תוכנה 1 בשפת Java

שיעור מספר 9: "אל תסתכל בקנקן"

מנגנוני השפה: טבלת השרותים, איסוף זבל,
טעינת מחלקות והשתקפות

בית הספר למדעי המחשב

אוניברסיטת תל אביב

הרצת תכנית - מבט מלמעלה

MyClass.java

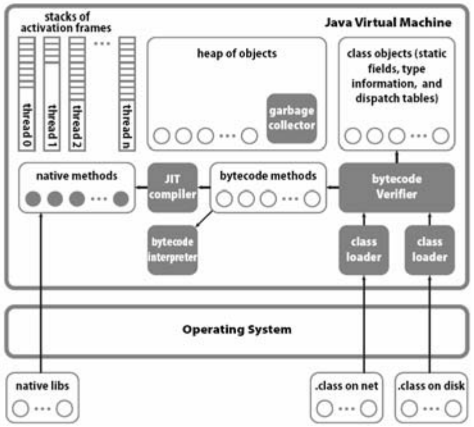
קוד מקור

מהדר (קומפיילר)
javac

MyClass.class

Byte code
(ייצוג בינארי)

תכנית "ילידה" java
(כתובה בדר"כ בשפת C)



Java Virtual Machine (JVM)

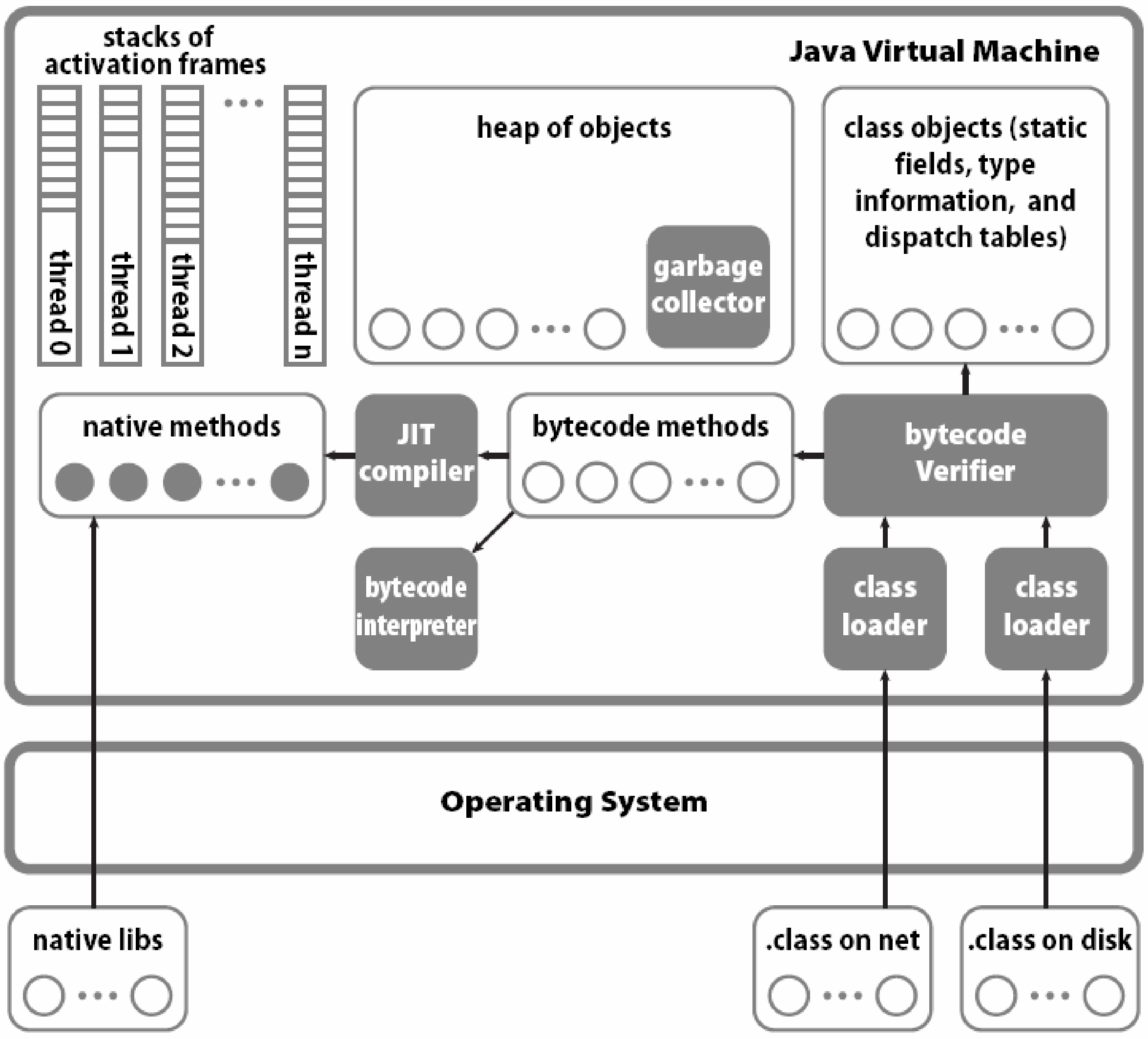
טעינת הקבצים הבינאריים של התכנית (קבצים בדידים או במארז jar.)

ביצוע פקודות התכנית (כולל הקצאת ושחרור זיכרון וקריאה לשירותים)

הידור (קומפילציה)

- מה הקומפיילר צריך לדעת כאשר הוא מקמפל מחלקה?
  - השדות והשירותים שמוגדרים בכל טיפוס שהמחלקה משתמשת בו בשדות, במשתנים, ובארגומנטים שלה
- 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

Java Virtual Machine (JVM)



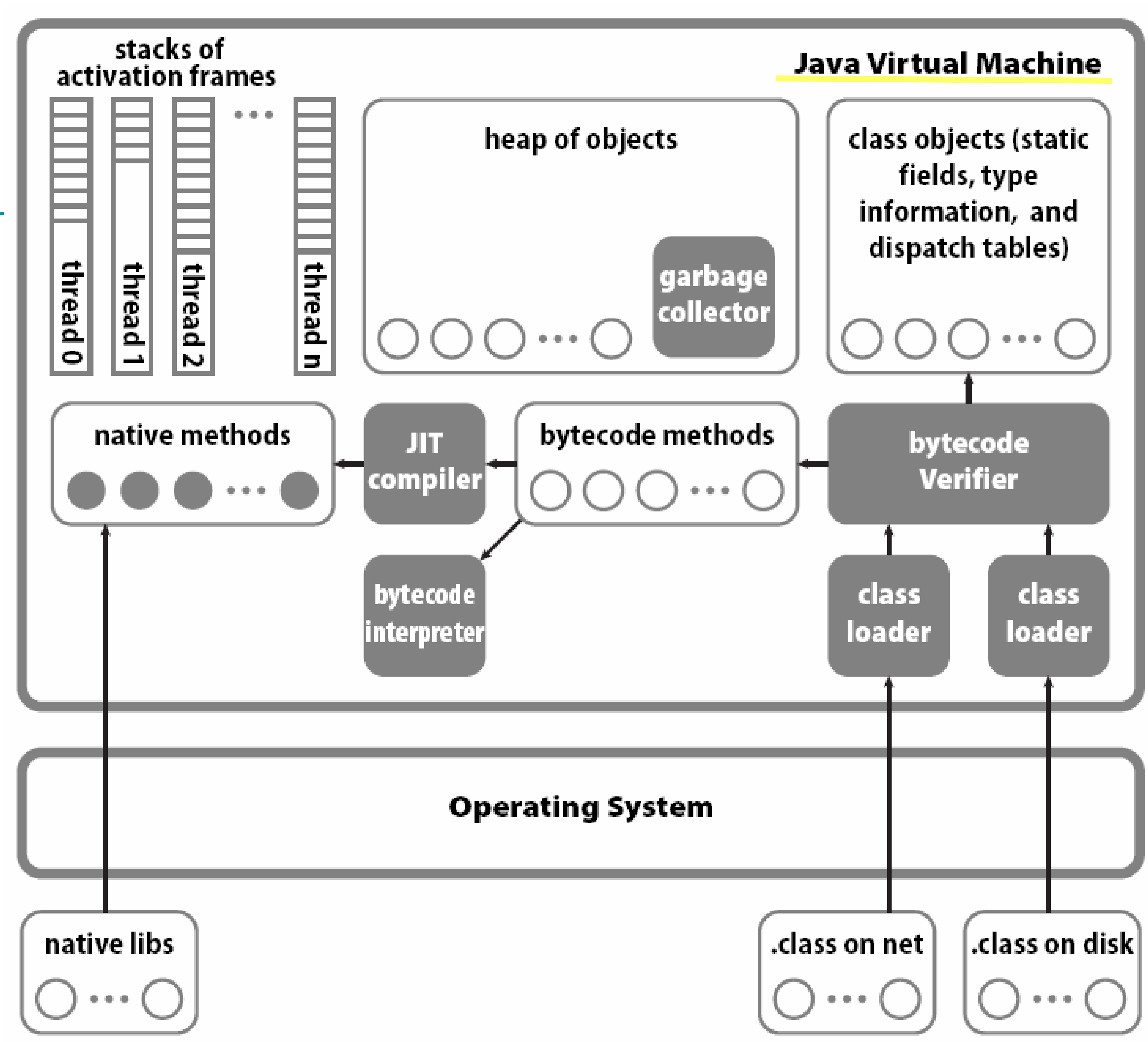
מבני הנתונים של ה-JVM

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

- מבנה נתונים השומר לכל מחלקה (שנטענה) את:
  - הקוד של המחלקה (bytecode): שירותים, בנאים, אתחול סטטי
  - לחלק מהשירותים יכול להישמר גם קוד בשפת מכונה.
  - אוסף השדות הסטטיים שלה.
- **Heap**: שומר לכל עצם שנוצר את אוסף שדות המופע
- **מחסנית זמן ריצה (Execution Stack)**
  - שומרת לכל שרות שנקרא וטרם הסתיים את אוסף הפרמטרים האקטואליים והמשתנים המקומיים (פיסת מידע זו נקראת **רשומת הפעלה** או **activation record**)

החלקים הביצועיים של ה-JVM

- **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 collector
class objects (static fields, type information, and dispatch tables)
native methods
JIT compiler
bytecode methods
bytecode Verifier
bytecode interpreter
class loader
class loader
Operating System
native libs
.class on net
.class on disk

מחסנית זמן ריצה (Execution Stack)

- 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 חוזרת ראשונה). לכן, רשומות ההפעלה שמורות במחסנית.
- 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

```
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

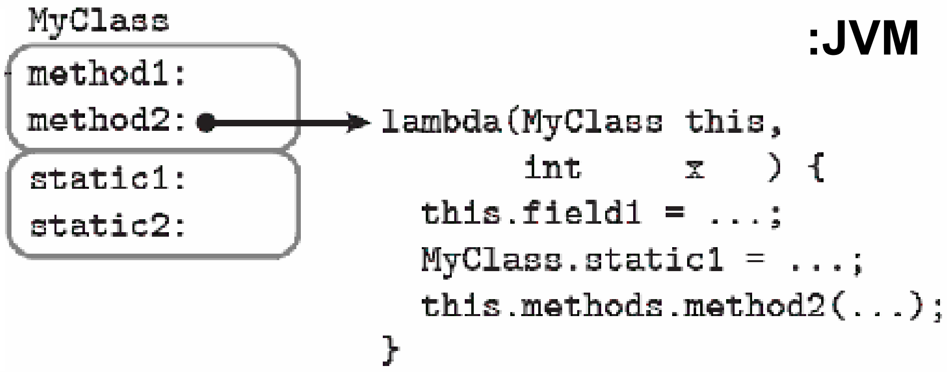
יצוג מחלקה בזמן ריצה

- **מחלקה** מיוצגת ע"י מבנה נתונים הכולל :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.

קוד מקור:

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

:JVM



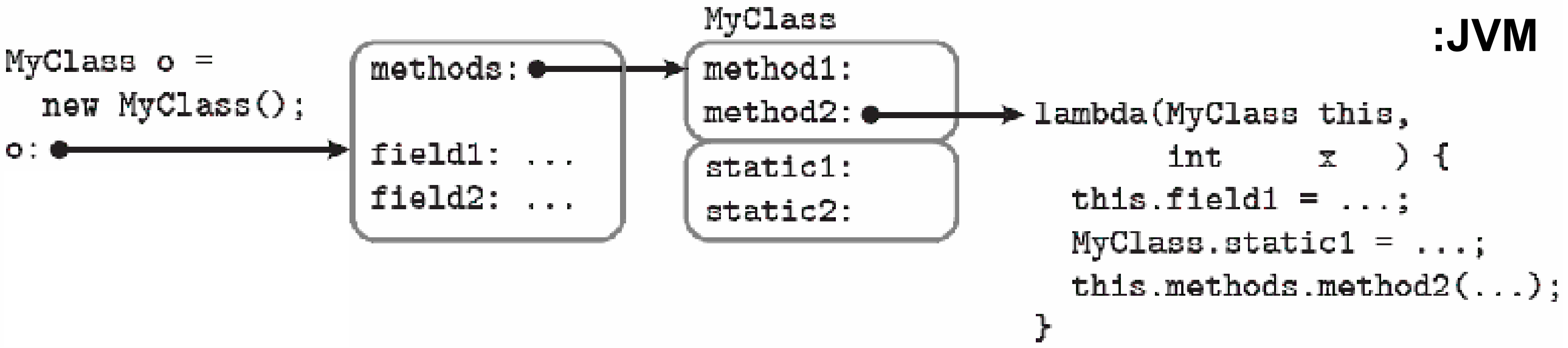
יצוג עצם בזמן ריצה

- **עצם** מיוצג ע"י מבנה נתונים שמכיל הצבעה למבנה הנתונים של המחלקה שהוא שייך אליה ואת ערכי שדות המופע.
- **התייחסות** היא הצבעה למקום בזיכרון שבו מתחיל המבנה של העצם המיוחס (לפעמים יותר מסובך; נסביר בהמשך).
- עצמים נוצרים באופן דינאמי באזור זיכרון שנקרא **heap**.

קוד מקור:

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

:JVM



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 ה-dispatch של מחלקה מכילה הצבעות לשירותים שהגדירה המחלקה וגם לשירותים שהמחלקה ירשה 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 dispatch של המחלקה

שימוש בשדות

- **שימוש בשדה מופע:**
  - בדוגמא השתמשנו בסימון this.field1
  - בפועל זה יופיע ככתובת ב-byte code לשדה ע"י הוספת ההיסט (המרחק מתחילת העצם) של השדה לכתובת this.
- **שימוש בשדה מחלקה** (שימוש פשוט יותר):
  - כאשר המחלקה נטענת לזיכרון, נקבעות הכתובות של שדות המחלקה, שלא זזים במהלך התכנית.
  - לכן, אפשר להחליף כל התייחסות לשדה מחלקה בהתייחסות לכתובת של השדה בזיכרון.
  - בדוגמה הסימון היה MyClass.static1, אבל בפועל זו כתובת אבסולוטית.

הפעלה של שירות

```
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 <u>הפעלה וירטואלית</u>, מכיוון שהשירות שיופעל תלוי בטיפוס הדינאמי, לא הסטאטי

Devirtualization - אופטימיזציה

- 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` (כי במצבים אלו אין דריסה של השירות)
- 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]`

הפעלה סטטית לעומת הפעלה דינאמית

- כדי לחדד את ההבדל בין הפעלה דינאמית של שרות (וירטואלית) ובין הפעלה סטטית של שרות נסתכל על ההבדל בין שרותי מופע ושרותי מחלקה
- הפעלה של שרותי מופע ב- Java היא דינאמית
- הפעלה של שרותי מחלקה היא סטטית
- כתוצאה מכך, כאשר מחלקה יורשת מגדירה שרות מחלקה באותו שם של שרות מחלקה במחלקת הבסיס היא אינה דורסת (override) את השרות הנורש את מסתירה (hide) אותו

```java
public class Animal {
     public static void hide() {
          System.out.format("The hide method in Animal.%n");
     }
     public void override() {
          System.out.format("The override method in Animal.%n");
     }
}
```

```java
public class Cat extends Animal {
     public static void hide() {
          System.out.format("The hide method in Cat.%n");
     }
     public void override() {
          System.out.format("The override method in Cat.%n");
     }
 }
```

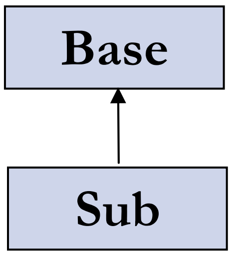
```java
public class Client{
     public static void main(String[] args) {
          Cat myCat = new Cat();
          Animal myAnimal = myCat;
          //myAnimal.hide();    //BAD STYLE
          Animal.hide();        //Better!
          myAnimal.override();
     }
}
```

מה יודפס?

```
The hide method in Animal.
The override method in Cat.
```

מבנה עצם ממחלקה מרחיבה

- מבנה של מחלקה מרחיבה (Sub) נגזר ממבנה מחלקת הבסיס (Base):
  - שירותים שנוספו ב-Sub יופיעו לאחר השירותים שנורשו או נדרסו מ-Base
- מבנה של עצמים מ-Sub נגזר ממבנה עצמים ממחלקת הבסיס Base:
  - שדות מופע בעצמים של Sub יופיעו לאחר שדות המופע שב-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 מממש 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 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, vitrual 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)

הרצה וקומפילציה של bytecode

- בקובץ class. השירותים מיוצגים ב-bytecode
  - **bytecode**: שפת מכונה של מחשב וירטואלי
  - בשפה זו לא כל הפקודות פשוטות, למשל invokeinterface
- לאחר טעינת תכנית לזיכרון, ה-JVM יכול להריץ שירותים על ידי סימולציה של המחשב הוירטואלי.
  - **Bytecode interpreter**: רכיב ה-JVM שמבצע את הסימולציה
  - בסימולציה יש מחיר גבוה לפעולות פשוטות, כמו חיבור 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 (JIT)

- קומפילציה מ-bytecode לשפת מכונה (native code)
- נקראת כך כי היא מתבצעת בזמן ריצה, ממש לפני השימוש בקוד, ולא כחלק מהכנת התוכנה להפצה
- לרוב JIT פועל על שירות לאחר שהתברר שהוא מופעל הרבה
  - מונע קומפילציה יקרה של שירותים שכמעט (או לא) מופעלים
  - הביצועים של תוכנית משתפרים לאורך הריצה
- 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

אז למה bytecode?

- אם ממילא תכנית java מתקמפלת בסופו של דבר לשפת מכונה, למה לקמפלה בזמן ריצה ולא בעת אריזתה להפצה?
- **קומפילציה בעת אריזה יעילה יותר**:
  - היא מתבצעת פעם אחת עבור הרצות רבות
  - בזמן אריזה אפשר להפעיל אופטימיזציות יקרות (בזמן ריצה לא)
- הפצת תוכנה כ-bytecode משיגה שתי מטרות;
  - היכולת להשתמש **בתוכנה ארוזה אחת על פלטפורמות שונות** (מעבדים שונים ומערכות הפעלה שונות)
  - **בטיחות**: ה-bytecode verifier בודק את התכנית לפני הרצתה כדי: לוודא קיום דרישות השפה, שיפור בטיחות מערכות המחשב ומניעת סוגים מסוימים של תקיפות

תוכנה לשימוש בפלטפורמות שונות

- שתי רמות ליכולת תוכנה לרוץ על פלטפורמות שונות:
  - קוד המקור מתאים למספר פלטפורמות, אבל צריך לקמפל אותו לכל פלטפורמה בנפרד.
  - התאמה לא רק ברמת קוד המקור, אלא גם ברמת התוכנה הארוזה להפצה.
- תוכנית Java שייכת לרמה השנייה הודות לשימוש ב-bytecode.
- סיסמת Sun לשיווק java: **Write Once, Run Anywhere**
  - משמעות: לכתוב ולארוז את התוכנה פעם אחת ולהריץ אותה על פלטפורמות רבות (על מגוון מעבדים ומערכות הפעלה)
  - יכולת זו חשובה כי אריזת תוכנה להפצה היא פעולה ויקרה.

שפת מכונה וירטואלית כמנגנון הפצת 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

- (מנגנון שלא קיים עדיין)
- 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?

- **זבל**: עצמים שאין אליהם התיחסות
- יותר קל להגדיר את **העצמים שאליהם כן אפשר להתייחס**:
  - עצמים שיש אליהם התייחסות משם גלובאלי; בג'אווה, שמות גלובליים מתאימים בדיוק לשדות מחלקה
  - עצמים 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
  - 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
- **שורשים**: התייחסויות שברור שלתוכנית יש גישה אליהם
- שורשים בג'אווה כוללים:
  - שדות מחלקה
  - 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 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 (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 p1 בזיכרון בצד א' לכתובת p2 בצד ב'
- משנים את תוכן שטח הזיכרון בכתובת p1
- מדליקים סיבית שמסמנת שהעצם כבר הועתק לצד ב'
- כותבים בשטח הזיכרון את הכתובת החדשה p2
- 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 q1 שולפים מהעצם ב-q1 את הכתובת החדשה של העצם q2 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 ההתייחסויות 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 ההתייחסויות, 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**: אוסף הזבל לא עוצר את התכנית לכל זמן האיסוף; הוא עוצר את התכנית לזמן קצר, אוסף קצת זבל, והתכנית ממשיכה לרוץ; מיועד לתוכניות אינטראקטיביות.
- **Concurrent Collectors**: מיועדים למחשבים מרובי מעבדים; התוכנית ואוסף הזבל רצים במקביל

חיסכון ביצירת עצמים
(אינו רלוונטי ל-JVM-ים חדשים)

- אם נוצרים הרבה עצמים שמתקיימים זמן קצר, מנגנון איסוף הזבל יופעל לעיתים קרובות, ויגזול זמן ריצה רב.
- המתכנת יכול 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
  - ננסה למחזר עצמים מטיפוס זה שאינם נחוצים יותר; כאשר לא צריך יותר עצם מסוים, נחזיר אותו למאגר של העצמים החופשיים

חיסכון ביצירת עצמים
(אינו רלוונטי ל-JVM-ים חדשים)

- כאשר צריך עצם חדש מהמחלקה: אם יש במאגר עצם קיים שאינו בשימוש, נשתמש בו. אחרת ניצור עצם חדש
- זה 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 לשרות מחלקה free

חיסכון ביצירת עצמים
(אינו רלוונטי ל-JVM-ים חדשים)

```
class MyClass {
  private static MyClass free_list;
  private        MyClass () {...}
  public  static MyClass alloc()  {…}
  public         void    free()   {…}

  // other fields and methods
  private MyClass previous;
}
```

מאגר העצמים ממומש כרשימה מקושרת כשהשדה free_list מצביע על האיבר הראשון שבה

חיסכון ביצירת עצמים
(אינו רלוונטי ל-JVM-ים חדשים)

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

חיסכון ביצירת עצמים
(אינו רלוונטי ל-JVM-ים חדשים)

```
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;
```

חיסכון ביצירת עצמים
(אינו רלוונטי ל-JVM-ים חדשים)

- הפתרון לעיל מבוסס על תבנית התיכון Object Pool
- פתרון זה אינו טוב ואפילו פוגע בביצועים של אוספי זבל חדשים:

http://www-128.ibm.com/developerworks/java/library/j-jtp09275.html?ca=dgr-jw22JavaUrbanLegends

- ב-JVM-ים חדשים יש מימוש מקביל ל-Object Pool
- כמו כן, Object Pool הממומש ע"י המשתמש פוגע באוספי זבל מסוג Copying. למה?
  - כי לעצמים יש טווח חיים ארוך

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:
  - מערך או מבנה נתונים ששומר התייחסויות לעצמים שהתוכנית אכן צריכה, אבל גם לעצמים שהיא לא צריכה יותר. עצמים אלו לא ישתחררו; צריך השמה ל-null.
  - התייחסות שאינה בשימוש זמן רב ולאחר מכן מוחלפת
  - 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 היתה בין עצמים **נגישים** ל**לא נגישים**, אולם יש גם עצמים שהם נגישים, אבל אולי אנו מוכנים לוותר עליהם
- בג'אווה מוגדרות 5 רמות נגישות, בעזרת מחלקות לתאור התייחסויות (היורשות מהמחלקה `Reference<T>`):
  - אלו סוגי התייחסויות שלא גורמות לאוסף הזבל לסמן את העצם כנגיש (בחוזקה)
  - **התייחסויות רכות** (soft references): מאפשרות שחרור אם אין התייחסויות רגילות לעצם ואם חסר זיכרון
    - שימושיות בהטמנה רגישת זיכרון (sensitive memory cache)
  - **התייחסויות חלשות** (weak references): גורמות לאוסף הזבל לשחרר את העצם אם אין אליו התייחסויות יותר חזקות (רגילות/רכות)
    - שימושיות במיפוי של עצמים (canonicalizing mappings)

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 url
    cache = new SoftReference<Data>(content);
  }  // only a soft ref remains!
```

התייחסויות רכות (המשך)

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

- `get()` הוא שרות של `SoftReference<T>` שמחזיר ערך מטיפוס T
- לאוסף הזבל מותר לשחרר עצמים שיש אליהם רק התייחסויות 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

- עצם שיש אליו רק התייחסות חלשה (`java.lang.ref.WeakReference`) מסומן ע"י 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

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

- **טעינה סתומה (טעינה ללא בקשה מפורשת):**
  - כאשר תכנית מתייחסת למחלקה חדשה שלא היתה עד כה בשימוש, מתבצעת טעינה אוטומטית של המחלקה.
  - מחלקות אינן נטענות בדרך כלל לפני שיש בהן צורך
  - לרוב, ה-JVM מחפש את המחלקה שצריך לטעון במדריכים ומארזי jar שמוגדרים ב-class path של התכנית (ניתן לקבוע אותו ע"י משתנה סביבה או ארגומנט ל-JVM)
- בנוסף, תכנית יכולה לטעון מחלקות באופן יזום ומפורש
  - ע"י הרחבה של: **`java.lang.ClassLoader`**
  - מתאים ל-plug-in

השתקפות (Reflection)

- בג'אווה מבנה הקוד (מחלקות, שירותים, ושדות) זמין בזמן ריצה וניתן לחקור אותו בעזרת המחלקות:
- **`java.lang.Class`**
- **`java.lang.Package`**
- **`java.lang.reflect.Constructor`**
- **`java.lang.reflect.Method`**
- **`java.lang.reflect.Field`**
- למשל, Class היא מחלקה "רגילה" המשמשת בתור **מטה-מחלקה** (מחלקה המתארת מחלקות אחרות)

המחלקה Class

- מספר דרכים לקבל עצם מסוג Class עבור מחלקה:

```
Rectangle r = …
Class x = r.getClass();// an Object method
Class y = Rectangle.class;// literal
Class z = Class.forName("Rectangle");  // lookup
```

- עצם שמייצג מחלקה מחזיק פרטים לגביה: שמה, את מי היא מרחיבה וממשת, את השדות שלה, את השירותים והבנאים שלה
- בניית עצם ע"י שימוש בבנאי ברירת המחדל:

```
Rectangle a = (Rectangle) z.newInstance();
```

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)