

חלק יבש - Types and values

Kotlin

1. קראו על בקרת זרימה (Control Flow) בשפת Kotlin. מהו הדמיון ומהו השוני בין if-else של Kotlin לבין if-else של שפת C?

הדומה - המבנה זהה, כל if-else בשפת C ניתן לרשום גם ב-kotlin.
השונה - If-else בשפת Kotlin הוא מיוחד משום שאנחנו יכולים להתייחס ל-if-else כביטוי. לדוגמה, if-else מחזיר ערך אשר אנחנו יכולים לתת לערך או משתנה, לכן אנחנו לא צריכים אופרטור טרינרי ב-kotlin. בשונה משפת C.

מהו הדמיון ומהו השוני ביחס ל if-then-else של ML?
הדומה – בשתי השפות,
השונה - ב-ML ה-If-else דומה לטרינרי, מכיוון שהוא תמיד מחזיר ערך (שפה פונקציונאלית).

האם לאופרטור הטרינארי בשפת C קיים מקביל בשפת Kotlin? אם כן מהו? אם לא, מדוע?

האופרטור הטרינארי – אופרטור אלביס, קיים ב-kotlin:
האופרטור אלביס מחזיר את הצד השמאלי אם הוא לא NULL, ואת הצד הימני אחרת: בדרכ "משתמשים בו עם return":

```
Val foo = bar() ?: return
```

2. האם פונקציה היא טיפוס בשפה? הסבירו.

התשובה היא כן, פונקציה ב-kotlin מוגדרת ע"י סוגי הטיפוסים אותם היא מקבלת, והטיפוס אותו היא מחזירה. אזי, פונקציה יכולה להשתייך לקבוצה, כלומר, בעלת טיפוס.

האם פונקציה ב Kotlin יכולה להיות פולימורפית? אם כן, מה ההבדל בין פונקציה שכזו ב Kotlin לבין פונקציה פולימורפית בשפת ML?

התשובה היא כן, פונקציה ב-kotlin יכולה להיות פולימורפית, למשל פונקציית sort ב-Array.
בשפת Kotlin נידרש להכריז לפני קריאה לפונקציה פולימורפית על סוג המשתנה "T" עליו הפונקציה תתבצע, לעומת ML שם אין צורך להכריז כל פעם על סוג המשתנה, אלא המפרש מבצע זאת באופן עצמאי(בתנאי שהמשתנה המתאים לדרישות הפונקציה, למשל אם בפונקציה יש שימוש באופרטור השוואה אז למשתנה שמוכנס חייב להיות אופרטור השוואה).

ב Kotlin ניתן להגדיר פונקציה ב infix notation, השוו בין הגדרה שכזו בשפת ML לבין זו שב Kotlin (ההשוואה צריכה לכלול: מתי ניתן להגדיר, ואיך ההגדרה מתבצעת)

- Kotlin: ההגדרה על Infix מתבצע בהכרזה על הפונקציה הרלוונטית, בנוסף חייב לשייך את הפונקציה לטיפוס מסוים(טיפוס this), הגדרה: בתחילת הפונקציה ע"י המילה השמורה infix.
- ML: ההגדרה על Infix יכול להיות מוגדר בכל מקום, כלומר גם לפני ההגדרה על הפונקציה וגם אחרי. הגדרה: ע"י שימוש במילה השמורה infix ואחריה השם של הפונקציה/תו הפעולה(למשל +).

3. הסתכלו בשקפים של פרק 3. מהי תכונת Void Safety? האם Kotlin היא Void Safe? אם כן, הסבירו כיצד תכונה זו באה לידי ביטוי בשפה. אם לא, תנו דוגמת קוד שמוכיחה זאת. [אל תפספסו את האופרטור של Elvis](#).

בטיחות void יכולה להיות בכל שפה, גם אם היא לא מונחת עצמים. בדרך כלל, מדובר על שפות שיש בהן reference semantics בשפות כאלו, יכולה להיות בעיה, והבעיה אכן קורית, של גישה למצביע שהוא NULL המונה VOID SAFETY מתייחס למאמצים למנוע שגיאה של גישה לערך שהוא NULL עוד בזמן הידור, תוך שימוש בבדיקת טיפוסים סטטית.

האם Kotlin היא Void Safe? אם כן, הסבירו כיצד תכונה זו באה לידי ביטוי בשפה. אם לא, תנו דוגמת קוד שמוכיחה זאת. אל תפספסו את האופרטור של Elvis. ברוב המקרים Kotlin היא Void/Null safe, מכיון שהשפה מוכוונת להיות Null safe לעומת Java כדי לא לבצע בדיקות exception מיותרות. תכונה זו באה לידי ביטוי ע"י שימוש בסימן "?" לפי משתנים, למשל: b.length? כאשר b הוא מחרוזת. במידה ו-b לא מאותחל, לא יהיה כניסה לlength. אך למרות הניסיון להפוך את השפה ל Void Safe, אפשר לעקוף זאת ע"י שימוש בסימן "!!" אחרי משנה, או כאשר יש שימוש במודולים חיצוניים שנכתבו בשפת Java.

4. [קראו את הסיכום על פולימורפיזם](#). מהם סוגי הפולימורפיזם הקיימים בשפת Kotlin, הסבירו בפירוט והביאו דוגמאות לכל אחד מן הסוגים (לפחות דוגמה אחת לכל סוג) [Overloading](#):

פעולת העמסה על פעולות (operators) מתבצע ע"י הגדרת פונקציה על האופרטור עם טיפוסים אחרים/חדשים. למשל: a.plus(b) יבצע העמסת אופרטור על "+".

[Coercion](#):

בKotlin יש המרת טיפוסים אוטומטית רק לחלק מהטיפוסים וגם לא לשני הכיוונים, למשל פונקציה המקבלת Int לא תקבל double במקום, אך ההפך יעבור. בנוסף, קיים בKotlin המרות בין טיפוסים ע"י פונקציות מתאימות, למשל: המרה של int ל char מתבצע ע"י Int.toChar().

[Parametric](#):

בKotlin קיימת אפשרות להגדרת פונקציות המקבלות/מחזירות משתנה T, כלומר פונקציה העובדת על מספר רב (לא מוגבל) של טיפוסים. למשל: פונקציית sort למערכים, fun <T> Array<out T>.sort() יכולה לבצע מיון של מערך למספר לא מוגבל של מערכים (עם טיפוסים שונים).

[Subtyping](#):

שפת Kotlin מבוססת OOP, וכל מערכת הטיפוסים שלה מסוג אובייקטים, למשל: Int יורש Number שירש Any. לכן Subtyping מובנה בשפה, ולמשל: מערך של Any יכול להחזיר איברים מטיפוס Int.

5. הגדירו מהו **type punning** (או תרגמו את השקף המתאים). תנו דוגמאות ל **type punning** בשתי השפות: **C, Rust**.

Type punning – התייחסות לערך כאוסף ביטים ולפרש ייצוג זה באופן שלא תואם את הטיפוס שלו.
ל-Type punning יש את הכוח:

- (1) "להציץ" לתוך רצף הביטים של המימוש של הטיפוס.
- (2) "להתעלל" בערך ע"י שימוש במקומות שלא מתאימים לערך שלו.

שפת C: ניתן לבצע casting של מצביעים ל-int.
שפת RUST: casting: 65.4321 -> 65 -> A

```
let decimal = 65.4321_f32;
let integer = decimal as u8;
let character = integer as char;
```

6. כתבו תכנית קצרה בשפת **C** המדפיסה **LE** אם ערכים מיוצגים בזיכרון ב **Little Endian**, ומדפיסה **BE** אם ערכים מיוצגים בזיכרון ב **Big Endian**. הסבירו בקצרה מדוע התכנית מבצעת את הנדרש.

```
short a = 753;
char* p = &a;
if (*p + (*(p+1))*256 == a)
    printf("LE\n");
else
    printf("BE\n");
```

נבדוק האם הערכים בזיכרון מיוצגים ב **BE** או **LE**,
נבצע זאת ע"י הקצאת משתנה מטיפוס **short** שטופס שני
בייטים בזיכרון, ונעבור עם מצביע מסוג **char** על המידע
בזיכרון.
נשווה את המשתנה **a** לסכום הערכים של המידע במקום
הראשון בזיכרון + המידע במקום הבא בזיכרון כפול "ההזזה"
של המידע לבייט העליון, כלומר כפול 256.

במידה ויש שיוון של המספר אז זה **Little Endian** אחרת זה **Big Endian**.

7. הסבירו מדוע **Union** בשפת **C** לא מממש בצורה מדויקת את בנאי הטיפוסים **Disjoint Union**.

ערך של טיפוס שנבנה באמצעות **union** בשפת **C** מכיל בכל רגע נתון ערך אחד השייך לאחד הטיפוסים שמהם הוא נבנה, לעומת **disjoint union** אשר מהווה אופרטור מתורת הקבוצות, ומחזיק תיג לקבוצה ממנה הוא נוצר. כלומר, ב-**Union** בשפת **C** אין באמת דרך לדעת לאיזה טיפוס שייך הערך הנוכחי.

8. הגדירו מהם **Mixed typing** ו **Gradual typing**. מהו ההבדל בין המושגים?

Gradual typing – תוספת המאפשרת לכתוב תוכניות בטיפוסיות דינאמית אך עם התפתחות התוכנית להוסיף שיוך של טיפוסים ומשתנים, ערכי החזרה וכו'. שפת התכנות תודיע על סתירות העלולות לגרום לשגיאות טיפוסים בזמן ריצה, וכן תודיע על הגדרות מיותרות. כך יורדו "ההוצאות" בזמן ריצה. באופן כללי, השפה תומכת בכך שעבור ישויות מסויימות הבדיקה היא בזמן ריצה, ואחרות יהיו בזמן הידור.

Mixed typing – השילוב בין טיפוסיות סטטית ודינאמית. כלומר חלק מבדיקות הטיפוסים מבוצעות הזמן קומפילציה וחלק בזמן ריצה. ייתכן כי חלק מהבדיקות יתבצעו גם וגם.

9. למדו על סוגי המערכים מהשקפים [בפרק 5.2](#). באיזו שיטה של ייצוג מערכים משתמשת [שפת התכנות](#)

[NIMROD](#)? הביאו ציטוטים התומכים בתשובתכם.

:Array

השיטה היא: static array, מכיוון שהגודל נקבע בזמן ההידור.

:Open array

השיטה היא: flexible arrays, מכיוון שהגודל גמיש וניתן לשינוי בזמן ריצה, והאינדוקס ב-int.

:Sequences

השיטה היא: flexible arrays, מכיוון שהגודל גמיש וניתן לשינוי בזמן ריצה, והאינדוקס ב-int.

ציטוטים תומכים:

Arrays are a homogeneous type, meaning that each element in the array has the same type. Arrays always have a fixed length which is specified at compile time (except for open arrays). They can be indexed by any ordinal type. A parameter A may be an *open array*, in which case it is indexed by integers from 0 to len(A)-1. An array expression may be constructed by the array constructor [].

Sequences are similar to arrays but of dynamic length which may change during runtime (like strings). Sequences are implemented as growable arrays, allocating pieces of memory as items are added. A sequence S is always indexed by integers from 0 to $\text{len}(S) - 1$ and its bounds are checked. Sequences can be constructed by the array constructor [] in conjunction with the array to sequence operator @. Another way to allocate space for a sequence is to call the built-in newSeq procedure.

A sequence may be passed to a parameter that is of type *open array*.

<https://users.atomshare.net/~zlmch/nimdoc/upload/manual.html>

Sequences are similar to arrays but of dynamic length which may change during runtime (like strings). Since sequences are resizable they are always allocated on the heap and garbage collected.

<https://nim-lang.org/docs/tut1.html#advanced-types-arrays>