Kolory

W załączonym pliku `ColorDisplayer.java` znajduje się kod umożliwiający wyświetlenie listy kolorów w postaci kolorowych klastrów. Program oczekuje, że kolory będą reprezentowane w formacie **HSV** (Hue, Saturation, Value, więcej informacji: https://pl.wikipedia.org/wiki/HSV_(grafika)). Wykonaj poniższe polecenia i pamiętaj, że nie musisz wnikać w przygotowany kod tworzący kontrolki graficzne. W całej klasie będzie Cię interesować jedynie metoda:

```
private List<HSVColor> createColorsToDisplay() {
	// Implement me :(
}
```

1. Stwórz klasę **HSVColor** i wszystkie potrzebne atrybuty:
   - **hue** (odcień) - reprezentowany jako liczba stopni <0, 360>
   - **saturation** (nasycenie) - promień reprezentowany jako liczba rzeczywista z zakresu <0, 1>
   - **value** (jasność) - wysokość stożka, reprezentowana jako liczba rzeczywista z zakresu <0, 1>
2. Pamiętaj żeby zdefiniować odpowiedni konstruktor oraz gettery.
3. W klasie `ColorDisplayer` w metodzie `createColorsToDisplay()` stwórz i zwróć listę 20 losowych kolorów. Pamiętaj by dla każdej składowej losować wartość z odpowiedniego zakresu!
4. Jeśli do tej pory wszystko zostało wykonane poprawnie, po uruchomieniu programu powinien się pokazać pasek z 20 komórkami z losowo wybranymi kolorami.
5. Zmodyfikuj klasę `HSVColor` oraz metodę `createColorsToDisplay()` tak by wyświetlane kolory były posortowane od najjaśniejszego do najciemniejszego. Wykorzystaj interfejs `Comparable`.
6. (*) Zamiast ograniczać sortowanie kolorów zawsze do wartości V, stwórz 3 różne klasy **komparatorów** - każda będzie pozwalała na sortowanie kolorów po innej składowej. Wskazówka: komparatora można użyć wykonując sortowanie bezpośrednio na liście, np. `colorsList.sort(comparator)`.

Przydatne informacje

Porządek obiektów

Intefejs `Comparable`

Narzędzia programowania obiektowego takie jak polimorfizm i interfejsy służą nam nie tylko do tworzenia zupełnie nowych abstrakcji, ale również do rozszerzania istaniejących,

np. w bibliotece standardowej Javy.

Wyobraźmy sobie na przykład, że chcemy uporządkować obiekty naszej klasy według jakichś wytycznych. Jeśli nasz typ danych jest skonstruowany tak, że można wprowadzić relację porządku między obiektami tego typu to znaczy, że obiekty te są **porównywalne (ang. comparable)**. Tego typu cecha wyróżnia nie tylko nasz typ danych, ale i wiele innych. Korzystając z faktu, że coś jest porównywalne możemy na przykład posortować takie elementy. Zwrócmy uwagę, że mechanizm sortujący nie musi nic więcej wiedzieć o takich obiektach, wystarczy że wie jak je porównać.

Twórcy biblioteki standardowej Javy przewidzieli tego typu sytuacje i przygotowali interfejs **Comparable**, który wygląda następująco:

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

Interfejs określa, że jeśli chcemy by coś było porównywalne, musimy dostarczyć implementację dla metody `compareTo()`, która porówna nasz obiekt do jakiegoś innego. Można tu dostrzec analogię do metody `equals()`, przy czym pamiętajmy że `equals()` jest zdefiniowane w klasie `Object` więc dziedziczymy je w każdej klasie, którą definiujemy. Oznacza to, że można sprawdzić, czy dowolne dwa obiekty są równoważne, ale stwierdzić który jest większy tylko dla tych, których typy implementują `Comparable`.

Przewidywane zachowanie metody `compareTo` zostało opisane w oficjalnej dokumentacji. W dużym skrócie - jeśli nasz obiekt jest większy niż porównywany metoda powinna zwrócić wartość większą od 0. Jeśli jest mniejszy - wartość mniejszą od 0. Jeśli równy - 0.

Przykład:

```
public class Book implements Comparable<Book> {

  private int releaseYear;

  private String title;

  . . .

  @Override
  public int compareTo(Book otherBook) {
     return this.releaseYear - otherBook.releaseYear;
  }
}
```

W klasie reprezentującej książkę mamy m. in. atrybut oznaczający rok wydania książki. Klasa deklaruje, że jej obiekty są porównywalne i jednocześnie dostarcza logikę, według której książka "jest większa" od innej gdy jej rok wydania jest późniejszy.

Zakładając, że posiadamy gdzieś listę książkę o nazwie `books` możemy ją posortować używając polecenia:

```
Collections.sort(books);
```

Sortowanie będzie korzystać z przygotowanej przez nas metody porównującej.

Intefejs `Comparator`

Czasem nie chcemy umieszczać logiki porównywania elementów bezpośrednio w klasie, którą chcemy porównywać. Może się tak zdarzyć np. w sytuacji gdy jest kilka różnych sposobów porównywania obiektów danej klasy. Wówczas możemy zaimplementować tzw. `Comparator`, czyli klasę która służy do porównywania obiektów. Odnosząc się do przykładu powyżej:

```
public class BookReleaseYearComparator implements Comparator<Book> {

    @Override
    public int compare(Book book1, Book book2) {
        return book1.getReleaseYear() - book2.getReleaseYear();
    }
}
```

W jaki sposób możemy wykorzystać taki komparator? Tym razem wywołamy sortowanie bezpośrednio na liście:

```
books.sort(new BookReleaseYearComparator());
```