Вступ

Мова програмування Java це одна з найпопулярніших об’єктно-орієнтовних мов програмування, але , на жаль, в неї є свої недоліки. Мабуть одним з найголовніших недоліків є потреба в написанні великого шаблонного коду. В цьому проекті я хочу розповісти про бібліотеку Lombok – бібліотеку, яка вирішує цю проблему.

Основна частина

Отож, Lombok це бібліотека яка використовується для мінімізування або взагалі відсутності шаблонного коду. Використовуючи анотації ми можемо зекономити та полегшити читання вихідного коду.

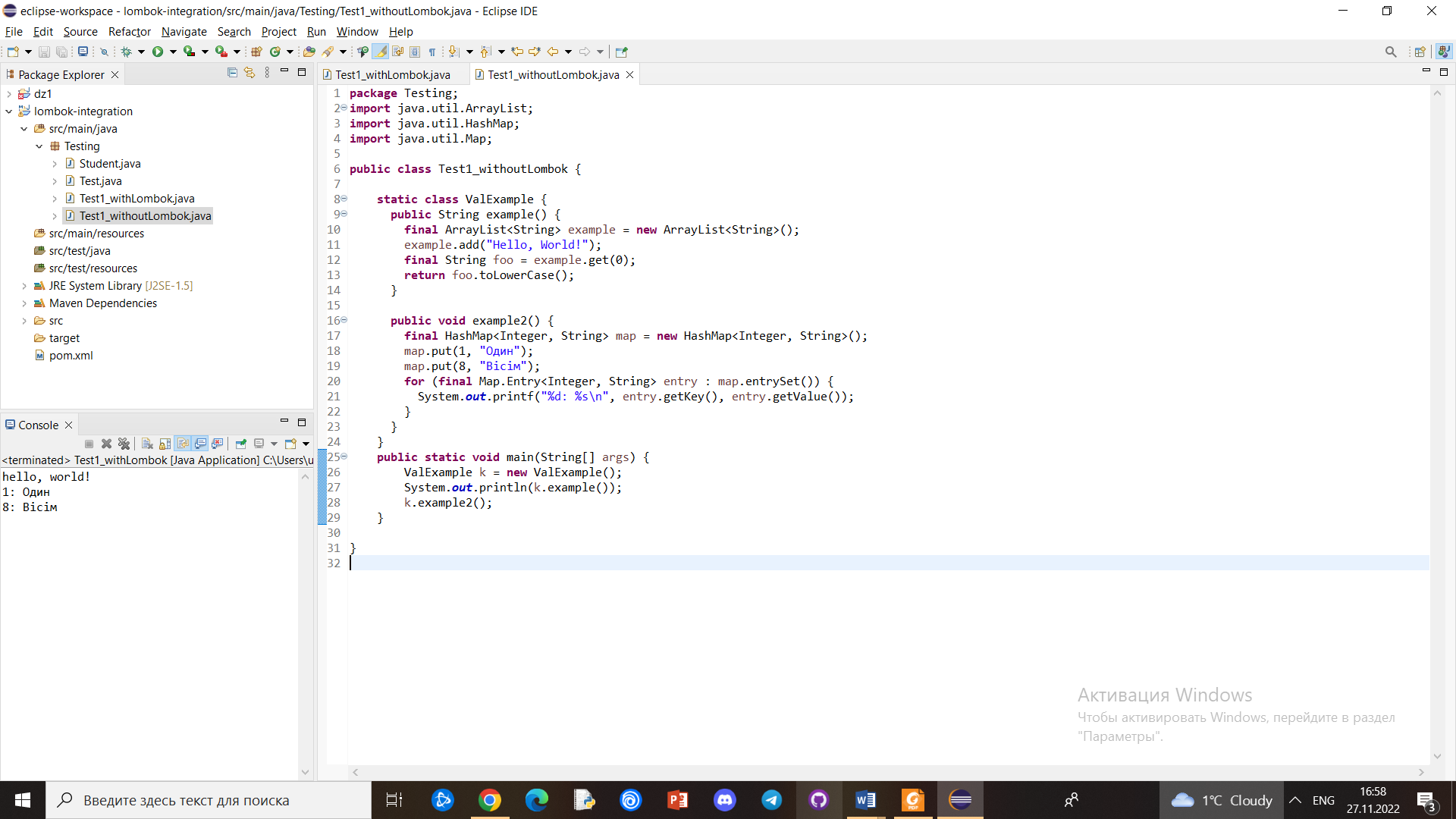
Lombok автоматично підключається до IDE та інструментів створення, щоб оживити нашу програму Java. Може здатись що через це Lombok та IDE виконують одну й ту саму роботу, але це не так. Коли ми використовуємо IDE для генерації цих шаблонних кодів (getters і setters), ми позбавляємо себе від написання геттерів і сеттерів вручну, але вони фактично існують у вихідному коді, що збільшує кількість рядків коду та зменшує зручність обслуговування та читабельність. У той час як проект Lombok додає всі ці шаблонні коди під час компіляції у файл класу замість того, щоб додавати цей шаблонний код у оригінальний вихідний код.

Тепер розберемо які класси, аннотації , типи змінних та інше, які містить Lombok

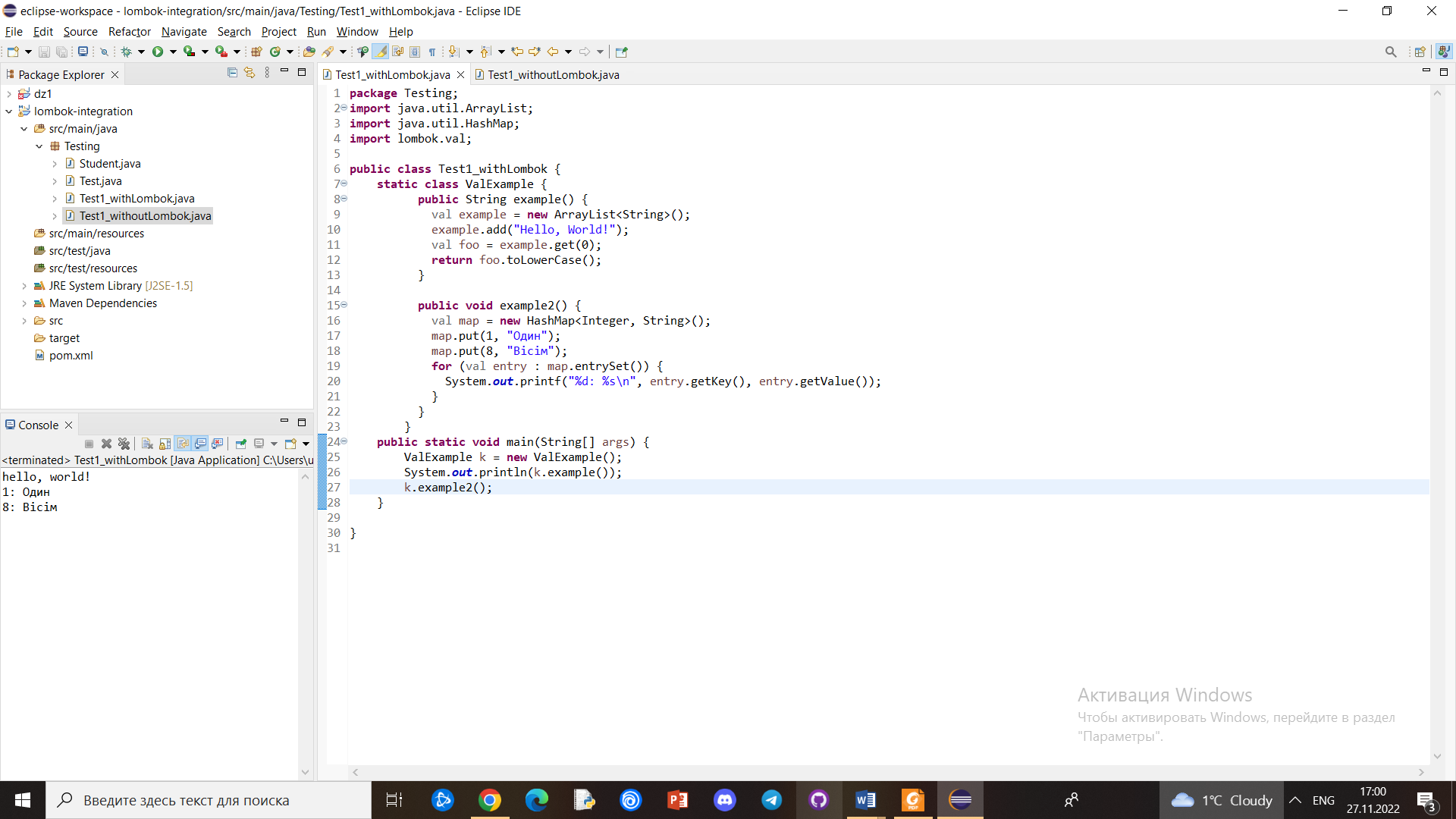
1. Val

Val може використовуватись як тип оголошення локальної змінної замість фактичного написання типу. Після того, як ми її оголосимо, тип буде виведено з виразу ініціалізатора. Локальна змінна також буде остаточною. Ця функція працює лише з локальними змінними та циклами foreach, а не з полями

val фактично є свого роду «типом» і існує як реальний клас у пакеті lombok. Користувач повинен імпортувати його, щоб val працював (або використовуйте lombok.val як тип). Наявність цього типу в оголошенні локальної змінної запускає як додавання ключового слова final, так і копіювання типу виразу ініціалізації, який перезаписує тип «фальшивий» val. Розглянемо приклад використовування(Test1\_withoutLombok та Test1\_withLombok): Test1\_withoutLombok:



Test1\_withLombok :



Бачимо, що тип Val більш варіативніший та універсальний

1. Var

var працює так само, як val, за винятком того, що локальна змінна не позначена як final.

Тип все ще повністю походить від обов’язкового виразу ініціалізатора, і будь-які подальші призначення, хоча й тепер є законними (оскільки змінна більше не є остаточною), не розглядаються для визначення відповідного типу.

Наприклад, var x = "Hello"; x = Color.RED; не працює; тип x вважатиметься java.lang.String і, отже, призначення x = Color.RED не вдасться. Якби тип x був java.lang.Object, цей код був би скомпільований, але це не працює var.

1. @NonNull

@NonNull може використовуватись для компонента запису або параметра методу чи конструктора. Це призведе до того, що lombok створить для вас оператор перевірки нуля.

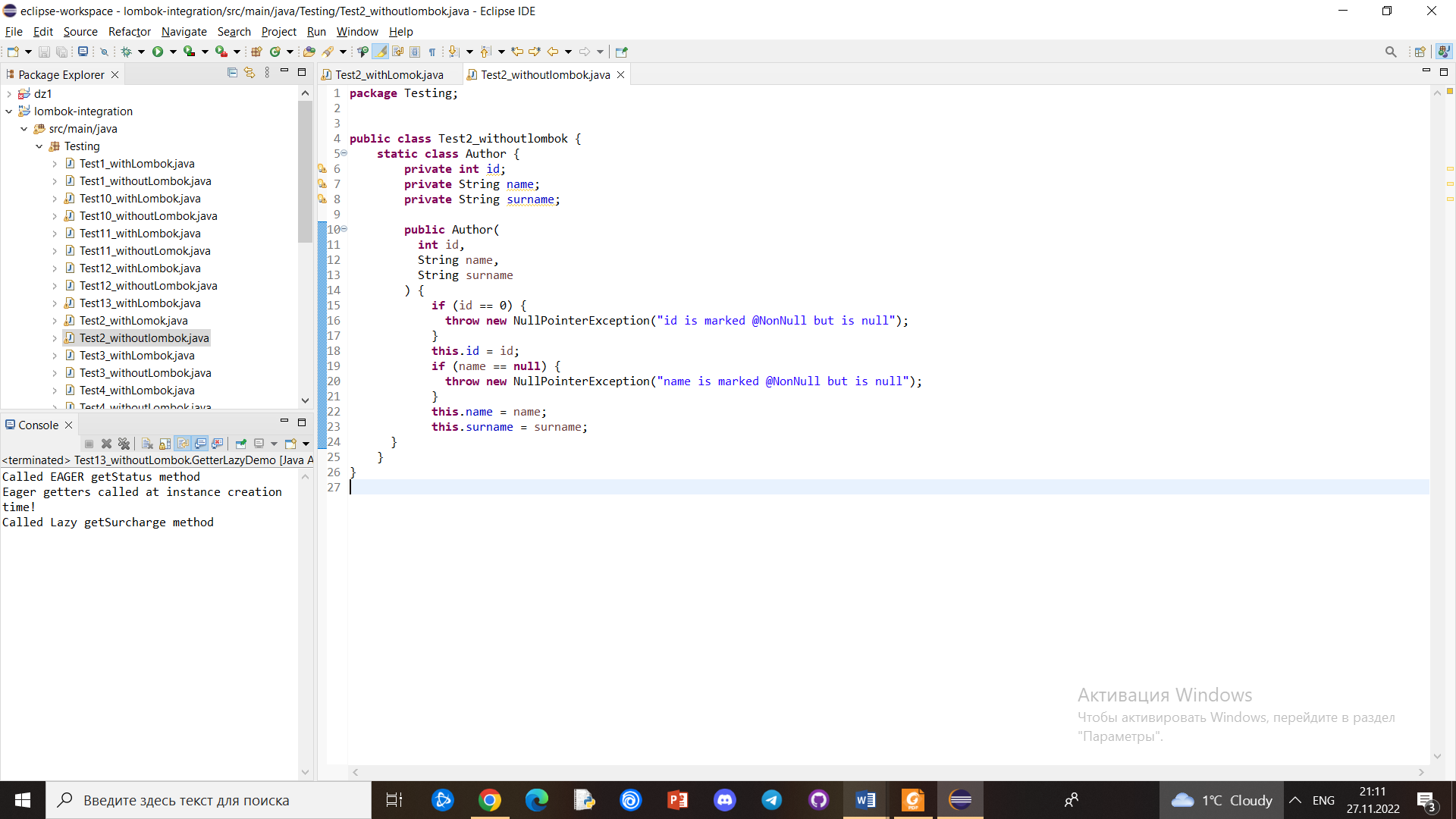
Lombok завжди розглядав різні анотації, які зазвичай називаються @NonNull у полі, як сигнал для генерації нульової перевірки, якщо Lombok генерує цілий метод або конструктор для вас, наприклад, через @Data. Однак використання власного @lombok.NonNull lombok для параметра або компонента запису призводить до вставки нульової перевірки у верхній частині цього методу.

Перевірка на нуль виглядає так, if (param == null) throw new NullPointerException("параметр позначено як ненульовий, але є нульовим"); і буде вставлено в самому верху вашого методу. Для конструкторів нульова перевірка буде вставлена ​​одразу після будь-яких явних викликів this() або super(). Для компонентів запису перевірку нуля буде вставлено в «компактний конструктор» (той, який взагалі не має списку аргументів), який буде згенеровано, якщо у вас немає конструктора. Якщо конструктор записаний у розширеній формі (з параметрами, які точно відповідають вашим компонентам), тоді нічого не станеться – натомість доведеться анотувати параметри цього конструктора розширеної форми.

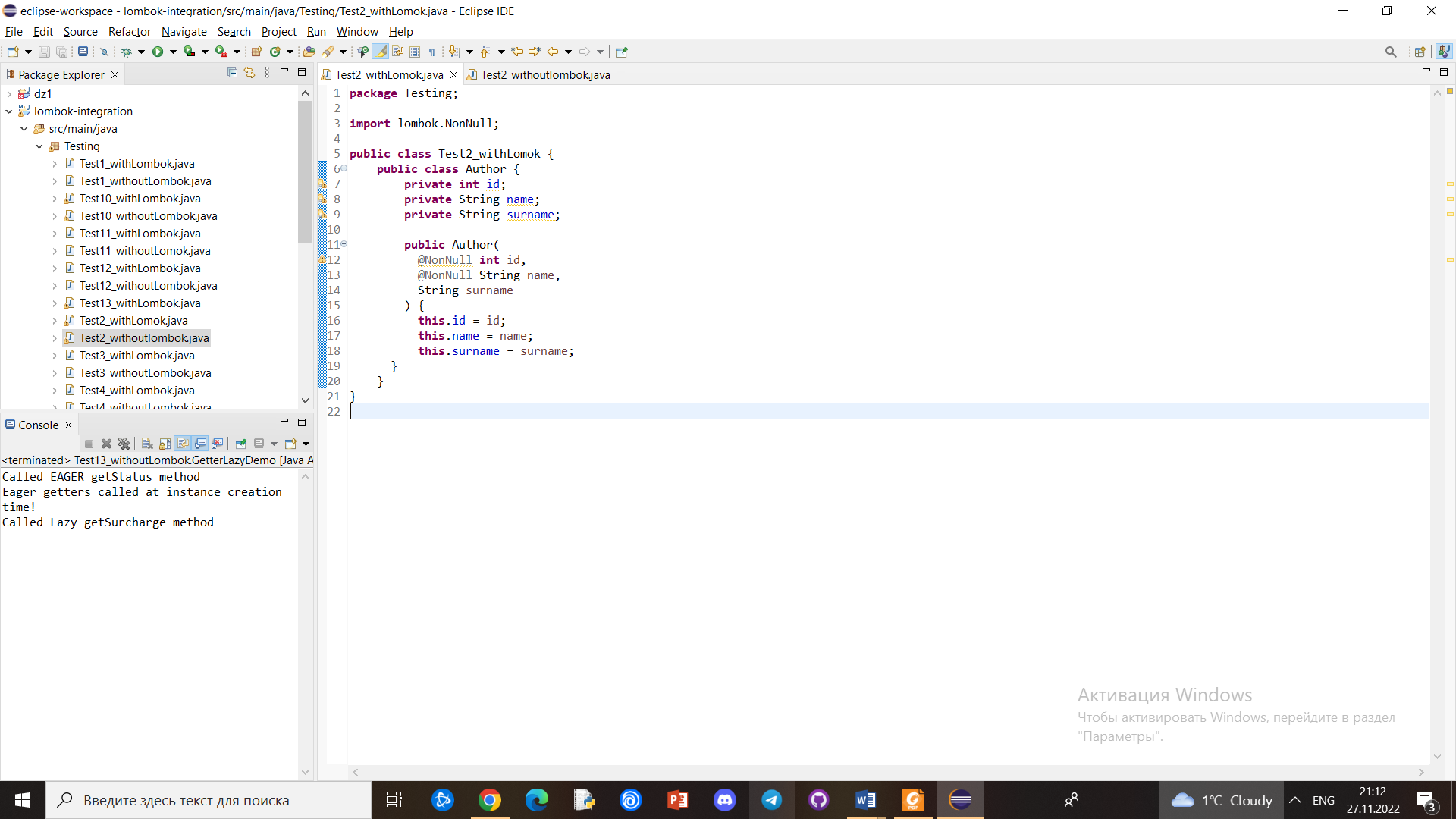
Якщо нульова перевірка вже присутня вгорі, додаткова нульова перевірка не буде створена.

Розглянемо застосування(Test2\_withoutLombok та Test2\_withLombok): Test2\_withoutLombok:

Test2\_withoutLombok:



Test2\_withLombok:



1. @Cleanup

можна використовувати @Cleanup, щоб забезпечити автоматичне очищення даного ресурсу до того, як шлях виконання коду вийде з вашої поточної області. Робиться це анотуючи будь-яке оголошення локальної змінної за допомогою анотації @Cleanup так:

@Cleanup InputStream in = new FileInputStream("some/file");

У результаті в кінці області, в якій ви перебуваєте, викликається in.close(). Цей виклик гарантовано виконується за допомогою конструкції try/finally. Подивіться на приклад нижче, щоб побачити, як це працює.

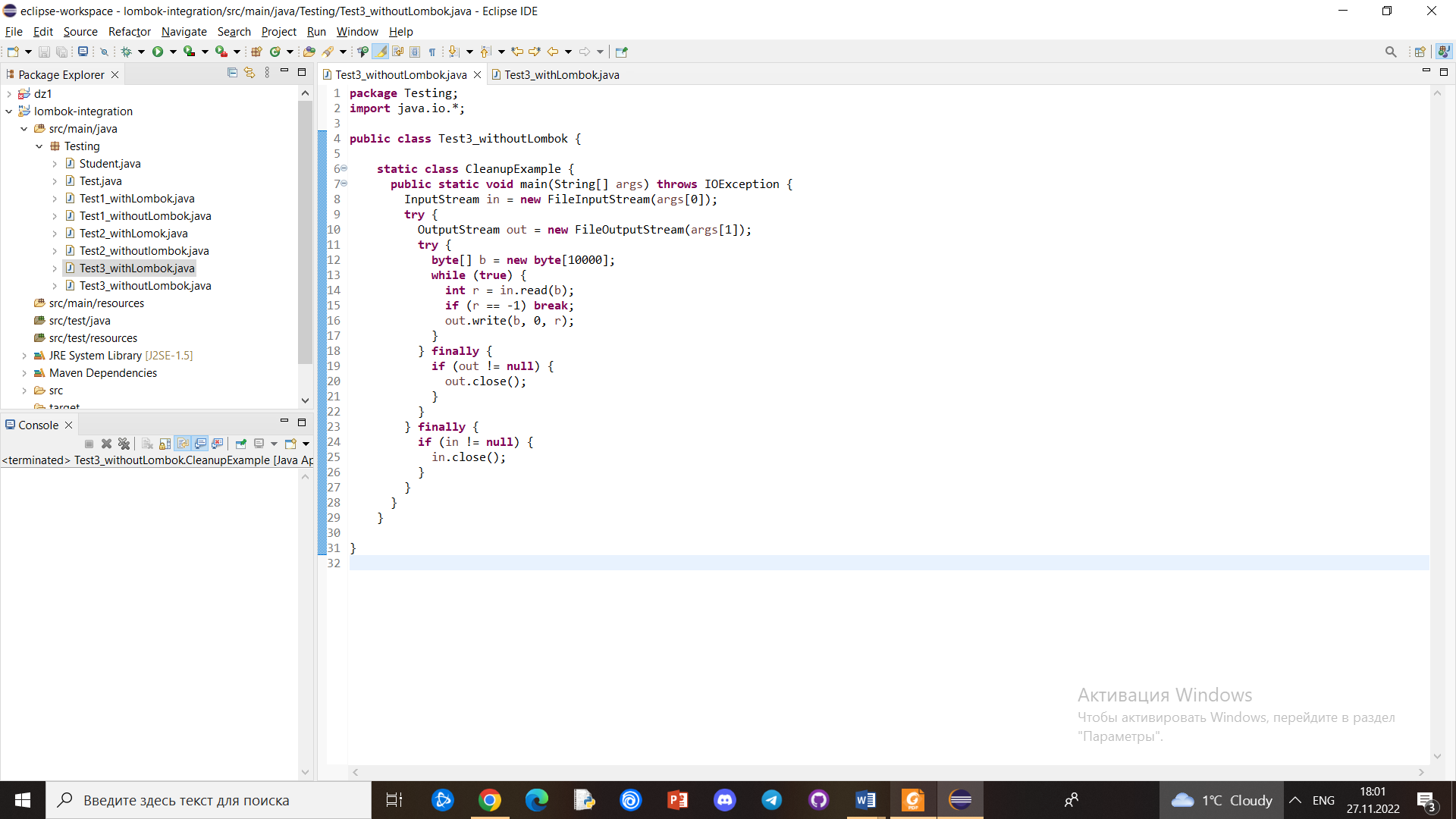
Якщо тип об’єкта, який треба очистити, має не метод close(), а якийсь інший метод без аргументів, можна вказати назву цього методу так:

@Cleanup("dispose") org.eclipse.swt.widgets.CoolBar bar = new CoolBar(parent, 0);

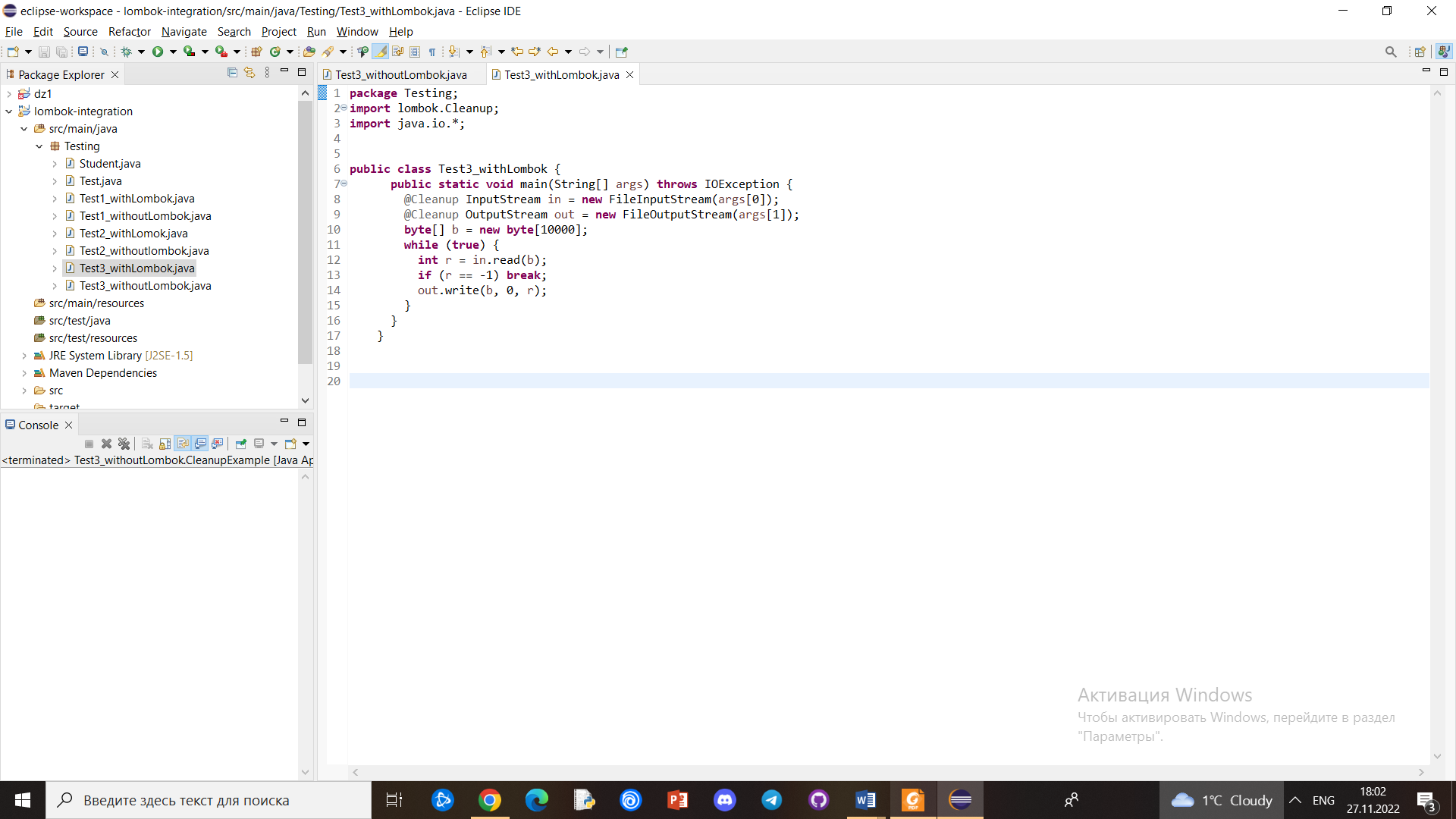
За замовчуванням передбачається, що метод очищення є close(). Метод очищення, який приймає 1 або більше аргументів, не можна викликати через @Cleanup.

Розглянемо застосування(Test3\_withoutLombok та Test3\_withLombok): Test3\_withoutLombok:

Test3\_withoutLombok:



Test3\_withLombok:



1. @Getter and @Setter

Ви можете анотувати будь-яке поле за допомогою @Getter та/або @Setter, щоб дозволити lombok автоматично генерувати геттер/сеттер за замовчуванням.

Геттер за замовчуванням просто повертає поле та називається getFoo, якщо поле називається foo (або isFoo, якщо тип поля логічний). Налаштувач за замовчуванням називається setFoo, якщо поле називається foo, повертає void і приймає 1 параметр того самого типу, що й поле. Він просто встановлює це значення для поля.

Згенерований метод getter/setter буде відкритим, якщо ви явно не вкажете AccessLevel, як показано в прикладі нижче. Рівні легального доступу: ПУБЛІЧНИЙ, ЗАХИЩЕНИЙ, ПАКЕТ і ПРИВАТНИЙ.

Ви також можете додати до класу анотацію @Getter та/або @Setter. У цьому випадку це так, ніби ви анотуєте всі нестатичні поля в цьому класі анотацією.

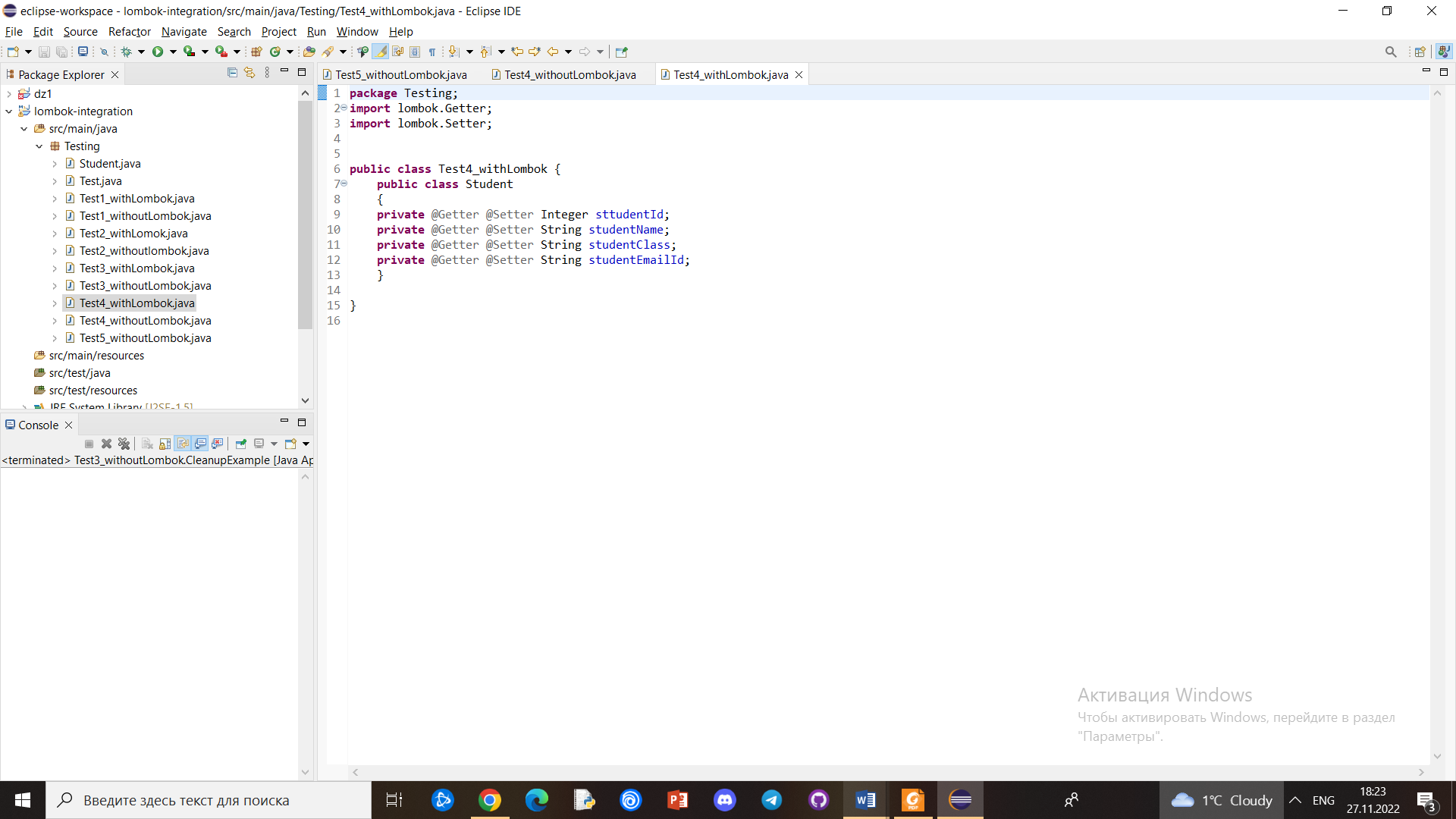
Ви завжди можете вручну вимкнути створення геттера/сеттера для будь-якого поля за допомогою спеціального рівня доступу AccessLevel.NONE. Це дозволяє змінити поведінку анотації @Getter, @Setter або @Data для класу.

Щоб додати анотації до згенерованого методу, ви можете використати onMethod=@\_\_({@AnnotationsHere}); щоб додати анотації до єдиного параметра згенерованого методу налаштування, ви можете використати onParam=@\_\_({@AnnotationsHere}).

Розглянемо застосування(Test4\_withoutLombok та Test4\_withLombok): Test4\_withoutLombok:



Test4\_withLombok:



Бачимо дуже сильну економію коду

1. @ToString

Анотування класу за допомогою @ToString змусить lombok створити реалізацію методу toString(). Ви використовуєте параметри конфігурації, щоб вказати, чи потрібно включати назви полів, але в інших випадках формат є фіксованим: ім’я класу, за яким слідують дужки, що містять поля, розділені комами, наприклад. MyClass(foo=123, bar=234).

Установивши для параметра includeFieldNames значення true, ви можете додати трохи ясності (але також досить довжини) виводу методу toString().

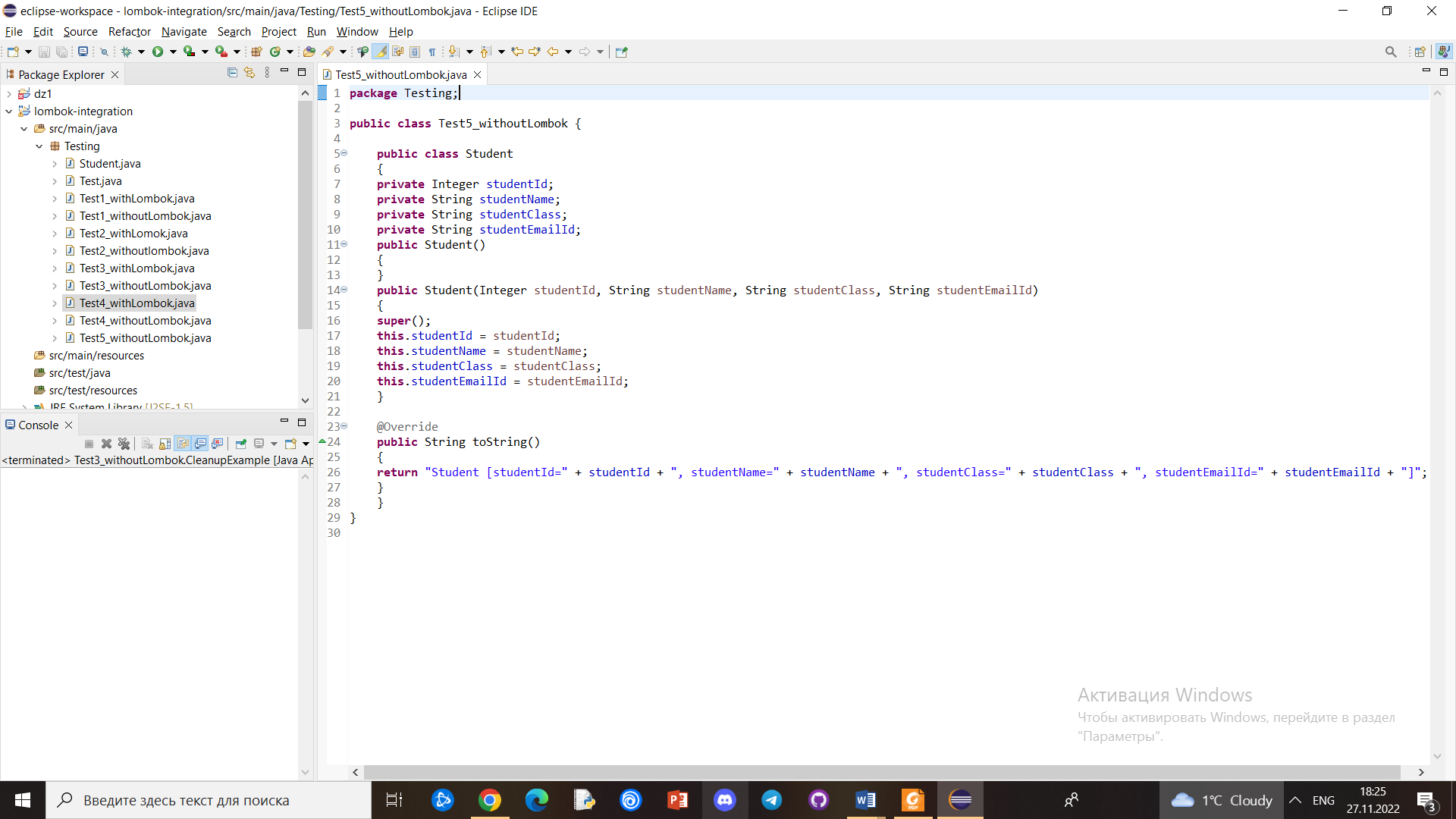
За замовчуванням друкуються всі нестатичні поля. Якщо ви хочете пропустити деякі поля, ви можете примітити ці поля за допомогою @ToString.Exclude. Крім того, ви можете точно вказати, які поля ви хочете використовувати, використовуючи @ToString(onlyExplicitlyIncluded = true), а потім позначаючи кожне поле, яке ви хочете включити, @ToString.Include.

Встановивши для callSuper значення true, ви можете включити вихідні дані реалізації суперкласу toString до вихідних даних.

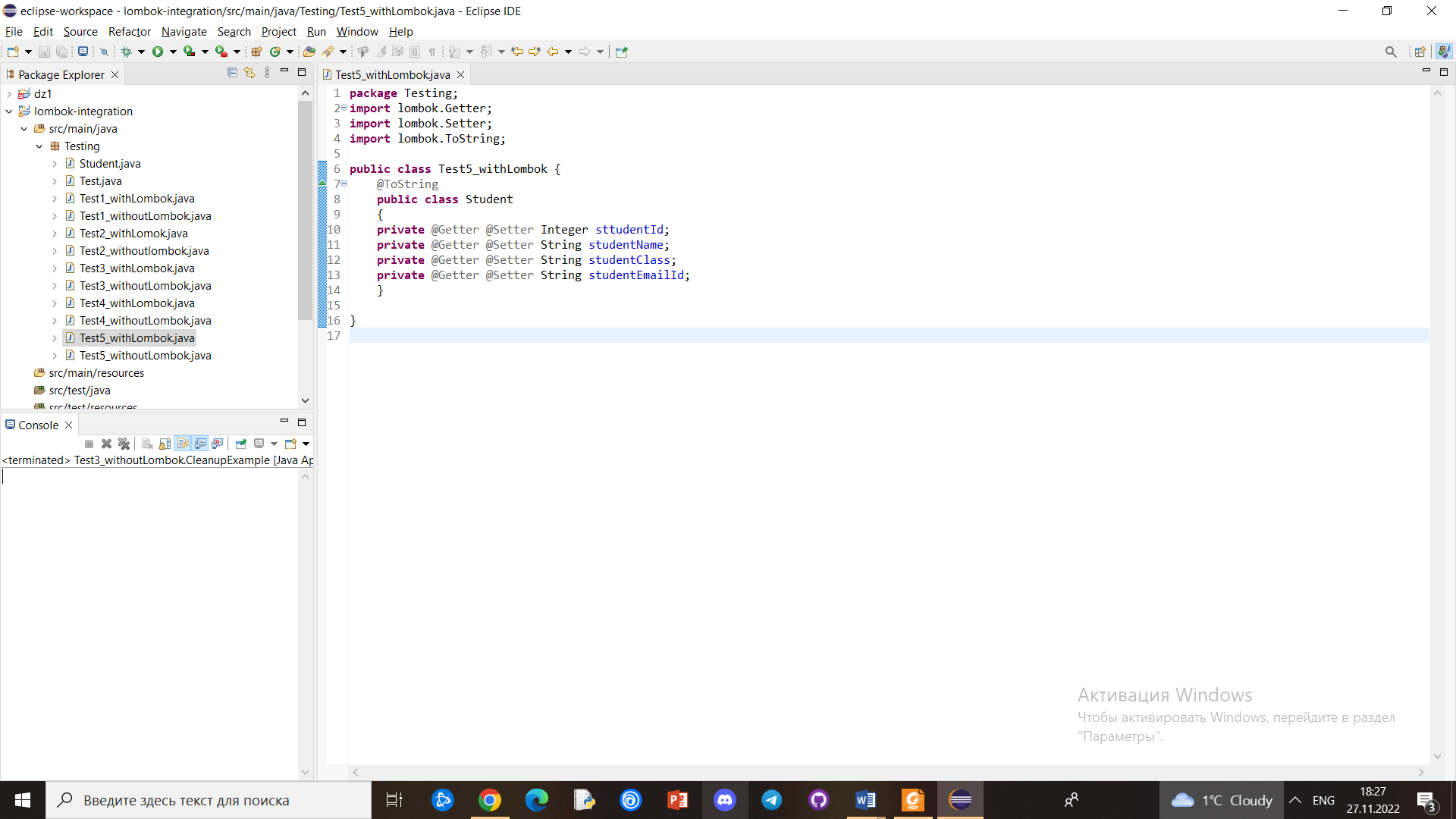
Ви також можете включити вихід виклику методу в toString. Можна включити лише екземпляри (не статичні) методи, які не приймають аргументів. Для цього позначте метод @ToString.Include.

Ви можете змінити ім’я, яке використовується для ідентифікації учасника, за допомогою @ToString.Include(name = "якесь інше ім’я"), а також ви можете змінити порядок друку членів за допомогою @ToString.Include(rank = -1). Члени без рангу вважаються такими, що мають ранг 0, члени вищого рангу друкуються першими, а члени того самого рангу друкуються в тому самому порядку, в якому вони вказані у вихідному файлі.

Розглянемо застосування(Test5\_withoutLombok та Test5\_withLombok): Test5\_withoutLombok:



Test5\_withLombok:



Бачимо дуже сильну економію коду

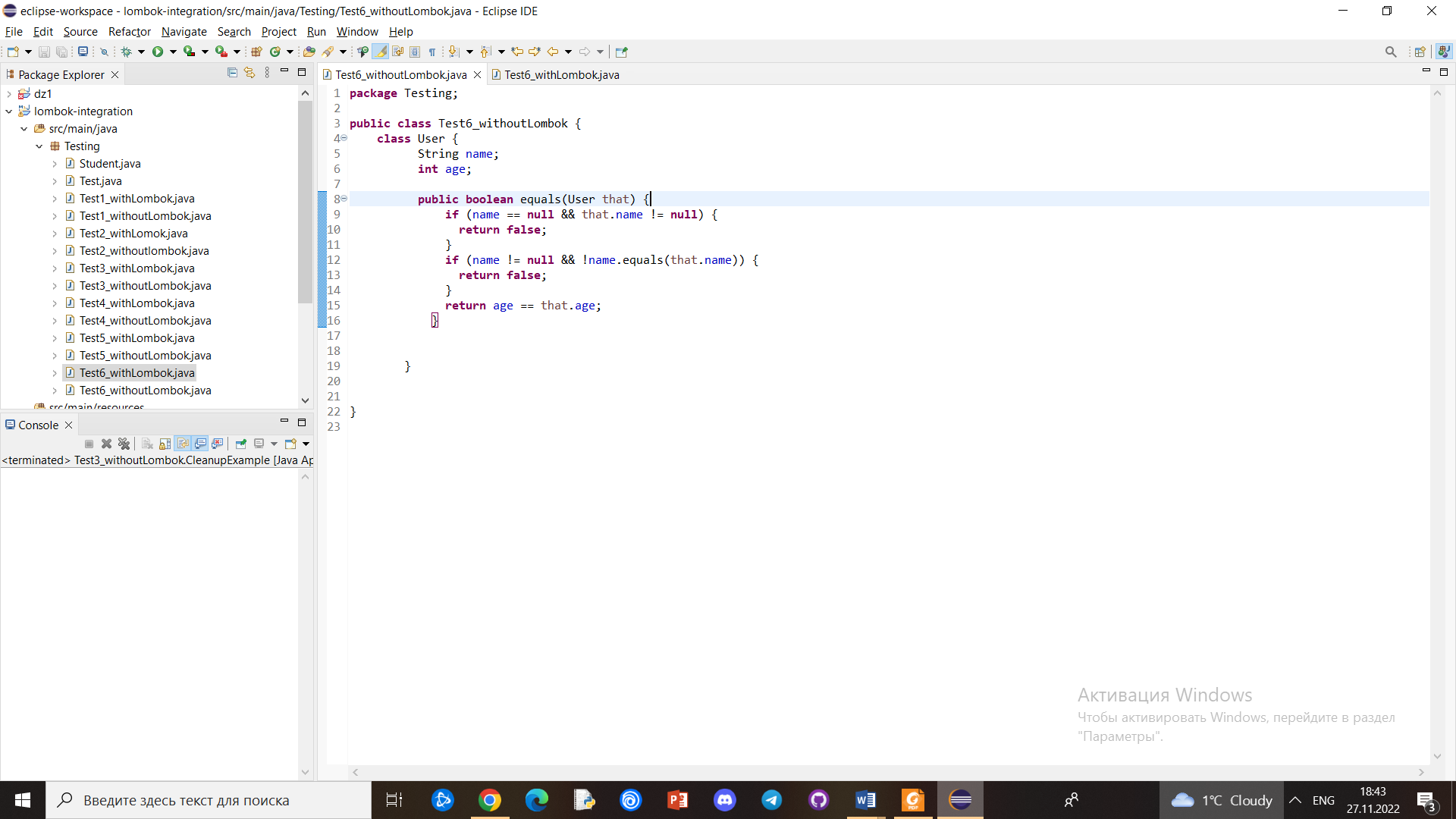
1. @EqualsAndHashCode

Будь-яке визначення класу може бути анотовано @EqualsAndHashCode, щоб дозволити lombok генерувати реалізації методів equals(Object other) і hashCode(). За замовчуванням він використовуватиме всі нестатичні, неперехідні поля, але ви можете змінити, які поля використовувати (і навіть вказати, що потрібно використовувати вихідні дані різних методів), позначивши члени типу @EqualsAndHashCode.Include або @EqualsAndHashCode.Exclude. Крім того, ви можете точно вказати, які поля чи методи ви хочете використовувати, позначивши їх @EqualsAndHashCode.Include і використовуючи @EqualsAndHashCode(onlyExplicitlyIncluded = true).

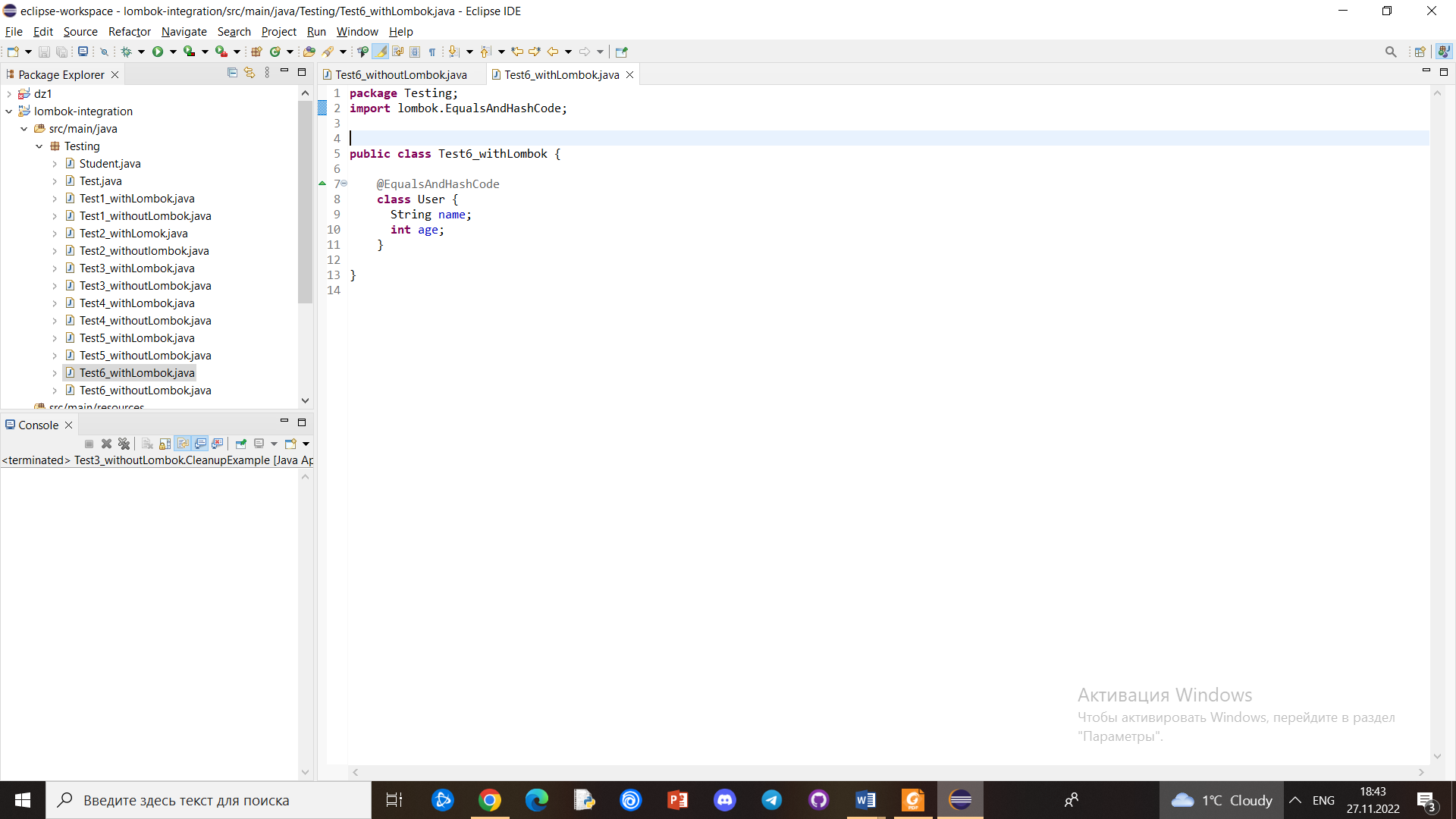
Якщо застосувати @EqualsAndHashCode до класу, який розширює інший, ця функція стає трохи складнішою. Як правило, автоматичне генерування методу equals і hashCode для таких класів є поганою ідеєю, оскільки суперклас також визначає поля, для яких також потрібен код equals/hashCode, але цей код не буде згенеровано. Встановивши для callSuper значення true, ви можете включити методи equals і hashCode вашого суперкласу в згенеровані методи. Для hashCode результат super.hashCode() включається в алгоритм гешування, а forequals, згенерований метод поверне false, якщо суперреалізація вважає, що він не дорівнює переданому об’єкту. Майте на увазі, що не всі рівноправні реалізації справляються з цією ситуацією належним чином. Однак реалізації рівностей, згенерованих ломбоком, справляються з цією ситуацією належним чином, тому ви можете сміливо називати свій суперклас equals, якщо він також має згенерований методом рівності ломбок. Якщо у вас є явний суперклас, ви змушені надати певне значення для callSuper, щоб підтвердити, що ви його врахували; невиконання цього призводить до попередження.

Встановлення для callSuper значення true, коли ви нічого не розширюєте (ви розширюєте java.lang.Object), є помилкою під час компіляції, оскільки це перетворило б згенеровані реалізації equals() і hashCode() на таку саму поведінку, як і просте успадкування цих методи з java.lang.Object: тільки той самий об’єкт буде рівним один одному та матиме однаковий хеш-код. Невстановлення callSuper значення true, коли ви розширюєте інший клас, генерує попередження, тому що, якщо суперклас не має полів (важливих для рівності), lombok не може створити для вас реалізацію, яка враховує поля, оголошені вашими суперкласами. Вам потрібно буде написати власні реалізації або покладатися на засіб зв’язування callSuper. Ви також можете використовувати ключ конфігурації lombok.equalsAndHashCode.callSuper.

Розглянемо застосування(Test6\_withoutLombok та Test6\_withLombok): Test6\_withoutLombok:



Test6\_withLombok:



1. @NoArgsConstructor , @RequiedConstructor, @AllArgsConstructor

Цей набір із 3 анотацій створює конструктор, який прийматиме 1 параметр для певних полів і просто призначає цей параметр полю.

@NoArgsConstructor створить конструктор без параметрів. Якщо це неможливо (через кінцеві поля), замість цього виникне помилка компілятора, якщо не використовується @NoArgsConstructor(force = true), тоді всі кінцеві поля ініціалізуються 0 / false / null. Для полів з обмеженнями, таких як поля @NonNull, перевірка не генерується, тому майте на увазі, що ці обмеження зазвичай не виконуватимуться, доки ці поля не будуть належним чином ініціалізовані пізніше. Деякі конструкції Java, такі як hibernate та інтерфейс постачальника послуг, вимагають конструктора без аргументів. Ця анотація корисна насамперед у поєднанні з @Data або іншим конструктором, що створює анотації.

@RequiredArgsConstructor генерує конструктор з 1 параметром для кожного поля, яке потребує спеціальної обробки. Усі неініціалізовані кінцеві поля отримують параметр, а також будь-які поля, позначені як @NonNull, які не ініціалізовані там, де вони оголошені. Для полів, позначених @NonNull, також генерується явна нульова перевірка. Конструктор створить виняток NullPointerException, якщо будь-який із параметрів, призначених для полів, позначених @NonNull, містить значення null. Порядок параметрів відповідає порядку відображення полів у вашому класі.

@AllArgsConstructor генерує конструктор з 1 параметром для кожного поля у вашому класі. Поля, позначені @NonNull, призводять до нульових перевірок цих параметрів.

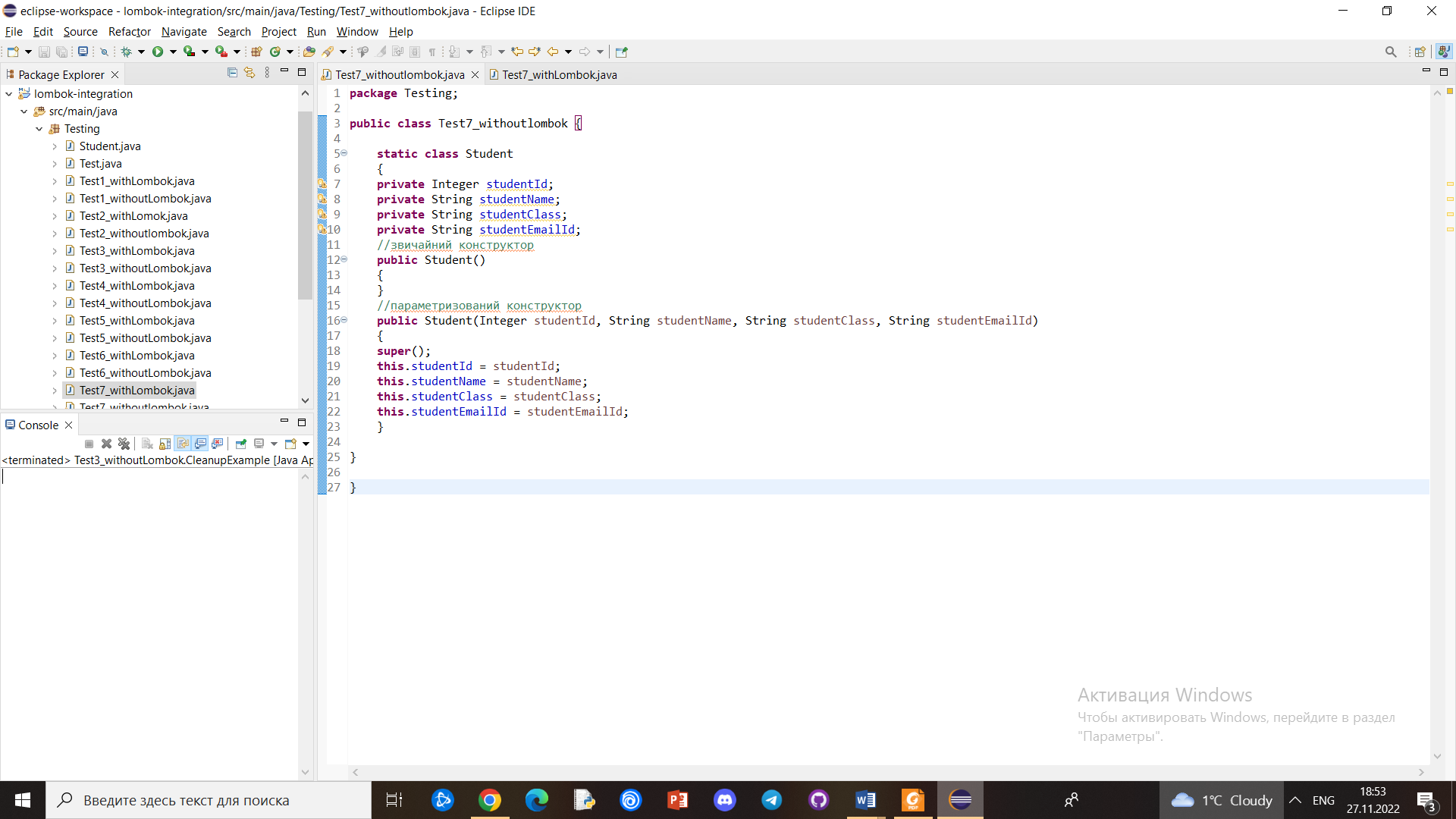
Кожна з цих анотацій допускає альтернативну форму, де згенерований конструктор завжди є приватним, і генерується додатковий статичний фабричний метод, який обертається навколо приватного конструктора. Цей режим активується шляхом надання значення staticName для анотації, наприклад: @RequiredArgsConstructor(staticName="of"). Такий статичний фабричний метод виводить дженерики, на відміну від звичайного конструктора. Це означає, що ваші користувачі API отримують запис MapEntry.of("foo", 5) замість значно довшого нового MapEntry<String, Integer>("foo", 5).

Щоб додати анотації до згенерованого конструктора, ви можете використати onConstructor=@\_\_({@AnnotationsHere}), але будьте обережні; це експериментальна функція. Для отримання додаткової інформації дивіться документацію щодо функції onX.

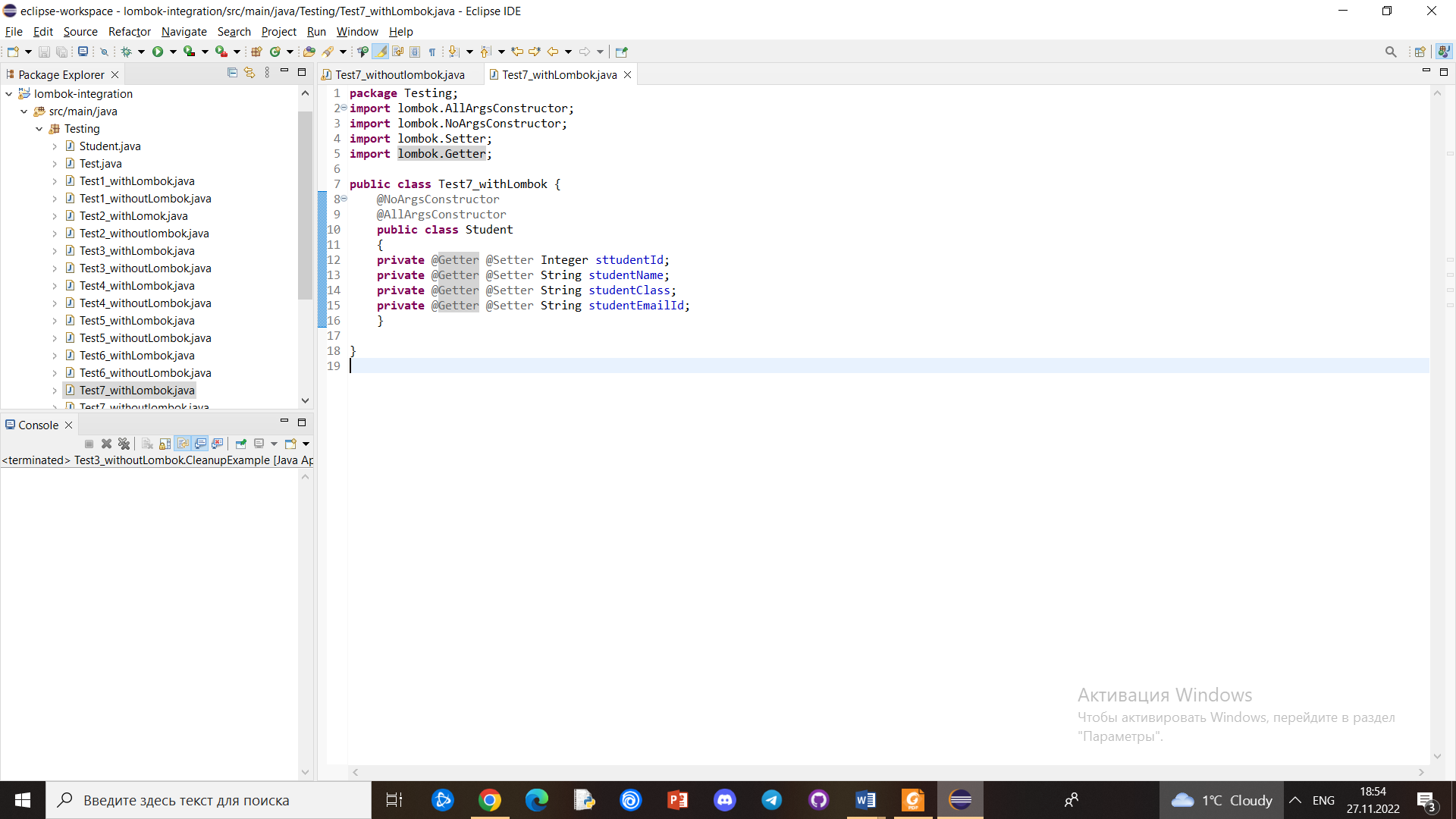
Ці анотації пропускають статичні поля.

На відміну від більшості інших анотацій ломбок, існування явного конструктора не заважає цим анотаціям генерувати свій власний конструктор. Це означає, що ви можете написати свій власний спеціалізований конструктор, а також дозволити lombok генерувати шаблонні конструктори. Якщо виникне конфлікт (один із ваших конструкторів матиме той самий підпис, що й той, який генерує lombok), виникне помилка компілятора.

Розглянемо застосування(Test7\_withoutLombok та Test7\_withLombok): Test7\_withoutLombok:



Test7\_withLombok:



1. @Data

@Data — це зручна анотація швидкого доступу, яка об’єднує функції @ToString, @EqualsAndHashCode, @Getter / @Setter і @RequiredArgsConstructor разом: іншими словами, @Data генерує всі шаблони, які зазвичай асоціюються з простими POJO (прості старі об’єкти Java). ) і компоненти: getters для всіх полів, setters для всіх нефінальних полів і відповідних реалізацій toString, equals і hashCode, які включають поля класу, і конструктор, який ініціалізує всі кінцеві поля, а також усі нефінальні поля без ініціалізатора, позначеного @NonNull, щоб гарантувати, що поле ніколи не буде нульовим.

@Data схоже на наявність неявних анотацій @Getter, @Setter, @ToString, @EqualsAndHashCode та @RequiredArgsConstructor для класу (за винятком того, що конструктор не буде створено, якщо вже існують явно написані конструктори). Однак параметри цих анотацій (наприклад, callSuper, includeFieldNames і exclude) не можна встановити за допомогою @Data. Якщо вам потрібно встановити нестандартні значення для будь-якого з цих параметрів, просто додайте ці анотації явно; @Data досить розумний, щоб відкладати ці анотації.

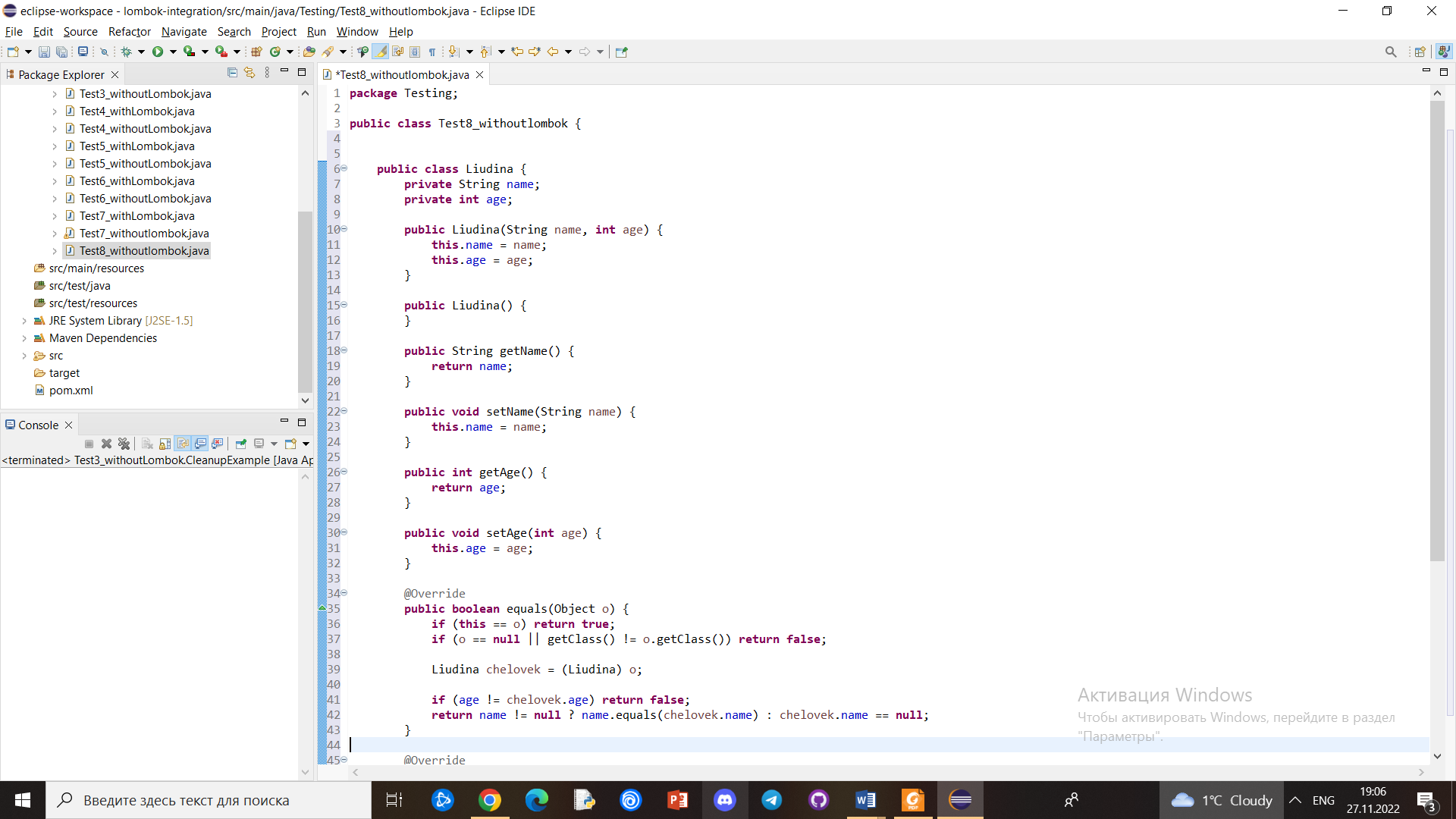
Усі згенеровані геттери та сетери будуть загальнодоступними. Щоб перевизначити рівень доступу, додайте до поля або класу явну анотацію @Setter та/або @Getter. Ви також можете використовувати цю анотацію (об’єднавши її з AccessLevel.NONE), щоб взагалі заборонити генерацію геттера та/або сеттера.

Усі поля, позначені як тимчасові, не будуть розглядатися для hashCode . Усі статичні поля будуть повністю пропущені (не враховуються для жодного зі згенерованих методів, і для них не створюватиметься сетер/гетер).

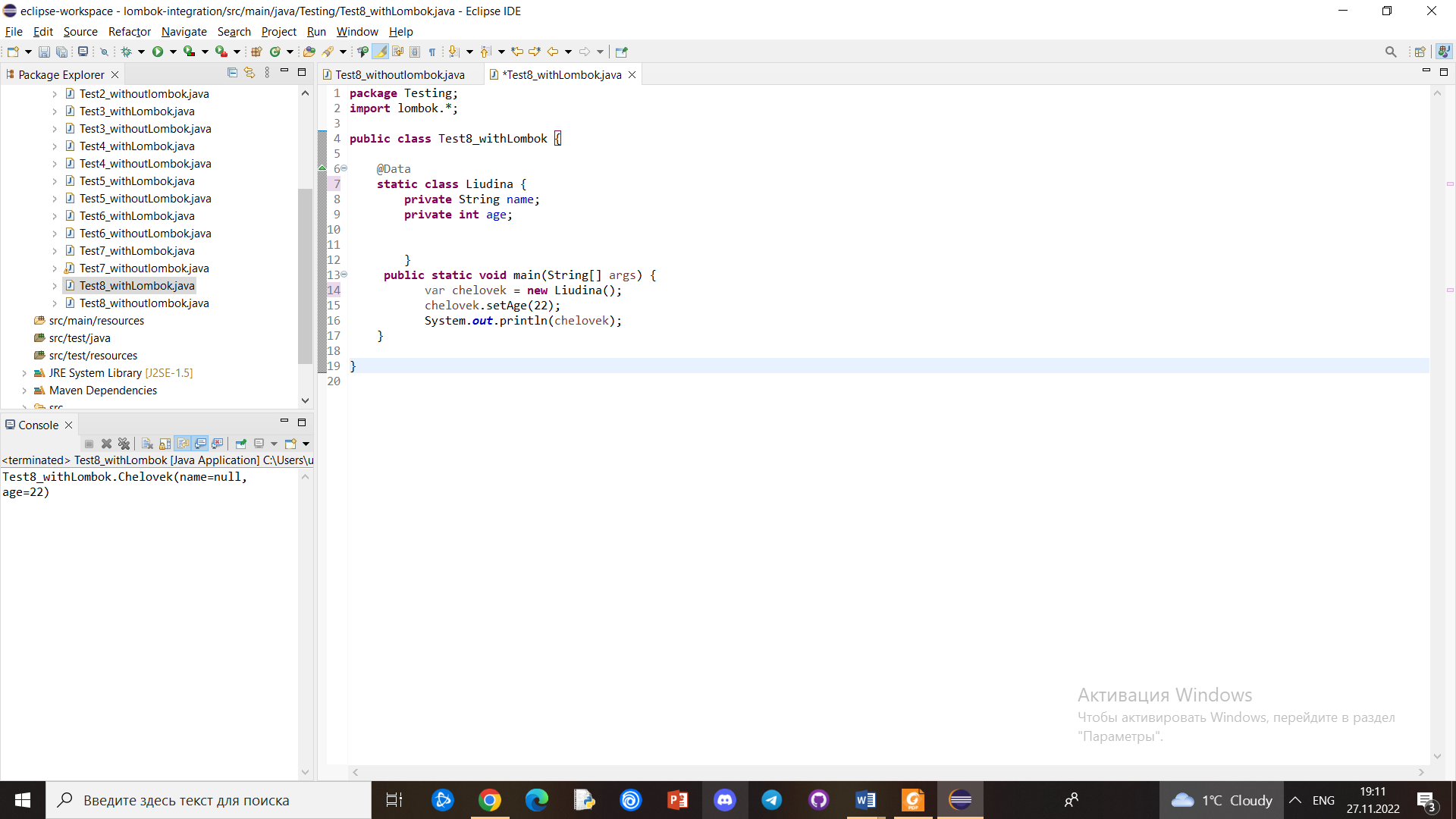
Якщо клас уже містить метод із таким же ім’ям і кількістю параметрів, що й будь-який метод, який зазвичай генерується, цей метод не генерується, і не видається попередження чи помилка. Наприклад, якщо у вас уже є метод із підписом дорівнює (параметр AnyType), метод дорівнює не буде створено, навіть якщо технічно це може бути зовсім інший метод через наявність різних типів параметрів. Те саме правило застосовується до конструктора (будь-який явний конструктор перешкоджатиме його генерації @Data), а також до toString, дорівнює та всіх методів отримання та встановлення. Ви можете позначити будь-який конструктор або метод @lombok.experimental.Tolerate, щоб приховати їх від lombok.

@Data може добре обробляти загальні параметри для полів. Щоб зменшити шаблон під час конструювання об’єктів для класів із узагальненнями, ви можете використовувати параметр staticConstructor для створення приватного конструктора, а також статичний метод, який повертає новий екземпляр. Таким чином, javac визначить назву змінної. Таким чином, оголосивши так: @Data(staticConstructor="of") class Foo<T> { private T x;} ви можете створити нові екземпляри Foo, написавши: Foo.of(5); замість того, щоб писати: new Foo<Integer>(5);.

Розглянемо застосування(Test8\_withoutLombok та Test8\_withLombok): Test8\_withoutLombok:



Test8\_withLombok:



1. @Value

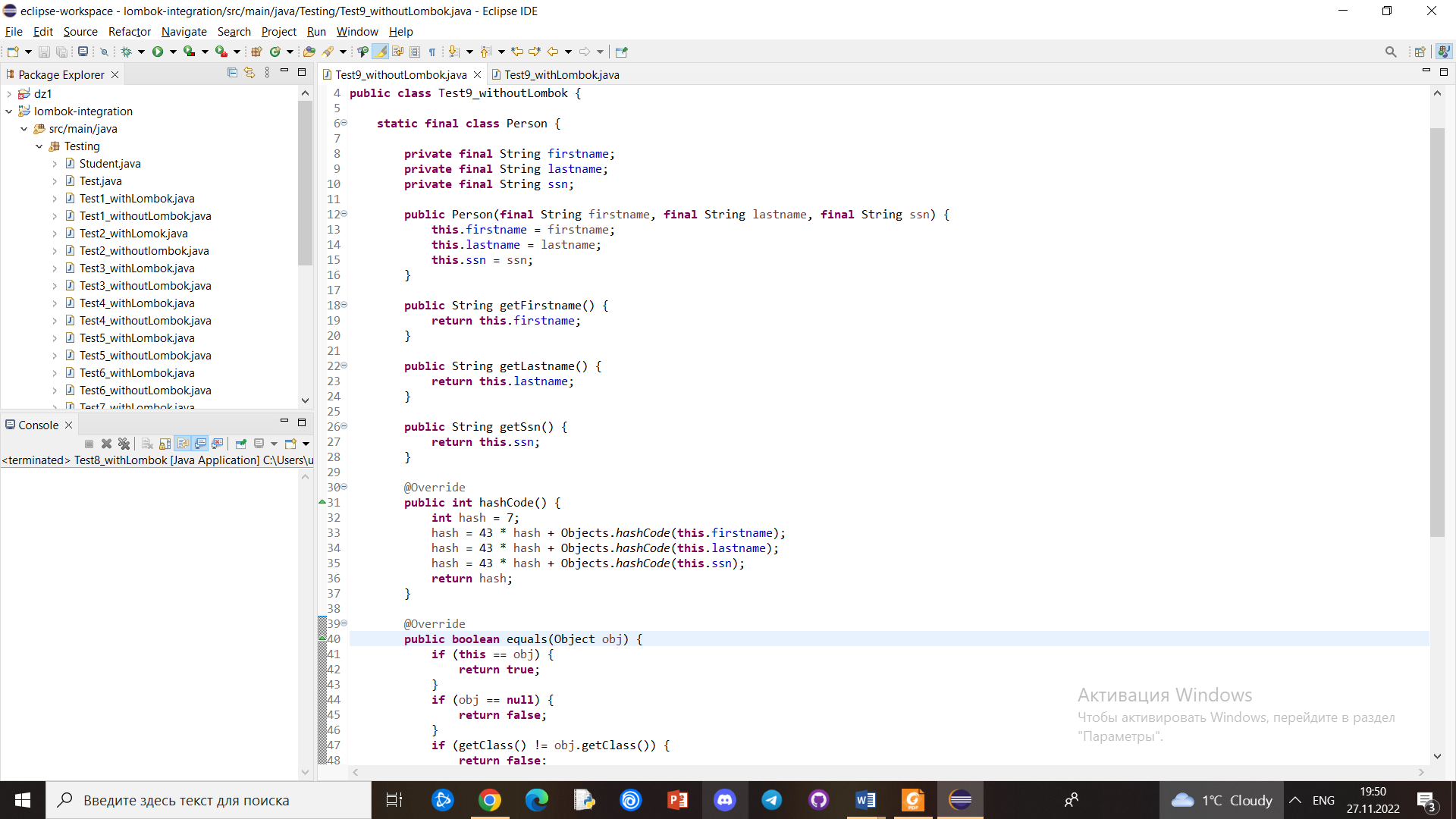
Значення є незмінним варіантом @Data; за замовчуванням усі поля стають приватними та остаточними, а сетери не генеруються. Сам клас також робиться остаточним за замовчуванням, тому що незмінність не є чимось, що можна примусово нав’язати підкласу. Як і @Data, також генеруються корисні методи toString(), equals() і hashCode(), кожне поле отримує метод отримання, а також генерується конструктор, який охоплює кожен аргумент (крім кінцевих полів, які ініціалізуються в декларації поля). .

На практиці @Value є скороченням для: final @ToString @EqualsAndHashCode @AllArgsConstructor @FieldDefaults(makeFinal = true, level = AccessLevel.PRIVATE) @Getter, за винятком того, що явне включення реалізації будь-якого з відповідних методів просто означає, що ця частина буде t буде створено, і попередження не буде видано. Наприклад, якщо ви напишете свій власний toString, помилки не станеться, і lombok не створить toString. Крім того, будь-який явний конструктор, незалежно від списку аргументів, передбачає, що lombok не генеруватиме конструктор. Якщо ви хочете, щоб lombok генерував конструктор усіх аргументів, додайте @AllArgsConstructor до класу. Зауважте, що якщо і `@Builder`, і `@Value` знаходяться в класі, приватний конструктор allargs пакету, який `@Builder` хоче зробити, "перемагає" над загальнодоступним, який `@Value` хоче зробити. Ви можете позначити будь-який конструктор або метод @lombok.experimental.Tolerate, щоб приховати їх від Lombok.

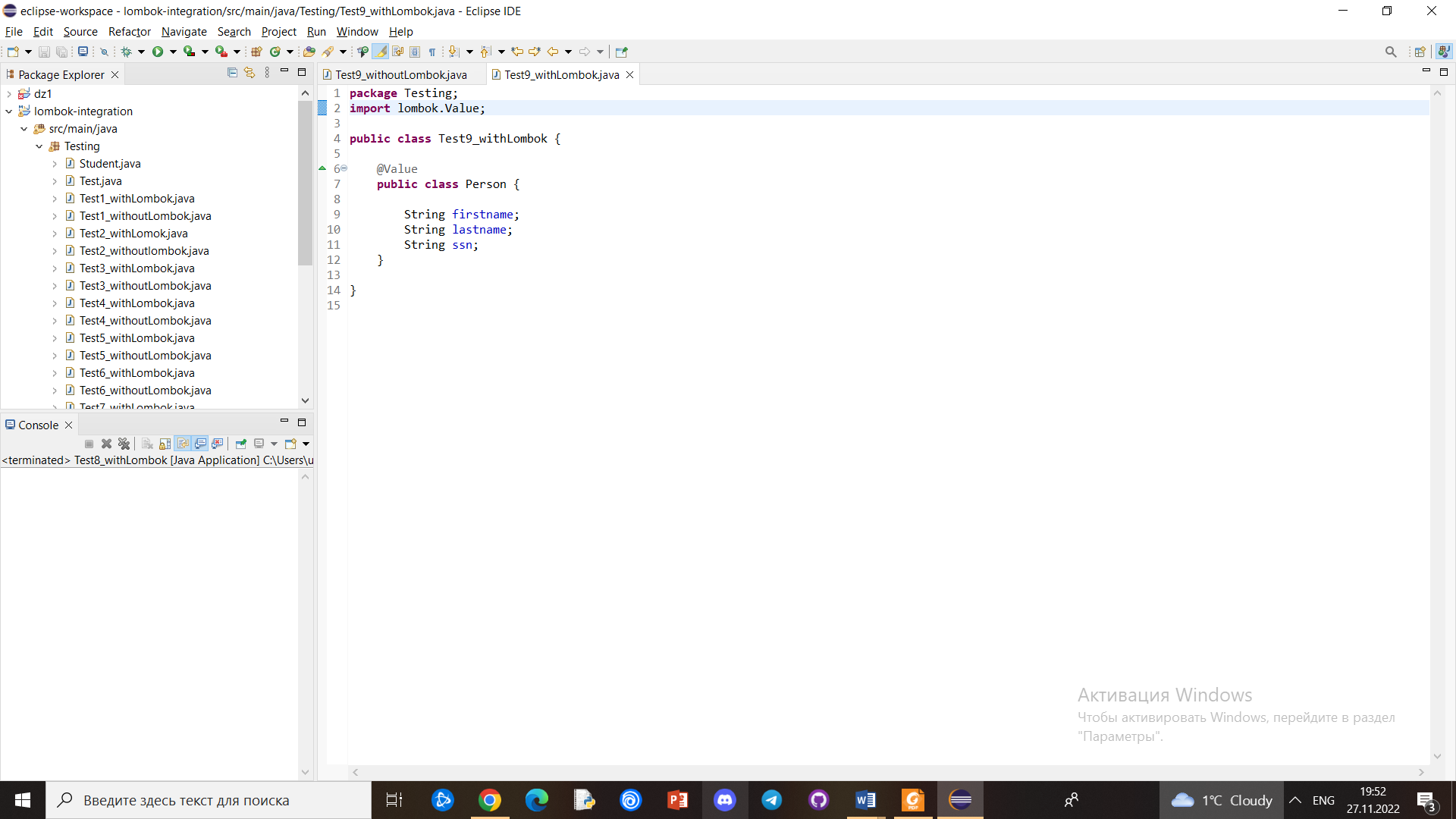
Можна перевизначити поведінку final-by-default і private-by-default, використовуючи або явний рівень доступу до поля, або використовуючи анотації @NonFinal або @PackagePrivate. @NonFinal також можна використовувати в класі, щоб видалити ключове слово final.

Можна перевизначити будь-яку поведінку за замовчуванням для будь-якої з «частин», які складають @Value, використовуючи цю анотацію.

Розглянемо застосування(Test9\_withoutLombok та Test9\_withLombok): Test9\_withoutLombok:



Test9\_withLombok:



1. @Builder

Анотація @Builder створює складні API конструктора для ваших класів.

@Builder дозволяє вам автоматично створювати код, необхідний для створення екземпляра вашого класу за допомогою такого коду, як:

Person.builder()

.name("Адам Севідж")

.city("Сан-Франциско")

.job("Руйнівники міфів")

.job("Звільнена реакція")

.build();

@Builder можна розмістити в класі, або в конструкторі, або в методі. У той час як режими «на класі» та «на конструкторі» є найпоширенішими варіантами використання, @Builder найпростіше пояснити за допомогою варіанта використання «методу».

Метод, анотований @Builder (відтепер називається цільовим), призводить до генерації наступних 7 речей:

Внутрішній статичний клас під назвою FooBuilder з аргументами того самого типу, що й статичний метод (званий конструктором).

У конструкторі: одне приватне нестатичне нефінальне поле для кожного параметра цілі.

У конструкторі: приватний порожній конструктор без аргументів.

У конструкторі: метод, схожий на «setter», для кожного параметра цілі: він має той самий тип, що й цей параметр, і те саме ім’я. Він повертає сам конструктор, так що виклики налаштування можуть бути з’єднані, як у наведеному вище прикладі.

У конструкторі: метод build(), який викликає метод, передаючи кожне поле. Він повертає той самий тип, що й мета.

У конструкторі: розумна реалізація toString().

У класі, що містить ціль: метод builder(), який створює новий екземпляр конструктора.

Кожен згенерований елемент у списку буде мовчки пропущено, якщо цей елемент уже існує (без урахування кількості параметрів і перегляду лише назв). Це включає сам конструктор: якщо цей клас уже існує, lombok просто почне впроваджувати поля та методи всередині цього вже існуючого класу, якщо, звичайно, поля / методи, які потрібно вставити, уже існують. Однак ви не можете розміщувати будь-який інший метод (або конструктор), що генерує анотацію lombok, до класу конструктора; наприклад, ви не можете розмістити @EqualsAndHashCode у класі конструктора.

@Builder може генерувати так звані «сингулярні» методи для параметрів/полів колекції. Вони беруть 1 елемент замість цілого списку та додають елемент до списку. Наприклад:

Person.builder()

.job("Руйнівники міфів")

.job("Звільнена реакція")

.build();

призведе до того, що поле List<String> jobs матиме 2 рядки. Щоб отримати таку поведінку, поле/параметр потрібно анотувати @Singular. Функція має власну документацію.

Якщо у вас є явний конструктор, додайте анотацію @Builder до конструктора, а не до класу. Зауважте, що якщо ви розміщуєте як `@Value`, так і `@Builder` до класу, приватний конструктор пакета, який `@Builder` хоче згенерувати, "перемагає" та пригнічує конструктор, який `@Value` хоче створити.

Якщо ви використовуєте @Builder для створення конструкторів для створення екземплярів вашого власного класу (це завжди так, якщо не додати @Builder до методу, який не повертає ваш власний тип), ви можете використовувати @Builder(toBuilder = true) для також генерації метод екземпляра у вашому класі під назвою toBuilder(); він створює новий конструктор, який починає з усіма значеннями цього екземпляра. Ви можете додати анотацію @Builder.ObtainVia до параметрів (у випадку конструктора чи методу) або полів (у випадку @Builder до типу), щоб вказати альтернативні засоби, за допомогою яких значення цього поля/параметра отримано з цього екземпляр. Наприклад, ви можете вказати метод, який потрібно викликати: @Builder.ObtainVia(method = "calculateFoo").

Ім’я класу конструктора – FoobarBuilder, де Foobar – це спрощена форма типу повернення цілі з регістром заголовків, тобто ім’я вашого типу для @Builder у конструкторах і типах, а також ім’я типу повернення для @Builder щодо методів. Наприклад, якщо @Builder застосовано до класу з назвою com.yoyodyne.FancyList<T>, тоді ім’я конструктора буде FancyListBuilder<T>. Якщо @Builder застосовано до методу, який повертає void, конструктор матиме назву VoidBuilder.

Конфігуровані аспекти конструктора:

Назва класу конструктора (за замовчуванням: тип повернення + «Конструктор»)

Назва методу build() (за замовчуванням: "build")

Назва методу builder() (за замовчуванням: "builder")

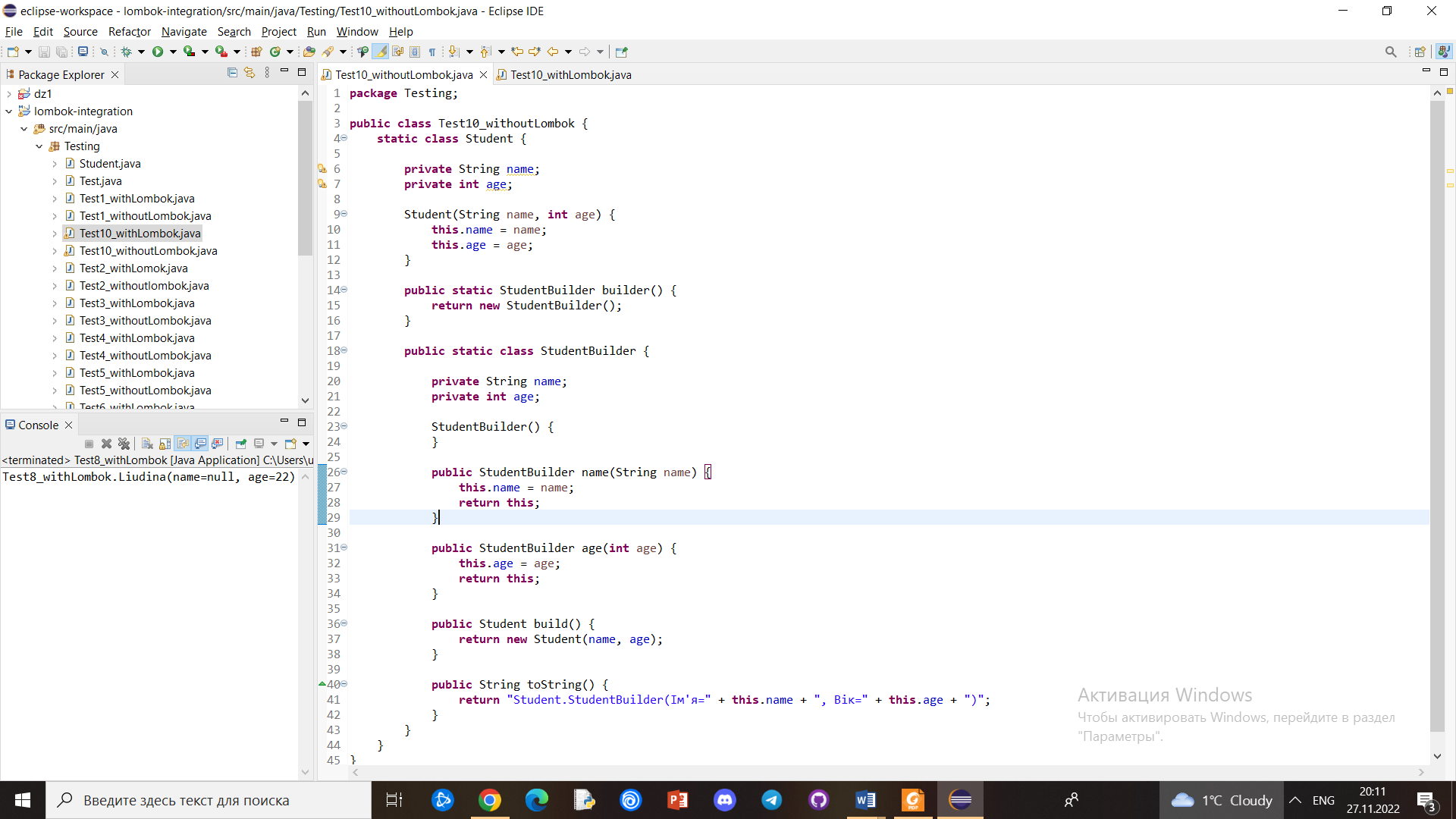
Якщо ви хочете Builder() (за замовчуванням: ні)

Рівень доступу до всіх згенерованих елементів (за замовчуванням: публічний).

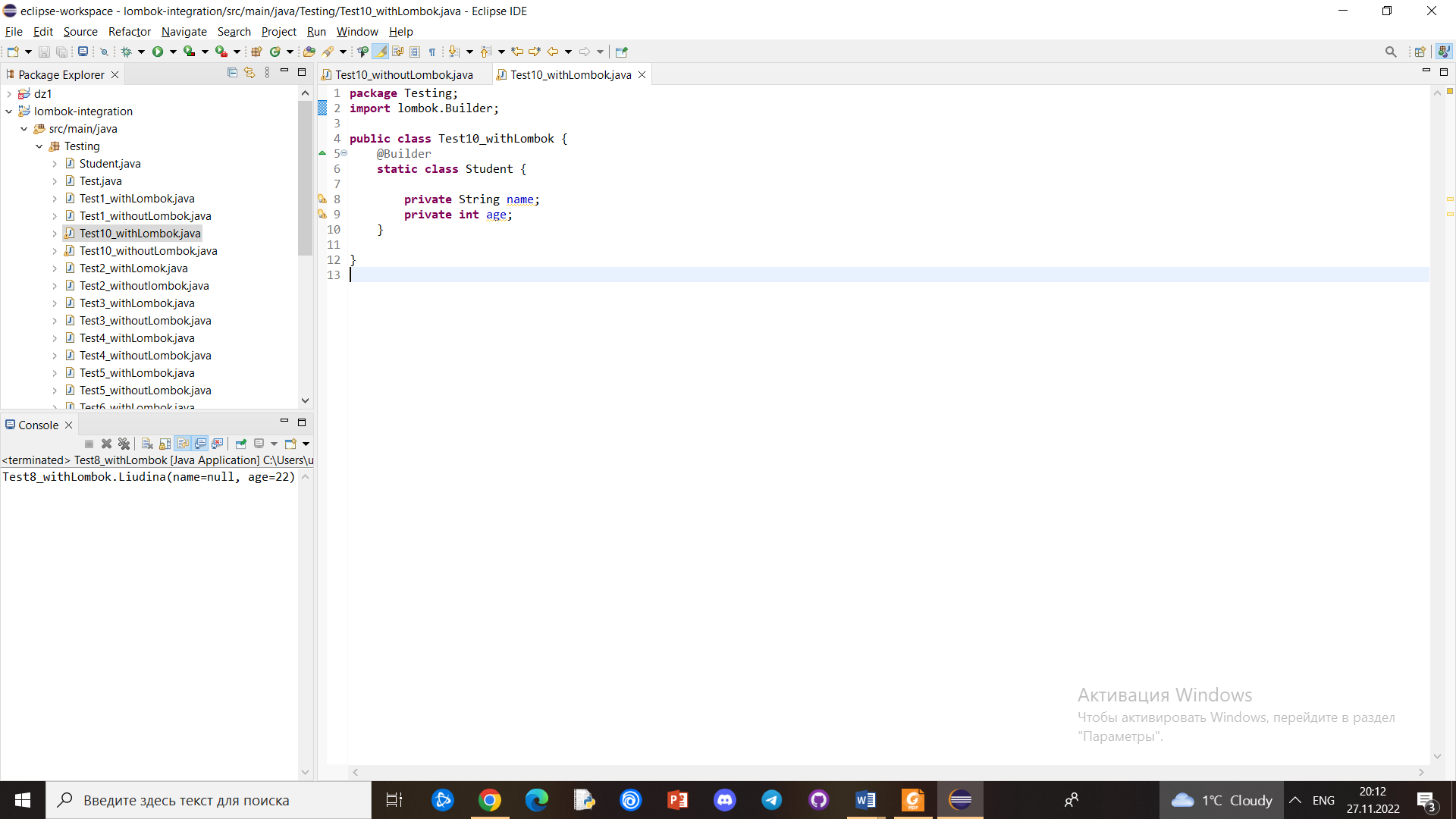
Приклад використання де всі параметри змінюються від стандартних:

@Builder(builderClassName = "HelloWorldBuilder", buildMethodName = "виконати", builderMethodName = "helloWorld", toBuilder = true, access = AccessLevel.PRIVATE, setterPrefix = "set")

Розглянемо застосування(Test10\_withoutLombok та Test10\_withLombok): Test10\_withoutLombok:



Test10\_withLombok:



1. @SneakyThrows

@SneakyThrows можна використовувати для прихованого викидання перевірених винятків, фактично не оголошуючи це в пункті throws вашого методу. Цю дещо суперечливу здатність, звичайно, слід використовувати обережно. Код, згенерований lombok, не ігноруватиме, не загортатиме, замінюватиме чи іншим чином змінюватиме викинуте перевірене виключення; він просто підробляє компілятор. На рівні JVM (файл класу) усі винятки, перевірені чи ні, можуть бути викинуті незалежно від пункту throws ваших методів, тому це працює.

Типові випадки використання, коли ви хочете відмовитися від перевіреного механізму винятків, зосереджені навколо 2 ситуацій:

Невиправдано суворий інтерфейс, такий як Runnable - незалежно від того, який виняток поширюється з вашого методу run(), перевірений чи ні, він буде переданий обробнику необроблених винятків Thread. Перехоплення перевіреного винятку та загортання його в якийсь RuntimeException лише приховує справжню причину проблеми.

«Неможливий» виняток. Наприклад, new String(someByteArray, "UTF-8"); заявляє, що він може викидати UnsupportedEncodingException, але відповідно до специфікації JVM, UTF-8 завжди має бути доступним. UnsupportedEncodingException тут приблизно така ж імовірна, як ClassNotFoundError, коли ви використовуєте об’єкт String, і ви також не вловлюєте їх.

Обмеження непотрібно суворими інтерфейсами особливо поширене при використанні лямбда-синтаксису (arg -> action); однак лямбда-вирази не можна анотувати, що означає, що не так просто використовувати @SneakyThrows у поєднанні з лямбда-виразами. Неможливо безпосередньо перехопити потайки викинуті перевірені типи, оскільки javac не дозволить вам написати блок catch для типу винятку, який жоден виклик методу в тілі try не оголошує як викинутий.

1. @Synchronized

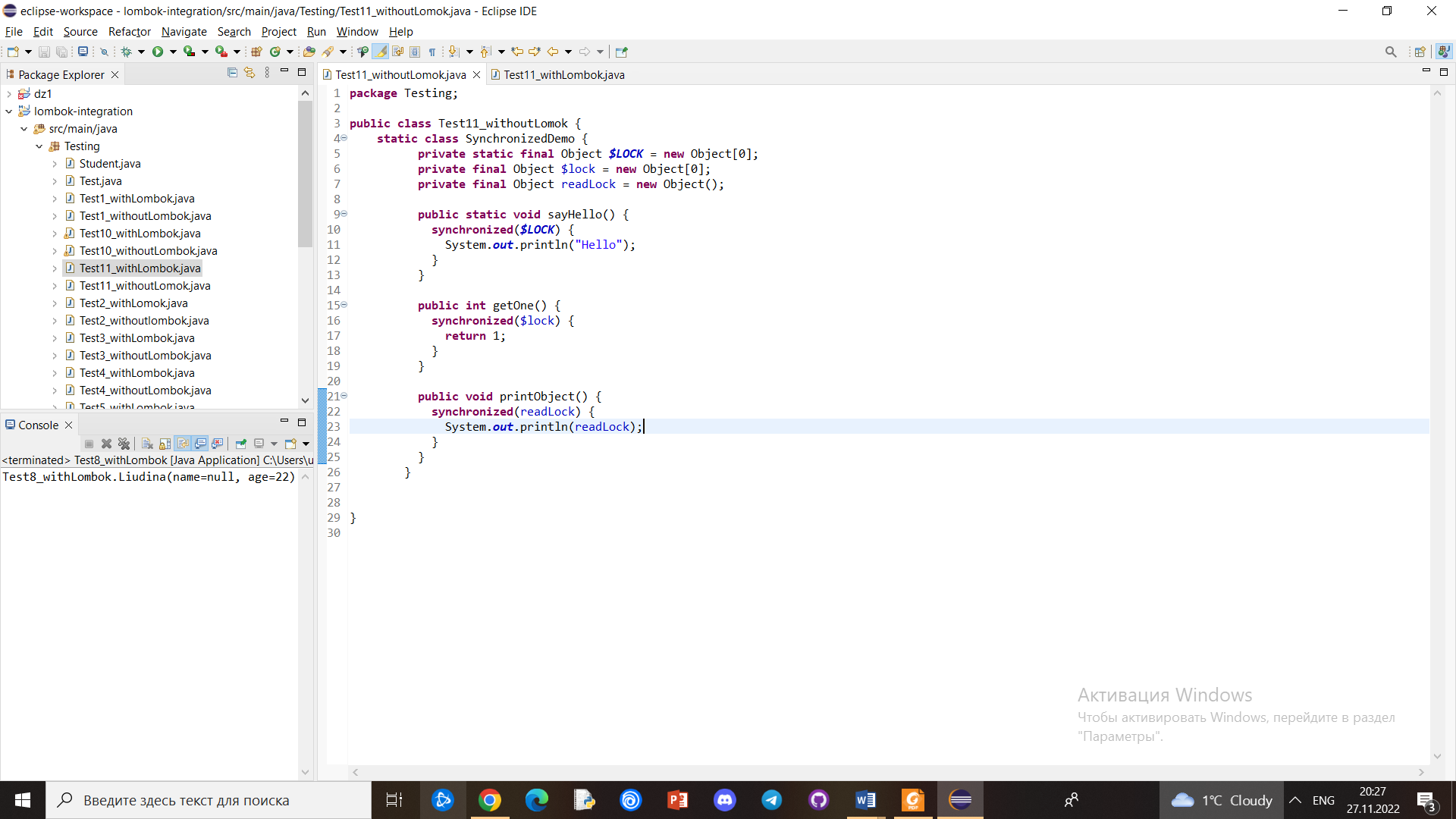
@Synchronized — безпечніший варіант модифікатора методу synchronized. Як і synchronized, анотацію можна використовувати лише для статичних методів і методів екземплярів. Він працює подібно до ключового слова synchronized, але блокує інші об’єкти. Ключове слово блокується на цьому, але анотація блокується на полі з назвою $lock, яке є приватним.

Якщо поле не існує, воно створюється для вас. Якщо ви анотуєте статичний метод, анотація блокується на статичному полі з назвою $LOCK.

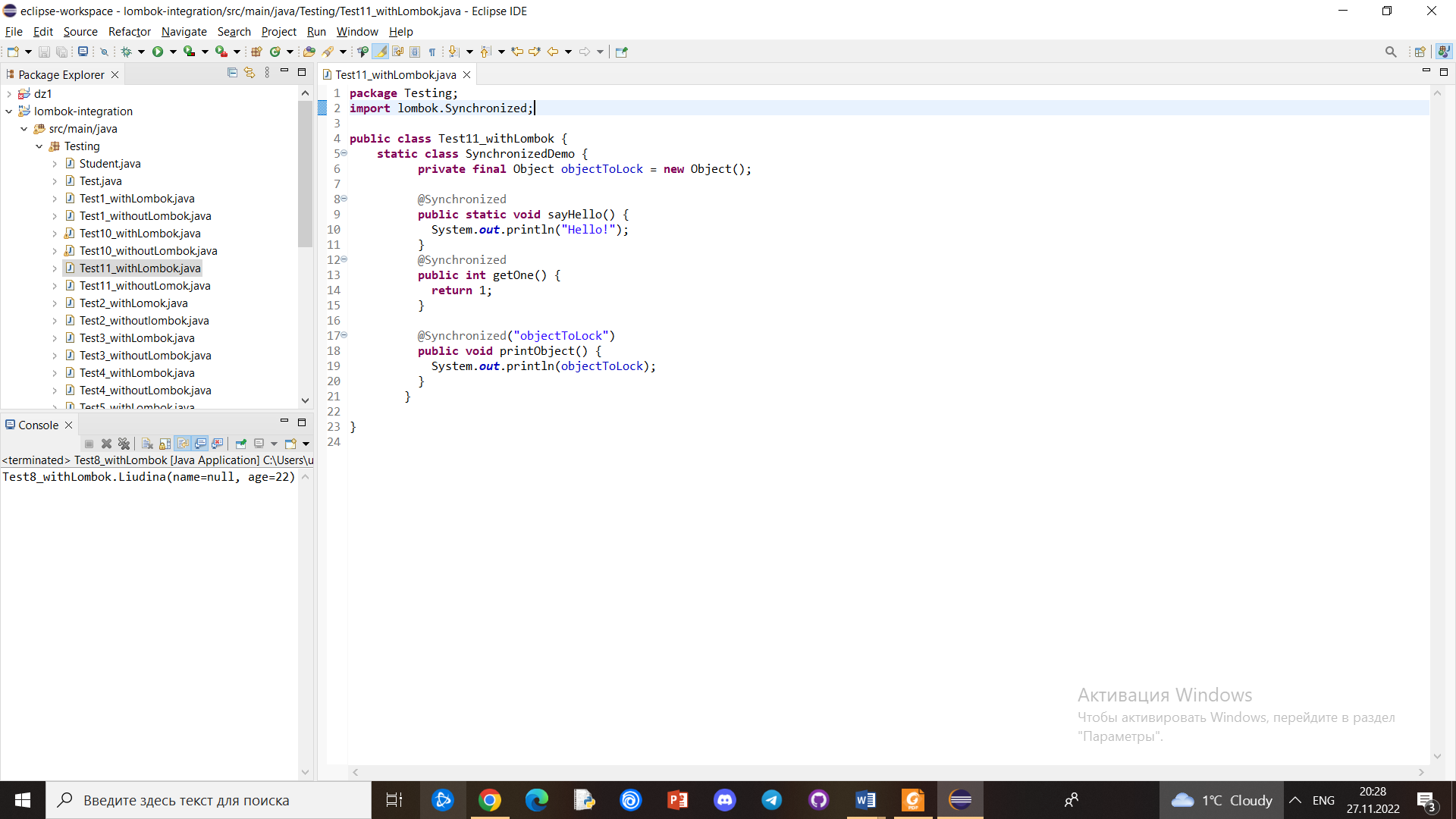
При бажанні ви можете створити ці замки самостійно. Звичайно, поля $lock і $LOCK не будуть згенеровані, якщо ви вже створили їх самостійно. Ви також можете заблокувати інше поле, вказавши його як параметр для анотації @Synchronized. У цьому варіанті використання поля не створюватимуться автоматично, і ви повинні явно створити їх самостійно, інакше буде видано помилку.

Блокування цього або вашого власного об’єкта класу може мати неприємні побічні ефекти, оскільки інший код, який не знаходиться під вашим контролем, також може блокувати ці об’єкти, що може спричинити умови змагання та інші неприємні помилки, пов’язані з потоками.

Розглянемо застосування(Test11\_withoutLombok та Test11\_withLombok): Test11\_withoutLombok:



Test11\_withLombok:



1. @With

Наступною найкращою альтернативою сеттеру для незмінної властивості є створення клону об’єкта, але з новим значенням для цього поля. Метод генерації цього клону – це саме те, що генерує @With: метод withFieldName(newValue), який створює клон, за винятком нового значення для пов’язаного поля.

Наприклад, якщо ви створюєте публічний клас Point { private final int x, y; }, сеттери не мають сенсу, оскільки поля остаточні. @With може створити для вас метод withX(int newXValue), який поверне нову точку з наданим значенням для x і тим самим значенням для y.

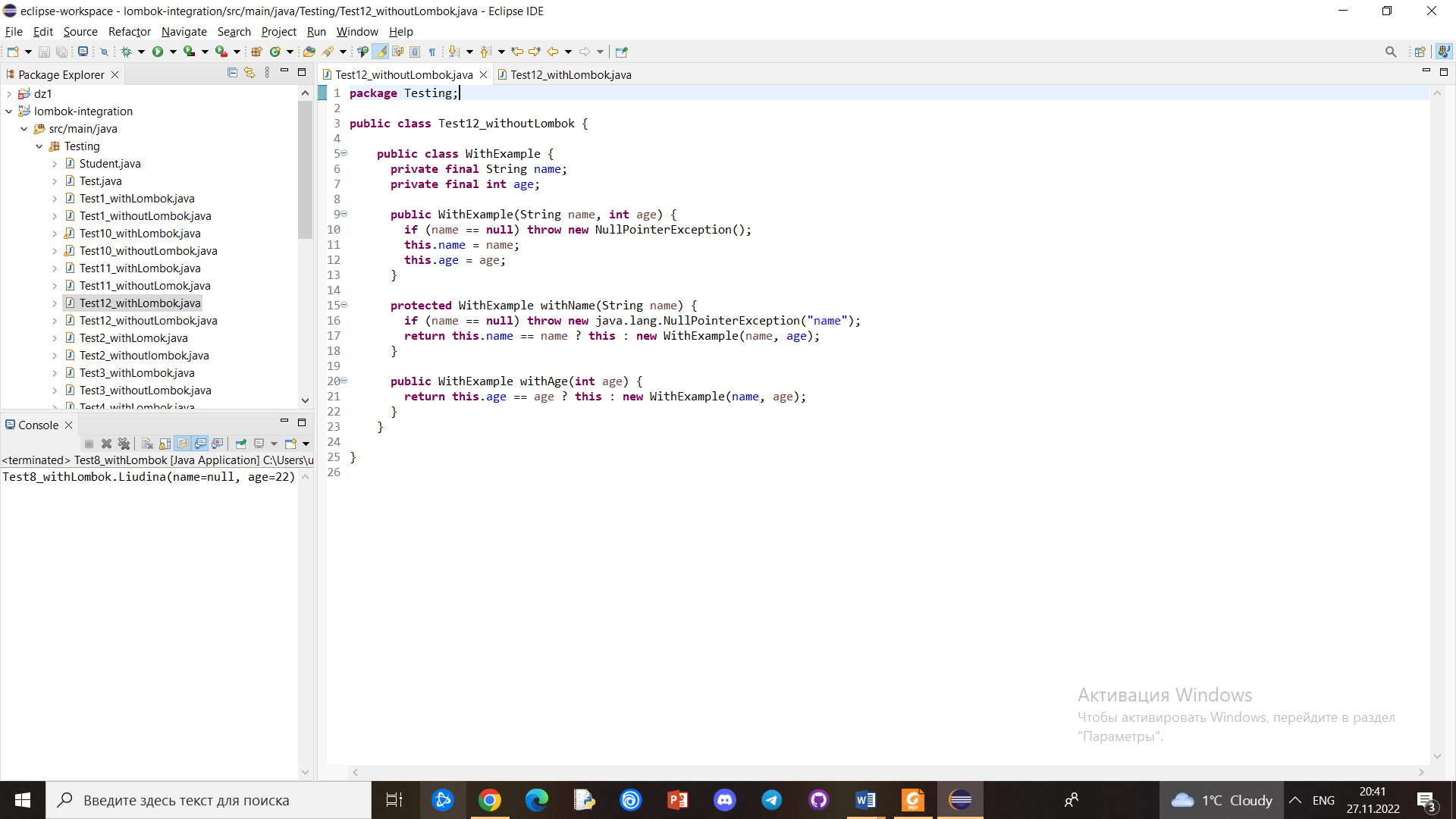
Для виконання своєї роботи @With покладається на конструктор для всіх полів. Якщо цей конструктор не існує, ваша анотація @With призведе до повідомлення про помилку під час компіляції. Ви можете використовувати власний @AllArgsConstructor Ломбока, або, оскільки Value автоматично створюватиме конструктор усіх аргументів, ви також можете використовувати його. Звичайно, також прийнятно, якщо ви вручну напишете цей конструктор. Він повинен містити всі нестатичні поля в тому самому лексичному порядку.

Подібно до @Setter, ви можете вказати рівень доступу на випадок, якщо ви хочете, щоб згенерований метод був чимось іншим, ніж публічний:

@With(level = AccessLevel.PROTECTED). Так само, як @Setter, ви також можете додати анотацію @With до типу, що означає, що метод with генерується для кожного поля (навіть для нефінальних полів)

Щоб додати анотації до згенерованого методу, ви можете використати onMethod=@\_\_({@AnnotationsHere}).

Розглянемо застосування(Test12\_withoutLombok та Test12\_withLombok): Test12\_withoutLombok:



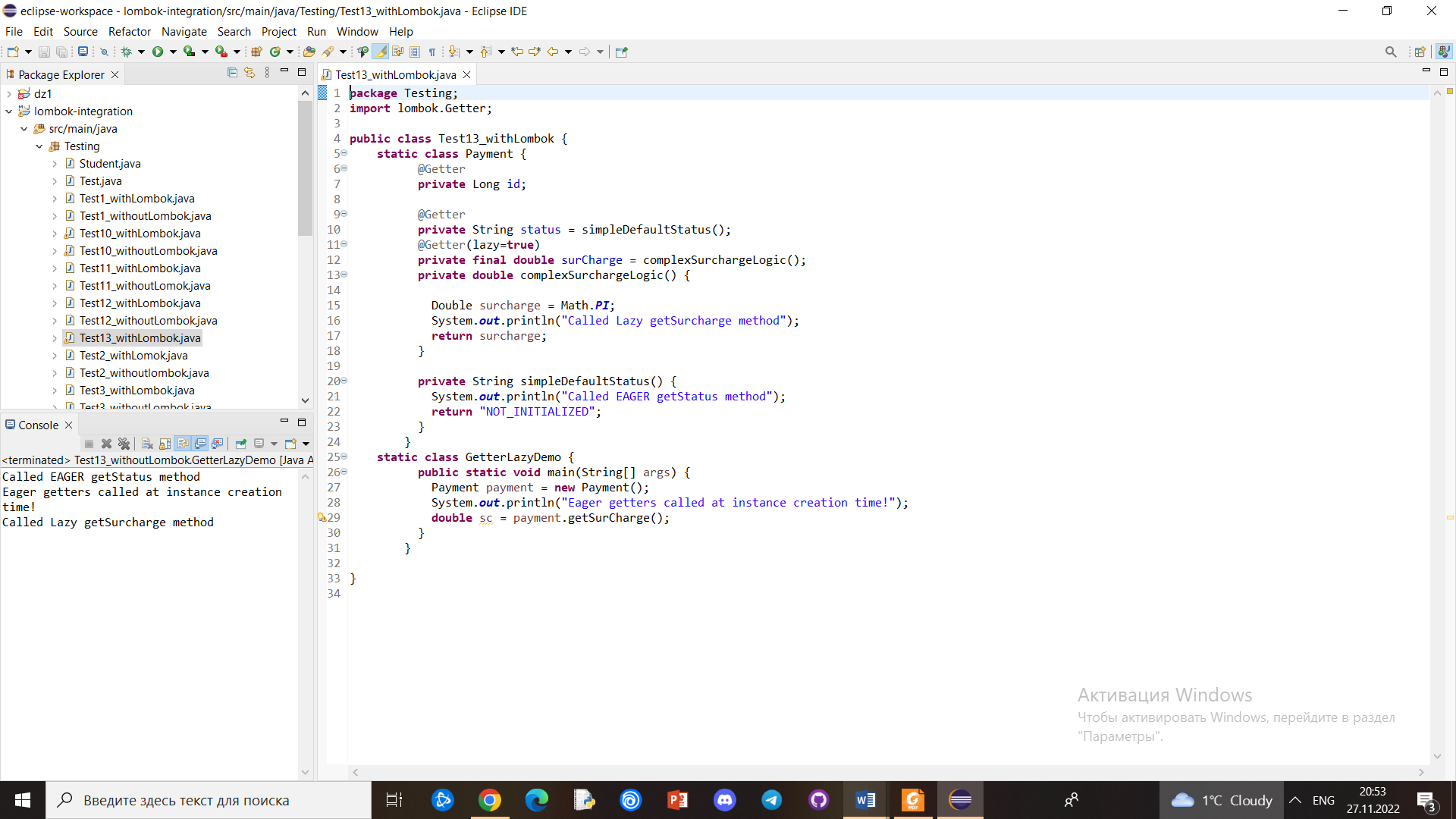
Test12\_withLombok:



1. @Getter(lazy=true)

Ви можете дозволити lombok генерувати геттер, який обчислить значення один раз, під час першого виклику цього геттера, і з того часу кешуватиме його. Це може бути корисним, якщо обчислення значення займає багато ЦП або значення потребує багато пам’яті. Щоб скористатися цією функцією, створіть приватну кінцеву змінну, ініціалізуйте її виразом, виконання якого є дорогим, і додайте анотацію до свого поля за допомогою @Getter(lazy=true). Поле буде приховано від решти вашого коду, а вираз буде обчислено не більше одного разу під час першого виклику геттера. Немає магічних значень маркерів (тобто, навіть якщо результат вашого складного обчислення дорівнює нулю, результат кешується), і ваші складні обчислення не повинні бути потокобезпечними, оскільки lombok піклується про блокування.

Розглянемо застосування тільки на прикладі з використанням гетеру(оск це част вип того що було раніше) (Test13\_withLombok):



1. @Log

Ви розміщуєте варіант @Log у своєму класі (незалежно від того, який з них підходить для системи реєстрації, яку ви використовуєте); тоді у вас є статичне остаточне поле журналу, ініціалізоване, як це зазвичай призначається для фреймворку журналювання, який ви використовуєте, яке потім можете використовувати для запису операторів журналу.

Є кілька доступних варіантів:

@CommonsLog

Створює приватний статичний фінал org.apache.commons.logging.Log log = org.apache.commons.logging.LogFactory.getLog(LogExample.class);

@Flogger

Створює приватний статичний остаточний журнал com.google.common.flogger.FluentLogger = com.google.common.flogger.FluentLogger.forEnclosingClass();

@JBossLog

Створює приватний статичний фінал org.jboss.logging.Logger log = org.jboss.logging.Logger.getLogger(LogExample.class);

@Log

Створює приватний статичний фінал java.util.logging.Logger log = java.util.logging.Logger.getLogger(LogExample.class.getName());

@Log4j

Створює приватний статичний фінал org.apache.log4j.Logger log = org.apache.log4j.Logger.getLogger(LogExample.class);

@Log4j2

Створює приватний статичний фінал org.apache.logging.log4j.Logger log = org.apache.logging.log4j.LogManager.getLogger(LogExample.class);

@Slf4j

Створює приватний статичний остаточний журнал org.slf4j.Logger = org.slf4j.LoggerFactory.getLogger(LogExample.class);

@XSlf4j

Створює приватний статичний остаточний журнал org.slf4j.ext.XLogger = org.slf4j.ext.XLoggerFactory.getXLogger(LogExample.class);

@CustomLog

Створює приватний статичний остаточний журнал com.foo.your.Logger = com.foo.your.LoggerFactory.createYourLogger(LogExample.class);

Цей параметр вимагає, щоб ви додали конфігурацію до свого файлу lombok.config, щоб указати, що має робити @CustomLog.

Наприклад:lombok.log.custom.declaration = com.foo.your.Logger com.foo.your.LoggerFactory.createYourLog(TYPE)(TOPIC), який створить наведений вище оператор. Спочатку йде тип, який є типом вашого реєстратора, потім пробіл, потім тип вашої фабрики реєстратора, потім крапка, потім назва методу фабрики реєстратора, а потім 1 або 2 визначення параметрів; щонайбільше одне визначення з ТЕМОЮ та щонайбільше одне без ТЕМИ. Кожне визначення параметра вказується як список типів параметрів у круглих дужках, розділених комами. Доступні такі параметри: TYPE (передає цей декорований тип @Log як клас), NAME (передає повне ім’я цього декорованого типу @Log), TOPIC (передає явно вибраний рядок теми, встановлений в анотації @CustomLog), і NULL ( передає нуль).

Тип реєстратора необов’язковий; якщо він опущений, використовується заводський тип реєстратора. (Отже, якщо ваш клас реєстратора має статичний метод, який створює реєстратори, ви можете скоротити визначення свого реєстратора).

Висновок

Дана бібліотека ще має функції, але вони єкспериментальні та знаходяться в розробці.

Отже, бібліотека Lombok дуже полегшує і робить чистішим код на будь-яку тему і зберігає час програміста.