

סיכום מתומצת למבחן - תכנות מונחה עצמים

21 ביולי 2025

הסיכום כולל מושגים ו"טריקים" שחוזרים על עצמם במבחנים.
גיא יער-און.
אם מצאתם טעות אשמח לעדכון:

מבוא לג'אווה וקצת בסיס על תכנות מונחה עצמים

* אם יש לי קובץ הרצה על המחשב שלי ואנסה להעבירו למחשב אחר, הוא לא בהכרח ירוץ. ייתכן מאוד שלא ירוץ. זה תלוי בכך שהאם יש לי את אותה הארכיטקטורה במחשב ואת אותה משפחת מערכות הפעלה.
* בג'אווה, המופע של המחלקות יופיע בתוך *heap*. בג'אווה אין מצביעים, יש *references*. כאשר עושים *new* לאובייקט זה יוצר רפרנס, מאחורי הקלעים קיימת עבודה של *malloc*.
* *this*: כאשר אנחנו מעוניינים לפנות לאחת השדות מתוך המחלקה. זו לא חובה להשתמש ב*this*. אך מאפשר *this.x = x* שזה כמובן נוח.

* *getters&setters*: כיוון שאסור לגשת לשדות מחוץ ל*class*, אנחנו נשתמש בגטרים וסטרים על מנת לקבל ערך של אובייקט או להציב בו. כמובן שיש לממש אותם.

* בג'אווה הכל אובייקטים. גם מערכים ומחרוזות. שדות ומתודות יחד יקראו *class members*.

* בתוך מחלקות שונות, ניתן לקרוא למתודות בשם זהה.

* מתודה פרטית: נניח ונרצה לבצע טסט על מתודה שהיא פרטית, הבעיה היא שלא ניתן לגשת אליה. יש שיטה לפיה אנחנו יודעים כי אם המתודה פרטית, אזי רק בתוך המחלקה ניתנת לקרוא לה לפעולות שונות. היא לא נגלית למשתמש. בוודאות משתמשים בה במתודות אחרות, אחרת אין לה נחיצות, לכן אנחנו נבדוק אותן, נבדוק ונבצע מספיק טסטים על המתודות שכן ניתן, הציבוריות, וכך נבדוק גם אותה על הדרך.

* לשים לב מהיכן ניגשים לשדה פרטי - מתוך המחלקה מותר!!

* **(***)** אם אני עם משתנה *Integer* כשדה שלי, ואני ב*getter* מחזיר *this.name* שלו, אזי אני שובר אנקפסוליישן כי הוא לא פרמיטיבי וזה ממש יחזיר את המשתנה עצמו!
* מבחינת קומפילר חוקי לחלוטין סינטקס כמו *(new B).f()* וכן למשתנה מסוג זה יהיה טיפוס סטטי ודינמי שיהיו אותו דבר - *B* במקרה הזה כי יצרת מופע של *B*.

Constructor

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

העמסה (*Overloading*) של בנאים: נוכל ליצור כמה בנאים שנרצה, שהשינוי הוא במס' הארגומנטים שמקבלים. ניתן לקרוא לבנאי אחד מתוך בנאי קיים, באמצעות *this* וזה חייב להיות מהשורה הראשונה (כמו *super*).

מה קורה מאחורי הקלעים בהפעלת קונסטרוקטור? ראשית מקצים זכרון לכל השדות של האובייקט (כולל *Vtable* וכן מחלקת *object*), אח"כ נקצה ערך דיפולטי לכל אחד מהשדות של המופע. אם בתוך הקונסטרוקטור היה קריאה לקונסטרוקטור אחר - קודם נבצע אותו, גם אם כתוב וגם אם לא נצטרך ללכת לטפל ב*super*, אם כבר יש *super*

נעשה אותו ואם לא אנחנו צריכים ללכת ל-*super* של מחלקת האבא *object* (תמיד תתבצע קריאה ל-*object* בהפעלת קונסטרקטור) וגם אם קראנו לבנאי אחר, הוא כבר יקרא לאובג'קט. ובסוף, נמשיך את גוף הקונסטרקטור הנוכחי. בנאי לא יכול להיות רקורסיבי. ערך התחלתי שמאותחל בבנאי (אם לא אתחלנו למשהו אחר) הוא אפס עבור המשתנים המוכרים, *false* עבור משתנה בוליאני ועבור *refrence* יהיה *null*.

אם יש לבנאי של אבא קונסטרקטור ריק ללא פרמטרים גם אם לא נכתוב בבנאי של בן *super* הוא יקרא אוטומטית, רק אם יש בנאי ריק - אם אין חייבת קריאה ישירה.

אם כתבנו משהו כמו *A.func* בקוד אפשר לדעת בוודאות שמתודה *func* היא סטטית כי הופעלה על שם מחלקה אבל ייתכן שהיא שייכת לאבא של *A* ולא בהכרח *A* עצמו. כנ"ל על *A.func(A.x)* יתכן כי *x* שדה סטטי של אב קדום של *A*. בכל מקרה ברור שהם שייכים ל-*A*!!!! כי *A* יורש - אבל ייתכן שמוגדרים אצל אבא של *A*.

הערה - קונסטרקטור לא חייב להיות *public* מבחינת קומפילר

Garbage Collector

מנקה האשפה של ג'אווה. תפקידו "לנקות אחרינו את העבודה". למשל - נניח ועשינו פעולה `point p=new point(2,3)`, לאחר מכן נרצה לעשות `p= new Point(4,5)` בזכותו זה יהיה אפשרי ואפילו חוקי, הוא ינקה את הערכים הקודמים. יש לשים לב שזה כמובן יעלה לנו ביעילות. ניתן להפעיל אותו באופן ידני מתוך התוכנית עם `system.gc()` אך זה מאוד מאוד לא מומלץ!

Encapsulation

בעברית נקרא גם כימוס. העקרון הוא פשוט: למנוע מהמשתמש לנסות לשנות לי דברים בקוד. לקוח לא צריך לדעת בין היתר כיצד מימשנו את הדברים ובאיזה שדות השתמשנו. מאפשר למנוע מהמשתמש לשנות את המידע כיצד שהוא רוצה, אני אחליט מה הוא יוכל לעשות. למשל, אם ניצור קונסטרקטור עם נקודה *point* שיקבל כערך, ונאתחל בקונסטרקטור `this.point=point`, זו תהיה שגיאה באנקפסוליישן שכן כעת ניתן לגשת לאותה נקודה מבחוץ. מה יהיה הפתרון? `this.point=new point(point.getX(),point.getY())` כעת, כיוון שיצרנו את הנקודה בתוך הקונסטרקטור לא יוכלו לגשת אליה מבחוץ. המטרה היא להגן על השדות הפרטיים שלנו ממתקפה, *private* לא תמיד יעזור לכך.

Static Members

סוג מיוחד של *class members*. מדובר בתכונה ששייכת למחלקה ואיננה ברמת האובייקט. נשמר ב-*data*. למשל, אם נרצה לספור כמה פעמים יצרנו אובייקט מסוג זה לאורך התוכנית נוכל להשתמש בשדה סטטי שישכום בכל פעם שאנחנו יוצרים מופע חדש. גם אם לא יצרנו אף אובייקט עוד, הערך הסטטי קיים כיוון שהינו ברמת המחלקה. ניתן ליצור מתודות סטטיות - מתודות ששייכות למחלקה. כאשר נרצה לגשת אליהם נעשה זאת עם שם המחלקה. עם זאת, ניתן לגשת למשתנים סטטיים / מתודות סטטיות גם מתוך מופע של אובייקט. זה יתקמפל - אך זה לא יפה ולא נהוג.

נשים לב שהמחלקה השימושית *math* היא מחלקה שכל המתודות בתוכה הם סטטיות. מדוע? נרצה שכל המתודות יהיו ברמת המחלקה, כיוון שבג'אווה כל מתודה צריכה להיות מוגדרת בתוך מחלקה מסויימת, הקודקודים של ג'אווה ראו לנכון לרכז את כולם בספרייה אחת. מתוך מתודה סטטית, לא ניתן לגשת לתוך מתודה שאינה סטטית. מדוע? מתודה סטטית היא ברמת המחלקה ואילו מתודה שאינה סטטית היא ברמת האובייקט. אם נשתמש הרבה בסטטיק, כנראה שבחרנו שפת תכנות שאיננה נכונה לפרויקט שלנו. יש להצדיק שימוש במשתנים ומתודות סטטיות.

נשים לב כי גם אם למחלקה יש רק שדה סטטי יחיד - היא איננה סטטית וניתן להגדיר בה מתודות שאינם סטטיות.

אם יש לי מחלקה *A* שיורשת ממחלקה *B*, יש בשתייהן מתודה *f* סטטית - לא מתבצעת דריסה אלא הסתרה (*hidden*) - כאשר נעשה אח"כ במיין שלי `B - b = new A();` ואז נפעיל `b.f` זה יהיה הפעלת *f* של טיפוס סטטי של משתנה.

מותר לעדכן משתנה סטטי בתוך בנאי - אפשר לגשת אליו מכל מקום במחלקה ובפרט בבנאי. הפעם היחידה שאי אפשר לשנות משתנה סטטי זה אם נצהיר עליו גם כ-*final* אז הוא גרסה אחרונה ולא ניתן לשנותו.

Polymorphism

היכולת לשייך טיפוסים שונים לאותו אובייקט.
בעברית - רב צורתיות, משה יכול להיות סטודנט אך במקביל יכול להיות גם חייל מילואים או אח או לחלופין אבא של הילדה נועה. בעזרת אילו מתודות נתממש עם משה? הוא הרי יכול להיות כמה וכמה דברים.
ממשק מגיע לפתור את הבעיה הזו. הממשק הינו הצהרת כוונות של המתכנת שכל מי שמממש אותו (implements nameOfInterface) יהיה חייב לממש את הפונקציות שנמצאות בתוכו. למשל, אם נגדיר ממשק polygon ונרצה שהמחלקה ריבוע תממש אותו, שהרי היא פוליגון, ונניח שבממשק יש מתודה יחידה שאיננה ממומשת : שטח, אזי מחלקת ריבוע תהיה חייבת לממש את המתודה של השטח. אחרת נקבל שגיאה.
ייתכן ויהיה מימוש בתוך ממשק, אך זו לא המטרה אלא להגדיר התנהגות.
נניח ויש לי ממשק כלשהו, קוראים לו *AI*, ונרצה לבצע את הפעולה הבאה *IA a=new IA()*, תמיד זה יוביל לשגיאת קומפילציה. הגדרנו כאן *new*, השאלה היא *new* על מה? אין קונסטרקטור בתוך ממשק ולא יתכן שיהיה קונסטרקטור בתוך ממשק. עם זאת, אם יש לי *A* שמממש את *AI*, כן ניתן לבצע פעולות כמו *IA a=new A()*, ושוב בהנחה שאכן *A* מוגדר טוב עם קונסטרקטור.

ג'אוה מאפשרת ממשקים מרובים, כלומר ניתן לממש כמה ממשקים.
מתודה דיפולט: ג'אוה מאפשרת להגדיר מימוש דיפולטיבי בתוך הממשק. נשתמש במילה השמורה *default* וכך נוכל למעשה לקבוע מימוש דיפולטיבי לתוכנית. זה יכול להיות זמני, כך שבעתיד נוכל לדרוס את המתודה ולממש אותה במחלקות השונות שמממשות את הממשק.
שדות בתוך הממשק: גם אם ציינו וגם אם לא ציינו, אם נגדיר בממשק *int x = 5* תמיד תמיד זה יהיה פבליק, סטטיק ופינל. כלומר ממש *constant*.

בממשק ייתכן מתודות פרייבט, כמו כן מתודות אבסטרקטיות, סטטיות ודיפולט. מה אסור? פינל, פרוטקטד ואבסטרקט פרייבט.

בעיית היהלום: נניח ויש לנו שני ממשקים *A, B* שמצהירים על מתודה *f* שהיא *default* ואנחנו מעוניינים ליצור ממשק *C* שרוצה לרשת משניהם. אזי - זה לא אפשרי. שכן נהיה בקונפליקט מאיזה ממשק לקחת את המתודה. עם זאת, אם בממשקים אין מתודות ממומשות עם אותה החתימה, הורשה מרובה של ממשקים כן עובדת.
קאסטינג: נניח שיש אינטרפייס חיה ויש מחלקה כלב שמממש אותו. באינטרפייס יש מתודה אחת, *a* נקרא לה. ניתן לבצע קאסטינג הבא *Dog d=new Dog()* ואז את הקאסטינג *Animal b=(Animal) d*, מה ביצענו כאן? אמרנו למערכת - תתייחס אליו כעת כחיה. כעת, הוא יוכל להתייחס אליו רק באמצעות המתודות של האינטרפייס, כלומר *a*. נניח ויש מתודה *bbb* בתוך *Dog*, אזי כעת אין לו גישה אליה. **למה בכלל שנעשה את זה?** פולימורפיזם. אכפת לי מחיה כלשהי, לא משנה אם היא כלב חתול או תוכי.

בתוך ממשק לא יכולות להיות ממומשות מתודות. רק מתודות *default* יכולות להיות בפנים וכמובן כאלו ללא מימוש.

לאינטרפייס יש *vtable* גם: מתודה דיפולט תופיע ב*vtable* גם אם לא מימשנו בכלל אינטרפייס זה.

לאינטרפייס יש אפשרות לשים מתודה דיפולטיבית. באופן אוטומטי מי שיממש אותו יקבל אותה. ניתן לדרוס את המימוש הדיפולטיבי הזה!!!. כמו כן - כמובן שמי שיממש ממשק זה יחזיק במתודה הדיפולטיבית שתהיה עבור כל אחד באותה הכתובת. - **זה חוקי אך לא לפי עקרונות OOP.**

שדה יחיד שיכול להיות בתוך ממשק הינו *public static final* אך גם אם נרשום *int x=5* זה *implicitly* נחשב כאילו כתבנו את כולם וחוקי.

בתוך ממשק ניתן לכתוב מתודה דיפולטיבית כמו שהוער לעיל, וכן אפשר לשים מתודה סטטית. כמו כן ניתן גם להשתמש במתודה פרייבט בתוך ממשק - אפשר לעשות זאת רק בתוך הממשק כלומר אי אפשר לגשת אליה מבחוץ לממשק גם אם מחלקה שלי מממשת ממשק (זה בעיקר יהיה עבור חישובים מסובכים בתוך מתודות דיפולט שהרי רק מהם אפשר לקרוא להן). **מתודה פרוטקטד, פינל ומשתנים לא קבועים אינם יכולים להיות בממשק וכמובן שלא קונסטרקטורים. אבסטרקד כן שזה הרי כל הרעיון (הן מראש אבסטרקד גם אם לא כותבים).**

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 מחלקה שבה נשים את כל התכונות המשותפות והמתודות המשותפות. שאר המתודות יירשו ממנה.

התכונה *override* מאפשרת לנו לדרוס מתודות. למשל, נניח ואנחנו במחלקה *A* שמרחיבה את *B*, במחלקה *B* ישנה מתודה *f*. אנחנו ירשנו מ*B*, אבל אנחנו מעוניינים במימוש אחר של *f*. אין בעיה - נעשה *@override* ונממש אותה מחדש, כפי שרצינו. נראה כי אם דרסנו את כל המחלקה שירשנו, בשביל מה ירשנו בכלל?

מתי מותר לעשות אובריייד? אם החתימה זהה, שם המתודה סדר וטיפוס המשתנים זהה, ערך ההחזרה במתודה הדורסת מאותו טיפוס או ספציפי יותר וכן נראות המתודה הדורסת גדולה יותר. כמו כן, לא ניתן לבצע אובריייד למתודה פרטית, סטטית, *final* או שייכת למחלקה *final* ואין לה גרסה קודמת. אחרת - נקבל שגיאת קומפילציה. התכונה *upcasting*: המרה של אובייקט ממחלקת בן למחלקת אב. תמיד ניתן לעשות אפקאסטינג. באופן סימטרי כמעט, התכונה *downcasting* היא המרה של אובייקט ממחלקת אב למחלקת בן. חייבים לכתוב דאוןקאסט בצורה מפורשת וכן הקומפילטר לא תמיד יסכים לעשות. נעיר כעת 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 ממחלקת אב *Animal*, הדוגמה הראשונה היא לקחת איזשהי חיה, *animal* ולהגיד לה "את כלב מעכשיו", כלומר שנמכנו אותה להיות כלב. זה עובד כמובן, שכן החיה הוגדרה להיות כלב, ואז עם קאסטינג אמרנו לה את כלב ואכן יתקמפל. לעומת זאת, בדוגמה השנייה אנחנו הצהרנו על חיה, ואז ניסינו לכפות עליה להיות כלב. מי אמר שהיא כלב בכלל? היא הוגדרה כחיה. ומה אם היא חתול? הקומפילטר עומד לאכול את השקר שהאכלנו אותו בזמן הקומפילציה - אך בזמן ריצה יכשל. ג'אווה לא מאפשרת הורשה משולשת. בכלל.

הפעולה *super*: מילה שמורה שמאפשרת לגשת לתכונות מחלקת האב מתוך מחלקת הבן. ניתן מתוך הקונסטרקטור להפעיל *super* על הקונסטרקטור של מחלקת האב, **ניתן לעשות זאת רק מהשורה הראשונה.** עם זאת, ניתן גם להפעיל *super* על מתודות אחרות במחלקת האב, למשל אני מגיע לדרוס את המתודה *f*, אך כן מעוניין להשתמש במימוש הקודם שלה ולהוסיף משהו, **אזי ניתן במקרה זה (שאינו בקונסטרקטור) להפעיל *super* מכל שורה שהיא.**

אפשר להפעיל *super* רעיונית גם בתוך מחלקה רגילה שלא יורשת מאף אחד - בקונסטרקטור: תתבצע קריאה לקונסטרקטור של אובג'קט ישירות, ובמתודות: אם נרצה להפעיל מתודות של *object*. לא תמיד חייב להיות סופר, יכול להיות סופר ריק כדיפולטיבי. תמיד נהיה חייבים לבצע סופר במחלקת בן למחלקת אב. אם אין קונסטרקטור ריק - אזי אנחנו חייבים לקרוא מפורשות לקונסטרקטור. שכן יש קונסטרקטור ריק רק אם לא דרסנו אותו עם קונסטרקטור אחר.

הפעם היחידה שבא לא יהיה סופר זה בקונסטרקטור של *Object*.
המילה השמורה *protected*: תכונה שנוכל להוסיף לשדות במקום *private*. באמצעות התכונה ניתן להשתמש במחלקות הבן ישירות בתכונות של מחלקת האב שהוגדרו *protected*. אך, אם ננסה להשתמש בתכונות במחלקות שאינן יורשות - זו שגיאה כמובן.

יום יבוא ויבטלו את *override* תאורטית - מה שיקרה הוא שנוכל עדיין לעשות מתודה עם אותה חתימה במחלקת אב ובמחלקת בן, פשוט כל רעיון הדינמיק ביידינג יעלם - לא יהיה צורך ב*table* 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 *Am* עדיין - אפשר ליצור *new* גם אם אין ריקות כי הבנאי הדיפולטיבי כאן יפנה *super* אל *Al* הריק שיפנה ל*super* של *object*.

נראות - נשים לב שבמחלקה יורשת אם עושים דריסה של מתודה, חייבת להיות נראות גדולה יותר. כלומר - ייתכן רק בסדר הזה: *public* → *protected* → *default* → *private*. לא ייתכן פרוטקטד למשל במחלקה ופרייבט במחלקה יורשת (באותה מתודה!).

הערה חשובה: כידוע תמיד צריך כאשר מחלקה *B* יורשת *Am* בבנאי של *B* לקרוא *super* אך אם בנאי של *A* לא קיים מפורשות (כלומר *jvm* בנה לו מאחורי הקלעים דיפולטיבי) - אין חובה לעשות מפורשות *super* במחלקה *B*! זה המקרה היחיד שלא חייבים. (ובאובג'קט כמובן)

כאשר קוראים ל*this* מתוך בנאי, כלומר קריאה מבנאי אחד לבנאי אחר, זה חייב להיות בשורה ראשונה ויכולה להיות רק קריאה אחת כזו! לא יתכן פעמיים קריאה *this*.

Abstract Class

מחלקה אבסטרקטית. ישנם מחלקות שלא נרצה שנוכל ליצור מהם טיפוסים חדשים. בשביל למנוע מהמשתמש ליצור אובייקטים מאותה המחלקה נוסיף לשם המחלקה את המילה *abstract*. כעת, אם משתמש ינסה ליצור מופע של המחלקה הוא יתקל בשגיאת קומפילציה. אנחנו נותנים מקום למחלקה בפרויקט, אך לא ליצור ממנה. במחלקה

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

עבור כל מתודה במחלקה, או שנממש אותה או שנשים לה *abstract* ונטיל את המימוש שלה על מישהו אחר. קלאס שמרחיב קלאס אבסטרקטי, לא חייב להיות אבסטרקטי אך הוא צריך לממש את הפעולות האבסטרקטיות של המחלקה. אם עשינו מחלקה וכל המתודות בה אבסטרקטיות, אזי שהיינו צריכים ליצור ממשק.

ייתכן כי יהיה קונסטרקטור לקלאס אבסטרקטי, על אף שלא ניתן ליצור ממנו שדות. יכול להיות שימושי בתוך הורשה, נגדיר קונסטרקטור שיאתחל את השדות (אמנם אי אפשר ליצור מופע שלו), אך מי שירש ממנו יכול לעשות בקונסטרקטור שלו סופר על הקונסטרקטור הקדום, ואז ממנו ליצור מופע. אם לא יצרנו קונסטרקטור באופן ישיר גם כאן ג'אווה תכניס אחד דיפולטיבי.

כמובן שיתכנו שדות למחלקה אבסטרקטית. כמו כן ייתכנו מתודות פיינל בתוך מחלקה אבסטרקטית. לא ייתכן מחלקה שהיא פיינל אבסטרקט! כמו כן לא ייתכנו מתודות שהן גם פיינל וגם אבסטרקט. אבסטרקט קלאס יכול לממש ממשק.

ניתן שיהיה למחלקה אבסטרקטית מיין ולרוץ כתוכנית ג'אווה רגילה - מדוע? *main* הוא סטטיק, אז לא צריך ליצור אובייקט מהמחלקה.

מחלקה מסוג 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 ייתכן שכולן ממומשות.

לא ניתן להגדיר מתודה שהיא סטטית ואבסטרקטית בו זמנית. כמו כן, אם כתבנו *abstract* לא ניתן לממש בכל זאת את המתודה.

תתכן מתודה פיינל בתוך מחלקה אבסטרקטית (אם כמובן לא תהיה גם אבסטרקטית).

Class Object

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

מתודה *getClass*: מתודה מיוחדת של המחלקה. מחזירה 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 מתודות של המחלקה. עליהן נרחיב מטה.

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

Equals

פעולה שקיימת בכל מחלקה (כי כל מחלקה יורשת מאובג'קט). עלינו לדרוס אותה ולממש אותה בעצמנו שכן הערך הדיפולטי שלה הממומש לא שווה כלום. הפעולה הדיפולטית בודקת האם הרפרנסים זהים ולא אם התוכן זהה.

הערה חשובה: מחלקת *String* כן שכתבה את הפעולה *equals* ואם נבדוק *t.equals(s)* על שתי מחרוזות זהות בתוכן, כן נקבל *true*. מה שנכתב מעלה הוא על מחלקות אחרות שצריכות לשכתב פעולה זו מחדש. באופן דומה גם *Double, Integer, Long, Short* וכו'...

תכונה מקורס "תכנות מתקדם" שצביקה הזכיר: *StringPool* - נניח ויש לנו מצב בו *s = "hi"* וכן *t = "hi"*, אם נעשה *s == t*, נקבל *true*. מדוע? ג'אווה החליטה לשמור בזכרון באותו מקום מחרוזות זהות, לכן, למרות שמתבצעת השוואה של רפרנסים, נקבל בכל זאת *true*. לעומת זאת, יש לשים לב שבהינתן שני מערכים אם נעשה *arr1 = arr2* זה יעתיק את הרפרנסים ולא את הערכים, ולכן שינוי במערך אחד ישפיע על המערך השני, לכן יש להעתיק את כל הערכים אחד אחד ב $O(n)$.

כמו כן, ישנה דרך לעקוף סטרינגפול. נשים לב שאכן ג'אווה רוצה לייעל ושומרת אובייקטים עם אותו שם באותו מקום אך אם כתבתי *s = newString("hi")* ואז *t = newString("hi")* כאן נוצרו שני אובייקטים שונים! עקפתי את זה והם לא נשמרו באותו מקום ולכן *s == t* 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" מזהה לכל אובייקט וכאשר נפעיל את הפעולה נדפיס את ההאש קוד ולא את התיאור שחשבנו שלכאורה יודפס. נרצה לדרוס מתודה זו.
Integer, Double, String מימשו *toString* כבר בצורה תקינה.

Wrapper

בג'אווה קיים *wrapper – class* עבור כל טיפוס פרמיטיבי, קאסטינג בין *primitive* ל-*wrapper* שלו בשני הכיוונים נעשה באופן אוטומטי ע"י קומפיילר. מעבר ל-*Integer* נקרא *autoBoxing* ומעבר הפוך נקרא *unBoxing*. למשל דוגמה: יש לי משתנה *inta = 1* ויש לי מתודה *print(Object – o)*. אם נפעיל *print(a)* קוד יתקמפל. אובג'קט הוא אב קדמון לכל קלאס ולכן אם נעשה אוטובוקסינג קומפיילר מבצע אימפליסיט קאסטינג מאינטגר לאובג'קט ולכן קוד עובד ומבצע כנדרש.

לעומת זאת, קוד דומה על מערכים למשל $int - A = [1, 2, 3]$ לא יעבוד, צריך להכיר את זה כי לפעמים שפות תכנות בכלל וג'אווה בפרט מתנהגות בצורה לא אינטואיטיבית. במקרה הזה קומפיילר לא מסכים לבצע אוטובוקסינג ולא אימפליסיט קסטינג מחשש לבעיות שעוללות להיווצר. אין דרך לנחש מקרה זה - אלא להכיר. **לעומת זאת אם היינו מגדירים מערך $Integer - A$ ואז היינו משתמשים במתודה פרינט מסוג $print(Object[] A)$ אז זה כן היה חוקי!**
את אותה הפעולה קודם ניתן לעשות גם מטיפוס גנרי עם מתודה `public static <T> void print(T[] A)` שלמעשה שוב אם נשלח $Integer A[] = [1, 2, 3]$ כן יעבוד עליה. קומפיילר ימיר זאת לאחר קומפילציה ל-*Object*.

instanceOf

הפעולה מחזירה האם אובייקט הוא מסוג אב קדום של אובייקט אחר. למשל `system.out.println(s instanceof String)` יניב *true*. עם זאת, נשים לב שלא נוכל לדעת האם אכן קיבלנו אובייקט מסויים. שכן, אם יהיה לנו *smartPoint*, *Point* ו-*smartPoint* הוא *instanceOf* של *Point* שכן *Point* הוא אב קדום שלו. ולכן בשביל לשכתב פעולת *equals* למשל אנחנו נשתמש בשורה הבאה:

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

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

חשוב - משווה לפי הערך הדינמי של משתנה ולא סטטי.

Final class

מחלקה שהינה פיינל קלאס היא מחלקה שלא ניתן לרשת ממנה. מתודה *final* היא מתודה שלא ניתן לדרוס אותה, ואם ננסה נקבל שגיאת קומפילציה. אם יש לי שדה פרטי שהוא פיינל - מתוך הקונסטרקטור נוכל לשנותו אמ"מ הוא עוד לא אותחל - אפשר רק פעם אחת.
צריך לשים לב ש-*final* לא גורר *immutable* ובפרט אם כל המתודות של המחלקה וכן המשתנים הם *final* זה לא יגרור *immutable*.
מותר לעשות *overload* למתודה פיינל (והיא כמובן לא חייבת להיות פיינל המתודה שעושים לה *overload*)

Static&Dinamic biding

מה זה אומר? נתבונן בדוגמה - `Polygon p=new name`, הטיפוס הדינמי שלנו הינו *name*, זה שמגיע מימין. הטיפוס הסטטי שלנו הינו *polygon*.

בזמן הריצה, הקומפיילר בונה טבלה בשם *vTable* לכל מחלקה בזכרון. הטבלה מחזיקה חלק מהכתובות של המתודות במחלקה. אם מישהו ירש, הכתובת למתודה שהוא ירש תהיה אותה הכתובת למתודה של האבא. הקומפיילר דוחף שדה *vptr* לכל אובייקט שיתאר את הכתובת שלו בזכרון. בזמן הקומפילציה, הקומפיילר מתייחס לסטטיק ביידינג בלבד. זה מה שהוא רואה בזמן הקומפילציה. בזמן הריצה, הוא יכול לגלות דברים אחרים בדמות הטיפוס הדינמי. השאלה שנשאלת - מי מופעל ומתי? האם תמיד מופעל דינמיק ביידינג?

סטטיק ביידינג: מתרחש בזמן הקומפילציה, יותר משתלם בזמן ריצה, מופעל עבור מתודות סטטיות, מתודות *private*, קונסטרקטור וסטטיק פיינל. כמו כן, אם הצהרנו על משתנה בתחילת המחלקה, `int x` כלשהו, אזי גם עליו

יופעל סטטיק בידינג. כלומר אם $x = 5$ הוגדר במחלקה A $x = 6$ הוגדר במחלקה B ונבצע $A a = \text{new } B$, ואז נעשה $a.x$, נקבל את הטיפוס הסטטי, כלומר נלך ל- x של A ונקבל 5.

דינמיק בידינג: דינמי - מופעל בזמן ריצה, על מתודות רגילות, מתודות שהן אובריידיד, אבסטרקטיות וכן פיינל. הקומפילר לא יודע מראש איזו מתודה תופעל. במהלך הריצה הקוד יבחר איזו גרסה של הקוד להפעיל, בהתאם לסוג האובייקט שבזכרון. אם אפשר להמנע ממנו - יחסוך בזמן ריצה. מתודות סטטיות, פרייבט ופיינל לא יופיעו ב- $vTable$.

נעיר כי מתודה מסוג אבסטרקטית כזו תכנס לזכרון, ותצביע על null. כלומר - אם יש לי מחלקה אבסטרקטית ובה מתודה אבסטרקטית, ומחלקה שיורשת שמממשת אותה. כמובן ב- $vTable$ של היורשת תופיע המתודה, ובמחלקת האב יופיע עבור המתודה ב- $vTable$ פשוט $null$. **באופן כללי הרישום null בתוך vtable עבור מתודה מעיד שלמתודה אין מימוש בקלאס הספציפי הזה.**

כאשר אני כותב משהו כמו $(A)b.f$ כאשר כמובן B יורש מ- A אני אומר לקומפילר תעשה $\text{upcast} =$ חוקי לגמרי, תתייחס ל- b ברמה הדינמית כמו קודם וברמה הסטטית כא.

נכנס ל- $vTable$: מתודות רגילות, abstract מתוד'ס עם $protected, null$. לא נכנס? $final, static, private$ וכן $constructors$. מתי מתודה מסוג $final$ כן נכנסת? אם קלאס שמכיל מתודה מסוג $final$ מרחיב קלאס אחר אז המתודה $final$ כן תכנס ל- $vtable$. כמו כן אם הטיפוס הסטטי של הרפרנס הוא לא המחלקה שמכילה את המתודה ה- $final$ יתבצע קריאה למתודה לפי דינמיק בידינג.

$Vtable$ נוצר רק אחרי קומפילציה ולכן בהכרח הקוד התקמפל אם אני מקבל טבלת $Vtable$.

לסיכום:

אבסטרקטיות - דינמיק בידינג
פרייבט - סטטיק בידינג
סטטיות - סטטיק בידינג
פיינל - דינמיק בידינג
סטטיק פיינל - סטטיק בידינג
רגילות ($void$) - דינמיק בידינג
קונסטרקטור - סטטיק בידינג
העמסה - סטטיק בידינג בזמן קומפילציה

Overloading

נרצה לפעמים מתודות בשם זהה. מדוע? נניח ויש לי רצון לעשות מתודה ששואלת שאלה מסויימת, אבל ייתכן שמישהו ירצה להכניס יחד עם השאלה פרמטר בוליאני, מישהו אחר לא יכניס כלל ואחר יכניס משתנה ממשי. מה נעשה? אפשר על הנייר לעשות שלוש מתודות בשמות שונים ולהשתמש במידת הצורך בכל אחת אבל זה מסורבל. ניתן לקרוא למתודות הללו באותו שם, מצב זה נקרא העמסה.

מתי מותר לבצע העמסה? אם יש שוני בכל פעם בארגומנטים ששלחנו, אסור מצב של: אותו שם, פרמטרים זהים, שינוי רק בערך החזרה של המתודה - זה מצב שנקרא $ambigius$ (דו משמעי)

גם שוני בשמות הארגומנטים לא רלוונטי, אלא רק הטיפוס שלהם והכמות שלהם. למשל, נניח ויש לנו מחלקה B שמרחיבה את מחלקה A . נניח ועשינו $f(A\ x, B\ y), f(B\ y, A\ x)$ נקבל שגיאת קומפילציה שכן אותו ערך החזרה - אותו שם - ומס' המשתנים ששלחו לי זהה. לא יתכן.

הערה חשובה - כיצד נקבע לאיזו גרסה של מתודה תתבצע? קומפילר בוחר מה החתימה של המתודה שיש להריץ לפי סטטיק בידינג של ארגומנטים שהעברנו למתודה, ובזמן ריצה מופעל דינמיק בידינג שבוחר בין המתודות בעלת החתימה שנבחרה את המימוש הדורס האחרון.

Generics and collections

אנחנו נמצאים בתוך מחלקה של נקודה ורוצים למיין את הנקודות. מצד אחד נוכל למיין לפי הקרבה לציר האיקס או לראשית הצירים או לפי צבע הנקודה. לפנינו אפשרות לממש כל אחת מהאפשרויות הללו אך זו כפילות קוד וזה נגד עקרונות ה- oop . לכן נשתמש ב- $genrics$, כעת נוכל להגדיר אינטרפייס לכל סוג של אובייקט. אך אם יהיה לנו אלף אובייקטים, ניצור אלף אינטרפייסים? וודאי שלא. אנחנו נכיר כעת את הממשק $comperator$ שנמצא כבר בילד אין בג'אווה. הוא נראה כך -

```
point interface Comperator <T>{  
    int compare ( T a, T b);
```

כעת, כאשר ניצור מחלקות ונרצה למיין אנחנו נממש אותו ונעשה אובריידיד על המתודה $compareTo$ ונממשה מחדש.

כעת נדבר על *collection*: אוספים גנריים מאפשרים שימוש בתבנית יחידה לאוסף כללי כלשהו. נוכל ליצור אוסף מבלי לציין את שם הטיפוס. יש הרבה אוספים כמו $List < E >$ וכן $Set < E >$ וכן $Map < E >$ וכן $Queue < E >$ ועוד... כאשר נרצה ליצור אוסף מטיפוס מסויים אנחנו מתחייבים אל הטיפוס הזה בפני הקומפילר ואם כמובן ננסה להפר זאת לאחר מכן נקבל שגיאת קומפילציה.

נעיר שבידינו אפשרות אחרת: מדוע שלא נגדיר אוסף בצורת מערך מסוג *object*? הרי כולם יורשים ממנו. ובכן, זה חוקי אך לא נשתמש כי זה מסורבל - בכל פעם עלינו לזכור את הטיפוס של האיבר שהכנסנו וזה יוביל לבעיות כמעט וודאי שנשכח מי בידינו וזה יוביל לשגיאת זמן ריצה.

קלאס עם סימן שאלה $< ? > Class$ - אנחנו נקבע בזמן שאנחנו נשלח את הטיפוס המדויק, אם נגדירו להיות *string* ולאחר מכן נשלח איבר שאיננו *string* נקבל שגיאת זמן ריצה. נשים לב ש $Class$ זו מחלקה של ג'אווה שהיא גנרית. כיצד נשלח לבנאי של מחלקת הסימן שאלה את הטיפוס? *String.class* יהיה הדרך לשלוח. זו דרך נחמדה אך לא מומלץ להשתמש בה. עדיף גנריקס. בפירוט יתר אודות $< ? >$: מותר רק לקרוא ממנה נתונים ואסור להכניס אליה משהו. מדוע? זו הייתה רשימה כלשהו למשל $List < Integer >$ וקיבלנו אותה כ $List < ? >$. אין לנו מושג מה היה הטיפוס בה קודם ולכן אם נכניס כנראה שנהרוס. מותר רק לקרוא באמצעות $Object - val = list.get(i)$ למשל כי כל טיפוס בג'אווה הוא בפרט אוב'קט. אם מראש יודעים מה סוג הטיפוס למשל *string* אז אפשר להכניס רק טיפוסים מסוג סטרינג, הכנסת אינטגר למשל תביא לשגיאה.

לסיכום:

$Comparable < T >$ הוא ממשק להשוואה טבעית ונממשו בתוך המחלקה, חובה לממש מתודה *compareTo()* שנותנת מימוש יחיד

$Comparator < T >$ ממשק להשוואה חיצונית, אפשר לממשו מחוץ למחלקה, נשלח אותו כ *cmp* ארגומנט בשביל למיין לפי יחס סדר כלשהו. מחייב לממש מתודה *compare()*. עצם קיומו מאפשר לממש *sort* בצורה גנרית. תתכן מתודה *compareTo* אם לא מיממשי *comparable* (הרי זה שם) אך ההפך לא הנכון.

ברמה סינטקטית - T אומר שזה דטה טייפ גנרי, אם מופיעים (T, T) יתכן שזה מאותו טיפוס או מאותו טיפוס פולימורפי. אם יופיע $< T, V >$ זה אומר ששלחנו שני טיפוסים גנריים שונים, יתכן שהם שווים אך לא בהכרח קיים קשר (אפילו פולימורפי) בין השניים. בסופו של דבר הקומפילר יתרגם הכל לאוב'קט אבל זה רק אחרי שידוק שהכל חוקי.

כל מבני הנתונים הגנריים בג'אווה מימשו את *toString* ואת *equals*.
הן *Integer* והן *String* וכן *Double* וכו'... מממשות *comparable*.

File IO

(לא יהיה במבחן)

נרצה לעבוד עם קבצים. *streams* יהיה זרם של נתונים ממקום למקום - כמו זרם של מים. אנחנו פותחים קובץ, כעת ניתן לקרוא ממנו תו תו, ובסוף נסגור אותו בשביל להמנע מדליפת זכרון. כך עובדים בכל שפה עם קבצים. אנחנו עובדים על האובייקט שמקושר לקובץ ולא על הקובץ עצמו. כאשר נעשה *fis.read()* אנחנו מושכים יחידה אחת שמורכבת מבייט אחד של הקובץ, אנחנו משכנו בייט ולא ביט יש לשים לב. כאשר תגמר הקריאה, יחזיר *fis.read()* -1 ואז נדע שהקובץ הסתיים. את הנתונים נניח ששמרנו בתוך מערך, כעת הפעולה *write* יכולה לקחת את הנתונים ששמרנו במערך ולהתחיל לכתוב אותם. ניתן גם לקרוא שורה עד להופעת בקסלאש אן ראשון עם *readLine()*, הבאפרידר יעבוד עד לקליטת בקסלאש אן ויעתיק בייט לתוכו עד שיראה בקסלאש, ואז אפשר להדפיס את השורה שהייתה בו והוא ירוקן כל מה שהיה בתוכו. כאשר נעבוד עם *files* אנחנו נוסיף לכתובת זרוע שגיאה במידה ולא הצלחנו מסיבה כזו או אחרת לעבוד עם הקובץ. כיצד ניתן לכתוב לתוך קובץ? עם הפקודה *FileOutputStream*.

שים לב - כשאתה עושה *close* זו לא שגיאה לסגור משהו שהוא כבר סגור! (סגור אותו כמה שתראה זו לא שגיאת קומפילציה)

Exceptions

מדובר בחריגות. אנחנו מדברים כעת על מושג של לא תקין (ולא תקין אך לא נכון כמו עקרונות *oop*). יש אפשרות בג'אווה לבדוק האם קובץ קיים, נניח ובדקתי והוא קיים. ייתכן שרגע אחרי הוא יעלם כי מישהו מחק אותו באותו הזמן. זו בעיה עבורי בקוד, אני צריך לדעת זאת כאשר אני מתעסק עם קובץ. אנחנו נחלק את הנושא לשניים: בעיות בשליטתנו ובעיות שלא בשליטתנו אך צריך לטפל בהם.

מה יקרה כשזוהתה שגיאה? מערכת ההפעלה תהיה המעורבת ותהיה הראשונה שיודעת על הבעיה. מערכת ההפעלה שולחת סיגנל ל *JVM* (java virtual machine), שאומר לה: תעשה משהו עם הסיגנל אחרת התוכנית שלך תקרוס. *JVM* יוצר אובייקט מיוחד מסוג *error* שנזרק ככדור לתפוס את הבעיה בתוכנית.

בשביל שהתוכנית שלנו תוכל בכלל לתפוס חריג אנחנו צריכים לעטוף את הקוד עם `try`, כלומר נסה לעשות מה שתצא. אבל בסוף תגדיר בלוק של `catch`, אם וכאשר יגיע הכדור הזה - אנחנו נתפוס אותו בבלוק הזה. לחתימת המתודה יצטרף `throws exception`. נשים לב שניתן לשים כמה `catch` שנרצה, אבל זה כמו `switch case`, כלומר אנחנו הולכים תמיד לראשון שמופיע ולכן את ההכי ספציפי נשים הכי למעלה.

כאשר תפסנו את החריגה, ניתן להדפיס אותה בשביל לדעת מה השגיאה עם `system.err.println(e.getMessage())`. זה ידפיס הודעה מיוחדת בטרמינל. נוכל גם להדפיס `stack trace` בשביל לעקוב אחרי קריאות למתודות שהובילו לחריגה. כמו כן יש `activationFrame` שם יש שמירה של הזכרון של כל המתודות.

המחלקה העקרית בה רוב השגיאות נקראת `IOException`, יש בה שגיאות כמו קובץ לא קיים, אי אפשר לעבוד עם קובץ כי אין הרשאה, וכו'. יש מחלקות שיוורשות ממנה כמו `FileNotFoundException`, שמהשם ניתן להבין איזה חריגה תזרוק.

תמיד עלינו לסגור לבסוף את ה-`stream` על מנת להמנע מאיבוד זכרון. כיצד נסגור אותו בתוך `catch`? הרי אם נסגור בסוף המקרה הראשון, לא נוכל להגיע קדימה. על הנייר, נוכל לשים `try` ו-`catch` בכל סוויץ' קיים, אך נקבל כפילות קוד **וכאן זה לא טוב לפי עקרונות oop**. (אך זו לא שגיאה כמובן). לשם כך נגדיר `finally` - בכל פעם כאשר נעשה טרי אנד קץ' נוסיף `finally`, מדובר בבלוק של הקוד שמתבצע תמיד. בין אם קרתה השגיאה או שלא. הוא ירוץ תמיד אחרי הביצוע ב-`catch` שתפס את השגיאה. תמיד הקוד שם יתבצע. מתי לא? מקרה אחד בדיוק - אם יש פעולה של `exit` ב-`catch` בו יצאנו.

חשוב ליזכור - תחבירית לא חובה בכלל לשים `finally` בסוף!!!! וכמוכן שלא חובה `catch` תחבירית!

עלינו לדעת שבכל שפה יש שלושה `streams`: `system.in` זה ההקלדה למחשב, `system.out` להדפסות ו-`system.err` לשגיאות.

מתודות שזורקות שגיאה: מי שקרא למתודה כזו חייב לטפל בשגיאה וליצור לה טרי-קאץ', בקריאה למתודה יש לטפל בכך. אחרת תהיה שגיאת קומפילציה. בג'אווה הדרך הנכונה היא לטפל בשגיאה היכן שקראנו למתודה, שכן הטיפול הינו פרטני. האחריות היא תמיד על מי שקרא למתודה. נשים לב שישנה בעיה - אם תזרק שגיאה לא נגיעה לשורת ה-`close` כי בזמן ה-`try` אנחנו נקבל שגיאה ולא נחזור למתודה כלל, ושם הרי סוגרים את הסטרים, אז לא נסגור אותו ותהיה דליפה. מדוע אי אפשר לסגור אותו דרך ה-`main`? הרפרנס לקובץ הינו לוקלי של המתודה, ב-`main` לא מכירים אותו ואי אפשר לסגור אותו. לכן אנחנו נפעל בשיטה שנקראת `try with resource`, למעשה אנחנו נשים את הבאפר בתוך ה-`try` וכאשר אנחנו נצא מהתוכנית זה תמיד יסגור אותו לבד - וירד מאיתנו הלחץ. מימוש של זה יראה כך -

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

חשוב לשים לב! תמיד יקרא ה-`close` ב-`trywith...` וזה יקרה לפני שנגש אל ה-`catch` ואל ה-`finally`.

סוגי שגיאות: יש סוגי בעיות שניתן לתפוס אך זה אחריות המתכנת לטפל בבאג, ולא אמורים לשלוח שגיאה. זה נקרא `check`. אם לא נטפל הקומפילר יכעס ולא יתן לי לקמפל. לעומת זאת יש שגיאות שכן לא באחריות המתכנת - לכן נעשה עליהם טרי וקאץ'. למשל מה לא באחריות המתכנת? קובץ לא קיים וכדומה. זה נקרא `unchecked`. בג'אווה החליטו ש-`main` יכול לזרוק `exception`, מי שנמצא מעל ה-`main` הינו ה-`JVM` שיתפוס את השגיאה והתוכנית תקרוס.

אין חובה שה-`exception` שנתפוס תהיה בהכרח קשורה לבעיה בקוד, זה אמנם חסר טעם אך כן יתקמפל. כמו כן, מתודה יכולה לזרוק `unchecked` חריגה גם מבלי להצהיר בחתימה `...throws`.

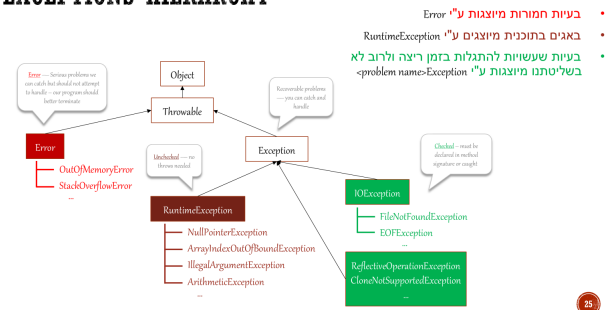
סיכום:

מקרה ראשון - לא נזרקת חריגה בתוך ה-`try`: קוד בתוך `try` רץ כרגיל ולא נכנסים לשום `catch`, בסיום ה-`try` המערכת סוגרת את כל המשאבים עם `close` (חשוב - `close` חייב לממש ממשק `closeable`), הולכים אל `finally` ומשם יוצאים מהבלוק להמשך הקוד.

מקרה שני - כן נזרקת חריגה בתוך ה-`try`: הקוד עוצר בעת החריגה היכן שהיא קרתה ולא ממשיך את השורות הבאות, לפני שקופצים לקאץ' ופיינלי **קודם כל `close` נסגר**, אח"כ מחפשים `catch` מתאים לחריגה - אם זה צק' אז קיימת כזו בהכרח אחרת הקוד לא היה מתקמפל, אם זה אנצ'ק אז הקוד יתקמפל ואם לא ימצא כזו בשלב זה הקוד יקרוס בזמן ריצה, אם מצאנו קיים - מריצים אותו, אח"כ כתמיד `finally` בסוף ואז ממשיכים רגיל בקוד (אם לא הייתה שגיאה בזמן ריצה)

מה יקרה אם בתוך `catch` זרקנו חריגה חדשה? הקוד ייעצר וייתייחס רק לחריגה החדשה, זה נחשב כאילו יצאת מבלוק טרי וקאץ' וזה מחפש מבחוץ אליו `catch` 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 שלי. בג'אווה יש המון תבניות עיצוב שהוצעו כבר למצבים יחסית שכיחים.

1. *Decorator / Delegation*: **האצלה**. "קשתן". נשתמש בו כאשר נרצה לקשט את האובייקט עם תכונות נוספות. למשל, נרצה להוסיף צבע לכדור או לסובב אותו וכדומה. נוסיף מחלקות שיקשנו אותו מסביב. מדוע צריך אותו? **נובע מכך שלא ניתן לרשת משני מחלקות ולכן הפתרון הוא להשתמש בתבנית עיצוב** - נוסיף שדה של *ball*. הקשתן מאפשר ליצור מחלקת בסיס עם תכונות והתנהגויות שיוגדרו מראש, ולהוסיף תכונות רצויות (למשל לצבוע אותו) באמצעות הקשתן. בנראות הקוד זה נראה כאילו ממש אנחנו מקשטים את האובייקט הפנימי ביותר החוצה. הקשתן אומר משהו פשוט - תעשה קודם מה שלימדו אותך לעשות, אני מבטיח לך שאשמור על ההתנהגות הזו, וארחיב עליה בדרך שאמרת לי לעשות (קשתן מרחיב - *extend*). למשל - אם נרצה ליצור משולש מסתובב עם נקודות, לא נוכל שמשולש ירש את משולש עם נקודות ומשולש מסתובב, לכן נקשט אותו במשולש מסתובב ומשולש עם נקודות. כל אחד מהם ירחיב בדרך שצווה לעשות. כך זה יראה במימוש - *ITrianglerst = new RotatingTriangle(new SpottedTriangle(new EquilateralTriangle(...)));* הקשתן מרחיב את ההתנהגות בזמן הריצה, לא בזמן קומפילציה, לכן יותר גמיש ודינמי. הטיפוס המאוצל יכול להקבע בזמן ריצה ולכן דינמי. מה החסרונות? פחות יעיל, לא קריא וקשה למעקב. בהורשה נשתמש כאשר אנחנו מתכננים היררכיית מחלקות ומתקיים עקרון *is - a* (מחלקת בן היא סוג של מחלקת אב) וכן כאשר אנחנו מעוניינים שכל תכונות ההורה יחולו גם על הבן. אחרת, נעדיף האצלה. **זה כלי חשוב שמתחרה עם הורשה אך לפעמים נעדיף להשתמש בו כאשר לא מתקיים עקרון ההחלפה (כל שימוש כרגע ועתידי A יהיה מתאים לשימוש ב B)**

דליגשיין בעקרון זה כאשר אני רוצה להפנות הלאה טיפול למישהו אחר - באה אליי בקשה להרצה של מתודה ואני רוצה להפנות אותה לשדה שלי שיעשה זאת אבל *extend* לא מתאים כאן כי האובייקט הזה לא בדיוק אני - למשל אפשר להאציל את טליה ממבני נתונים עם מרינה על הקורס כי יש קשר בין השניים אבל עדיין אין בניהם עקרון *isa* בין מרינה לטליה. (זה בול הדוגמה עם הריבוע)

קשתן - הוספה של התנהגות ו/או שדות תוך כדי שמירה על שדות והתנהגות מקוריים. הקישוט מתבצע ע"י האצלת כל הפקודות לאובייקט המקושט, בתוספת שדות והתנהגות רצויים.

2. *Observer/listner*: נרצה אובייקט שנוכל להסתכל עליו ולהאזין לו, כאשר נשנה משהו נרצה שכל המקומות שקוראים לו גם יעשו משהו. יש קשר ביניהם. בכל פעם שקורה לו משהו הוא יבצע *notify* לכל האחרים שיתעדכנו גם כן. למשל, כאשר כדור יעבור מהירות מסוימת נרצה להקטין אותה, אז נבצע *notify*. התבנית מאפשרת לאובייקט אחד להודיע לאחרים הצופים בו על שינויים במצבו, כך נוצר קשר דינמי בין האובייקט לצופים. הצופים יקבלו עדכונים באופן אוטומטי כאשר הם מתרחשים. **האובייקט הנצפה** - *subject*: מנהל רשימה של *Observers*. מאפשר ל-*Observers* להירשם או להסיר את עצמם מהרשימה. מכיל פונקציה *notify()* ששולחת עדכון לכל ה-*Observers* הרשומים. **הצופים** - *Observers*: אובייקטים שמחוברים ל-*Subject*. מקבלים הודעה אוטומטית כאשר יש שינוי ב-*Subject*. מיישמים ממשק מסוים, לדוגמה, *update()*.

3. *Factory*: תבנית עיצוב שמפרידה בין יצירת אובייקטים לבין השימוש בהם. מאפשר ליצור אובייקט מבלי לדעת מה הטיפוס המדויק של האובייקט. בכל פעם המפעל מייצר אובייקטים בהתאם לבקשת המשתמש. למשל - משחק איקס עיגול. מכרנו אותו ללקוח והוא בא יום אחרי ואמר שהוא רוצה שדרוג במשחק להוסיף *smartUser*. איך נממש? הרי צריך לעדכן את המשחק. הפתרון יהיה יצירת ממשק בשם *supplier* וכן מחלקות שיממשו את הממשק הזה, במקרה שלנו: שחקן מול רובוט, שני שחקנים, שני רובוטים. אנחנו יוצרים אובייקט מבלי לדעת את הטיפוס המסוים של האובייקט. מתי נשתמש בתבנית זו? כאשר מתודה במחלקה מסוימת מבקשת ליצור מופע של אובייקט ספציפי, אך איננו יודעים מראש מיהו כי זה תלוי למשל בבחירת המשתמש. לכן במצב כזה נשתמש בתבנית זאת, וניתן לו כמה אפשרויות ליצור מהם (ואז הוא יבחר, ובהתאם לסוויץ' קיים נגדיר את השחקן). בתבנית עיצוב זו יש חלוקת תפקידים די ברורה, המפעל אחראי לספק לי אובייקטים וליצור אותם והמתודות אחראיות על השימוש. ה-*supplier* יהיה אחראי לתת למפעל אובייקט, והמתודות יכתבו אותו דבר ללא שכפול קוד והיו מוכנות לעבוד עם אובייקט כללי.

הערה - *Builder* יתאים יותר למצבים מורכבים בתוכנית שלנו (ניו בתוך ניו....) נשתמש בו בעיקר כדי להגדיר מחלקה שמכילה מתודות שבונות שילובים נפוצים ומורכבים.

בעקרון לסיכום פקטורי מאוד פשוט, כשאומרים לו *get* מביא לי אובייקט, מייצרים פקטורי כשלא מסוגלים לבצע *new* ישר כשאני לא יודע איזה טיפוס אני רוצה. אך הבנייה עצמה של אובייקט פשוטה בניגוד לבילדר.

לפי מרינה סטודנטים מתבלבלים בין פקטורי לקלון, קלון זה שיש לי כבר אובייקט ביד ותעתיק אותו ופקטורי זה תן לי אובייקט. כמו כן לפי מרינה גם מתבלבלים בין פקטורי לסינגלטון - אין קשר בסינגלטון אתה מגביל כמות ובפקטורי יוצר אובייקט.

5. *singleton*: נרצה להגביל את מס' המופעים של אובייקט למופע יחיד (או מס' מופעים קבוע), כיצד? נגדיר במחלקה שדה סטטי בשם *instance*. נהפוך את בנאי המחלקה לפרטי, וכך נמנע יצירה של אובייקטים מחוץ למחלקה. כמו כן, ניצור מתודה סטטית שתחזיר את המופע. היא תבדוק האם *instance = null* ואם כן היא תקרא לבנאי, ותחזיר את המופע למשתמש. אחרת, הוא כבר קיים ויש לנו מגבלה של אחד ולכן היא פשוט תחזיר את השדה. **על הנייר סינגלטון זה רק עם שדה סטטי אבל עקרונית ניתן לממש באמצעות פקטורי דזיין פטרן.**

טוב לדעת - הסיבה שמותר לעשות בג'אוה *forEach* כלומר *for (int num: collection)* היא שמאחורי הקלעים ממומש איטרטור של המחלקה שנקרא.

נשים לב שיש ב-*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, המשתמש לא ידע. אבל אי אפשר לחשוף את המידע הזה כרגע כמובן. לכן אנחנו הולכים להשתמש במושג שהזכרנו, מדובר במחלקה בתוך מחלקה. ישנה מחלקה ובתוכה מחלקה פנימית. המחלקה הפנימית כן מכירה את תכונות *private* של המחלקה הראשית וזה החידוש. נגדיר את המחלקה הפנימית כמחלקה פרייבט וכעת גם אי אפשר בכלל ליצור אובייקט ממנה. כעת, נוכל ליצור את האיטרטור מבפנים ולא ישבר עקרון האנקפסולציה. למעשה מדובר על בנייה של האיטרטור מבפנים המחלקה.

Cloning&Immutability

Immutable - 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 חיי התוכנית הוא יהיה 4. בהמשך נדון בכך (שנה ב'). מה זה אומר בפועל? בהינתן קוד של *integer x=2, integer y=x,y++*, מה יתקבל כאשר נדפיס את *x*? 2. מדוע? *integer* הוא *immutable* ולכן זה יוצר אובייקט חדש ולא רפרנס.

clone - שכפול, בהינתן אובייקט נרצה להכפיל אותו ולקבל אחד זהה אליו. באופן נאיבי (לכאורה), נוכל לשלוח עותק של האובייקט. הבעיה? שלחנו רפרנס, ומישהו מבחוץ יוכל לשנות את האובייקט, ונשבר כך עקרון האנקפסולציה. כיצד נפתור את הבעיה? ניצור העתק של האובייקט, כך אף אחד לא יוכל לגשת אליו. אם כן, זה הרבה זכרון שנתפס. כמו כן, איננו יודעים את הטיפוס המדויק של האובייקט - בהינתן סטודנט, ייתכן שהוא סטודנט למדעי המחשב או סטודנט למדעי הדשא. לשם כך - בג'אוה בתוך *class object* מוגדרת מתודה בשם *clone*. המתודה היא מסוג *protected* (לא ניתן לדרוס אותה), היא מחזירה אובייקט, יוצרת העתק שטחי (נקרא גם *shallow copy*) של כל השדות של האובייקט המועתק. המתודה זורקת חריגה עבור כל מחלקה שלא תממש את *cloneable* - כלומר חייב לממש אותו. מדובר בממשק ריק שמסמן מחלקות שתומכות ב-*clone*, זה בעצם אישור מה *jvm* להתקדם. ייתכן שאובייקט ידרוש העתקה עמוקה (נקרא *deep copy*): למשל, נניח ויש לנו מחלקה סטודנט. למחלקה יש שם, כתובת (שזה ממש אובייקט) ותעודת זהות. ובכן, נבצע *clone* ותתבצע העתקה רדודה. זה מעתיק את השדות ללא לחשוב על ההשלכות - מהם ההשלכות? לסטודנט יש שם, ובכן, אנחנו יודעים שבג'אוה המחרוזות הזהות נשמרות באותו מקום, אבל זה בסדר כי לכאורה 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'אוה *string* הוא *immutable* - כלומר אי אפשר לשנות אותו, לכן אם יש שדות פרמיטיביים שהם *immutable* זה בסדר. אך הבעיה עם האובייקט כתובת שאליו עשינו העתקה רדודה - כעת נצטרך בתוך המחלקה של כתובת להגדירה גם כמממשת את *cloneable* ולבצע העתקה נוספת של השדות. כך למעשה נמשיך ליצור *clone* לכל מתודה במעין לופ עד שנגיע לאובייקט שמכיל רק משתנים פרמיטיביים. כלומר, עד שנגיע לאובייקט שהינו *immutable*.

הן *Integer* והן *String* הם *immutable*.

ובכן, פתרנו את הבעיה של השכפול. כעת נרצה לקחת רשימה של תלמידים ולשכפל אותם, זה יעלה לי המון!!! ויקח לי המון זכרון למעשה. ננסה לחשוב על פתרון אחר -

בהינתן שהסטודנט שאני מעוניין לשכפל היה *immutable*, היה קל. עם *clone* אחד וסיימנו. נוכל ליצור מחלקה שהיא תהיה *immutableList*, כלומר מחלקה שאיננה ניתנת לשינוי. גם היא תממש את *cloneable* והשדה שבפנים יהיה רשימה *mutable* אך *private* - לא יהיה ניתן לשנות אותה מבחוץ! לכן אם נבנה את המחלקה בצורה נכונה כן תשמר אנקפסוליישן. נראה כי בכל פעם שנרצה לבצע *add* לרשימה אנחנו נעתיק את כל הרשימה למשל. נזכר - מדוע בנינו את המחלקה? להמנע מבזבז זכרון וזמן. אבל כאן מבצעים אנסוף העתקות, בכל פעולת *add* מעתיקים את כל הרשימה. לכן, **נפצל למקרים. שני הפתרונות שלנו היו גרועים לאותה בעיה. בכל מקרה נעדיף להשתמש במשהו אחר בהתאם לאפליקציה שנפתח. אם אנחנו מעדכנים הרבה את הרשימה, נעדיף להשתמש בשיטה הראשונה.**

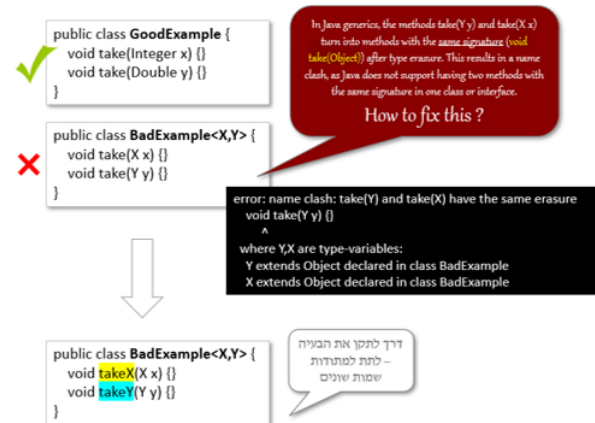
הערה על *immutable*: כל המחלקות העוטפות *Integer, Double* הן *immutable*. כאשר נעשה $Integer - x = 5$ ואז $x = 10$ כאן למעשה לא נשנה את ערך האובייקט המקורי אלא x יקבל הפנייה חדש לאובייקט *integer* חדש שמכיל

אם כל שדות המחלקה (כולל סטטיים) הם *immutable* אזי המחלקה היא *immutable*. זה שקלאס שלי אימיוטיבל 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 לזה. אם אני אימיוטיבל אני מרוויח אנקפסוליישן אוטומטית. לקלון יש חריגה *cloneNotSupportedException* והיא צ'ק.

OOP Advanced Features

כאן נציג מספר טריקים שפלים בג'אווה. יש המון המון כאלו. לפי מרינה - מצופה מאיתנו להכיר רק מה שכאן ברשימה.

1. *Generics*: נתבונן בדוגמה מטה. על פניו, המתודה נראית תקינה. אותו השם, אך מדובר בטיפוס שונה. מה כבר לא חוקי? המתודה השנייה, הקומפיילר יגיד שיש להם אותו *erasure* ולכן בזמן הקומפילציה, ג'אווה הגדירה שמתודות מטיפוס גנרי יהפכו ל-*take(object)*, מה שיביא לשתי מתודות עם אותו השם שמקבלות אותו טיפוס - וזו כמובן שגיאה. כיצד נטפל בכך? הדרך היחידה היא לשנות את השם של המתודות. בכל הקלאסים הגנריים, המתודות בסוף מתורגמות לאוב'קט.



דוגמה נוספת - נניח ואנחנו עובדים עם טיפוס גנרי ורוצים להפעיל על הטיפוס *x.compareTo(y)*, נקבל שגיאת קומפילציה. מדוע? הקומפיילר לא מוצא את המתודה. *object* שוב הוא הטיפוס אליו מתורגמים טיפוסים גנריים והוא לא מממש את הממשק *comparable*. מה הפתרון? צריך להוסיף לחתימה של המתודה בה אנחנו עובדים את השורה הבאה! `public static <T extends Comparable<T>> max...` **נשים לב** שזה יכול לעשות בעיות שכן ייתכן שמחלקה שאנחנו תלויים בה לא עשתה מימוש ל-*comparable* **למרות שהיא מחויבת (מתכנת גרוע) ואז נקבל שגיאה.**

תכונה נוספת על generics - נתבונן בדוגמה מטה. נחליט שאנחנו שולחים אל המתודה איזשהו *comperator*. נראה שהדוגמה הראשונה חוקית ומתקמפלת. נראה משהו מגניב בדוגמה של הקריאה - אנחנו קראנו ל-*max* ובצענו מה? הגדרנו בתוך קריאה למתודה, *new* לממשק! זה לא חוקי כמובן.... יצרנו **קלאס אנונימי** = חשוב מאוד!, אין לנו דרך לדעת מה השם של האובייקט (*jvm* נותן לו שם מוזר כלשהו מאחורי הקלעים), מדובר במחלקה שה-*jvm* מגדיר שמממש את *comperator* ולאחר מכן גם מממש את המתודה היחידה שיש בתוך המחלקה האנונימית - *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. *lambda* - נרצה להשתמש בקיצורי דרך. אנחנו נעשה חץ כמו בדוגמה הבאה: $\max("Hi", "Hello", (s1, s2) \rightarrow s1.length() - s2.length());$; $s1.length() - s2.length()$; זו הכתיבה הכי מקוצרת שיש בג'אווה כאשר מוגדר בפנים קלאס אנונימי. למעשה אמרנו לקומפיילר, כאשר אתה מפעיל מתודה מקס, אתה יודע שהרי אתה צריך לקבל שני שמות ואיזשהו *comparator*. אז, אני כבר מממש לך אותו. החל מהחץ, מתחיל המימוש של המתודה או הממשק. למעשה, את המתודה $a + b$ ניתן לכתוב ממש כ $(a, b) \rightarrow a + b$. בלי נקודה פסיק אפילו! מדוע? התכונה במקור מגיעה מפייתון. **נעיר כי ניתן לעשות שימוש בלמדה רק עבור מימוש מתודה אחת בלבד במחלקה אנונימית.** בנוסף, ישנה מתודה בשם *forEach* שאפשר להפעיל על *ArrayList*, כך למשל אפשר להריץ פקודה כמו *arr.forEach((i) → system.out.println(i))*. חשוב לזכור, כאשר נעשה למדה בתוך לולאת פור, המחלקה האנונימית שתיוצר תהיה זהה לכל הלולאות. לעומת זאת, כאשר מחקים את רעיון הפור ומבצעים שכפול קוד 3 פעמים למשל במקום לרוץ עד שלוש, יוצרו 3 מחלקות אנונימיות שונות.

כשכותבים למבדה נוצר קלאס אנונימי.

דוגמה מבלבלת: האם הקוד הבא תקין? הסוג של *obj* הוא תמיד *IA*, לממשק יש שתי מתודות. לאחת מהן יש מימוש דיפולטיבי (ריק), ולכן מותר להשתמש ב*lambda* גם עבור מקרים כמו אלו אשר המחלקה מממשת בפועל רק מתודה אחת (ואכן ממומשת כאן אחת בדיוק!). שהרי מדוע זה מבלבל? קודם אמרנו שמותר להשתמש בלמדה רק עבור מחלקות עם מתודה אחת לא ממומשת וכך ממשים אותה, אך מצב של *default* מתוד כן בסדר מבחינתנו.

```
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());
    }
}
```

user defined class - מחלקות שהיוצר יוצר בהרצה. *main* נחשב לאחת כזו (כי לכל קלאס אובייקט המתאר אותו) וכן גם *lambda* נחשבת לכזו כי היא מחלקה אנונימית. נזכור שבעת *for* נוצרת מחלקה אנונימית אחת (אופטימיזציה של ג'אווה) ובשכפול קוד יוצרו מס' מחלקות אנונימיות. **לכל קלאס יש אובייקט שמתאר אותו ולכן כאשר יוצרים מחלקה אנונימית נוצר אובייקט שמתאר אותו.**

3. *lambda* עבור מחלקה אבסטרקטית - נתבונן בדוגמה מטה. הדוגמה משמאל לא תעבוד ותחזיר שגיאה, מימין עם מחלקה אנונימית דווקא כן תעבוד. לא ניתן להשתמש בלמדה באופן ישירה על מחלקה אבסטרקטית ולכן משתמשים בקלאס אנונימי במקום.

```
abstract class A {
    public abstract void f();
}
class Main {
    public static void main(String[] args) {
        A a = () -> System.out.println("Hi");
        a.f();
    }
}
```

➔

```
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();
    }
}
```

error: incompatible types: A is not a functional interface
A a = () -> System.out.println("Hi");

5. *User defined class* - מחלקה שהמתכנת כתב. לא מחלקה של ג'אווה מובנית אלא כזו שאתה אחראי ליצירתה. כמובן שמחלקה שאני עובד בה נחשבת לכזו וכן מחלקות למבדה יהיו כאלו. נדגיש כי *string[] args* איננו אובייקט שנוצר כתוצאה ממשמש.

6. *Lazy class loading* - בג'אווה ה*JVM* לא טוען את כל המחלקות בזמן שהקוד נטען (כשתוכנית מתחילה לרוץ), במקום זאת היא טוענת את המחלקות רק כאשר הן באמת נדרשות במהלך הריצה של התוכנית. כלומר המשמעות היא שברגע שתעשה שימוש בפעם הראשונה במחלקה (למשל תיצור מופע או תקרא למתודה סטטית) אז תתרחש טעינת המחלקה, הטעינה קורית רק כאשר המחלקה נדרשת. לאחר מכן ניגשים לשדה סטטי או מתודה סטטית של מחלקה, קוראים *class.forName()* בשביל לטעון את המחלקה, מחלקת אב נטענת ואז גם מחלקות בן שגולשות אליה. למבדה בתוך לולאה תגרום לקומפיילר ליצור רק מחלקה אחת עבור הלמבדה ולא מחלקות חדשות בכל איטרציה (הסיבה היא בגלל *lazy*), הקומפיילר מבצע אופטימיזציה כדי למנוע יצירה מיותרת של מחלקות חדשות בכל פעם כשמבצעים איטרציה בלולאה.

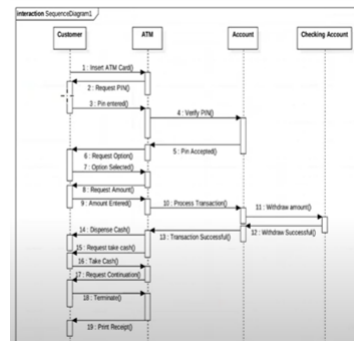
TDD- Test Driven devolpment

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

UML

נשתמש בגרפיקה על מנת לתאר את הקוד שלנו. בתכנות בחוץ ובתעשייה עובדים עם *uml* שעוזרים לעשות סדר בפרויקט גדול. כאשר ניגשים לכתוב אחד כזה נרצה להבין אילו אינטרקציות קיימות. למשל, בתוך מערכת של בנק - כאשר אני מפעיל מתודה של משיכת כסף, מי מעורב בכך? חשבון הבנק, הבנק, מערכת הפריטה. כאשר ניגשים לפרויקט אנחנו נעבוד בשלבים הבאים: נחשוב מה הדרישות שיהיו לנו מהמערכת (או אם אני בפרויקט קיים שמשפר - נחשוב איזה שיפורים לקוח ביקש ממני), נבין מי השחקנים במשחק, איזה מחלקות צריך עבורם ואיזה פעולות נרצה שהמערכת תבצע. וכן נתכנן את המערכת לפי מחלקות והורשה. ישנם סוגים רבים של *uml*, הוא מתחלק לשניים: יש כאלו שמגדירים את המבנה ויש כאלו שמסבירים התנהגות.

דיאגרמת ההתנהגות החשובה ביותר נקראת sequence diagram, היא מציגה את התקשורת בין הפעולות במחלקות השונות. ניתן ממש להמחיש את התהליך שקורה מבחינת זמן. אילו פעולות יתבצעו תוך כדי. דוגמה לדיאגרמה כזו:



הדיאגרמה שמתארת את המבנה הנפוצה ביותר הינה class diagram, והיא מכילה תיאור של הפרויקט שכולל את המחלקות שבו והפעולות וכן הקשרים ביניהם. כיוון שיש על כך שאלה במבחן שבו נתון דיאגרמה כזו ועלינו להסיק מהקשרים דברים, נתעכב וחשוב מאוד ליזכור את הסימונים הבאים:

מס' המופעים של הקשרים בניהם.

קו ישר עם מעויין ריק - מייצג קשר בו מחלקה אחת מכילה מחלקה אחרת וכן המעויין תמיד בצד של המכילה. הקשר אינו חזק במיוחד בין השניים במקרה זה שכן החלקים יכולים להתקיים גם בלי השלם. למשל - סטודנטים והרצאה, ייתכן הרצאה ללא סטודנטים (באסה למרינה). קשר שייכות חלש!

קו ישר עם מעויין אחד מלא (שחור) בצד אחד - מייצג קשר חזק בו מחלקה אחת מורכבת מאובייקטים של המחלקה השנייה (השנייה זו עם המעויין), ואין לה קיום בלעדיה. למשל - מנה ורכיבים של המנה. קשר שייכות חזק!

קו מקוקו עם חץ ריק בקצה - קשר של מימוש ממשק, החץ יוצא מהמחלקה ומצביע לכיוון הממשק.
קו ישר עם חץ ריק בקצה - קשר של ירושה בין מחלקות, החץ הוא לכיוון המחלקה שממנה אנחנו יורשים. כלומר כלב חץ חיה.

Iterator

בהרחבה על *iterator* כי זה מופיע בכל מבחן שאלות 18 – 20:

itertor: נרצה לעבור על אוסף כלשהו מבלי שאנחנו יודעים את מבנה הנתונים שהאוסף מאוחסן בו. נרצה לבצע על כל אחד מהאיברים פעולה. נוכל לעשות בקלות על *ArrayList*, אך אם נעשה זאת על רשימה זה לא יעיל. יעלה $O(n)$ רק בשביל לחפש את האיבר. זה קורה עבור n איברים ולכן מדברים על $O(n^2)$ זמן לבצע פעולה על המבנה. ומי אמר גם שהאיברים אחד אחרי השני שניתן ככה לעבור? למשל טבלת האש. לכן נרצה אובייקט בשם *iterator* שידע לעבור בצורה יעילה על מבנה הנתונים, מבלי להכיר מראש את המבנה שכן רוצים אנקספוליישן. לכל מבנה נתונים נבנה איטרטור שיראה תמיד אותו דבר ונוכל להשתמש בו בכל מבנה, יש ממשק שנכתב מראש וניתן להשתמש בו:

```
public interface Iterator<E>{
```

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

המטרה היא גישה אחידה וכללית לאיטרציה על אלמנטים.

א. `hasNext()` מתודה שתחזיר `true` כל עוד יש איברים שאפשר לעבור עליהם. מאפשרת ללולאה 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 נוכחי (זוג, צלע בגרף,), שהוא הרי זה שיודפס ב `while` של האיטרטור.

ג. `parpareNext()` לא חייבת להיות לרוב, אבל היא מתודה פרטית שתפקידה להכין מראש את האיבר הבא שאליו `next` יתייחס ולשמור אותו במשתנה פנימי. היא לא חלק מממשק רשמי אך נהוג להשתמש בה בשביל לשמור את המצב הבא מראש, להפריד בין לוגיקת האיתור של האיבר הבא ולבין פונקציות `next()` ו `hasnext()`.

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

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