Zaawansowane programowanie obiektowe
Lab. 8
(Refleksja, adnotacje)

1 (REFLEKSJA, 0.75 pkt.)
Proszę napisać klasę o nazwie MaxSearchAlgorithms, która będzie zawierała 3 algorytmy (jako osobne metody o domyślnej/pakietowej widoczności) szukania maksimum w tablicy liczb typu int: od lewej do prawej, od prawej do lewej oraz najpierw czytane elementy na indeksach parzystych, a następnie na nieparzystych (uwaga: nie należy zakładać, że tablica przekazana jako argument zawiera parzystą bądź nieparzystą liczbę elementów). Wszystkie te 3 funkcje mają zwracać tablicę/listę (np. ArrayList) elementów, które były „chwilowymi" elementami maksymalnymi przy odnośnej strategii skanowania.

Przykład: jeśli na wejściu funkcji skanującej od prawej do lewej będzie tablica {4, 10, 3, 7, 4, 1, 6, 2}, to funkcja ma zwrócić tablicę {2, 6, 7, 10}.

Opisane 3 metody mają być wywołane na rzecz obiektu klasy MaxSearchAlgorithms utworzonego w innej klasie, za pomocą mechanizmu refleksji. Należy mianowicie odczytać wszystkie metody zdefiniowane w tej klasie, a następnie „odfiltrować" te, które nie zawierają w nazwie ciągu „Scan" (opisane wyżej 3 metody będą taki ciąg znaków w nazwie posiadały). Metody zawierające ww. napis mają zostać uruchomione (metoda invoke(...) z klasy Method), a zwrócone przez nie tablice wypisane na ekran.

2 (REFLEKSJA I ADNOTACJE, 1 pkt)
Utwórz klasę posiadającą przynajmniej 4 pola (prywatne) różnych typów (w tym prymitywnych), np. int, String, boolean...
Wygeneruj, przy użyciu IDE (Eclipse lub IntelliJ), settery i gettery dla tych pól.
Następnie napisz na dwa sposoby metodę equals(...) porównującą obiekty Twojej klasy, uwzględniając wszystkie pola z wyjątkiem jednego (np. przy klasie Osoba można zignorować pole telefon):
rozwiązanie tradycyjne („ręczne" porównywanie pól),
z użyciem refleksji i adnotacji.

Ad a) Opatrz metodę equals(...) odpowiednią adnotacją oznaczającą metodę przeciążoną (z Object).

Ad b) Dodaj własny typ adnotacyjny:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface IgnoreEquals {}
```

a następnie wykorzystaj go przy wybranym polu stosując odpowiednią metodę z klasy Method.

Rozwiązanie przetestuj.

3. (REFLEKSJA, 1.25 pkt)

Napisz aplikację GUI (dalsze odwołania dotyczą biblioteki Swing, ale możesz równie dobrze wykorzystać JavaFX), zawierającą w dolnym (BorderLayout.SOUTH) panelu trzy suwaki (JSlider) odpowiedzialne za składowe barwne (R, G, B) koloru głównego panelu aplikacji, który to kolor ma się zmieniać na bieżąco wraz z użyciem suwaków. Ustaw zakres 0..255 i wartości startowe 130 dla suwaków oraz wyświetl (niezbyt gęstą) podziałkę.

Dodatkowo, dolny panel ma również zawierać pole wyboru (*checkbox*) z etykietą „Show color name"; jeśli pole jest ustawione, to poniżej ma się wyświetlać angielska nazwa koloru, z repertuaru nazw kolorów określonych w java.awt.Color (czyli: BLACK, BLUE, CYAN... – łącznie 13 kolorów). Etykieta ta ma być najlepszym przybliżeniem (*) aktualnie wybranego (przy pomocy suwaków) koloru. Na koniec (jeśli program działa ☺) zastąp jeden z suwaków pokrętłem (JSpinner). Parametry analogiczne (zakres 0..255, wartość startowa 130), dodatkowo krok: 5. Wskazówka: w konstruktorze JSpinnera przekaż odpowiedni model.

(*) Zastosowany algorytm wyboru nazwy koloru jest bardzo prosty (i zapewne zawodny, bo nie uwzględnia on percepcji człowieka). Dla aktualnego koloru liczymy jego odległości w przestrzeni cech sRGB od 13 kolorów z java.awt.Color (WYKORZYSTAJ REFLEKSJĘ ORAZ METODY getRed(), getGreen(), getBlue()) i wybieramy kolor najbliższy. Przy liczeniu odległości stosujemy metrykę miejską (suma modułów różnic po składowych).

Przykład liczenia odległości: kolor1 = (50, 200, 100), kolor2 = (30, 240, 120). Odległość miejska między kolor1 i kolor2 = |50–30| + |200–240| + |100–120| = 80.
Remisy (dwa najbliższe kolory równie odległe) rozstrzygaj w dowolny sposób.