הבסיס (Basic)

הקדמה – Introduction

פעולות פלט פשוטות - Simple output

הערות בגוף התכנית - Remarks

הטיפוסים הבסיסיים - Native Data Types

הצהרה על משתנים - Declared Variables

שמות מזהים – Identifiers

קבועים – Literals

אופרטורים וביטויים - Expressions & Operators

משפטים פשוטים ומורכבים - Simple & Compound Statements

משפטי בקרה- Control Structure

משפט השמה - Assignment Statement

טווח ההכרה של משתנים - Local Variable Scope

השימוש ב - enum

© 2007 Zindell Technologies, Ltd.

הקדמה - Introduction

כדי ליצור שפה שקל ללמד אותה, התבסו מפתחיה של Java על התחביר של C ושל C++. הצלחתה של השפה תלויה במידת הפופולריות שלה. שפה שדומה לשפה מוכרת קל יותר ללמוד, ומכאן ההחלטה להתבסס על השפות C ו- C++.

אם שתי השפות האלה מוכרות לך, פרק זה יהיה די קליל עבורך. למרות שבמבט ראשון ניתן לראות שהפקודות שמוצגות בפרק קיימות גם ב-C++, כדאי מאד לעבור באופן יסודי על הפרק. הפקודות שקיימות גם ב-JAVA וגם ב-C++\C בדרך כלל לא פועלות באופן זהה.

JAVA משתמשת בתחביר שמאוד דומה לתחביר של שפת התכנות C בנושאים הבאים:

1. אופרטורים
2. ביטויים
3. משפטי תנאי ומשפטי בקרה

כמו כן, ניתן לגלות דמיון רב מאוד בין התחביר של JAVA לתחביר של C++ בנושאים הבאים:

1. אופן הגדרתן של מחלקות חדשות
2. אופן הגישה למשתנים ולפונקציות ששייכות למחלקה

פעולות פלט פשוטות - Simple Output

כדי לבצע פעולת פלט פשוטה של מחרוזת תווים אל המסך ניתן להשתמש במתודות print ו- println באופן הבא:

```
System.out.println(stringToPrint);

System.out.print(stringToPrint);
```

© 2007 Zindell Technologies, Ltd.

ההבדל בין שתי המתודות האלה הוא שהראשונה גם גורמת לכך שסמן הטקסט יורד לשורה חדשה לאחר הדפסת המחרוזת. המתודה השניה לא גורמת לסמן הטקסט לרדת לשורה חדשה.

אם תפיסת תכנות מונחה העצמים מוכרת לך אז תוכל להבין מעט יותר מהתחביר הנ"ל. המתודה print וכמוה גם המתודה println הן מתודות שמוגדרות במחלקה PrintStream. במחלקה System מוגדר משתנה סטטי אשר מכיל reference לאובייקט מטיפוס PrintStream. שמו של המשתנה הסטטי הנ"ל הוא out. במחלקה PrintStream מוגדרת המתודות print ו- println, ולכן, ניתן להפעיל באמצעות out (על האובייקט שה-reference שלו בתוך out) את שתי המתודות האלה.

אם רוצים לבצע פעולות פלט של מחרוזת תווים בצירוף ערכו של משתנה כלשהו אז יש לשרשר אל מחרוזת התווים את שם המשתנה באמצעות הסימן +.

לדוגמא:

```
System.out.println("name=" + name);
```

שתי תכניות הדוגמא הבאות מדגימות את פעולות הפלט הפשוטות. התכנית הראשונה מדגימה את פעולת הפלט בצורתה הפשוטה ביותר. התכנית השניה מציגה את האפשרות לבצע פלט למסך מבלי לרדת לשורה חדשה באמצעות המתודה print, וכמו כן, את האפשרות להשתמש בחלק מקודי החילוף שקיימים ב-C (כגון '\n').

```
class PrintlnDemo
{
    public static void main(String args[])
    {
      int year=1999;
      int month=4;
      int day=1;
```

© 2007 Zindell Technologies, Ltd.

```
        System.out.println("The year is 1999 and the month is 4");
        System.out.println("The year is " +
         year +  " and the month is " + month);
      }
}


 class PrintDemo
{
    public static void main(String args[])
    {
      int num1,num2,num3,num4;
      num1=1;
      num2=2;
      num3=3;
      num4=4;
      System.out.println("num1="+num1);
      System.out.println("num2="+num2);
      System.out.println("num3="+num3);
      System.out.println("num4="+num4);
      System.out.print("num1="+num1);
```

© 2007 Zindell Technologies, Ltd.

```
    System.out.print("num2="+num2);

    System.out.print("num3="+num3);

    System.out.print("num4="+num4);

    System.out.println("\n\n");

    System.out.println("println moves one line down");

    System.out.println("println moves one line down");

    System.out.println("println moves one line down");

    System.out.print("print with \\n the same\n");

    System.out.print("print with \\n the same\n");

  }

}
```

כפי שוודאי ראית, שרשור מחרוזות תווים נעשה באמצעות האופרטור +. לאופרטור זה יש, למעשה, שני תפקידים: האחד אריתמטי (לבצע פעולות חיבור בין ערכים מספריים), והשני לשרשר מחרוזות תווים אחת לשניה ובכך לקבל מחרוזת תווים חדשה שמורכבת מהמחרוזות שהיו נתונות. כאשר האופרטור + מבצע חיבור בין מחרוזת תווים לערך מספרי אז הערך המספרי הופך למחרוזת תווים, ובדרך זו מקבלים חיבור בין שתי מחרוזות.

הערות בגוף התכנית - Remarks

הערות בגוף התכנית (בתוך קובץ קוד המקור) הן קטעי טקסט שמשולבים בתוך התכנית, והמהדר מתעלם מהן. ב-JAVA אימצו את מה שכבר קיים בשפות C ו- C++. ניתן לשלב בתוך קוד המקור שכתוב ב-JAVA גם הערות בסגנון שמקובל ב-C, וגם בערות בסגנון המקובל ב- C++.

© 2007 Zindell Technologies, Ltd.

הערה בסגנון C

```
/* This is a remark */
```

הערה בסגנון C++

```
// This is a remark
```

בדומה ל-C, גם ב-JAVA הערה בסגנון שמקובל ב-C יכולה להשתרע על פני יותר משורה אחת, ובדומה ל- C++, גם ב-JAVA הערה בסגנון שמקובל ב-C++ יכולה להשתרע על פני שורה אחת בלבד.

סוג נוסף של הערות שניתן לשלב בתכנית שכתובה ב-JAVA הוא הערות עבור ה-javadoc כפי שהוצג בפרק הראשון.

דוגמא להערה שמיועדת ל-javadoc:

```
/** This is a javadoc remark.
  * This tool is very efficient
  * @author Haim Michael      */
```

הטיפוסים הבסיסיים - Native Data Types

בדומה ל-C, ולשפות אחרות, גם ב-JAVA מוגדרים טיפוסי ערכים בסיסיים (פרימיטיביים) כחלק מהשפה. טיפוסי הערך הפרימיטיביים ניתנים לשימוש באופן אוטומטי מבלי שיהיה צורך להגדירם. הטבלה הבאה מספקת את האינפורמציה העיקרית על הטיפוסים הבסיסיים שקיימים ב-JAVA.

הטיפוס	סוג הערך	מספר הביטים	ברירת המחדל	ערכי מינימום ומקסימום
boolean	true או false	1	false	false או true
char	Unicode Characters	16	\U0000	\U0000 עד \UFFFF

© 2007 Zindell Technologies, Ltd.

byte	מספר שלם חיובי או שלילי או אפס	8	0	128- עד 127
short	מספר שלם חיובי או שלילי או אפס	16	0	32768- עד 32767
int	מספר שלם חיובי או שלילי או אפס	32	0	2146483648- עד 2147483647
long	מספר שלם חיובי או שלילי או אפס	64	0	ממספר שלם מאוד גדול וחיובי עד מספר שלילי קטן במיוחד
float	מספר ממשי (יכול להיות שלם ויכול להיות עם שבר עשרוני)	32	0.0	טווח מספרים ממשיים גדול (עם שבר עשרוני ובלי שבר עשרוני). שליליים/חיוביים: מ- 3.40282347E+38 עד- -3.40282347E+38 ובמספרים קטנים במיוחד: מ- 1.402398E-45 עד- -1.402398E-45

© 2007 Zindell Technologies, Ltd.

double	מספר ממשי (יכול להיות שלם ויכול להיות עם שבר עשרוני)	64	0.0	טווח המספרים הממשיים גדול מטיפוס זה עוד יותר גדול ויותר מדויק מהטיפוס הקודם

כדאי לשים לב לכך, שכיוון ש-JAVA פועלת על ה-JVM שמותקן במחשב, בכל מחשב ומחשב ה-JVM מספק, למעשה, סביבה אחידה, ולכן מספר הבתים שדרוש לייצוג כל אחד מהטיפוסים הבסיסיים זהה בכל מחשב ומחשב. תכונה זו – להזכירך – איננה מתקיימת, בהכרח, בשפות אחרות כגון C ו- C++.

ערך ברירת המחדל מוכנס למשתנים של מחלקה כאשר הם לא מאותחלים באופן מפורש בערך אחר. יש לשים לב לכך, שמשתנים מקומיים במתודות לא מאותחלים באופן אוטומטי בערך ברירת המחדל, ובאחריות המתכנת לדאוג לאיתחולם. אם שורות אלה היו מעט מבלבלות עבורך כדאי שתחזור ותקרא אותם שוב כאשר תסיים לקרוא את הפרק שמסביר את אופן יצירתם של אובייקטים, ואת תפקיד המחלקות.

boolean

זהו טיפוס ערך (לאחרונה הופיע גם ב-C++) אשר יכול לקבל את הערך true או false בלבד. כל ביטוי לוגי (אם אינך יודע מהו ביטוי לוגי תוכל לחזור ולקרוא שורות אלה לאחר שתסיים את הפרק) הוא בעל ערך מטיפוס boolean. טיפוס זה לא ניתן לשנות לטיפוס אחר, ולכן, בכל מקום שבו אמור להופיע ערך מטיפוס boolean (בתוך משפט תנאי למשל) לא ניתן לרשום ביטוי בעל ערך מספרי כפי שניתן לעשות ב-C וב-C++. במקומות שבהם יש לרשום ערך מטיפוס boolean (במשפטי תנאי, בלולאות...) חייבים לרשום ערך מטיפוס boolean ולא ניתן לרשום ערכים מטיפוס אחר כדי שיעברו casting וישנו את טיפוסם ל-boolean.

Integer Types

כבר במבט ראשון על הטבלה ניתן להבחין במאפיינים הבאים:

א. ב-JAVA לא קיים ה-modifier: unsigned כפי שהוא קיים ב- C\C++. ב-JAVA לא ניתן להוסיף לפני שם של טיפוס מספרי את המילה unsigned ולקבל בכך טיפוס חדש אשר יוכל להכיל אך ורק ערכים אי שליליים. ב-JAVA,

© 2007 Zindell Technologies, Ltd.

כל אחד מארבעת הטיפוסים המספריים השלמים שנתונים יכול להכיל ערכים מספריים שלמים, חיוביים ושליליים. לא קיימת האפשרות להגביל את אף אחד מארבעת הטיפוסים הללו כך שיוכל להכיל אך ורק ערכים אי-שליליים.

ב. ב-JAVA, המלים long ו- short מהוות טיפוסים בפני עצמם. הם לא משמשים כמקדמים (modifiers) אשר ניתן להוסיף לפני שם טיפוס אחר ולקבל משמעות אחרת לטיפוס.

ג. ב-JAVA, הטיפוס char תופס 2 בתים ולא בית אחד כמו ב-C\C++. הסיבה לכך היא התמיכה ש-JAVA מעניקה לשימוש בטבלת ה-unicode (טבלה ענקית אשר מכילה קודים לתווים של רוב השפות הקיימות). טבלת ה-unicode זהה ב-128 התווים הראשונים שלה לטבלת ה-ASCII, וכדי לייצג את מרבית התווים שמופיעים בה יש צורך בשני בתים כדי לאחסן את הקודים שלהם.

ד. הטיפוסים: int, short, byte ו- long יכולים להכיל אך ורק ערכים שלמים עם סימן. ההבדל שיש ביניהם הוא רק במספר הבתים שמשמש את כל אחד מהם, כלומר, בטווח הערכים האפשריים שכל אחד מהם יכול לייצג.

הטיפוס	מספר הביטים	טווח הערכים
byte	8	$-2^7 \ldots 2^7 - 1$
short	16	$-2^{15} \ldots 2^{15} - 1$
int	32	$-2^{31} \ldots 2^{31} - 1$
long	64	$-2^{63} \ldots 2^{63} - 1$

בין ערכים מטיפוס כל אחד מארבעת הטיפוסים הללו ניתן לבצע פעולות כפל/חיבור וכדומה . . .

התכנית הבאה מציגה פעולת החלפה בין ערכיהם של שני משתנים.

```
class SwapExample

{
```

© 2007 Zindell Technologies, Ltd.

```
    public static void main(String args[])
    {
        int num1,num2;
        num1=3;
        num2=4;
        System.out.println("\nnum1="+num1);
        System.out.println("\nnum2="+num2);
        num1=num1+num2;
        num2=num1-num2;
        num1=num1-num2;
        System.out.println("\nnum1="+num1);
        System.out.println("\nnum2="+num2);
    }
}
```

Floating Point Types

קבוצה זו כוללת את הטיפוסים: float ו- double. ניתן לבצע פעולות חשבון בסיסיות בין ערכים מטיפוסים אלה, ולצרפם לביטוי חשבוני. ההבדל בין שני הטיפוסים הללו הוא מספר הבתים שמשמש כל אחד מהם, כלומר, טווח הערכים ומידת דיוקם.

חלוקת ערך מטיפוס double או float ב-0 תניב את אחת משלושת התוצאות הבאות:

כאשר מחלקים ערך חיובי ב-0:

© 2007 Zindell Technologies, Ltd.

PI(Positive Infinity)

על המסך יופיע: **Infinity**

כאשר מחלקים ערך שלילי ב-0:

NI(Negative Infinity)

על המסך יופיע: **-Infinity**

כאשר מחלקים 0.0 ב-0:

NaN(Not A Number)

כאשר משווים NaN לכל ערך אחר, לרבות NaN, תמיד מקבלים false.

המלים השמורות: Infinity ו- -Infinity אינם הערכים הגדולים/הקטנים ביותר שיכולים להיות במחשב. אלה שתי מלים שמורות שמייצגות אינסוף בלבד.

בין ערכים מטיפוס double ו- float ניתן לבצע פעולות כפל/חיבור וכדומה.

התכנית הבאה מדגימה את שהוסבר.

```
class SpecialFloatValues
{
    public static void main(String args[])
    {
        System.out.println("pNum divide into 0 : "+(12.4/0));
        System.out.println("nNum divide into 0 : "+(-12.4/0));
```

© 2007 Zindell Technologies, Ltd.

```
        System.out.println("zeroNum divide into 0 : "+(0.0/0));
    }
}
```

כבר עתה, למרות שעדיין אינך נדרש/ת להכיר את הנושאים שקשורים בתכנות מונחה עצמים, כדאי לשים לב לדברים הבאים:

כאשר מדובר במשתנה מסוג instance variable (משתנה שמופיע בכל אובייקט) או מסוג class variable (משתנה סטטי של המחלקה) - אם לא מאתחלים אותו אז הוא מקבל ערך התחלתי בהתאם לערך ברירת המחדל של הטיפוס של אותו משתנה.

כאשר מדובר במשתנה מסוג local variable (משתנה מקומי של פונקציה) אז חייבים לאתחל אותו. משתנה מקומי יכול להיות גם משתנה מקומי שמוגדר בתוך בלוק. במקרה כזה טווח ההכרה שלו ואורך החיים שלו יהיו הבלוק עצמו בלבד.

כאשר מדובר במשתנה שאיננו מטיפוס בסיסי כי אם מטיפוס מחלקתי אז ערך ברירת המחדל שלו הוא null. יש לשים לב לכך ש-null ב-JAVA נכתב באותיות קטנות. משתנה מטיפוס מחלקתי יכיל null כל עוד לא הוכנס לתוכו reference לאובייקט מסוים (כאשר הוא לא משתנה מקומי).

הצהרה על משתנים - Declaring Variables

יצירתם של משתנים ב-JAVA נעשית בדומה ל-C\C++. תחילה יש לרשום בתחילת השורה את שם הטיפוס, ובהמשך השורה (משמאל לימין) יש לרשום את שם המשתנה או את שמות המשתנים שרוצים לייצור. אם רושמים שמות של משתנים אז יש לציין פסיקים מפרידים ביניהם.

לדוגמא:

```
int num1, num2;
```

אם רוצים, אז ניתן גם לאתחל את המשתנים בערך התחלתי כבר בשורת ההצהרה.

© 2007 Zindell Technologies, Ltd.

לדוגמא:

```
int num1=12, num2=24;
```

יש לשים לב לכך ש-short ו-long ב-JAVA אינם modifiers (מלים שניתן להוסיף לשמו של טיפוס קיים ולקבל טיפוס מעט שונה).

שמות מזהים - Identifiers

Identifier הוא כל שם (שם של משתנה, מתודה, קבוע, מחלקה וכדומה) אשר מופיע בתכנית. השם יכול להיות מורכב אך ורק מספרות, $, קו תחתי או אותיות. השם חייב להתחיל באחת האפשרויות הבאות: אות, קו תחתי או $.

ב-JAVA בדומה ל-C\C++ קיימת רגישות להבדלים שבין אותיות קטנות וגדולות. יש לשים לב לכך. רוב הטעויות של מי שלומד לתכנת בפעם הראשונה נובעות מחוסר תשומת לב לרגישות זו.

המוסכמה היא שיש לכתוב את שמות המשתנים באותיות קטנות בלבד. אות גדולה תופיע בתוך שם של משתנה רק בתור הפרדה ויזואלית בין מלים שונות שמתחברות יחדיו למילה אחת (שם של משתנה אחד).

לדוגמא:

```
int numOfTheStudents;
```

המוסכמה בנוגע לשמות של מחלקות היא לכתוב אותם עם אות גדולה בהתחלה. כך לדוגמא, המחלקה System שכבר הכרת. (אם פסקה זו איננה מובנת לך אל תתעכב. חזור אליה לאחר קריאת הפרק שדן במחלקות).

גם ב-JAVA, בדומה ל-C\C++ , קיימות מלים שמורות, אשר לא ניתן להשתמש בהן בתור שמות מזהים. ההכרות עם קבוצת המלים השמורות תיעשה תוך לימוד השפה.

קבועים – Literals

את הקבועים שקיימים ב-JAVA ניתן לחלק למספר סוגים:

© 2007 Zindell Technologies, Ltd.

קבועים מסוג integer

אם לא צוין אחרת, אז כל מספר שלם שנכתב בתכנית נחשב לקבוע מטיפוס int. אם רושמים אותו בצירוף התוספת המקדימה 0x אז הוא ייחשב למספר בבסיס 16. אם רושמים אותו בתוספת 0 אז הוא ייחשב למספר בבסיס 8. אם רושמים מספר ללא אף אחת מבין שתי התוספות הללו אז הוא ייחשב למספר בבסיס 10 ("10 אצבעות לי יש ...").

ניסיון לרשום מספר, שכדי לייצגו במחשב יש צורך ביותר מ-32 ביטים, ומבלי לציין שהוא מטיפוס long באמצעות הוספת האות 'L' או 'l' בסופו, יגרום להופעת הודעת שגיאה מצד המחשב. הוספת האות 'L' (אות גדולה או קטנה – אין לכך חשיבות) לסופו של המספר השלם תגרום לכך שהטיפוס שלו יהיה long ולא int.

לדוגמא:

```
long num = 500000000000000L;
```

אי הוספת האות 'L' אל סופו של המספר במקרה זה הייתה גורמת לשגיאה כי הקבוע המספרי היה חורג מטווח הערכים של ערך מטיפוס int.

קבועים מטיפוס float או double

מספר עם נקודה עשרונית ייחשב לקבוע מטיפוס double. אם רוצים שהוא ייחשב לקבוע מטיפוס float אז יש לרשום בסופו את האות 'f' או 'F' (אין חשיבות להיותה של האות גדולה או קטנה).

כך לדוגמא,

```
float avg = 68.5f;
```

יש לשים לב לכך שאם האות F לא הייתה מוספת אז השורה הזו לא הייתה עוברת קומפילציה כיוון שלא ניתן היה לבצע השמה של ערך מטיפוס double אל תוך משתנה מטיפוס float ללא ביצוע ה-casting המתאים.

מספר בכתיב מדעי (45E32 למשל) גם ייחשב למספר מטיפוס double.

ניתן לבצע casting למספר שלם כך שיהיה מטיפוס double על ידי הוספת האות 'd' או 'D' אל סופו.

ניתן לבצע casting למספר שלם כך שיהיה מטיפוס float על ידי הוספת האות 'f' או 'F' אל סופו.

© 2007 Zindell Technologies, Ltd.

קבועים מטיפוס char

קבוע מטיפוס char יופיע בתור תו עם גרשיים בודדים סביבו. תווים שאין להם מראה (כגון התו שאחראי לירידת סמן הטקסט לשורה חדשה במסך) ייכתבו באמצעות התו \ כשאחריו אות מסוימת. כך למשל, התו שאחראי לירידת סמן הטקסט שמופיע על המסך לשורה חדשה ייכתב באופן הבא:

'\n'

השפה JAVA תומכת ברוב קודי החילוף שקיימים ב-C. קוד חילוף הוא תו בלתי נראה (יש לו קוד ASCII) שבשליחתו לפלט של המסך מתבצעת פעולה מסוימת, כגון: ירידה לשורה חדשה, תזוזת סמן הטקסט לכיוון ימין למרחק של tab אחד וכו'.

השפה JAVA מאפשרת לכתוב קבועים תווים במספר דרכים:

1. כתיבת התו באמצעות גרשיים בודדים סביבו. דוגמא: 'A'
2. כתיבת התו באמצעות ייצוגו המספרי על פי טבלת ה-Unicode, ובבסיס 16: '\U0041'
3. כתיבת התו באמצעות ייצוגו המספרי על פי טבלת ה-Unicode, ובבסיס 8: '\101'

שתי הדרכים האחרונות מאפשרות, למעשה, לעשות שימוש בתווים שאין להם ייצוג גרפי על המסך. התכנית הבאה מציגה אפשרויות אלה.

```
class CharDemo
{
    public static void main(String args[])
    {
        char tavA1='A';
        char tavA2='\101';
        char tavA3='\u0041';
```

© 2007 Zindell Technologies, Ltd.

```
        System.out.println("tavA1="+tavA1);

        System.out.println("tavA2="+tavA2);

        System.out.println("tavA3="+tavA3);

    }

}
```

קבועים מטיפוס boolean

ב-JAVA הוגדרו הקבועים: true ו- false. יש לשים לב לשוני לעומת C\++C. ב-Java הקבוע true לא שקול לערך 1, וגם לא לאף ערך אחר. false לא שקול לערך 0, וגם לא לאף ערך אחר.

כמו כן, כדאי לשים לב לכך שב-JAVA: true, false ו- null מופיעים באותיות קטנות ולא באותיות גדולות כמו ב- C\++C.

מחרוזות תווים

ב-JAVA, בדומה לשפות אחרות, קבוע מחרוזתי הוא אוסף של תווים (תו אחד או יותר או אפילו 0 תווים) אשר מופיעים בתוך גרשיים כפולים.

לדוגמא:

"I LOVE JAVA"

אופרטורים וביטויים - Operators Expressions

הקדמה

ביטוי הוא כל מה שיש לו ערך. הביטוי הפשוט ביותר הוא משתנה (או קבוע) אחד שיש לו ערך מטיפוס מסוים.

ביטויים יותר מורכבים כוללים בתוכם אופרנדים ואופרטורים.

© 2007 Zindell Technologies, Ltd.

אופרטור הוא סימן אשר מבצע פעולה מסוימת על אופרנד אחד או יותר.

אופרנד הוא כל מה שאופרטורים יכולים לפעול עליו.

דוגמא:

כאשר כותבים בתכנית: num1 + num2

הסימן + הוא אופרטור שפועל על שני אופרנדים (המשתנים num1 ו- num2).

אופרטורים מתמטיים שפועלים על ערכים שלמים

אופרטורים אלה נחלקים לשני סוגים:

<u>אופרטורים בינריים</u>

- לחישוב ההפרש בין שני ערכים, לדוגמא: num1-num2

+ לחישוב הסכום של שני ערכים, לדוגמא: num1+num2

* לחישוב המכפלה בין שני ערכים, לדוגמא: num1 * num2

/ לחישוב המנה שמתקבלת מחלוקת ערך אחד באחר, לדוגמא: num1/num2

% לחישוב השארית שמתקבלת מחלוקת מספר אחד במספר שני,

כך לדוגמא, num1%num2, אם נניח ש-num1 שווה ל-4 ו-num2 שווה ל-3 אז ערכו של הביטוי num1%num2 יהיה שווה ל- 1.

בדוגמאות של האופרטורים הבאים אדגים, לעתים, באמצעות ציור של 16 ביטים ולא 32 מתוך נוחות.

& - אופרטור זה מבצע פעולת AND לוגי ברמת הביטים בין שני ערכים מספריים שלמים. האופרטור פועל על שני ערכים מספריים שלמים ותוצאת פעולתו היא מספר שלם שבייצוגו הבינרי כל ביט יהיה דלוק (שווה ל-1) רק אם היה דלוק בשני המספרים שעליהם האופרטור פעל. בכל מקרה אחר הביט יהיה כבוי (שווה ל-0).

© 2007 Zindell Technologies, Ltd.

דוגמא:

בהנחה ש-num1 שווה ל-5, ו-num2 שווה ל-12 אז num1&num2 יהיה שווה ל-4.

הסבר:

ייצוגו הבינרי של המספר 5 0000 0000 0000 0101

ייצוגו הבינרי של המספר 12 0000 0000 0000 1100

תוצאת פעולת ה-& בין שני הערכים האלה היא: 0000 0000 0000 0100

| - אופרטור זה מבצע פעולת OR לוגי ברמת הביטים בין שני ערכים מספריים שלמים. אופרטור זה פועל על שני ערכים מספרים שלמים ותוצאת פעולתו היא מספר שלם וחדש שבייצוגו הבינרי כל ביט שהיה דלוק לפחות באחד שני המספרים יהיה דלוק גם במספר החדש.

דוגמא:

בהנחה ש-num1 שווה ל-5, ו-num2 שווה ל-12 אז num1 | num2 יהיה שווה ל-13.

הסבר:

ייצוגו הבינרי של המספר 5 0000 0000 0000 0101

ייצוגו הבינרי של המספר 12 0000 0000 0000 1100

תוצאת פעולת ה- | בין שני הערכים האלה היא: 0000 0000 0000 1101

^ - אופרטור זה מבצע פעולת XOR לוגי ברמת הביטים בין שני ערכים מספריים שלמים. אופרטור זה פועל על שני ערכים מספרים שלמים ותוצאת פעולתו היא מספר שלם וחדש שבייצוגו הבינרי כל ביט שהיה דלוק לפחות באחד שני המספרים אך לא היה דלוק בשניהם יהיה דלוק גם במספר החדש.

דוגמא:

בהנחה ש-num1 שווה ל-5, ו-num2 שווה ל-12 אז num1 ^ num2 יהיה שווה ל-9.

הסבר:

© 2007 Zindell Technologies, Ltd.

ייצוגו הבינרי של המספר 5 (כשהוא מטיפוס int): 0101 0000 0000 0000

ייצוגו הבינרי של המספר 12 (כשהוא מטיפוס int): 1100 0000 0000 0000

תוצאת פעולת ה- ^ בין שני הערכים האלה היא: 0000 0000 0000 1001

>> - אופרטור זה משמש להזזת הביטים שמייצגים את המספר שמאלה. מספר הצעדים שבהם הביטים יוזזו הוא המספר שימוקם מימין לאופרטור. הערך המספרי השלם שעליו האופרטור פועל ממוקם משמאל לאופרטור. במקומות הפנויים שנוצרים מוכנסים אפסים.

לדוגמא:

בהנחה שערכו של המשתנה num הוא 4, אז הערך של הביטוי num<<2 יהיה 16.

<< - אופרטור זה משמש להזזת הביטים שמייצגים את המספר ימינה. מספר הצעדים שבהם הביטים יוזזו הוא המספר שימוקם מימין לאופרטור. הערך המספרי השלם שעליו האופרטור פועל ממוקם משמאל לאופרטור.

לדוגמא:

בהנחה שערכו של המשתנה הוא 4, אז הערך של הביטוי num>>2 יהיה 1.

אם ביט הסימן דלוק אז הוא יועתק ימינה כמספר הצעדים שבהם האופרטור פועל.

לדוגמא:

ערכו של הביטוי 5>>1- הוא 1-.

ייצוגו הבינרי של 1 הוא: 0000 0000 0000 0001. כדי לדעת מהו ייצוגו הבינרי של 1- יש לחשב אותו על פי שיטת המשלים ל-2.

0000 0000 0000 0001

1111 1111 1111 1110

+1

© 2007 Zindell Technologies, Ltd.

1111 1111 1111 1111

כאשר בוחנים את ייצוגו הבינרי של 1- רואים שהוא מורכב כולו מאחדות, לרבות ביט הסימן. מסיבה זו, כאשר מתבצעת ההוזזה של הביטים ימינה, לכל מקום שמאלי שמתפנה מועתק ביט הסימן, שבמקרה זה הוא 1.

<<< - אופרטור זה דומה בפעולתו לאופרטור <<. ההבדל היחידי בין השניים הוא שבעוד שהאופרטור << העתיק את ביט הסימן לכל מקום שהתפנה, אופרטור זה מעתיק לכל מקום שמתפנה את הסיפרה 0.

לדוגמא:

ערכו של הביטוי 31<<<1- כבר לא יהיה 1-, כי אם 1.

ייצוגו הבינרי של 1- הוא:

1111 1111 1111 1111 1111 1111 1111 1111

הוזזת הביטים ימינה 31 צעדים באמצעות האופרטור <<< תגרום ל-31 מקומות במספר להתפנות ולהתמלא ב-0. בדרך זו נקבל:

0000 0000 0000 0000 0000 0000 0000 0001

גם באופרטור זה, וגם בשניים שקדמו לו, כאשר עליהם לפעול במספר צעדים שאיננו קטן ממספר הביטים שמייצגים את המספר אז מחושבת השארית מחלוקת מספר הצעדים שהאופרטור קיבל במספר הביטים שמייצגים את המספר. השארית שמתקבלת היא מספר הצעדים שיבוצעו בפועל. מסיבה זו, ערכו של הביטוי 32<<<1- נותר 1-.

אופרטורים אונריים

+ אופרטור זה מותיר את הערך שעליו פעל ללא שינוי בסימן שלו.

© 2007 Zindell Technologies, Ltd.

- אופרטור זה גורם לערך שעליו פעל "להפוך סימן".

++ אופרטור זה יכול לפעול רק על משתנה. הוא לא יכול לפעול על ביטוי שאיננו משתנה יחיד. אופרטור זה גורם לערכו של המשתנה לגדול ב-1. ניתן למקם אותו גם מימין למשתנה וגם משמאלו. הערך של המשתנה יגדל ב-1 בשני המקרים. אם נמקם אותו משמאל למשתנה אז ערכו של המשתנה בתוספת האופרטור (כביטוי) יהיה ערכו החדש לאחר ההגדלה ב-1. אם נמקם אותו מימין למשתנה אז ערכו של הביטוי (המשתנה בתוספת האופרטור) יהיה ערכו של המשתנה, כפי שהיה, לפני ההגדלה.

דוגמא:

בהנחה שערכו של המשתנה num הוא 10, אזי גם num++ וגם ++num יגרמו להגדלת ערכו של num ב-1. ההבדל בין השניים יהיה בכך שערכו של הביטוי num++ הוא 10, ואילו ערכו של הביטוי ++num הוא 11.

-- דומה בפעולתו לאופרטור ++, אלא שבמקום הגדלה ב-1 הוא גורם להקטנה ב-1.

~ אופרטור זה פועל, בדומה לאופרטורים הקודמים, על משתנה אחד ותוצאת פעולתו היא ערך מספרי חדש שבו כל ביט שהיה דלוק כעת כבוי, וכל ביט שהיה כבוי כעת דלוק. לדוגמא: אם ערכו של num1 הוא 1- אז, ערכו של ~num1 יהיה שווה ל-0.

אופרטורים מתמטיים שפועלים על ערכים מטיפוס float ו-double

כל האופרטורים שפועלים על ערכים מספריים שלמים (למעט האופרטרים שפועלים על סיביות) פועלים גם על ערכים ממשיים מטיפוס float ו- double, אלא שתוצאת פעולתם, לפעמים, מעט שונה.

האופרטורים ++ ו- --, אשר ב-C פעלו רק על משתנים מטיפוס מספרי שלם, פועלים ב-JAVA גם על משתנים מטיפוס float ו- double. אופרטורים אלה מוסיפים/מחסירים 1.0.

האופרטור % אשר ב-C פעל רק בין ערכים מספריים שלמים, פועל ב-JAVA גם על ערכים מטיפוס float ו- double.

התכנית הבאה מדגימה את חידושיה של JAVA לעומת C.

© 2007 Zindell Technologies, Ltd.

```
class FloatyOperators
{
    public static void main(String args[])
    {
        float num1,num2,reminder;
        num1=14;
        num2=4;
        num1++;
        reminder=num1%num2;
        //reminder=num1- ((int)(num1/num2)*num2)
        System.out.println("reminder="+reminder);
    }
}
```

אופרטורים להשוואה בין ביטויים

האופרטורים להשוואה בין ביטויים הם אופרטורים שפועלים על שני ביטויים ומחזירים true או false. יש לשים לב להבדל שקיים באופן פעולתם לעומת C. הערך שאופרטורים אלה מחזירים ב-JAVA איננו 1 או 0 כמו ב-C, כי אם, true או false.

להלן טבלה מסכמת אשר מציגה את פעולתם של אופרטורים אלה. הנח כי A ו-B הם שני ביטויים מספריים.

האופרטור	דוגמא	מהות
<	A<B	האם ערכו של A קטן מערכו של B ?

© 2007 Zindell Technologies, Ltd.

<	A>B	האם ערכו של A גדול מערכו של B ?
<=	A<=B	האם ערכו של A קטן או שווה לערכו של B ?
>=	A>=B	האם ערכו של A גדול או שווה לערכו של B ?
==	A==B	האם ערכו של A שווה לערכו של B ?
=!	A!=B	האם ערכו של A שונה מערכו של B ?

<u>אופרטורים שפועלים על ערכים בוליאנים</u>

קיימים מספר אופרטורים, שניתן להפעיל על ערכים מטיפוס boolean, ולקבל כתוצאה מפעולתם ערך חדש מטיפוס boolean.

! - אופרטור זה פועל על ערך בוליאני ומחזיר את ערכו ההופכי. כך למשל, אם מפעילים את האופרטור ! על ביטוי בוליאני שערכו true, מקבלים את הערך הבוליאני false, ולהפך.

& - אופרטור זה פועל על שני ערכים מטיפוס boolean ומחזיר ערך חדש מטיפוס boolean על פי הטבלה הבאה:

ערכו של הביטוי A	ערכו של הביטוי B	ערכו של הביטוי A & B
true	true	true
true	false	false
false	false	false
false	true	false

| - אופרטור זה פועל על שני ערכים מטיפוס boolean ומחזיר ערך חדש מטיפוס boolean על פי הטבלה

© 2007 Zindell Technologies, Ltd.

הבאה:

ערכו של הביטוי A	ערכו של הביטוי B	ערכו של הביטוי A | B
true	true	true
true	false	true
false	false	false
false	true	true

^ אופרטור זה פועל על שני ערכים מטיפוס boolean ומחזיר ערך חדש מטיפוס boolean על פי הטבלה הבאה:

ערכו של הביטוי A	ערכו של הביטוי B	ערכו של הביטוי A ^ B
true	true	false
true	false	true
false	false	false
false	true	true

&& - אופרטור זה פועל בדומה לאופרטור &. ההבדל ביניהם הוא שאם משתמשים ב-&&, ובמהלך החישוב של הביטוי שהאופרטור && קובע את ערכו (true או false) תוצאת הביטוי כולו כבר ידועה מראש אז חישוב המשך ערכו של הביטוי מופסק.

|| - אופרטור זה מבצע את פעולת ה-OR הלוגי כמו האופרטור |. בדומה לאופרטור &&, אשר שיפר את פעולתו של האופרטור &, כך גם האופרטור || משפר את פעולתו של האופרטור |. אם במהלך פעולת החישוב שמבצע

© 2007 Zindell Technologies, Ltd.

האופרטור |, התוצאה כבר מתבהרת באופן סופי אז פעולת החישוב מופסקת.

כאשר משתמשים בשני האופרטורים האחרונים שהוצגו, יש לשים לב לכך שאם הביטוי כולל משפטי השמה אז הם עלולים לא להתבצע. דוגמא לכך מהווה התכנית הבאה:

```
 class BeCare
{
    public static void main(String args[])
    {
        int num1,num2,num3,num4,sum;
        sum=99;
        num1=1;
        num2=2;
        num3=3;
        num4=4;

        if (num1+num2<num1 && num4>(sum=num3))
            System.out.println("\nHello Israel");
        System.out.println("\nsum="+sum);
    }
}
```

© 2007 Zindell Technologies, Ltd.

האופרטור הטרנרי

האופרטור היחידי שפועל על יותר מאופרנד אחד הוא האופרטור הטרנרי, אשר קרוי גם בשם אופרטור סימן השאלה. פעולתו של אופרטור זה ב-JAVA דומה לפעולתו בשפת התכנות C.

BooleanExpression ? expression1 : *expression2*

האופרטור הטרנרי, אשר ידוע גם בשם: אופרטור סימן השאלה, בודק את ערכו של הביטוי expression, אשר חייב להיות מטיפוס boolean. אם ערכו true אז האופרטור הטרנרי מחזיר את ערכו של expression1, ואם הוא false אז הוא מחזיר את ערכו של expression2. שני הביטויים (גם expression2 וגם expression1) חייבים להיות מאותו טיפוס.

תכנית הדוגמא הבאה מציגה את פעולתו של האופרטור:

```
class Trenary
{
    public static void main(String args[])
    {
        float num1,num2,max;
        num1=14;
        num2=4;
        max=(num1>num2)?num1:num2;
        System.out.println("max="+max);
    }
}
```

© 2007 Zindell Technologies, Ltd.

אופרטור ההשמה

אופרטור ההשמה ב-JAVA הוא הסימן =, כמו בשפת התכנות C. בצד שמאל של האופרטור יש למקם את שמו של המשתנה, ובצד ימין של האופרטור יש למקם ביטוי. ערכו של הביטוי שבצד ימין יוכנס אל תוך המשתנה שרשום בצד שמאל.

לדוגמא:

;num1 = num2 + num3

במשפט השמה זה יוכנס אל תוך המשתנה num1 ערכו של הביטוי: num2+num3.

אופרטורים מורכבים לביצוע השמה

כאשר אופרטור בינרי פועל על משתנה ועל ביטוי נוסף, ותוצאת פעולתו מוכנסת אל תוך אותו משתנה שעליו הוא פעל, אז ניתן לכתוב את הפעולה כולה באופן מקוצר.

בהנחה שהאופרטור OP מייצג אופרטור מתמטי בינרי, ונתון משפט ההשמה הבא:

VariableName = variableName OP expression

ניתן יהיה לרשום אותו באופן המקוצר הבא:

VariableName OP = *expression*

דוגמא:

את משפט ההשמה הבא: num = num + (4*sum) ניתן גם לכתוב באופן הבא: num += 4*sum.

פעולת קיצור זו ניתן לבצע על כל אחד מהאופרטורים הבינריים שהכרנו, ובדרך זו לקבל את אופרטורי ההצבה הבאים: +=, -=, *=, /= וכדומה.

הערות כלליות בנוגע לטיפוס הערך של ביטוי

בחלק זה, אסב את תשומת ליבך למספר כללים חשובים שיש להכיר כאשר עוסקים בביטויים ב-JAVA.

© 2007 Zindell Technologies, Ltd.

טיפוס הערך של ביטוי שכל מרכיביו הם ערכים מספריים שלמים תמיד תהיה מטיפוס int, אלא אם אחד ממרכיבים מטיפוס long (במקרה זה הביטוי יהיה מטיפוס long). ערכו של ביטוי שכל מרכיביו ערכים מספריים שלמים לעולם לא יוכל להיות מטיפוס char או byte (אלא אם בוצע casting לביטוי כולו). גם אם כל מרכיביו של הביטוי מטיפוס byte, ערכו של הביטוי כולו יהיה מטיפוס int.

ניתן לבצע casting (המרה של טיפוס הערך של ביטוי לטיפוס אחר) באופן מכוון כמו ב-C. כל שיש לעשות הוא לרשום לפני הביטוי, שאת טיפוסו רוצים לשנות, את הטיפוס החדש בתוך סוגריים. כך למשל, אם num הוא מטיפוס long ורוצים לקבל את ערכו כך שיהיה מטיפוס int כל שיש לעשות הוא לרשום את הביטוי הבא: (int)num. בביצוע casting זה המשתנה נותר עדיין באותו טיפוס (טיפוס המשתנה num לא משתנה, ונותר long). מה שמשתנה הוא טיפוס הערך של הביטוי כולו. הערך של (int)num הוא ערך מטיפוס int.

בניגוד 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.

```
class CastingDemo
{
  public static void main(String args[])
  {
    byte b=1,a=0,c;
    int i=0;
    long l=0;
    int num=0;
    num=num+1;
```

© 2007 Zindell Technologies, Ltd.

```
System.out.println("b="+b+"  i="+i+"   l="+l);

l=b+i;

System.out.println("l=i+b");

System.out.println("b is a byte type and i is an integer type ");

System.out.println("so the type of b will be changed from byte ");

System.out.println("to int and the all right expression will be int.");

System.out.println("Later, the right expression will be changed from ");

System.out.println("int to long since l is long !");

System.out.println("l="+l);

System.out.println("\na b and c are byte type");

//c=a+b;

System.out.println("c=a+b is ERROR ! since the right expression");

System.out.println(" is an int type and since c is a btye type ");

System.out.println("without using a forced casting on the right expression ");

System.out.println("to become byte instead of int it will be a mistake ");

System.out.println(" since the right expression is a bigger type");

System.out.println("(int bigger than byte !)");

// c=(byte)a+b;

System.out.println("\nc=(byte)a+b   is still an ERROR !");

c=(byte)(a+b);
```

© 2007 Zindell Technologies, Ltd.

```
    System.out.println("\nc=(byte)(a+b) is O.K.  !");

    System.out.println("c="+c);

  }

}
```

ערכו של ביטוי שיש בו מרכיבים מטיפוס float, אך לא מטיפוס double הוא float. די בכך שאחד המרכיבים יהיה מטיפוס double כדי שערך הביטוי כולו גם יהיה מטיפוס double. קבוע ממשי בברירת המחדל שלו נחשב לביטוי מטיפוס double, אלא אם הוספה לו בסופו האות F.

המרת טיפוס מספרי שלם ל-float, והמרת float ל-double נעשית בהתאם לצורך באופן אוטומטי בדומה ל-C. המרות אלה מתרחשות, למשל, בעת הצבתו של ערך אל תוך משתנה מטיפוס ששונה מטיפוס הביטוי המושם. במקרים אלה, פעולת ה-casting, שמתבצעת באופן אוטומטי, משנה את טיפוס הערך של הביטוי המושם כך שיהיה זהה לטיפוס המשתנה. ההבדל שקיים בין JAVA ל-C קיים באותם מקרים שבהם יש צורך בביצוע casting בכיוון ההפוך. ה-casting לא מתבצע באופן אוטומטי, ויש לבצעו באופן יזום.

דוגמא:

```
float fNum = 1.0f;
```

אם לא היה מבוצע casting באופן יזום לערך המספרי 1.0 הייתה מתקבלת שגיאת קומפילציה. 1.0 הוא מטיפוס double (ברירת מחדל), ולכן, אם לא מוסיפים את האות f לסופו, ההשמה שנעשה ניסיון לבצעה היא של ערך מטיפוס double אל תוך משתנה מטיפוס float. כיוון שזוהי השמה של טיפוס גדול אל תוך משתנה מטיפוס קטן יותר תופיע שגיאת קומפילציה.

דוגמא נוספת:

```
float fNum = (float)(f2+1.0);
```

תחת ההנחה שהמשתנה f2 הוא מטיפוס float.

גם בדוגמא זו חייבים לבצע את ה-casting המכוון. הביטוי f2+1.0 הוא ביטוי מטיפוס double כיוון שמספר ממשי

© 2007 Zindell Technologies, Ltd.

(המספר 1.0) תמיד יהיה מטיפוס double כברירת מחדל, אלא אם בוצע לו casting.

אופרטורים שקיימים ב-C\C++ ולא קיימים ב-JAVA

האופרטורים שקיימים ב- C\C++, ולא קיימים ב-JAVA הם:

Sizeof	אופרטור שמשמש ב-C למציאת מספר הבתים שמשמשים טיפוס/משתנה
*	אופרטור שמשמש ב-C לקבלת הערך שנמצא בכתובת מסוימת
&	אופרטור שמשמש ב-C לקבלת הכתובת של משתנה/אובייקט
::	אופרטור שמשמש ב-C++ לייחוס הגדרתה של פונקציה למחלקה מסוימת
,	אופרטור זה משמש לביצוע הפרדה בין ביטויים. אופרטור הפסיק לא קיים ב-JAVA - למעט בתוך הביטויים הראשון והשלישי שמופיעים בתוך במשפט לולאת ה-for.

כללי הקדימויות של אופרטורים

האופרטורים מחולקים לקבוצות שמגדירות את קדימותן. האופרטורים מתבצעים על פי סדר הקדימות שמוגדר על ידי קבוצות אלה. בין האופרטורים ששייכים לאותה דרגת קדימות סדר הפעלתם יהיה בדרך כלל משמאל לימין.

בכל מקום שבו יש לך ספק בנוגע לסדר הקדימות של האופרטורים כדאי להוסיף סוגריים.

הטבלה הבאה מציגה את סדר הקדימות של האופרטורים:

הקבוצה	סדר הקדימות
[ ] . () קריאה להפעלת מתודה	מימין לשמאל

(data type) new - + -- ++ ~ !	מימין לשמאל

© 2007 Zindell Technologies, Ltd.

% / *	משמאל לימין
- +	משמאל לימין
<<< << >>	משמאל לימין
instanceof >= <= > <	משמאל לימין
!= ==	משמאל לימין
&	משמאל לימין
^	משמאל לימין
|	משמאל לימין
&&	משמאל לימין
||	משמאל לימין
?:	מימין לשמאל
-= += %= /= *= = |= ^= &= <<<= <<= >>= =	מימין לשמאל

משפטים פשוטים ומורכבים Simple & Compound Statements

בדומה ל- C\C++ גם ב-JAVA יש משפטים פשוטים ומורכבים. משפט פשוט הוא הוראת ביצוע בודדת (פקודה אחת) שמשתרעת על פני שורה אחת (בדרך כלל) ומסתיימת בנקודה-פסיק, ; .

לדוגמא:

```
System.out.println("Hello Israel");
```

משפט מורכב הוא קבוצה של משפטים פשוטים (יש בה משפט אחד או יותר) אשר תחומים בתוך סוגריים מסולסלות.

דוגמא:

© 2007 Zindell Technologies, Ltd.

```
{
  System.out.println("Hello Israel");
  System.out.println("Hello USA");
  System.out.println("Hello JAPAN");
}
```

כאן גם המקום לציין, שגם ביטוי בודד בתוספת נקודה-פסיק, ; , נחשב למשפט פשוט.

דוגמא:

```
num1+num2+num3;
```

בכל מקום, שבו ניתן על פי כללי השפה למקם משפט אחד פשוט, ניתן גם למקם משפט מורכב. כדאי לזכור כלל זה בהמשכו של הספר.

משפטי בקרה - Control Structure

משפטי תנאי

משפטי התנאי ב-JAVA דומים למשפטי התנאי ב-C למעט הדרישה לרשום בתור תנאי ביטוי מטיפוס boolean. ב-C ניתן היה לרשום בתור תנאי כל ביטוי, והכלל שקבע הייתה שאלת היותו של הביטוי שווה ל-0 או לא. ב-JAVA חייבים לרשום בתור תנאי ביטוי מטיפוס boolean.

קיימים שני סוגים בסיסיים של משפטי תנאי:

<u>משפט ה-if הפשוט</u>

```
if (booleanExpression)
        statement
```

© 2007 Zindell Technologies, Ltd.

משפט ה- if..else

if (*booleanExpression*)

statement

else

statement

אם תבין/תביני כי משפט יכול להיות גם משפט מורכב וגם משפט פשוט, וכי משפט התנאי נחשב למשפט תוכל/תוכלי להבין כיצד ניתן לייצור משפטי תנאי מורכבים יותר.

משפט האופרטור הטרנרי (אופרטור סימן השאלה)

משפט האופרטור הטרנרי פועל באופן הבא:

(*booleanExpression*)? *Statement1* : *statement2*

אם ערכו של הביטוי הלוגי true אז יבוצע המשפט הראשון, ואם ערכו false אז יבוצע המשפט השני. כיוון שמשפט האופרטור הטרנרי גם נחשב למשפט, ניתן לקנן משפט אופרטור טרנרי בתור משפט פשוט שמופיע במסגרתו של משפט אופרטור טרנרי אחר.

משפט ה- switch

switch(*expression*)

{

case *literal*:

statement;

break;

© 2007 Zindell Technologies, Ltd.

```
    case literal:
        statement;
        break;
    default:
        statement;
}
```

משפט ה-switch ב-JAVA פועל כמו ב-C, למעט הדרישה לרשום בתור *expression* ביטוי מטיפוס char, integer, short או byte בלבד. אחרי כל case חייב להופיע קבוע (literal). משתנה לא יוכל להופיע אחרי המילה case.

משפט לולאת ה - for

להלן האופן שבו יש לרשום את משפט ה-for:

```
For (initialization ; booleanExpression ; nex-iteration-setup)
    Statement;
```

ניתן להגדיר בתוך ה-*initialization* משתנה. במקרה כזה אורך חייו של המשתנה יוגבל רק לאותה לולאה.

ה-*statement* יכול גם להיות משפט פשוט, משפט מורכב (בלוק) וגם משפט לולאת for (במקרה האחרון נקבל לולאה מקוננת). בלולאת ה-for ניתן לשלב את אופרטור הפסיק בדומה ל-C\C++.

דוגמא ללולאת for:

© 2007 Zindell Technologies, Ltd.

```
for (int index=0; index<100; index++)
{
        System.out.println("index="+index);
}
```

משפט לולאת ה – for המקוצרת

החל מ 1.5 java ניתן לכתוב לולאת for אשר מתייחסת לאיברי מערך / collection באופן מקוצר.רושמים את משפט ה-for באמצעות שני ביטויים. הביטוי הראשון הוא הצהרה על המשתנה שלתוכו ייכנסו כל אחד מערכי ה-collection/מערך תוך כדי פעולת הלולאה. הביטוי השני הוא ה-reference למערך/collection שבו מדובר. בין שני הביטויים יש למקם ':'.

הדוגמא הבאה מציגה שימוש בסיסי בלולאת ה-for המקוצרת:

```
int vec[] = {1,3,5,7};
for (int a: vec)
{
          sum += a;
}
System.out.println(sum);
```

דוגמא זו מדפיסה למסך את סכום המספרים שמופיעים במערך.

© 2007 Zindell Technologies, Ltd.

משפט לולאת ה - while

while (*booleanExpression*)

statement

משפט לולאת ה-while ב-JAVA דומה למשפט לולאת ה-while בשפות אחרות כגון C\C++. ההבדל בכתיבת לולאה זו בין JAVA לבין C\C++ הוא שב-JAVA יש לרשום בתור תנאי ביטוי מטיפוס boolean. בלולאת ה-while מתבצע ה-*statement* שוב ושוב כל עוד התנאי של הלולאה true. אם כבר בפעם הראשונה שבה הביצוע של התכנית מגיע ללולאה התנאי false אז המשפט שעליו הלולאה פועלת לא יתבצע אפילו לא פעם אחת.

משפט לולאת ה - do..while

do

Statement

While (booleanExpression);

משפט לולאת ה-do..while ב-JAVA דומה למשפט לולאת ה-do..while בשפות אחרות כגון C\C++. ההבדל בכתיבת לולאה זו בין JAVA לבין C\C++ הוא שב-JAVA יש לרשום בתור תנאי ביטוי מטיפוס boolean. בלולאת ה-do..while ב-JAVA (כמו ב-C\C++) מתבצע המשפט לפחות פעם אחת. כאשר הביצוע של התכנית מגיע ללולאת do..while הוא מבצע את המשפט בפעם הראשונה שלו, ולאחר מכן בודק את התנאי. אם התנאי false אז הלולאה נפסקת (למעשה היא טרם התחילה) ואם התנאי true אז הביצוע חוזר ומבצע את המשפט פעם נוספת ואחר כך שוב ניגש ובודק את התנאי, וחוזר חלילה.

© 2007 Zindell Technologies, Ltd.

משפט ה-continue

הפעלת הפקודה continue גורמת לכך שהאיטרציה הנוכחית מפסיקה, ואיטרציה חדשה של הלולאה מתחילה. את הפקודה continue ניתן למקם בתוך כל אחת משלושת סוגי הלולאות.

אם משפט ה-continue מופיע בתוך לולאת for אז החלק שמקדם את משתנה הבקרה בלולאה יבוצע לפני שהאיטרציה תתחיל מחדש. מייד לאחר פעולת הקידום, ולפני התחלתה של האיטרציה החדשה התנאי של הלולאה ייבדק שוב, ורק אז תתחיל האיטרציה החדשה.

אם משפט ה-continue מופיע בתוך לולאת while או בתוך לולאת do..while אז הביצוע יעבור ישר אל התנאי של הלולאה.

משפט ה-break

הפעלת משפט ה-break גורמת ליציאה מהלולאה (שבירת הלולאה והפסקתה). אם הלולאה ש"נשברה" פנימית ללולאה אחרת אז היציאה תהיה רק מהלולאה הפנימית. את פקודת ה-break ניתן למקם בתוך כל אחת משלושת סוגי הלולאות, וגם בתוך משפט ה-switch. פעולת משפט ה-break כשהוא בתוך משפט ה-switch תגרום להפסקת פעולתו של משפט ה-switch.

משפטי ה- break וה- continue בתוספת תווית

ניתן להוסיף תווית (label) אל תוך קוד המקור כדי לציין מיקום מסוים בתכנית. מיקום זה יכול להיות, למשל, תחילתה של לולאה. כאשר עושים זאת ניתן להורות באמצעות משפט ה-break או ה-continue להפסיק את פעולתה של לולאה מסוימת (שימוש ב-break) או לסיים את האיטרציה של לולאה מסוימת ולהמשיך הלאה לאיטרציה הבאה אם אמורה להיות (שימוש ב-continue).

כדי לסמן מיקום מסוים במסמך באמצעות תווית יש לרשום את התווית בצירוף נקודותיים במיקום שרוצים לסמן. לאחר מכן, ניתן יהיה להתייחס אליה באמצעות ציון שמה אחרי הפקודה break או continue. התכנית הבאה מדגימה את

© 2007 Zindell Technologies, Ltd.

השימוש בתווית.

```
class LabelDemo
{
  public static void main(String args[])
  {
    int index1, index2;
    vodka:for(index1=0; index1<10; index1++)
      for(index2=0; index2<10; index2++)
      {
        System.out.println("index1="+index1+" index2="+index2);
        if (index1==index2)
          continue vodka;
      }
  }
}
```

משפט השמה - Assignment Statement

בדומה ל-C\C++ גם ב-JAVA, משפט ההשמה מהווה ביטוי שערכו הוא הערך שהושם. משפט ההשמה מורכב מסימן השוויון כשמימינו הביטוי שערכו מושם אל תוך המשתנה. את המשתנה יש לכתוב בצידו השמאלי של סימן השוויון.

variableName = *expression*;

כיוון שכל משפט השמה נחשב לביטוי, שערכו הוא הערך המושם, ניתן לשרשר משפטי השמה באופן הבא:

© 2007 Zindell Technologies, Ltd.

variableName = variableName2 = variableName3 = ... = expression;

לדוגמא:

;num1 = num2 = num3 = 72

בדוגמא זו הערך 72 הוכנס אל כל אחד מהמשתנים שצוינו. סדר פעולתם של סימני השוויון בעת ביצוע ההשמות הוא מימין לשמאל (כפי שהוסבר בחלק שדן בנושא האופרטורים).

ניתן לנתח את השורה הזו באופן הבא:

num3=72 – זהו משפט השמה אשר גורם לכך שאל תוך המשתנה num3 יוכנס הערך 7. בתור משפט השמה זהו גם ביטוי שערכו 7, ולכן, ניתן להתייחס אל השורה: num2=num3=72 כאל משפט השמה שבו מוכנס הערך של הביטוי num3=72 (ערכו 72) אל תוך המשתנה num2. באותו אופן ניתן גם להסביר כיצד מוכנס הערך 72 אל תוך המשתנה num1.

טווח ההכרה של משתנים - Local Variables Scope

משתנה מקומי מוגדר בתוך מתודה (פונקציה). טווח ההכרה שלו הוא מותך המתודה עצמה בלבד, ואורך החיים שלו הוא כמשך פעולת המתודה. מן הרגע שבו נוצר המשתנה המקומי, ועד לתום חייו, לא ניתן לייצור משתנים מקומיים נוספים באותו שם. לא ניתן לעשות זאת – אפילו לא בתוך בלוקים פנימיים. כאן המקום לציין שניתן להגדיר משתנה מקומי בתוך בלוק. במקרה כזה, אורך חייו של המשתנה יהיה כאורך חייו של הבלוק.

משתנים גלובליים כמו שיש ב-C\++C לא קיימים ב-JAVA. ב-JAVA אין משתנים גלובליים כיוון ש ב-JAVA כל התכנית מורכבת ממחלקות בלבד.

בדומה ל- C\++C, גם ב-JAVA ניתן לשלב בתוך הביטוי הראשון בלולאת ה-for הצהרה על משתנה חדש. ההבדל בין JAVA ל- C\++C הוא שב-JAVA המשתנה חי רק בתוך הלולאה. ב-C וב-++C (בחלק מהקומפיילרים) המשתנה שמוצהר עליו בתוך לולאת ה-for ממשיך להתקיים גם אחריה.

© 2007 Zindell Technologies, Ltd.

השימוש ב-enum

אם בעבר אופן יצירתם של קבועים היה באמצעות הגדרתם כמשתנים שסומנו ב-final הרי שהחל מ-Java 1.5 ניתן להשיג את אותה מטרה תוך שימוש בהגדרתו של enum, שמהווה אוסף של שמות קבועים אשר ניתן להשתמש בהם בתכנית. כדי להגדיר enum יש להשתמש במילה השמורה enum ומייד אחריה לכתוב את שמו של ה enum אשר מגדירים ולאחריו לפרט בתוך סוגריים את הערכים הקבועים האפשריים:

```
enum enumName {CONSTANT_1, CONSTANT_2, CONSTANT_3...}
```

הדוגמא הבאה מציגה יצירתו של enum פשוט לצורך תיאור חודשי השנה.

```
public class EnumSample
{
    static enum Month {January, February, March, April, May, June, July, August, September,
                        October, November, December};
    public static void main(String args[])
    {
        Month first = Month.January;
        Month last = Month.December;
        System.out.println("first month is "+first);
    }
}
```

את ה-enum לא ניתן להגדיר כמשתנה מקומי. זו הסיבה שבדוגמא זו הוא הוגדר כמשתנה סטטי. להסברים נוספים נא להמתין לפרק 4.

השימוש ב-enum שקיים ב-Java מאפשר ביצוען של פעולות רבות נוספות אשר תיסקרנה בפרקים הבאים.

© 2007 Zindell Technologies, Ltd.