שפת Java למתכנתי C++ - חלק ראשון

ניר אדר

מסמך זה הורד מהאתר www.underwar.co.il.

אין להפיץ מסמך זה במדיה כלשהי, ללא אישור מפורש מאת המחבר.
מחבר המסמך איננו אחראי לכל נזק, ישיר או עקיף, שיגרם עקב השימוש במידע המופיע במסמך, וכן לנכונות התוכן של הנושאים המופיעים במסמך. עם זאת, המחבר עשה את מירב המאמצים כדי לספק את המידע המדויק והמלא ביותר.

כל הזכויות שמורות ל**ניר אדר**

Nir Adar
Email: **nir@underwar.co.il**

אנא שלחו תיקונים והערות אל המחבר.

1. תוכן עניינים

2. מבוא

2.1. רקע

שפת Java היא שפת תכנות חדשה יחסית, מ-1995. שפה זו הינה שפת תכנות מונחית עצמים שירשה אלמנטים רבים מ-C++.
התחביר של Java מבוסס C++ אם כי ישנם מספר הבדלים משמעותיים.

נפרט חלק מהיתרונות של Java על C++:

- פשטות – Java ויתרה על כמה מהמנגנונים המסובכים של C++, כגון unions ומצביעים.
- תכנות מונחה עצמים – ב-Java כל דבר הוא אובייקט.
- רובסטית – לא ניתן להשתמש במשתנים שלא אותחלו או להצביע אל אובייקט שלא אותחל. אין אפשרות לחרוג מגבולותיו של מערך. ניסיון לחרוג מגבולותיו של מערך ב-Java יגרום לשגיאה (Exception).
- בטוחה – סביבת הריצה של Java מכילה מנגנון code verifier הבודק שהקוד של המשתמש חוקי ומתנהג לפי ההרשאות שניתנו לו. בנוסף, בשפת Java לא קיימים מצביעים, וזאת הן על מנת למנוע באגים בתוכניות המשתמש, והן כדי להגביר את האבטחה של השפה. הורדת המצביעים גורמת לכך שמתכנתים מתוחכמים לא יוכלו לעקוף את מנגנון האבטחה של השפה.
- Java - Garbage Collector מאפשרת למשתמש להקצות זיכרון דינאמי, אולם שחרור הזיכרון איננו באחריותו של המשתמש. Java משתמשת ב-Garbage Collector, הפועל כל העת ברקע שהוא אחראי לאתר את הזיכרון שאיננו בשימוש עוד ולשחרר אותו.
- ב-Java ניתן לבצע הורשה מאב יחיד בלבד. האפשרות לבצע הורשה מרובה ב-C++ גרמה לעתים לתקלות קשות. תכנון נכון של המחלקות מתממש תמיד בהורשה יחידה בלבד.
- מכילה מספר רב של מחלקות סטנדרטיות הכוללות תמיכה בקלט/פלט, עבודה מול מסדי נתונים, מבני נתונים שונים, תמיכה בריבוי תהליכים ועוד.

2.2. Java Virtual Machine

אחד האלמנטים שהופכים את Java לשפה חשובה הוא ה-Java Virtual Machine (JVM).
הרעיון: כאשר אנחנו מהדרים קוד Java, המהדר איננו יוצר קוד בשפת מכונה, אלא יוצר קוד ביניים, בשפה המכונה byte code. רגע לפני שהתוכנית שאנחנו כותבים רצה על מחשב היעד, היא עוברת קומפילציה נוספת כדי להתאים לשפת המכונה של אותו מעבד – native code.
בצורה כזו – תוכנית שנכתבה ב-Java יכולה לרוץ על כל מערכת הפעלה שתומכת ב-Java ללא שינויים ואף ללא צורך בהידור מחדש.

2.3. תוכנית ראשונה

התוכנית הבאה מציגה את המילים Hello, World! על מסך המשתמש.

Hello.Java

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

כמה דברים שניתן לראות כבר מדוגמא זו:
ב-Java אין פונקציות שאינן שייכות למחלקות. כל פונקציה שייכת למחלקה.
ריצת התוכנית מתחילה מפונקציה סטטית בשם main. בדומה למחלקות בשפת C++, פונקציה סטטית היא פונקציה השייכת למחלקה ולא לאובייקט של המחלקה.
כמו כן, שם הקובץ בו נשמרת המחלקה חייב להיות זהה לשם המחלקה. בכל קובץ נשמרת מחלקה אחת בלבד (אם כי יתכן שיהיו לה תתי מחלקות בתוכה).

הידור הקובץ נעשה על ידי התוכנה Javac (Java Compiler) בצורה הבאה:

C:\> **Javac Hello.Java**

לאחר הידור מוצלח, הרצת התוכנית תעשה כך:

C:\> **Java Hello**

יש לשים לב שאין צורך להוסיף את הסיומת class. כשמפעילים את התוכנית. הוספת הסיומת class.

תחשב אף לשגיאה, ונקבל הודעה דוגמת ההודעה הבאה:

C:\> **Java Hello.Java**

Exception in thread "main" Java.lang.NoClassDefFoundError: Hello/Java

2.4. אלמנטים הקיימים ב-C++ שאינם קיימים ב-Java

נציג כעת רשימה של כל התכונות הקיימות בשפת C++ ואינן קיימות ב-Java, לצורך השוואה מלאה.

- הנחיות אל ה-preprocessor (כגון #define וכדו').
- משפטי goto
- מבנים כגון struct, union ו-enum.
- Typedef
- מצביעים – בכל מקום בו היינו משתמשים במצביעים נשתמש כעת ב-references.
- חפיפת אופרטורים.
- ירושה מרובה ממחלקות (אם כי קיים מנגנון חלופי – ירושה מרובה מממשקים).
- קבצי header – קיימים רק קבצי java ולא קיימים קבצי h.
- תבניות (templates) – כרגע לא קיימות בשפה, אם כי מתוכנן לשלב אותם באחת הגרסאות הבאות שלה.

3. ארגון המחלקות ב-packages

3.1. משפט package

כאשר בתוכנית שלנו מספר מחלקות רב נרצה לחלק אותן ל-"חבילות" לפי נושאים משותפים.
נוכל ב-Java להגדיר חבילות של מחלקות שמכונות packages.

על מנת להגדיר שמחלקה או מחלקות הנמצאים בקובץ מסוים שייכים ל-package, נכתוב בתחילת הקובץ את ההצהרה:

```
package <package name>;
```

כאשר <package name> זהו שם החבילה הרצוי.

לדוגמא:

```
// File MyClass.Java

package MyFirstPackage;

class MyClass
{


}
```

בהוספת שורה זו, המחלקה (או המחלקות אם יש יותר ממחלקה אחת בקובץ) יהיו שייכים ל-package ששמו MyFirstPackage.

אם לא מציינים באמצעות משפט ה-package כי המחלקה/ות בקובץ מסוים שייכות ל-package מסוים אז המחלקות יהיו שייכות ל-default package.

כל קבצי ה-class של מחלקות ששייכות ל- package מסוים יישמרו בספריה ששמה זהה לשם של ה-package.
מבחינת שמות, נהוג לתת ל-package שם באותיות קטנות.

לאחר כתיבת קובץ קוד המקור שמכיל מחלקות ששייכות ל-package מסוים יש להדר אותו כך שקובצי ה-class שייווצרו יישמרו בספריה ששמה זהה לשם של ה-package.
כדי לעשות זאת יש להדר באמצעות הוספת התגית d- לפקודה Javac. כמו כן, יש לרשום אחרי תגית זו הספרייה שתחתיה תיווצר ספריית קבצי ה-package.
לדוגמא:

C:\> **Javac -d C:\ MyClass.Java**

כתיבת שורה זו בשורת הפקודה תגרום לכך שקובץ ה-class שייווצר ישמר בספריה ששמה MyFirstPackage, והספרייה הזו תיווצר תחת הספרייה C:\.

ניתן לייצור היררכיה של packages. כלומר, package אשר יכיל packages אחרים.
היררכיה של packages תבוא לידי ביטוי בהיררכיה מתאימה של ספריות.

כאשר יוצרים מחלקה ששייכת ל-package מסוים, שמה איננו עוד השם שנתנו לה בשורת הגדרתה בקוד המקור. כעת שמה המלא הוא השם שניתן בשורת הגדרתה בצירוף שם ה-package שאליו היא שייכת או בצירוף שמות ה-packages אם ה-package שאליו המחלקה שייכת שייך ל-package אחד. אם יש יותר מ-package אחד שצריך לציין את שמו, השמות יצוינו עם נקודות מפרידות ביניהן.

3.2. משפט import

במידה ואנחנו משתמשים פעמים רבות במחלקה מסוימת בקוד שלנו, לא נרצה לכתוב כל הזמן את שמה המלא. על מנת לעזור לנו מגיע משפט ה-import.
התחביר:

```
import <package name>;
```

ניתן גם להשתמש בסימן * כדי לצרף קבוצה של packages.
במידה וצרפנו package מסוים, נוכל להשתמש בכל המחלקות שבו בשמן הקצר, ללא צורך לכתוב את שם ה-package.

3.3. הערות

- מחלקות השייכות לאותו package יכולות לגשת אחת אל השנייה ללא ציון שמה המלא של המחלקה השנייה, וללא משפט import.
- ה-package ששמו Java.lang מיובא באופן אוטומטי לכל קובץ שאנו כותבים ב-Java, ולכן אין צורך להוסיף לתחילת הקובץ.
- אם הקובץ מכיל גם משפט package וגם משפטי import, אז ה-package יופיע לפני ה-import.

4. יסודות שפת Java

4.1. טיפוסים בסיסיים

הטבלה הבאה מסכמת את הטיפוסים הבסיסיים של שפת Java. נשים לב שבניגוד לשפות כמו C/C++ שבהם גודל כל טיפוס לא הוגדר באופן מספרי (הוגדרו היחסים בין המשתנים, אולם לא אמרנו במפורש כי int, למשל, הוא בן 4 בתים תמיד), Java מגדירה גודל קבוע ויחיד עבור הטיפוסים שלה.
כמו כן ב-Java לא קיימת המילה unsigned כפי שהיא קיימת ב-C/C++. כל טיפוסי המספרים השלמים יכולים לקבל מספרים חיוביים ושליליים ולא ניתן להגביל טיפוס לקבל רק מספרים אי שליליים

Type	Size	Minimum	Maximum	Literals	Default*
boolean	-	-	-	true, false	False
char	16-bit	Unicode 0	Unicode $2^{16}-1$	'x'	'\u0000'
byte	8-bit	-128	127	(byte)1	(byte)0
short	16-bit	-2^{15}	$2^{15}-1$	(short)1	(short)0
int	32-bit	-2^{31}	$2^{31}-1$	1, 0754, 0xfe	0
long	64-bit	-2^{63}	$2^{63}-1$	1L	0L
float	32-bit	IEEE754	IEEE754	1.2f	0.0f
double	64-bit	IEEE754	IEEE754	1.2	0.0d
void	-	-	-	-	-

* - ערכי ברירת המחדל הינם רק עבור members variables.

4.2. Wrappers

על מנת לבצע פעולות שונות על הטיפוסים היסודיים, Java מספקת עבור כל אחד מהטיפוסים הבסיסיים מחלקה שלמעשה עוטפת אותו, ומאפשרת להפעיל עליו מתודות שונות.
לדוגמא, המחלקה Integer מיועדת כדי לעטוף משתנים מסוג int:

```
Integer n = new Integer("4");
int m = n.intValue();
```

4.3. Case Sensitivity

Java היא שפה Case Sensitive: המזהה main שונה מהמזהה Main.
כל המילים השמורות של השפה נכתבות בעזרת אותיות קטנות בלבד.

4.4. קונבנציית שמות ב-Java

קונבנציית שמות ב-Java היא כלהלן:

- שיטות, משתנים ואובייקטים מתחילים באות אנגלית קטנה.
- שמות של מחלקות מתחילות באות אנגלית גדולה.
- משתנים קבועים (final) נכתבים כשכל המילה נכתבת באותיות גדולות.
- הפרדת מילים בשמות נעשית על ידי אות גדולה, למשל getPrice(), מלבד בקבועים, עבורם ההפרדה נעשית על ידי קו תחתון.

4.5. הערות

Java מספקת למתכנת מספר סוגי הערות.
ניתן להשתמש בהערות בסגנון C++:

```
// blah blah
/* this is a blah blah */
```

וכמו כן ישנן הערות מיוחדות, המיועדות עבור מנגנון התיעוד של Java, ששמו Javadoc.
הערות אלו נכתבות בצורה הבאה:

```
/** comment */
```

4.6. אופרטורים

האופרטורים בשפת Java כמעט זהים לאלו ב-C++.
רשימה חלקית של האופרטורים, להלן:

1. אופרטורים פשוטים של מתמטיקה : , %(), +, *, /, -
2. אופרטורים של השמה : =, =+, = -, =*
3. אופרטורים של השוואה : =!, ==, =>, =<, >, <
4. אופרטורים לוגיים: !, &&, ||
5. ++, --

הבדל בין Java ל-C הוא שבניגוד ל-C, ב-Java חייבים לבצע casting מכוון כאשר מבצעים אל תוך משתנה השמה של ערך מטיפוס שהפיכתו לטיפוס המשתנה עלולה לגרום לאיבוד חלקים מערכו.

האופרטורים שקיימים ב- C/C++ ולא קיימים ב-Java הם:

sizeof אופרטור שמשמש ב-C למציאת מספר הבתים שמשמשים טיפוס/משתנה.

***** הערך שנמצא בכתובת מסוימת.

& קבלת הכתובת של משתנה/אובייקט.

:: שיוך פונקציה או משתנה סטטי למחלקה.

, אופרטור זה משמש לביצוע הפרדה בין ביטויים. אופרטור הפסיק לא קיים ב-Java - למעט בתוך הביטויים הראשון והשלישי שמופיעים בתוך במשפט לולאת ה-for.

4.7. מבני בקרה

כל מבני הבקרה הבסיסיים של Java זהים לאלו של C++. נציג את התחביר שלהם ללא הסברים נוספים:

if/else

```
if (expression)
{
      // block
}
else
{
      // alternate block
}
```

do/while

```
do
{
      // block
}
while (expression);
```

for loop

```
for (initalize; condition; step)
{
      // block
}
```

switch... case

```
switch(indifier)
{
      case VALUE1:
      case VALUE2:
            // act 1
            break;
      default:
            // act2
            break;
}
```

4.8. מחלקות

שפת Java מבוססת מסביב למחלקות. מחלקות הינן ליבו של התכנות מונחה העצמים.
מחלקה מכילה פונקציות חברות ומשתנים חברים.
גישה אל שדות במחלקות נעשית ב-Java על ידי האופרטור . (נקודה).

4.8.1. הרשאות

ב-Java קיימים אותם סוגי הרשאות כמו בשפת C++: private, public, protected.
פונקציה או משתנה המוגדרים public, ניתנים לגישה מכל מחלקה.
פונקציה או משתנה המוגדרים private, ניתנים לגישה מפונקציות של אותה המחלקה בלבד.
פונקציה או משתנה המוגדרים protected, ניתנים לגישה מפונקציות של אותה המחלקה בלבד ומהמחלקות היורשות ממנה.
ב-Java רישום סוג ההרשאה הוא לפני כל פונקציה או משתנה, ומתקיים רק לגביו, וזאת בניגוד ל-C++ שם היה ניתן לרשום את סוג ההרשאה ועד להודעה חדשה הוא התקיים לגבי כל המשתנים והפונקציות שהוגדרו אחריו.
בנוסף להרשאות אלו, ישנה הרשאה נוספת – package friendly. ומשמעה: במידה ולא ציינו במפורש את ההרשאה של משתנה או פונקציה הוא יקבל הרשאה זו. לכל האובייקטים השייכים לאותה חבילה (package) תהיה גישה חופשית למשתנה זה, ומחוץ ל-package הגישה אליו תהיה חסומה.

4.8.2. פונקציות בונות

פונקציה בונה היא פונקציה הנקראת ברגע שאובייקט חדש נוצר, ותפקידה הוא אתחול האובייקט.
שם הפונקציה הוא כשם המחלקה, והיא איננה מחזירה אף משתנה.
פונקציה בונה יכולה לקבל פרמטרים או לא. כמו כן, ניתן להגדיר פונקציות בונות חופפות.

4.8.3. Default Constructor

אם לא מגדירים עבור מחלקה כלשהי ב-Java פונקציה בונה, הקומפיילר מספק פונקציה בונה משלו כברירת מחדל. פונקציה זו, איננה מקבלת פרמטרים, ומאפסת את כל הערכים לאפס, את כל המשתנים הבוליאניים ל-false ואת כל האובייקטים ל-null.
אם מוגדרת פונקציה בונה כלשהי במחלקה, פונקצית ברירת המחדל לא נקראת.
נשים לב להבדל בין פונקציה זו לפונקצית ברירת המחדל ב-C++, שאיננה מאתחלת את המשתנים במחלקה.

4.8.4. משתני התייחסות

כדי לייצור אובייקט חדש ממחלקה יש לבצע instantiation מאותה מחלקה, כלומר לציין במפורש שאנו רוצים ליצור אובייקט חדש, וזאת על ידי המילה new.

לאחר יצירתו של instance חדש ממחלקה ניתן לאחסן את כתובתו (ה-reference שלו) במשתנה מטיפוס אותה מחלקה. מספר משתני התייחסות יכולים להצביע אל אותו אובייקט.

ב-Java כמעט כל המשתנים הם משתנים מסוג התייחסות. כאשר אנו מעבירים מחלקה לפונקציה, מועברת תמיד התייחסות אל המחלקה, ולא עותק שלה. רק המשתנים הבסיסיים (int, char וכדו') מועברים לפי ערך.

הדרך ליצירתו של משתנה מטיפוס מחלקה זהה לדרך ליצירת משתנה מטיפוס בסיסי.
יש לרשום את שם המחלקה שמטיפוסה רוצים לייצור את המשתנה, ואחריו את שם המשתנה שרוצים לייצור. לדוגמא:

```
Box myBox;
```

בדוגמא זו נוצר משתנה מטיפוס המחלקה Box.

במשתנה שהוא מטיפוס מחלקה מסוימת ניתן לאחסן reference של אובייקט מטיפוס המחלקה שמטיפוסה נוצר המשתנה (ניתן גם לאחסן reference לאובייקט מטיפוס מחלקה שיורשת מהמחלקה שמטיפוסה נוצר המשתנה).
יש לשים לב להבדל לעומת C++: ב-C++ משתנה מטיפוס מחלקה היווה כבר אובייקט בפני עצמו. ב-Java הוא איננו אובייקט. הוא רק מכיל reference של אובייקט.

ניתן בשורת ההצהרה על המשתנה לשלב גם את יצירתו של אובייקט והכנסת כתובתו אל תוך המשתנה. לדוגמא:

```
Box myBox = new Box();
```

משתנה מטיפוס מחלקה יכול להכיל reference לאובייקט או null.

נדגיש את ההבדל שבין משתנה מטיפוס מחלקתי למשתנה מטיפוס בסיסי. בעוד שהאחרון מכיל ערך, הראשון מכיל את ה-reference לערך. אלה הם שני סוגי המשתנים שקיימים ב-Java.

ניתן לבצע השמה ממשתנה מטיפוס Class-Type אחד אל משתנה אחר מטיפוס אותו Class-Type. במקרה כזה, מה שיוכנס אל תוך המשתנה האחר הוא הכתובת שאוחסנה בראשון, ובדרך זו שניהם יצביעו למעשה על אותו אובייקט. לדוגמא:

```
Car familyCar, sportCar;
familyCar = new Car();
sportCar = familyCar;
```

לאחר פקודות אלה המשתנים familyCar ו- sportCar יכילו את אותו reference. כלומר, שינויים שיבוצעו באובייקט ש- familyCar מכיל את ה-reference שלו, יבואו לידי ביטוי גם באובייקט ש-sportCar מכיל את ה-reference שלו כיוון שמדובר, למעשה, באותו reference.

4.8.5. השוואת אובייקטים

האופרטור == המוגדר ב-Java מחזיר אמת אם שני האופרנדים המושווים הם בעלי אותו ערך.
אופרטור זה עובד כראוי עבור סוגי המשתנים הבסיסיים, אולם במקרה של מחלקות הוא משווה בין משתני ההתייחסות בלבד, ולא בין האובייקטים שהם מצביעים אליהם. הדוגמא הבאה תמחיש את הנושא.
נניח כי כתבנו את שורות הקוד הבאות:

```
Integer i1 = new Integer("3");
Integer i2 = new Integer("3");
Integer i3 = i2;
```

אזי הביטוי הבא יתפרש לערך אמת:

```
i1 == i1 && i1 != i2 && i2 == i3
```

נשים לב שזו לא תמיד ההתנהגות הרצויה.
כדי להשוות בין אובייקטים נשתמש בפונקציה boolean equals(Object o), בצורה הבאה למשל:

```
i1.equals(i1) && i1.equals(i2)
```

ביטוי זה יתפרש לערך אמת כמצופה.

כל מחלקה שמגדירים ב-Java יורשת באופן אוטומטי את המשתנים והמתודות שהוגדרו במחלקה Object. אחת השיטות שמוגדרות במחלקה Object ומועברת בהורשה אל כל מחלקה שנגדיר היא השיט equals. באמצעות השיטה equals ניתן להשוות בין אובייקט נתון לאובייקט אחר. השיטה משווה את כתובות שני האובייקטים ומחזירה true רק אם הן זהות. פעולתה זהה למעשה לפעולתו של אופרטור ההשוואה: ==, אשר גם כן משווה בין שתי הכתובות. אם זאת, רוב המחלקות בשפת Java, וגם המחלקות שאנחנו נכתוב יצרכו לעשות זאת, חופפות את שיטה זו, על מנת שתחזיר ערך משמעותי, למשל, תשווה בין הערכים המוצבעים על ידי משתני ההתייחסות.

דוגמא לחפיפת פונקציה זו:

```
public class Name
{
	String firstName;
	String lastName;
	...
	public boolean equals(Object o)
	{
		if (!(o instanceof Name)) return false;
		Name n = (Name)o;
		return firstName.equals(n.firstName) &&
			lastName.equals(n.lastName);
	}
}
```

4.8.6. מערכים

מערכים ב-Java הם אובייקטים, כלומר הם מוקצים על הערימה ולא על המחסנית.

גודל מערך הינו קבוע ואיננו יכול להשתנות מרגע שהוגדר. גודל המערך ניתן לנו על ידי השדה הקבוע length הנקבע בעת יצירת המערך.

דוגמא לאתחול מערך של אובייקטים:

```
Animal[] arr; // nothing yet …

arr = new Animal[4]; // only array of pointers

for(int i=0 ; i < arr.length ; i++)
	arr[i] = new Animal();

// now we have a complete array
```

יש ב-Java הבדל בין אתחול מערך של משתנים בסיסיים לבין מערך המכיל אובייקטים של מחלקות.

לולאת ה-for שהצגנו תידרש רק במקרה השני.

איברי המערך של טיפוס נתונים פשוט מאותחלים תמיד ל-0 במקרה של char או מספר, ו-false עבור משתנה מסוג boolean. איברי מערך מסוג אובייקטים מאותחלים ל-null עם יצירת המערך.

העתקת מערכים ב-Java איננה יכולה להתבצע בעזרת האופרטור =, כפי שהקוד הבא מנסה לעשות:

```
int vec1[] = {1,2,3};
int vec2[] = {8,7,6,5};
vec1 = vec2; // Won't copy the array!
```

וזאת מכיוון ששורות אלו ייגרמו ששני המשתנים יצביעו אל אותו מערך, ולא להעתקת תוכנו של המערך.

העתקת תוכן של מערך נעשית על ידי פונקציה שבאה כחלק מ-Java, והיא arraycopy.

השיטה arraycopy מוגדרת כשיטה סטטית במחלקה System:

```
public static void arraycopy(Object src,
                             int src_position,
                             Object dst,
                             int dst_position,
                             int length);
```

דוגמא לשימוש בפונקציה:

```
int []vec1 = {1,2,3,4,5};
int []vec2 = new int[vec1.length];
System.arraycopy(vec1, 0, vec2, 0, vec1.length);
```

4.8.7. מחרוזות

מחרוזות ב-Java הן גם אובייקטים. אם נכתוב, למשל:

```
String s = "Hello, World";
```

ראשית תוקצה מחרוזת חדשה, ותוכנה ייקבע ל-"Hello, World", ולאחר מכן s יצביע אל המחרוזת שיצרנו. ב-Java אין מחרוזות קבועות, בניגוד לשפת C/C++.

מן הרגע שבו נוצר אובייקט חדש מטיפוס String לא ניתן לשנות את המחרוזת שהוא מייצג. יש לשים לב, שעם זאת, עדיין ניתן לשנות את ה-reference שמוחזק במשתנה שמצביע על אובייקט ה-String כך שיצביע על אובייקט אחר מטיפוס המחלקה String (אובייקט אשר ייצג מחרוזת אחרת).

4.8.8. פונקציות ומשתנים סטטיים

משתנים סטטיים שמוגדרים בתוך מחלקה נקראים גם משתנים מחלקה (class variables). אלה הם משתנים שנוצרים פעם אחת בלבד, והם באים לתאר את המחלקה כולה או משהו שמשותף לכל האובייקטים.
הגדרת משתנה סטטי נעשית על ידי הוספת המילה static להגדרה שלו.
ניתן לגשת אל משתנים סטטיים גם ללא יצירת אובייקט מהמחלקה, על ידי פנייה אליהם דרך שם המחלקה. (בדומה ל-C++, ניתן ליצור אובייקטים של המחלקה ולגשת אליהם גם דרך האובייקטים).

פונקציות סטטיות הן פונקציות ששייכות למחלקה ולא לאובייקט.
פונקציות סטטיות מסוגלות לגשת רק אל משתני המחלקה הסטטיים, ובדומה למשתנים הסטטיים, גם להן ניתן לקרוא ללא יצירת אובייקט מן המחלקה.

4.9. הורשה

כפי שכבר צוין, בשפת Java ניתן לרשת רק ממחלקה אחת, ולא ממחלקות מרובות.

דוגמא להורשה בשפת Java:

```
class Base
{
      Base(){}
      Base(int i) {}
      protected void foo() {…}
}

class Derived extends Base
{
      Derived() {}
      protected void foo() {…}
      Derived(int i)
      {
            super(i);
            …
            super.foo();
      }
}
```

דוגמא נוספת, מעשית יותר:

```
class Base
{
      void foo()
      {
            System.out.println("Base");
      }
}

class Derived extends Base
{
      void foo()
      {
            System.out.println("Derived");
      }
}

public class Test
{
      public static void main(String[] args)
      {
            Base b = new Derived();
          b.foo();  // Derived.foo() will be activated
      }
}
```

נשים לב לעובדה חשובה לגבי Java: ב-Java כל הפונקציות מתנהגות כמו פונקציות וירטואליות של שפת C++. זהו מנגנון נוסף שנועד למנוע באגים שהיו נוצרים בשפת C++.

5.9.1. Object

בשפת Java, כפי שכבר צוין, כל המחלקות נגזרות מהטיפוס Object.
מחלקה זו כוללת את השיטות הבאות, ועוד:

```
boolean equals(Object o);
Object clone();
int hashCode();
String toString();
```

מחלקה זו משמשת כמכנה המשותף הנמוך לכל המחלקות של השפה.

5.9.2. פונקציות ומחלקות מופשטות

פונקציה מופשטת היא פונקציה שאין לה מימוש במחלקת הבסיס, ועל מחלקות הבנים לממש פונקציה זו.
מחלקה מופשטת היא מחלקה שלא ניתן ליצור ממנה אובייקטים. מחלקה שיש לה פונקציה מופשטת אחת או יותר חייבת להיות מוגדרת כמחלקה מופשטת.
פונקציה מופשטת היא למעשה הפונקציה הוירטואלית הטהורה של שפת C++.
דוגמא:

```
public abstract class Shape
{
      public abstract void draw();

      public void move(int x, int y)
      {
            setColor(BackGroundColor);
            draw();
            setCenter(x,y);
            setColor(ForeGroundColor);
```

```
            draw();
      }
}


public class Circle extends Shape
{
      public void draw()
      {
            // draw the circle ...
      }
}
```

5.9.3. סיום חייו של האובייקט - Finalization

במחלקה Object מוגדרת השיטה finalize(). שיטה זו מופעלת באופן אוטומטי רגע לפני שהזיכרון של האובייקט משתחרר. כברירת מחדל פונקציה זו איננה עושה כלום, אולם ניתן לחפוף אותה כך שתבצע פעולות מיוחדות של הרגע האחרון. התפקיד שמיועד בדרך כלל לשיטה זו הוא שחרור משאבי מערכת: סגירת קבצים, סגירת ערוצי תקשורת וכו'. יש לציין כי חשיבותה פחותה מחשיבות ה-Destructor של שפת C++, מכיוון שאיננו יודעים בדיוק מתי היא תקרא – ה-Garbage Collector הוא האחראי על ניהול הזיכרון הדינאמי, וכן מכיוון שב-C++ רוב הפעולות שבוצעו בסיום היו פעולות שחרור זיכרון, שלא קיימות ב-Java.

ביצירתה של המתודה finalize יש מספר כללים:

1. יש לתת לה את השם finalize().
2. אסור שיהיו לה פרמטרים.
3. עליה להחזיר void.
4. עליה להיות עם הרשאת הגישה protected.

תנאים אלה נדרשים כדי שהיא תבוא במקום המתודה finalize() אשר מועברת בהורשה מהמחלקה Object.

4.10. ממשקים

ממשק הוא למעשה הגדרה של "צורת תקשורת" בין מחלקות.

ממשק דומה בהצהרתו למחלקה, אך הוא כולל:

- פונקציות ציבוריות (public) בלבד
- משתנים המוגדרים כ-public, static ו-final.

מחלקה יכולה לרשת מממשק, וכן היא יכולה לרשת ממשקים מרובים. דוגמא:

```
interface IChef
{
      void cook(Food);
}

interface Singer
{
      void sing(Song);
}

interface SouthParkCharacter
{
      void curse();
}

class Chef implements IChef, SouthParkCharacter
{
      public void curse() { … }
      public void cook(Food f) { … }
}
```

4.11. final

המילה השמורה final יכולה להירשם לצד מחלקה, שיטה או משתנה.

מחלקה המוגדרת כ-final היא מחלקה ממנה אי אפשר לרשת מחלקות חדשות.

פונקציה המוגדרת כ-final היא פונקציה אותה לא יהיה ניתן לחפוף במחלקות שייגזרו ממחלקה זו.

משתנה המוגדר כ-final הוא למעשה קבוע. הוא מקבל ערך עם הגדרת המשתנה ונשאר איתו.

5. Javadoc

5.1 מבוא

אחד הנושאים החשובים בתכנות הוא תיעוד הקוד שנכתב. Java מספקת למפתחים כלי על מנת להקל על עבודת התיעוד. כלי זו הוא Javadoc. ה-Javadoc היא תכנית שבעת הפעלתה על תכנית הכתובה ב-Java היא יוצרת דפי HTML אשר מכילים תיעוד מפורט לתכנית עפ"י הערות מתאימות שנשתלות בקוד המקור.

5.2 תחביר

צורתה הכללית של הערת Javadoc היא **/ בהתחלה ו- */ בסוף.
כל הערת Javadoc נחלקת לשניים: בתחילתה מופיע טקסט ובהמשך נכתבות תגיות ה-Javadoc.
בגרסאות ישנות של Javadoc נדרש גם שבין שני החלקים (הטקסט המתאר והתגיות) תופיע שורה ריקה שבתחילתה *.

דוגמא:

```
/**
 * This is the descriptive text of the doc comment.
 *
 * @Xxx     Comment for the tag.
 * @Yyy     Comment for the tag.
*/
public class Aaa
{
}
```

הערת ה-javadoc חייבת להופיע בצמידות מעל הגדרתה של מתודה/משתנה/מחלקה וכו'... וללא כל שורת ריווח ביניהם.

גם בשורת ההתחלה וגם בשורת הסיום, מלבד סימני ההתחלה: /** ו- */ לא יופיע דבר.
כל שורה נוספת אשר תופיע בהערת ה-javadoc תתחיל בסימן *.

ניתן להשתמש בהערות בתגי HTML בתוך ההערות. תגי ה-HTML יועברו כמו שהם אל קובץ הפלט.
בד"כ נשתמש למשל בתגית <P> כדי לעבור לפיסקה חדשה או בתגיות <LI></LI> כדי ליצור רשימה של אלמנטים.

המשפט הראשון בכל הערת Javadoc צריך להיות משפט קצר אשר מהווה תיאור קצר של הפריט שאליו ההערה מתייחסת. משפט ההתחלה מסתיים עם הנקודה הראשונה שאחריה מרווח או סוף שורה. אם מעוניינים בכך שהנקודה הראשונה לא תהווה את סיומו של המשפט הקצר אז יש לדאוג שלא יהיה אחריה ריווח. המשפט הראשון יופיע בחלק ה-summary, כתיאור תמציתי לכל פריט ופריט.

5.3. הערות אותן Javadoc משלב באופן עצמאי

קיימים מצבים שבהם ה-Javadoc משלב הערות באופן עצמאי. הכרת מקרים אלה יכולה לחסוך בקידוד מיותר:

- כאשר במחלקה או ממשק שמגדירים, שיטה דורסת שיטה אחרת שמגיעה בהורשה ממחלקה אחרת או ממשק אחר אז ה- Javadoc ישלב בהערות שהוא ייצר עבור השיטה הדורסת את המשפט: "Overrides" עם קישור לתיעוד של השיטה שנדרסה (במחלקה או הממשק המוריש).

- כאשר במחלקה שמגדירים, שיטה דורסת שיטה אחרת שמגיעה מ-interface אשר מיושם אז ה- Javadoc ישלב בהערות שהוא ייצר עבור השיטה הדורסת את המשפט: "Specified by" עם קישור לשיטה (כפי שהוגדרה ב-interface).

- בכל אחד מהמקרים הנ"ל, אם השיטה אשר הוגדרה לא לוותה בהערות עבור ה-Javadoc אז יצטרפו אליהן אוטומטית ההערות שניתנו לשיטה הנדרסת (כפי שנכתבו במחלקה המורישה או ב-interface המיושם). מסיבה זו, אם ההערות שנכתבו ב-interface המיושם או במחלקה המורישה מספקות אז לא כדאי להוסיף הערות נוספות.

כאשר משלבים בתוך הערת ה-javadoc שמות (של מחלקות,packages, interfaces, מתודות, משתנים, ארגומנטים ודוגמאות של קוד) או מלים שמורות אז יש לסמנם באמצעות התגיות <code> ו-</code> לדוגמא:

```
/**

 * This class is an improved version of the <code>Button</code>
class.

 *
 * . . .
 *

 */
```

ניתן לשלב בתוך הערת Javadoc קישור למסמך אחר אשר מתאר מחלקה אחרת באמצעות link@. יש להימנע מקישור (באמצעות link@) של כל אחד משמות המחלקות אשר מופיעים בהערות ה-Javadoc אל המסמך המתאר את אותה מחלקה.

המשפט הראשון בכל הערת javadoc איננו חייב להיות משפט. הוא יכול להיות גם ביטוי או צירוף של מספר קטן של מלים שאיננו מהווה משפט אך דיו כדי להסביר. בדרך זו ניתן, לעתים, לקבל בהירות גבוהה יותר. הדבר תקף בייחוד בנוגע לתגית param@. דוגמא:

```
/**
 * Requests that this applet be resized.
 *
 *  @param   width    The new requested width for the applet
*/
```

5.4. התגיות שקיימות ב-javadoc

בתוך הערות ה-javadoc ניתן, כפי שהוצג, לשלב תגיות בעלות משמעות מיוחדת בתוצאה הסופית של פעולת תכנית ה-javadoc. תגיות אלה מתחילות בסימן @ וניתן למקמן בתחילת השורה בלבד. אם משלבים את אותה תגית מספר פעמים בתוך אותה הערת javadoc אז יש למקם את התגיות הללו בקבוצה אחת. כך, למשל, אם התגית @author מופיעה יותר מפעם אחת בתוך אותה הערה אז יש לרשום את כל השורות הכוללות תגית זו יחד, אחת מתחת לשניה. בדרך זו, תכנית ה-javadoc תתייחס אל קבוצת התגיות הזהות באופן מיוחד.

את תגיות ה-javadoc ניתן לחלק לשלוש קבוצות:

קבוצת התגיות שניתן לשלב בהערות javadoc אשר נכתבות לצורך יצירת תיעוד של field:
בתוך הערת javadoc אשר מוצמדת ל-field ניתן לכלול את התגיות הבאות: **@see**, **@since** ו-**@deprecated** בלבד.

קבוצת התגיות שניתן לשלב בהערות javadoc אשר נכתבות לצורך יצירת תיעוד של class או interface:

@author *author name*
משמשת לציון שמו של כותב ה-interface/class.
ניתן לשלב מספר תגיות @author בתוך אותה הערת javadoc.

@see *className*
ליצירת קישור שלחיצה עליו תוביל לתיעוד של class, interface, method או field מסוימים. קיימות מספר אפשרויות בעת הפעלת תגית זו:
ניתן להפנות לתיעוד של field או method (בשם שצוין) אשר שייכים לאותה מחלקה:
@see # *nameOfMethodOrField*
ניתן להפנות לתיעוד של field או method (בשם שצוין) אשר שייכים למחלקה אחרת:
@see *nameOfOtherClass#nameOfMethodOrField*
ניתן להפנות לתיעוד של method מסוים מתוך מספר מתודות בעלות שם זהה (overriding methods) אם מציינים בתוך סוגריים אשר באים אחרי שמה של המתודה את הטיפוס/הטיפוסים של הפרמטר/הפרמטרים שלה.
@see java.awt.Container#add(String,Component)
ניתן לשלב בתוך הערת javadoc של class\interface יותר מתגית @see אחת.

@since *sinceText*

באמצעות תגית זו מציינים כי ה-field\method\constructor\class או interface נתמכים החל מגרסת JDK מסוימת.

@deprecated *deprecatedText*

תגית זו באה לציין כי ה-class או ה-interface המתועדים נחשבים ל-deprecated (מיושנים), ולכן יש להימנע מלהשתמש בהם. ההערה שנהוג למקם אחרי התגית הזו היא הפניה למתודה/מחלקה שבהם יש להשתמש במקום. אם אין כל תחליף למרכיב שהוכרז כ-deprecated אז נהוג לשלוח אל התגית "No replacement".

קבוצת התגיות שניתן לשלב בהערות Javadoc אשר נכתבות לצורך יצירת תיעוד ל- method או constructor:

@param *parameterNameDescription*

כדי לתת הסבר לפרמטר של ה-method או של ה-constructor. דוגמא: @param size long size of the file. אחרי התגית param@ יש לציין תחילה את שמה של התגית ורק אחריו הסבר קצר. המילה הראשונה בהסבר תהא הטיפוס של אותו פורמט, ולכן מקובל להקדימה ב-"a", "an" או "the". דוגמא: @param num the int number to be tested. את הטיפוס יש לרשום באותיות קטנות (שיהיה ברור שלא מדובר בשמה של מחלקה אלא באובייקט). ה-javadoc יוסיף את תגיות ה-CODE סביב שמו של הפרמטר באופן אוטומטי (סביב הארגומנט הראשון שישלח לתגית param@). משפט ההסבר שיופיע אחרי שמו של הפרמטר לא יסתיים בנקודה, יתחיל באות קטנה ויהיה ביטוי/אוסף מלים שאיננו מהווה משפט מלא. מסיבה זו, ההסבר יתחיל באות קטנה.

@return *description*

כדי לתת הסבר לערך המוחזר על ידי ה-method. דוגמא:@return length of the file במתודות שמחזירות void וב-construcotrs אין להשתמש בתגית זו.

@exception *fulltQualifidClassNameDescription*

כדי לתת הסבר לטיפוס ה-exception אשר עלול להיזרק. התוצאה לשימוש בתגית זו היא הופעת Throws: במסמך ה-HTML אשר ייוצר ובו רשימת סוגי ה-exception אשר עלולים להיזרק מן המתודה. כל סוג של exception יופיע כקישור למסמך ה-javadoc אשר מתאר אותו (את מחלקת ה-exception המתאימה). דוגמא: @exception IOException if the file is too big

התגית throws@ זהה בפעולתה לתגית exception@. יש לתעד כל exception שצוין ב-throws או exception שיש סבירות גבוהה להניח שהמתכנת ירצה לנסות לתפוס אותו (למעט errors ו-NullPointerException).

@see *className*

תגית זו, שהוסברה לעיל, גורמת להוספת "See Also" מתאים. תגית זו ניתן להפעיל במספר אופנים (ראה מעלה).

@since *sinceText*

ראה את ההסבר שכבר הוצג. התגית since@ תופיע רק בהסבר אשר ניתן ל-class/interface. אין לכלול את התגיות since@ בהסברים שניתנים למרכיבי ה-class/interface אלא אם הם הוספו מאוחר יותר בגרסה יותר מאוחרת של אותה class/interface.

@deprecated *deprecatedText*

לציון method\constructor מיושנים (deprecate). מייד לאחר התגית נהוג לספק הסבר שהמשפט הראשון שבו יאמר מתי זה נהיה deprecated ולאחריו מהו התחליף שבו יש להשתמש. המשפטים שיבואו אחר כך יכולים לכלול הסברים למה זה נהיה deprecated. כאשר מסבירים מהו התחליף יש להשתמש בתגית link@ כדי שיופיע קישור למיקום שבו מופיע התיעוד של התחליף.

אם אין כל תחליף למה שנקבע כ-deprecated אז יש לציין "No replacement".

{@link}

תגית זו באה לציין קישור למקור אחר.

להלן דוגמא לשימוש בתגית זו בתוך קובץ קוד המקור של המחלקה Applet:

```
    /**
     * Returns an absolute URL naming the directory of the document
in which
     * the applet is embedded.  For example, suppose an applet is
contained
     * within the document:
     * <blockquote><pre>
     *    http://java.sun.com/products/jdk/1.2/index.html
     * </pre></blockquote>
     * The document base is:
     * <blockquote><pre>
     *    http://java.sun.com/products/jdk/1.2/
     * </pre></blockquote>
     *
     * @return  the {@link java.net.URL} of the document that
contains this
     *          applet.
     * @see     java.applet.Applet#getCodeBase()
     */
    public URL getDocumentBase() {
      return stub.getDocumentBase();
    }
```

יצירת תיעוד ל- Default Constructor:

כדי לתעד default constructor יש להגדירו מפורשות מחדש ולשלב בהגדרה שלו את ההערות שבהן מעונינים.

5.5. תיעוד ה-packages

כדי לתעד את ה-packages יש ליצור את המסמך package.html, לשמור אותו בתיקיה שבה שמורים קבצי קוד המקור (*.java). בכל תיקיית קבצי קוד מקור ששייכים ל-package מסוים יש לרשום את המסמך package.html אשר מתעד את אותו package מסוים.

המסמך package.html אשר יישמר בכל תיקיה של package יכלול הערות אשר יתווספו לתקציר המחלקות וה-interfaces . תחילה יופיע תקציר המחלקות וה-interfaces ורק אחריו יופיע תוכנו של המסמך package.html (כל מה שמופיע בתוך מרכיב ה- BODY שבתוכו). תוכנו של package.html צריך לכלול תיאור של השירותים שאותו package מספק, את הקשרים שיש בין המחלקות וה-interfacesc אשר מרכיבים אותו ואף קישורים למסמכי תיעוד נוספים חיצוניים ל API documentation.

התוכן של קובץ זה הוא למעשה הערה גדולה אחת הכתובה בפורמט HTML. אין צורך לסמן את הקובץ בעזרת **/ ו-/*.

להלן תוכנו של package.html:

בתחילה:

יופיע החלק "Package Xxx Description". חלק זה יתחיל במשפט סיכום קצר אשר מציין מה ה-package מספק. למשל, " Provides classes and interfaces for handling mobile communication".. אחריו יש לספק תיאור יותר מפורט.

בהמשך: יש לספק את ה-package specification: תיאורים וקישורים לאינפורמציה רלוונטית אחרת.

אחר כך: יש לספק Related Documentation: קישורים למסמכים אחרים (דוגמאות, מדריכים וכו').

כאשר ניתן יהיה להשתמש בתגית @category אז יופיע חלק נוסף:

תיאור לוגי של קבוצות ה-classes וה-interfaces שיש ב-package זה. כמו כן, קישורים ל-packages, classes ו-interfaces אחרים.

5.6. הפעלת Javadoc

את תכנית ה-Javadoc מפעילים בשורת הפקודה באופן הבא:

javadoc [options] [package | source.java]

כלומר, מייד לאחר שכותבים בשורת הפקודה "Javadoc" יש לרשום את שם (שמות) ה-package (ה-packages) שעבורם ייווצרו מסמכי HTML. מסמכי ה-HTML שייווצרו כוללים תיעוד של המרכיבים שהרשאת הגישה שלהם היא public או protected בלבד. מסמכים אלה כוללים מסמך תיעוד לכל מחלקה ומסמך תיעוד לכל package. בנוסף, תכנת ה-Javadoc גם יוצרת מסמך HTML אשר מתאר את היררכית המחלקות אשר נוצרת (tree.html) ומערך מסמכי HTML אשר מהווה index לכל האלמנטים אשר תועדו.

חלק מה-options שניתן לשלב לפני שמו/שמם של קובץ קוד המקור/קבצי קוד המקור (או שמו/שמם של ה-package\packages) הינם:

public-
מסמכי ה-HTML יכללו תיעוד של מחלקות ורכיבי מחלקות שהרשאות הגישה שלהם היא public בלבד.

protected-
מסמכי ה-HTML יכללו תיעוד של מחלקות ורכיבי מחלקות שהרשאות הגישה שלהם הן public או protected.

package-
מסמכי ה-HTML יכללו תיעוד של מחלקות ורכיבי מחלקות שהרשאות הגישה שלהם הן public, protected או package friendly בלבד.

private-
מסמכי ה-HTML יכללו תיעוד של כל המחלקות ורכיבי המחלקות הקיימים.

-author

כדי שמסמכי ה-HTML יכללו את תוצאותיהן של התגיות @author יש להשתמש באפשרות זו.
בברירת המחדל תגיות אלה לא נלקחות בחשבון בעת יצירת מסמכי ה-HTML.

-d directory

כדי לקבוע את הספרייה שבה יישמרו מסמכי ה-HTML אשר ייווצרו.

-sourcepath path

באמצעות אפשרות זו ניתן לציין את ה-path למיקום שבו נמצאת ספרית ה-package העליונה ביותר בהיררכית הספריות אשר מייצגות את ה-packages השונים (אם ה-Javadoc מופעל כדי לייצור תיעוד לפרוייקט כולו) או את ה-path למיקום שבו נמצאים קבצי ה-source code המסוימים שאותם רוצים לתעד בהערות javadoc. בדרך כלל אין כל צורך להשתמש באפשרות זו כיוון שנהוג להפעיל את javadoc בספריה שבה נמצאת הספרייה אשר מייצגת את ה-package העיקרי (ה-package שבראש היררכית ההורשה).