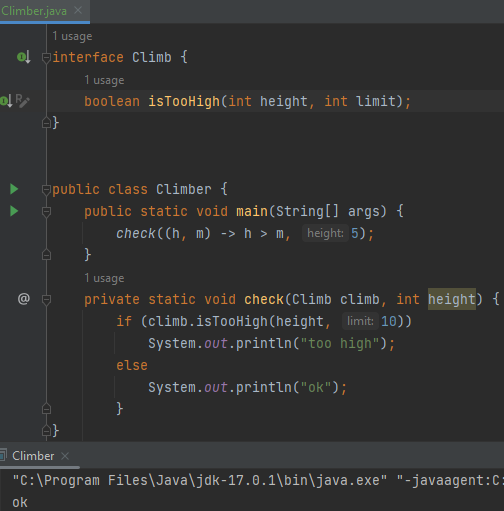
Como se puede ver en la captura de arriba, el resultado es que en consola se imprime *match*. Justifiquemos este resultado. En la línea 12 tenemos la llamada pred.test(panda), la cual procede a comparar el atributo age del objeto *panda* pasado como parámetro al método *check* para compararlo con el valor entero 5, siguiendo lo que indica la expresión lámbda que se pasa como parámetro en la línea 9. El resultado es *match* porque se valida que el valor del atributo *age* del objeto *panda* es menor a 5.

### Parte 2

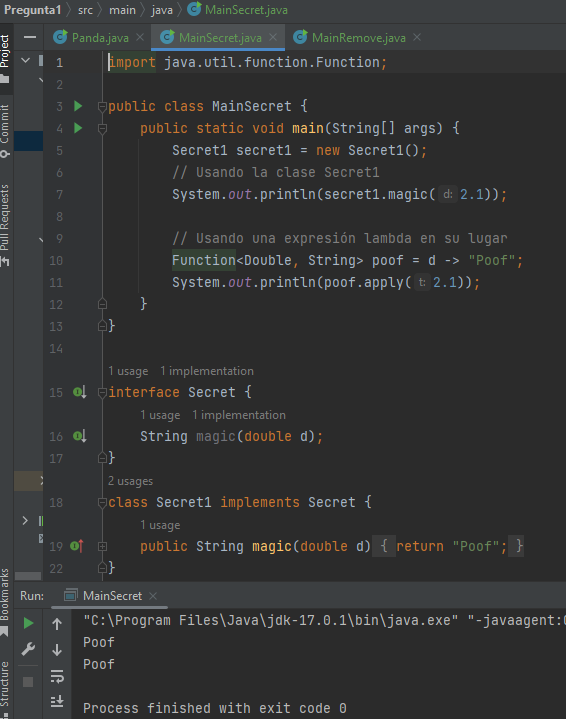


Arriba se muestra el resultado de correr el código tal como está en la pregunta 2. Como se ve, la expresión lambda de la línea 8 se pasa como parámetro de tipo Climb al método estático check(), con lo cual los parámetros *h* y *m* han de ser de tipo entero, pero esto significa que la instrucción h.append(m) es imposible, ya que *h* ni siquiera es un objeto.

Si intuimos el sentido de este pequeño programa, lo que se quiere es comparar dos valores enteros para validar el primero, la altura. Con eso en mente, en la línea 8 cambiamos la expresión lambada para que realice la comparación y retorne su valor de verdad. Como se ve en la siguiente captura, el método ahora funciona como se esperaba, imprimiendo un *ok* dada una altura de 5 menor al límite que es 10.



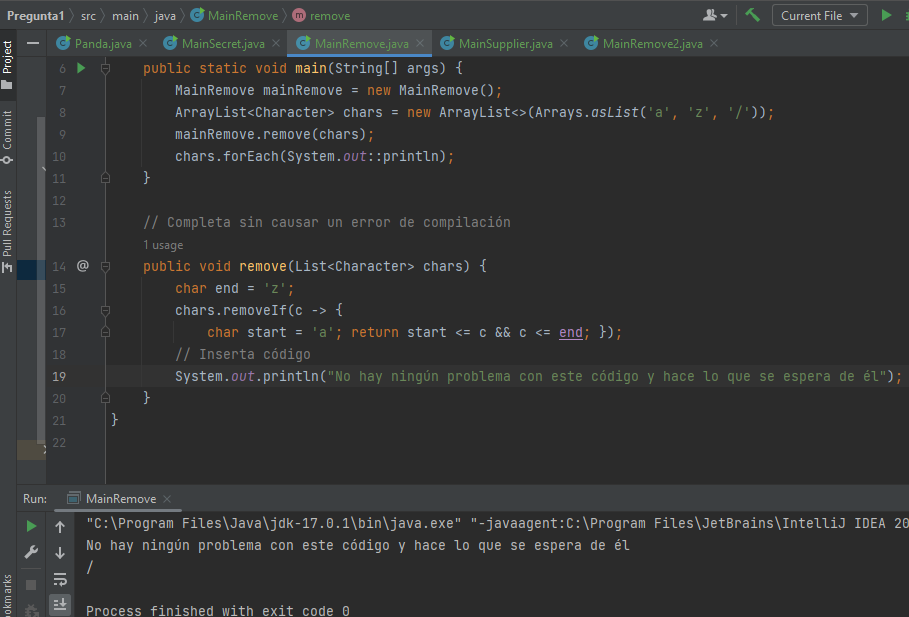
### Parte 3



Como se puede ver en la captura de pantalla de arriba, creamos la clase *MainSecret* y su método principal para probar los resultados de usar la clase Secret1 versus usar una expresión lambda. En la línea 9 se usa un objeto de tipo Secret1 para llamar al método *magic*, pasarle como parámetro un double y obtener la cadena de caracteres *poof*. Pero logramos exactamente la misma funcionalidad usando la expresión lambda definida en la línea 12 y aplicada en la siguiente, y esto se comprueba viendo que el resultado en consola de la línea 9 y de la 13 es el mismo. Por lo tanto, la clase Secret1 puede ser reemplazada por el lambda definido en la línea 12:

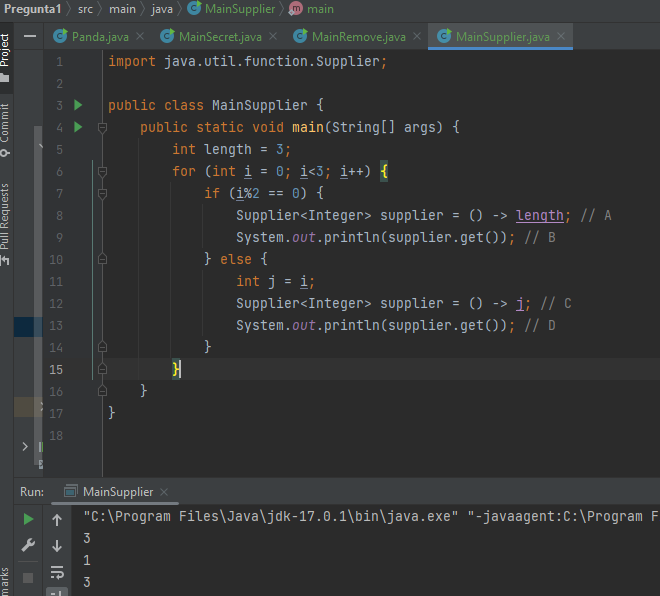
Function<Double, String> poof = d -> "Poof";

### Parte 4



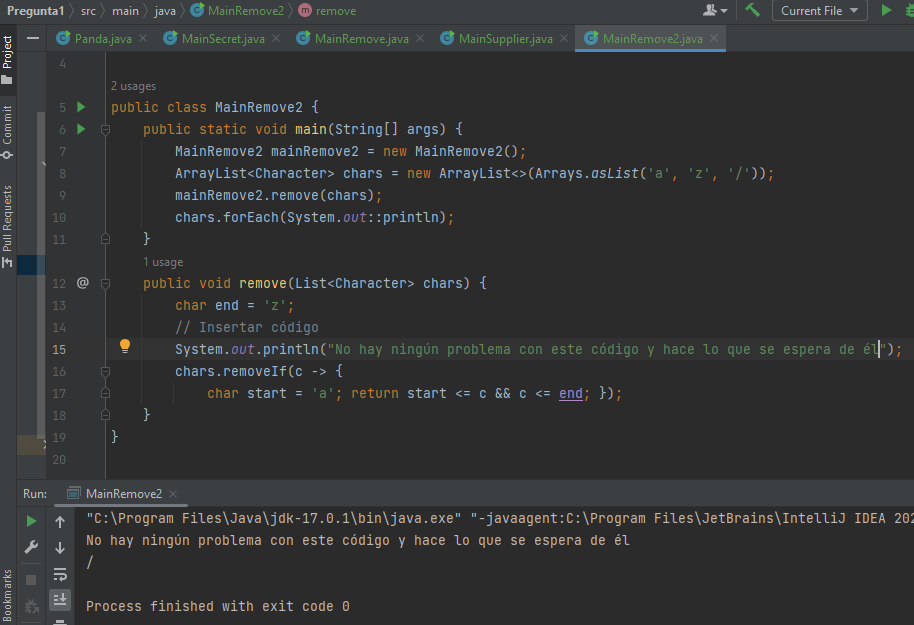
Como se ve en la captura, se agregó la clase MainRemove para compilar y ejecutar el código planteado en la pregunta sin ninguna modificación, pero el código ya parece hacer lo que se supone que debe hacer y se no observó ningún error de compilación. Como eso era lo que se buscaba, eso concluiría la pregunta.

### Parte 5



Como se ve en la captura, tanto en A (la línea 8) como en C (la línea 12) se define una expresión lambda de tipo Supplier. El método que permite llamar este tipo de lambda es *get*, y lo que hace es retornar un valor sin considerar ningún parámetro. El lambda de A retorna el valor de la variable *length* y esto se imprime en B; mientras que el lambda C retorna el valor que copia la variable *j* de la variable de iteración *i* cuando esta es impar, y esto se imprime en D.

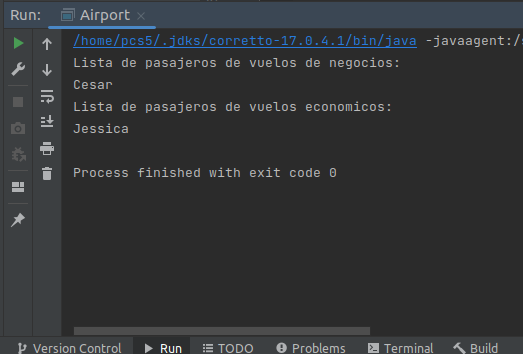
### Pregunta 6



# Pregunta 2

### Pregunta 1

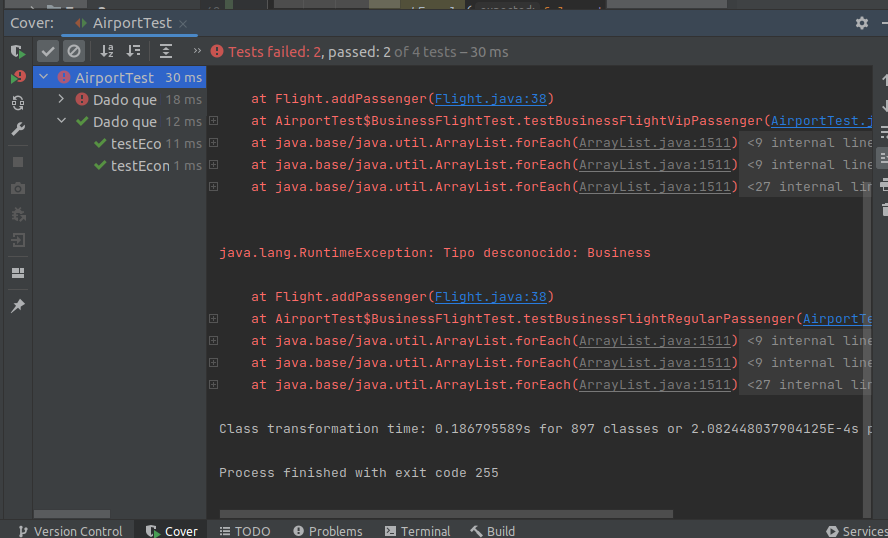
Resultados de la clase Airport en la carpeta Anterior:



Como se observa, el método principal de Airport imprime en consola las listas de pasajeros de vuelos de negocios y de vueltos económicos: Cesar y Jessica. Esto se logra con los bucles de la línea 16 y la 21. Los objetos businessFlight y economyFlight contienen la lista de los pasajeros que van en cada vuelo y el método getPassengersList() retorna dicha lista para luego imprimir los nombres con el método getName() de la clase Passanger.

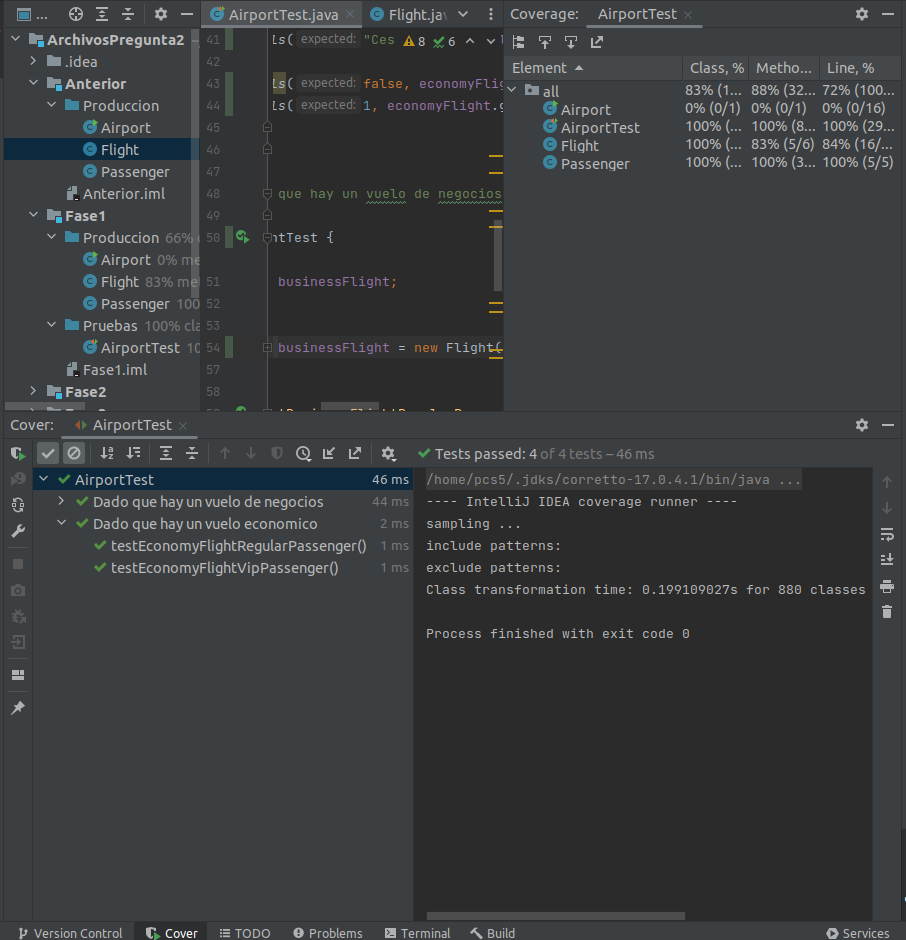
### Pregunta 2

Antes de poder ver los resultados pedidos, vemos que hay un error en el código de pruebas de AirportTest en la carpeta Fase 1/Pruebas.



El error es un tipo de vuelo (flightType) desconocido: Business. Debió estar en español: Negocios.

Arreglado eso, estos son los resultados de las pruebas con cobertura desde IntelliJ IDEA de la carpeta Fase 1/Pruebas:



Todas las pruebas pasaron, pero en la parte superior derecha vemos que la cobertura es del 83 %, con lo cual no es completa. Esto se debe a que la clase Airport no tiene cobertura. Esto es entendible dado que solo contiene un método principal que ejecuta una pequeña demostración de lo que hace el programa; la lógica que queremos testear se encuentra en las clases Flight y Passenger del código de producción.

### Pregunta 3

La necesidad de refactorización de la aplicación está en que, si las reglas de negocio cambian y se añade un nuevo tipo de vuelo como PremiumFight, se tiene que cambiar la implementación del bloque condicional switch en la clase Flight para agregar dicho tipo de vuelo. Esto viola el principio abierto-cerrado de los principios SOLID. La refactorización ayuda a estructurar de mejor manera la implementación sin alterar la funcionalidad, y esto es justamente lo que quiere John. Concretamente, se buscaría reemplazar este código condicional con polimorfismo, de forma que cada tipo de vuelo se implemente en un clase que hereda de la clase Flight. Así ya no será necesario introducir cambios en esta clase si queremos añadir un tipo de vuelo nuevo: solo habrá que crear la clase correspondiente.

# Pregunta 3