הקדמה:

בתרגיל מסכם זה נכתוב פרויקט קטן שנשלב בתוכו את תבניות העיצוב ותבניות ה concurrency הבאות:

- Thread Pool – Fork-Join Pool עם משימה מקבילית רקורסיבית
- Proxy
- Flyweight
- Decorator
- Observer
- Thread Safe Singleton

עליכם לכתוב לעצמכם את הבדיקות (כל קבוצה לעצמה) וכאשר אתם חושבים שהפרויקט מוכן תוכלו להגיש אותו לתיבת ההגשה. בנוסף לקבצי הפרויקט, תצטרכו גם להגיש את main הבדיקות שעשיתם והוא מהווה חלק מסוים מהציון לתרגיל.

בהצלחה!

תרגיל מסכם

ברצוננו לממש מנוע לחיפוש טקסט בתוך תיקיית קבצים.

כמובן, התיקייה עשויה להכיל תת-תיקיות שבתורן מכילות תת-תיקיות וכו' ובתוכן קבצים רבים.

בהינתן מחרוזת ותיקיית שורש נרצה לקבל בחזרה מפה (MAP) כאשר

- המפתח הוא שם הקובץ המלא (כולל ה path המלא, לדוגמה "/eli/kh/myText.txt")
- והערך הוא קבוצה של כל השורות מתוך הקובץ שבהן המחרוזת הופיעה.

נרצה לעשות זאת בצורה חכמה. מצד אחד לא נרצה לשמור את כל האינפורמציה של מערכת הקבצים בזיכרון (כלומר ב RAM), ומצד שני לא נרצה בהכרח על כל חיפוש לבצע כל כך הרבה IO עבור קריאת הקבצים.

תוצאת החיפוש צריכה להינתן בתוך אובייקט מסוג Result שזהו הממשק שלו:

```
public interface Result {
      String getQuery(); // the searched string
      // Map< file name, Set <lines the query appeared in> >
      Map<String,Set<String>> getAnswer();
}
```

אלגוריתם חיפוש צריך לממש את הממשק הבא:

```
public interface TextSearcher {
      Result search(String text, String rootPath);
}
```

טיפ: כדי לחקור תיקייה תוכלו ליצור אובייקט מסוג File עבור שם התיקייה.

משימה א'

תחילה נממש את הפתרון הטריוויאלי. ממשו את המחלקה IOSearcher כסוג של TextSearcher, אשר עוברת על קובצי התיקייה rootPath (ובאופן רקורסיבי על כל תת-תיקייה), ומחפשת בתוכן של כל קובץ שהסיומת שלו היא "txt." את מחרוזת החיפוש ומחזירה Result.

משימה ב'

ממשו את המחלקה ParallelIOSearcher כסוג של IOSearcher. אובייקט זה ישתמש ב fork join pool וב recursive task כדי לבצע את החיפוש בכל תת-תיקייה במקביל כמשימה של ה thread pool. כאשר החיפוש מסתיים כל הת'רדים צריכים להיסגר בצורה מסודרת.

משימה ג'

ממשו את המחלקה CacheIOSearcher כ Proxy של IOSearcher. על אובייקט זה לנהל flyweight של אובייקטים מסוג Result. דהיינו, אם כבר בוצע חיפוש מסוים בעבר אז התוצאה תישלף מהזיכרון ב O(1) זמן, במקום לבצע חיפוש תלוי IO. אחרת, יתבצע חיפוש תלוי IO והתוצאה תישמר בזיכרון להבא.

מחלקה זו תממש את הממשק CacheSearcher שהרחיבה את TextSearcher ע"י המתודות הבאות:

- getCachedResults():Set<Result> אשר תחזיר לנו את קבוצת כל ה Results השמורים בזיכרון.
- clear():void אשר מנקה את כל התוצאות שנשמרו בזיכרון.
- remove(:Result):void אשר בהינתן תוצאה היא תסיר אותה מהזיכרון.

נשים לב שה CacheIOSearcher יכול לעטוף גם IOSearcher וגם ParallelIOSearcher.

נשים לב לעוד תופעה:

לאחר מספיק חיפושים, ה CacheIOSearcher עלול לשמור בזיכרון כמות גדולה יותר מסכום כל הטקסט שבקבצים בדיסק. הרי אותה השורה עלולה להישמר שוב ושוב עבור כל אחת מהמילים הייחודיות שנמצאות בה אם חיפשנו אותן. ולא באמת ניתן לסמוך על הלקוח שלנו שיבצע clear() בכל פעם שהזיכרון מתמלא.

משימה ד'

ממשו את המחלקה LRUCacheSearcher כ Decorator של CacheSearcher. מחלקה זו תשתמש באובייקט מסוג CacheSearcher (המתקבל כפרמטר הראשון בבנאי) כדי לבצע את החיפוש. אולם, כאשר כמות האובייקטים ב cache מגיעה לערך סף (מסוג int, המתקבל כפרמטר השני בבנאי) אז LRUCacheSearcher תסיר מה cache את התוצאה שהיא ה least recently used, כלומר התוצאה עבור השאילתה הישנה ביותר תוסר מה cache.

למען הסר ספק, ברגע שבוצע חיפוש חוזר של מחרוזת כלשהי אז היא כבר לא הישנה ביותר, כי הרגע חיפשו אותה.

משימה ה'

באופן דומה, ממשו את המחלקה LFUCacheSearcher כ Decorator של CacheSearcher. אלא שהפעם ברגע שכמות ה Results ב cache עוברת את הסף, אז מחלקה זו תסיר את ה Result שחיפשו אותה הכי פחות פעמים. אם כמות החיפושים שווה אצל כל ה Results אז יש להסיר את ה Least Recently Used.

משימה ו'

באופן דומה ממשו את המחלקה ObservableCacheSearcher כ Decorator של CacheSearcher. הפעם מחלקה זו תהיה חלק מה Observer Design Pattern. כידוע, לאחר כל חיפוש ייתכן שינוי ב cache (הוספה או הסרה של result). כ Observable, מחלקה זו תשלח נוטיפיקציה בכל פעם שהיה שינוי ב cache של האובייקט שעטפה.

כזכור הנוטיפיקציה כוללת שני פרמטרים: מיהו ה Observable שהפעיל אותה, ופרמטר "אודות". פרמטר האודות יהיה מסוג String ויכלול את ה query של ה Result, רווח, ואז המילה added או removed בהתאמה למה שקרה לאותה ה Result. במקרה של גם הסרה וגם הוספה באותו האירוע, תחילה יופיע טקסט ה removed, ואז נקודה-פסיק ";", ואז הטקסט של ה added. לדוגמה:

Hello removed;world added

הערה: יש להשתמש במחלקה Observable וממשק Observer הקיימים ב Java. אם אתם מקבלים אזהרה שהם deprecated אז תתעלמו \ תגדירו את תאימות הפרויקט לגרסה 1.8.

משימה ז'

ממשו את המחלקה Logger כסוג של Observer. היא תקבל בבנאי שם קובץ (מסוג String) ומופע של ObservableCacheSearcher. ה Logger ירשם כ observer של ה ObservableCacheSearcher. כל נוטיפיקציה שתתקבל ממנו תישמר בקובץ באופן הבא:

פרמטר האודות, רווח, הגודל הנוכחי של ה cache. לדוגמה:

```
Hello added 1

World added 2

…

Nice added 400

Hello removed;Work added 400

World removed;Bye added 400
```

בנוסף על ה Logger להיות Thread Safe Singleton. הדרך תקבל מופע תהיה דרך המתודה הסטטית:

Logger logger = Logger.getInstance();

משימה ח'

כתבו מחלקה בשם Test עם מתודת main. שם עליכם לכתוב בדיקות לכל אחת מהמחלקות לעיל. כמובן, ניתן ורצוי להשתמש במתודות עזר לשם כך. כל בדיקה שנכשלת תדפיס למסך הודעת שגיאה מתאימה.

חישבו לא רק על בדיקת תוצאות הריצה (החזרה של תוצאות חיפוש נכונות) אלא גם על בדיקות design. תוכלו לחלץ מכל אובייקט משתנה מסוג Class<> ע"י getClass() ולחלץ משם אינפורמציה אודות הטיפוס שלו. את מי ירש \ מימש, את מי מכיל, אלו מתודות יש לו וכו'.

כאמור, חלק מהציון גם ניתן גם על אופי ואיכות הבדיקות ולכן כל אחד כותב את הבדיקות לעצמו.

הגשה

עליכם להגיש את הקבצים הבאים:

- IOSearcher.java
- ParallelIOSearcher.java
- CacheIOSearcher.java
- LRUCacheSearcher.java
- LFUCacheSearcher.java
- ObservableCacheSearcher.java
- Logger.java
- Test.java

ואותם בלבד. תלכו ליצור מחלקות עזר כמחלקות פנימיות.

כל המחלקות צריכות להיות מוגדרות בתוך package בשם **test**.

בהצלחה!