

BYTECODE



תרגול 6 - הנדסה לאחר - אביב תשפ"ג
©אלעד קינסברונר



מה זה?

- קבצי הרצה הם מאוד רגישים לארכיטקטורה...
 - למעבדים שונים יש instruction set שונה
 - כמות רגיסטרים שונה
 - גודל כתובת שונה: 16 או 32 או 64 ביט
 - לכל מערכת הפעלה יש פורמט קבצי הרצה משלה: PE, ELF, MachO
- למה שלא ניצור פורמט אחיד לקוד, ואחרי זה כל מעבד יפרש אותו?
 - זה הרעיון מאחורי Bytecode!
- קיימים כמה סוגי Bytecode, שכל אחד מהם מתאים למכונה וירטואלית משלו.
- Java Bytecode שרץ על ה-JVM הוא המוכר ביותר

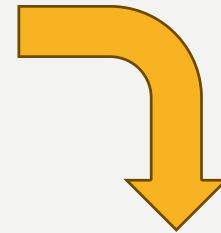
יתרונות וחסרונות

- Bytecode בדור"כ מפוענח בזמן ריצה באמצעות תוכנה הנקראת VM
 - לפעמים ה-VM רק מפענח ומריץ, לפעמים הוא מבצע JIT
- למה שנרצה כזה דבר?
 - תאימות עם כל ארכיטקטורה שנרצה, כולל ארכיטקטורות שעוד לא הומצאו
 - ה-VM מספק שירותים נוספים כמו garbage collection
 - מאפשר מודלי זיכרון יותר נוחים
- למה אולי לא?
 - זה יותר איטי – המעבד צריך לעשות משהו יותר מורכב מפשוט להריץ פקודות
 - עדיין צריך לקמפל VM מיוחד לכל ארכיטקטורה בנפרד
 - אין תאימות פשוטה עם קוד בשפות אחרות

JAVA BYTECODE

- שפת התכנות Java היא, לפי חלק מהמדדים, שפת התכנות הפופולרית ביותר בעולם!
- מתכנתים כותבים קוד ג'אווה בקבצי java. ואז הקומפיילר של Java מקמפל אותו לקבצי class. (כמו קבצי object) ואז אורז אותם בקובץ jar. אחד (כמו קובץ הרצה).

```
public static void main(String[] args) {  
    System.out.println("Hello world!");  
}
```



```
public static void main(java.lang.String[]);  
0: getstatic #7 // Field java/lang/System.out:Ljava/io/PrintStream;  
3: ldc #13 // String Hello world!  
5: invokevirtual #15 // Method java/io/PrintStream.println:(Ljava/lang/String;)V  
8: return
```

JVM

- קבצי JAR מורצים ע"י ה-Java Virtual Machine
 - ה-JVM נבנה בנפרד לכל ארכיטקטורה, אבל קבצי JAR ניתנים להעברה חופשית ללא צורך לקמפל מחדש
- ל-JVM אין רגיסטרים אלא **מחסנית**
 - למה?
- ה-JVM מנהלת את הזיכרון באופן אוטומטי באמצעות מנגנון Garbage Collection
 - לא צריך לעשות "delete" בג'אווה

שפות JVM אחרות

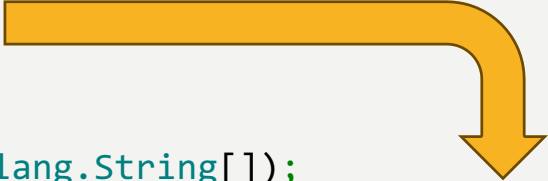
- קיימות שפות אחרות שמתקמפלות ל-Java Bytecode ועל כן ניתן להריץ אותן על ה-JVM יחד עם קוד Java.

- השפה הפופולרית ואהובה ביותר מביניהן היא Kotlin

- פופולרית במיוחד לתכנות Android

- קוד Java יכול לגשת לקוד Kotlin ולהפך

```
fun main(arr: Array<String>?) =  
    print("Hello world!")
```



```
public static final void main(java.lang.String[]);  
0: ldc #9 // String Hello world!  
2: getstatic #15 // Field java/lang/System.out:Ljava/io/PrintStream;  
5: swap  
6: invokevirtual #21 // Method java/io/PrintStream.print:(Ljava/lang/Object;)V  
9: return
```

פורמט קבצים

- קובץ JAR אינו אלא קובץ zip עם סיומת אחרת, שמכיל קבצי class וקבצי metadata.
- כל קובץ class מייצג מחלקה יחידה בשפת Java
 - כל קובץ מקור ב-Java מכיל בדיוק מחלקה אחת ברמה הראשית, אבל ייתכנו מחלקות מקננות שבגללן יוצר יותר מקובץ class אחד עבור קובץ מקור מסוים
- קובץ class מתחיל במספר קסם: CA FE BA BE

פורמט קבצים

- לאחר מכן, יגיע מאגר הקבועים: טבלה המתאימה אינדקס לקבוע כלשהו
- הקבועים יכולים להיות ערכים בשפה או שמות של שדות, מתודות ומחלקות.
- במאגר הקבועים מתודה מיוצגת ע"י החתימה שלה.
- מבנה חתימה של מתודה ב-Java:
`path/to/class.method_name:(arg_types)ret_type`
- לדוגמה:
`java/lang/Integer.valueOf:(I)Ljava/lang/Integer;`
טיפוסים פרימיטיביים: I – int, B – byte, Z – bool, V – void, ...
טיפוסי אובייקט: `L <path/to/class>` ; (עם נקודה פסיק בסוף)
טיפוסי מערך: `[<some_type>` (רק סוגר שמאלי)

פורמט קבצים

- לאחר מאגר הקבועים, יופיעו הרכיבים הבאים:
- דגלי הגישה של המחלקה
- גם public, private וכו' אבל גם abstract, static ועוד רבים
- משחק מהנה להעברת הזמן העודף במבחן:
 - https://www.sporcle.com/games/robv/java_keywords
- שם המחלקה הנוכחית ומחלקת האב שלה
- פרטי הממשקים שהמחלקה מממשת (אם יש) ופרטי השדות שיש לה (אם יש)
- פרטי המתודות של המחלקה (אם יש)
- מאפיינים נוספים כמו שם קובץ המקור של המחלקה

מבוא ל-JAVA BYTECODE

- פקודות ב-Java Bytecode מבוססות על שימוש במחסנית אופרנדים ובמאגר הקבועים, כמו גם מערך משתנים מקומיים.
- כמו ב-AT&T syntax assembly, לפקודות רבות יש תחילית או סיומת המתאימה לטיפוס של האופרנדים.
- לא כמו באסמבלי, ה-JVM באמת בודק את הנתון הזה.
- החשובים: i עבור int, a עבור אובייקטים (רפרנסים)
- דוגמה לפקודה בסיסית: `iconst_0`
 - דוחפת את הערך 0 על המחסנית
 - יש כאלה גם ל-1 עד 5, ולמינוס 1

פקודות בסיסיות

- הפקודות $iconst_n$ עבור $n \in \{-1, \dots, 5\}$
- שימו לב! הערך מקודד בתור חלק מהפקודה, הוא לא פרמטר
- הפקודה $iadd$ (ובדומה $imul$, $idiv$, $isub$ ועוד)
 - מוציאה את שני הערכים העליונים מהמחסנית ודוחפת את הסכום שלהם
 - יש גם $dadd$, $fadd$, $ladd$, ...
- הפקודה dup
 - משכפלת את הערך שבראש המחסנית
- הפקודה $ldc <idx>$
 - טוענת את הקבוע שמספרו idx מתוך מאגר הקבועים

אחסון וטעינה

- הפקודה `iload <idx>` והפקודות `iload_0, ..., iload_3`
 - דוחפת את הערך מהמשתנה המקומי שמספרו `idx` על המחסנית
- הפקודה `istore <idx>` והפקודות `istore_0, ..., istore_3`
- פקודות בסיסיות נוספות שכדאי לשים לב אליהן:
- הפקודה `getstatic <idx>`
 - דוחפת את הערך של השדה הסטטי שהאינדקס של החתימה שלו הוא `idx`
- הפקודה `getfield <idx>`
 - דוחפת את הערך של השדה של האובייקט בראש המחסנית שהאינדקס של החתימה שלו הוא `idx`

קריאה לפונקציות

- ב-JBC יש 5 פקודות שונות לקריאה לפונקציות:
invokedynamic, invokeinterface, invokespecial, invokestatic, invokevirtual
- אנחנו נתמקד בשתיים: invokestatic, invokevirtual
- הפקודה <idx> invokestatic
 - קוראת למתודה הסטטית שהאינדקס של החתימה שלה הוא idx עם הפרמטרים שבראש המחסנית ודוחפת את התוצאה
- הפקודה <idx> invokevirtual
 - קוראת למתודה הווירטואלית שהאינדקס של החתימה שלה הוא idx, על האובייקט שבראש המחסנית ועם הפרמטרים שאחריו, ודוחפת את התוצאה

דוגמה

- מה מודפס אחרי הקוד הבא?

```
0: ldc #7
2: istore_1
3: ldc #8
5: istore_2
6: getstatic #5
9: iload_1
10: iload_2
11: iadd
12: iload_2
13: iadd
14: invokevirtual #4
17: return
```

Index	Value
0	-
1	java/lang/Object."<init>":()V
2	32
3	java/io/List.add:(Ljava/lang/Object;)Z
4	java/io/PrintStream.println:(I)V
5	java/lang/System.out:Ljava/io/PrintStream;
6	54
7	9
8	11

תנאים וקפיצות

- ניתן להציב תוויות – labels כמו באסמבלי
- הפקודות <offset> ifeq, ifge, ifgt ודומותיהן – אם הערך בראש המחסנית הוא אפס/גדול-שווה אפס/גדול מאפס, קפוץ לפקודה בהיסט offset
- הפקודות <offset> if_icmpeq, if_icmpge, if_icmpgt ודומותיהן – אם הערך בראש המחסנית גדול/גדול-שווה/שווה לערך שמתחתיו, קפוץ לפקודה בהיסט offset
- הפקודה goto <offset> – כמו jmp

שונות

- הפקודה `pop` (שה-`opcode` שלה הוא 00)
- הפקודות `pop`, `pop2`
- הפקודה `aconst_null`
- הפקודה `return`
 - לא מחזירה ערך – `void`
- הפקודות `ireturn` וכן הלאה
 - מחזירה את הערך שבראש המחסנית
- ספציפית `aload_0`
 - שמור להחזקת `this` במתודה שאינה סטטית

סיכום

- ניתוח סטטי של Java Bytecode זה הרבה יותר קל מניתוח סטטי של סתם אסמבלי
- עברנו על רוב הפקודות העיקריות
- נותרו הרבה פקודות שקשורות לעבודה עם עצמים כמו `instanceof`, `new`, `invokevirtual` ועוד
- למידע נוסף, ניתן לקרוא את התיעוד הרשמי:
<https://docs.oracle.com/javase/specs/jvms/se12/html/jvms-6.html>

PYTHON BYTECODE

- גם שפת Python עובדת על Bytecode, אבל באופן קצת שונה
- שפת Python היא שפה דינמית שעוברת interpretation ולא קומפילציה: מריצים את התוכנה python ישירות על קובץ המקור
- מבחינתנו, זה אומר שני דברים:
 - ה- interpreter הוא גם VM וגם קומפיילר, ולכן ניתן לגשת ל-bytecode בזמן ריצה
 - ניתן לקמפל חלקים נוספים ל-bytecode בזמן ריצה
- ה- interpreter שומר קבצי פייתון מקומפלים בתיקייה `__pycache__`

PYTHON BYTECODE

- גם בפייתון, ה-Bytecode הוא מבוסס מחסנית ויש מאגר קבועים
- בפייתון, בניגוד ל-Java, אין כמעט אופטימיזציה של זמן קומפילציה

– *במימוש הנפוץ, CPython

- ניתן לגשת ל-Bytecode בזמן ריצה באמצעות המודול `dis`:

```
def f(num):  
    return num + num - num
```

```
import dis  
dis.dis(f)
```



```
1 0 RESUME 0  
  
2 2 LOAD_FAST 0 (num)  
4 LOAD_FAST 0 (num)  
6 BINARY_OP 0 (+)  
10 LOAD_FAST 0 (num)  
12 BINARY_OP 10 (-)  
16 RETURN_VALUE
```

קומפילציה בזמן ריצה

- בפייתון, ניתן לקמפל ל-Bytecode ולהריץ ביטויים שרירותיים שנקבעים בזמן ריצה:

```
def square(num):  
    return eval(f"{num} * {num}")  
  
print(square(5))
```

- קיימות מספר פונקציות המטפלות בהרצה של קוד שרירותי
 - eval – מחשבת ביטויים
 - exec – מריצה פקודות (כלשהן)
 - compile – גם וגם, ועוד הרבה אופציות
 - כנראה מטרות טובות ל-injection