תכנות מונחה עצמים

סיכום החומר בקורס "תכנות מונחה עצמים" בטכניון

סיכם: דוד ארינזון

מסמך זה הורד מהאתר http://www.underwar.co.il.

אין להפיץ מסמך זה במדיה כלשהי, ללא אישור מפורש מאת המחבר.

מחבר המסמך עשה כל שביכולתו למנוע טעויות, אולם הוא אינו אחראי לכל נזק, ישיר או עקיף, שיגרם עקב השימוש במידע המופיע במסמך, וכן לנכונות התוכן של הנושאים המופיעים במסמך.

המסמך נכתב על ידי **דוד ארינזון**

מנגנוני אבסטרקטיה:

1. מנגנון האובייקט
2. מנגנון המחלקה
3. מנגנון הירושה

כיוון האבסטרקציה

- קיימות שפות שמכילות רק אובייקטים, ולא קיים בהן מושג המחלקה והירושה. דוגמא לשפה כזאת היא Java Script. בשפה הנ"ל, ניתן לבצע clone, ובצורה הזאת לקבל אובייקט נוסף. כאן, מנגנון האבסטרקציה הוא אובייקטים.
- אובייקט – יחידה בעלת מצב בזכרון והתנהגות.

עקרונות תכנות מונחה עצמים:

1. ***כל דבר הוא אובייקט***.

בשפות כמו C++ העקרון הנ"ל לא מתקיים, כי לדוגמא int הוא אינו אובייקט. משתנים פרימיטיביים לדוגמא הם גם לא אובייקטים, הם אכן תופסים מקום בזכרון, אך אינם יודעים להגיב להודעות (דרך התקשורת בין אובייקטים בשפות מונחות עצמים).

2. ***תקשורת בין אובייקטים נעשית רק באמצעות משלוח הודעות***.

אם נקח לדוגמא את האופרטור + ב-C++, אזי הוא לא הודעה, אלא אופטרטור גלובלי, אשר מקבל שני פרמטרים. כמו כן, if הוא אכן מבנה גלובלי, אך לא מנוהל ע"י הודעות. זאת בניגוד ל-Squeak, בו משתנה תנאי הוא למעשה אובייקט בוליאני אשר ניתן לשלוח לא הודעה (ifTrue/ifFalse...)

3. ***לכל אובייקט יש אזור בזכרון שמגדיר את המצב שלו (ערך השדות שלו)***.

נוסף לכך, כך אובייקט שהוא מורכב, השדות שלו הם אובייקטים אחרים.

4. ***כל אובייקט הוא מופע של מחלקה, ויש קשר בינו לבין המחלקה שלו.***

הדבר הנ"ל לא קיים ב-C++. שוב, נחזור לדוגמא של int, הוא אינו מופע של מחלקה, אלא משתנה פרימיטיבי.

ב-Java יש טיפוסים פרימיטיביים, אך לכל טיפוס פרימיטיבי יש מחלקה ש"עוטפת" אותו. (לדוגמא int נעטף ע"י Integer).

5. ***מחלקות מאורגנות בירושה בעלת שורש יחיד.***

ההיגיון מאחורי החשיבה הזאת הוא שיש פעולות שכל האובייקטים מכירים.

ב-C++, מחלקה יכולה לרשת מיותר ממחלקה אחת, וגם, יש לה אפשרות כלל לא לרשת מאף מחלקה.

ב-*Java*, אכן יש שורש יחיד שהוא Object. אין ירושה מרובה, אך ישנו פיצוי על כך (Interface).

סוגי טיפוסים:

1. טיפוסים פרימיטיביים/אטומיים – בדרך כלל חלק מהשפה. לדוגמא: int, float, double ב-C++.
2. טיפוסים מורכבים – נוצרים מהפעלה של Type Constructor שמקבל כקלט מספר טיפוסים, ויוצר מהם טיפוס חדש.

   לדוגמא, int[100] נוצר ע"י הפעלת Type Constructor (אופרטור הסוגריים) על הטיפוס int.

מערכות טיפוסים:

סוגי מערכות טיפוסים:

1. ***Untyped*** – אין בדיקת טיפוסים. ניתן לראות זאת כמערכת שבה קיים רק טיפוס אחד, וכל הפעולות מותרות. לדוגמא, אסמבלר של PDP11 וגרסאות ישנות של Perl.
2. ***Statically Typed*** – בדיקות הטיפוסים מתבצעות בזמן קומפילציה. לדוגמא C, ML, Java. (זה קורה לרוב ב-Java, אשר קיימים בה גם אספקטים דינמיים).

   יתרון של מערכת הטיפוסים הנ"ל הוא קוד בטוח, מכיוון שהבדיקות נעשות לפני הרצה (קומפילציה/קישור). בנוסף, הקוד יותר קריא, והביצועים טובים יותר (נחסכות בדיקות זמן ריצה).
3. ***Dynamically Typed*** – בדיקות הטיפוסים מתבצעות בזמן ריצה. דוגמא לכך היא Squeak.

   יתרון של מערכת הטיפוסים הנ"ל הוא אורך הקוד, בדר"כ הקוד קצר יותר. כמו כן, המערכת הנ"ל מאפשרת השתנות של חוקיות של פעולות, בהתאם לשלב שבו נמצאים בזמן ריצה.

חוזק מערכת הטיפוסים (מידת אכיפתה קשר בין פעולה לטיפוס):

1. ***Strong Typing*** – קיים קשר חזק מאוד בין פעולה לטיפוס. לא ניתן לבצע פעולות שלא מתאימות לטיפוס.

   יתרון של שיטה זו הוא מניעה של טעויות רבות.
2. ***Weak Typing*** – שפות בהן הקשר בין פעולה לטיפוס ניתן לשבירה. ניתן לבצע פעולות שלא בדיוק מתאימות לטיפוס.

   דוגמא לכך היא שפת C. בשפה הנ"ל, ניתן לבצע על פוינטרים בערך כל מה שרוצים (לדוגמא כפל פוינטרים).

   יתרון של שיטה זו הוא גמישות, ניתן לבצע פעולות מתוחכמות (לדוגמא... פוינטרים).

<u>סיווג נוסף של מערכת הטיפוסים</u>:

1. ***Structural Typing*** – השקילות של הטיפוסים נעשית לפי מבנה הטיפוס.
2. ***Nominal Typing*** – השקילות של הטיפוסים נעשית על בסיס של הטיפוס, באותו הקשר. (אשמח להגדרה יותר טובה...)

Pascal
- נומינלית כמעט לחלוטין.

C
- Type Constructors (לדוגמא, מערכים ומצביעים לפונקציות) – שקילות סטרקטורלית.
- Struct, Union – שקילות נומינמלית.

C++
- שקולה ל-C, פרט לכך ש-Templates הם בשקילות סטרקטורלית.

Java
- מחלקה – שקילות נומינלית.
- מערכים ו-generics – שקילות סטרקטורלית.

- קיימת בשקילות נומינלית בעית תקשורת בין תוכנית ובין מודולים.
  - ***Pascal*** – נניח וקיים קובץ, ותוכנית צריכה לקרוא אותו, והשנייה לכתוב בו. בשתי התוכניות יש לחזור על ההגדרה של הטיפוס. אולם, שתי התוכניות לא יוכלו לתקשר ביניהן באותו הקובץ, מכיוון שבשקילות נומינלית צריך להיות באותו שם ובאותו הקשר. לכן, ב-Pascal, התקשורת בין תוכניות היא רק באמצעות טיפוסים פרימיטיביים.
  - ***C/C++*** – ניתן לשפכל הגדרות. או לחילופין, ניתן להגדיר קובץ משותף שיכיל את ההגדרות, ואז לעשות לו include (ה-Pre Processor יחליף את ההגדרות, ומבחינת הקומפיילר התוצאה תהיה שקולה לשכפול). מכאן, כי ניתן לתקשר באמצעות טיפוסים מורכבים. השפות מאפשרות זאת, מכיוון שהן Weakly Typed. מכיוון ש-Struct הוא נומינלי, נחשוב כי תהיה כאן אותה בעיה כמו ב-Pascal. אולם, כאן Struct ו-Union אכן נומינליים ברמת הקובץ, אבל לא נומינליים בצורה טהורה בין קבצים (שני טיפוסים עם אותו שם ובהקשר שונה, והשפה מאפשרת לעשות השמות).
  - ***Java*** – התקשורת בין תוכניות היא באמצעות קובץ נתונים. Java היא יותר Strongly Typed מאשר ++C, ולכן יש כאן בעיה דומה לשל פסקל בתקשורת באמצעות קובץ. הפתרון שמוצע ב-Java הוא **סיריאליזציה**, כלומר, לכתוב ולקרוא נתונים באמצעות וידוא של מבנה הקובץ.

האובייקט

האובייקט מורכב משלושה דברים עיקריים:

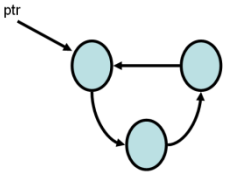
1. ***מצב*** – אוסף הערכים אשר מוצבים באובייקט.
2. ***התנהגות*** – אוסף ההודעות שהאוביקט יודע להגיב אליהן, והפעולות שמתבצעות בתגובה להודעות.
3. ***זהות*** – תכונה שמייחדת את האובייקט ומבדילה בינו לבין אובייקטים אחרים.

- ניתן לראות **פונקציה בשפת C** כאובייקט:
  - מצב – ניתן לשמור אותו באמצעות משתנים סטטיים ברמת הפונקציה.
  - התנהגות – הפעלה של הפונקציה (לדוגמא, קידום מונה).
  - זהות – יש מופע יחיד של הפונקציה => זהות יחידה.
- אם נגדיר **מודול** שמכיל משתנים ופונצקיות, גם הוא יהיה אובייקט:
  - מצב – ערכי המשתנים שבמודול.
  - התנהגות – הפונקציות שבמודול (ייתכנו התנהגויות שונות).
  - זהות – יחידה, כי יש מופע יחיד.
- הקירוב הטוב ביותר לאובייקט בשפת C הוא ה**קובץ**:
  - מצב – תוכן הקובץ, מיקום ה-file pointer.
  - התנהגות – הפעולות שניתן לעשות על הקובץ (פתיחה, קריאה, כתיבה...)
  - זהות – ה-file pointer, יכול להיות יותר מאחד.
- ***ב-Java***, יש שני סוגי משתנים: פרימיטיביים, ואובייקטים שהם מופעים של מחלקות. האובייקט חי מהרגע שיוצרים אותו, ועד הרגע "שלא צריך אותו", כלומר, אין רפרנס אליו. (וכאן נכנס ה-Garbage Collection...)

Garbage Collection:

- יתרון: מונע דליפות זיכרון.
- חיסרון: עוצר את התוכניות בנקודה מסוימת ומתחיל "לטפל" באובייקטים. (לוקח את השליטה מהמתכנת על זמן ה-CPU).

- **<u>אלגוריתמים עיקריים</u>**:



  - ***Reference Counting*** – הוסף מונה רפרנסים לכל אובייקט, עדכן אותו כאשר יש שינויים (הוספת/הסרת רפרנסים). כאשר המונה מגיע ל-0, הסר את האובייקט.
    - יתרון: שיטה אשר לא עוצרת את התוכנית כדי לבצע פעולת פינוי גדולה.
    - חיסרון: אם רפרנס מסוים מצביע למעגל, ברגע שרפרנס הנ"ל יצביע למשהו אחר, לא נזהה שיש זבל, מכיוון שקיימות הצבעות מעגליות והמונים לא מתאפסים.
  - ***Mark & Sweep*** – התוכנית רצה, וברגע שסף הזיכרון יורד מתחת לערך מסוים, התוכנית עוצרת ומתחיל איסוף הזבל – סורק את כל הזיכרון, אשר מתחיל ממשתני התוכנית (בודק את כל האובייקטים שמצביעים טרנזיטיבית לאובייקטים אחרים), כל אובייקט כזה יסומן כנמצא בשימוש. בשלב הבא, עוברים על כל ה-heap וכל אלו שלא מסומנים לשימוש, נמחקים.
    - יתרון: overhead קטן, וביט אחד בלבד לכל אובייקט.
    - חיסרון: עצירה של התוכנית, ופעולה יקרה מאוד.
  - ***Stop & Copy*** – מרחב הזיכרון מחולק לשני חלקים, כאשר בכל פעם משתמשים רק בחלק אחד. ברגע שחלק הזיכרון שבשימוש כמעט התמלא, עוצרים את התוכנית, ומעתיקים את כל האובייקטים "החיים" לחלק השני של הזיכרון, ועתה התוכנית תמשיך לפעול על אזור הזיכרון השני, וכן הלאה.
    - יתרון: לא עוברים על כל האובייקטים, אלא רק על אלה הנגישים. אין צורך בעוד ביט.
    - חיסרון: הקטנה של הזיכרון פי 2. בהעתקת אובייקט, משנים את כל הרפרנסים אליו כדי שיצביעו למקום חדש בזיכרון.

בדיקת הנגישות (בשתי השיטות האחרונות) מסתמכת על זה שיש לנו רשימה של משתני התוכנית ומפה עבור כל אובייקט – מה גודלו, ולאילו איברים הוא מצביע.

<u>השוואת זהות</u>:

- ***C++*** - הזהות היא הכתובת של האובייקט בזיכרון.

  לדוגמא: A a,b; &a == &b; מתקיימת כאן השוואה של כתובות – השוואה בין אובייקטים.

- ***Java*** – a==b יחזיר את הזהות (1 אמ"מ a ו-b הם רפרנסים לאותו אובייקט).

<u>השוואת מצב</u>:

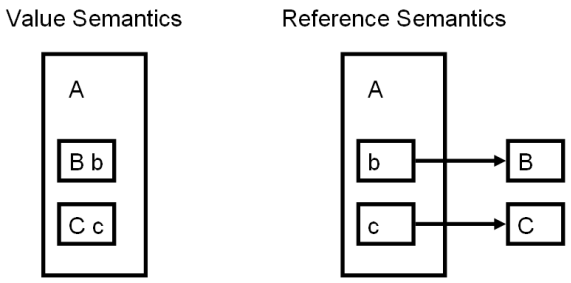
- ***Shallow Compare*** – מעבר על השדות של a ו-b, ואם הם שווים => נחזיר true.

אם יש שדה שמצביע לאובייקט אחר (ערך ההצבעה שונה), נקבל false.

- ***Deep Compare*** – אם יש אובייקטים מוצבעים, נבדקים גם השדות שלהם זה מול זה.
- ***C++*** - יש להגדיר אופרטור השוואה בצורה מפורשת (לא נוצר default).
- ***Java*** – קיימת המתודה equals ב-Object, אשר בודקת שדה שדה (מומלץ בדר"כ לדרוס אותה).

<u>Value / Reference Semantics</u>:

Class A { B b; C c; };



- יתרון של שימוש ב-reference מאשר pointer, הוא העובדה ש-reference חייב להיות מאותחל, בעוד ש-pointer יכול לקבל את הערך NULL.
- Reference תמיד מצביע על אותו האובייקט, ולא ניתן לגרום לו להצביע על אובייקט אחר.
- ***C++*** - Value Semantics.
- ***Java*** – Reference Semantics (למרות שבדומה ל-Pointers ב-C++, ניתן להציב null).
- ***C#*** - כמו ג'אווה.
- ***Eiffel*** – טיפוסים פרימיטיביים – Value, מחלקות – Reference.
- אין משמעות ל-Garbage Collection ב-Value Semantics.
- ב-Reference Semantics, השוואת זהות גוררת השוואת מצב (כי ההתייחסות היא לאותו האובייקט).

<u>אספקטים סטטיים של אובייקט</u>:

- תכונות (שדות):
יכולות להיות לא ניתנות לשינוי, ויכולות להיות ניתנות לשינוי.

  - ***C++-ב***, const נותן אפשרות להגדיר תכונה שהיא immutable – לא ניתנת לשינוי, אולם בעירבון מוגבל.
  לדוגמא:

    - **Int const *p** – p מצביע ל-const int, כלומר, לא ניתן לשנות את המשתנה עליו p מצביע באמצעות p (אולם, ניתן בעזרת אחרים).
    - **Int *const p** – p הוא מצביע קבוע, ולא ניתן לשנות אותו.
    - **Int const* const p** – p מצביע קבוע, ולא ניתן לשנות דרכו את מי שהוא מצביע עליו.

  קיימת בעיה עם const, לדוגמא, אם ב-A יש שדה b, ו-p הוא מצביע אליו מהסוג השלישי, אזי לא נוכל לשנות את p->b. אולם, אם ל-b יש שדה d, נוכל לבצע שינוי דרך p->b->d.

  - ***Java-ב***, final דומה ל-const, אך לא ניתן להחיל אותו על רמות נוספות בהיררכיה של ההצבעה.
- בעוד שתכונות ומבנה לא ניתנות לשינוי, מצב של אובייקט משתנה לאורך הזמן (למעט שפות שבהן האובייקטים הם immutable).

לדוגמא, string ב-Java, היא מחלקה שהאובייקטים שלה הם immutable (אם משנים, נוצר אובייקט חדש).

<u>אספקטים דינמיים של אובייקט</u>:

- ישנם שני סוגי אובייקטים:
  - ***פסיביים*** – מבצעים פעולות בתגובה לפעולות מן החוץ.

דוגמא לכך, היא כל האובייקטים של Squeak, שהיא שפה Pure OOP, בה כל האובייקטים מתקשרים אחד עם השני בעזרת הודעות.

  - ***אקטיביים*** – פועלים ללא הודעה מן החוץ.

לדוגמא, threads, שהם חלק מ-Runnable ב-Java.

- כאשר אובייקט מקבל הודעה, ייתכנו שתי אפשרויות:
  - האובייקט מכיר את ההודעה, ויבצע את ההתנהגות המתאימה.
  - האובייקט לא מכיר את ההודעה, ותתקבל הודעת שגיאה.

בשפות סטטיות נקבל שגיאת קומפילציה (Java, C...), כאשר בשפות דינמיות נקבל שגיאת זמן ריצה.

ב-Java, איננו יודעים איזו פונקציה תיקרא, אולם אנחנו בטוחים שהיא חוקית, מכיוון שעברנו את בדיקת הקומפילציה.

- Dispatching הוא מנגנון אשר אחראי לראות מהי ההודעה, ומי מקבל את ההודעה, ולפיהן מחליט איזו מתודה להריץ.
  - ***Single Dispatch*** – ההתנהגות נקבעת לפי מקבל ההודעה בלבד (הטיפוס הדינאמי).
  - ***Multiple Dispatch*** – ההתנהגות נקבעת לפי אחד יותר מהארגומנטים בהודעה.
- סוגי מתודות:
  - ***Setters/Mutators*** – מתודות אשר משנות את מצב האובייקט.
  - ***Getters/Selectors/Inspectors*** – מתודות שמחזירות מידע על האובייקט.

Convertor הוא סוג של Getter, אך הוא מחזיר את המצב של האובייקט בצורה אחרת (למשל toString ב-Java).

- ***Constructors*** – הקצאת זיכרון ואתחול.
- ***Destructors*** – שחרור זיכרון וניקוי.
- ***Revealers*** – פעולות שחושפות מצב של אובייקט.

המחלקה

- המחלקה היא מנגנון אבטרקציה שלוכד את המשותף למספר אובייקטים.
- מחלקה אינה לוכדת זהות, כלומר, לאובייקטים שונים במחלקה יש זהות שונה.
- המחלקה לוכדת את התכונות של האובייקט, את המבנה, אבל לא את המצב (פרט למשתנים סטטיים).
- המחלקה גם כן לוכדת את ההתנהגות והפרוטוקול.

העמסת אופרטורים:

1. ניסיון להגדיר מחלקות בדומה לטיפוסים הפרימיטיביים ב-C++.
2. לא קיימת ב-Java.
3. קיימת ב-C# כתאימות ל-C++.

המחלקה מורכבת מחמישה חלקים:

1. ***Forge*** – פרוטוקול ליצירת עצמים, כלומר, חתימות ה-Constructors.
2. ***Mill*** – מימוש הפרוטוקול ליצירת עצמים, כלומר, מימוש ה-Constructors.
3. ***Protocol*** – טיפוס (התחייבות המחלקה כלפי חוץ), כלומר, חתימות כל המתודות והשדות שאינם private.
4. ***Behavior*** – מימוש הטיפוס, כלומר, מימוש המתודות.
5. ***Structure*** – מבנה/תבנית האובייקט בזיכרון.

רמות ויזיביליות:

C++	Public, protected, friend, private
Java	Public, protected, private, default (כאשר default משמעו שמחלקות באותו ה-package יקבלו הרשאת גישה)
Eiffel	Export to… (כלומר, ניתן להגדיר עבור כל מתודה למי היא חשופה)
Squeak	מתודות – Public, שדות - Private

- יחידת האנקפסולציה ב-Squeak היא האובייקט, כאשר ב-C++ היא המחלקה. מסיבה זו, לא ניתן לגשת מאובייקט אחד לשדות של אובייקט אחר ב-Squeak (בניגוד ל-C++).
- בשביל Garbage Collection, אנו נזקקים למפת זיכרון. המחלקה עשויה גם להכיל את מפת הזיכרון הזו.
- **Interface** – מכיוון שאין ב-Java ירושה מרובה, הוכנס המנגנון הנ"ל. הוא מגדיר מתודות ברמת הפרוטוקול (חתימות). מחלקה שיורשת ממנו, חייבת לממש את כל המתודות הנ"ל. כל מתודה

תהיה public, וכל שדה יהיה public final. כמו כן, לא ניתן ליצור אובייקטים מהטיפוס הנ"ל (ומכאן ניתן להסיק כי אין לו Forge ו-Mill).

- ***ב-Java*** כל השדות מאותחלים לערכי ברירת מחדל, ולכן, אין צורך לאתחל אף שדה, אלא אם כן הוא final. הסיבה לאתחול הנ"ל נעוצה בהפעלת של Ctors ומתודות בתוכן.
- ***המחלקה האנונימית ב-Java*** – קיים למחלקה מבנה, כי היא מגידרה שדות. זאת מחלקה מנוונת כי יש לך רק אובייקט אחד. מאחר ואין לה שם, לא ניתן לקרוא לאובייקטים שלה. אם ניצור אח"כ מחלקה אנונימית באותו אופן, תיווצר מחלקה אחרת, מכיוון שב-Java קיימת שקילות נומינלית.

כמו כן, למחלקה האנונימית אין Ctor (כי אין לה שם), אולם קיים לה Mill מכיוון שניתן להשתמש ב-instance initializer שמאפשר אתחול שדות (הוא מופעל בכל פעם שנוצר אובייקט של המחלקה).

- ***המחלקה האבסטרקטית ב-C++*** - לא ניתן ליצור מופעים שלה. בשפה הנ"ל, מחלקה תוגדר כאבסטרקטית אם היא מכילה או יורשת מתודה שהיא pure virtual.

למחלקה הנ"ל יהיה Ctor, כי הוא נחוץ לאתחול השדות. כמו כן, כל מחלקה אבסטרקטית יכולה להכיל מתודות שאינן pure virtual, וכן מימוש שלהן. אזי בהכרח למחלקה כגון זו תהיה מוגדרת התנהגות.

Virtual Ctor	אין לו משמעות, כלומר, לא תתאפשר קומפילציה.
Private Ctor	הגדרה שלו לא תאפשר לאחרים ליצור עותקים של המחלקה.
Virtual Dtor	לדוגמא, אם B יורש מ-A, ומתבצעת קריאה ל-Dtor של B (למשל מחיקה של מצביע ל-B), אזי ההגדרה הנ"ל תכריח שתהיה קריאה גם ל-Dtor של A, ותמנע זליגת זיכרון.
Pure Virtual Dtor	אם אין מימוש, אזי זאת שגיאת קומפילציה. בהינתן מימוש, למעשה זה מאפשר למחלקות יורשות ליצור אובייקטים (כאשר תתבצע הריסה של המחלקות היורשות, תיקרא גם ההריסה של המחלקה הבסיסית שהגדירה את ה-Dtor). אספקט נוסף הוא שמחלקה אשר ממשת Dtor כזה מוגדרת כאבסטרקטית, וכל המחלקות היורשות יוגדרו כקונקרטיות.

- ***מה קורה ב-Squeak?***

קיימת המתודה initialize, שהיא פונקציית אתחול שנקראת בעת יצירת האובייקט ע"י new.

במחלקה, הפרוטוקול נותן לקומפיילר כלי עזר על מנת לבדוק את נכונות ההודעות שנשלחות לאובייקט. Squeak היא שפה דינמית, ובשפות דינמיות אין משמעות לפרוטוקול או לטיפוס (בדיקות הקומפילציה הינן מועטות מאוד).

מטא-מחלקה

- בשפות שאינן OOP ו-ST (כמו C), אין קיום לטיפוס לאחר זמן קומפילציה.
- בשפות שהן DT יש חשיבות לטיפוס לאחר זמן קומפילציה כי הבדיקות מתבצעות בזמן ריצה.
- בשפות שהן OOP אבל ST, נראה הגיוני לתת ייצוג למחלקות בזמן ריצה.

במקרה של קישור דינמי, הפתרון הוא שבכל שפה מונחת עצמים, יש בזכרון טבלת קישור וטבלת מתודות אשר מכילה שני שדות:

- שם הפונקציה.
- כתובת הפונקציה.

כל אובייקט מכיל מצביע לטבלה והטבלה משותפת לאובייקטים במחלקה. (הנ"ל יופיע בתחילת האובייקט)

- ***ב-C++*** הייצוג של המחלקה מוגבל מאוד, ומסתכם בטבלה הזו משתי סיבות:
  - עקרון הסופרמרקט – "תשלם רק על מה שקניתי".
  - עקרון התקורה – features של השפה שלא משתמשים בהם לא יגרמו לתקורה מבחינת מקום וזמן.

מכאן כי מחלקה שלא מכילה מתודות וירטואליות, לא תהיה עבורה טבלה כזו.

ללא הטבלה, הקומפיילר פשוט שותל את הכתובת של הפונקציה במקום שקיימת אליה גישה.

- ***ב-Java*** ו***ב-Squeak*** קיים ייצוג בזמן ריצה למחלקות. ובייצוג הנ"ל נעזר מנגנון ה-Reflection.

מודל השכבות

נבדיל בין שני סוגי אובייקטים:

1. אובייקטים טרמינליים – לא ניתן להפעיל עליהם Ctor על מנת ליצור אובייקטים נוספים.
2. אובייקטים שמייצגים מחלקות (כולל המטא-מחלקות), וניתן להפעיל עליהם Ctor על מנת ליצור אובייקטים נוספים. כאשר, הפעלת Ctor על מטא-מחלקה תיצור מחלקה.

- **רמה אחת**:
  - אין מחלקות, כל אובייקט מתאר את עצמו.
  - **דוגמאות: JavaScript, self.**
  - אובייקטים נוצרים בעזרת Clone.
  - אין הקבלה למחלקה, ואין ירושה, אולם, ניתן בעזרת מצביע parent לממש מבנה דומה.

- **שתי רמות**:
  - רמה תחתונה – אובייקטים, רמה עליונה – מחלקות, שאינן אובייקטים בפני עצמן.
  - **דוגמא: C++.**
  - קשה מאוד לממש Garbage Collection בשפות כאלו, כי איסוף זבל דורש מפת זיכרון, וכל מה שקיים כאן הוא מידע על המתודות.

- **שלוש רמות**:
  - אובייקטים => אובייקטים שמתארים מחלקות => מטא-מחלקה יחידה, שהיא מופע של עצמה.
  - **דוגמא: Java, C#.**
  - כל מחלקה מתארת את הרמה שמתחתיה, ברמה שרשימת המשתנים במחלקה הם השדות של האובייקט.
  - לכל אובייקט שמתאר מחלקה, יש רשימה של מתודות. רשימה זו היא ההודעות שניתן לשלוח לכל אובייקט שהוא מופע של המחלקה.
  - המחלקה Class מתארת את עצמה.
  - בהינתן הודעה, נלך לפי חץ ה-Instantiation למחלקה היוצרת, ונראה אם ההודעה קיימת בה. אם לא, נמשיך באלגוריתם במעלה חצי הירושה.
  - במע' של 3 רמות יש מטא-מחלקה יחידה, המאתרת את עצמה, ויורשת מ-Object, שהוא גם מופע של המטא-מחלקה.
  - מבנה חיפוש המתודות (מחלקה יוצרת, עץ ירושה) הוא גמיש מאוד, ומאפשר שימוש במתודות שנוצרות בזמן ריצה בשפות דינמיות.
  - אין טבלה אחת למחלקה, בה מרוכזות ההודעות של המחלקה, אלא היא פרושה ברמות הירושה.
  - עקב מגבלת שלוש הרמות, לא ניתן לממש משתנים סטטיים (הם לא יכולים להישמר במחלקה, ולא במטא-מחלקה).

- **ארבע רמות**:
  - אובייקטים => מחלקות המתארות את האובייקטים הנ"ל (לא סינגלטון) => מטא-מחלקה לכל מחלקה, אשר מתארת אותה (סינגלטון) => מטא-מחלקה יחידה (מופע של עצמה, ויורשת מ-Object).
  - ניתן לממש משתנים סטטיים ברמה השלישית.

- **חמש רמות**:
  - במודל של ארבע רמות, קיימת מטא-מחלקה יחידה שהיא מופע של עצמה. כאן, קיימת רמה חמישית, אשר המטא-מחלקה היא מופע שלה, ובנוסף, הרמה החמישית היא מופע של המטא-מחלקה.
  - **דוגמא: Squeak.**
  - במודל הנ"ל, הכלל אומר שמערכת הירושה של המטא-מחלקות עוקבת אחרי מערכת הירושה של המחלקות. לדוגמא, אם Integer יורש מ-Object, אזי Integer Class תירש מ- Object Class.

- **פולימורפיזם** – אבסטרקציה מעל טיפוס, כלומר, יישויות שיכולות להכיל מספר טיפוסים.
- בשפות דינמיות, הפולימורפיזם תמיד מתקיים. כלומר, משתנה יכול להכיל מס' טיפוסים, ומתודות יכולות להכיל ערכים מכל טיפוס.
- **פולימורפיזם אד-הוק** – ייצור אחד שמשמש למס' מטרות אשר אין קשר סיסטמטי ביניהן. מס' המטרות הוא סופי, ולפיכך לא חייב להיות מכנה משותף.
  - פולימורפיזם אשר נובע מהמרות (coercions), בין אם מוגדרות בשפה ובין אם מוגדרות ע"י משתמש.
  - פולימורפיזם אשר נובע מהעמסה (overloading), מספר פונקציות אם אותו שם אך מימוש שונה.
- **פולימורפיזם אוניברסלי** – הקשר בין המופעים אינו מקרי, ומס' המופעים אינו חסום.
  - פולימורפיזם פרמטרי – מס' אינסופי של טיפוסים שאין קשרי ירושה ביניהם. דוגמא לכך היא Template.
  - פולימופריזם הנובע מקשרי ירושה.

Strict Inheritance

- כל התכונות של מחלקת האב עוברות כמו שהן למחלקת הבן.
- **<u>השפעה על המחלקה</u>**:
  - המבנה מתרחב.
  - הפרוטוקול מתרחב (התחייבות האובייקט כלפי חוץ – ניתן להוסיף תכונות והתנהגות, אך לא ניתן להסיר!).
  - ההתנהגות מורחבת, יש מימוש רק של פרטי הפרוטוקול החדשים, ואין השפעה על אלו אשר הגיעו מהאב.
  - Forge – ה-Ctors אינם עוברים בירושה, לכן למחלקה החדשה יהיה Forge חדש.
  - Mill – חדש, אך בדר"כ יפעיל את ה-Mill של האב.
  - מבנה זיכרון – אם B יורש מ-A, אזי הרישא של B תהיה מורכבת מ-A, ולאחר מכן התוספות (אם יש).
- **<u>השלכות</u>**:
  - מובטחת תאימות מלאה. כלומר, פונקציה אשר מקבלת אובייקט מטיפוס A, תוכל לקבל גם אובייקט מטיפוס B (B ירש מ-A).
  - הקישור יכול להיות תמיד סטטי. אין צורך בטבלת מתודות וירטואלית, נחסכת כאן תקורה.
  - מתקיים יחס של תת-טיפוס. כלומר B הוא תת-טיפוס של A.

- כאשר מדובר על ירושה, מתודות הן תמיד פולימורפיות, כלומר, יכולות לקבל את הטיפוס וכל תת-טיפוס שלו. קיימת כאן הסתייגות, כאשר מדובר על מחלקה שהיא final (לא ניתן לרשת ממנה) - היא אינה מקיימת פולימורפיות.
- ***משתנים ב-Squeak*** הם פולימורפיים, כי הם יכולים להחזיק כל טיפוס.
- ***This*** הוא פולימורפי, כי הוא יכול להצביע על אובייקטים מטיפוסים שונים.
- ***C++, Java*** – כל משתנה שהוא מצביע/רפרנס למחלקה הוא פולימורפי (יכול להצביע גם למחלקות בנים).
- משתנה מטיפוס ***void**** גם הוא פולימורפי.

<u>המרות</u>:

- בכל פעם שמציבים לתוך משתנה ערך שהוא תת-טיפוס שלו, מתרחש up-cast (המרה במעלה היררכיית הירושה, מהבן לאבא). ישנם שלושה מקרים עיקריים כאלו:
  - הצבות – A& a = b.
  - קריאת מפורשת לפונקציה עם פרמטר – f(A).
  - מתודה של האב מופעלת על אובייקט ממחלקת הבן – המרה של ה-this.

- בכל פעם שמגדירים מחלקת בן, הקומפיילר מגדיר פ' המרה מהבן לאב (חילוץ השדות של האב מהבן).
- הפעולה ההפוכה ל-up-cast היא down-cast. עבור המרה כזאת, נקבל ב-Java שגיאת זמן ריצה. בשפות כגון C++, שהן Weakly Typed, נוכל לגשת לשדות לא חוקיים, ותהיה קריסה.
- **Casting Operators**:
  - ***Static_cast<>*** - בודק בזמן קומפילציה שניתן לבצע את ההמרה (שיש קשר של ירושה, לא משנה מה הכיוון). הקומפיילר בודק שקשרי ירושה קיימים, אין בדיקה בזמן ריצה.
  - ***Dynmaic_cast<>*** - מאפשר להציג כל טיפוס בזמן קומפילציה, אם בזמן ריצה הטיפוס אינו מתאים, נקבל שגיאת זמן ריצה.
  - ***Interepert_cast<>*** - המרה שלוקחת בתים בזיכרון ומפענת אותם בצורה אחרת לגמרי.
  - ***Const_cast<>*** - אחלה דבר.

מערכים:

- ***ב-C***, ניתן להמיר int ל-char, אולם לא ניתן להמיר []int ל-[]char. המרה של מערכים משמעה להעתיק תא-תא, זוהי פעולה יקרה, ומכיוון שאין הסתרה של פעולות יקרות מהמשתמש ב-C, אין יחס של subtype בין מערכים.
- ***ב-C++***, אין המרה בין מערכים כלל. (למרות שעקרונית ניתן היה לאפשר המרה בין מערכים של מצביעים, מכיוון שמצביעים לוקחים מקום קבוע בזכרון).
- ***ב-Java*** מתקיים כי אם B הוא subtype של A, אזי גם []B הוא subtype של []A.

Non-strict inheritance

- התפתחות האובייקט עוקבת אחרי התפתחות המחלקה, ניתן לראות זאת באספקטים שונים:
  - מבנה האובייקט – בתוך אובייקט הבן, יהיה אובייקט האבא (אובייקט האבא קודם בזיכרון).
  - הטבלה הוירטואלית של הבן היא תמיד הרחבה של הטבלה הוירטואלית של האב (כלומר, ניתן לראות את השתקפות שושלת הירושה בטבלה).
  - סדר הבנייה (קריאה ל-Ctors) הוא קבוע. תחילה נקרא זה של האב, ואז של הבן. כלומר, הבנייה עוקבת אחרי סדר הירושה.
  - סדר ההריסה (קריאה ל-Dtors) הוא הפוך.
- מתוך הבן ניתן לגשת לשדות/מתודות של האב בעזרת super ב-Java, ובעזרת אופרטור :: ב-C++.

רכיבי המחלקה	Strict inheritance	Non-strict inheritance
Forge	חדש (Ctors לא עוברים בירושה)	חדש (Ctors לא עוברים בירושה)
Mill	חדש (בדר"כ מפעיל את ה-Mill של האב)	חדש (בדר"כ מפעיל את ה-Mill של האב)
Structure	מתרחב	מתרחב
Protocol	מתחרב (ניתן להוסיף מתודות נוספות)	מתחרב (ניתן להוסיף מתודות נוספות)
Behavior	מימוש פריטים חדשים בפרוטוקול בלבד	מימוש פריטים חדשים בפרוטוקול, ושינוי/הרחבה של פריטים קיימים.

- Inclusion Polymorphism – פולימורפיזם שנובע מקשרי ירושה.

Binding (קישור):

מתי יתבצע הקישור בין האובייקט להודעה?

- ***Static Binding*** – קישור ההודעה לטיפוס הסטטי של האובייקט.

דוגמא: C++. (כדי ליצור קישור דינאמי ב-C++, יש להכריז לפני מתודה virtual).

- ***Dynamic Binding*** - קישור ההודעה לטיפוס הדינאמי של האובייקט.

דוגמא: Squeak, Java.

- ***ב-C#***, ברירת המחדל היא ירושה strict. אם רוצים לדרוס מתודה של האב, יש להגדיר אותה כ-virtual אצלו. ואצל הבן קיימות שתי אפשרויות:
  - Override – דריסה ויצירת קישור דינאמי.
  - New – דריסה ויצירת קישור סטטי.

- ***ל-super יש קישור סטטי.***
- ***ל-this קישור דינאמי.***
- ***ב-Java*** ניתן לקבל מידע על הטיפוס בעזרת instance of.
- ***ב-Squeak*** ניתן לקבל זאת בעזרת isMemberOf, isKindOf.
- ***על מנת לקבל מידע בזמן ריצה על אובייקט ב-C++,*** יש להשתמש במנגנון ה-RTTI. בהינתן A* a, המנגנון מאפשר לדעת מיהו הטיפוס הדינאמי של a בזמן ריצה.

<u>עידונים</u>:

- ***עידון אלפא*** – המתודה הדורסת היא זו שיוזמת את העידון. (משמעות, העברת האובייקט B לתת-אובייקט שלו A).
- ***עידון בטא*** – המתודה שיוצרת את הקשר היא המתודה הנדרסת. (הבטחה שמתודת האב תיקרא).

<u>Dynamic Binding</u>:

שפות סטטיות, ירושה יחידה.

- ***Java, C#*** - מחלקות בלבד (לא Interface).
- ***C++*** - ללא ירושה מרובה.

- ***ב-C++***, בהינתן מחלקה עם פונקציה וירטואלית, היא תמיד תכיל מצביע לטבלת המתודות הוירטואליות, אשר תהיה משותפת לכל האובייקטים מהמחלקה. הטבלה מכילה את שם המתודה, ומסודרת לפי סדר הגעת המתודות.
במצב של ירושה, הטבלה של המחלקה היורשת ממשיכה את הטבלה הנורשת.

הטבלה הנ"ל היא המטא-מידע היחיד ב-C++.

- האם פונקציה שמוגדרת private, יכול להיות וירטואלית?
  - ***C++*** - כן. ההיגיון אומר שפ' מעטפת שקוראת לפ' הפנימית, ואנו רוצים להיות מסוגלים לשנות את המתודה הפנימית לפי הטיפוס הדינמי.
  - ***Java, C#*** - גישה שונה, הבן לא מכיר מתודות private, ולכן אין סיבה שידרוס אותן. אולם, ב-Java, לא ניתן לכלול אסור על מתודות וירטואליות להיות private, מאחר וב-Java כל המתודות (פרט לסטטיות) הן וירטואליות. אך, עבור המתודות שהן private, קיים קישור סטטי.
  - B יורש מ-A, ב-A יש f שקוראת ל-g, g מתודה private. B דורס את g... מה יקרה?
    - ***C++*** - b.f() תפעיל את f של A, כי רק ל-A יש את f. הקריאה ל-g תפעיל את g של B.
    - ***Java*** – b.f() תפעיל את f של A. יש שם קריאה ל-g, אבל בגלל שהקישור סטטי עבור מתודות private, תופעל g של A ולא של B.

<u>מיקום ה-Vptr (מצביע לטבלת הפ' הוירטואליות)</u>

1. ***Borland style*** – מחזיקים את המצביע בתחילת האובייקט.
2. ***Gnu style*** – המצביע לתחילת הטבלה לא תמיד יהיה בתחילת האוביקט אל במקום בו הופיעה הפ' הוירטאולית הראשונה.

וריאציה נוספת של Gnu style היא מיקום המצביע היכן שהוגדרה המתודה הוירטואלית הראשונה (יכולה להיות מוגדרת בין שדות למשל).

<u>שפות דינמיות</u>:

- אין טיפוס סטטי, לכן שיטת הטבלאות הוירטואליות לא תעבוד.
- כאשר שולחים הודעה לאובייקט, הולכים למחלקה שלו, ובודקים האם קיימת ההודעה הנ"ל. אם לא, נמשיך לעלות בהיררכיית הירושה, עד שנמצא את המתודה, או עד שנמצה את כל ההיררכייה, ונקבל שגיאה.
  - המנגנון הנ"ל גמיש ומאפשר להוסיף מתודות בזמן ריצה.
  - לא קיימת בו הבטיחות של שפות סטטיות.
  - חיסרון בולט של ביצועים (הצורך בחיפוש).
- בשפות דינמיות, טבלת המתודות אינה רציפה בזיכרון אלא מורכבת משרשרת של טבלאות. בשפות סטטיות, היא בהכרח רציפה.
- החיפוש בשפות סטטיות הוא בעזרת אינדקסים, כאשר בשפות דינמיות זהו תהליך חיפוש המשתרע על מבנה דמוי Hash table.

<u>Binding ב-Ctors</u>:

מה קורה אם ב-Ctor מופעלת פ' וירטואלית?

- ***C++*** - תהליך הבנייה מורכב משני שלבים:
  - הקצאת זיכרון בגודל הדרוש.
  - קריאה ל-Ctor שיאתחל את האובייקט.

    לכן, ה**קישור** חייב להיות **סטטי**. אם הקישור היה דינאמי, עלולים לקרוא למתודה שתתייחס לחלק שלא אותחל (לדוגמא ב-Ctor של A, כאשר קיים חלק B אשר לא אותחל עדיין, במקרה ש-B יורש מ-A).

  - אחת ההשלכות היא שהמצביע לטבלת הפ' הוירטואליות משתנה במהלך בניית האובייקט.
  - מה יקרה אם תהיה קריאה לפ' pure virtual מתוך Ctor?
    - אם הקריאה היא ישירה – נקבל שגיאת קומפילציה.
    - אם הקריאה היא עקיפה (דרך פ' מתווכת) – הקומפיילר לא יתפוס את השגיאה ונקבל שגיאת זמן ריצה.
- ***Java*** – תהליך הבנייה מורכב משלושה שלבים:
  - הקצאת זיכרון.
  - השמת ערכי ברירת מחדל לכל המשתנים.
  - הפעלת Ctors.

הדבר הנ"ל מאפשר **קישור דינאמי** בתוך Ctors, מכיוון שכל השדות מאותחלים לערכי ברירת מחדל, ואין חשש לקבלת ערכי זבל.

- ***לא ניתן לתת ערכי ברירת מחדל ב-C++*** לפי עקרון הסופרמרקט, ויש גם שדות שלא ניתן לתת להם ערכי ברירת מחדל (לדוגמא references).

Conformance

- כלל התאימות – צריכים להיות מסוגלים לקרוא לפ' הדורסת בכל אותם תנאים שנקראת הפ' הנדרסת.
- **<u>טיפוסי פרמרטים</u>**:
  - ***מודל המציאות מכתיב Co variance.***
  - ***No variance*** – לא ניתן לשנות את טיפוסי הפרמטרים לפונקציה. **(C++, Java, C#).**
  - ***Co variance*** – אם הפונקציה מקבלת טיפוס A, אזי היא תוכל גם לקבל כל טיפוס אשר יורש מ-A. (ירידה בעץ הירושה). **(Eiffel).**
  - ***Contra variance*** – אם הפונקציה מקבל טיפוס A, אזי היא תוכל גם לקבל כל טיפוס אשר A יורש ממנו. (עלייה בעץ הירושה). **(Sather).**
- **<u>ערך חזרה</u>**:
  - ***מודל המציאות מכתיב Co variance.***
  - Co variance - ***Java, C++***.
- **<u>Input/Output parameters</u>**:
  - ***מבחינת תאימות המודל Input = contra, Output = co***, החיתוך הוא no variance, ולכן עבור פרמטרים אשר יכולים להיות גם קלט וגם פלט, מתקיים no variance.
  - מה קורה אם ***ב- Java*** מגדירים 2 פונקציות שהיחס ביניהן הוא co-variance? למשל B תירש מ-A. ו-D יירש מ-C. ונקבל כי קיימת ב-A f(C) וב-B f(D).

    כאן יתקיים יחס של ***overloading***.
  - ***ב- C++***, במקרה הזה, מתקיים מצב של ***hiding***.
- **<u>דריסה</u>**:
  - ***ב- Java ו- C++***, הארגומנטים צריכים להיות מאותו טיפוס, או מס' ארגומנטים, ואותו סדר ביניהם על מנת שתתקיים דריסה. אם יש הבדל בארגומנטים, ב-C++ מתקיים מצב של החבאה. (במצב של A שמכילה f() ו-B יורשת המכילה f(), מתקיים מצב של דריסה. אולם ניתן לקרוא ל-f() של A בעזרת A::f()).
- תאימות שדות: ???
- **<u>Exceptions</u>**:
  - A f() throws {e1…en}, B מחלקה יורשת המכילה f() throws {x1…xn}.

ב- Java, הפונקציה הקוראת צריכה להתמודד עם ה-Exceptions של הפונקציה שהיא קוראת לה (תפיסה/זריקה כלפי מעלה). כלומר {x1…xn} מוכלת ממש ב-{e1…en}. כלומר, צריך להתקיים שלכל xi קיים ej אשר מאותו טיפוס כמו xi או ש-xi הוא תת טיפוס של ej.

- **<u>הרשאות גישה</u>**:
  - קיימת ב-A פונקציה f(), וב-B יורשת פונקציה f(). ל-f של A יש קבוצת הרשאות x ול-f של B יש קבוצת הרשאות y.
  - הפונקציה הדורסת יכולה להגדיל את הרשאות הגישה, אך לא להקטין אותן. (***Java***, אך עם יוצא דופן אחד שהוא default, אשר מאפשר הכרה ברמת ה-package).
  - ***C++:***
    - ניתן להגדיר מה רמת הירושה (protected למשל) – f הוא Public ב-A, והירושה היא protected, אז f הוא protected ב-B.
    - יש אפשרות שירושה תקטין את ההרשאות של המחלקה היורשת. למשל: Class B: protected {...}. כל מה שהיה public הופך להיות protected.
  - ***ב-C++***, חובות וזכויות אינם עוברים בירושה (סתירה לעקרון התאימות)
  - ***ב-Eiffel***, בזמן הגדרת מתודה, ניתן לומר (בעזרת export) למי לחשוף אותה. כמו כן, בשפה הנ"ל, זכויות עוברות בירושה, אבל חובות לא עוברות בירושה.

<u>תמונת מצב Java:</u>

מרכיבי הטיפוס	תאימות	Java
טיפוסי פרמטרים	Contra – variance	No variance
הטיפוס המוחזר	Co – variance	Co – variance
שדות read only	Co - variance	אין דריסה של שדות (יש החבאה)
שדות read/write	No variance	אין דריסת שדות
Exception	כל חריגה שנזרקת מ-F' היא תת טיפוס של חריגה הנזרקת מ-F (A מכילה את F, ו-B היורשת מכילה את F')	

<u>וריאציות על מושג המחלקה והמתודה</u>

- מחלקות/מתודות אבסטרקטיות:
  - ***ב-Eiffel***, מתודות (ומחלקות) אבסטרקטיות מוגדרות ע"י deferred.
  - ***ב-C# ו-Java***, מתודות (ומחלקות) אבסטרקטיות מוגדרות ע"י abstract.
  - ***ב-C++***, מתודות אבסטרקטיות מוגדרות ע"י pure virtual. המתודות הנ"ל שונות מ-abstract של Java, מכיוון שבניגוד ל-Java, ניתן לתת להן גוף.
  - ***ב-C++***, מחלקה שיש בה מתודה אבסטרקטית תהיה מחלקה אבסטרקטית. בניגוד ל-Java, שבה למחלקה אבסטרקטית לא חייבת להיות מתודה אבסטרקטית.
  - מקרה מיוחד של מתודה virtual & pure virtual היא Dtor. כאשר מגדירים אותו, חייבים לתת לו גוף (כי Dtors) לא עוברים בירושה.

המשמעות של Dtor וירטואלי היא שהמחלקות היורשות יהיו קונקרטיות.

- ***מחלקות שהן final*** – ניתן רק לקרוא למתודות ולא לדרוס אותן. ב-C# המילה השמורה היא sealed.
- ***ב-C++*** הקירוב הטוב ביותר ל-final הוא למעשה static (כי אי אפשר לדרוס מתודה סטאטית).
- ***מתודות שהן final ב-Java*** נקראות ע"י קישור סטטי ולא ע"י קישור דינאמי, כאופטימיזציה של הקומפיילר.
- **Mixins** - (???)
- **Traits** – יחידות של התנהגות. כל Trait יגדיר לאילו מתודות הוא מצפה, ואילו מתודות הוא מספק.

  יתרונות:
  - אין סדר לינארי.
  - Trait הוא טיפוס.
  - אין קונפליקטים של שמות שדות, מכיוון שב-Trait יש רק מתודות.

  חסרונות:
  - קונפליקט כאשר שני Traits ממשים את אותן המתודות.

  ב-Java, מחלקה עדיפה על Trait.

  ב-Squeak, הקונפליקט נפתר ע"י שתי שיטות:
  - ניתן "למחוק" מתודה מ-Trait ע"י הסינטקס (#MethodName) – TraitName.
  - Renaming, ע"י סינטקס מיוחד.

<u>וריאציה על דריסה של מתודות:</u>

- החתימה של הפונקציה נקבעת בזמן קומפילציה, אך גוף הפונקציה נקבע בזמן ריצה, לפי הטיפוס הדינאמי של ה-receiver.
- קישור של מתודה הוא דינאמי עבור ה-receiver, אך סטטי מבחינת הפרמטרים – ***הגישה של C++ וב-Java***.
- ***Single Dispatch*** – קביעת הקישור לפי פרמטר יחיד (receiver).
- ***Multiple Dispatch*** – קביעת הקישור לפי קבוצה של פרמטרים. (קיימת הרחבה של Java בשם MultiJava, אשר תומכת בגישה הנ"ל).

<u>ירושה מרובה</u>

מנגנון אשר קיים ב-C++, וב-Eiffel עבור מחלקות, וב-Java עבור Interfaces.

- יתרונות:

- פולימורפיזם.
- שימוש חוזר בקוד.

- חסרונות:
  - סיבוך המצב.
  - יצירת מצבים של דו-משמעות.
- חסר כאן כל מה שקשור ל-Thunks ושטויות על טבלאות, אשמח להשלמה (!??)

אלגוריתם כללי להפעלה של מתודה וירטואלית

.p->f()

עושים ל-p upcast למחלקה הגבוהה ביותר בה הוגדרה f, כי בה תהיה כניסה עבור f. ייתכן מצב שיהיו כמה מחלקות עליונות ביותר (בגלל ירושה מרובה) – נבחר אחת מהן, ונסמנה B.

ל-B יש VTBL, ובמקום קבוע בה יש כניסה ל-f. נלך אליה, ושם תהיה כתובת הפונקציה / thunk. ניגש אליה. ה-thunk מכיל תיקון ל-this, אם צריך, כתובת לביצוע, ותיקון לערך חזרה – אם צריך.

- **זמן קומפילציה**: הקומפיילר שותל קוד לביצוע אשר מממש את האלגוריתם הנ"ל.
- **זמן ריצה**: האלגוריתם מבוצע ע"י מערכת זמן הריצה.

נניח כי D יורש מ-B1 ו-B2.

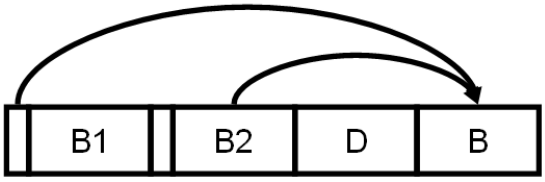
מימוש ב-C++:

- בכל אובייקט של D יש תת אובייקט מסוג B1 ותת אובייקט מסוג B2.
- בכל תת אובייקט יש מצביע לטבלת פונקציות וירטואליות.

בכל תת אובייקט מסוג B1 יש מצביע לטבלה הכוללת את כל הפונקציות שהוגדרו לראשונה ב-B1. אם פונקציות אלו נדרסו ב-D, תופיע הכתובת של הפ' הדורסת.

- לאובייקט D עצמו יש מצביע לטבלת פונקציות וירטואליות שהוגדרו לראשונה ב-D (לרוב, הטבלה הזו מתלכדת, ויהיה שיתוף של המצביע עם זה של תת-האובייקט הראשון).
- קריאה לפונקציה וירטואלית – לעיתים יש צורך לבצע תיקון ל-this או תיקון הערך המוחזר (בעזרת thunks).
- קיים מקרה בעייתי של ירושה במבנה יהלום. הגישה ***ב-C++*** לפתרון היא שאם B1 ו-B2 יירשו מ-B, הם יחליטו אם יהיה עותק יחיד או שניים של B כשייווצר D.

באם B1 ו-B2 יירשו וירטואלית מ-D, המבנה של D יהיה כדלהלן:



- האובייקט המשותף תמיד נמצא בסוף.
- באובייקט ממחלקה יורשת, סדר תתי האובייקטים הוא לפי הכרזת הירושה, ובמקרה של ירושה וירטואלית, במקום תת אובייקט יופיע מצביע לתת האובייקט הוירטואלי שיופיע בסוף האובייקט.
- **דו משמעות מקרית** – מחלקת יורשת משתי מחלקות ב"ת, ושתי המחלקות הללו מתנגשות (בעלות מתודה עם אותו השם).
- **דו משמעות אינהרנטית** – מחלקה יורשת משתי מחלקות שיש להן בסיס משותף.
- הגישה היא שהמחלקה התחתונה ביותר שתיתקל בדו-משמעות תתן פתרון אותה בעת ההגדרה.

ב-C++ יש שתי דרכים לפתור זאת:

- ברמת המחלקה , אם מחלקה C יורשת משתי מחלקות A, ו-B שבהן יש מתודה f(), אזי C תגדיר מתודה f() שתקרא לאחת מהן.
- ברמת הלקוח, בעזרת ציון ה-Scope (אופרטור ::)

Generics

- תהליך של כתיבת Template ב-C++ (ובכל שפה) מורכב מ-2 שלבים:

- שני השלבים הללו מתבצעים בזמן קומפילציה, בדיקות הטיפוס מתבצעות בשלב האינטנציאציה וכמעט ולא מתבצעות בעת הגדרת המבנה הגנרי. כשמקבל את הטיפוס, הקומפיילר מייצר את הקוד עבורו. לכן הקוד צריך להיות זמין לקומפיילר בזמן קומפילציה, ולכן הוא צריך להימצא בקובץ header.
- ***C++***, אם לדוגמא הוגדר Stack<int>, וקיימת שורה נוספת אשר מגדירה Stack<int>, יוגדר רק עותק בודד.

- ***ב-Java***, כשהקומפיילר נתקל ב-Stack<int>, הוא הולך למבנה הגנרי ומחליף את T בטיפוס, ואז מחליף ב-Object.
- ***ב-C#***, המחלקה הגנרית מתקמפלת פעם אחת לקובץ Object, ואם יש צורך, נוצרים בזמן ריצה עותקים מתאימים.
- במקרה של פונקציה Template, במקרה של פרמטר שהוא מחלקה, יתאימו גם פרמטרים יורשים.
- במקרה של פרמטר פרימיטיבי, רק פרמטרים שיש coercions ביניהם יתאימו.
- פרמטרים ל-Templates הם בדר"כ טיפוסים, אולם ב-C++ ניתן להוסיף קבועים מספריים (לא משתנים שמחזיקים קבועים!), וגם כתובות של פונקציות.
- חסרון בולט מאוד של ה-Templates הוא ניפוח גדול מאוד של קובץ הריצה.
- לא ניתן ב-Template לשים מחזורת אנונימית, וכמו כן לא ניתן לבחור כפרמטר float, מכיוון שאין לו ייצוג מדויק (למשל שברים מחזוריים).
- גישות לגבי טיפוסים ל-Template:

1. ***no restriction*** – כל דבר שהוא טיפוס חוקי למבנה גנרי, המבנה הגנרי אמור לדעת להתמודד איתו.
   המבנה הגנרי יכול לבצע פעולות שיהיו חוקיות על כל טיפוס, ורק אותן.

   יתרון – executable יחיד (שימוש ב-ML).

   - הנ"ל ***לא מתאים ל-C++***, כי כאן יש הנחה שכל טיפוס מקיים מפרט מסוים.
   - הנ"ל ***מתאים ל-Java***, כי כל האובייקטים יורשים מ-Object.
2. ***dynamic checking*** – בדיקה תוך כדי אינסטנסיאציה.

זאת הגישה של C++. בזמן העברת טיפוס אקטואלי כפרמטר, נבדוק התאמה (סטרקטורלית), למה שהמבנה הגנרי מצפה. הבדיקה היא לפי חתימות של אופרטורים ומתודות.

החיסרון הוא שברגע שישנה שגיאה, לא ידוע היכן היא מתקבלת.

3. כל מבנה גנרי יספק רשימת אילוצים, ומי שעונה עליה, יוכל להיות פרמטר.

זאת הגישה ב-***Ada***. כל מבנה גנרי מספק רשימה של הנחות (מתודות, אופרטורים, ירושה), ובזמן קומפילציה נבדק שימוש בהנחות, ובזמן ריצה נבדק קיום ההנחות.

4. משתמשת בירושה כדי להגדיר טיפוסים שיכולים לעבור למבנה הגנרי.

<u>סיווג מבנים גנריים לפי צורת האינטניסיאציה</u>:

1. אינסטיאציה מרומזת – המקרה של function template ב-C++.

Max(I,j) – לפי טיפוס i,j מחליטים באיזו פ' Max להשתמש.

- החסרון הוא דו-משמעות, לדוגמא Max(2,2.5), יש יותר מאפשרות המרה אחת.

- קיימת דרישה שבה כל פרמטר שהוא פרמטר ל-Template יהיה טיפוס של פרמטר פורמלי לפונקציה. (למשל נגדיר template <T,S> S max (Ta, Tb) – זאת בעיה!).

2. אינסטנסיאציה שתיווצר בפעם הראשונה שנשתמש במבנה הגנרי עם טיפוס נתון – עותק אחד בלבד בכל יחידת קומפילציה.
3. ***אינסטנסיאציה מפורשת*** – יש להצהיר מראש שמתכוונים להשתמש במחלקה כלשהי עם פרמטר מסוים. לדוגמא template class Stack<int>.
   - ניתן לבצע אינסטנסיאציה מפורשת גם למתודה, ולא תיווצר דו משמעות אם הפרמטר של ה-template יופיע כפרמטר פורמלי במתודה.

Java Generics

- ניסיון להימנע מהטעויות של C++ (התפיחה של ה-executable, ואילוצים קוד גנרי ידועים).
- התבססות על אי שינוי ה-JVM.
- ***ב-Java*** גם מתודות יכולות להיות גנריות, גם אם המחלקה לא גנרית.
- ***ב-Java*** יש רק אינסטנסיאציה מרומזת למתודות.
- רק טיפוסים הם פרמטרים חוקיים ל-Generics, ורק טיפוסים שהם Reference type (פרימיטיביים אינם חוקיים).
- בתוך המבנה אסור לנו לקרוא לכל מתודה, רק לפי עקרונות הירושה. לדוגמא, לפרמטר T לא מוגדר מהיכן הוא ירש, אולם ידוע כי הוא ירש מ-Object (חייב), ולכן ניתן להחיל עליו את המתודות של Object, ורק אותן.
- ***ב-Java***, List<Integer> אינו תת-טיפוס של List<Object>, למרות ש-Integer הוא תת-טיפוס של Ojbect. שיטת המימוש של Generics ב-Java שונה ולא מאפשרת לבצע בדיקות זמן ריצה שיאפשרו תמיכה בירושה כזו. (טיפוס זמן הריצה של שתי הרשימות הנ"ל הוא List, ולכן לא ניתן להבדיל ביניהן בזמן ריצה).
- הסימן "?" משמעו שלא ניתן לגשת לטיפוס הפנימי, ולא ניתן לבצע עליו שום הנחות, וידוע רק שהוא יורש מ-Object.
- ניתן לתת ב-Generics גבולות עליונים וגבולות תחתונים באמצעות המילה super.
- המימוש של Generics הוא באמצעות טכניקת type erasure. Generics קיימים רק בזמן קומפילציה. ב-Class files אין כמעט זכר ל-Generics.
   - **חלק ראשון** – הצהרה על המבנה הגנרי: בדיקות טיפוס. משתמשים ב-T רק עם פעולות שמתאימות לגבולות שלו. אחרי שביצענו את הבדיקות, מחוקים את הטיפוס הפרמטרי – מחליפים כל מופע של T בגבול העליון שלו, ומקמפלים ל-Byte code.
   - **חלק שני** – שימוש במבנה הגנרי: הקומפיילר מוחק את שם המחלקה, מתייחס כאילו זה Cell עם Object, וקיימת התאמה בעזרת אנוטציה.