עבודה עצמית שבוע 10 (רשות!) – Logging

מדוע חשוב לנו לנהל את הלוגים שלנו נכון?

במערכת תוכנה גדולה, נרצה לייצר פלט (למטרות דיבאג או מעקב) ל console. הדרך הנאיבית ביותר היא שימוש ב System.out.println, אבל זה כמובן פתרון בעייתי. מה יקרה אם נרצה לכבות חלק מההדפסות? (כי הן כבר לא רלוונטיות עבורנו בשלב זה של פיתוח התוכנה). ומה יקרה אם נרצה להמיר את ההדפסות ל console לכתיבה לקובץ?

מסיבות אלה, מערכות גדולות משתמשות בכל מני כלי Logging המאפשרים לנהל את רמת הפירוט של כל לוג וגם את פורמט הכתיבה. בתרגול זה נשתמש ב Log4J – חבילה מאוד פופולרית לניהול לוגים ב Java.

העקרונות שנראה פה הם כמובן משותפים לכל חבילת ניהול לוגים, גם ב Java וגם בשפות תכנות אחרות.

כשנדבר על לוגים, נרצה לדבר על כמה מאפיינים:

1. היררכיית כתיבה – נרצה לחלק את הפלט שלנו לכמה רמות חומרה, ולהחליט החל מאיזו רמת חומרה אנחנו מעוניינים לתעד. בשלבים מוקדמים של פיתוח המערכת נרצה אולי לתעד הכל. בשלבים מאוחרים יותר נרצה לתעד רק אירועים "חריגים" או "חשובים ביותר".
2. פורמט – באיזה פורמט נכתב הפלט. לדוגמא – בכל הדפסה ללוג היינו רוצים להדפיס גם את חותמת הזמן שבו היא קרתה.
3. לאן הפלט יוצא – מסך? קובץ? גם וגם?

התקנה:

הקובץ הבא מכיל שלושה קבצים: שני קבצי jar וקובץ קונפיגורציה (עם סיומת xml) אחד.
http://courses.cs.tau.ac.il/software1/2021a/lectures/tutorials/resources/tutorial_10_files.zip

את קבצי ה jar עליכם להוסיף ל classpath של ריצת התוכנית כפי שכבר ראינו בקורס, ואת קובץ הקונפיגורציה עלינו להעביר כפרמטר זמן ריצה ל JVM באחד מהאופנים הבאים:

בשורת הפקודה: אם נרצה להריץ את התוכנית MyProg, נעביר את הפרמטר כך

```
java -Dlog4j.configurationFile=path_to_log4j.xml_file MyProg
```

ב eclipse, תחת VM_aguments (באותו המסך שבו אנחנו מעבירים את הארגומנטים של שורת הפקודה) נוסיף -Dlog4j.configurationFile=path_to_log4j.xml_file

path_to_log4.xml_file מציין את המיקום של הקובץ log4j.xml – מיקום אבסולוטי או יחסי (עליכם לתת המיקום של הקובץ log4j.xml אצלכם על המחשב).

בהמשך התרגול נסביר את תוכנו של הקובץ ואף נערוך אותו.

תוכנית ראשונה:

הריצו את התוכנית הבאה על פי ההנחיות שמופיעות בסעיף הקודם. אתם אמורים לקבל 4 הדפסות למסך

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Log4jExample {
	public static Logger log = LogManager.getRootLogger();

	public static void main(String[] args){
		log.debug("debug message");
		log.info("info message");
		log.warn("warning message");
		log.error("error message");
	}
}
```

ההדפסות שאנחנו מקבלים כוללות את תאריך ההדפסה, את שם הפונק', את חומרת (Level) ההודעה ואת התוכן עצמו.

בתוכנית זו השתמשנו ב 4 רמות שונות: debug/info/warn/error. עבור כל logger אנחנו יכולים להגדיר את רמת ה"חומרה" שהחל ממנה הוא מופעל. הרמות בתוכנית מסודרות לפי סדר חומרה עולה – כלומר, אם אנחנו רוצים להפעיל את ה logger עבור כל דבר שהוא info ומעלה, יודפסו 3 הקריאות התחתונות.
ב log4j קיימות רמות חומרה נוספות (הצגנו כאן את הכי נפוצות לשימוש), ואף ניתן להגדיר רמות חומרה חדשות. כל זה מאפשר לנו להיות גמישים יותר בקונפיגורציה – אם יש 10 קטגוריות חומרה, נוכל להגדיר את ההדפסה ללוגים ב 10 דרכים שונות.
לקריאה נוספת על רמות חומרה שונות ב log4j ניתן להיכנס לקישורית הבאה:
https://logging.apache.org/log4j/log4j-2.3/manual/customloglevels.html

המנגנון של שימוש ברמות חומרה קיים בכל מערכת ניהול לוגים, כאשר בספריות\שפות תכנות שונות הרמות יכולות להיקרא בשמות שונים, אבל העקרון הוא אחיד.

כיצד נשנה את רמת החומרה שבה ה logger מופעל כרגע? לצורך כך, עליכם לערוך את הקובץ log4j.xml.
ניתן לערוך אותו דרך ה eclipse או דרך גישה ישירה לקובץ ב file system.
חפשו את השורה
<"Root level="debug>
והמירו אותה ל
<"Root level="info>
וודאו שלאחר ההמרה, השורה הנכתבת ע"י log.debug אכן לא מופיעה בפלט (debug נחשבת רמת חומרה נמוכה יותר מ info).

העובדה שהשורה שנכתבת ע"י log.debug לא מופיעה בפלט היא כמובן משמחת, כיוון שזו ההתנהגות הרצויה. זה יאפשר לנו בעצם למלא את הקוד בהרבה הדפסות שיכולות להיות מעניינות, ולכבות אותן במקום אחד בלבד (קובץ הקונפיגורציה) מבלי לשנות שום דבר בקוד עצמו. יחד עם זאת, הפונקציה log.debug עדין נקראת, וגם אם היא לא עושה כלום, אנחנו עדין מבצעים קריאה לפונקציה ושולחים לה פרמטרים שיכולים להיות מחושבים בתהליך יקר.
נסתכל בדוגמא הבאה – החליפו את פונקציית ה main הקודמת בשתי הפונקציות המצורפות, והריצו את התוכנית.

```
public static String func(){
	System.out.println("calling func()");
	return "abc";
}

public static void main(String[] args){
	log.debug("debug message {}" , func());
	log.info("info message");
	log.warn("warning message");
	log.error("error message");
}
```

השימוש ב {} בגוף המחרוזת מאפשר לנו להעביר פרמטרים אשר יחליפו את הביטוי {}, מבלי להשתמש בשרשור מחרוזות עם + (שכבר ראיתם בתרגול שהוא לא יעיל).
אנחנו רואים שלמרות שה debug לא מבצע הדפסה, אנחנו צריכים לחשב את המחרוזת שהפונק' מקבלת כפרמטר, ולכן קוראים לפונקציה func. לו func היתה יקרה לחישוב, היינו מעוניינים להריץ אותה רק אם log.debug באמת עושה משהו עם המחרוזת שהוא מקבל.
ה"טריק" לחישוב עצל (lazy) של הפרמטר של הפונקציה הוא באמצעות שימוש בביטויי למבדא שראינו בשיעור שעבר.
נמיר את השורה הראשונה ב main לשורה הבאה:

```
log.debug("debug message {}", ()->func());
```

אנחנו רואים שהפרמטר לפונקציה debug הוא כבר לא הערך של func אלא ביטוי למבדא שמגדיר פונק' שקוראת ל func. הטריק הוא כמובן בכך שרק הגדרנו את הפונקציה אבל לא קראנו לה, ולכן func תיקרא רק אם אם debug תדע שהיא אמורה לייצר פלט.

המירו את הדפסת ה debug בשורה שעושה שימוש בביטוי למבדא וודאו שאכן לא מבוצעת קריאת ל func כשה logger מופעל מרמת חומר info ומעלה.
כאשר ה logger שלנו מכיל קריאות לשירותים "כבדים", צריך להשתמש בשיטה הזו בשביל לא לחשב תמיד את כל המחרוזות.

כעת, ננסה לשנות קצת את פורמט ההדפסה. נחזור ל log4j.xml ונשנה את השורה שבה מופיע PatternLayout לשורה הבאה.

```
      <PatternLayout pattern="%d{HH:mm:ss} [%C.%t:%L] %-5level
%logger{36} - %msg%n"/>
```

התוספות לפורמט צבועות באדום. כשנריץ את התוכנית נראה שכעת אנחנו מדפיסים את השם המלא של המחלקה (כולל החבילה), ובנוסף אנחנו מדפיסים גם את מספר השורה שבה נקראת ההדפסה.

אופציות נוספות לתבניות הפורמט של ה logger ניתן למצוא כאן (תחת Patterns):
https://logging.apache.org/log4j/2.x/manual/layouts.html

ניהול לוגים פר מחלקה:
נרצה להגדיר לוגים שונים למחלקות שונות. בפרט, נרצה שבחלק מהמחלקות רמת הפירוט תהיה אחת, ובשניה תהיה אחרת.
עד כה עבדנו עם RootLogger, וכעת נייצר לוגים עבור כל מחלקה בנפרד.

הוסיפו את המחלקה הבאה (לצורך הפשטות, ניתן להוסיף אותה לאותו הקובץ שבו מופיעה המחלקה Log4jExample. מכיוון שהיא אינה בניראות public, זה תקין)

```
class AnotherClass{
      public static Logger log = LogManager.getLogger(AnotherClass.class);
      public static void func(){
            log.debug("calling func");
      }
}
```

ל main של המחלקה Log4jExample נוסיף את השורה

```
            AnotherClass.func();
```

שימו לב שב AnotherClass נעשה שימוש ב Logger שאינו ה RootLogger. אנחנו מייצרים Logger חדש שיהיה יחודי למחלקה. ניתן להעביר לשירות getLogger גם מחרוזת, אבל זה שימוש פחות נפוץ.

עכשיו בעצם יש לנו שני logger-ים במערכת, אז אנחנו צריכים לחזור לקובץ הקונפיגורציה ולקנפג גם את logger השני (לכל logger קיימת קונפיגורציה דיפולטית, אבל אנחנו לא מעוניינים בה).
בקובץ log4j.xml נוסיף את השורות הבאות מתחת לשורה שמתחילה ב Root/:

```
    <Logger name="class10" level="debug" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
```

כעת, בתוך Loggers מוגדרים לנו 2 logger-ים. אחד שהוא Root עם רמת חומרה של info, והשני עבור class10 עם רמת חומרה של debug.
השם class10 הוא בעצם שם ה package וישפיע על כל ה logger-ים שהשם שלהם מתחיל בתחילית זו. השם של ה logger של AnotherClass הוא השם המלא של המחלקה, ומתחיל ב class10, ולכן ה logger הזה מתאים לו.

אם נריץ את התוכנית ניראה שההדפסות ב Log4jExample הן החל מרמת info, בעוד שההדפסות של AnotherClass כוללות גם את debug.

לאיזה צורך אנחנו מוסיפים את המאפיין "additivity="false?

החליפו את הקריאה ל debug בקריאה ל error בתוך הפונקציה func. כעת, גם ה RootLogger "מכסה" מבחינת רמת חומרה את הקריאה הזו. למרות שאנחנו לא עושים שימוש ב RootLogger ב AnotherClass, הוא עדין נקרא, כיוון שההוא ההורה של ה logger שלנו. אם נוסיף את המאפיין "additivity="false זה ימנע טיפול "כפול" של שני logger-ים שמתאימים.

שימוש ב appender-ים שונים.
נסתכל שוב על קובץ הקונפיגורציה שלנו

```
<Configuration>
  <Appenders>
    <Console name="Console">
      <PatternLayout pattern="%d{HH:mm:ss} [%C.%t:%L] %-5level %logger{36}
- %msg%n"/>
    </Console>

  </Appenders>
  <Loggers>
    <Root level="info">
      <AppenderRef ref="Console"/>
    </Root>
    <Logger name="class10" level="debug" additivity=”false”>
      <AppenderRef ref="Console"/>
    </Logger>
  </Loggers>
</Configuration>
```

כאשר הגדרנו את ה RootLogger ואת ה logger של class10, הגדרנו לכל אחד מהם appender בשם Console. אינטואיטיבית אנחנו מבינים שמדובר בהגדרה שמציינת שאנחנו כותבים את הפלט ל console. ה appender שנקרא Console מוגדר גם הוא בקובץ הקונפיגורציה (החלק העליון), ובו אנחנו מגדירים את פורמט הפלט. לו היינו רוצים להדפיס מתוך class10 בפורמט השונה מזה של ה RootLogger, היינו צריכים להגדיר appender נוסף, לתת לו שם אחר (למשל ConsoleClass10) ולהתייחס אליו בהגדרה של ה logger.

ומה אם נרצה שהלוג שלנו יכתב לקובץ?
לצורך כך, נצטרך להגדיר appender מסוג חדש.
בקובץ הקונפיגורציה, תחת Appenders, נוסיף:

```
    <File name="FileClass10" fileName="./class10.log">
      <PatternLayout pattern="%d{HH:mm:ss} [%C.%t:%L] %-5level %logger{36}
- %msg%n"/>
    </File>
```

ונעדכן את ה logger ששמו class10 כך שה AppenderRef יכיל את שמו של ה appender החדש שהוספנו, כלומר FileClass10.

כשנריץ את התוכנית נגלה שההדפסה מתוך AnotherClass כבר אינה מופיעה ב console. במקום זאת, הופיע קובץ בשם class10.log תחת תיקיית הפרוייקט, והוא מכיל את ההדפסה שלנו. כמובן שבמערכת אמיתית מומלץ לכתוב את כל הלוגים לתוך תיקיה יעודית.

ל log4j יש עוד יכולות רבות ומגוונות. למשל, ניתן לכתוב לוגים בפורמט xml. דוגמא נוספת: באמצעות ה appender שנקרא RollingFile ניתן לייצר לוגים שיודעים להתפצל למספר קבצים כתלות בגודל\זמן, ואף לקבוע את מדיניות מתן השמות (למשל, בכל פעם שגודל הלוג עובר גודל מסויים הוא מועתק הצידה). בתרגול זה הראינו ממש טעימה מהבסיס בשביל להציג את העקרונות עליהם נשענת עבודה עם logger-ים.
ניתן להרחיב בקריאה ולראות דוגמאות נוספות בקישור הבא:
https://logging.apache.org/log4j/2.0/manual/api.html