Zadanie 1. IMPLEMENTACJA WZORCA PRODUCENT – KONSUMENT TWORZĄCEGO STATYSTYKI WYRAZÓW

Zaimplementuj aplikację, która będzie analizowała zawartość plików znajdujących się we wskazanym katalogu głównym. W zadaniu należy skorzystać z kolejki blokującej np. *LinkedBlockingQueue* (z pakietu *java.util.concurrent*) o ograniczanej parametrem pojemności. Jeden wątek (producent) powinien przechodzić przez drzewo podkatalogów i plików wyszukując pliki tekstowe, a także wstawiać ścieżki na nie wskazujące do kolejki blokującej (ścieżka powinna być obiektem klasy *java.nio.file.Path* opakowanym w *java.util.Optional* np. *Optional<Path> optPath = Optional.ofNullable(path);*). Kolejne dwa wątki (konsumenci) będą pobierały/usuwały obiekty z kolejki i wyszukiwały dziesięć najczęściej występujących wyrazów we wskazanych plikach. W tego typu zadaniu prawdopodobnie wątek producenta szybko wypełni kolejkę obiektami wskazującymi konkretne pliki i zostanie zablokowany do czasu, gdy wątki odbierające elementy wykonają swoją pracę i ponownie opróżnią zapełnioną kolejkę.

Do tworzenia statystyki wyrazów można utworzyć metodę przetwarzająca dane za pomocą strumieni np.

```
private Map<String, Long> getLinkedCountedWords(Path path, Long wordsLimit) throws IOException {
   try (BufferedReader reader = Files.newBufferedReader(path)) {//lub ...(path, StandardCharsets.UTF_8)
      return reader                                             //bez buforowania - Files.readAllLines(path)
         .lines()
          // TODO ...
         .collect(Collectors.toMap(...);
   }
}
```

Zliczanie słów musi być niewrażliwe na wielkość liter i uwzględniać tylko słowa dłuższe od dwóch znaków (liczby, znaki specjalne i różne ich kombinacje powinny być pomijane). Znalezione słowa powinny być posortowane względem częstotliwości ich występowania w kolejności malejącej i umieszczone w zwracanej mapie typu *LinkedHashMap* (ten typ mapy zachowa kolejność posortowanych danych).

Przykładowy wynik umieszczony w mapie: `{git=224, data=118, version=102, stored=96, ...`

Producent kończąc pracę musi umieścić w kolejce specjalne obiekty tzw. *poison pills*, które przekażą konsumentom informację o konieczności zakończenia ich pracy. Można do tego celu wykorzystać obiekt *Optional.ofNullable(**null**)*, poniżej przykładowy fragment kodu konsumenta wykrywający takie obiekty.

```
// WĄTEK KONSUMENTA
while(...){
      ...
   // pobieranie (lub oczekiwanie) obiektu z kolejki
   Optional<Path> optPath = ... ;
      ...
   if(optPath.isPresent()) {
      // przetwarzanie pliku tekstowego ...
      Map<String, Long> countedWords = getCountedWords(optPath.get(), 10); // dla 10 słów
      ...
   }else{
      //poison pill – trzeba zakończyć wątek
      break;
   }
      ...
}
```

Wszystkie wątki powinny być uruchamiane za pomocą wykonawców np.

```
final ExecutorService executor = Executors.newFixedThreadPool(liczbaProducentow + liczbaKonsumentow);
```

Do zatrzymywania producenta można wykorzystać zmienną typu *AtomicBoolean* (z pakietu *java.util.concurrent.atomic*) lub ewentualnie zadeklarować zmienną logiczną jako *volatile*.

```java
package com.lab.statistics;

import java.awt.BorderLayout;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.FileVisitResult;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.PathMatcher;
import java.nio.file.Paths;
import java.nio.file.SimpleFileVisitor;
import java.nio.file.attribute.BasicFileAttributes;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.Stream;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.UIManager;

public class MainFrame {

   private JFrame frame;
   // ścieżka do katalogu z plikami tekstowymi
   private static final String DIR_PATH = "files";
   // określa ile najczęściej występujących wyrazów bierzemy pod uwagę
   private final int liczbaWyrazowStatystyki;
   private final AtomicBoolean fajrant;
   private final int liczbaProducentow;
   private final int liczbaKonsumentow;
   // pula wątków – obiekt klasy ExecutorService, który zarządza tworzeniem
   // nowych oraz wykonuje 'recykling' zakończonych wątków
   private ExecutorService executor;
   // lista obiektów klasy Future, dzięki którym mamy możliwość nadzoru pracy wątków
   // producenckich np. sprawdzania czy wątek jest aktywny lub jego anulowania/przerywania
   private List<Future<?>> producentFuture;

   public static void main(String[] args) {
      try {
         UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
      } catch (Exception e) {
         e.printStackTrace();
      }
      EventQueue.invokeLater(new Runnable() {
         public void run() {
            try {
               MainFrame window = new MainFrame();
               window.frame.pack();
               window.frame.setAlwaysOnTop(true);
               window.frame.setVisible(true);
            } catch (Exception e) {
               e.printStackTrace();
            }
         }
      });
   }
```

```
public MainFrame() {
   liczbaWyrazowStatystyki = 10;
   fajrant = new AtomicBoolean(false);
   liczbaProducentow = 1;
   liczbaKonsumentow = 2;
   executor = Executors.newFixedThreadPool(liczbaProducentow + liczbaKonsumentow);
   producentFuture = new ArrayList<>();
   initialize();
}

/**
 * Initialize the contents of the frame.
 */
private void initialize() {
   frame = new JFrame();
   frame.addWindowListener(new WindowAdapter() {
      @Override
      public void windowClosing(WindowEvent e) {
         executor.shutdownNow();
      }
   });
   frame.setBounds(100, 100, 450, 300);
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   JPanel panel = new JPanel();
   frame.getContentPane().add(panel, BorderLayout.NORTH);
   JButton btnStop = new JButton("Stop");
   btnStop.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         fajrant.set(true);
         for (Future<?> f : producentFuture) {
            f.cancel(true);
         }
      }
   });
   JButton btnStart = new JButton("Start");
   btnStart.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         multiThreadedStatistics();
      }
   });
   JButton btnZamknij = new JButton("Zamknij");
   btnZamknij.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         executor.shutdownNow();
         System.exit(0);
      }
   });
   panel.add(btnStart);
   panel.add(btnStop);
   panel.add(btnZamknij);
}

 /**
  * Statystyka wyrazów (wzorzec PRODUCENT - KONSUMENT korzystający z kolejki blokującej)
  */
 private void multiThreadedStatistics() {
    for (Future<?> f : producentFuture) {
      if (!f.isDone()) {
         JOptionPane.showMessageDialog(frame, "Nie można uruchomić nowego zadania!
                                    Przynajmniej jeden producent nadal działa!", "OSTRZEŻENIE", JOptionPane.WARNING_MESSAGE);
         return;
      }
    }
    fajrant.set(false);
    producentFuture.clear();

    final BlockingQueue<Optional<Path>> kolejka = new LinkedBlockingQueue<>(liczbaKonsumentow);
    final int przerwa = 60;

    Runnable producent = () -> {
       final String name = Thread.currentThread().getName();
       String info = String.format("PRODUCENT %s URUCHOMIONY ...", name);
       System.out.println(info);

      while (!Thread.currentThread().isInterrupted()) {
            if(fajrant.get()) {

                // TODO przekazanie poison pills konsumentom i zakończenia działania

            } else {

                // TODO Wyszukiwanie plików *.txt i wstawianie do kolejki ścieżki (w Optional) lub oczekiwanie jeśli kolejka
                // pełna, do wyszukiwania plików można użyć metody Files.walkFileTree oraz klasy SimpleFileVisitor<Path>

            }
```

```
            info = String.format("Producent %s ponownie sprawdzi katalogi za %d sekund", name, przerwa);
            System.out.println(info);
            try {
               TimeUnit.SECONDS.sleep(przerwa);
            } catch (InterruptedException e) {
               info = String.format("Przerwa producenta %s przerwana!", name);
               System.out.println(info);
               if(!fajrant.get())
                  Thread.currentThread().interrupt();
            }
         }
         info = String.format("PRODUCENT %s SKOŃCZYŁ PRACĘ", name);
         System.out.println(info);
      };

       Runnable konsument = () -> {
          final String name = Thread.currentThread().getName();
          String info = String.format("KONSUMENT %s URUCHOMIONY ...", name);
          System.out.println(info);

          while (!Thread.currentThread().isInterrupted()) {
             try {

                // TODO pobieranie ścieżki i tworzenie statystyki wyrazów
                // lub oczekiwanie jeśli kolejka jest pusta

            } catch (InterruptedException e) {
               info = String.format("Oczekiwanie konsumenta %s na nowy element z kolejki przerwane!", name);
               System.out.println(info);
               Thread.currentThread().interrupt();
            }
         }
         info = String.format("KONSUMENT %s ZAKOŃCZYŁ PRACĘ", name);
         System.out.println(info);
      };

      //uruchamianie wszystkich wątków-producentów
      for (int i = 0; i < liczbaProducentow; i++) {
         Future<?> pf = executor.submit(producent);
         producentFuture.add(pf);
      }
      //uruchamianie wszystkich wątków-konsumentów
      for (int i = 0; i < liczbaKonsumentow; i++) {
         executor.execute(konsument);
      }
   }

   /** Metoda zwraca najczęściej występujące słowa we wskazanym pliku tekstowym
    */
   private Map<String, Long> getLinkedCountedWords(Path path, int wordsLimit) throws IOException {
      //konstrukcja 'try-with-resources' - z automatycznym zamykaniem strumienia/źródła danych
      try (BufferedReader reader = Files.newBufferedReader(path)) {// wersja ze wskazaniem kodowania
                                                                   // Files.newBufferedReader(path, StandardCharsets.UTF_8)
           return reader.lines() // można też bez buforowania - Files.readAllLines(path)
            //TODO
            // 1. podział linii na słowa, można skorzystać z wyrażeń regularnych np. "\\s+"
            // 2. filtrowanie słów - tylko z przynajmniej trzema znakami, użyj wyrażenia
            //    regularnego np. "[a-zA-Z]{3,}" (nie uwzględnia polskich znaków)
            // 3. konwersja do małych liter, aby porównywanie słów było niewrażliwe na wielkość liter
            // 4. grupowanie słów względem liczebności ich występowania, można użyć
            //    metody collect z Collectors.groupingBy(Function.identity(), Collectors.counting()),
            //    po tej operacji należy zrobić konwersję na strumień tj. entrySet().stream()
            // 5. sortowanie względem przechowywanych w mapie wartości, w kolejności malejącej,
            //    można użyć Map.Entry.comparingByValue(Comparator.reverseOrder())
            // 6. ograniczenie liczby słów do wartości z wordsLimit
             .collect(Collectors.toMap( //umieszczenie elementów strumienia w mapie zachowującej kolejność tj. LinkedHashMap
                   Map.Entry::getKey,
                   Map.Entry::getValue,
                    (k,v) -> { throw new IllegalStateException(String.format("Błąd! Duplikat klucza %s.", k)); },
                    LinkedHashMap::new));
      }
   }
}
```