חלק 10

כיצד פועלים מנגנוני השפה

מבט מלמעלה

- הקומפיילר מתרגם את קוד המקור ל-byte code, קובץ `.class`., שמכיל ייצוג בינארי דחוס של מחלקה; לא נדון כמעט בשלב הזה
- 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

קומפילציה

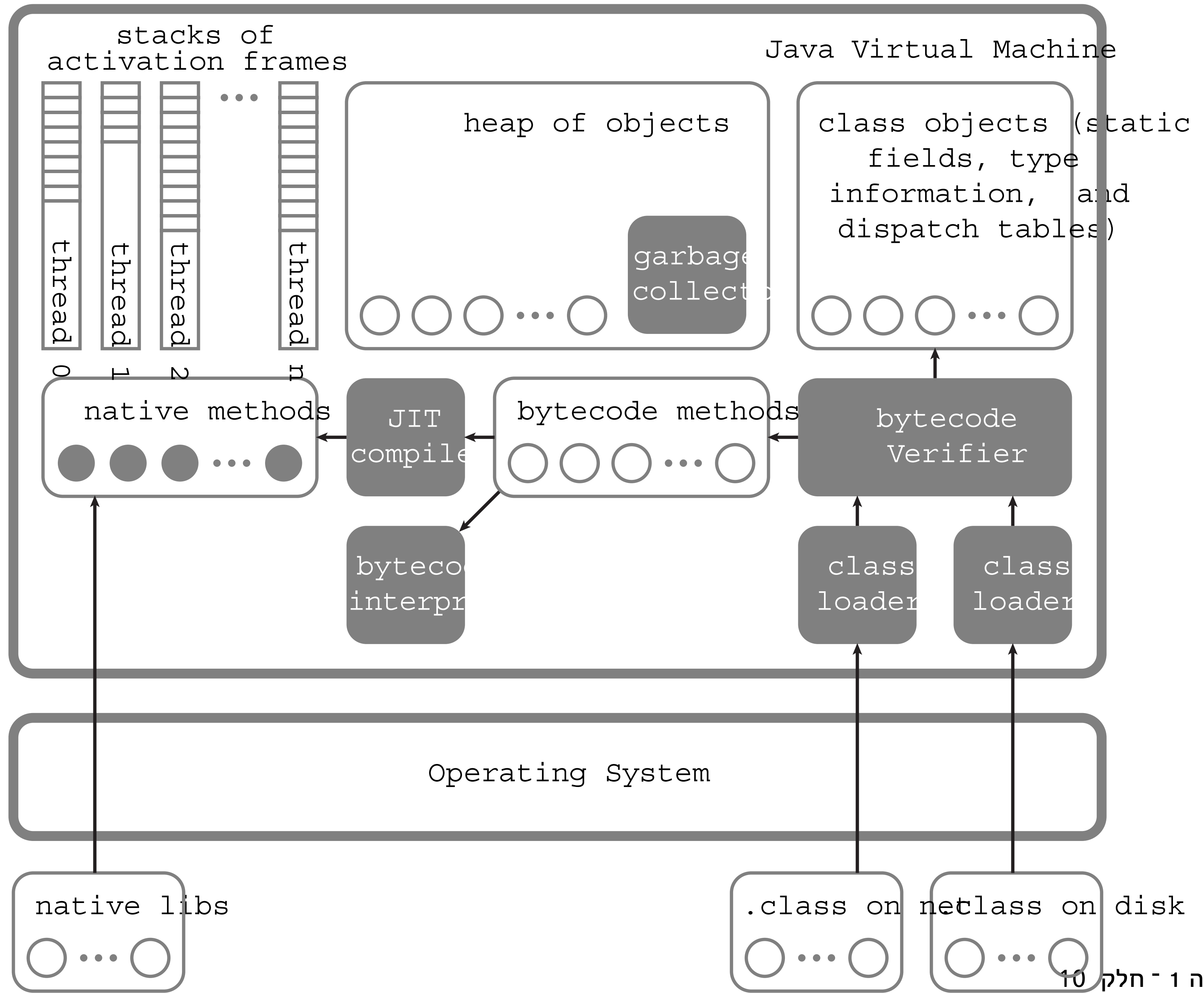
- מה הקומפיילר צריך לדעת כאשר הוא מקמפל מחלקה?
- צריך לדעת מהם השדות והשירותים שמוגדרים בכל טיפוס שהמחלקה משתמשת בו בשדות, משתנים, וארגומנטים
- מהיכן הקומפיילר שואב את המידע הזה?
- בדרך כלל, מהקבצים הבינאריים של הטיפוסים הללו
- אבל אם הם עדיין לא עברו קומפילציה, הקומפיילר מחפש את קוד המקור ומקמפל אותם 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

מבני הנתונים של ה JVM

- 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 (נקרא ה heap)
- לכל שרות שנקרא וטרם הסתיים, אוסף הפרמטרים האקטואליים והמשתנים המקומיים (נקרא רשומת הפעלה activation record): מחסנית זמן ריצה (פירוט בהמשך).

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
- אוסף הזבל (garbage collector) מופעל כדי למחזר קטעי זיכרון שאינם בשימוש.
- JIT compiler מתרגם שירותים מסוימים מ bytecode לשפת המכונה של המחשב לפי הצורך.



stacks of
activation frames
Java Virtual Machine
thread 0
thread 1
thread 2
thread n
heap of objects
garbage
collect
class objects (static
fields, type
information, and
dispatch tables)
native methods
JIT
compile
bytecode methods
bytecode
Verifier
byteco
interpr
class
loader
class
loader
Operating System
native libs
.class on net
.class on disk

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), ...
- 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.

קוד מקור לדוגמא

איך מיוצגים המחלקה והעצם, מה יתבצע בקריאה?

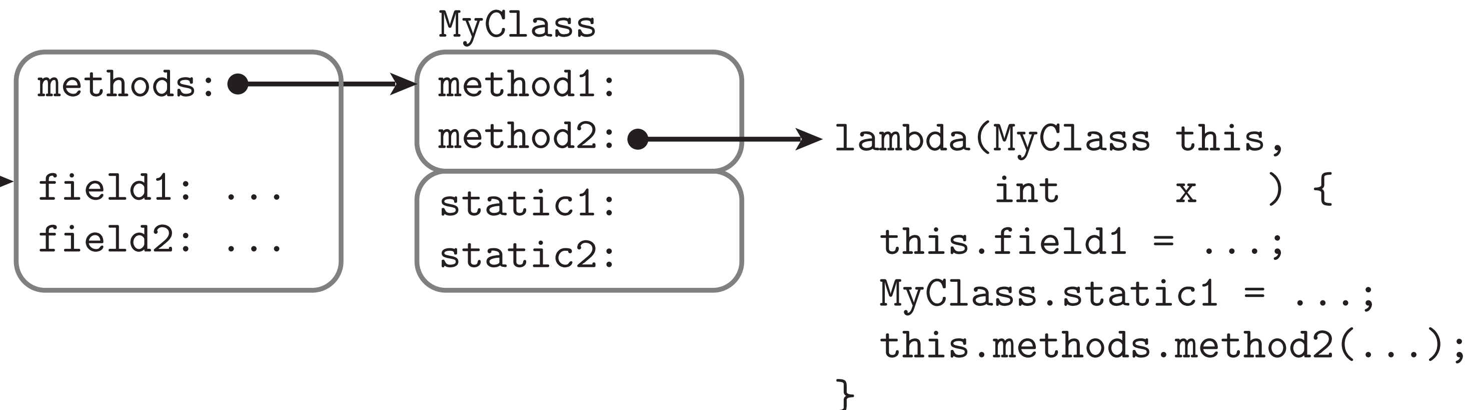
```
class MyClass {
  static int static1, static2;
         int field1, field2;
  void method1(int x) {
    field1 = ...; static2 = ... ;
    method2(...);     }   end method1
void method2(int y) { ... }
}                               end MyClass
MyClass o = new MyClass();
o.method1(5);
```

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, בסדר כלשהו, 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) {...}
 ...
}
```



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 o.method1(5)

- 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 o.methods[0] (o, 5)
- להפעלה כזו של שירות קוראים הפעלה וירטואלית, מכיוון שהשירות שיופעל תלוי בטיפוס הדינאמי, לא הסטאטי

אופטימיזציה: devirtualization

- אם ברור שהטיפוס הדינאמי של o זהה לטיפוס הסטאטי שלו, אז אין צורך בהפעלה וירטואלית
- למשל, בקוד

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

clearly o is a member of MyClass

- או אם `MyClass` מוגדר `final` או שהשירות `method1` מוגדר במחלקה `final`; זה מונע דריסה שלו
- במקרים כאלה, בזמן הטעינה של המחלקה שקוראת לשירות אפשר להחליף את ההפעלה הוירטואלית בהפעלה של השגרה המסוימת שצריך להפעיל על פי כתובתה בזיכרון
- אין צורך בחישוב הכתובת בעזרת ביטוי כמו `o.methods[0]`

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 `o` שטיפוס הסטאטי שלו הוא המנשק `I1`?
- 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 וירטואלית
- 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 tables)

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 הווירטואלי; הרכיב של ה-JVM שמבצע את הסימולציה נקרא bytecode interpreter
- 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

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'אוה 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, למשל).

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 דיגיטאלית), 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)

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 (לא יפורטו בקורס הזה).
- אוסף הזבל מופעל בדרך כלל באופן אוטומטי ע"י ה JVM כאשר הזיכרון שעומד לרשותו (כמעט) נגמר.
- יש אפשרות למתכנת לקרוא במפורש לאוסף הזבל.

חיסכון ביצירת עצמים

- אם נוצרים הרבה עצמים שמתקיימים זמן קצר, מנגנון איסוף הזבל יופעל לעיתים קרובות, ויגזול זמן ריצה רב.
- המתכנת יכול לנסות לחסוך חלק מהזמן הזה, על ידי שישלוט בעצמו על חלק מניהול הזיכרון.
- זה אפשרי בעיקר כאשר העצמים הרבים שנוצרים לפרק זמן קצר הם כולם ממחלקה אחת, למשל MyClass
- המתכנת 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, המתכנת יבצע:

```
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

גווני אפור

- עד עכשיו ההבחנה היא בין עצמים נגישים ללא נגישים
- בעצם יש גם עצמים שהם נגישים, אבל אולי אנו מוכנים לוותר עליהם
- בג׳אווה יש שני סוגים כאלה
- אלו סוגי התייחסויות שלא גורמות לאוסף הזבל לסמן את העצם כנגיש
- סוג אחד, התייחסויות רכות (soft references), מאפשרות שחרור אם אין התייחסויות רגילות לעצם ואם חסר זיכרון
- סוג שני, התייחסויות חלשות (weak references), גורמות לאוסף הזבל לשחרר את העצם אם אין אליו התייחסויות יותר חזקות (רגילות או רכות)

התייחסויות רכות

```
class CachedFile {
  String url;
  java.lang.ref.SoftReference<Data> cache;
  public CachedFile( String url ) {
    this.url = url;
    load();
  }
  private void load() {
    Data content = get it from the URL in ur
    cache =
       new SoftReference<Data>(content);
  }   only a soft ref remains!
```

התייחסויות רכות (המשך)

```
  public byte[] get() {
    if (cache.get() == null) load();
    return (byte[]) cache.get();
  }
}
```

reload

- get( ) הוא שרות של SoftReference<T> שמחזיר ערך מטיפוס T
- לאוסף הזבל מותר לשחרר עצמים שיש אליהם רק התייח-סויות רכות, והוא עושה זאת כאשר זה חיוני (בלי זה יהיה צורך להודיע על OutOfMemoryError)
- בעייה בפתרון הזה: אם יתבצע איסוף זבל מיד אחרי ה load

התייחסויות חלשות

- עצמים שיש אליהם רק התייחסויות חלשות (`java.lang.ref.WeakReference`) מסומן על ידי האוסף כזבל
- שימושי במקרים שבהם רוצים לשמור התייחסות לעצם מבלי שזו תמנע את איסופו; התייחסות בלי בעלות
- דוגמא: המחלקה `WeakHashMap` שמאפשרת לזכור מיפוי, אבל באופן שבו אם אין התייחסויות חזקות או רכות למפתח, המיפוי שקשור למפתח נעלם מאליו והמפתח משתחרר
- מבנה הנתונים הזה מונע דליפת זיכרון בגלל אי-הוצאת המיפוי ממבנה הנתונים
- בדרך כלל עדיף להוציא מפורשות את המיפוי
- לסיכום, סוג התייחסויות לא שימושי כל כך

תורי התייחסויות

- על ידי קשירת התייחסות חלשה/רכה/פאנטום לעצם מסוג `java.lang.ref.ReferenceQueue`, אפשר לקבל מאוסף הזבל מעין הודעה שהעצם המיוחס נאסף
- אם התייחסות רכה/חלשה כזו קשורה לתור, אוסף הזבל מוסיף את ההתייחסות לתור לאחר שהעצם המיוחס נאסף והזיכרון שוחרר; קריאה ל-`get` תחזיר `null`
- התייחסויות מסוג פאנטום (`PhantomReference`) מיועדות אך ורק לקבלת הודעה אודות איסוף של עצם; כמו התייחסויות חלשות הן אינן מונעות איסוף, אבל השירות `get` שלהן תמיד מחזיר `null`

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

טעינה וקישור דינאמי של מחלקות

- תוכנית ג׳אווה יכולה לטעון במהלך הריצה שלה מחלקות באופן סתום או מפורש
- כאשר תוכנית מתייחסת למחלקה חדשה שלא הייתה בשימוש עד לאותו רגע, מתבצעת טעינה אוטומטית של המחלקה; מחלקות אינן נטענות בדרך כלל לפני שיש בהן צורך
- זו טעינה סתומה: התוכנית אינה מבקשת מפורשות לטעון את המחלקה
- היכן ה-JVM מחפש מחלקות? בדרך כלל במדריכים ומארזי jar שמוגדרים בתור ה-class path של התוכנית; ניתן לקבוע אותו על ידי משתנה סביבה או על ידי ארגומנט ל-JVM
- אבל לפעמים ה-JVM מחפש במקומות אחרים, והתוכנית גם יכולה לטעון מחלקות באופן יזום ובאופן מפורש

טועני מחלקות (class loaders)

- טעינה של מחלקות מתבצעת בעזרת הרחבות של המחלקה המופשטת `java.lang.ClassLoader`
- המחלקה המופשטת הזו מגדירה שירותים שמקבלים ייצוג של מחלקה בפורמט של קובץ בינארי (קובץ class.) ובונים ייצוג שלה בתוך ה-JVM, ייצוג שבעצמו מיוצג על ידי עצם מהמחלקה `java.lang.Class`
- שני השירותים הללו הם `defineClass` ו-`resolveClass`
- מחלקות מרחיבות מגדירות את השירות `loadClass` שתפקידו למצוא את הייצוג הבינארי ולטעון אותו בעזרת שני השירותים של המחלקה המופשטת

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
- טעינה דינאמית של מחלקות מקנה גמישות ומאפשרת ליצור תוספים לתוכניות (התוכניות צריכות לתמוך בכך)