

# סיכום מתומצת לבחן - תוכנות מונחה עצמים

21 ביולי 2025

הסיכום כולל מושגים ו"טריקים" שהוזרים על עצמם ב מבחנים.  
גיא יער-און.  
אם מצאתם טעות אשמה לעדכון :)

## מבוא לגאואה וקטת בסיס על תוכנות מונחה עצמים

- \* אם יש לי קובץ הרצה על המחשב שלי ואנסה להעבירו למחשב אחר, הוא לא בהכרח יroz. "תכן מאוד שלא יroz. זה תלוי בכך שהוא הארכיטורה במוחש ואת אותה משפחת מערכות הפעלה.
- \* בגאואה, המופיע של המחלקות יופיע בתוך *heap*. בגאואה אין מצביעים, יש references. כאשר עושים new לאובייקט זה יוצר רפנס, מאחרי הקלעים קיימת עבודה של malloc.
- \* כאשר אנחנו מעוניינים לפנות לאחת השדות מתוך המחלקה. זו לא חובה להשתמש this. אך אפשר לעשות这种事情 *this.x = x*.
- \* getters&setters: כיוון שאסור לגשת לשדות מחוץ class, אנחנו משתמשים בGETTERS וSETTERS על מנת לקבל ערך של אובייקט או להציג בו. כמו כן שיש למשם אותם.
- \* בגאואה הכל אובייקטים. גם מערכיים ומחרוזות. שדות ומגודאות יחד יקראו . class members .
- \* בתוך מחלקות שונות, ניתן לקרוא למגודאות בשם זהה.
- \* מגודאה פרטית: נניח ונרצה לבצע טסט על מגודאה שהיא פרטית, הבעיה היא שלא ניתן לגשת אליה. יש שיטה לפיה אנחנו יודעים כי אם המגודאה פרטית, אז רק בתוך המחלקה ניתן לקרוא לה לפעולות שונות. היא לא נגלית למשתמש. בודאות משתמשים בה ב镆ודות אחרות, אחרת אין לה נחיצות, לכן אנחנו נבדק אותן, נבדק ונבצע מספיק טסטים על המגודאות שכן ניתן, הציבירות, וכך נבדק גם אותה על הדרך.
- \* לשים לב מהיכן ניגשים לשדה פרטוי - מתוך המחלקה מותר!!
- \* (\*\*\*)אם אני עם משתנה Integer כשדה שלי, ואני ב getter מוחזר this.name שלו, אז אני שובר אנקפסולישן כי הוא לא פרמייטיבי וזה ממש יחזיר את המשתנה עצמו!  
\* מבחינת קומפיילר חוקי לחולtein סינטקס כמו (f).new () וכן למשתנה מסווג זה יהיה טיפוס סטטי ודינמי שייהיו אותו דבר - B במקרה הזה כי יוצר מופיע של B.

## Constructor

תפקידו לבנות אובייקט חדש. אין יותר מדי מה להרחיב. מותר לבנות כמה קונסטרוקטורים שנרצה. ניתן לגשת מותר קונסטרוקטור לקונסטרוקטור אחר. אם הגדרנו מחלקה ללא בנאים, יהיה בניי דיפולטיבי. אח"כ שנבנה מופיע של המחלקה שיש בה בניי דיפולטיבי כמוון שלא לו פרמטרים - כי אין קונסטרוקטור עם פרמטרים שיקבל אותם. מדוע נרצה זאת? לא תמיד נרצה לחייב את הקונסטרוקטור לשלווח את כל הפרמטרים (ומה אם מישחו לא רוצה למלא מין בטופס?....).

העמסה (Overloading) של בנאים: נוכל ליצור כמה בנאים שנרצה, שהשני הוא במס' הארגומנטים שמקבלים. ניתן לקרוא לבנאי אחד מתוך בניי קיימים, באמצעות this, באמצעות מהשורה הראשונה (כמו super).

מה קורא לאחרי הקלעים בהפעלת קונסטרוקטור? ראשית מקרים זכרו לכל השדות של האובייקט (כולל Vtable object), אח"כ נקצה ערך דיפולטי לכל אחד מהשדות של המופיע. אם בתוך הקונסטרוקטור היה קרייה וכן מחלקהsuper, אס אס כתוב וגם אם לא נצורך ללקת לטפלsuper, אם כבר יש

נעשה אותו ואם לא אנחנו צריכים למכת `super` של מחלקת האבא `object` (תמיד תבוצע קריאה `object` בהפעלת קונסטרקטור) וגם אם קרנו לבניי אחר, הוא כבר יקרה לאובג'קט. ובסוף, נמשיך את גוף הקונסטרקטור הנוכחי.

בנאי לא יכול להיות רקורסיבי.  
ערך התחלתי שמאוחתול בבניי (אם לא אתחלו למשהו אחר) הוא אף עבר המשתנים המוכרים, `false` עבור משתנה בולייאני ומעבר `null`.

אם יש לבניי של אבא `Konstruktor` ריק ללא פרמטרים גם אם לא נכתב בבניי של בן `super` הוא יקרא אוטומטית, רק אם יש בניי ריק - אם אין חיבת קריאה ישירה.

אם כתבנו משהו כמו `A.func` בקוד אפשר לדעת בזדות שמתודה `func` היא סטיטית כי הפעלה על שם מחלקת אבל יתכן שהיא שייכת לאבא של `A` ולא בהכרח ל`A`. `new Point(4,5)` יתכן כי `x` שדה סטיטי של אב קדום של `A`. בכל מקרה ברור מהם שייכים לא!!! כי `A` יורש - אבל יתכן שמודדרים אצל אבא של `A`.

## הערה - קונסטרקטור לא חייב להיות `public` מבחינת קומפיילר

### Garbage Collector

מנקה האשפה של ג'אווה. תפקידו "לנקות אחרינו את העבודה". למשל - נניח ועשינו פעולה `point = new point(2,3)`, לאחר מכן נרצה לעשות `p = point.getx()`. `p` בזאתו זה יהיה אפשרי ואיפילו חוקי, הוא ינקה את הערכים הקודמים. יש לשים לב שזה כמובן עלה לנו בעילות. ניתן להפעיל אותו באופן ידני מתוך התוכנית עם `System.gc()` אך זה מאוד לא מומלץ!

### Encapsulation

בעברית נקרא גם כימוס. העקרון הוא פשוט: למנוע מהמשתמש לנסות לשנות לי דברים בקוד. ל Kohr לא צריך לדעת בין היתר כיצד מימושו את הדברים ובאיזה שמות השתמשו. מאפשר למנוע מהמשתמש לשנות את המידע כיצד שהוא רוצה, אניählit מה הוא יכול לעשות. למשל, אם ניצור קונסטרקטור עם נקודה `point` שקיבל כערך, ונתחל בקונסטרקטור `this.point = point`, זו תהיה שגיאה באנפקסליישן שכן ניתן לגשת אותה נקודה מבוחר. מה יהיה הפתרון? `this.point = new point(point.getx(), point.gety())` כתעת, כיון שיצרנו את הנקודה בתוך הקונסטרקטור לא יוכל לגשת אליה מבוחר. המטרה היא להגן על השדות הפרטיים שלנו מmotkpha, `private` לא תמיד יעזור לכך.

### Static Members

סוג מיוחד של `class members`. מדובר בתוכונה ששייכת למחלוקת ואיננה ברמות האובייקט. נשמר ב-`data`. למשל, אם נרצה לספר כמה פעמים יצרנו אובייקט מסווג זה לאורך התוכנית נוכל להשתמש בשדה סטיטי שיסכום בכל פעם שאחננו יוצרים מופע חדש. גם אם לא יצרנו אף אובייקט עוד, הערך הסטיטי קיים כיון שהוא ברמת המחלוקת. ניתן ליצור متודות סטיטיות - מתודות ששייכות למחלוקת. כאשר נרצה לגשת אליהם נעשה זאת עם שם המחלוקת. עם זאת, ניתן לגשת למשתנים סטיטיים / מתודות סטיטיות גם מתוך מופע של אובייקט. זה יתאפשר לא יפה ולא נהוג.

נשים לב שהמחלקה השימושית `math` היא מחלוקת שכל המתודות בתחום הם סטיטיות. מדוע? נרצה שכל המתודות יהיו ברמת המחלוקת, כיון שבג'ואה כל מתודה צריכה להיות מוגדרת בתוך מחלוקת מסוימת, הקודקודים של ג'אווה רואו לנו לרכיב את כולם בספרייה אחת. מתודה שאיתנה סטיטית היא ברמת המחלוקת וairo מתודה שאינה סטיטית היא ברמת האובייקט. אם משתמש הרבה רבסטטיק, כנראה שבחרנו שפת תכנות שאינה לפוריקט שלנו. יש להצדיק שימוש במשתנים ומתחדות סטיטיות.

נשים לב כי גם אם למחלוקת יש רק שדה סטיטי יחיד - היא אינה סטיטית ונitin להגדיר בה מתודות שאינם סטיטיות.

אם יש לי מחלוקת `A` שיורשת ממחלוקת `B`, יש בשתייהן מתודה `f` סטיטית - לא מתבצעת דרישת אלא הסתירה (`hidden`) - כאשר נעשה אח"כ במיין שלי `b = newA()`; וזה נפעיל עליו גם `b.f` זה יהיה הפעלת `f` של טיפוס סטיטי של משתנה.

אפשר לעדכן משתנה סטיטי בתחום בניי - אפשר לגשת אליו מכל מקום למחלוקת ובפרט בתחום. הפעם היחידה שאי אפשר לשנות משתנה סטיטי זה אם נצהיר עליו גם `final` - אז הוא גרסה אחרונה ולא ניתן לשנותו.

## Polymorphism

היכולת לשield טיפוסים שונים לאותו אובייקט. בעברית - רב צורותית, משה יכול להיות סטודנט אך במקביל יכול להיות גם חייל מילאים או אף או לחלופיןABA של הילדה נועה. בעזרת אליו מתודות נتمמש עם מהה? הוא הורי יכול להיות כמו וכמה דברים.

**ממשק** מגע לפטור את הבעיה זו. הממשק הינו הצהרת כוונות של המתכנת שכל מי שimplements(nameOfInterface) יהיה חייב לממש את הפונקציות שנותצאות בתוכו. למשל, אם נגיד ממשק polygon ונרצת שהמחלקה Ribou תמשש אותו, שהרי היא פוליגון, ונניח שבממשק יש מתודה ייחודית שאינה ממומשת: שטח, אז מחלוקת ריבוע תהיה חייבות לממש את המתודה של השטח. אחרית נקבל שגיאת.

"**יתכן** ויהיה מימוש בתחום, אך זו לא המטרה אלא להציג התנהגות. נניח ויש לי ממשק כלשהו, קוראים לו AI, ונרצה לבצע את הפעולה הבא () IA a=new IA, תמיד זה יוביל לשגיאת קומפליציה. הגדרנו כאן new, השאלה היא על מה? אין קונסטרקטור בתוך ממשק ולא יתכן שהייה קונסטרקטור בתוך ממשק. עם זאת, אם יש לי A שמשמש את AI, כן ניתן לבצע פעולות כמו () A a=new A, ושוב בהנחה שאכן A מוגדר טוב עם קונסטרקטור.

ג'אווה מאפשרת ממשקים מרובים, כלומר ניתן לממש כמה ממשקים. **מתודה דיפולט:** ג'אווה מאפשרת להציג מימוש דיפולטיבי בתוך הממשק. השתמש במילה השמורה default וכן פעולה לקובע מימוש דיפולטיבי לתוכנית. זה יכול להיות זמני, כך שבעתיד יוכל לדرس את המתודה ולמשמש אותה במקרה החדש. עם אם ציינו וגם אם לא ציינו, אם נגיד במשמעות  $5 = x - \text{int}$  תמיד זה יהיה פblkיך, סטטייך ופיניל. ככלומר ממש constant.

**בממשק יתכן מתודות פריבט, כמו כן מתודות אבסטרקטיות, סטטיות ודיפולט. מה אסור? פיניל, פרוטקטד ואבסטרקט פריבט.**

**בעיית הילוס:** נניח ויש לנו שני ממשקים B, A שמצהירים על מתודה f שהיא default ואנחנו מעוניינים ליצור ממשק C שרוצה לרשות משניהם. אז - זה לא אפשרי. שכן נהיה בקונפליקט מאייה ממשק לקחת את המתודה. עם זאת, אם במשקדים אין מתודות ממושכות עם אותה החתימה, הורשה מרובה של ממשקים כן עובדת.

**קאסטיינג:** נניח שיש אינטראפיקס חיה ויש מחלוקת כלב שמשמש אותו. באינטראפיקס יש מתודה אחת, a נקבע קאסטיינג הבה () Dog d=new Dog(), ואז את הקאסטיינג d Animal b=(Animal), מה ביצענו כאן? אמרנו למערכת - תתייחס אליו בעת כחיה. כתע, הוא יוכל להתייחס אליו רק באמצעות המותודות של האינטראפיקס, ככלומר a. נניח ויש מתודה bbb בתוך Dog, איז כתע אין לו גישה אליה. **למה בכלל שנעשה את זה?** פולימורפים. אכפת לי מהיה כלשהי, לא משנה אם היא כלב חתול או תוכי.

בתוך ממשק לא יכולות להיות ממושכות מתודות. רק מתודות default יכולות להיות בפנים וכמהן יכולים אפילו לא מימוש.

לאינטראפיקס יש vtable גם: מתודה דיפולט תופיע ב vtable גם אם לא מימשו בכלל אינטראפיקס זה.

לאינטראפיקס יש אפשרות לשים מתודה דיפולטיבית. באופן אוטומטי מי שימושה אותו קיבל אותה. ניתן לדرس את המימוש הדיפולטיבי הזה!!!. כמו כן - כموון שמי שימוש ממשק זה יחזק במתודה הדיפולטיבית שתהיה עבר כל אחד באותה הכתובת. **זה חוקי אך לא לפי עקרונות OOP.**

שדה ייחד שיכול להיות בתחום ממשק הינו public static final int x=5 זה *implicitly* נחשב כאילו כתבנו את כולם וחוקי. בתחום ממשק ניתן לכתוב מתודה דיפולטיבית כמו שהוער לעיל, וכן אפשר לשים מתודה סטטית. כמו כן ניתן גם להשתמש במתודה פריבט בתחום ממשק - אפשר לעשות זאת רק בתחום הממשק ככלומר אי אפשר לגשת אליה מבוחר לממשק גם אם מחלוקת שלו מומשת ממשק (זה בעיקר יהיה עבור חישובים מסוימים בתחום דיפולט שהרי רק מהם אפשר לקרוא להן). **מתודה פרוטקטד, פיניל ומשתנים לא קבועים אינם יכולים להיות בממשק וכמוון שלא קונסטרוקטורים. אבסטרקטן בו שזה הרי כל הרעיון (הן מראש אבסטרקט גם אם לא כתבים).**

## Inheritance

ירושה. אם יש לנו שתי מחלקות זהות עם אותן המתודות אך השם שונה, זה די סותר את עקרונות oop. לשם כך מגיעה ההורשה לתפוס מקום - ניצור מחלוקת שבה נשים את כל התכונות המשותפות והמתודות המשותפות. שאר המתודות יירשו ממנה.

התכוונה override מאפשרת לנו לדرس מתודות. למשל, נניח ונחנו במחלקה A שמרחיבה את B, במחלקה B ישנה מתודה f. אנחנו ירשנו מ B, אבל אנחנו ממעוניינים במימוש אחר של f. אין בעיה - נעשה @override וنمמש אותה מחדש, כפי שרצינו. נראה כי אם דרשו את כל המחלוקת שירשנו, בשביל מה ירשנו בכלל?

מתי מותר לעשות אובייד? אם החתימה זהה, שם המתוודה סדר וטיפוס המשתנים זהה, ערך החזירה במתודת הדורשת מאותו טיפוס או ספציפי יותר וכן נראהות המתוודה הדורשת גדולה יותר. כמו כן, לא ניתן לבצע אובייד final או שיכת למחלקה final והוא לה גרסה קודמת. אחרת - נקבל שגיאת קומפילציה. התוכנה upcasting : המרה של אובייקט ממחלקה בן למחלקה אב. תמיד ניתן לעשות אפקסטינג. באופן סימטרי כמעט, התוכנה downcasting היא המרה של אובייקט ממחלקה אב למחלקה בן. חיבים לכטוב דואוקאסט בצהורה מפורשת וכן הקומפיאילר לא תמיד יסכים לעשות. נעיר כתה כי זה יכול להיות מבלבל: נניח ויש לנו את הדוגמה הבאה -

1. Animal animal = new Dog();
- Dog dog = (Dog) animal;
2. Animal animal = new Animal();
- Dog dog = (Dog) animal;

יש כמובן dog שירש ממחלקה אב Animal, הדוגמה הראשונה היא לחת איזשי חיים, ולחגיד לה "את כלב מעכשו", ככלומר שנמכנו אותה להיות כלב. זה עובד כמובן, שכן החיים הוגדרה להיות כלב, ואז עם קאסטינג אמרנו לה את כלב ואכן יתקמפל. לעומת זאת, בדוגמה השנייה אנחנו החרנו על חיים, ואז ניסינו לכפות עליה להיות כלב. מי אמר שהיא כלב בכלל? היא הוגדרה כחייה. ומה אם היא חתול? הקומפיאילר עומד לאכול את השקר שהאכלנו אותו בזמן הקומפילציה - אך בזמן ריצה יכשל.

**הפעולה super :** מילה שומרה שמאפשרת לגשת לתוכנות מחלקה האב מתוך מחלקה הבן. ניתן מותך הקונסטרוקטור להפעיל super על הקונסטרוקטור של מחלקה האב, ניתן לעשות זאת רק מהשורה הראשונה. עם זאת, ניתן גם להפעיל super על מתחודות אחרות במחלקה האב, למשל אני מגייע לדרוס את המתוודה f, אך כן מעוניין להשתמש במימוש הקודם שלו ולהוסיף משהו, אז ניתן במקרה זה (שאינו בקונסטרוקטור) להפעיל super מכל שורה שהיא.

אפשר להפעיל super ריעונית גם בתוך מחלקה רגילה שלא יורשת מאף אחד - בקונסטרוקטור: תבוצע קריאה לקונסטרוקטור של אובייקט ישירות, ובמתחודות: אם ריצה להפעיל מתחודות של object. לא תמיד חייב להיות סופר, יכול להיות סופר ריק כדיפולטיבי. תמיד נהיה חיבים לבצע סופר במחלקה בן למחלקה אב. אם אין קונסטרוקטור ריק - אז אנחנו חיבים לקרוא מפרשות לקונסטרוקטור. שכן יש קונסטרוקטור ריק רק אם לא דרשו אותנו עם קונסטרוקטור אחר.

**הפעם היחידה שבא לא יהיה סופר זה בקונסטרוקטור של Object.** המילה **השומרה protected**: תוכנה שנוכל להוציא לשדות במקום private. באמצעות התוכנה ניתן להשתמש במחלקות הבן ישרות בתוכנות של מחלקה האב שהוגדרו protected. אך, אם ננסה להשתמש בתוכנות במחלקות שאין יורשות - זו שגיאה כמובן.

יום יבוא ויבטלו את override תאורתית - מה שיקרה הוא שנוכל עדין לעשות מתוודה עם אותה חתימה במחלקה אב ובמחלקה בן, פשוט כל רעיון הדינמיκ בידיניג יעלם - לא יהיה צורך vtable כי פשוט הכל יקבע לפי הטיפוס הסטטי.

גם אם לא כתבנו בתוך מחלקה A כלום אפשר ליצור ממנה new כי יש בנאי דיפולטיבי וגם אם יצרנו B שירשה מאין עדין - אפשר ליצור new גם אם אין ריקות כי הבנאי הדיפולטיבי כאון יפנה לא הריק שיפנהsuper של object.

נראות - נשים לב שבמחלקה יורשת אם עושים דרישת של מתוודה, חייבת להיות נראהות גדולה יותר. ככלומר - יתכן רק בסדר זהה: לא יתכן פרוטקטד למשל במחלקה פרוייבט במחלקה יורשת (באותה מתוודה!).

הערה חשובה: כדי עתידי תמיד צריך כאשר מחלקה B יורשת מאין super אבל בנאי של B לא קיים מפרשות (כלומר jvm בנה לו מஅחרורי הקלעים דיפולטיבי) - אין חובה לעשות מפרשות super במחלקה B! זה המקרה היחיד שלא חיבים. (ובאובייקט כמובן)

כאשר קוראים this מותך בנאי, ככלומר קריאה מבנאי אחד לבנאי אחר, זה חייב להיות בשורה ראשונה ויכולת להיות רק קריאה אחת צו! לא יתכן פעמים קריאה this.

## Abstract Class

מחלקה אבסטרקטית. ישנו מחלקות שלא ריצה שנוכל ליצור מהם טיפוסים חדשים. בשביל למנוע המשמש ליצור אובייקטים מסוימתה המחלקה נוסף לשם המחלקה את המילה abstract. בעת, אם משתמש ינסה ליצור מופיע של המחלקה הוא יתקל בשגיאת קומפילציה. אנחנו נותנים מקום למחלקה בפרויקט, אך לא ליצור ממנה. במחלקה

אבסטרקטית, ניתן ליצור מתודה אבסטרקטית. מתודה אבסטרקטית היא מתודה ללא גוף - יש חתימה ואין מימוש. נමמש את המתודה במחלקה היורשת.

עבור כל מתודה במחלקה, או שנמש שאותה או שנשים לה *abstract* ונטייל את המימוש שלה על מישחו אחר. קלאס שמרחיב קלאס אבסטרקט, לא חייב להיות אבסטרקטי אך הוא צריך למש את הפעולות האבסטרקטיות של המחלקה. אם עשינו מחלקה וכל המתודות בה אבסטרקטיות, אז שהיינו צריכים ליצור ממשק.

יתכן כי יהיה קונסטרוקטור לקלאס אבסטרקטי, על אף שלא ניתן ליצור ממנו שדות. יכול להיות שימושי בתוך הורשה, נגדיר קונסטרוקטור שיאתחל את השדות (אםנס אי אפשר ליצור מופע שלו), אך מי שירש ממנו יוכל לעשות בקונסטרוקטור שלו סופר על הקונסטרוקטור המקורי, וזה ממנו ליצור מופע. אם לא יצרנו קונסטרוקטור באופן ישיר גם כן ג'אווה תכenis אחד דיפוליטיבי.

כמובן שיתכנו שדות למחלקה אבסטרקטית. כמו כן יתכנו מתודות פיניל בתוך מחלקה אבסטרקטית. לא יתכן מחלקה שהיא פיניל אבסטרקט! כמו כן לא יתכנו מתודות שונות גם פיניל וגם אבסטרקט.

אבסטרקט קלאס יכול למש ממשק. ניתן לומר אבסטרקטית מין ולו רוץ כתוכנית ג'אווה רגילה - מדוע? *main* הוא סטטי, אז לא צריך ליצור אובייקט מהמחלקה.

מחלקה מסווג אבסטרקט יכולה להרחיב מחלקה רגילה שאינה אבסטרקטית, ומחלקה רגילה יכולה להרחיב מחלקה שהיא אבסטרקטית.

מחלקה אבסטרקטית יכולה להיות עם מימוש מלא של כל המתודות בתוכה - כלומר יתכן שכולן ממומשות.

לא ניתן להגיד מתודה שהיא סטטית ואבסטרקטית בו זמן. כמו כן, אם כתבנו *abstract* לא ניתן למש בכל זאת את המתודה.

יתכן מתודה פיניל בתוך מחלקה אבסטרקטית (אם כמובן לא תהיה גם אבסטרקטית).

## Class Object

מחלקה מובנית ומהחשובות בג'אווה. באופן אוטומטי, קלאס אובייקט היא מחלקת האב של כל מחלקה בג'אווה. כך, גם אם לכורה נראה לנו שמחלקה לא יורשת מף אחד, היא כן יורשת מאובייקט. עבור מחלקות יורשות, זה נקרא *sub class*. ניתן ליצור אובייקט של מחלקה אובייקט.

**מתודה *getClass*:** מתודה מיוחדת של המחלקה. מוציאה אובייקט שמתאר את המחלקה. ניתן להפעיל אותה על כל אובייקט בג'אווה.

**מתודה *hashCode*:** מתודה של המחלקה שמחזירה "תעודת זהות" של האובייקט.

גם *toString* ו*equals* הן מетодות של המחלקה. עליהם נהריב מטה.

**פעולה *getClass.getName()*:** יוצרת אובייקט, חשוב לזכור זאת. ניתן לקבל את השם של האובייקט עם *getClass.getName()*.

## Equals

פעולה שקיימת בכל מחלקה (כי כל מחלקה יורשת מאובייקט). עלינו לדרוס אותה ולמש אותה בעצמנו שכן הערך הדיפולטי שלה הממוש לא שווה כלום. הפעולה הדיפולרית בודקת האם הרפרנסים זהים ולא אם התוכן זהה.

**הערה חשובה:** מחלקה *String* כן שכבה את הפעולה *equals* ואמ נבדוק (*t.equals(s)*) על שתי מחרוזות זהות בתוכנן, כן נקבל *true*. מה שנכתב מעלה הוא על מחלקות אחרות שcreativecommons לשכתב פעולה זו מוחדש. באופן דומה גם *Double, Integer, Long, Short*

תמונה מקורס "תכנות מתקדם" שצביבה הזכיר: *StringPool* - נניח ויש לנו מצב בו *s = "hi"* ו- *t = "hi"*, אם נעשה *t == s*, נקבל *true*. מדוע? ג'אווה החלטה לשמור בזיכרון באותו מקום מחרוזות זהות, אך, למרות שמתבצעת השוואת רפרנסים, נקבל בכלל זאת *true*. לעומת זאת, יש לשים לב שבהינתן שני מערכיים אם נעשה *arr1 = arr2* על שתי מחרוזות זהות בתוכנן, זה יעתיק את הרפרנסים ולא את הערכים, וכך שינוי במערך אחד ישפייע על המערך השני, שכן יש להעתיק את כל הערכים אחד אחד ב(*O(n)*).

כמו כן, ישנה דרך לעקוף סטירינגל. נשים לב שאכן ג'אווה רוצה ליעיל ושמורת אובייקטים עם אותו שם באוטו מקום אך אם כתבתني ("hi") *s = newString("hi")* ו- *t = newString("hi")* כאן נוצרו שני אובייקטים שונים! עקפתה את זה והם לא נשמרו באותו מקום ולכן *t == s* יחזיר *false*.

## Tostring

פעולה שנמצאת בכל מחלקה (כי כל מחלקה יורשת מאובג'קט). גם לה יש ערך דיפולטי שאינו שווה לכלום. הפעולה הדיפולטית מדביקה "האש קוד" מזאה לכל אובייקט וכאשר נפעיל את הפעולה נdfs את האש קוד ולא את התיאור שהשכנו שלכורה יודפס. נרצה לדרוס מודה זו.

מימושו *toString* כבר בזרה תקינה.

## Wrapper

בגאווה קיים *wrapper – class* עבור כל טיפוס פרמייטיבי, אסתיניג בין primitive *wrapper* שלו בשני הכוונים נעשה באופן אוטומטי ע"י קומפלייר. מעבר מ *int* ל *Integer* נקרא *autoBoxing* ומעבר הפוך נקרא *unBoxing*.

למשל דוגמה: יש לי משתנה *inta = 1* ויש לי מודה *o* *print(Object o)*. אם נפעיל *print(a)* קוד יתקפל. אובייקט הוא אב קדמוני לכל קלאס ולכנן אם נעשה אוטובוקסינג קומפלייר מבצע אימפליסיטי אסתיניג מאינטגר לאובייקט ולכנן קוד עובד ומבצע כנדרש.

לעומת זאת, קוד דומה על מערכים למשל *int – A = [1, 2, 3]* לא יעבד, צריך להכיר את זה כי לפעמים שפות תכנות בכלל וגאווה בפרט מתנהגות בזרה לא אינטואטיבית. במקרה הזה קומפלייר לא מסכים לבצע אוטובוקסינג ולא אימפליסיט קסטיניג מחוש בעיות שעולות להיווצר. אין דרך לנחש מקרה זה - אלא להכיר. **לעומת זאת אם הינו**

**מגדירים מערך Integer – A ואו הינו משתמשים במתודה פרינט מסוג print(Object[] A) או זה כן היה חוקי!** את אותה פעולה קודם ניתן לעשות גם מטיפוס גנרי עם מודה *<T> void print(T[]A)* public static <T> void print(T[]A) שלמעה שוב אם נשלח *IntegerA* [*1, 2, 3*] כן יעבד עלייה. קומפלייר ימיר זאת לאחר קומפלייצה *Object*.

## instanceOf

הפעולה מהזירה האם אובייקט הוא מסווג אב קדום של אובייקט אחר. למשל *smartPoint instanceof Point* יניב *true*. אם זאת, נשים לב שלא יכול לדעת האם אכן קיבלנו אובייקט מסוים. שכן, אם יהיה לנו *smartPoint* שבן *Point* הוא אב קדום שלו. ולכנן בשביל לשכתה פעולה *equals* אשר נשתמש בשורה הבאה:

```
if(other.getClass().getName().equals(this.getClass().getName()))
```

מה כתבנו כאן? אם שם המחלקה של הנוכחי של *other*, זהה לשם המחלקה של *other* איתו אני משווה, אז אנחנו יודעים שמדובר באותו טיפוס. ככלומר לצערנו *instanceOf* לא פותר את הבעיה זו.

חשוב - משווה לפי הערך הדינמי של משתנה ולא סטטי.

## Final class

מחלקה שהינה פיניל קלאס היא מחלקה שלא ניתן לרשת ממנה. מודה *final* היא מודה שלא ניתן לדרוס אותה, ובמסגרתה נקבל שגיאת קומפליציה.

אם יש לי שדה פרט שהוא פיניל - מתוך הקונסטרקטור יוכל לשנותו אמ"מ הוא עוד לא אוחחל - אפשר רק פעם אחת.

צורך לשים לב *final* לא גורר *immutable* ובפרט אם כל המתודות של המחלקה וכן המשתנים הם *final* זה לא גורר *immutable*.

אפשר לעשות *final* למודה פיניל (והיא כמובן לא חייבת להיות פיניל המתודה שעושים לה *overload*)

## Static&Dinamic biding

מה זה אומר? נתבונן בדוגמה – *Polygon p=new name*, הטיפוס הסטטי שלנו הינו *polygon*.

בזמן הריצה, הקומפלייר בונה טבלה בשם *vTable* לכל מחלקה בארכון. הטבלה מחזיקה חלק מהכטבות של המתודות במחלקה. אם מישחו ירש, הכתובת למודה שהוא ירש תהיה הכתובת למודה של האבא. הקומפלייר דוחף שדה *vptr* לכל אובייקט שייתאר את הכתובת שלו בארכון. בזמן הקומפליציה, הקומפלייר מתייחס לטטטיק בידיניג בלבד. זה מה שהוא רואה בזמן הקומפליציה. בזמן הריצה, הוא יכול לגלוות דברים אחרים בדמות הטיפוס הדינמי. השאלה שנשאלת – מי מופעל ומה? האם תמיד מופעל דינמי בידיניג?

**סטטיק בידיניג:** מתרחש בזמן הקומפליציה, יותר מעתלים בזמן ריצה, מופעל עברו מותודות סטטיות, מותודות,

*private*, קונסטרקטור וסטטיק פיניל. כמו כן, אם החרנו על משתנה בתחילת המחלקה, *x* כלשהו, אז גם עלי

ופעול סטטי בידינגן. ככלומר אם  $x = \text{new } A$  הוגדר במחלקה  $A$  ומבצע  $x = \text{new } B$ , אז נעשה  $x.a$ , נקבל את הטיפוס הסטטי, ככלומר נלך  $x$  של  $A$  ונקבל 5. **דינמי בידינגן** - מופעל בזמן ריצה, על מתודות רגילות, מתודות שהן אובייד, אבסטרקטיות וכן פינל. הקומפיאיר לא ידע מושג מותודה תופעל. במהלך הריצה הקוד יבחר איזו גרסה של הקוד להפעיל, בהתאם לסוג האובייקט שבחזרון. אם אפשר להמנע ממנה - יחסוך בזמן ריצה.

**מאותה מסוג אבסטרקטית צו תכנס לזכרן, ותבצע על null.** ככלומר - אם יש לי מחלקה אבסטרקטית ובה מותודה אבסטרקטית, ומחלקה שיורשת שэмמשת אותה. כMOVED ב  $vTable$  של היורשת תופיע המותודה, ובמחלקה הاب יופיע עבור המותודה ב  $vTable$  פשוט  $null$ . **באופן כללי הרישום null בתוך vTable עבור מותודה מעיד של מותודה אין שימוש בklass השפציפי זהה.**

כאשר אני כותב משהו כמו  $f(b)$  כאשר  $C$  מושג מ  $A$  ירש מ  $A$  אני אומר לקומפיאיר **תעשה upcast** = חוקי למגררי, **תתייחס לברמה הדינמית כמו קודם וברמה הסטטית כ- $A$ .**

**נכns לVtable: מותודות רגילות**, **abstract** מותוד'ס עם  $final, static, private, protected$ . **לא נכנס?** אם קלאס שמכיל מותודה מסוג  $final$  מרחיב קלאס אחר אז המותודה  $final$  כן תכנס  $vTable$ . כמו כן אם הטיפוס הסטטי של הרפרנס הוא לא המחלקה שמכילה את המותודה  $final$  יתבצע קריאה למותודה לפי דינמיק בידינגן.

ונוצר רק אחרא קומפיאציה ולכון בהכרח הקוד התקמל אם אני מקבל טבלת  $Vtable$ .

לסיכום:

אבסטרקטיות	- דינמיק בידינגן
פריבט	- סטטי בידינגן
סטטיות	- סטטי בידינגן
פינל	- דינמיק בידינגן
סטטי פינל	- סטטי בידינגן
רגילות (void)	- דינמיק בידינגן
קונסטרוקטור	- סטטי בידינגן
העמסה	- סטטי בידינגן בזמן קומפיאציה

## Overloading

נרצה לפעמים מותודות בשם זהה. מדוע? נניח ויש לי רצון לעשות מותודה ששואלת שאלת מסויימת, אבל יתכן שימושו ירצה להזכיר ייחד עם השאלה פרמטר בוליאני, מישו אחר לא יכנס כלל ואחר יכנס משתנה ממשי. מה נעשה? אפשר על הננייר לעשות שלוש מותודות בשמות שונים ולהשתמש במידת הצורך בכל אחת אבל זה מסורבל. ניתן לקרוא למותודות הללו באותו שם, מצב זה נקרא העמסה.

מתי מותר לבצע העמסה? אם יש שניים בכל פעם בארגומנטים שלחצנו, אסור מצב של: אותו שם, פרמטרים זרים, שניינו רק בערך החזירה של המותודה - זה מצב שנקריא **ambigius** (דו משמעי)

שם שונה הארוגומנטים לא רלוונטי, אלא רק הטיפוס שלהם והכמות שלהם. למשל, נניח ויש לנו מחלקה  $B$  שמרחיבה את מחלקה  $A$ . נניח ועשינו  $(A x, B y), f(B y, A x)$  קיבל שגיאת קומפיאציה שכן אותו ערך החזירה - אותו שם - ומש' המשתנים ששלחו לי זהה. לא נכון.

**הערה חשובה** - כיצד נקבע לאיזו גרסה של מותודה תבוצע? **קומפיאיר** בוחר מה החתימה של המותודה שיש להריץ לפיטיק בידינגן של ארוגומנטים שהעבכנו למותודה, ובזמן ריצה מופעל דינמיק בידינגן שבודח בין המותודות בעלת החתימה שנבחרה את המימוש הדורס האחרון.

## Generics and collections

אנחנו נמצאים בתוך מחלקה של נקודה ורוצים למיין את הנקודות. מצד אחד יוכל למיין לפי הקבוצה לציר האיקס או לראשית הצירים או לפי צבע הנקודה. לפניינו אפשרות למש כל אחת מהאפשרויות הללו אך זו כפילות קוד וזה נגד עקרונות **oop**. לכן נשתמש **genrics**,創ת נוכל להגיד אינטראפיקס לכל סוג של אובייקט. אך אם יהיה לנו אלף אובייקטים, ניצור אלף אינטראפיקסים? וזה לא שלא. אנחנו נזכיר創ת את הממשק **comparator** שנמצא כבר בילד אין ביג'ואה. הוא נראה כך -

```
point interface Comperator <T>{}
```

```
int compare ( T a, T b);
```

創ת, כאשר ניצור מחלקות ונרצה למיין אנחנו נמש אותו וنعשה אובייד על המותודה **compareTo** ונמשה מחדש.

כעת נדבר על *collection*: אוסףים גנריים מאפשרים שימוש בתבנית ייחוד לאוסף כללי כלשהו. נוכל ליצור אוסף *mbali* לצין את שם הטיפוס. יש הרבה אוספים כמו *List < E >* וכן *Set < E >* וכן *Queue < E >* ועוד... כאשר נרצה ליצור אוסף מטיפוס מסוים אנחנו מתחייבים אל הטיפוס הזה בפני הקומפיאר ואם כמובן ננסה להפר זאת לאחר מכן נקבל שגיאות קומפליציה.

עיר שבדינו אפשרות אחרות: מדוע שלא נגידר אוסף בצורת מערך מסוג *object*? הריcolm יורשים ממנו. ובכן, זה חוקי אך לא נשמש כי זה מסורבל - בכל פעם עליינו לזכור את הטיפוס של האיבר שהכנסנו וזה יוביל בעיות כמעט שונשח מי בידינו וזה יוביל לשגיאת זמן ריצה.

כלאש עם סימן שאלה *Class < ? >* – אין חנו נקבע בזמן שאנו נשלח את הטיפוס המדוק, אם נגידרו להיות *string* ולאחר מכן נשלח איבר שאינו *String.class* נקבל שגיאת זמן ריצה. נשים לב *Class* זו מחלוקת ג'אווה שהיא גנרטית. כיצד נשלח לבניי של מחלוקת הסימן שאלת את הטיפוס? יהיה הדרך לשולח. זו דרך נחמדת לא מומלץ להשתמש בה. עדיף גנרטס. בפירות יותר אודות > ?<: מוטר רק לקרו ממנה נתונים ואסור להכנס אליה משהו. מדוע? זו הייתה רשיימה כלשהו למשל *List < Integer >* וקיבלו אותה כ> ?<. אין לנו מושג מה היה הטיפוס בה קודם ולכן אם נכניס כנראה שנחרוס. מוטר רק לקרו באמצעות *Object-val = list.get(i)* למשל כי כל טיפוס בג'אווה הוא בפרט אובייקט. אם מראש יודעים מה סוג הטיפוס למשל *string* אז אפשר להכנס רק טיפוסים מסוג סטרינג, הכנסת אינטגר למשל תביא לשגיאה.

**לסיכום:** *Comparable < T >* הוא ממשק להשוואה טبيعית ונימושו בתחום המחלוקת, חובה למשתמש *compareTo()* שנותנת מימוש יחיד *Comperator < T >* ממשק להשוואה חיצונית, אפשר למשוח מחלוקת, נשלח אותו כ*cmp* ארגומנט בשביל למיפוי יחס סדר כלשהו. מחייב למשתמש *compareTo()*. *compare()*. עצם קיומו מאפשר *sort* בצורה גנרטית. תתכן מותודה *comparable* אם לא מימושי *compareTo* (הרי זה שם) אך ההפק לא הנכו.

ברמה סינטקטית – *T* אומר שהוא דטה טיפ גנרי, אם מופיעים (*T, T*) יתכן שהוא מאותו טיפוס פולימורפי. אם יופיע *T, V* זה אומר ששלהנו שני טיפוסים גנריים שונים, יתכן שהם שווים אך לא בהכרח קיים קשר (אפיו פולימורפי) בין השניים. בסופו של דבר הקומפיאר יתרגם הכל לאובייקט אבל זה רק אחרי שיבדק חוקי.

כל מבני הנתונים הגנריים בג'אווה מימושו את *toString* ואת *equals*.  
הו *String*, *Double* וכו'... מימושות *comprable*.

## File IO

**(לא יהיה במחהן)** נרצה לבדוק עם קבצים. *streams* יהיה זרם של נתונים ממוקם למקום – כמו זרם של מים. אנחנו פותחים קובץ, כתע ניתן לקרוא ממנו تو תו, ובסוף נסגור אותו בשביל להמנע מדילפת זכרו. כך עובדים בכל שפה עם קבצים. אנחנו עובדים על האובייקט שמקשור לקובץ ולא על הקובץ עצמו. כאשר נעשה (*File.read()*) אנחנו מושכים יחיר (*File.read(1)*) שמורכבת מביט אחד של הקובץ, אנחנו משכנו ביט ולא בית יש לשים לב. כאשר תגמר הקיראה, יחיר (*File.read(-1)*) וזה נדע שהקובץ הסתיים. את התווים נניח לשמורתו בתחום מערך, כתע הפעלה *write* יכולה לחת את התווים ששמרנו במערך ולהתחליל לכתוב אותם. ניתן גם לקרוא שורה עד להופעת בקורסיאן ראשון עם (*readLine()*), הבפירידר יבודע עד לקליטת בקורסיאן והוא יעתיק ביט לתוכו עד שיראה בקורסיאן, וזה אפשר להדפיס את השורה שהייתה בו והוא יירוקן כל מה שהיא בתוכו. כאשר נבודע עם *files* אנחנו נוסיף לוכרטת זורק שגיאה במידה ולא הצלחנו מסיבה צו או אחרת לעבוד עם הקובץ. כיצד ניתן לכתוב לתוכך קובץ? עם הפקודה *FileOutputStream*.

שים לב – כשאתה עושה *close* זו לא שגיאה לסגור שהוא כבר סגור! (סגור אותו כמה שתרצה זו לא שגיאת קומפליציה)

## Exceptions

מדובר בחירגות. אנחנו מדברים כתע על מושג של לא תקין (ולא תקין אך לא נכון כמו עקרונות *oop*). יש אפשרויות בג'אווה לבדוק האם קובץ קיים, נ nich ובודקי והוא קיים. יתכן שרגע אחריו יעלם כי מישחו מחק אותו באותו הזמן. זו בעיה עבורי בקוד, אני צריך לדעת זאת כאשר אני מתעסק עם קובץ. אנחנו נחלק את הנושא לשניים: בעיות שליטתו ובעיות שלא בשליטתו אך צריך לטפל בהם.

מה יקרה כשזהה שגיאה? מערכת הפעלה תהיה המעורבת והתהיה הראשונה שידועה על הבעיה. מערכת הפעלה של שולחת סיגナル *JVM* (java virtual machine), שאומר לה: תעשה משהו עם הסיגナル אחרית התוכנית שלך תקרוס. יוצר אובייקט מיוחד מסוג *error* שנזרק ככדור לתפוס את הבעיה בתוכנית.

בשביל שהתוכנית שלנו תוכל בכלל לתפוס חריג אנהנו צריכים לעטוף את הקוד עם `try`, ככלmor נסה לעשות מה שתרצה. אבל בסוף תגדר בлок של `catch`, אם וכאשר יגיע הבדיקה זה - אנחנו נתפס אותו בבלוק הזה. לח堤מת המתוודה יטרוף throws exception . נשים לב שניתן לשים כמה `catch` שנרצתה, אבל זה כמו `switch case`, ככלmor אנחנו הולכים תמיד לרגע ראשון שמאופיע ולכון את ההכי ספציפי נשים הכל למלعلا.

כאשר תפכנו את החריגה, ניתן להדפיס אותה בשביל לדעת מה השגיאה עם `System.out.println(e.getMessage())`. זה ידפיס הודעה מיוחדת בטרמינל. יוכל גם להדפיס stack trace בשביל לעקב אחריו קראות למתוודות שהובילו לחריגה. כמו כן יש `activationFrame` שם יש שמייה של הזרון של כל המתוודות.

המחלקה העיקרית בה רוב השגיאות נקראות `IOException`, יש בה שגיאות כמו קובץ לא קיים, אי אפשר לעבוד עם קובץ כי אין הרשאה, וכו'. יש מחלקות שיורשות ממנה כמו `FileNotFoundException`, שמהם ניתן להבין איזה חריגה תזורך.

תמיד עליינו לסגור לבסוף את `stream` על מנת למנוע מאיבוד זכרון. כיצד נסגור אותו בתוך `catch`? הרי אם נסגור בסוף המקירה הראשון, לא יוכל להגיע קדימה. על הניר, יוכל לשים `try` ו- `finally` בכל סוויצ' קיס, אך נקבל כפילות קוד **וכאן זה לא טוב לפיערכנות oop**. (אך זו לא שגיאה מובן). לשם כך נגיד `try` - בכל פעם כאשר עשה טרי אנד קז' נוסף `finally`, מדובר בבלוק של הקוד שמתבצע תמיד. בין אם קرتה השגיאה או שלא. הוא יROUT תמיד אחרי הביצוע בשטאף את השגיאה. תמיד הקוד שם יתבצע. מתי לא? מקרה אחד בדיק - אם יש פעולה של `catch` בו יצאו.

**חשוב לציין - תחבירת לא חובה בכלל לשים `finally` וסבירן שלא חובה `catch` תחבירת!**

**עלינו לדעת שבכל שפה יש שלושה streams** `System.in` להדפסות ו- `System.out` להקלדה למחשב, `System.err` לשגיאות.

מתוודות שזורךות שגיאה: מי שקרה למתוודה כזו חייב לטפל בשגיאה וליצור לה טרי-קז', בקריאת למתוודה יש לטפל בכך. לאחרת תהיה שגיאת קומפליציה. בගואזה הדרך הנכונה היא לטפל בשגיאה היכן שקרה לנו למתוודה, שכן הטיפול הינו פרטני. האחריות היא תמיד על מי שקרה למתוודה. נשים לב שינוי בעיה - אם תזרק שגיאה לא נגעה לשורת `close` כי בזמן `try` אנחנו נקבע שגיאה ולא נחזיר למתוודה כלל, ושם הרי סוגרים את הסטרים, אז לא נסגור אותו ותהייה דליפה. מדוע אי אפשר לסגור אותו דרך `main`? הרפרנס לקובץ הינו לוקלי של המתוודה, בלא מכיריהם אותו ואי אפשר לסגור אותו. לכן אנחנו נפעל בשיטה שנקרו `try with resource`, למשה אנחנו נשים את הבארפר בתוך `try` וכאשר אנחנו נצא מהתוכנית זה תמיד יסגור אותו בלבד - וירד מאייתנו הלחץ. מימוש של זה יראה כך -

```
try(BufferedReader reader= new BufferedReader(new FileReader("a.txt")))
```

**חשוב לשים לב! תמיד לקרוא `close` ב... `trywith` catch ואל ה... `finally`.**

**סוגי שגיאות:** יש סוגים בעיות שנitinן לתפוס אך זה אחריות המתכנת לטפל בבאג, ולא אמורים לשלוח שגיאה. זה נקרא `check`. אם לא נטפל הקומפיילר יкус ולא יתון לי למפל. לעומת זאת יש שגיאות שכן לא באחריות המתכנת - **לכן נעשו עליהם טרי וקז'**. למשל מה לא באחריות המתכנת? קובץ לא קיים וכדומה. זה נקרא `unchecked`. **בגואזה החלטתו ש `main` יכול לזרוק exception**, מי שנמצא מעלה `main` הינו JVM שיתפוץ את השגיאה והתוכנית תקרוס.

אין חובה שהexception שנטפס תהיה בהכרח קשורה לבעה בקוד, זה אמן חסר טעם אך כן יתקמפל. כמו כן, מתוודה יכולה לזרוק `unchecked` חריגה גם מבלי להצהיר בחתיימה ...`throws`

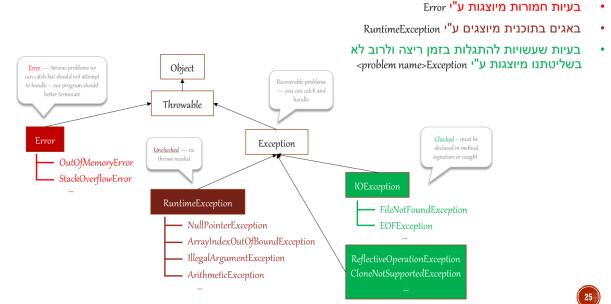
**סיכום:**

מקרה ראשון - לא נזרקת חריגה בתוך `try`: קוד בתוך `try` רץ כרגע ולא נכנים לשום `catch`, בסיום `try` המערכת סוגרת את כל המשאים עם `close` (חווב `close` חייב למשם ממש `closeable`), הולכים אל `finally` ומשם יוצאים מהבלוק להמשך הקוד.

מקרה שני - כן נזרקת חריגה בתוך `try`: הקוד עוצר בעת החריגה היכן שהיא קرتה ולא ממשיך את השורות הבאות, לפני ש קופצים לקאז' ופינלי **קובץ כל close נסגר**, אח"כ מתחפשים `catch` מתאימים לחריגה - אם זה צ'ק' אז קיימת צ'ז' באחרית הקוד לא היה מתקמפל, אם זה אנט'ק' אז הקוד יתקמפל ואם לא ימצא צ'ז' בשלב זה הקוד יקרס בזמן ריצה, אם מצאנו קיים - מרים אותו, אח"כ כתמיד `finally` בסוף ואז ממשיכים רגיל בקוד (אם לא הייתה שגיאה בזמן ריצה)

**מה קרה אם בתוך `catch` זר��נו חריגה חדשה?** הקוד יעצר ויתיחס רק לחריגה החדשה, זה נחשב כאילו יצאת מבלוק טרי וקז' וזה מփש מבחן אליו לתפוס אם קיימים בכלל (מחוץ ל `try` נוכחי), אם אין אחד שמתאים לה התוכנית תקרוס.

## EXCEPTIONS HIERARCHY



## Design patterns

מה זה עיצוב? תכנון נכון של התוכנית שלו. בג'אווה יש המונח **תבניות** שעיצוב שהוצעו כבר למצבים ייחודיים. 1. **האלצה**: *Decorator / Delegation*. נשתמש בו כאשר נרצה לחשוף את האובייקט עם תכונות נוספות. למשל, נרצה להוסיף צבע לכדור או לסובב אותו וכדומה. נוסיף מחלקות שיקשו אותנו מסביב. מדובר צרייך אותו? **נובע מכך** שלא ניתן לרשף **משני** מחלקות ולבן הftptron **הוא להשתמש בתבנית עיצוב** - נוסיף שדה **ball**. הקשتن מאפשר ליצור מחלקה בסיס עם תכונות והתנהגות שיגדרו מראש, ולהוסיף תכונות רצויות (למשל לצבעו אותו) באמצעות הקשتن. בнерאות הקוד זה נראה כאילו ממש אנחנו מקשטים את האובייקט הפנימי ביותר החוצה. הקשتن אומר שהוא פשוט - תעשה קודם מה שלימדו אותך לעשות, אני מבטיח לך שאשמור על התנהגות הזה, וארכיב עליה בדרך שאמרת לי לעשות (קשتن מרחב - *extend*). למשל - אם נרצה ליצור מושולש מסתובב עם נקודות, לא נוכלشم שמשולש ירש את משולש עם נקודות ומשולש מסתובב, لكن נקשט אותו במושולש מסתובב ומשולש עם נקודות. כל אחד מהם ירחיב בדרך שצווה לעשות. כך זה יראה במימוש - *ITrianglerst = newRotatingTriangle(newSpottedTriangle(newEquilateralTriangle((((....))))))*. הקשتن מרחב את התנהגות בזמן הריצה, לא בזמן קומPILE, ולכן גמיש וдинמי. הטיפוס המקורי יכול להקבע בזמן ריצה ולכן דינמי. מה החסרונות? פחתות יעיל, לא קרייה וקשה למתקבב. בהורשה נשתמש כאשר אנחנו מתכווניםhirercity מחלקות ומתקנים עקרון *a - is* (מחלקת בן היא סוג של מחלקה אב) וכן כאשר אנחנו מכוונים שכל תכונות הורה יכולים גם על הבן. אחרת, נעדיף האלצה. **זה כדי חשוב שמתחרה עם הורשה אך לפחות נעדיף להשתמש בו מאשר לא מתקנים עקרון ההחלפה (כל שימוש ברגע ועתידי בא היה מתאים לשימוש B)**

دلינגיון בעקרון זה כאשר אני רוצה להפנות הלאה טיפול למשהו אחר - באה אליו בקשה להרבה של מותודה ואני רוצה להפנות אותה לשדה שלי שיעשה זאת אבל *extend* לא מתאים כאן כי האובייקט הזה לא בדיקן אני - למשל אפשר להציג את טליה ממבני נתונים עם מרינה על הקורס כי יש קשר בין השניים אבל עדין אין בהם עקרון *isa* בין מרינה לטליה. (זה בול הדוגמה עם הריבוע)

קשتن - הוספה של התנהגות ו/או שדות תוך כדי שירה על שדות והתנהגות מקוררים. הקישוט מתבצע ע"י האלצה כל הפקודות לאובייקט המקורי, בתוספת שדות והתנהגות רצויים.

2. **Observer/listner**: נרצה אובייקט שנוכל להסתכל עליו ולהאזין לו, כאשר נשנה משהו שככל המוקומות שקוראים לו גם יעשו משהו. יש קשר ביניהם. בכל פעם שקרה לו משהו הוא יבצע *notify* לכל האחרים שיתעדכו גם כן. למשל, כאשר כדור יעבור מהירות מסוימת נרצה להקטין אותו, אז נבצע *notify*. התבנית מאפשרת לאובייקט אחד להודיע אחרים הצופים בו על שינויים במצבו, כך נוצר קשר דינמי בין האובייקט לבין הצופים. הצופים מקבלו עדכונים באופן אוטומטי כאשר הם מתרחשים. **האובייקט הנצפה - subject**: מנהל רשימה של *Observers*.אפשר *notify* להירשם או להסיר את עצם מהרשימה. מכל פונקציה(*notify*) ששולחת עדכון לכל ה-*Observers* הרשומים. **הצופים**: *Observers*: אובייקטים שמחוברים ל-*Subject*. מקבלים הודעה אוטומטית כאשר יש שינוי ב-*Subject*. מיישמים ממש מוסים, לדוגמה, *update()*.

3. **Fecatory**: בניית עיצוב שמאפרידה בין יצירת אובייקטים לבין השימוש בהם. מאפשר ליצור אובייקט מוביל לדעת מה הטיפוס המדוקיק של האובייקט. בכל פעם המפעל מייצר אובייקטים בהתאם לבקשת המשתמש. למשל - משחק איקס עיגול. מכרכנו אותו לקובוח והוא בא יום אחריו וומר שהוא שדרוג במשחק להוסיף *smartUser*. איך נמסים? הרוי צרייך לעדכן את המשחק. הftptron יהיה יצירת ממשק בשם *supplier* וכן מחלקות שימושו את הטיפוס המסוים במקורה שלנו: שחזור מול רובוט, שני שחknim, שני רובוטים. אנחנו יוצרים אובייקט מוביל לדעת את הטיפוס המסוים של האובייקט. מתי נשתמש בתבנית זו? כאשר מותודה במלקה מסוימת מבקשת ליצור מופיע של אובייקט ספציפי, אך איןנו יודעים מראש מיהו כי זה תלוי למשל בבחירה המשתמש. לכן במצב זה נשתמש בתבנית זאת, וניתן לו כמה אפשרויות ליצור מהם (ואז הוא יבחר, ובהתאם לסוג'ץ' קיים נגידר את השחקן). בתבנית עיצוב זו יש חלוקת תפקידים די ברורה, המפעל אחראי לספק לי אובייקטים וליצור אותם והמתודות אחרות על השימוש. **הsupplier** יהיה אחראי לתת למפעל אובייקט, והמתודות ייכבו אותו דבר ללא שכפול קוד וייהיו מוכנות לעבוד עם אובייקט כללי.

הערה - **יתאים יותר למצבים מורכבים בתוכנית שלנו** (ניו בתוך ניו....) **נשתמש בו בעיקר כדי להגדיר מחלקה** **שמכילה מתודות שבונות שלובים נפרדים ומורכבים.**  
**בעקרון לסטיקום פקטורי מאד פשוט**, כשהוארים לו `get` מביא ל' אובייקט, מייצרים פקטורי כשלא מסוגלים לבצע `new` ישר כשאני לא יודע איזה טיפוס אני רוצה. אך הבנייה עצמה של אובייקט פשוטה בណיגוד לבילדר.  
**לפי מרינה סטודנטים מtablbulim בין פקטורי לקלון, קלון זה שיש לי כבר אובייקט ביד ותעתיק אותו ופקטוריו זה תן- לי אובייקט. כמו כן לפי מרינה גם מtablbulim בין פקטורי לסינגלטון - אין קשר בסינגלטון אתה מגיביל במות ובפקטוריו יוצר אובייקט.**

5. **singletone:** נרצה להגביל את מס' המופעים של אובייקט למופע יחיד (או מס' מופעים קבוע), כיצד? נגיד במחלקה שדה סטטי בשם `instance`. נהפוך את בנייה המחלקה לפרטוי, וכך נמנע יצירתה של אובייקטים מחוץ למחלקה. כמו כן, ניצור מתודה סטטית שתחזיר את המופע. היא תבדוק האם `instance = null` ואם כן היא תקרא לבנייה, ותחזיר את המופע לשימוש. אחרת, הוא כבר קיים ויש לנו מגבלה של אחד ולכן היא פשוט תחזיר את השדה. על הניר **סינגלטון זה רק עם שדה סטטי אבל עקרונית ניתן למשם באמצעות פקטורי דזין פטרו.**

**טוב לדעת -** הסיבה שモתר לעשות בג'אווה `for (int num: collection) {lomer}` **for each** (int num: collection) **כולם** היא שמאחורי הקלעים ממומש איטרטור של המחלקה שנקרה.  
**נשים לב שיש iterator בעיה.** דבר שיפנה אותנו למושג הבא Nested class , מה הבעיה? לפעמים האיטרטור כן צריך לדעת את המבנה הפנימי של האובייקט, רק עבור האיטרטור עצמו, המשמש לא ידע. אבל אי אפשר לחוש את המידע הזה ברגע מבון. לכן אנחנו הולכים לשימוש במושג שהזכרנו, מדובר במחלקה בתוך מחלקה. ישנה מחלקה ובתוכה מחלקה פנימית. המחלקה הפנימית כן מכירה את תוכנות private של המחלקה הראשית וזה החידוש. נגיד את המחלקה הפנימית כמחלקה פריבט וכעת גם אי אפשר בכלל ליצור אובייקט ממנה. כתע, יוכל ליצור את האיטרטור מבפנים ולא ישבר עקרון האנקפסולציה. למעשה מדובר על בנייה של האיטרטור מבפנים המחלקה.

## Cloning&Immutability

- לא ניתן לשינוי וכן `Mmutable` זה כן ניתן לשינוי. יש שפות רבות שלא ניתן לשנות בהם, למשל אם הגדרנו `x = 4`, נגמר הסיפור. לכל אורך חי התוכנית הוא היה 4. בהמשך נדוע בכך (שנה ב'). מה זה אומר בפועל? בהינתן קוד של `integer x=2, interger y=x,y++`, מה יתקבל כאשר נדפיס את `x`? 2. מודיע? `integer` הוא immutable יוצר אובייקט חדש ולא רפרנס.   
**clone** - שכפול, בהינתן אובייקט נרצה להכפיל אותו ולקבל אחד זהה אליו. באופן נאיבי (לכארה), יוכל לשולח עותק של האובייקט. הבעיה? שלחנו רפרנס, ומישו מבחן יכול לשנות את האובייקט, ונשבר כך עקרון האנקפסולציה. כיצד נפתר את הבעיה? ניצור העתק של האובייקט, כך אף אחד לא יוכל לשנות אליו. אם כן, זה הרבה זכרו שנטפס. כמו כן, אנחנו יודעים את הטיפוס המדויק של האובייקט - בהינתן סטודנט, יתכן שהוא סטודנט למדעי המחשב או סטודנט למדעי הדשה. לשם כך - בג'אווה בתוך class object מוגדרת מתודה בשם `clone`. המתודה היא מסוג `protected`(לא ניתן לדרוס אותה), היא מחזירה אובייקט, יוצרת העתק שטיחי (נקרא גם `shallow copy`) של כל השדות של האובייקט המקורי עבור כל מחלקה שלא תמש את **המשתק cloneable** - ככלומר חייב למשם אותו. מדובר ריק שמסמן מחלקות שתומכות ב `clone`, זה בעצם אישור מה `jvm` להתקדם. יתכן שאובייקט ידרוש העתקה عمוקה (נקרא `deep copy`): למשל, נניח ויש לנו מחלקה סטודנט. מחלקה יש שם, כתובת (`String` מזה ממש אובייקט) ותעדות זהות. ובכן, נבצע `clone` ותתבצע העתקה רדודה. זה מעטיק את השדות ללא לחשב על היחסיות - מהם היחסיות? לסטודנט יש שם, ובכן, אנחנו יודעים שבג'אווה המחרוזות זהות נשמרות באותו מקום, אבל זה בסדר כי לכארה לא נרצה לשנות את השם. וכן, כנ"ל על `id` בהנחה שייצגנו אותו באמצעות `string`. מה הבעיה שלנו? הבעיה עם האובייקט שהינו כתובת. אנחנו יודעים שבג'אווה `String` הוא immutable - ככלומר אי אפשר לשנות אותו, שכן אם יש שדות פרמייטיביים שהם `immutable` כתובת להגדירה גם כممמשת את `clonable` ולבצע העתקה נוספת על השדות. כך למעשה נמישק ליצור `clone` לכל מושא במעין לופ עד שנגיע לאובייקט שמכיל רק משתנים פרמייטיביים. ככלומר, עד שנגיע לאובייקט שהינו immutable .  
**String Integer** והן **Integer** הם **immutable**.

ובכן, פתרנו את הבעיה של השכפול. כתע נרצה לקחת רשימה של תלמידים ולשכפל אותם, זה יעלה לי המונ!!! ויקח לי המון זכרו למשעה. ננסה לחוש על פתרון אחר -  
**בhinatan shastodnt shani muunin lshcpel hia immutable**, היה קל. עם `clone` אחד וסיימנו. יוכל ליצור מחלקה שהיא תהיה `immutableList`, ככלומר מחלקה שאינה ניתנת לשינוי. גם היא תמש את `cloneable` והשדה שבפניהם יהיה רשימה private אך mutable לא יהיה ניתן לשנות אותה מבחן! לכן אם נבנה את המחלקה בצוורה הנוכחי כן תשמור אנקפסולישן. נראה כי בכל פעם שנרצה לבצע `add` לרשימה אנחנו נעתיק את כל הרשימה למשן. נזכר - מודיע ב Hinatan את המחלקה? להמנע מבחן זכרו זמן. אבל כאן מבצעים אנסוף העתקות, בכל פעולה `add` מעתיקים את כל הרשימה. לכן, נפעל למקרים. **שני הפתרונות שלנו היו גורעים לאותה בעיה.** בכל מקרה **נעדי להשתמש בשמו אחר בהתאם לאפליקציה** שפתח. אם אנחנו מעדכנים הרבה את הרשימה, **נעדי להשתמש בשיטה הראשונה.**

הערה על `ummutable`: כל המחלקות העוטפות `Integer, Double` הן `immutable`. כאשר נעשה `5 = Integer` ואז `x = 10` כאן למעשה לא נשנה את ערך האובייקט המקורי אלא `x` קיבל הפניה חדשה חדש לאובייקט `integer` חדש שמכיל

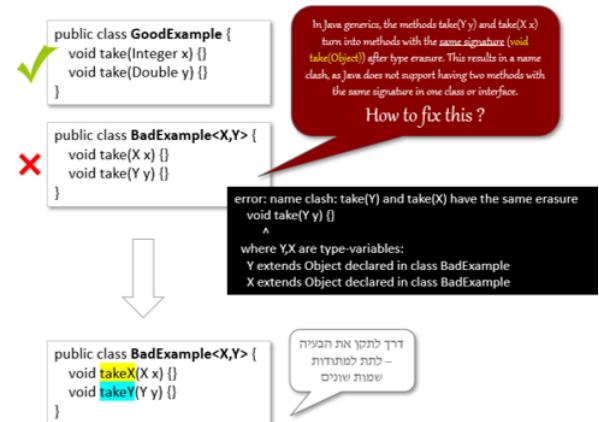
10. וכן איבדנו גישה למשתנה 5.

אם כל שדות המחלקה (כולל סטטיים) הם *immutable* אז המחלקה היא *immutable*. זה שקלאס שלי אימיטיביל זה אומר שלא ניתן לשנות כלום באובייקט אך זה לא אומר שאין גטר וסטר, יש הם פשוט לא משנים את האובייקט שנוצר - קלאס כבר מותאם לה. אם אני אימיטיביל אני מרווח אנקפסולישן אוטומטית. לקלון יש חריגה *cloneNotSupportedException* והוא צ'ק.

## OOP Advanced Features

כאן נציג מספר טריקים של פלילים בג'אווה. יש המונח המונח כאלו. לפי מרינה - מצופה מאייתנו להכיר רק מה שכាន ברשימה.

1. *Generics*: נתבונן בדוגמה מטה. על פניו, המתודת נראהיה תקינה. אותו השם, אך מדובר בטיפוס שונה. מה כבר לא חוקי? המתודת השניה, הקומפילר יגיד שיש להם אותו erasure ולכן בזמן הקומPILEציה, ג'אווה הגדירה שמתודות מטיפוס גנרי הפקו ל- *take(object)*, מה שביא לשתי מתודות עם אותו השם שמקבלות אותו טיפוס - וזה מבון שגיאה. כיצד לטפל בכך? הדרך היחידה היא לשנות את השם של המתודות. בכל הקלאסים הגנריים, המתודות בסוף מתרגמות לאובייקט.



**דוגמה נוספת** - נניח ואנחנו עובדים עם טיפוס גנרי ורוצים להפעיל על הפעיל על הטיפוס *x.compareTo(y)*, נקבל שגיאת קומPILEציה. מדוע? הקומפילר לא מוצא את המתודת *object* שוב הוא הטיפוס אליו מתרגמים טיפוסים גנריים והוא לא מממש את הממשק *comparable*. מה הפתרון? צריך להוציא את החתימה של המתודת בה אנחנו עושים את השורה הבאה! *public static <T extends Comparable<T>> max....* public static *max*(...) מגדיר בתוך הטיפוס נרצה ליצור הרחבה. **שים לב** שהזיה יכול לעשות בעיות שכן יתכן שמחלקה שאנחנו תלויים בה לא עשתה שימוש **comparable** למראות שהיא מחייבת (מתכנת גרוע) וזה נקבל שגיאיה.

**תמונה נוספת על generics** - נתבונן בדוגמה מטה. נחליט שאנחנו שולחים אל המתודת *comparator* גנרי אחד שהוא הראשונה חוקית ומתקמפלת. נראה משהו מגניב בדוגמה של הקריאה - אנחנו קראנו *max* וביצענו מה? הגדרנו בתוך קריאה למתודת, *new* לממשק?! זה לא חוקי כמובן.... יצרנו **קלאס אונוניימי** = חשוב מאד!, אין לנו דרך לדעת מה השם של האובייקט (jvm נותן לו שם מזר קלשו מאחריו הקלעים), מדובר במחלקה שהJvm מגדיר שמדובר את השם של האובייקט *comparator* ולאחר מכן גם ממש את המתודת היחידה שיש בתוך המחלקה האונונית *compareTo*.

```
public static <T> T max[T x, T y, Comparator<T> cmp] {
    if (cmp.compare(x, y) > 0) return x;
    return y;
}

public class StringComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}

max("Hi", "Hello", new StringComparator());

max("Hi", "Hello", new Comparator<String> {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
});

max ("Hi", "Hello", (s1,s2) -> s1.length() - s2.length());
max ("Hi", "Hello", (s1,s2) -> s2.compareTo(s1));
```

2. נרצה להשתמש בקיצור דרכ. אנחנו נעשו ח' כמו בדוגמה הבאה: `> - lambda`. `s1.length() - s2.length()`; זו הכתיבה הכיתובה מוקוצרת שיש בג'ואהו כאשר מוגדר בפנים קלאס אונוני. למשה אמרנו לקומפיילר, כאשר אתה מפעיל מתודה מסוימת, אתה יודע שהרי אתה צריך לקבל שנ שמות ואיזשהו `comparator`. אז, אני כבר מומש לך אותו. החל מהחץ, מתחילה המימוש של המתודה או הממשק. למעשה, את המתודה `a + b` ניתן כתוב ממש כ `c = a + b`. בלי נקודה פסיק אפל! מודיע? התוכנה במקור מגיעה מפייתון. **עיר כי ניתן לעשות שימוש במקרה רק עבור מימוש מתודה אחת בלבד במחלקה אונונית.** בנוספ, ישנה מתודה בשם `forEach` שאפשר להפעיל על `ArrayList`, כך למשל אפשר להציג פקודת כמו `→ system.out.println(i)`.

חשוב לציין, כאשר נעשו למזהה בתוכן לולאט פור, המחלוקת האונונית שתтворצ תהיה זהה לכל הלולאות. לעומת זאת, כאשר מחקים את רעיון הפור ומבצעים שכפול קוד 3 פעמים למשל במקום לזרע עד שלוש, יוצרו 3 מחלוקות אונוניתות שונות.

#### כשפותיים למזהה נוצר קלאס אונוני.

**דוגמה מבלבלת:** האם הקוד הבא תקין? הסוג של `obj` הוא תמיד `IA`, למשק יש שתי מתודות. לאחת מהן יש מימוש דיפולטיבי (ריך), וכך מותר להשתמש ב-`lambda` גם עבור מקרים כמו אלו כאשר המחלוקת מימושה בפועל רק מתודה אחת (ואכן מומשת כאן אחת בדיקוק)! שחריר מודיע זה מבלבל? קודם שמוור להשתמש במקרה רק עבור מחלוקות עם מתודה אחת לא ממומשת וכן ממשים אותה, אך מצב של `default` מותוד כו בסדר מבחינותו.

```
public interface IA {
    public int f();
    public default void g(int x, int y) {}
}

public class Main {
    public static void main(String[] args) {
        IA obj1 = () -> 1 + 2;
        IA obj2 = () -> 2 * 3;
        IA obj3 = () -> 4 / 5;
        System.out.println(obj1.f() + obj2.f() + obj3.f());
    }
}
```

- מחלוקת שהיוזר יוצר בהרצה. `main` נחשב לאחת כזו (כי לכל קלאס אובייקט המתאר אותו) וכן גם נחשבת לכך כי היא מחלוקת אונונית. נזכר שבעת `for` נוצרת מחלוקת אונונית אחת (אופטימזיה של ג'ואהו) ובSCPOL קוד יוצרים מס' מחלוקות אונוניתות. **לכל קלאס יש אובייקט שמתאר אותו ולכן כאשר יוצרים מחלוקת אונונית נוצר אובייקט שמתאר אותו.**

3. **עבור מחלוקת אבסטרקטית** - נתבונן בדוגמה מטה. הדוגמה משמשת לא תעבור ותחזיר שגיאה, מימין עם מחלוקת אונונית דזוקא כו תעבור. לא ניתן להשתמש במזהה באופן ישירה על מחלוקת אבסטרקטית ולכן משתמשים בקלאס אונוניים במקום.

```
abstract class A {
    public abstract void f();
}
class Main {
{
    public static void main(String[] args) {
        A a = () -> System.out.println("Hi");
        a.f();
    }
}
error: incompatible types: A is not a functional interface
A a = () -> System.out.println("Hi");
```

```
public abstract class A implements IA {
    public abstract void f();
}

public class Main {
    public static void main(String[] args) {
        A a = new A() { // Anonymous class instance
            public void f() {
                System.out.println("Hi");
            }
        };
        a.f(); Hi
    }
}
```

5. User defined class - מחלוקת שהמזהה כתוב. לא מחלוקת של ג'ואהו מובנית אלא כזו שאתה אחראי ליצורה. כמובן מחלוקת שאני עובד בה נחשבת לכזו וכן מחלוקת למזהה יהיו כאלה. נציג כי `String[] args` איןנו אובייקט שנוצר בתוצאה משתמש.

6. **Lazy class loading** - בג'ואהו `VM` לא טוען את כל המחלוקות בזמן שהקוד נטען (כשותוכנית מתחילה לזרע), במקומות זאת היא טוענת את המחלוקות רק כאשר הן באמת נדרשות במהלך הריצעה של התוכנית. לעומת זאת המשמעות היא שברגע שתעשה שימוש בפעם הראשונה במחלוקת (למשל תיזור מופיע או תקרה למזהה סטטי) אז תתרחש טיענית המחלוקת, הטעינה קורית רק כאשר המחלוקת נדרשת. לאחר מכן ניגשים לשדה סטטי או מתודה סטטית של מחלוקת, קוראים `Class.forName` בשביל לטוען את המחלוקת, מחלוקת אב נטען וזה גם מחלוקת בן שגולשות אליה למזהה בתוכן לולאט ג'ואהו רק מחלוקת אחת עבר המזהה ולא מחלוקת חדשות בכל איטרציה (הסיבה היא בגל `lazy`), הקומפיילר מבצע אופטימזיה כדי למנוע יצרה מיותרת של מחלוקות חדשות בכל פעם כשהמבצעים איטרציה בלולאה.

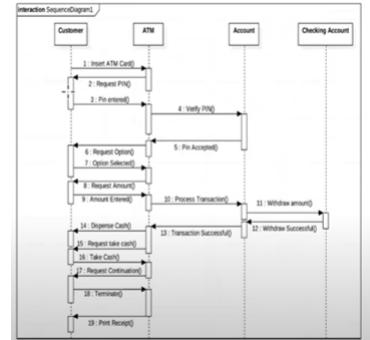
## TDD- Test Driven devolpment

עשינו דיאגרמה וכתבנו אילה קוד. הרצנו כמה פעמים, אח"כ מצאנו באג. תיקנו אותו, עשינו טסט. מסתבר עשוינו אחרת. אולי נתקל בתוך לו. הרעיון הוא פשוט - קודם נכתב את הטסט ותמיד נכשל בו.Cut, המטרה היא לפתח קוד שלא נכשל בו. כאן אנחנו מנסים להוכיח אל הטסט, ולא סתם על עיוור ואז להוכיח טסט. בכל פעם שנעבור טסט נתקדם לטסט הבא, ככל שלב אנחנו נריץ את הטסט הנוכחי ואת כל קודמי וזה מאוד חשוב שכן אם טיפלנו בבאג יתכן שיצרנו אחד אחר. גישה זו תבטיח שזה לא יקרה.

## UML

נستخدم בגרפיקה על מנת לתאר את הקוד שלנו. בתוכנות בחוץ ובתעשייה עובדים עם *uml* שעוזרים לעשות סדר בפרויקט גדול. כאשר ניגשים לכטוב אחד כזו נרצה להבין אילו אינטראקציות קיימות. למשל, בתוך מערכת של בנק - אשר אני מפעיל מתודה של MISCHIT כספר, מי מעורב בכך? חשבו הבנק, הבנק, מערכת הפריטה. כאשר ניגשים לפרויקט אנחנו עוסקים בשלבים הבאים: נחשוב מה הדרישות שייראו לנו מהמערכת (או אם אני בפרויקט קיים שמשפר נחשוב איך שיפורים ל��ח בקש ממני), נבין מי השחקנים במשחק, איך מחלקות צריך עבורם ואיזה פעולות נרצה שהמערכת תבצע. וכן נתכנן את המערכת לפי מחלקות והורשה. ישנם סוגים רבים של *uml*, **הוא מתחלך לשניים:** יש

**כלו שגדירים את המבנה ויש כלו שմסבירים התנהוגות.**  
דיאגרמת ההתנהוגות החשובה ביותר נקראת **sequence diagram**, היא מיצגה את התקשרות בין הפעולות במחלקות השונות. ניתן ממש להמחיש את התהליך שקרה מבחינת זמן, אילו פעולות יבוצעו תוך כדי. דוגמה לדיאגרמה כזו:



הDİAGRAM שמתארת את המבנה הנפוצה ביותר בינוי הינה **class diagram**, והוא מכילה תיאור של הפרויקט שככלל את המחלקות שבו והפעולות וכן הקשרים ביניהם. כיוון שיש על כך שאלת במכון שבו נתון דיאגרמה כזו ועלינו להסיק מהקשרים דבירם, נתעכט וחשוב מאד לזכור את הסימונים הבאים:

+ מצין מותודה **public** , - מצין מותודה **private** וכן # מצין מותודה **protected** .  
**קו ישר בין מחלקות** - מייצג קשר כללי ומתמך בין שני המחלקות, למשל עובד וחברה. לעיתים גם כתוב ליד את מס' המופעים של הקשרים בינם.

**קו ישר עם מעוין ריק** - מייצג קשר בו מחלוקת אחת מכילה מחלוקת אחרת וכן המועוין תמיד מצד של המכילה. הקשר איינו חזק במיוחד בין השניים במרקחה זה שכן החלקים יכולים להתקיים גם בלי השלים. למשל - סטודנטים והרצתה, יתכן הרצתה ללא סטודנטים (באספה למינינה). קשר שיקיות חלש!

**קו ישר עם מעוין אחד מלא (שחור)** - מייצג קשר חזק בו מחלוקת אחת מורכבת מאובייקטים של המחלוקת השנייה (השנייה או עם המועוין), ואין לה קיום בלבדה. למשל -מנה ורכיבים של המנה. קשר שיקיות חזק!

**קו מקווקו עם חזק בקצת** - קשר של שימוש ממשק, החזק יוצא מחלוקת ומצביע לכיוון המשק.  
**קו ישר עם חזק בקצת** - קשר של ירושה בין מחלקות, החזק הוא לכיוון המחלוקת שמנה אנחנו יורשים. ככל חזק חיה.

## Iterator

בהרחבה על **iterator** כי זה מופיע בכל בוחן שאלות 20 – 18 :  
. נרצה לעبور על אוסף כלשהו מבלי שאנו יודעים את מבנה הנתונים שהאוסף מואחסן בו. נרצה לבצע על כל אחד מהאיברים פעולה. נוכל לעשות בקהלות על **ArrayList**, אך אם נעשה זאת על רשימה זה לא יעיל. עליה ( $O(n)$ ) רק בשביל לחפש את האיבר. זה קורה עבור  $n$  איברים ולכל מדברים על ( $O(n^2)$ ) זמן לבצע פעולה על המבנה. וכי אמר גם שהאיברים אחד אחרי השני שניתן ככה לעבור? למשל טבלת האש. لكن נרצה אובייקט בשם **iterator** שידע לעבור בצורה יעילה על מבנה הנתונים, מבלי להזכיר מראש שכך רוצים אנקספוליאישן. לכל מבנה נתונים נבנה איטרטור שיראה תמיד אותו דבר ונוכל להשתמש בו בכל מבנה , יש ממשק שנכתב מראש ונitinן להשתמש בו:  
public interface Iterator<E>{

```
boolean hasNext();  
E next();
```

המטרה היא גישה איחודית וככללית לאיטרציה על אלמנטים.  
א. (*hasNext*) מトודה שתחזיר *true* כל עוד יש איברים שאפשר לעבור עליהם. אפשרותה לולאה לדעת מתי להפסיק ולרוב משווה אינדקס למס' האיברים  
ב. (*next*) מחזירה את האיבר הבא ומקדמת את המצב הפנימי (למשל, אינדקס). אם אין עוד איברים תזרוק *NoSuchElementException*. לאחר מכן אנחנו צריכים להכין את הדבר הבא עם *prapareNext()* ואז צריך להחזיר עצם נכון (זוג, צלע בגרף,...) שהוא הרוי זה שידפס ב- *while* של האיטרטור.  
ג. (*parpareNext*) לא חייבת להיות לרוב, אבל היא מותודה פרטית שתפקידה להכין מראש את האיבר הבא שליי *next* יתיחס ולשמור אותו במשתנה פנימי. היא לא חלק מממשק רשמי אך נהוג להשתמש בה בשביל לשימור את המצביע הבא מראש, להפריד בין לוגיקת האיתור של האיבר הבא ובין פונקציות (*next()* ו(*hasnext()*)).

חשוב לציין - לרוב נרצה לשמר הפרדה בין החזרת הערכים (למשל בהדפסתם של עץ לפי רמות) לבין החזרת *nodes* של השכבה הבאה.

**חשוב לזכור כאשר עושים *for – each* נוצר אובייקט של *!!!iterator***