

# תוכנה 1 בשפת Java

## שיעור מספר 13: "אל תסתכל בקנקן"

### דן הלפרין

בית הספר למדעי המחשב  
אוניברסיטת תל אביב

# תוכן

- JVM (המכונה הוירטואלית)
- מבני הנתונים ותהליך הביצוע
- ניהול הזיכרון בזמן ריצה - איסוף זבל

# מבט מלמעלה

- הקומפילר מתרגם את קוד המקור ל-byte code, קובץ class, שמכיל ייצוג בינארי דחוס של מחלקה
- הקובץ הבינארי משקף כמעט בדיוק את מבנה הקוד
- כדי להריץ תוכנית ג'אווה, מערכת ההפעלה מפעילה תוכנית "ילידה" (כתובה בדרך כלל בשפת C ומשתמשת ישירות במנשקים של מערכת ההפעלה) בשם java
- התוכנית הזו היא Java Virtual Machine (JVM) והיא טוענת קבצים בינאריים, בדידים או במארז jar, ומבצעת את הפקודות שהם מכילים, כולל הקצאת ושחרור זיכרון וקריאה לשירותים
- יש חופש גדול במימוש JVM; המפרט קובע מה JVM צריכה לעשות, אך לא איך לעשות זאת

<http://docs.oracle.com/javase/specs/jvms/se7/html/>

# קומפילציה

- מה הקומפילר צריך לדעת כאשר הוא מקמפל מחלקה?
  - צריך לדעת מהם השדות והשירותים שמוגדרים בכל טיפוס שהמחלקה משתמשת בו בשדות, משתנים, וארגומנטים
- מהיכן הקומפילר שואב את המידע הזה?
  - בדרך כלל, מהקבצים הבינאריים של הטיפוסים הללו
  - אבל אם הם עדיין לא עברו קומפילציה, הקומפילר מחפש את קוד המקור ומקמפל אותם ביחד עם המחלקה הנוכחית
- אין הפרדה בין קובץ שמכיל רק הצהרות על השדות והשירותים ובין קובץ שמכיל את ההגדרות שלהם, כמו שיש בשפות אחרות (למשל ++C), ולכן אין סיכוי שההצהרות וההגדרות לא יתאימו אלה לאלה

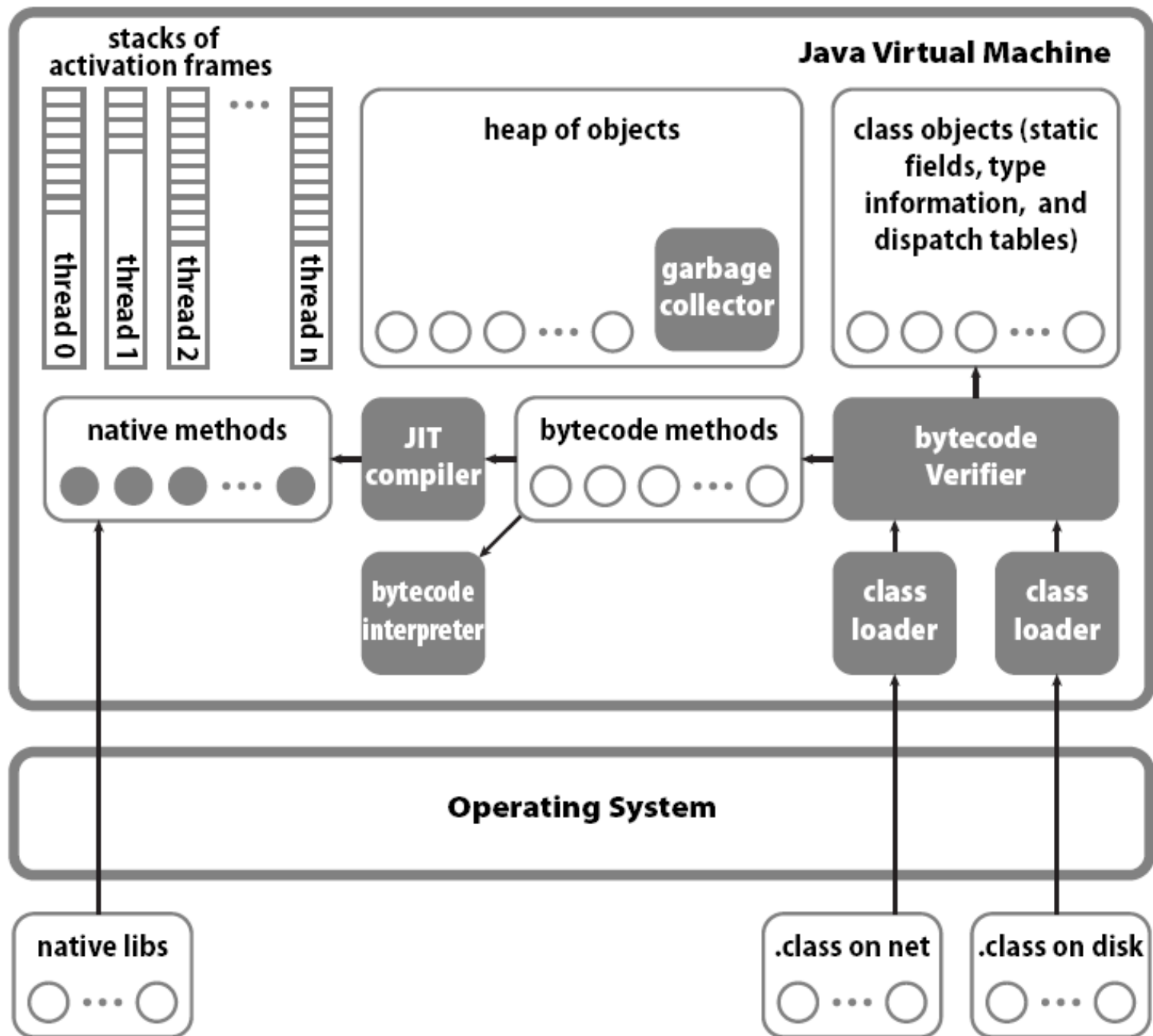
# JVM מבני הנתונים של ה

המכונה הוירטואלית משתמשת במספר מבני נתונים לייצג את כל המידע הדרוש לביצוע התכנית:

- לכל מחלקה (שנטענה), הקוד של המחלקה (bytecode) : שירותים, בנאים, אתחול סטטי
- לחלק מהשירותים יכול להישמר גם קוד בשפת מכונה
- לכל מחלקה, אוסף השדות הסטטיים שלה
- אוסף שדות המופע של העצמים שנוצרו
- לכל שרות שנקרא וטרם הסתיים, אוסף הפרמטרים האקטואליים והמשתנים המקומיים (נקרא רשומת הפעלה (activation record): מחסנית זמן ריצה (runtime stack)

# החלקים הביצועיים של ה JVM

- החלק הביצועי של המכונה הוירטואלית כולל מספר חלקים:
  - טוען המחלקות (class loader) קורא קבצי class. מזיכרון משני, או מהרשת (או במקרים מסוימים יוצר אותם בדרך אחרת)
  - ה verifier בודק שה bytecode שנטען תקין
  - המשערך (פרשן, interpreter) מבצע את ה bytecode ; שם כללי יותר: execution engine
  - אוסף הזבל (garbage collector) מופעל כדי למחזר קטעי זיכרון שאינם בשימוש
  - JIT compiler מתרגם שירותים מסוימים מ bytecode לשפת המכונה של המחשב לפי הצורך



# מחסנית זמן ריצה

- מבנה זה דרוש בכל שפה שיש בה פרוצדורות עם רקורסיה
- לכל שרות שנקרא וטרם הסתיים תישמר רשומת הפעלה (activation record) שבה: פרמטרים אקטואליים, משתנים מקומיים, הערך שהשרות מחזיר, כתובת החזרה (המקום בקוד בו יימשך הביצוע לאחר שהשרות יסתיים), ועוד ...
- קריאות לשרות נעשות בסדר של LIFO (הקריאה האחרונה שבוצעה חוזרת ראשונה), ולכן רשומות ההפעלה ייוצגו במחסנית (נקראת מחסנית זמן ריצה)
- בקריאה לשרות תיבנה רשומת הפעלה בראש המחסנית
- בחזרה משרות תבוטל רשומת ההפעלה בראש המחסנית
- בתכנית עם תהליכים מקבילים, לכל תהליך מחסנית משלו

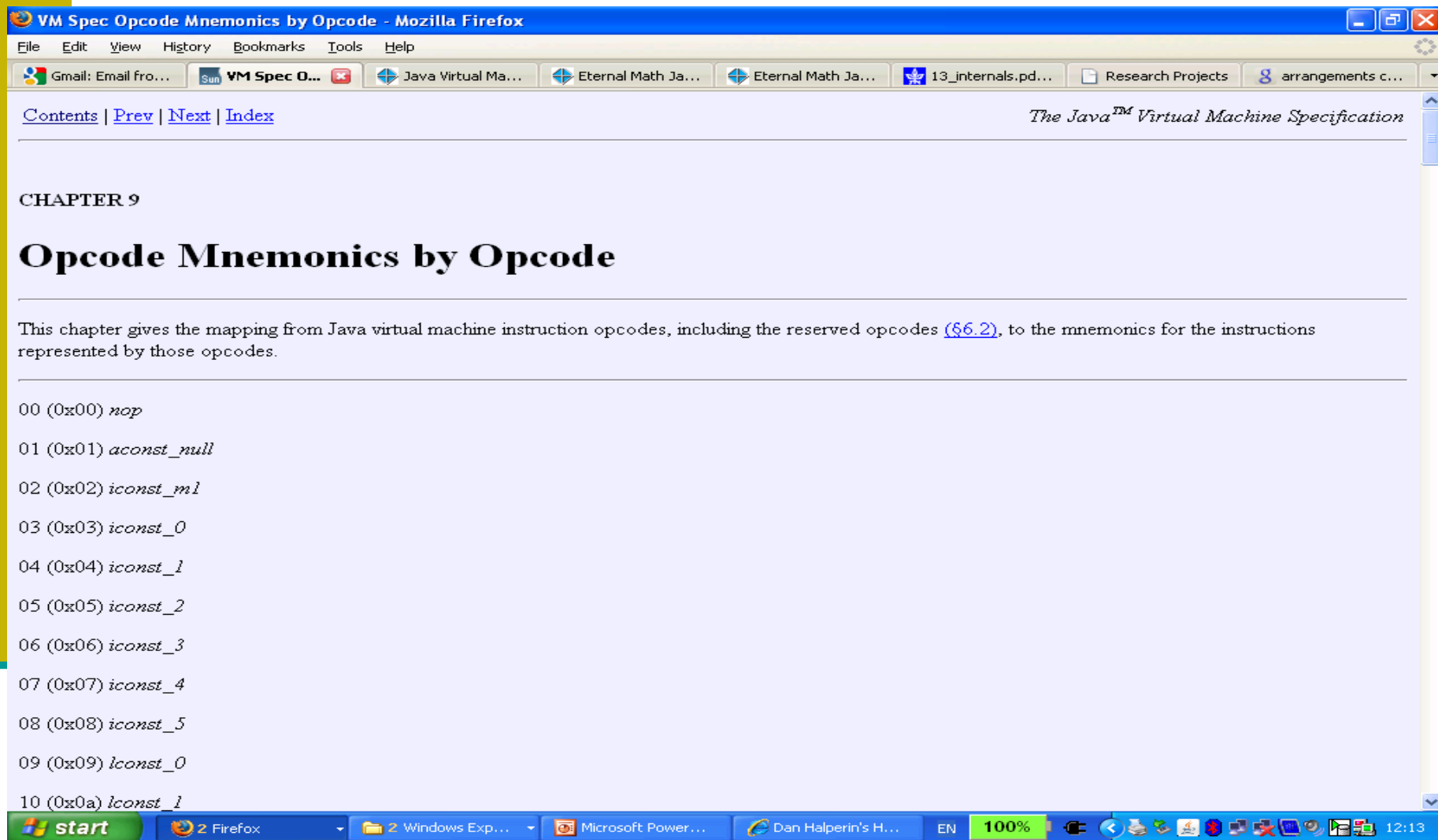


# הרצה וקומפילציה של bytecode

- בקובץ class. השירותים מיוצגים ב-bytecode, שפת מכונה של מחשב וירטואלי (לא כל הפקודות פשוטות, למשל invokeinterface)
- לאחר טעינה של מחלקה לזיכרון, ה-JVM יכול להריץ שירותים על ידי סימולציה של המחשב הוירטואלי; הרכיב של ה-JVM שמבצע את הסימולציה נקרא bytecode interpreter או execution engine
- בסימולציה כזו יש מחיר גבוה גם לפעולות מאוד פשוטות, למשל חיבור של שני שלמים
- כדי להימנע מתקורה קבועה על כל פעולה, תקורה שנובעת מהסימולציה, ה-JVM יכול לקמפל את הקוד של שירות לשפת מכונה של המעבד שהתוכנית רצה עליו

# מנוע הביצוע ורשימת הפקודות

- כל מתודה מתורגמת לרצף של bytecode
- כל פקודה מורכבת מבית אחד של שם (opcode) מלווה באפס או יותר אופרנדים (בית אחד כדי שקובץ ה- class יהיה קופקטי)
- המנוע מבצע פקודה אחת בכל פעם
- לפי הפקודה המנוע עשוי לחפש נתונים במקומות שונים: משתנים מקומיים ברשומת הריצה של המתודה, על מחסנית האופרנדים (אף היא ברשומת הריצה), ועוד



# דוגמא, קוד המקור

```
class Act {  
  
    public static void doMathForever() {  
        int i = 0;  
        for (;;) {  
            i += 1;  
            i *= 2;  
        }  
    }  
}
```

# דוגמא, bytecode

Bytecode stream: 03 3b 84 00 01 1a 05 68 3b a7 ff f9

```
0 iconst_0    // 03, Push int constant 0.
1 istore_0    // 3b, Pop into local variable 0: int i = 0;
2 iinc 0, 1    // 84 00 01, Increment local variable 0 by 1:
                i += 1;
5 iload_0     // 1a, Push int in local variable 0
6 iconst_2    // 05, Push int constant 2
7 imul        // 68, Pop two ints, multiply, push result
8 istore_0    // 3b, Pop into local variable 0: i *= 2;
9 goto 2      // a7 ff f9, Jump unconditionally to 2:
                for (;;) {}
```

האופרנד של goto הוא המספר שיש להוסיף ל - program counter

(-7 במקרה הזה)

# דוגמא, אמולטור

■ מתוך

Inside the Java Virtual Machine/Bill Veners

<http://www.artima.com/insidejvm/applets/EternalMath.html>

■ משתנה מקומי יחיד  $i$  (מקום 0 במערך המשתנים המקומיים)

■ מחסנית האופרנדים גדלה כלפי מטה  $optop$  – מצביע לאחד אחרי ראשה

■  $>pc$  מראה היכן נמצא ה –  $program\ counter$  בכל רגע נתון

# Just-in-Time Compilation

- קומפילציה כזו מ-bytecode לשפת מכונה (native code) נקראת just-in-time compilation, כי היא מתבצעת ממש לפני השימוש בקוד, ולא כחלק מהכנת התוכנה להפצה
- בדרך כלל, JIT מופעל על שירות לאחר שהתברר ל-JVM שהשירות מופעל הרבה; זה מונע קומפילציה יקרה של שירותים שאינם מופעלים או כמעט ואינם מופעלים
- יתכנו גם אסטרטגיות אחרות: לעולם לא לבצע JIT, לבצע באופן מיידי בזמן הטעינה של מחלקה, לבצע באופן מיידי אבל ללא אופטימיזציות ולשפר אחר כך את הקומפילציה של שירותים שנקראים הרבה, לבצע באופן גורף אבל רק כאשר המעבד נח והמחשב מחובר לחשמל, ועוד
- עם JIT, הביצועים של תוכנית משתפרים לאורך הריצה

# אז למה bytecode?

- אם ממילא תוכנית הג'אווה מתקמפלת בסופו של דבר לשפת המכונה של המעבד, למה לא לקמפל אותה לשפת מכונה בזמן אריזת התוכנה להפצה, ולא בזמן ריצה?
- **קומפילציה בזמן אריזה יעילה יותר**, מכיוון שהיא מתבצעת פעם אחת עבור הרצות רבות; בזמן אריזה כדאי להפעיל אופטימיזציות יקרות, בזמן ריצה לא
- הפצת תוכנה כ-bytecode משיגה שתי מטרות;
  - האחת, את היכולת להשתמש בתוכנה ארוזה אחת על **מעבדים שונים (ומערכות הפעלה שונות)**
  - המטרה השנייה היא **בטיחות**; ה-bytecode verifier בודק את התוכנית לפני הרצתה לוודא קיום דרישות השפה; משפר את בטיחות מערכות המחשב ומונע סוגים מסוימים של תקיפות



# תוכנה לשימוש במספר פלטפורמות שונות

- היכולת לארוז תוכנה פעם אחת ולהשתמש בה במחשבים עם מגוון של מעבדים ומערכות הפעלה היא יכולת חשובה, מכיוון שאריזת תוכנה להפצה היא פעולה מורכבת ויקרה
- ללא bytecode, צריך לקמפל ולארוז את התוכנה עבור כל פלטפורמה בנפרד. למשל: חלונות על אינטל, חלונות על מעבד אחר (למשל מחשב כף יד או טלפון), מקינטוש, ....
- סיסמת השיווק של סאן לגבי ג'אווה הייתה "write once run anywhere": לכתוב ולארוז את התוכנה פעם אחת ולהריץ אותה על פלטפורמות רבות

# תוכנה לשימוש במספר פלטפורמות שונות

■ שתי רמות יכולת של תוכנה לרוץ על מספר פלטפורמות:

■ ברמה הראשונה: קוד המקור מתאים למספר פלטפורמות, אבל צריך לקמפל אותן לכל פלטפורמה בנפרד.

■ ברמה השנייה: התאמה לא רק ברמת קוד המקור, אלא גם ברמת התוכנה הארוזה להפצה. תוכניות ג'אווה שייכות לרמה הזו, הודות לשימוש ב-bytecode.

# שפת מכונה וירטואלית כמנגנון להפצת תוכנה

■ הרעיון אינו חדש

■ גרסאות מסוימות של פסקל השתמשו בו לפני שנים רבות

■ חסרונות שמנעו שימוש נרחב בעבר:

■ **ביצועים:** פרשן (bytecode interpreter) הוא איטי לעומת ביצוע

קוד בשפת מכונה. לכן, שימוש ב-bytecode דורש מעבד מהיר מאוד, או קומפילציה בזמן ריצה (JIT) (או שניהם)

■ **מגוון קטן יחסית של מעבדים.** הגידול במגוון המעבדים

(ומערכות ההפעלה) הביא אפילו את מיקרוסופט, לחפש דרכי הפצה כאלה, והוביל לארכיטקטורת ה-.NET, משפחה של שפות תכנות וסביבת זמן ריצה דומות לג'אווה

# הטמנת תרגומים לשפת מכונה

- באופן עקרוני, אין סיבה לבצע JIT בכל ריצה של התוכנית
- ה-JVM (או מערכת ההפעלה במקרה של .NET). יכולה להטמין (cache) תרגומים של bytecode לשפת מכונה ולהשתמש בהם שוב ושוב (כיום זה לא נעשה)
- אם התרגומים לשפת מכונה נשמרים בקבצים שרק ל-JVM יש אליהם גישה (ואולי חתומים דיגיטלית), בדיקת התקינות שבוצעה נשארת תקפה ואפשר להשתמש בהם ללא חשש
- אפשר גם לבצע קומפילציה עם אופטימיזציות יקרות כשהמחשב אינו פעיל או בזמן התקנת התוכנה
- כבר היום יש מערכות הפעלה שמבצעות אופטימיזציות מסוימות בזמן התקנת תוכנה (תוכנה ילידה)

# ייצוג מחלקה ועצם, קוד מקור לדוגמא

```
class MyClass {
    static int static1, static2;
    int field1, field2;

    void method1(int x) {
        field1 = ...; static2 = ... ; method2 (...);
    } // end method1

    void method2(int y) { ... }
} // end MyClass
```

איך מיוצגים המחלקה והעצם?  
מה יתבצע בקריאה?

```
MyClass o = new MyClass ();
o.method1 (5);
```

Client.java

# מבנה של עצם ושל ייצוג של מחלקה

- בזמן ריצה, עצם הוא מבנה נתונים שמכיל הצבעה למבנה הנתונים של המחלקה שהוא שייך אליה ואת הערכים של שדות המופע
- התייחסות היא הצבעה למקום בזיכרון שבו מתחיל המבנה של העצם המיוחס (לפעמים יותר מסובך; נסביר בהמשך)
- הייצוג של מחלקה כולל מידע על הטיפוס (בעיקר איזה מנשקים היא מממשת), טבלת הצבעות לשירותים (dispatch table), ואת הערכים של שדות המחלקה
- הייצוג של המחלקה נבנה בזמן הטעינה שלה, ואינו משתנה
- עצמים נוצרים באופן דינאמי, ולכן את אזור הזיכרון שלהם (נקרא heap) צריך לנהל

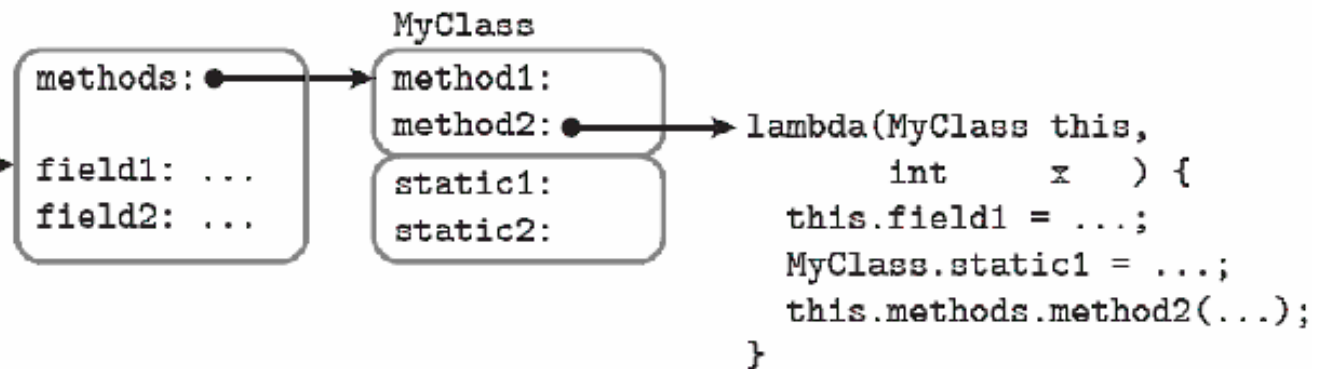
# מבנה של עצם

```
class MyClass {
    static int static1,static2;
        int field1, field2 ;
    void method1(int x) { field1=...; static2=...; method2(...); }
    void method2(int y) {...}
    ...
}
```

```
MyClass o =
    new MyClass();
```

*o: ●*  
*what happens*  
*when we invoke*  
*a method?*

```
o.method1(5);
```



# ייצוג של שירות

- שירות מופע (לא static) הוא שגרה שמקבלת את הארגומנטים הפורמאליים של השירות, וגם ארגומנט נסתר שבו מועברת הכתובת בזיכרון של העצם (this)
- טבלת ההצבעות לשירותים של מחלקה יכולה להצביע לשירותים שהגדירה וגם לשירותים שירשה ולא דרסה; לכן, שירות מופע m צריך להיות מוכן לקבל this שמצביע לעצם שאינו מהמחלקה שהגדירה את m אלא ממחלקה מרחיבה
- שירות מחלקה (static) הוא שגרה שמקבלת את הארגומנטים הפורמאליים של השירות, ולא מקבלת מצביע ל- this
- פרט לכך שירותי מופע ומחלקה זהים, ושני הסוגים מופיעים באותה טבלת ההצבעות לשירותים של מחלקה



# שימוש בשדות

- שימוש בשדה מופע מתבצע על ידי הוספת ההיסט (המרחק מתחילת העצם) של השדה לכתובת `this`; הכתובת שמתקבלת היא הכתובת של השדה
- בדוגמה השתמשנו בסימון `this.field1` (בפועל זה יופיע ב `.(bytecode`
- שימוש בשדה מחלקה יותר פשוט; כאשר המחלקה נטענת לזיכרון, נקבעות הכתובות של שדות המחלקה, שלא זזים במהלך התוכנית; אפשר להחליף כל התייחסות לשדה מופע בהתייחסות לכתובת של השדה בזיכרון
- בדוגמה הסימון היה `MyClass.static1`, אבל בעצם זו כתובת אבסולוטית

# הפעלה של שירות

למשל: (5) `method1`.

- ההתייחסות `o` מצביעה למקום של עצם בזיכרון
- מבנה של העצם כולל הצבעה לטבלת השירותים שלו (השדה הנסתר `methods`)
- השירות `method1` הוא השירות הראשון של המחלקה, ולכן ההצבעה לשגרה תהיה במקום הראשון בטבלת השירותים
- את השגרה הזו מפעילים, כאשר בארגומנט הראשון (הנסתר) שלה תועבר הכתובת של `o` ובארגומנט השני הערך `5`
- בסימונים שלנו, זה `( 5, o )` `o.methods[0]`
- להפעלה כזו של שירות קוראים הפעלה וירטואלית, מכיוון שהשירות שיופעל תלוי בטיפוס הדינאמי, לא הסטאטי

# מבנה עצם ממחלקה מרחיבה

- כאשר מחלקה Sub מרחיבה את המחלקה Base, המבנה של עצמים מהמחלקה המרחיבה נגזר ממבנה עצמים ממחלקת הבסיס
- גם המבנה של ייצוג המחלקה עצמה נגזר מייצוג הבסיס
- שדות מופע בעצמים של Sub יופיעו לאחר שדות המופע שהוגדרו כבר ב-Base
- שירותים שנוספו ב-Sub יופיעו לאחר השירותים שנורשו או נדרסו מ-Base
- זה מבטיח שהתייחסות לעצם דרך מצביע מטיפוס Base תפעל נכון: השדות והשירותים נמצאים באותו מקום יחסי ב-Sub וב-Base

# הקושי בהפעלת שירותים על מנשקים

■ כאשר מחלקה Sub מרחיבה את Base, שמרחיבה, למשל, את Object, אז השירותים והשדות של Object הם תת קבוצה של אלה של Base שהם תת קבוצה של אלה של Sub

■ זה מאפשר לסדר את טבלת השירותים ואת השדות כך שתתי הקבוצות יופיעו תמיד כתחיליות; אפשר להשתמש בעצם דרך כל אחד משלושת הטיפוסים

- המצב יותר מסובך אם Sub מממשת שני מנשקים, I1 ו- I2
- אין קשר בין השירותים ששני המנשקים מצהירים עליהם
  - אי אפשר לסדר את השירותים כך שאפשר יהיה למצוא את השירותים של I1 ואת השירותים של I2 באותה טבלת שירותים (dispatch table) בלי להתייחס לטיפוס הדינאמי
  - בהפעלה o.methods[0] לא התייחסנו לטיפוס הדינאמי

# הפעלת שירות על מנשק

- אז איך מפעילים את השירות  $m$  על עצם  $s$  שהטיפוס הסטאטי שלו הוא המנשק  $\tau_1$ ?
- בעיה דומה יש בשפות עם ירושה מרובה כמו C++ ו-Eiffel
- זו בעיה קשה שנמצאת עדיין בחזית המחקר (למימוש יעיל)
- מימושים **גרועים** של ג'אווה משתמשים באלגוריתם פשוט **שמחפש** את השירות הנחוץ; הפעלה כזו איטית בסדר גודל אחד או שניים מהפעלה של שרות שאינו וירטואלי
- במימושים מתוחכמים אין כמעט הבדל ביצועים בין הפעלה וירטואלית ובין הפעלה של מנשק מימושים אלה מסובכים למדי או בזבזניים בזיכרון - צריך מערך של טבלאות שירותים (dispatch vector, virtual tables)
- אנו נציג פתרון יעיל ופשוט, אבל אופייני דווקא ב C++

# הפעלה יעילה של שירות על מנשק

- עבור כל מחלקה, נשמור לא טבלת שירותים (dispatch table) אחת, אלא מערך שלם של טבלאות כאלה, אחת עבור כל טיפוס שהמחלקה מתאימה לו
- ליתר דיוק, צריך במערך טבלה אחת עבור הפעלות וירטואליות ועוד טבלה אחת עבור כל מנשק שהמחלקה מממשת ושמרחיב יותר ממנשק אחד
- התייחסות לעצם תכלול גם מצביע לייצוג של העצם (לשדות שלו) וגם את הכתובת של האיבר במערך טבלאות השירותים שמתאים לטיפוס הסטאטי של התייחסות
- המרה למעלה או למטה בייצוג כזה דורשת הזזה של הכתובת של האיבר במערך טבלאות השירותים
- הקומפיילר יודע בדיוק בכמה צריך להזיז בשביל המרה למטה (Down Casting)

# איסוף זבל

# איסוף זבל (garbage collection)

■ מנגנון אוטומטי לזיהוי עצמים ומערכים שהתוכנית לא יכולה להגיע אליהם יותר ושחרור הזיכרון שהם תופסים

```
Double w = new Double(2.0);  
Double x = new Double(3.0);  
Double y = new Double(4.0);  
Double z = new Double(5.0);  
w.compareTo(x);  
y = null; // Can we now release w, x, y, z?
```

■ קשה לדעת; compareTo הייתה עשויה לשמור במבנה נתונים כלשהו התייחסויות ל-w ו-x; אפשר לשחרר את (העצם שאליו התייחס) y, אבל ב-z השירות עוד עשוי להשתמש



# מהו זבל?

## לאיזה עצמים אי אפשר להתייחס?

- יותר קל להגדיר את העצמים שאליהם כן אפשר להתייחס
- ראשית, לעצמים שיש אליהם התייחסות ממשתנים אוטומטיים של גושי פסוקים שלא סיימו לפעול, כלומר שגרות וגושי פסוקים פנימיים שפעולתם הופסקה בגלל הפעלה של שירות או פרוצדורה או בגלל גוש פנימי יותר
- שנית, לעצמים שיש אליהם התייחסות משם גלובאלי; בג'אווה, שמות גלובליים מתאימים בדיוק לשדות מחלקה
- ושלישית, לכל עצם שיש אליו התייחסות מעצם שניתן להתייחס אליו; זו הגדרה רקורסיבית, אבל היא היחידה הנכונה
- כל השאר זבל

# שורשים

- תהליך איסוף זבל מתחיל בשורשים, התייחסויות שברור שלתוכנית יש גישה אליהם
- שורשים בג'אווה כוללים שדות מחלקה ואת כל המשתנים שנמצאים בחלק החי של כל המחסניות (אחת אם יש רק חוט/תהליכון אחד בתוכנית, יותר אם היא מרובת חוטים)
- אם נסמן את השורשים בצבע מיוחד, ואחר כך נצבע באותו צבע כל עצם לא צבוע שיש אליו התייחסות מעצם צבוע, ונמשיך עד שלא יהיה מה לצבוע, העצמים הצבועים אינם זבל וכל השאר זבל
- זו תמיד ההגדרה של זבל; יש אלגוריתמים לאיסוף זבל שפועלים ממש בצורה הזו (mark and sweep), ויש שפועלים בצורה אחרת

# איסוף זבל בשיטת mark & sweep

- אוסף הזבל עוצר את התוכנית
- עוברים על כל העצמים והמערכים שנגישים (reachable) מהשורשים, ומסמנים אותם (צובעים אותם)
- עוברים על כל העצמים, ומשחררים את הלא מסומנים; הזיכרון שתפסו יוקצה בהמשך לעצמים אחרים
- אבל יש עוד גישות
- אוסף הזבל מופעל בדרך כלל באופן אוטומטי ע"י ה JVM כאשר הזיכרון שעומד לרשותו (כמעט) נגמר
- יש אפשרות למתכנת לקרוא במפורש לאוסף הזבל

# איסוף זבל בשיטת ההעתקה (Copying)

- הזיכרון מחולק לשני חלקים באותו גודל, א' וב'
- בזמן שהתוכנית פועלת, כל העצמים והמערכים נמצאים בצד אחד; הצד השני ריק
- אוסף הזבל עוצר את התוכנית; נניח שכל העצמים בצד א'
- עוברים על כל העצמים והמערכים שנגישים מהשורשים, ומעתיקים כל אחד מהם לצד ב'
- המיקום של עצמים בזיכרון משתנה; צריך לעדכן התייחסויות אליהם; אפשר לעשות זאת על ידי סימון עצמים בא' שמועתקים כלא תקפים וסימון המקום החדש שלהם
- כאשר לא נשארים עצמים נגישים בצד א', מוחקים את כולו
- באיסוף הבא התפקידים של א' וב' מתחלפים

# עדכון התייחסויות באוסף מעתיק

- נניח שמעתיקים עצם מכתובת  $p_1$  בזיכרון בצד א' לכתובת  $p_2$  בצד ב'
- משנים את תוכן שטח הזיכרון בכתובת  $p_1$
- מדליקים סיבית שמסמנת שהעצם כבר הועתק לצד ב'
- כותבים בשטח הזיכרון את הכתובת החדשה  $p_2$
- לאחר סיום ההעתקה, עוברים על כל העצמים בצד ב'
- עבור כל עצם, מוצאים את כל ההתייחסויות שהוא מכיל (שדות מופע שהם התייחסויות ומערכים של התייחסויות)
- עבור כל התייחסות כזו לעצם בכתובת  $q_1$  שולפים מהעצם ב- $q_1$  את הכתובת החדשה של העצם  $q_2$  ומעדכנים מעדכנים באופן דומה את השורשים

# השוואת שתי הגישות לאיסוף זבל

- יש הבדלי ביצועים משמעותיים בין mark & sweep ובין copying collectors
- מה ההבדלים ואיזו גישה עדיפה?
- זמן הריצה של mark & sweep תלוי במספר העצמים בזיכרון בזמן האיסוף
  - טיפול בכל עצם דורש מספר קבוע של פעולות; סימון של עצמים נגישים ושחרור עצמים לא נגישים
- זמן הריצה של copying collectors תלוי בכמות הזיכרון הכוללת שתופסים העצמים הנגישים, כי צריך להעתיק אותם
  - אבל אין שום טיפול בעצמים לא נגישים; הם נמחקים בבת אחת

# ספירת התייחסויות

- גישה נוספת לאיסוף זבל, שלא עובדת בג'אווה, מבוססת על ספירת התייחסויות לכל עצם/מערך
  - בכל פעם שיוצרים התייחסות לעצם מקדמים את מונה ההתייחסויות של העצם, ובכל פעם שהורסים התייחסות כזאת מפחיתים 1 מהמונה (מאט שימוש בהתייחסויות!)
  - כאשר המונה מגיע ל- 0, משחררים את הזיכרון של העצם
- בגלל שבג'אווה מותרים מעגלים בגרף התייחסויות, המונה לא תמיד מגיע ל- 0 גם אם העצם כבר לא נגיש
  - עצם א' נגיש משורש ומצביע על ב' שמצביע חזרה על א'
  - ביטול ההצבעה מהשורש: שניהם לא נגישים אבל עם מונה 1
- בשימוש במערכות קבצים שבהן אי אפשר ליצור מעגלים

# אוספי זבל יותר מתוחכמים

- **generational collectors**: הזיכרון מחולק לשני אזורים (או יותר), אחד עבור עצמים צעירים והשני עבור ותיקים
  - זבל נאסף בתכיפות באזור הצעירים, אבל לעיתים נדירות באזור הותיקים
  - עצם צעיר ששורד מספר מסוים של מחזורי איסוף משודרג לאזור הותיקים
- **incremental collectors**: האוסף לא עוצר את התוכנית לכל זמן האיסוף; האוסף עוצר את התוכנית לזמן קצר, אוסף קצת זבל, והתוכנית ממשיכה לרוץ; מיועד לתוכניות אינטראקטיביות
- **concurrent collectors**: מיועדים למחשבים מרובי מעבדים; התוכנית ואוסף הזבל רצים במקביל



# חיסכון ביצירת עצמים

- אם נוצרים הרבה עצמים שמתקיימים זמן קצר, מנגנון איסוף הזבל יופעל לעיתים קרובות, ויגזול זמן ריצה רב.
- המתכנתת יכולה לנסות לחסוך חלק מהזמן הזה, על ידי שתשלוט בעצמה על חלק מניהול הזיכרון
- זה אפשרי בעיקר כאשר העצמים הרבים שנוצרים לפרק זמן קצר הם כולם ממחלקה אחת, למשל MyClass
  - המתכנתת ינהל בעצמו מאגר של עצמים מהמחלקה: רשימה מקושרת ששדה המחלקה `free_list` מצביע אליה
  - ננסה למחזר עצמים מטיפוס זה שאינם נחוצים יותר; כאשר לא צריך יותר עצם מסוים, נחזיר אותו למאגר של העצמים החופשיים

# חיסכון ביצירת עצמים (המשך)

- כאשר צריך עצם חדש מהמחלקה, אם יש במאגר עצם קיים שאינו בשימוש, נשתמש בו, אחרת ניצור עצם חדש
- זה כמובן דורש שלקוחות שצריכים עצם כזה ימנעו מקריאה לבנאי, כי זה תמיד ייצור עצם חדש
- לכן הבנאי יהיה פרטי. הלקוחות ייקראו לשרות מחלקה `alloc` שיכול לבנות עצמים על ידי קריאה לבנאי
- כאשר עצם אינו דרוש יותר, המתכנת צריך לקרוא ל `free`

# חיסכון ביצירת עצמים (המשך)

```
class MyClass {  
    private static MyClass free_list;  
    private          MyClass () {...}  
    public static MyClass alloc() {...}  
    public          void    free() {...}  
  
    // other fields and methods  
    private MyClass previous;  
}
```

# הקצאה

```
public static MyClass alloc() {  
    if (free_list == null)  
        return new MyClass();  
    else {  
        MyClass v = free_list;  
        free_list = v.previous;  
        return v;  
    }  
}
```

# שחרור

```
public void MyClass free() {
    this.field      = null;
    // for every non primitive field, so that the
    // referenced objects may be garbage collected
    this.previous = free_list;
    free_list      = this;
}
```

■ כדי לשחרר עצם שיש אליו התייחסות יחידה, המתכנתת תבצע:

```
x.free();
x = null;
```

# זיכרון דולף גם אם משתמשים באוסף זבל

יש עצמים נגישים, כלומר שיש מסלול של התייחסויות משורש אליהם, אבל שהתוכנית לא תיגש אליהם

אי אפשר לזהות אוטומטית את כל העצמים הללו; זו בעיה לא כריעה, יותר קשה מבעיית העצירה

דוגמאות נפוצות:

מערכים או מבנה נתונים ששומר התייחסויות לעצמים שהתוכנית אכן צריכה, אבל גם לעצמים שהיא לא צריכה יותר; הם לא ישתחררו; צריך השמה ל-null

התייחסות שאינה null אבל שהפעולה הבאה עליה תהיה השמה

בשפות שדורשות שחרור מפורש (C ו-C++, למשל) יש עוד סוג של דליפה, של עצמים שאינם נגישים אבל לא שוחררו

# finalize()

- שירות שכל עצם יורש מ-Object
- מופעל על ידי אוסף הזבל לפני שהעצם נמחק סופית
- דריסה שלו מאפשרת לבצע פעולות לפני שחרור; בעיקר שחרור משאבים שהעצם קיבל גישה אליהם (קבצים, למשל)
- עדיף לא להשתמש במנגנון הזה, כי ההפעלה של `finalize` עלולה להתבצע זמן רב לאחר שהעצם לא נגיש
- סיבוך נוסף נגרם מכך שהפעולה של `finalize` עלולה להפוך את העצם חזרה לנגיש; במקרה כזה הוא לא ישתחרר
- אם העצם יהפוך ללא נגיש בהמשך, אוסף הזבל ישחרר אותו, אבל לא יפעיל שוב את `finalize`

# סיכום נושא מנגנוני השפה

- תוכנה מופצת כ-bytecode; מורצת על ידי פרשן או מתקמפלת בזמן ריצה לשפת מכונה "ילידה"
- הפעלה של שגרה דרך מנשק היא מסובכת; לא צריכה להיות איטית, אבל בפועל לעיתים איטית
- יש הבדלי ביצועים גדולים בין JVM-ים שונים (JIT, מנשקים)
- איסוף זבל אוטומטי מפחית את כמות הפגמים בתוכנית, אבל יש לו מחיר בזמן ריצה ו/או בזיכרון; המחיר גדול במיוחד כאשר מספר גדול של עצמים קטנים חיים לאורך זמן
- תוכנית יכולה להשפיע על אוסף הזבל על ידי קריאה לו, על ידי שימוש בסוגי התייחסויות שלא מונעות איסוף, וע"י finalize