גירסה 1.00 – 17.12.2004

Little Smalltalk

ניר אדר

Nir Adar
Email: underwar@hotmail.com
Home Page: http://underwar.livedns.co.il

אנא שלחו תיקונים והערות אל המחבר.

תוכן עניינים

מבוא

Little Smalltalk (בקיצור LST) היא שפה מונחית עצמים משנות ה-80.

השפה פותחה על ידי Timothy Budd מאוניברסיטת אוריגון. השפה היא חופשייה (freeware).

מטרות הפיתוח:

- מפענח קרוב ככל הניתן לתקן המוגדר של השפה.
- מפענח הכתוב בשפת C להבטחת פורטביליות מקסימלית.

רעיונות ועקרונות השפה

- בשפת Smalltalk כל יישות הינה אובייקט – אין דרך לייצר אלמנטים שאינם אובייקטים. הכוונה ב-"כל ישות" היא ממש כל יישות – אפילו מספרים, תווים וכדומה.
- מצב האובייקט: לכל אובייקט יש **מצב**. לדוגמא: אובייקט המייצג חשבון בנק יכול להכיל את סכום הכסף שבחשבון. אובייקט מסוג מספר יודע איזה מספר שמור כרגע בו. כל אובייקט דורש כמות מסויימת של זיכרון עבור **המשתנים הפנימיים** שלו (instance variable).
- רק האובייקט עצמו יכול לגשת אל המשתנים הפנימיים שלו. משתנים פנימיים אלו יכולים להיות אובייקטים בעצמם.
- הודעות: ביצוע חישובים בשפת Little Smalltalk: על ידי שליחת **הודעות** לאובייקטים. לכל אובייקט יש **שיטות** (methods) המגדירות איך אובייקט מגיב להודעות שונות.
- חלוקה למחלקות: מכיוון שיכול להיות מספר עצום של אובייקטים, איננו מגדירים את השיטות והמשתנים הפנימיים של כל אובייקט בנפרד. האובייקטים מחולקים למחלקות - כל אובייקט שייך למחלקה אחת בדיוק. המחלקות עצמן מאורגנות בעץ הורשה. השיטות והמשתנים הפנימיים של כל אובייקט נקבעים לפי המחלקה אליה הוא שייך.
- הורשה יחידה: ב-LST קיימת ירושה יחידה בלבד – כל מחלקה נורשת ממחלקה אחת אחרת.
- זהות: לכל אובייקט זהות משלו. כאשר אנו משנים משתנה פנימי השייך לאובייקט אחד ממחלקה מסויימת, משתנה פנימי זה אינו משתנה עבור האובייקטים האחרים השייכים לאותה מחלקה.

הרצת Little Smalltalk

צורת העבודה עם Little Smalltalk היא בד"כ זו: ראשית נכתוב את קבצי ההוראות של התוכנית שלנו. לאחר מכן נטען את סביבת Little Smalltalk. כאשר נעלה את Smalltalk, לאחר טעינת הסביבה, נקבל שורת prompt שאומרת שהשפה מוכנה לקבל פקודה מאיתנו.
מתבצעת לולאת read-evaluate-print: השפה קולטת שורה עם ביטוי אחד או יותר מהמתכנת, השפה מהדרת את הביטויים לשפת מכונה, מריצה אותם ושולחת אל אובייקט התוצאה את ההודעה print.

כדי לטעון את קבצי ההוראות שהכנו, נכתוב את שם הקובץ במרכאות בודדות, ולאחריו את האות r. שמות קבצים הינם בהתאם למקובל ב-DOS – עד 8 אותיות לפני הנקודה, ו-3 אחריה. לדוגמא, כדי לטעון את הקובץ myfile.st, נכתוב:

```
'myfile.st' r
```

דרך נוספת הינה על ידי הפקודה הבאה:

```
File new fileIn: 'myfile.st'
```

חלק מהמימושים של שפת Little Smalltalk תומכים רק בצורה השניה, שהיא הצורה הרשמית המוגדרת על ידי השפה.

כדי לבצע סקריפט נשתמש בסימני ה-redirection, לדוגמא:

```
st < myfile.st > output.txt
```

כברירת מחדל, פקודה מתבצעת בסביבת העבודה מיד כשהמשתמש לוחץ Enter. אם נרצה לכתוב פקודה שתתפרס על מספר שורות, נכתוב בסוף כל שורה את הסימן \ (בדומה לסימן _ בשפת Visual Basic).

סיום העבודה ב-Little Smalltalk על ידי תו סיום הקלט - EOF. ב-DOS נשיג תו זה על ידי לחיצה על צירוף המקשים Ctrl+Z.

דוגמא ראשונה

נביט בדוגמא הבאה המציגה את המספרים כאובייקטים.

```
> 3+2
5
```

בתגובה לביטוי 3+2 פלטה המערכת את התוצאה 5.

מה תהיה תוצאת החישוב הבא?

```
> 1+2*3
```

התשובה: התוצאה תהיה 9.

הסבר: בשפת Smalltalk 1 הוא אובייקט. אנחנו שלחנו אליו את ההודעה + עם הארגומנט 2. התגובה להודעה זו הוא האובייקט 3. לאחר מכן האובייקט 3 מקבל את ההודעה * עם הארגומנט 3, ומחזיר 9.

האובייקטים 1, 2 ו-3 יודעים איך להגיב להודעות כגון חיבור וכפל מכיוון שכולם מספרים שלמים (עבורם הוגדרו הודעות אלו).

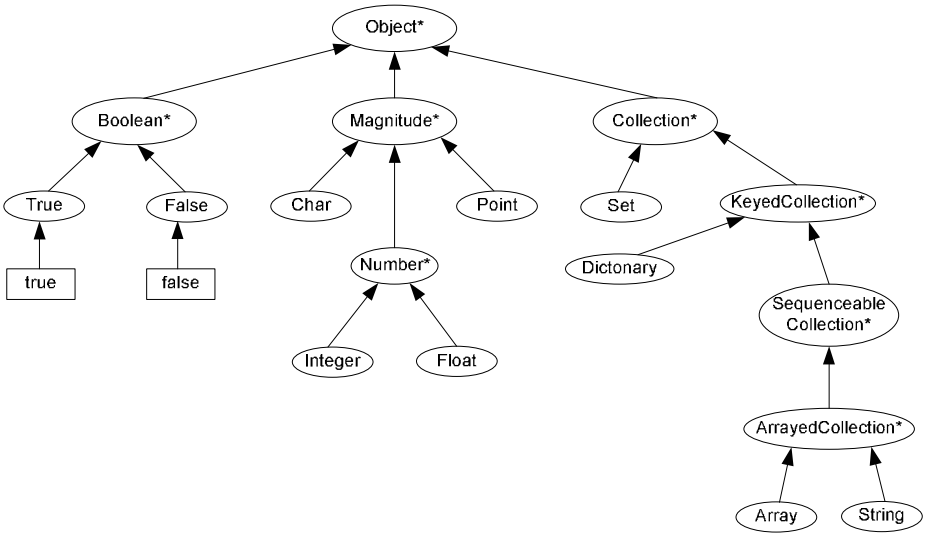
האובייקטים 1, 2 ו-3 הינם **מופעים** של המחלקה Integer.

היררכית המחלקות בשפת Little Smalltalk

נציג כעת את היררכית המחלקות של שפת Little Smalltalk, המתאימה לגירסה 1 של השפה. ההיררכיה השתנתה מעט בין גירסה לגירסה, אולם הרעיון נשאר. ההיררכיה המוצגת הינה חלקית בלבד.

סימונים:

היררכיה חלקית של המחלקות ב-Little Smalltalk:



אובייקטים שנוצרים אוטומטית עם עליית השפה: true, false, nil ועוד שנראה בהמשך. true, false, nil הם סינגלטונים – כלומר העצמים היחידים שנוצרים מהסוג של אותה מחלקה. (עובדה זו איננה נאכפת על ידי השפה, וזו אחריות המתכנת לא ליצור, למשל, אובייקטים נוספים מסוג True).

Object היא מחלקת הבסיס לכל היררכית המחלקות. מחלקה זו כוללת מספר שיטות השימושיות עבור כל המחלקות, לדוגמא – הדפסת האובייקט (כברירת מחדל פונקצית ההדפסה מדפיסה את שם המחלקה – אולם ניתן לחפוף שיטה זו כרצוננו).

כל מחלקה היא בעצמה אובייקט. המחלקה של Integer, למשל, היא Class.

מרכיבי השפה הבסיסיים

מזהים

מזהה עבור משתנים: שם משתנה מתחיל באות קטנה, ולאחריה אותיות (גדולות או קטנות) ומספרים.
מזהה עבור מחלקה: שם מחלקה מתחיל באות גדולה, ולאחריה אותיות (גדולות או קטנות) ומספרים.
משתנה גלובלי: ייכתב לעיתים גם כולו באותיות גדולות.
בניגוד לשפות כגון C, הסימן _ אינו תו חוקי במזהה!

עקב באג ב-Little Smalltalk, דרישות אלו לגבי המזהים אינן נאכפות וזוהי אחריות המשתמש לדאוג להן.

משתנים בשפת Little Smalltalk

חלוקת המשתנים השונים בשפה:

- גלובליים.
- פנימיים למחלקה – instance variable.
- פנימיים לפונקציה.

כל משתנה המוגדר בתוך ה-Command Line הוא גלובלי.
כל משתנה במתודה שלא הוגדר בצורה מפורשת כלוקלי, הוא משתנה גלובלי.

שפת Little Small Talk היא weakly typed – איננו מציינים את הסוג של כל משתנה.
נראה למשל הגדרת משתנה והשמת ערך לתוכו:

```
> i <- 2
```

אופרטור החץ משמש כאופרטור ההשמה בשפת LST. כאשר כתבנו את השורה לעיל, נוצר משתנה גלובלי בשם i. המשתנה הוא למעשה משתנה התייחסות (reference) לאובייקט 2. (האובייקט 2 הוא יחיד).

ניתן בהמשך על ידי הצבה נוספת לשנות את הערך ש-i מתייחס אליו (ואף את הסוג ש-i מתייחס אליו):

```
> i <- 'hello'
```

כל אובייקט שאיננו מבצעים לו השמה באופן מפורש מכיל את הערך nil.
למשל, אם בשלב הנוכחי, אחרי שהגדרנו רק את האובייקט i, ננסה לגשת אל האובייקט j, נקבל שגיאה.

מכיוון שהשפה היא weakly typed, כאשר פונקציה מקבלת ארגומנטים היא לא יודעת בפועל מה הוא סוג הפרמטרים שהיא מקבלת בפועל. אם נכתוב פונקציה ונרצה להיות בטוחים שהיא מקבלת ערכים מסוג מסויים, נצטרך לעשות בדיקה ידנית בקוד שלנו. הקומפיילר, בשונה מזה של C++, אינו מספק לנו אמצעי הגנה מפני העברת ערכים מסוג לא נכון לפונקציה.

הערות

הערות ב-LST נכתבות בין גרשיים: " ".
ניתן לשלב הערות וקוד, אולם מומלץ להמנע מכך עקב באגים באינטרפרטר של LST.

ליטרלים

ליטרלים בשפת Little Smalltalk: מספרים, סמלים, מחרוזות, תווים, מערך בתים ומערך.

המספרים מתחלקים למספרים שלמים, ממשיים ושברים:

- **מספר שלם (Integer)**: לדוגמא 5 או 7.
- **מספרים ממשיים (Float)**: לדוגמא 3.14.
- **שברים (Fraction)**: לדוגמא 3/4.

תווים (Char): $A, $8, $$

מחרוזות (String) – תת מחלקה של ArrayedCollection בגירסה 1, תת מחלקה של ByteArray בגירסה 3): לדוגמא: 'hello, world'.

סמלים: #abc, #+++%

מערכים:

- #(this is an array)
- #(12 'abc' (another array))

המערך הראשון הינו מערך של סמלים. המערך השני מכיל איברים מסוגים שונים.

נפרט כעת עוד על כל אחד מהליטרלים בשפה.

מספרים

מספרים יכולים להיות שלמים, ממשיים או שברים.

הודעות אריתמטיות: המספרים מכירים את ההודעות: +, -, *, /.

הודעות של השוואה: <, >, =, =<, =>, ~= (שונה).

הודעות חלוקה: חלוקה (בשלמים) :quo, שארית :rem. לדוגמא:

```
> 8 quo: 2
4
```

פעולות לוגיות בין ביטים: :bitAnd ,:bitInvert ,:bitOr ,:bitXor ,:bitShift

דוגמא:

```
> 1 bitOr: 2
3
> 1 bitShift: 3
8
```

הודעות נוספות: :abs, floor, :gcd, :lcm, sqrt, squared, strictlyPositive, negative, positive ועוד.

הסוג Integer מיועד לשמור מספרים קטנים יחסית. (בגירסה הנוכחית של Little Smalltalk – מספרים הקטנים ממש מ-16384). אם נגדיר מספרים גדולים יותר, הם יוגדרו כמספרים מטיפוס Float. אולם, נשים לב להתנהגות הבאה: אם נגדיר משתנה בצורה הבאה:

```
i <- 32000
```

אזי סוג המשתנה יהיה Float.

לעומת זאת, אם נגדיר את המשתנה בצורה כזו:

```
i <- 2 * 16000
```

אזי סוג המשתנה יהיה LongInteger. מבחינת המתכנת מקרים אלו לא אמורים להפריע לרוב.

אם נקבל משתנה מסוג Float ונרצה להמיר אותו לשלם, נשתמש בהודעה integerPart, לדוגמא:

```
> 32000 integerPart
032000
```

נשים לב שכתוצאה מבאג ב-Little Smalltalk מספרים מסוג LongInteger מופיעים לעתים עם 0 מוביל לפניהם.

תווים

ליטרלים מסוג תווים מסומנים על ידי תחילת של דולר ($).

הודעות של השוואה: <, >, =, =<, =>, =~.

הודעות נוספות:

- asInteger מחזירה את ערך ה-ASCII המתאים לאות.
- asString מחזירה מחרוזת באורך 1 שתוכנה הוא התו.
- isAlphabetic, isBlank, isDigit, isLowercase, isUppercase מחזירות true אם התנאי שהן בודקות מתקיים, ו-false אם לא.

מחרוזות

מחרוזות מזוהות על ידי טקסט המוקף בגרשיים בודדות. לדוגמא:

```
'Hello, World'
```

אם נרצה לכלול גרשיים במחרוזת, נשתמש פעמיים בגרש:

```
> '9 o''clock'
9 o'clock
```

שלמים בשפת Little Smalltalk

נדגיש כעת מעט יותר את ההבדל בין משתני int בשפות כגון C++ לבין האובייקטים המייצגים מספרים בשפת Little Smalltalk.

נניח בשפת C++ נכתוב את ההגדרה הבאה:

```
int i = 2, j = 2;
```

ייווצרו שני משתנים מסוג int שכל אחד מהם הוא container המכיל את המספר 2. זכרון המחשב ייראה כך:

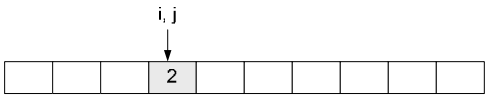
הביטוי `(i == j)` הינו **השוואת מצב**. ערכו בדוגמא זו יהיה true.

הביטוי `(&i == &j)` הינו **השוואת זהות**. ערכו בדוגמא זו יהיה false.

לעומת זאת, בשפת Little Smalltalk כל מספר הוא אובייקט, והמשתנים הם רק התייחסות אל האובייקט שהוא המספר. נכתוב את ההצהרות הבאות:

```
i <- 2
j <- 2
```

הזכרון במקרה זה ייראה כך:



הביטוי `(i = j)` הינו **השוואת מצב**. ערכו בדוגמא זו יהיה true.

הביטוי `(i == j)` הינו **השוואת זהות**. ערכו בדוגמא זו יהיה true, בניגוד למתרחש בשפת C++.

self, super

self ו-super הינם שני משתנים מדומים הקיימים בשפה. self מספק לנו התייחסות אל האובייקט הנוכחי.
משמעותו דומה לזו של this בשפת C++ או C#.
super מחזיר התייחסות לאובייקט הנוכחי, כאילו הוא אובייקט מסוג מחלקת האב של האובייקט הנוכחי. (אנו צריכים את super כאשר, למשל, אנו רוצים לגשת למתודות של אובייקט האב, אולם מחלקת הבן ביצעה להן חפיפה ולכן לא ניתן לגשת אליהן ישירות).

הודעות

קיימים מספר סוגי הודעות בשפה. הודעות אונריות, הודעות בינאריות והודעות המבוססות על מילות מפתח.
הודעות אונריות פועלות על האובייקט עליהן הן מופעלות, ואינן מקבלות ארגומנטים נוספים.
הודעות בינאריות מתבססות על אחד מהאופרטורים הבנויים בשפת Little Smalltalk. הן מורכבות מאובייקט, אופרטור, ואובייקט נוסף.
הודעה המורכבת ממילות מפתח מכילה מילות מפתח, כך שלאחר כל מילת מפתח מופיעים נקודותיים וארגומנט שמועבר למילת המפתח. בדרך זו אנחנו מסוגלים להעביר לאובייקט הודעה המורכבת ממספר פרמטרים.

אופרטור אונרי:

```
> 3 negated
-3
```

אופרטור בינארי:

```
> 2 + 2
4
```

הודעה המורכבת ממילות מפתח:

```
> 5 between: 3 and: 7
true
```

במידה ויש ביטוי המורכב ממספר הודעות, ראשית נשלחות ההודעות האונריות, לאחר מכן ההודעות הבינריות ולבסוף ההודעות המבוססות על מילות מפתח.

לדוגמא:

נביט בקוד הבא:

```
> 2 + 2 negated
0
```

לעומת הקוד:

```
> (2 + 2) negated
-4
```

Cascades

נניח שיש לנו מחלקה המייצגת בן אדם ואובייקט Foo מסוג מחלקה זו. נניח כי בין היתר למחלקה קיימות ההודעה :name המשמשת לקביעת שם האדם, וכן :addr המשמשת לקביעת כתובתו.

צורת הקריאה הבאה תהיה שגיאה:

```
Foo name: 'Nir' addr: 'Haifa'
```

וזאת מכיוון ש-Little Smalltalk חושב שאנחנו שולחים הודעה בעלת שני פרמטרים, והודעה כזו לא קיימת. אם זאת, במקרים רבים נרצה לשלוח לאובייקט הודעות ברצף, אחת אחרי השניה.

לצורך כך נשתמש בהפרדה המכונה cascades. בין כל שתי הודעות נשים נקודה-פסיק (;). ההודעות ישלחו לאובייקט אחת אחרי השניה. בהמשך לדוגמא הקודמת, הכתיבה הבאה הינה חוקית:

```
Foo name: 'Nir'; addr: 'Haifa'
```

הערך המוחזר על ידי cascade הינו אובייקט התוצאה שנוצר על ידי הביטוי שלפני ה-; הראשונים.

נביט כעת במספר דוגמאות נוספות הקשורות לערך המוחזר מה-cascades:

```
> 2; * 1000; factorial; sqrt
2
```

לפני ה-; הראשונים קיים האובייקט 2. אנחנו מבצעים על 2 הכפלה ב-1000. התוצאה (2000) נזרקת כי אנחנו לא עושים בה שום שימוש. לאחר מכן factorial פועל שוב על האובייקט הראשון (2, ולא 2000 כפי שניתן לחשוב בטעות). גם תוצאה זו נזרקת. וכך הלאה. בסופו של דבר מוחזר 2 שהוא האובייקט הראשון.

```
> 2 * 1000; sqrt; squared;
2000
```

בדוגמא זו הפונקציות sqrt, squared פועלות על האובייקט 2000, שהוא אובייקט התוצאה של הביטוי שלפני ה-; הראשונים.

```
> ( 'Smalltalk' copyFrom:1 to: 5) size
5
```

הפונקציה copyFrom:to: מאפשרת לנו לקבל תת מחרוזת של מחרוזת נתונה. הפונקציה size מחזירה את גודלה (בתווים) של המחרוזת. בקטע קוד זה אין cascades.

```
> 'Smalltalk' copyFrom:1 to: 5; size
Small
```

בדוגמא זו, בדומה לדוגמא הראשונה, מוחזר האובייקט הראשון. התוצאה ש-size חישב נזרקת ולא נעשה בה שימוש.

```
> 'Smalltalk' at:4 put:$r; at:5 put:$t
Smarttalk
```

בדוגמא זו, שיכולה להפתיע, כן התבצעו שתי הפעולות על האובייקט. לפי הדוגמאות הראשונות, היינו יכולים לצפות שיוחזר הערך Smarltalk ולא Smarttalk. הסיבה להבדל היא שהפונקציה at:put: משנה ממש את האובייקט עליו היא פועלת, ולא יוצרת אובייקט חדש. לפיכך כאשר הפעלנו את הפונקציה השנייה היא ממש שינתה את אובייקט המחרוזת, ובסוף התבטאה הפעולה שלה. צורה זו של שימוש – היא הצורה שלשמה נועד ה-cascade – היא הצורה היחידה הנותנת לנו ערך מעשי לשימוש ב-cascades.

נסכם: אם ההודעות אינן משנות את האובייקט הראשון, אין להן השפעה כאשר משתמשים בהן ב-cascades.

בלוקים

בלוק הוא מונח בשפת Little Smalltalk הקרוב להגדרת פרוצדורות בשפות אחרות. לבלוק ישנם ארגומנטים, והוא מחשב ערך. בלוק מזוהה על ידי קטע המוקף בסוגריים מרובעות.

דוגמאות:

```
[7]                       "A procedure which always returns 7."
['Hello, World' print]    "A procedure while prints 'Hello, World'"
[:x | x+1]                "A procedure which returns its argument plus 1"
[:a | a print. a negated]
                          "Prints the value of its argument, and then
                          returns the negative of the argument"
```

כדי להריץ בלוק, ניתן להשתמש בהודעה value, לדוגמא:

```
> ['Hello, World' print] value
Hello, World
Hello, World
```

הרצת בלוק עם פרמטרים:

```
> [:x | x+1] value: 7
8
```

כל בלוק הינו אובייקט שהמחלקה שלו היא Block.

בלוקים הם ערכים ממחלקה ראשונה: ניתן לשמור אותם בתוך משתנים, ניתן להעביר אותם כפרמטרים לפונקציות וערך מוחזר של פונקציה יכול להיות בלוק. דוגמא:

```
> Twice <- [:x | 2 * x]
Block
> Twice value: 10
20
```

המשתנה x המוגדר על ידי :x הוא משתנה לוקלי של הפונקציה.

כמו כן, נניח ישנה מתודה המחזירה בלוק, ובלוק זה מכיל התייחסות למשתנה X השייך לאובייקט, כל פעולה על בלוק זה תמשיך לפעול על אותו ה-X. לא יבוצע חיפוש ברגע השימוש עבור ה-X המתאים (כלומר נניח נפעיל את הבלוק בהמשך במקום בו מוגדר משתנה נוסף בשם X, הפעולה לא תפעל עליו אלא על המשתנה X השייך לבלוק).
יותר מכך, נניח כי במתודה המחזירה בלוק קיים משתנה פנימי X, אז המשתנה הפנימי X ימשיך להיות קיים אחרי סוף המתודה, והבלוק ימשיך להתייחס אליו.

בלוק יכול להכיל מספר ביטויים שיופרדו על ידי נקודה. ערכו של הבלוק הינו הערך של הביטוי האחרון.
לדוגמא:

```
> [1. 2. 3] value
3
```

המחלקה Object

כאמור, המחלקה Object היא מחלקת הבסיס ממנה יורשות כל המחלקות.
נציג מספר מן הודעות בהן היא תומכת. בהמשך המסמך נתרכז במחלקה זו ביתר פירוט.

- **isKindOf:** – מחזיר האם האובייקט הוא מסוג המחלקה הנתונה כפרמטר, או האם הפרמטר נמצא אי שם בענף הירושה שלו.
- **isMemberOf:** – מחזיר האם האובייקט הוא בדיוק מהסוג שצוין.

לדוגמא:

```
> 2 isKindOf: Number
true
> 2 isMemberOf: Number
false
```

- **print, printString** - מתודה נוספת היא print המשמשת להדפסת מחרוזת המתארת את האובייקט. המתודה printString מחזירה מחרוזת המתארת את האובייקט. נשים לב שמחלקות רבות מבצעות חפיפה של מתודות אלו, על מנת להגדיר את הערך שאובייקטים מסוג זה ידפיסו.

הסתעפויות

הסתעפויות – המקבילות למשפטי if בשפות אחרות, מושגות בשפת Little Smalltalk על ידי שימוש באובייקטים.

המחלקות לייצוג משתנים בוליאניים, True ו-False, מכילות את ההודעה :ifTrue:ifFalse.

נדגים למשל בדיקת תנאי פשוטה, הבודקת האם ערכו של המשתנה i גדול מ-10, ומציגה הודעה בהתאם (אנחנו מניחים כי i הוא משתנה המוכר לסביבה):

```
(i > 10)                                          \
ifTrue: [                                         \
      'i is bigger than 10' print.                \
] ifFalse: [                                      \
      'i is smaller or equal to 10' print.        \
]
```

ההודעה > כאשר היא נשלחת לאובייקט מסוג Integer מחזירה לנו אובייקט מסוג False או אובייקט מסוג True. לאובייקט זה אנחנו שולחים את ההודעה :ifTrue:ifFalse וכפרמטרים אנחנו מעבירים בלוק לביצוע במקרה שתנאי ה-if הוא אמת, ובלוק שני לביצוע למקרה שתנאי ה-if הוא שקר.

נניח שנרצה לבדוק האם המשתנה i שייך לתחום הערכים בין 10 ל-100, נשאל זאת כך:

```
((i >= 10) and: [i <= 100])
```

התוצאה של הסוגריים הראשונים – אובייקט מסוג True או מסוג False. השתמשנו בדוגמא זו בהודעה :and אותה False ו-True יורשים מהמחלקה Boolean. נשים לב כי ההודעה :and מצפה לקבל פרמטר מסוג Block. הסיבה לכך היא שאם הערך של הביטוי הראשון הינו false, אז הביטוי השני כלל לא מחושב - Lazy Evaluation.

בדומה ל-:and, קיימת גם ההודעה :or. כמו כן קיימות ההודעות :ifTrue ו-:ifFalse בנפרד (כלומר, איננו חייבים בכל הסתעפות לציין מה עושים במקרה שהתנאי אמת וגם במקרה שהתנאי שקר, אפשר להסתפק באחד מהם), וכן ההודעה :ifFalse:ifTrue (זהה ל-:ifTrue:ifFalse אבל מספקת גמישות למתכנת בקביעת הסדר בתוכנית שלו).

לולאות

שימוש ב-Block

לולאות ממומשות על ידי הודעות לאובייקט מסוג Block. על ידי שימוש בהודעות whileTrue: ו-whileFalse: אנו מסוגלים לבצע קטע קוד, כל עוד ערך הבלוק הוא true (או false בהתאמה).

לדוגמא: נגדיר משתנה x שערכו 1. נאמר: "כל עוד ערכו של x קטן מ-7, הדפס את x וקדם אותו. בשפת Little Smalltalk קוד זה ייראה כך:

```
> x <- 1
1
> [x < 7] whileTrue: [x print. x <- x + 1]
1
2
3
4
5
6
```

נשים לב ש-[x < 7] הוא בלוק. קטע זה חייב להיות בלוק מכיוון שההודעה whileTrue: שייכת למחלקה Block. הפרמטר של הודעה זו הוא בלוק נוסף, המתבצע כל עוד הערך של הבלוק הראשון אמת.

באופן סימטרי קיימת ההודעה whileFalse: הפועלת כל עוד הערך שבבלוק הוא שקר.

- הערה: Little Smalltalk מממשת את הלולאה על ידי רקורסיה. אי לכך, אם נבצע לולאה עם מספר רב מדי של איטרציות (במימוש הנוכחי – פחות מ-1000 איטרציות) הסביבה תקרוס עקב מילוי המחסנית.

ההודעה whileTrue הקיימת גם ב-Block זהה להודעה whileTrue: nil. במקרה זה כל העבודה מתבצעת בבלוק הראשון. לדוגמא:

```
> A <- 1
> [A print. A <- A+1. A <= 5] whileTrue
```

Interval

מלבד Block ישנן מחלקות נוספות המציעות לנו מבני בקרה המתאימים ללולאות.

לפני שנציג אותן, נכיר את המחלקה Interval. סוג זה הוא סוג הנוצר על ידי שליחת הודעות למחלקה int, והוא סדרה של מספרים.

לדוגמא:

```
> 1 to: 5 by: 2
Interval ( 1 3 5 )
```

ההודעה יצרה את רצף המספרים בין 1 ל-5 כאשר אנו מתקדמים בדילוגים של 2.

מבני לולאה נוספים

קיימים מבני לולאה עבור אובייקטים מסוג Integer, עבור מערכים ועבור Interval. תחביר ההודעות:

```
anInteger timesRepeat: aBlock
anArray do: aBlock
anInterval do: aBlock
```

ניתן לראות שכל מבני לולאה אלו עושים שימוש באובייקט מסוג Block על מנת להגדיר את הקטע שיבוצע בגוף הלולאה. דוגמאות לשימושים:

```
5 timesRepeat: [ 'Hello, world' print ]

sum <- 0
#(90 60 85) do: [ :current | sum <- sum + current ]


( 1 to: 10 by: 2 ) do: [ :i | i squared print ]
```

מחלקות

יצירת מחלקות חדשות

המתכנת ב-Little Smalltalk יכול להגדיר בעצמו מחלקות חדשות. המחלקות ימוקמו בקובץ שיטען אל סביבת העבודה.
כל הצהרה על מחלקה תתחיל בכותרת המתחילה במילה Class, ולאחריה מספר מתודות.

הקוד עבור הדוגמא הבאה לקוח מן התרגולים של הטכניון בקורס "תכנות מונחה עצמים".

נביט בהצהרה הבאה:

```
Class Employee Object EmpName EmpNum
```

מילת המפתח Class אומרת כי אנו מגדירים כרגע מחלקה חדשה. שם המחלקה נקבע להיות Employee.
Object זו המחלקה ממנה Employee נורש. כל מחלקה חייבת להגזר ממחלקה אחרת. שגיאה נפוצה היא לשכוח להגדיר את המחלקה ממנה אנחנו גוזרים.
לאחר מכן מופיעים רשימת השמות של שדות המחלקה. ב-LST השדות הפרטיים הם בהרשאת Private תמיד. לא ניתן להגדיר משתנים בתור Public או Protected.

הגדרת מתודות המחלקה:

```
Methods Employee
```

שורה זו משמשת כדי להגיד שאנו כרגע מתחילים לממש את המתודות של המחלקה. Methods צריך להכתב עם M גדולה. השם אחרי Methods הינו שם המחלקה עבורה אנו ממשים את המתודות.

כל מתודה מוגדרת בצורה הבאה: שם המתודה מופיע לבד בשורה הראשונה, לאחר מכן מופיע הגוף של המתודה, ובסוף המתודה מופיע הסימן |. בסוף המתודה האחרונה מופיע הסימן] במקום הסימן |.

המתודה new היא ה-Ctor של השפה. לא ניתן ב-ST ליצור Ctor עם ארגומנטים. התפקיד של new הינו לאתחל את השדות השונים.

כל שורה במתודה ב-Little Smalltalk מסתיימת בנקודה, מלבד השורה האחרונה, שאינה חייבת להסתיים בנקודה.
החזרת ערך בסוף מתודה נעשה על ידי ^ וציון הערך שאנו רוצים שיוחזר.

לדוגמא:

```
Class Employee Object empName empNumber empSalary
Methods Employee
      new
            empName     <- 'No Name'.
            empNum      <- 0.
            empSalary   <- 0.
            ^self
|
      name
            ^empName
|
      name: aName
            empName <- aName
|
      empNum
            ^empNum
]
```

Ctor, אם איננו מציינים במפורש, מחזיר self (מקביל ל-this בשפת C++).
ניתן גם להחזיר ב-Ctor את self באופן מפורש לצורך הבהירות, כפי שעשינו בדוגמא זו.

מתודות יכולות להיות בעלות שם זהה לזה של המשתנים הפנימיים, אולם הן אינן חייבות להיות עם שמות כאלה. בדוגמא שלנו – empNum הינה מתודה עם שם זהה למשתנה הפנימי. בתוך המתודה כל שימוש בשמות המשותפים למתודות ושדות יתייחס לשדות. אם נרצה לקרוא למתודה אחרת השייכת למחלקה נעזר ב-self, לדוגמא:

```
  (self funcName)
```

הגדרנו שתי מתודות בשם name, אחת המקבלת פרמטרים ואחת שאינה מקבלת פרמטרים, המשמשות לקביעת שם העובד וקבלת שמו, בהתאמה.

ניתן להגדיר גם מתודה המקבלת שני פרמטרים בצורה הבאה:

```
name: aName  num: empNum
```

שם המתודה יהיה :name: num

נביט בהרחבה הבאה למחלקה שהרגע הצגנו. בדוגמא זו הוספנו מתודות גישה לשדות השונים של המחלקה.

```
Class Employee Object empName empNumber empSalary
Methods Employee
      new
            empName     <- 'No Name'.
            empNum      <- 0.
            empSalary   <- 0.
            ^self
|
      name
            ^empName
|
      name: aName
            empName <- aName
|
      empNum
            ^empNum
|
      empNum: anEmpNum
      "Employee numbers must be between 1000 and 1999
       This is a corporate regulation."
            ((anEmpNum >= 1000) and: [anEmpNum <= 1999])
                  ifTrue: [
                        empNum <- anEmpNum.
                        ^self
                  ]
                  ifFalse: [
                        smalltalk error: 'Invalid employee number'
                  ]
|
      salary
            ^empSalary
|
      salary: aSalary
            empSalary <- aSalary
]
```

נדגים כעת יצירת אובייקט מן המחלקה ושימוש בו:

```
> emp <- Employee new;   \
  name: 'Moshe';         \
  empNum: 1234;          \
  salary: 2000.
Employee

> emp name
Moshe

> emp empNum
1234

> emp salary
2000
```

כאשר כתבנו:

```
emp <- Employee new;
```

יצרנו אובייקט חדש מסוג Employee. לאחר מכן על ידי cascades שלחנו לו מספר הודעות כדי לקבוע את ערכי השדות השונים שבו.

אחרי האתחול, שלחנו לאובייקט הודעות על מנת שיציג את המידע שהוא שומר.

בקוד המחלקה השתמשנו ב:

```
smalltalk error: 'Invalid employee number'
```

smalltalk הוא סינגלטון של המחלקה Smalltalk (בדומה לסינגלטון true של המחלקה True).

על ידי ההודעה error אנו זורקים שגיאה במידה ומספר העובד אינו חוקי – אינו בטווח שהגדרנו.

משתנים לוקליים

כמקובל, גם בשפת Little Smalltalk צורת העבודה הנכונה ביותר היא הגדרת משתנים לוקליים לכל מתודה בהם היא משתמשת, בנוסף לשדות של האובייקט עצמו (וללא שימוש במשתנים גלובליים).
נדגים כעת מתודה בה משתנה לוקלי. המשתנה מוגדר בין קווים | |, בתחילת הפונקציה:

```
Class MyClass Object
Methods MyClass
      countTo: aNumber
            | counter |
            counter <- 1.
            [counter <= aNumber] whileTrue: [
                  counter print.
                  counter <- counter + 1.
            ].
]
```

במתודה :countTo מוגדר משתנה לוקלי שהוא counter.

דוגמא לשימוש במתודה:

```
> x <- MyClass new
> x countTo: 7
1
2
3
4
5
6
7
```

המחלקה Class

יצירת אובייקטים חדשים

כפי שכבר נאמר, גם Class עצמו הוא אובייקט בשפת Little Smalltalk. כל המחלקות שאנחנו בתור משתמש כותבים הם אובייקטים של מחלקה זו.

בדרך כלל ההודעות היחידות שנשלח לאובייקטים מסוג מחלקות זה: צור אובייקט חדש מסוג המחלקה שלך.

```
new
```

צור אובייקט חדש מסוג המחלקה, ושלח לו את ההודעה new.

```
new: anInteger
```

הודעה המשמשת בעיקר עבור אובייקטים מסוג מערך (Array), המקבלת את מספר האיברים המבוקשת.

נשים לב שהודעה זו אינה שולחת לאובייקט את ההודעה new! ולכן לא נשתמש בה על מחלקות שאנחנו כותבים אלא אם נממש אותה במיוחד.

הודעות נוספות של המחלקה Class

1. **methods** - קבלת רשימת השיטות של המחלקה.

דוגמא:

```
> True methods
Dictionary ( not printString xor: ifTrue:ifFalse: )
```

2. **respondsTo** - מחזירה את כל המתודות שהמחלקה מכירה (לא רק את המתודות שלה, אלא גם את כל אלו הקיימות במחלקות מהן היא נורשת).

```
> Integer respondsTo
Dictionary ( , isShortInteger new abs negated trucateTo: <= deepCopy
hash shallowCopy printString exp reciprocal rem: isChar truncated >=
class isKindOf: + asFraction bitShift: radix: timesRepeat: // log:
isLongInteger = asString even lcm: odd isInteger to: basicAt:
basicAt:put: - bitAnd: factorial copy positive roundTo: ~~ isNil
isFraction generality maxgen: floor negative sign ~= between:and:
basicSize isFloat asCharacter asLongInteger bitOr: bitXor:
strictlyPositive * anyMask: \\ to:by:
message:notRecognizedWithArguments: < allMask: bitAt: gcd: isNumb
er == notNil bitInvert quo: fractionalPart squared min: print > ceiling
ln sqrt max: assign:value: display isMemberOf: respondsTo: / asDigit
asFloat raisedTo: )
```

3. **fileOut** - כתיבת קוד המקור של המחלקה לתוך קובץ טקסט בשם <class>.st. ניתן על ידי הודעה זו לקבל, למשל, את המימוש של המחלקות הפנימיות של Little Smalltalk.

Collections

מבוא

Collections הינם דרך לארגן אוסף של אובייקטים. Collection הוא מבנה נתונים המאפשר לרוב הוספה, הסרה וגישה אל איברים בו.
נציג בפרק זה מספר Collections הקיימים בשפת Little Smalltalk.

סוגי ה-Collections העיקריים אותם נציג:

- **Set** – קבוצה לא סדורה של איברים. איברים ניתנים להוספה או הסרה לפי ערכם.
- **Dictionary** – קבוצה לא סדורה של איברים, בה לכל איבר יש מפתח, המזהה אותו באופן ייחודי, וכן יש לו ערך.
- **Interval** – סדרה מונוטונית של מספרים.
- **List**– רשימה סדורה של איברים. הכנסת איברים נעשית בהתחלה או בסוף.
- **Array** – מבנה זו הוא Collection בגודל קבוע בו לכל איבר יש אינדקס. לא ניתן למחוק איברים אולם ניתן להחליף איברים באיברים חדשים.
- **String** – מקרה מיוחד של Array, בו כל תא הוא מסוג תו.

סיווג ה-Collections השונים נעשה לפי התכונות שלהם. התכונות שלפיהן נסווג את מבני הנתונים הינן:

1. **מספר האיברים ב-Collection**: קבוע מראש / בלתי מוגבל.
2. **סידור האיברים**: קיים סדר בין האברים / לא קיים סדר בין האיברים.
3. **גישה לאברים**: לפי הערך שלהם, לפי אינדקס, גישה סדרתית.

כאשר נבוא לפתור בעיה, בהתאם לצרכי הבעיה נבחר את ה-Collection המתאים להשתמש בו.

השוואה בין Collections

הטבלה לקוחה מן התרגולים של הטכניון בקורס "תכנות מונחה עצמים".

Name	Creating Method	Fixed Size?	Order?	Insertion Method	Access Method	Removal Method
Set	new	no	no	add:	includes:	remove:
Dictionary	new	no	no	at:put:	at:	removeKey:
Interval	n to: m	yes	yes	none	none	none
List	new	no	yes	addFirst: addLast:	first	removeFirst remove:
Array	new:	yes	yes	at:put:	at:	none
String	new:	yes	yes	at:put:	at:	none

נציג מיד מספר דוגמאות לשימוש במבנים אלו.

דוגמאות של Collections

מילון (Dictionary)

מבנה נתונים שהוא קבוצה לא סדורה של איברים, בה לכל איבר יש מפתח (key), המזהה אותו באופן ייחודי, וכן יש לו ערך (value).

- **יצירת מילון חדש**:

```
d <- Dictionary new
```

- **גישה לנתונים**: המתודה at: של Dictionary משמשת להכנסה או שליפה של נתונים מהמילון.

הכנסה למילון: הכנסה למילון של הערך 'one' תחת המפתח 1.

```
d at: 1 put: 'one'
```

שליפה מהמילון: שליפה של הערך הנמצא תחת המפתח 1.

```
d at: 1
```

- **מחיקת איבר**:

```
d removeKey: 1
```

- קבלת **רשימה** של כל המפתחות:

```
d keys
```

- קבלת **רשימה** של כל הערכים:

```
d values
```

רשימה (List)

רשימה סדורה של איברים. הכנסת איברים נעשית בהתחלה או בסוף.

- יצירת רשימה חדשה:

```
L <- List new
```

- הוספת איברים:

```
> L addLast: 'acharon'; addFirst: 'rishon'
List ( 'rishon' 'acharon' )
```

- מחיקת האיבר הראשון, מחיקה לפי ערך:

```
L removeFirst
L remove: 'acharon'
```

- קבלת האיבר הראשון:

```
L first
```

קבוצה (Set)

- יצירת קבוצה ריקה:

```
S <- Set new
```

- הוספת איברים:

```
S add: 'red'; add: 'green'; add: 'blue'
```

- מחיקת איבר:

```
S remove: 'green'
```

- בדיקה האם איבר קיים בקבוצה:

```
S includes: 'black'
```

Interval

מחלקה זו, המממשת סדרה של מספרים, הוצגה כבר קודם אולם לא התייחסנו אליה כ-Collection.

- יצירת Interval:

```
> x <- (1 to: 5 by: 2)
Interval ( 1 3 5 )
```

- שינוי הגבולות שלו (הוספת איברים נוספים / הפחתת איברים מהסדרה):

```
> x upper: 7 ; lower: 3
Interval ( 3 5 7 )
```

- בדיקה האם איבר קיים ב-Interval:

```
> x includes: 4
false
```

מערכים ומחרוזות

מערכים ומחרוזות הם Collections המיוחדים בעובדה שניתן ליצור אותם על ידי תחביר מיוחד השייך לשפה, ולמלא אותם ישירות בתוכן.

יצירת מחרוזת, למשל, מתבצעת על ידי שימוש בגרשיים בצורה הבאה:

```
myStr <- 'Hello, World'
```

כמו כן, ניתן ליצור מערך על ידי:

```
myArr <- #( 1 2 3 )
```

ההודעה new אסורה לשימוש עבור מערכים ומחרוזות. שימוש ב-new יגרור הודעת שגיאה. ההגיון – new אינה מקבלת את מספר האיברים שאנו רוצים שיהיו במחרוזת (או במערך), ולכן לא הגיוני להשתמש בה.

מערכים ניתנים לאתחול על ידי ההודעה :new והגדרת מספר האיברים המבוקשים במערך. במקרה זה כל האיברים במערך יהיו כברירת מחדל nil. מחרוזת לעומת זאת לא ניתן לאתחל על ידי :new.

לא ניתן לאתחל מערך על ידי ערך של משתנה. אם נרצה לבצע זאת, ניצור את המערך, ולאחר האתחול נכתוב שורה כגון:

```
myArr at: 1 put: myVar
```

- הערה: מערכים ומחרוזות מגדירים את האינדקס של התא הראשון להיות 1.

פעולות על Collections

בחירת איברים

כל ה-Collections השונים תומכים בהודעה :select המאפשרת לבחור תת קבוצה שלהם העומדת בתנאי מסויים. הפרמטר המועבר הוא בלוק לו ארגומנט יחיד, המחזיר true או false, לפי ההחלטה האם אלמנט מסויים צריך להופיע בתוצאה או לא. דוגמא:

```
> #( 1 2 3 4 5 ) select: [ :i | ( i rem: 2 ) = 0 ]
Array ( 2 4 )
```

סוג ה-Collection המוחזר הוא זהה ל-Collection המקורי, אם ה-Collection המקורי הוא Set, List או Array. אחרת – מוחזר Collection מסוג Array.

על ידי שימוש בהודעה :reject ניתן לקבל את הקבוצה המשלימה. נשים לב שב-:reject לא תומכים כל ה-Collections, למשל, Array לא תומך בהודעה זו. ההודעה מוגדרת על ידי List ו-Set בלבד.

```
> #( 1 2 3 4 5 ) asSet reject: [ :i | ( i rem: 2 ) = 0 ]
Set ( 1 3 5 )
```

ביצוע פעולות על איברים

ההודעה :do מאפשרת לנו לבצע פעולה על כל אחד מאיברי ה-Collection. דוגמא:

```
> B <- [ :x | ( x rem: 2 ) = 0                                    \
        ifTrue: [ (x printString , ' is even.' ) print ] \
        ifFalse: [ (x printString , ' is odd.' ) print ] ]
> #(1 2 3 4 5) do: B
1 is odd.
2 is even.
3 is odd.
4 is even.
5 is odd.
```

אסיפת תוצאות

ההודעה :collect דומה בפעולתה לפעולה :do, אבל במקום רק לבצע פעולה על כל איבר, היא מבצעת את הפעולה ומחזירה Collection חדש שהוא אוסף התוצאות של הפעלת הבלוק על איברי ה-Collection המקורי.

```
> #(1 2 3 4 5) collect: [ :i | (i rem: 2) = 0 ]
Array ( false true false true false )
```

צבירת תוצאה

ההודעה :inject:into היא שימושית על מנת לבצע חישוב על כל איברי ה-Collection והחזרת תוצאה יחידה.

הארגומנט הראשון הוא הערך ההתחלתי, והערך השני הוא בלוק בעל שני ארגומנטים המבצע חישוב.

הארגומנט הראשון של הבלוק הוא ערך החישוב הקודם, והערך השני הוא האיבר הנוכחי ב-collection.

הערך המחושב על ידי הבלוק הוא התוצאה שתעבור לאיטרציה הבאה.

דוגמאות:

נגדיר מערך:

```
A <- #( 1 2 3 4 5 )
```

הקוד הבא יחזיר לנו את סכום האיברים במערך:

```
(A inject: 0 into: [ :a :b | a + b ] )
```

הקוד הבא מוצא את האיבר הגדול ביותר במערך:

```
(A inject: 0 into: [ :a :b |  (a > b) ifTrue: [a] ifFalse: [b] ] )
```

מימוש Collections

המימוש של ה-Collections השונים (ושל כמעט כל המחלקות בהן נתקלנו במסמך זה) נעשה בשפת Little Smalltalk עצמה.

נציג מספר קטעים מהמימוש של ה-Collections. ננצל את הזדמנות זו כדי להכיר גם הודעות נוספות השייכות ל-Collections בהן עדיין לא נתקלנו.

מימוש Collection inject:into:

המימוש נעשה על ידי שמירת משתנה לוקלי last השומר את תוצאת החישוב הקודמת, וביצוע לולאת do:.

קוד המימוש:

```
inject: aValue into: aBlock    | last |
    last <- aValue.
    self do: [:x | last <- aBlock value:last value:x ].
    ^last
```

מימוש Collection size

size היא הודעה המחזירה את מספר האיברים שב-collection. החישוב נעשה על ידי מניה. קוד המימוש:

```
size
    ^self inject: 0 into: [ :x :y | x + 1 ]
```

Composite method - מתודה שהיינו יכולים בלעדיה אבל היא מוסיפה נוחות למחלקה. בד"כ ממומשת על ידי שימוש במתודות אחרות. לדוגמא המתודה שהרגע ראינו: size ב-collections

מימוש: Collection occurrencesOf

ההודעה :occurencesOf מקבלת אובייקט ובודקת כמה פעמים אובייקט זה מופיע ב-collection.
מימוש הודעה זו נעשה על ידי שימוש ב-inject לצורך ספירה של אברי ה-collection העומדים בתנאי.
קוד המימוש:

```
      occurrencesOf: anObject
      ^self inject: 0
            into: [ :x :y | ( y = anObject )
                              ifTrue:  [ x + 1 ]
                              ifFalse: [ x ] ]
```

דוגמא לשימוש ב-Collections

נדגים כעת מימוש של מחסנית על ידי Collection. נשים לב כי למרות שמחסנית היא מבנה נתונים השומר בתוכו ערכים, המחסנית המוצגת איננה Collection – היא איננה נורשת מן המחלקה Collection וכן אינה ממשת את כל ההתנהגויות שהגדרנו כי ה-Collections בשפת Little Smalltalk תומכים בהן.

```
Class Stack Object list
Methods Stack
      new
            list <- List new
|
      push: anObject
            list addFirst: anObject
|
      pop   | top |
            top <- list first. list removeFirst. ^top
|
      size
            ^list size
|
      do: aBlock
            list do: aBlock
]
```

המחסנית ממומשת על ידי שמירה של אובייקט מסוג רשימה, ושליחת הודעות אליו.

קבצים

המחלקה File מאפשרת לנו לעבוד עם קבצי טקסט השמורים בדיסק.

שיטת העבודה היא לפתוח (open) קובץ, לאחר מכן לקרוא (read) או לכתוב (write) לקובץ, ולבסוף לסגור את הקובץ (close).

המתודות של המחלקה File

- **name: aString**

קביעת שם הקובץ עימו אנו מתעסקים (שם קובץ קיים או חדש).

- **open: aString**

פתיחת קובץ. מצב העבודה נקבע על ידי הפרמטר. ערכים חוקיים לפרמטר:

'r' - פתיחת הקובץ לקריאה בלבד.

'w' - פתיחת הקובץ לכתיבה. אם היו קיימים בו נתונים קודמים, הם ימחקו.

'a' - פתיחת הקובץ להוספת נתונים בסופו.

- **close**

סגירת הקובץ.

- **getString**

קריאת שורה מהקובץ (עד לתו השורה החדשה). נשתמש במתודה זו רק כאשר הקובץ פתוח במצב קריאה.

כשנגיע לסיום הקובץ, מתודה זו תחזיר nil, אחרת, היא תחזיר את השורה שנקראה.

- **print: aString**
- **printNoReturn: aString**

כתיבת מחרוזת לתוך הקובץ. ההבדל בין המתודות הוא האם יוסף תו שורה חדשה לסוף הקובץ או לא.

דוגמא

נציג דוגמא לשימוש במתודות שהוצגו:

```
F <- File new name: 'myfile.txt'
F open: 'w'
F print: 'Test file'
F print: '------------'
F printNoReturn: 'Hello,'
F print: ' world!'
F close
```

בסוף ביצוע פעולות אלו יווצר קובץ בשם myfile.txt ותוכנו יהיה:

```
Test file
------------
Hello, world!
```

קבלת קוד המימוש של מחלקה

שפת Little Smalltalk מאפשרת לנו לראות את קוד המקור של כל מחלקה, כולל המחלקות הפנימיות של Smalltalk עצמה. נושא זה יעזור לנו מיד, כאשר נתעניין יותר במימוש הפנימי של שפת Little Smalltalk.

על מנת לראות את קוד המקור של מחלקה, נוכל לפעול בשתי צורות. הצורה הראשונה היא להשתמש במתודה :fileOutOn של המחלקה Class, המקבלת אובייקט מסוג File וכותבת לתוכו את קוד המחלקה. לדוגמא, כך נוכל לקבל את מימוש המחלקה Block:

```
F <- File new name: 'block.st'
F open: 'w'
Block fileOutOn: F
F close
```

הדרך השניה היא שימוש בהודעה fileOut, היוצרת קובץ ששמו כ-8 האותיות הראשונות של שם המחלקה, ולאחריהן הסיומת ".st". לדוגמא:

```
Block fileOut
```

מימוש פנימי של שפת Little Smalltalk

בפרק זה נציג קטעים מקוד המימוש של המחלקות הפנימיות של שפת Little Smalltalk. הנחת העבודה היא שדרך הבנת מימוש המחלקות הנ"ל (הממומשות בעצמן בשפת Little Smalltalk) נבין את השפה בצורה הטובה ביותר.

Primitives

לצורך הבנת המימוש הפנימי נציג סוג נוסף של ביטוי (expression) תחבירי בשפה שלא הוצג עד כה. ביטוי זה נקרא primitive.

Primitive זהו ביטוי הקורא למתודה פנימית של האינטרפרטר של Little Smalltalk. קיימות כ-150 מתודות כאלו בהן המתכנת יכול להשתמש. אלו אבני הבניין הבסיסיות ביותר, האטומיות, של השפה.

כאשר אנו כותבים קוד Little Smalltalk ברמת המשתמש איננו צריכים אף פעם להשתמש ב-Primitives מכיוון שהמחלקות הפנימיות של שפת Little Smalltalk עושות את הקריאות בעצמן כשמתעורר הצורך.

נתקל ב-Primitives אלו כאשר נבוא להבין את הקוד הפנימי, או אם נרצה להרחיב את השפה ולהוסיף primitives משלנו (על ידי שינוי קוד המקור בשפת C של האינטרפרטר).

התחביר של primitive, בצורה פורמלית, הינו:

```
"<" integer {primary} ">"
```

כאשר ineteger הוא מספר בין 0 ל-255 הבוחר את המתודה לביצוע. הפרמטרים של הפקודה מוגדרים על ידי {primary}.

דוגמא לקריאה ל-primitive (ללא התעמקות במה היא מבצעת בפועל):

```
<13 self>
```

המחלקה Object

המחלקה Object היא השורש של היררכית המחלקות של Little Smalltalk. כל מחלקה אחרת יורשת ממחלקה זו, באופן ישיר או עקיף.
הקיום של שורש יחיד לכל המחלקות שימושי עבורנו על מנת להבטיח שהודעות מסויימות יהיו מובנות על ידי כל אובייקט במערכת.

המתודות המוגדרות במחלקה Object מיועדות לשתי מטרות:

- הגדרת התנהגות משותפת של כל האובייקטים - מתודות אלו אינן צריכות להידרס. המימוש שלהן מבוסס על מתודות אחרות.
- הגדרת התנהגות ברירת-מחדל עבור האובייקטים - מתודות אלו צריכות להידרס על ידי המחלקות הנורשות (subclasses) במקרי הצורך.

המחלקה Object עצמה הינה מחלקה אבסטרקטית, כלומר, לא נרצה ליצור אובייקטים מסוג מחלקה זו.
כפי שכבר צוין, כל מחלקה צריכה לרשת ממחלקה אחרת. במקרה של Object, כדי לציין שזהו השורש, אנחנו מסמנים כי הוא נורש מ-nil.

```
Class Object nil
```

המתודות השונות של Object יוצגו כעת, לפי חלוקה לנושאי פעולתן.
מתודות רבות ב-Object ממומשות בעזרת ה-primitives אותם הצגנו קודם.

הדפסת אובייקטים

מתודות הקשורות להדפסה קובעות כיצד האובייקט יציג את עצמו. ראינו כבר את המתודות print ואת printString. נראה כעת כיצד הן ממומשות.

התנהגות ברירת המחדל של המתודה print היא לקרוא ראשית למתודה printString, ולהדפיס לפלט הסטנדרטי את המחרוזת המוחזרת ממנה.
התנהגות ברירת המחדל של printString הינה להחזיר את שם המחלקה של האובייקט כמחרוזת.

הקוד הרלוונטי, מתוך המחלקה Object:

```
Methods Object
      print
            self printString print
]
Methods Object
      printString
            ^ self class printString
]
Methods Object
      class
            ^ <11 self>
]
```

בדיקת שיוויון

המתודה == משמשת לבדיקת שיוויון בין אובייקטים (האם שני אובייקטים הינם אותו אובייקט), המתודה ~~ הינה המתודה המשלימה שלה, והמתודה = אמורה לשמש לצורך השוואה בין תוכן של אובייקטים.

האופרטור == ממומש על ידי primitive. האופרטור ~~ ממומש על ידי האופרטור ==.
לאופרטור = ניתן מימוש ברירת מחדל, אולם כאשר אנו בונים אובייקט, יש להחליפו.

מימוש ברירת המחדל שניתן למתודה = הינו קריאה לאופרטור ==, שבברור הוא איננו נכון, אולם למחלקה Object אין דרך טובה יותר לבצע השוואה בין אובייקטים, והיא מנסה לתת ממשק משותף, לכן מחלקות הרוצות לאפשר שימוש באופרטור זה עליהן חייבות לחפוף אותו. הקוד הרלוונטי, מתוך המחלקה Object:

```
Methods Object
      == aValue
            ^ <21 self aValue>
]
Methods Object
      ~~ aValue
            ^ (self == aValue) not
]
Methods Object
      = aValue
            ^ self == aValue
]
```

העתקת אובייקט

המתודה **shallowCopy** יוצרת אובייקט חדש בעזרת new ומעתיקה ממנו את כל ה-Instance Variables כמו שהם. האובייקט החדש מכיל את אותם אובייקטים, כלומר שינוי של אחד מה-Instance יגרור שינוי בשני האובייקטים. לדוגמא, נביט במחלקה הבאה:

```
Class MyClass Object mInt mString
Methods MyClass
      new
            mInt <- 7.
            mString <- 'Hello'.
            ^self
|
      getInt
            ^mInt
|
      setInt: value
            mInt <- value
|
      getString
            ^mString
|
      setString: value
            mString <- value
]
```

ניצור כעת אובייקט מסוג המחלקה, x, נפעיל עליו shallowCopy ונחקור מעט את ההתנהגות:

```
> x <- MyClass new
> y <- x shallowCopy
> x getString print
Hello
> y getString print
Hello
> x getString at: 1 put: $X
Xello
> y getString print
Xello
> x setInt: 9
> x getInt print
9
> y getInt print
7
```

כפי שניתן לראות מהקוד, האובייקטים חולקים את אותם instances של אובייקטים. שינוי של המחרוזת mString באובייקט אחד השפיע על השני. אם זאת מדובר בשני אובייקטים נפרדים. כאשר שיננו את האובייקט mInt אליו מתייחס x, אז y לא הושפע.

המתודה **deepCopy**, במחלקות בהן היא ממומשת, גם יוצרת עותק של האובייקט, אבל גם יוצרת לו העתק של כל אחד מהשדות הפנימיים שלו, על ידי קריאה רקורסיבית.
המתודה ממומשת, למשל, במחלקה Array, אולם היא איננה ממומשת במחלקה List. ב-List נשאר מימוש ברירת המחדל. נביט בדוגמא הבאה הממחישה זאת:

```
> myList1 <- List new; addFirst: 'abc'; addFirst: 'edf'
List ( 'edf' 'abc' )
> myList2 <- myList1 shallowCopy
List ( 'edf' 'abc' )
> myList3 <- myList1 deepCopy
List ( 'edf' 'abc' )
> myList1 first; at:1 put: $X
Xdf
> myList2 print
List ( 'Xdf' 'abc' )
List ( 'Xdf' 'abc' )
> myList3 print
List ( 'Xdf' 'abc' )
List ( 'Xdf' 'abc' )
```

במקרה של רשימה, לא קיים הבדל בין deepCopy ל-shallowCopy. שינוי של הרשימה myList1 שינה גם את myList2 וגם את myList3.

```
> myArr1 <- Array new: 3; at: 1 put: 'abc'; at: 2 put: 'edf'
Array ( 'abc' 'edf' nil )
object count 3039
> myArr2 <- myArr1 shallowCopy
Array ( 'abc' 'edf' nil )
object count 3042
> myArr3 <- myArr1 deepCopy
Array ( 'abc' 'edf' nil )
object count 3048
> (myArr1 at:1) at:1 put: $X
Xbc
object count 3048
> myArr2 print
Array ( 'Xbc' 'edf' nil )
Array ( 'Xbc' 'edf' nil )
object count 3048
> myArr3 print
Array ( 'abc' 'edf' nil )
Array ( 'abc' 'edf' nil )
object count 3048
>
```

במקרה של מערך, deepCopy אכן יוצר עותק של האיברים. ניתן לראות כי גם אחרי ששינינו את myArr1 אז המערך myArr3 נותר ללא שינוי.

נביט כרגע במימוש המתודות הנ"ל במחלקה Object:

```
Methods Object
      deepCopy    | newObj |
            newObj <- self class new.
            (1 to: self basicSize) do:
                  [:i | newObj basicAt: i put: (self basicAt: i) copy].
            ^ newObj
]
Methods Object
      shallowCopy | newObj |
            newObj <- self class new.
            (1 to: self basicSize) do:
                  [:i | newObj basicAt: i put: (self basicAt: i) ].
            ^ newObj
]
```

```
Methods Object
      copy
            ^ self shallowCopy
]
```

נביט במימוש של deepCopy של המחלקה Array. הפונקציה עוברת על התאים אחד אחרי השני וקוראת לפונקציה copy כדי ליצור העתק מהם. במידה ולא קיימת כזו מתודה, נקראת מתודת ברירת המחדל שהוגדרה ב-Object.

נשים לב שכשאנחנו רוצים לממש את deepCopy עבור אובייקט, אנחנו יכולים או לממש את הפונקציה deepCopy או לממש את copy, לה deepCopy קוראת כברירת מחדל. הדרך המתאימה נבחרת בהתאם לצורך.

```
Methods Array
      deepCopy
            ^ self deepCopyFrom: 1 to: self size
]
Methods Array
      deepCopyFrom: low to: high    | newArray newlow newhigh |
            newlow <- low max: 1.
            newhigh <- high min: self size.
            newArray <- self class new: (0 max: newhigh - newlow + 1).
            (newlow to: newhigh)
                  do: [:i |  newArray at: ((i - newlow) + 1)
                              put: (self at: i) copy ].
            ^ newArray
]
```

המחלקה Class

ב-Smalltalk כל דבר הינו אובייקט, אפילו מחלקות.

המחלקה Class מגדירה את המאפיינים והמתודות המשותפות לכל המחלקות של שפת Little Smalltalk.

המחלקה Class איננה מחלקה אבסטרקטית - כל מחלקה בשפת LST היא מופע של מחלקה זו.

נשים לב כי המחלקה Class יורשת מהמחלקה Object וגם מתקיים כי המחלקה Object היא מופע של Class.

מאפייני מחלקה כוללים את:

- שמה
- מחלקתה אב
- קבוצת מתודות
- קבוצת instance variables לכל אובייקט מסוג המחלקה
- גודל כל אובייקט מסוג המחלקה

כאשר אנחנו יוצרים אובייקט חדש על ידי שימוש ב-new, אנו קוראים למעשה ל-new שמוגדר בתוך המחלקה Class:

```
Methods Class
      new          | newObject |
            newObject <- self new: instanceSize.
            ^ (self == Class)
                  ifTrue: [ newObject initialize ]
                  ifFalse: [ newObject new ]
]
Methods Class
      new: size   " hack out block the right size and class "
            "create a new block, set its class"
            ^ < 22 < 58 size > self >
]
Methods Class
      initialize
            superClass <- Object.
            instanceSize <- 0.
            methods <- Dictionary new
]
```

המתודה new מקצה זיכרון לאובייקט חדש ולאחר מכן מפעילה את הפונקציה new המתאימה לו (אותה כותב המתכנת של המחלקה).

כאשר שולחים הודעה לאובייקט, ראשית אנו מחפשים האם ההודעה נמצאת במחלקה המתאימה לאובייקט. אם לא, אנחנו עולים בצורה רקורסיבית בעץ ההורשה, עד שאנחנו מוצאים הודעה מתאימה או עד שמגיעים לשורש.

```
Methods Class
      upSuperclassChain: aBlock
            aBlock value: self.
            (superClass notNil)
                  ifTrue: [ superClass upSuperclassChain: aBlock ]
]
Methods Object
      respondsTo: message
            self class upSuperclassChain:
                  [:c | (c methodNamed: message) notNil
                              ifTrue: [ ^ true ]].
            ^ false
]
```

מקורות

1. הטכניון (1996) – מכון טכנולוגי לישראל, **"תרגולים בתכנות מונחה עצמים"**.
2. Timothy A. Budd (1994), "**A Little more Smalltalk**"
3. Vincent Manis (1999), "**Little Smalltalk Reference Manual**"