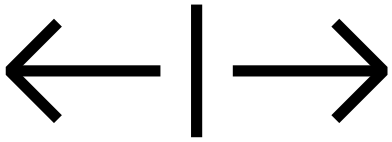
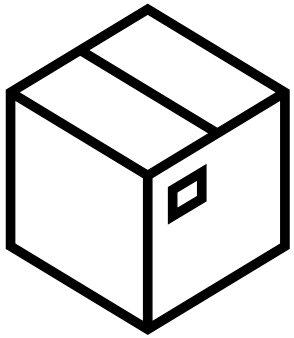
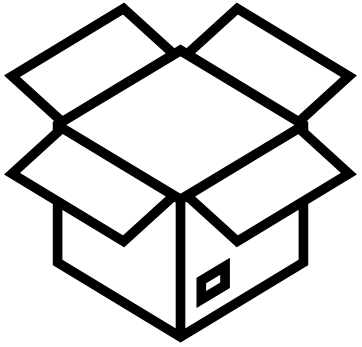
WIDENING
AUTO-BOXING
UNBOXING







הערות

-במצגת לא קיימת כלל התייחסות לנושא GENERICS , אשר נלמד בשלב יחסית מאוחר בקורס, או ל VAR_ARGS אשר לא נלמד כלל.

- אין התייחסות להסבות מפורשות (explicit casting) מתוך הנחה שנושא זה מכוסה בתחילת הקורס.

- המושגים "הרחבה", "קריאה קפדנית" , "קריאה רופפת" הם פירושים שלי למונחים באנגלית המצויים בתיעוד השפה.
- כל החומר עד עמ' 39 מוביל להבנה הדרושה למציאת מתודה קרובה ביותר בהעמסה בת פרמטר אחד.
- החל מעמ' 40 קיים החומר הדרוש עבור העמסה מרובת פרמטרים.

- קיים סיכום ממצה בסוף המצגת. הטבלה הראשונה מרכזת את נושא הטיפוסים והליטרלים. שאר חלקי הסיכום רלוונטיים יותר לנושא של בחירת מתודה קרובה ביותר בהעמסות.

Widening (הרחבה)

- הרחבה היא המרה מרומזת מטיפוס פרימיטיבי אחד לטיפוס פרימיטיבי אחר שמכיל את טווח הערכים שלו.
  תקינות ההרחבה נבדקת בזמן הידור.

* *הערה: בתיעוד השפה, מונח זה נקרא primitive widening, כדי להבדילו מ reference widening, שמתקיימת בין רפרנסים. במצגת זו נשתמש ב widening לציון הרחבה בין פרימיטיביים בלבד.*

- סדר ההכלה של טווחי הערכים עבור הטיפוסים המספריים:

byte ⊆ short ⊆ int ⊆ long ⊆ float ⊆ double

- **דוגמא להרחבה תקינה בטיפוסים מספריים**

int i = 6;

float f = i;

- **ניסיון הרחבה מגבוה לנמוך (ביחס לסדר ההכלה הנ"ל) יגרום לשגיאת הידור (הידועה כ "type mismatch")**

double d = 5.7;

float f = d;

- **boolean הפרימיטיבי נושא את הערכים true ו false שאינם מספריים ולכן לא ניתן לבצע הרחבה ממנו או אליו.**

- **char הפרימיטיבי יכול לשאת ערכים בתחום [0,65535], לכן הרחבות ממשתנה char יכולות להתבצע רק אל משתנים מטיפוס int ומעלה (לפי סדר ההכלה בעמוד 3). לא ניתן להרחיב אליו!**

- **דוגמא להרחבה תקינה ממשתנה char**

```
char c = 'A';
int i = c;
```

- ניסיון להרחיב <u>ממשתנה char</u> למשתנה נמוך מ int יגרום לשגיאת הידור

char c = 'A';

byte b = c;

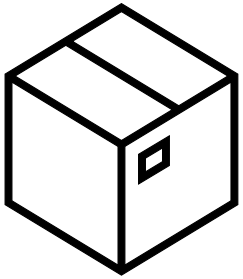
- ניסיון להרחיב <u>אל</u> משתנה char יגרום לשגיאת הידור

byte b = 50;

char c = b;

- **סיכום הרחבה :** byte ⊆ short ⊆ int ⊆ long ⊆ float ⊆ double

char ⊆

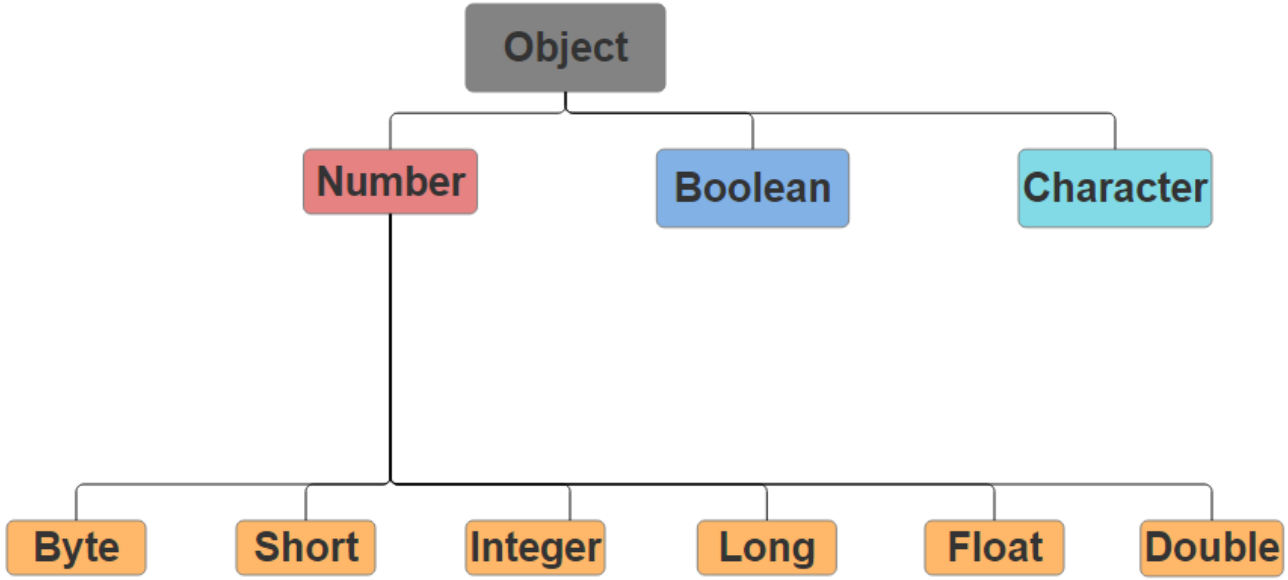


Auto-Boxing

- לכל פרימיטיבי יש מחלקה של טיפוס עוטף ואלו הן:

Integer עוטפת של int, Character עוטפת של char, שאר המחלקות העוטפות זהות בשמן לטיפוס הפרימיטיבי אך מתחילות באות גדולה (לדוגמא Double).

- עץ הירושה של הטיפוסים העוטפים:



- **Auto-Boxing הוא מנגנון הממיר פרימיטיבי <u>לעוטף שלו</u>.**

לדוגמא:

int i = 6;

Integer N = i;

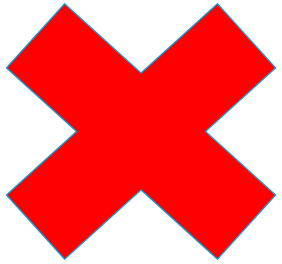
- **גם** *Integer N = 13;* **נחשב לauto-boxing** (דיון על <u>ליטרלים</u> מספריים בהמשך המצגת...).

- **ניסיון להצביע על <u>משתנה</u> <u>פרימיטיבי</u> מטיפוס אחד באמצעות <u>משתנה</u> <u>עוטף</u> של טיפוס אחר יגרום לשגיאת הידור:**

int num = 14;

Long l = num; // ERROR

Double d = num; // ERROR



ליטרל אות וליטרל בוליאני

- ליטרל אות (לדוגמא 'A') ייחשב כ **char** פרימיטיבי.

עבור השמות בלבד , קיים מנגנון המאפשר הצבעה על ליטרל אות באמצעות **Number,Byte,Short,byte,short** כל עוד הליטרל בטווח הערכים של המצביע.

- ליטרל בוליאני (true/false) ייחשב כ **boolean** פרימיטיבי.

- ההשמות הבאות חוקיות :

Character C = 'A'; // auto-boxing

Boolean B = true; // auto-boxing

int i = 'a'; // widening

byte b = 'a'; // in range & allowed by mechanism

Byte B = 'a'; // in range & allowed by mechanism

ליטרלים מספריים

- ליטרל שלם (דוגמת 6) יחשב כ **int** , אך עבור **השמות** קיים מנגנון המאפשר הצבעה על ליטרל שלם באמצעות טיפוסים נמוכים יותר (פרימיטיבי/עוטף) כל עוד הליטרל בטווח הערכים של המצביע, ולכן ההשמות הבאות חוקיות:

int i = 6; // obvious

byte b = 6; // in range & allowed by mechanism

Integer N = 6; // auto-boxing

Byte J = 6; // in range & allowed by mechanism

double d = 6; // widening

char c = 3; // in range & allowed by mechanism

- **ליטרל דצימלי (דוגמת 3.8) יחשב כ double . רק השמות כדלקמן חוקיות:**

double d = 3.8; //obvious

Double J = 3.8; //auto-boxing

- **ניתן לכפות על הקומפיילר להתייחס לליטרל <u>שלם</u> כאל long <u>(פרימיטיבי)</u> ע"י הצמדת האות L (קטנה או גדולה) מימין למספר:**

6L / 6l

- **ניתן לכפות על הקומפיילר להתייחס לליטרל מספרי כאל float (פרימיטיבי) ע"י הצמדת האות F (קטנה או גדולה) מימין למספר:**

3.8f / 3.8F / 6F / 6f

- **ניסיון להצביע על קבוע ליטרלי עם מצביע לא מתאים יגרום לשגיאת הידור. כל הדוגמאות הללו יכשלו:**

Float f = 20; //not allowed by mechanism

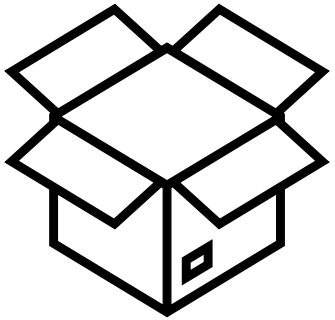
Double D = 3.4F; /*only float,

, double , Float , Number, Object allowed */

Integer I = 'A'; //not allowed by mechanism

Float G = 20.4; // can't convert double downwards

float f = 4.3; // can't convert double downwards



Unboxing

- זהו המנגנון בכיוון ההפוך לauto-boxing: משתנה מטיפוס פרימיטיבי יכול להצביע על משתנה מהטיפוס העוטף שלו.

לדוגמא:

Integer Num = new Integer(4);

int i = Num ;

לחצו כאן לעיון בטבלת סיכום חלק זה

מתודות עם טיפוסים פרימיטיביים/עוטפים

- שימוש נפוץ של autoboxing/unboxing הוא הצבעה על ערך מוחזר עוטף/פרימיטיבי באמצעות משתנה פרימיטיבי/עוטף שלו.

- ניסיון לדרוס מתודה ע"י שינוי הטיפוס המוחזר מפרימיטיבי לעוטף שלו או מעוטף לפרימיטיבי שלו יגרום לשגיאת הידור.

- שינוי פרמטרים של מתודה מעוטף לפרימיטיבי שלו או מפרימיטיבי לעוטף שלו מהווה העמסה חוקית.

העמסה בת פרמטר אחד

נדרשת הגדרה של כמה מושגים על מנת לדמות את עבודת המהדר בתהליך מציאת מתודה קרובה ביותר בהינתן מספר העמסות בעלות אותו שם.

* נתייחס רק לארגומנטים פרימיטיביים, עוטפים , מערכים, ממשקים ועצמים שאינם מחלקות גנריות.

היחס subtype

- מסומן : < , לדוגמא int <: float . זהו יחס רפלקסיבי וטרנזיטיבי. האופרנדים של היחס הם טיפוסים.

- יהי C טיפוס פרימיטיבי. אז C הוא subtype של עצמו ושל כל הפרימיטיביים הרחבים ממנו לפי יחס הכלת טווח הערכים מעמ' 6.

- יהי C עצם/ממשק. אז C הוא subtype של עצמו ושל כל מחלקות העל שלו/ממשקים מהם הוא יורש וכל הממשקים אותם הוא (ומחלקות-העל שלו) מממשים . כמו כן, כל הממשקים הם subtype של Object.

<u>לדוגמא</u>: נניח Animal ממשת MyInterface , ו Dog תת-מחלקה של Animal. אז מתקיים:

Animal <: MyInterface,

Dog <: MyInterface,

Dog <: Animal,

Animal <: Object,

Dog <: Object ,

MyInterface <: Object

** הערה : לפי תיעוד השפה , היחס subtype בין רפרנסים מגדיר את המונח widening reference. כאמור בתחילת המצגת, לא נשתמש במונח זה.*

- יהי $T[\,]$ טיפוס מערך של איברים מסוג T . אז מתקיים

$$T[\,] <: T[\,]$$
$$T[\,] <: Object$$
$$T[\,] <: Clonable$$
$$T[\,] <: \mathbf{java.io.Serializable}$$

- אם $S[\,], T[\,]$ טיפוסי מערכים של רפרנסים מהסוגים S , T בהתאמה , אז בנוסף מתקיים:

$$S[\,] <: T[\,] \quad \text{אמ"מ} \quad S <: T$$

היחס supertype

- מסומן $>:$, זהו היחס ההופכי ל subtype .

מכאן נובע:

- הוא רפלקסיבי וטרנזיטיבי

- בהינתן טיפוסים C ו D,
  $C <: D$ אמ"מ $D :> C$

קריאה קפדנית (STRICT Invocation)

- עבור ארגומנט נתון, קריאה למתודה נקראת קפדנית אם בחתימת המתודה, הפרמטר מהווה אחד מבין האופציות הבאות (נכנה אותן בשם "מקטעים"):

  - (same) אותו טיפוס עבור ארגומנט פרימיטיבי
  - (same) אותו טיפוס עבור ארגומנט רפרנס
  - widening עבור ארגומנט פרימיטבי
  - supertype עבור ארגומנט רפרנס

קריאה רופפת (LOOSE Invocation)

- עבור ארגומנט נתון, קריאה למתודה נקראת רופפת אם בחתימת המתודה, הפרמטר מהווה אחד מבין האופציות הבאות (נכנה אותן בשם "מקטעים"):

- Autobox עבור ארגומנט פרימיטיבי
- supertype of Autobox עבור ארגומנט פרימיטיבי
- Unboxing עבור ארגומנט רפרנס
- Widening of Unboxing עבור ארגומנט רפרנס

התכונה Applicable (יישימה) עבור מתודה

- עבור קריאה נתונה, מתודה תקרא applicable אם קיים מקטע מתוך Strict invocation או Loose invocation המוביל מהארגומנט לפרמטר של המתודה.
- עבור ארגומנט null (hard-coded) , כל חתימה עם פרמטר רפרנס היא applicable .
- לדוגמא, עבור הקריאה *method(3.5);* מתקיים :

```
void method(Number N) {…}  // applicable
void method (float f)   {…} // Not applicable
```

(אין מקטע שמוביל מ double ל float)

תהליך בחירת מתודה קרובה ביותר בהעמסה בת פרמטר אחד

בהינתן קריאה ; *method (arg)*

1) המהדר בודק האם בטווח ההכרה קיימות מתודות applicable עבור הקריאה.

- אם קיימות, הוא ימסור רק אותן לשלב 2.
- אם אין כלל – תהיה שגיאת הידור "not applicable".

2) מבין המתודות ה applicable שנמסרו משלב 1,
המהדר בודק האם יש מבניהן המקיימות STRICT
invocation .

- אם יש כאלה, המהדר ימסור רק אותן לשלב 3.

- אחרת, זה אומר שכל המתודות ה applicable מקיימות
LOOSE invocation , ולכן נמסור אותן לשלב 3.

3) מבין כל המתודות שנמסרו משלב 2, המהדר משווה פרמטרים בין כל שתי מתודות לפי היחס subtype . המטרה: מציאת מתודה שהפרמטר שלה הוא subtype של כל פרמטר בכל מתודה אחרת שנמסרה לשלב זה.

לדוגמא :

Public void method (int i)

Public void method (float f)

מתקיים

int <: float

- אם קיימת בדיקה מסוימת שבה לא ניתן לקבוע יחס subtype בין שני פרמטרים,
  (למשל עם אופרנד פרימיטיבי ואופרנד עצם , ש subtype לא מוגדר במקרה כזה) , אז תהיה שגיאת הידור עם הודעה "ambiguous method" .

- אם לא ניתן לקבוע ביחידות את המתודה הקרובה ביותר (דהיינו לא קיימת מתודה שהפרמטר שלה הוא subtype של כל פרמטר בכל מתודה אחרת שנמסרה לשלב זה) אז תהיה שגיאת הידור "ambiguous method".

- אחרת, תתבצע קריאה עם המתודה הקרובה ביותר.

הערות

- מקרה מיוחד הוא ארגומנט null (לחצו כאן לתיאור בעמ' 39).

- בקריאות המערבות פולימורפיזם (כגון *ad.func(K);* בקוד למטה) , תהליך החיפוש נותן את המתודה הכי קרובה במחלקת האב (או אלו שהיא יורשת מהן) – זו המועמדת להידרס במחלקות שנמצאות במסלול הירושה מהאב למחלקת הבן.
("שיטת הסלאש" המוכרת מההרצאות של חן אולמר).

- **בתוך גוף המתודה שנבחרה, הארגומנט עובר המרה לסוג הפרמטר (ההשפעה היא רק בגוף המתודה)**

  - לדוגמא, אם בקריאה *method (5) ;* **נבחרה המתודה**
  public void method (Integer I) ,
  אז שורה בגוף המתודה כמו *System.out.print(I.getClass());*
  תרוץ בלי בעיה.

- **במידה והארגומנט הוא עצם והפרמטר הוא מחלקת אב שלו, אז במתודה יש לבצע הסבה מפורשת לשם הפעלת מתודות יחודיות לארגומנט. רצוי לבדוק סוג ע"י instanceof לפני כן.**

- להלן מחלקות ולאחריהן סדר הדפסות ב main:

```
public class Animal {
	public void func(int i) {
		System.out.println("in Animal");

	}
}
```

```
public class Dog extends Animal {

	public void func(Integer I) {
		System.out.println("OverLoaded in Dog");
	}

	public void func(int num) {
		System.out.println("OverRidden in Dog");
	}

}
```

```java
public static void main(String[] args) {
    Animal a = new Animal();
    Animal ad = new Dog();
    Dog d = new Dog();

    int i = 5;
    Integer K = 8;

    a.func(0); // prints "in Animal"
    a.func(i); // prints "in Animal"
    a.func(K); // prints "in Animal"

    ad.func(0);// prints "OverRidden in dog"
    ad.func(i);// prints "OverRidden in dog"
    ad.func(K);// prints "OverRidden in dog"

    d.func(0);// prints "OverRidden in dog"
    d.func(i);// prints "OverRidden in dog"
    d.func(K);// prints "OverLoaded in dog"

}
```

- **נדגים את הקריאה** *ad.func(K);* **מהקוד לעיל**

הארגומנט הוא רפרנס.

1. המהדר מחפש במחלקה Animal את כל המתודות (רשומות או בירושה) בשם func שהן applicable עבור ארגומנט Integer.
קיימת רק אחת כזו, לכן היא נמסרת לשלב 2.

2. המתודה הזו היא עם פרמטר int , ולכן היא מהווה Loose invocation עבור הארגומנט. מכוון שזו המתודה היחידה בשלב זה, היא עוברת לשלב 3.

3. לא קיימות לה "מתחרות" , לכן <u>זו המועמדת להידרס בשלב ב' של שיטת הסלאש.</u>

עד כאן היה תהליך בחירת מתודה קרובה ביותר במחלקת האב. כעת נראה את שלב ב' של שיטת הסלאש עבור הדוגמה:

ה JVM מחפשת דריסה של המועמדת במסלול הירושה מ Dog ל Animal, ותיקח את הדריסה הקרובה ביותר לDog . במידה ולא נמצאה כזו , הקריאה תתבצע במתודה שהייתה מועמדת לדריסה.

במקרה הזה, קיימת דריסה טובה ביותר והיא במחלקה Dog - זו המתודה השנייה שכתובה שם.

מכאן, זו המתודה שתיקרא ולכן יודפס - "OverRiden in Dog".

נדגים וננתח קוד נוסף

```
public class Main {

      public static void main(String[] args) {
            short s = 4;
            method(s);
      }

      public static void method(int i) {   // Strict invocation (widening)
            System.out.println("from int");
      }

      public static void method(long l) {    // Strict invocation (widening)

            System.out.println("from long");
      }

      public static void method(Integer I) {  // NOT Applicable
            System.out.println("from Integer");
      }

      public static void method(Number N) { //  Loose invocation (Supertype of Autobox)
            System.out.println("from Number");
      }
}
```

הארגומנט הוא פרימיטיבי.

1. **המהדר מחפש מתודות בשם method שהן applicable עבור ארגומנט short . פרט לשלישית, כולן applicable (אין מקטע ב Invocations אשר מוביל מ short ל Integer).**
   כל ה applicables יימסרו לשלב 2.

2. **מבין אלו שמנסרו לשלב 2, המהדר בודק האם יש מתודות המהוות Strict invocation עבור הקריאה .**

השתיים הראשונות הן כאלה , ולכן אלו ימסרו לשלב 3.

3. כעת מתבצעת השוואת פרמטרים לפי יחס subtype בין שתי המתודות שהועברו לשלב זה. מתקיים int <: long , לכן המתודה הקרובה ביותר היא הראשונה.

מכאן, יודפס "from int".

נדגים וננתח קוד שייכשל בהידור

```
public class Main {

      public static void main(String[] args) {
            byte b = 4;
            method(b);

      }

      public static void method(char c) { // Not Applicable
            System.out.println("from char");
      }

      public static void method(Integer I) { // Not Applicable
            System.out.println("from Integer");
      }

      public static void method(Long L) { // Not Applicable
            System.out.println("from Long");
      }

}
```

הארגומנט הוא פרימיטיבי.

1.המהדר מחפש מתודות בשם method שהן applicable עבור ארגומנט byte. אין כאלו , לכן תהיה שגיאת הידור, עם הודעה: "method not applicable".

נסביר:

- המתודה הראשונה לא applicable כי לא קיים מקטע ב invocations אשר מוביל מ byte ל char .
- המתודה השנייה לא applicable כי לא קיים מקטע ב invocations אשר מוביל מ byte ל Integer .
- המתודה השלישית לא applicable כי לא קיים מקטע ב invocations אשר מוביל מ byte ל Long .

מקרה טריקי : ארגומנט null

- מדובר על **null** כארגומנט **hard-coded** בקריאה כמו *func(null);* , ולא כערך מוצבע ע"י ארגומנט רפרנס.

- רק מתודות עם פרמטר רפרנס הן **Applicable** עבור ארגומנט **null**.

- אם יש שתי מתודות **Applicable** עבור ארגומנט **null** שבין הפרמטרים שלהן לא מתקיים **subtype**, כמו func (Character c) ו func (Integer I) , אז בקריאת *func(null);* תהיה שגיאת הידור **"ambiguous method"**.

- תיבחר המתודה בעלת פרמטר רפרנס שהוא **subtype** של כל הפרמטרים האחרים , מבין המתודות שהן **applicable** כאמור לעיל .

נושא אחרון - העמסה מרובת פרמטרים

התהליך כאן די דומה, עם שינויים מעטים , כי מדובר על מתודות עם הרבה פרמטרים (2 ומעלה).

* שוב, נתייחס רק לארגומנטים פרימיטיביים ,עוטפים , מערכים, ממשקים ועצמים שאינם מחלקות גנריות.

התאמת המושגים Applicable,Strict, Loose להעמסה מרובת פרמטרים

עבור קריאה עם ארגומנטים $(e_1, e_2, \dots , e_n)$:

- מתודה תיחשב Strict invocation אם היא משתמשת באחד או יותר מהמקטעים של Strict (לחצו כאן לתזכורת), ומהם בלבד .
- מתודה תיחשב Loose invocation אם היא משתמשת בלפחות אחד מהמקטעים של Loose (לחצו כאן לתזכורת).
- מתודה תיחשב Applicable אם לכל ארגומנט e_i קיים מקטע מ Loose או Strict המוביל לפרמטר המתאים במתודה. עבור ארגומנט hard coded null , כל פרמטר רפרנס נחשב מתאים.

תהליך בחירת מתודה טובה ביותר בהעמסה מרובת פרמטרים

בהינתן קריאה ; *method (arg1,arg2, … , arg n)*

1) המהדר בודק האם בטווח ההכרה קיימות מתודות **applicable** עבור הקריאה.

- אם קיימות, הוא ימסור רק אותן לשלב 2.
- אם אין כלל – תהיה שגיאת הידור "**not applicable**".

2) מבין המתודות ה applicable שנמסרו משלב 1,
המהדר בודק האם יש מבניהן המקיימות STRICT
invocation .

- אם יש כאלה, המהדר ימסור רק אותן לשלב 3.

- אחרת, זה אומר שכל המתודות ה applicable מקיימות
LOOSE invocation , ולכן נמסור אותן לשלב 3.

3) מבין כל המתודות שנמסרו משלב 2, המהדר בודק פרמטר-פרמטר באותו אינדקס בחתימה בין כל שתי מתודות לפי היחס subtype . המטרה: מציאת מתודה שכל פרמטר שלה הוא subtype של הפרמטר התואם בכל מתודה אחרת בשלב.

לדוגמא :

Public void method (int i , double d ,Double e)

Public void method (float f , double k, Number n)

מתקיים

int <: float , double <: double , Double <: Number

- אם קיימת בדיקה מסוימת שבה לא ניתן לקבוע יחס subtype בין שני פרמטרים באותו אינדקס בחתימה,
(למשל עם אופרנד פרימיטיבי ואופרנד עצם , ש subtype לא מוגדר במקרה כזה) , אז תהיה שגיאת הידור עם הודעה "ambiguous method" .

- אם לא ניתן לקבוע ביחידות את המתודה הקרובה ביותר (דהיינו לא קיימת מתודה שכל פרמטר שלה הוא subtype של פרמטר באינדקס תואם בכל מתודה אחרת) אז תהיה שגיאת הידור "ambiguous method".

- אחרת, תתבצע קריאה עם המתודה הקרובה ביותר.

נדגים קריאה תקינה

```
public static void main(String[] args) {
      Integer J = 8;
      method(J, 2, 3);
}

public static void method(Object o, double d, long l) {// Strict invocation
      System.out.println("from first");
}

public static void method(Number n, int i, long l) {// Strict invocation
      System.out.println("from second");
}

public static void method(Number n, int i, float t) {// Strict invocation
      System.out.println("from third");
}

public static void method(int i, int j, int k) { // Loose invocation
      System.out.println("from fourth");
}

public static void method(float f, float g, int i) { // Loose invocation
      System.out.println("from fifth ");
}

public static void method(Character n, int i, float t) {// Not applicable
      System.out.println("from sixth");
}
```

1) עבור הקריאה , המתודה <u>האחרונה</u> היא **not applicable** – הארגומנט הראשון הוא **Integer** בעוד הפרמטר הראשון במתודה הוא **Character**, ולא קיים מקטע שמוביל **Integer** ל **Character** .

<u>כל שאר המתודות יועברו לשלב 2.</u>

2) עבור הקריאה ,המתודות היחידות שמקיימות **Strict invocation** הן הראשונה, השנייה והשלישית. לכן רק הן יועברו לשלב 3.

** נשים לב, למשל, שהמתודה הרביעית מהווה **loose invocation** כיוון שעבור הארגומנט הראשון נדרש **Unboxing** כדי להעבירו למתודה זו.*

3) במתודה השנייה, כל פרמטר מהווה subtype של הפרמטר התואם לו במתודות הראשונה והשלישית

Param1 : Number <: Object, Number <: Number

Param2 : int <: double , int <: int

Param3 : long <: long , int <: int

ולכן מתודה זו היא הקרובה ביותר והיא זו שתופעל.

מכאן, יודפס "from second".

נדגים קריאה שתיכשל "ambiguous"

```
public static void main(String[] args) {
      method(1, 2);
}

private static void method(Integer i, Integer j) { //loose
      System.out.println("from first");
}

private static void method(int i, Integer j) {//loose
      System.out.println("from second");
}
```

עבור הקריאה, כל המתודות applicable ו loose , לכן כולן עוברות לשלב 3.

בפרמטר הראשון לא מוגדר כלל היחס subtype בין Integer של המתודה הראשונה לבין int של המתודה השנייה. לכן תהיה שגיאת הידור מסוג "ambiguous method".

קריאה נוספת שתיכשל "ambiguous"

```
public static void main(String[] args) {
      method(5, 9);
}

private static void method(double i, int j) { // Strict
      System.out.println("from first");
}

private static void method(int i, double j) {// Strict
      System.out.println("from second");
}
```

עבור הקריאה, כל המתודות applicable ו strict , לכן כולן עוברות לשלב 3.
מכוון ש int <: double , בפרמטר הראשון המתודה השנייה קרובה יותר,
אך בפרמטר השני המתודה הראשונה היא הקרובה יותר. כלומר לא ניתן
לקבוע ביחידות את המתודה הקרובה ביותר ולכן תהיה שגיאת הידור
"ambiguous method" .

קריאה עם ארגומנט null שתיכשל "ambiguous"

```
public static void main(String[] args) {
    method(null, 3, 8);
}

public static void method(Number n, int i, float t) {// Strict invocation
    System.out.println("from first");
}

public static void method(Character n, int i, float t) {// Strict invocation
    System.out.println("from second");
}

public static void method(Object o, Integer d, long l) {// loose invocation
    System.out.println("from third");
}

public static void method(int i, int j, int k) { // NOT Applicable
    System.out.println("from fourth");
}
```

1) עבור הקריאה, המתודה האחרונה היא not applicable , כי ניתן להעביר null רק לפרמטר רפרנס, בעוד הפרמטר הראשון של המתודה הנ"ל הוא int . לכן שאר המתודות עוברות לשלב 2.

2) עבור הקריאה , שתי המתודות הראשונות הן Strict , והמתודה השלישית היא Loose בגלל שנדרש Autoboxing בפרמטר השני. לכן רק שתי המתודות הראשונות עוברות לשלב 3.

3) מכוון שלא ניתן לקבוע יחס subtype עבור הפרמטר הראשון בין שתי המתודות (הראשונה והשנייה), אז תהיה שגיאת הידור "ambiguous method".

סיכום (מומלץ להוסיף בעמ' 134-135 במדריך)

משתנה פרימיטיבי	ליטרל אות	ליטרל בוליאני	ליטרל שלם (ללא אות מוצמדת)	ליטרל דצימלי (ללא אות מוצמדת)	ליטרל שלם כפוי long	ליטרל מספרי כפוי float
הצבעה עליו ע"י פרימיטיבי אחר מותרת לפי סדר ההכלה (זה widening) - המצביע צריך להכיל את טווח הערכים של המשתנה המוצבע. אחרת, -> שגיאת הידור. **יכול להצביע** על העוטף שלו (זה unboxing), או על ליטרל בהתאם לחוקים משמאל.	נחשב char . אותן הגבלות כמו משתנה char, **אך בנוסף מותרת הצבעה עליו ע"י Number , Byte , Short , byte ,short** **כל עוד ערך הUnicode של האות נמצא בטווח הערכים של המצביע.**	נחשב boolean. אותן הגבלות כמו משתנה boolean. **לא משתתף ב widening .** לכן בסה"כ ניתן להצביע עליו רק באמצעות Boolean, boolean, Object.	נחשב int. אותן הגבלות כמו משתנה int, **אך בנוסף מותרת הצבעה עליו ע"י Byte, Short, Character , byte , short, char** **כל עוד המספר נמצא בטווח הערכים של המצביע.**	נחשב double. אותן הגבלות כמו משתנה double.	נחשב long . מסומן עם האות L (קטנה/גדולה) מימין למספר. אותן הגבלות כמו משתנה long.	נחשב float. מסומן עם האות F (קטנה/גדולה) מימין למספר. אותן הגבלות כמו משתנה float .
הצבעה עליו ע"י רפרנס, מותרת עם העוטף שלו (זה Autoboxing) או מחלקות-על של העוטף (זה שילוב Autoboxing עם פולימורפיזם). אחרת, -> שגיאת הידור.						

סדר ההכלה עבור הרחבה (= subtype עבור פרימטיביים)

byte ⊆ short ⊆ int ⊆ long ⊆ float ⊆ double

char ⊆

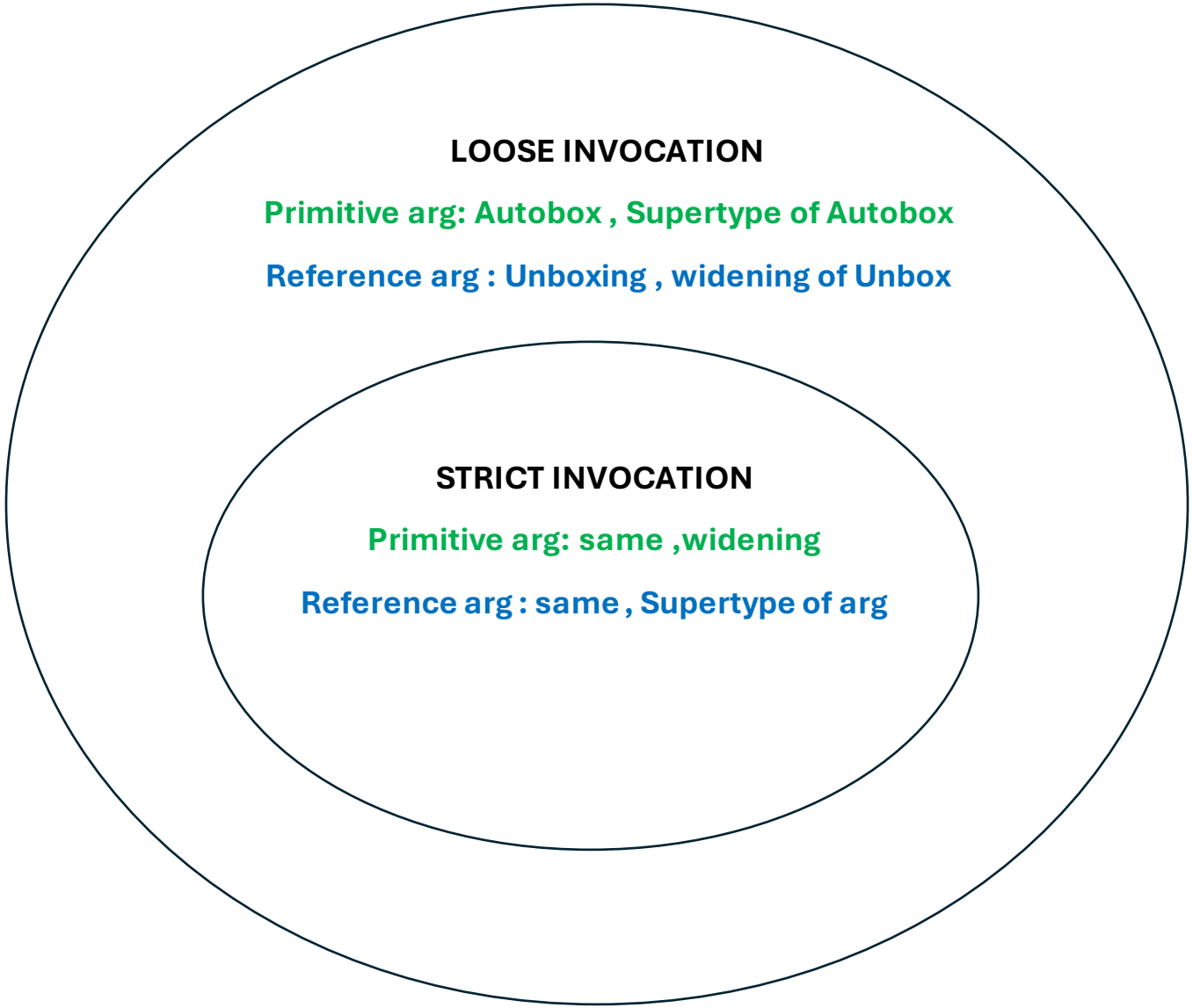
- ניסיון לדרוס מתודה ע"י שינוי הטיפוס המוחזר מפרימיטיבי לעוטף שלו או מעוטף לפרימיטיבי שלו יגרום לשגיאת הידור.

- שינוי פרמטרים של מתודה מעוטף לפרימיטיבי שלו או מפרימיטיבי לעוטף שלו מהווה העמסה חוקית.

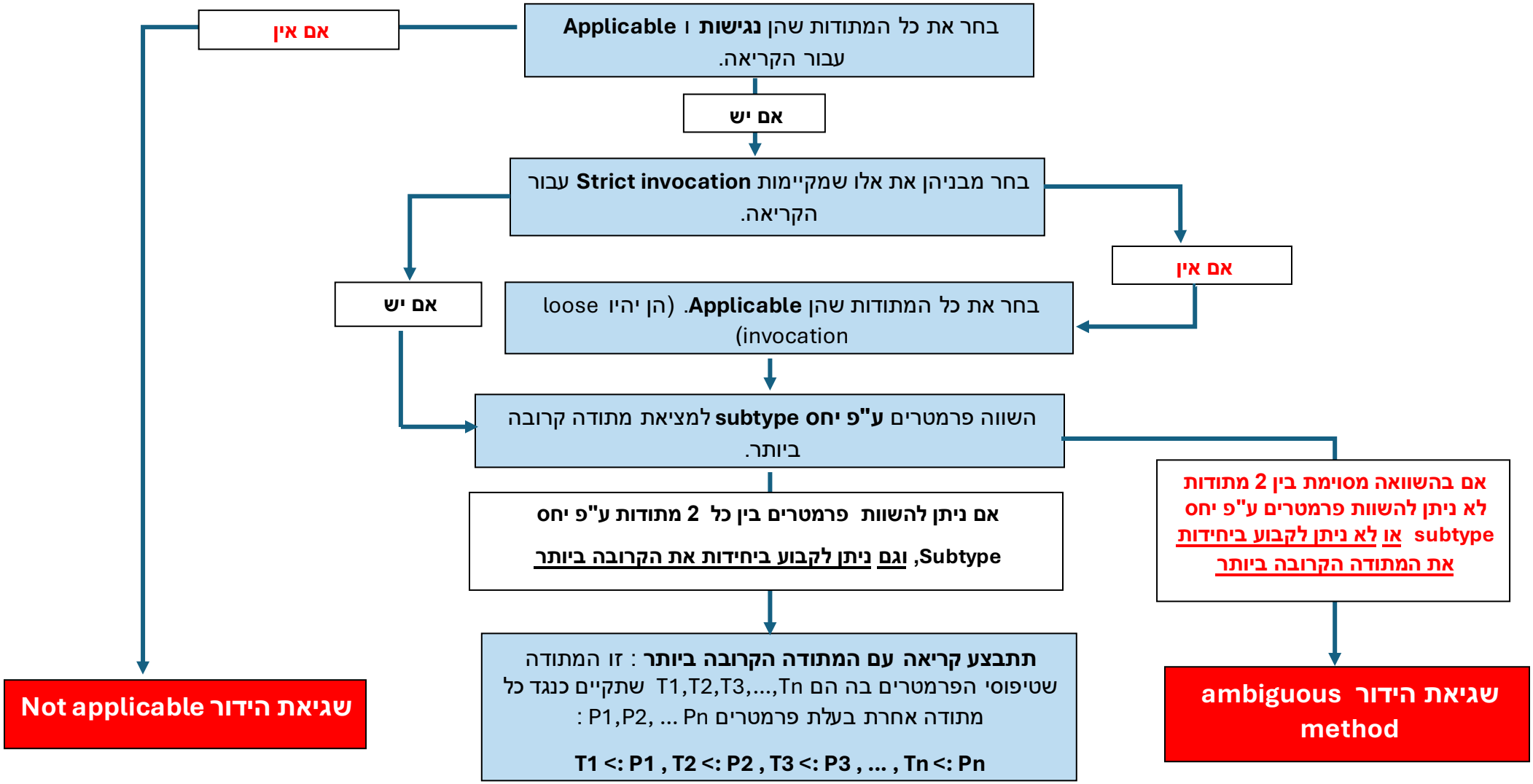
- היחס subtype , לחצו כאן

- מקרה טריקי של ארגומנט null – לחצו כאן.

Strict and Loose Invocations



קביעת מתודה קרובה ביותר מבין העמסות בעלות שם זהה



מקורות

Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2015). *The Java Language Specification Java SE 8 Edition*. Oracle.

נכתב ע"י נדב קופיט

https://www.linkedin.com/in/nadav-kopit

עדכון אחרון: 24 ספטמבר 2024