Zaawansowane Programowanie Obiektowe i Funkcyjne
Wyrażenia lambda
Zadanie oceniane nr 2a
14-11-2022

Dzisiaj będzie bez kodu wstępnego. Źródła powinny się znaleźć w Państwa katalogu roboczym (git) w podkatalogu zpoif_zadanie2a. Po zakończeniu pracy konieczne jest wgranie zmian.

"Amunicja"

W pewnym mieście znaleziono skrzynię z amunicją. Należy stworzyć model zawartości wraz z towarzyszącą logiką bazującą na temacie wiodącym dzisiejszego zadania – czyli **wyrażeniach lambda**.

Miejmy gdzieś z tyłu głowy fakt, iż w porównaniu z klasą anonimową obiekt zdefiniowany za pomocą wyrażenia Lambda, można traktować jako metodę anonimową, czyli byt pozornie jeszcze bardziej "okrojony" z niepotrzebnego ciała niż klasa "bez nazwy" (w kórym to ciele zawsze można wstawić coś więcej niż tylko metodę). Używamy minimum tego co jest potrzebne (jeśli coś da się zrobić "lambdami", **nie deklarujmy** anonimu lub czegoś większego). Nie wymagam użycia wzorca Wizytor – dzisiaj można iść na skróty i do woli używać **instanceof**a. Nie używamy też strumieni – o nich będzie za tydzień. Tradycyjnie trzymamy się zasady: minimum zasięgu i minimum tego co potrzebujemy (chyba że w zadaniu wskazane będzie inaczej). Zamiast publicznych pól używamy setterów i getterów. Za ładną strukturę pakietów i formatowanie kodu – tradycyjnie drobny extra bonus.

Prace do wykonania:

1. Stworzyć hierarchię klas reprezentującą następujące rodzaje amunicji:
   - Granat obronny (id*, zabezpieczony*** – tak/nie)
   - Granat przeciwpancerny (id*, zabezpieczony*** – tak/nie, emisjaCO2 – 220-250)
   - Granat zaczepny (id*, zabezpieczony*** – tak/nie)
   - Nabój (id**, kaliber*** – 4/5,56/7,62/9/12,7)

   * unikalny identyfikator bazujący na wspólnym liczniku dla wszystkich granatów
   ** unikalny identyfikator bazujący na wspólnym liczniku dla wszystkich nabojów
   *** bazując na dotychczasowej wiedzy zaproponować sensowne typy dla tych pól

Tam gdzie są wyszczególnione różne możliwe wartości, tam w ciele konstruktora dokonujemy losowania, wykorzystując do tego implementacje interfejsów funkcyjnych dostarczonych przez metody opisane w kolejnym punkcie.

2. Stworzyć klasę RandomSupplier posiadającą **raz** zainicjowane pole klasy Random z którego korzystają trzy metody statyczne zwracające implementacje interfejsów **java.util.function.Supplier** (jeżeli pasuje on i uda się go wykorzystać) lub zdefiniowanego przez siebie (jeśli Supplier nie podoła). Mają być one zaimplementowane rzecz jasna tylko za pomocą wyrażeń lambda. Zwrócone obiekty będą użyte przez konstruktory obiektów amunicyjnych (z poprzedniego punktu) w celu uzyskania wylosowanej wartości do inicjalizacji pól ("kaliber", "emisjaCO2" i "zabezpieczony"). Konstruktor wywołuje sobie interesującą go metodę, a następnie pobiera wartość ze zwróconego obiektu i inicjuje nią odpowiednie pole. Metody te są następujące:

   - provideRandomCaliberGenerator()
     Zwraca obiekt dostarczający wartość dla pola "kaliber".
   - provideRandomSafeGenerator(boolean alwaysUnlocked)
     Zwraca obiekt dostarczający wartość dla pola "zabezpieczony". Jeśli flaga alwaysUnlocked jest true, to zawsze jest generowana wartość "nie". W p.p. "tak" wypada z P=0.95. Konstruktor zawsze używa tej metody z flagą o wartości false. Oczywiście flaga ma być ustawialna na metodzie generatora.
   - provideRandomCO2EmissionGenerator(int a, int b)
     Zwraca obiekt dostarczający wartość dla pola "emisjaCO2". Losowana spomiędzy wartości dostarczonych jako argumenty metody generatora.

3. Stworzyć klasę AmmoChest posiadającą listę losowo pomieszanych obiektów amunicyjnych, która zawiera po 20 granatów (na każdy typ) oraz 2000 nabojów.

   Klasa zawiera poniżej wylistowane metody niestatyczne, z których każda wykonuje swoje zadanie bazując na iterowaniu wspomnianej listy za pomocą forEach() i dostarczając dla tej metody pożądaną implementację interfejsu java.util.function.Consumer. Kod consumera **nie może** się odnosić do elementów na zewnątrz jego bloku**. Jeżeli jest możliwe żeby użyć tutaj wyrażenia lambda to trzeba to zrobić za jego pomocą. W p.p. użyć klasy anonimowej lub inej. Wszelkim wykrywaniem, kolekcjonowaniem danych i ich udostępnianiem metodzie zajmuje się kod implementacji Consumera.**

   - getEcoArmourPiercingGrenades() – zwraca kolekcję ekologicznych granatów przeciwpancernych, czyli o emisji nie większej niż 225.
   - findUnlockedGrenades() – wypisuje na konsoli "Uwaga", za każdym razem gdy natrafi na niezabezpieczony granat obronny lub zaczepny.
   - getSummarizedCaliber() – uzyskuje informację o sumarycznym kalibrze co najwyżej 100 pierwszych napotkanych nabojów i wypisuje ją od tego momentu na konsoli.

4. Stworzyć podklasę klasy AmmoChest, której niestatyczne metody również działają na zainicjowanej w nadklasie liście. One także wykonują swoje działania bazując na iterowaniu metodą forEach dziedziczonej kolekcji z przekazanym do niej zaimplementowanym consumerem, z tą różnicą że tym razem kod consumera **może** się odnosić do elementów na zewnątrz bloku ale za to **musi** być definiowany w momencie przekazywania jako argument i nie wcześniej. Przykład: forEach((x) -> {impl.}). **Tutaj używamy tylko wyrażeń lambda.**

- upgradeCaliber(newCaliber) – podczas iteracji w przypadku natrafienia na nabój o kalibrze większym niż 5.56, wstawiana jest nowa wartość kalibru pobrana z parametru metody upgradeCaliber.
- replaceLocked4All() – podmienia informację o zabezpieczeniu wszystkich granatów obronnych na wartość wylosowaną poza blokiem wyrażenia lambda.
- getSummarizedCO2Emission() - ustala zsumowaną emisję CO2 napotkanych granatów przeciwpancernych i wypisuje ją na konsolę.

5. W ciele podklasy klasy AmmoChest (z punktu 4):
   - stworzyć własny interfejs funkcyjny MyAmmoGetter, zawierający metodę getGeneralAmmoByIndex(int index), która zwraca element po indeksie
   - metodę createMyAmmoGetter zwracającą implementację w/w interfejsu z **wykorzystaniem mechanizmu referencji**, który to zapożyczy ją od metody get(int index) należącej do instancji listy z obiektami amunicyjnymi (w klasie AmmoChest). Chodzi o "podwójne dwukropki" ;)