

סיכום לקורס OOP

מאז מירב כהן-גנוז, ע"ב הרצאות של oz.campus ותרגולים

תוכן עניינים

1.....	תוכן עניינים
3.....	 חלק 1 - מבוא.
3.....	סילבו
3.....	מירקורים בסיכום
3.....	mosagim_bosisim_ale Java
4.....	תהליך הקומפילציה בג'אווה
5.....	mosagim_bosisim_bashfa
6.....	*** עמודי התווך ***
7.....	 חלק 2 - מאנקפסולציה עד אבסטרקציה.
7.....	כימוס (אנקפסולציה)
7.....	מגדירים נראות
8.....	API
9.....	סיכום: OOP בקטנה
10.....	ENUM
11.....	 חלק 3 - ממשקים ופולימורפיזם
11.....	ממשק Interface
14.....	פולימורפיזם
14.....	עקרון האחריות היחידה
15.....	תכנות למשק (Program to Interface)
16.....	tabniot_yitzob
19.....	עקרון הבחירה היחידה
20.....	 חלק 4 - ירושא
20.....	העמסה Overloading
20.....	הכליה Has-a / composition
20.....	ירושא / Is-a / inheritance
21.....	דרישה Overriding
22.....	המילה השמורה super
23.....	ירושא ופולימורפיזם casting
23.....	המרה וירושא casting
25.....	ירושא - ממשק - הכליה
26.....	METHODS_AL_OBIKUT
27.....	חבילות Packages & access modifiers
28.....	מודולים modules
29.....	סביבת עבודה IDE וקיצורי מקלדת
30.....	משחק Bricker (תרגיל 2)
37.....	 חלק 5 - מנגנוני מיחזור קוד
37.....	מחלקות Abstraction
38.....	פולימורפיזם של ירושא וממשקים
39.....	שיטות ממושכות בממשק

41	חסרונות של ירושה.....
41	עיצוב תוכנה: הכליה ומיחזור קוד
43	המשר Bricker (תרגיל 2)
44	חלק 6 - עוד סוגים API
44	static final
45	שיטות סטטיות
46	איברים שימושיים יכול להכיל ((recap))
47	מחלקה Utility
47	Shadowing
48	tabniot Utzob
52	חלק 7 - מבני נתונים.....
52	Collection Framework
54	שימושים של מבני נתונים.....
57	hashCode
58	Generics
65	shawiot & Exceptions
71	Nested Classes
73	Memento
74	חלק 8 - Lambdas & Callbacks
74	מחלקות מקומיות ואוניבריות
77	Functional Programming
82	effectively final
83	Callbacks
84	לסיום

*** להתמצאות בסיכון בדרכ'ך: הצגה > הציגת המתאר

חלק 1 - מבוא

וילבוי

[גיאוש](#) לשילובו של הקורס השנה ע"פ שבועות.

מירקורים בסיכום

מושגים, מילים שמורות, מחלקות ושיטות, הדgesות כלליות, עקרונות, יתרונות וחסרונות, תבניות עיצוב, כלל אכבע.

** זה סימון להערה של'

(לפעמים ציינתי הבדלים בחומר הנלמד בין הרצאות באתר campus לבין התרגולים של הקורס השנה)

מושגים בסיסיים על Java

• JDK / SDK

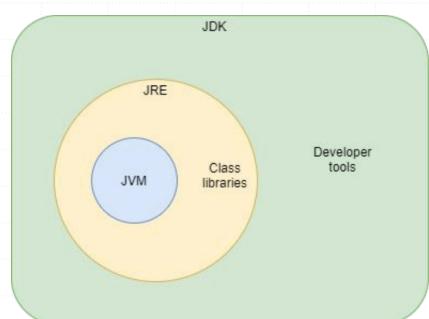
- JDK – Java Development Kit.
- The JDK is the full programming kit a developer needs to program in Java. It includes programs such as a compiler (javac), the Java Virtual Machine (java), a disassembler (javap), a debugger and a documentation tool (javadoc).
- It is important to note: JVM and JRE are included in the JDK.
- An SDK – Software Development Kit refers to any bundle of programs that a programmer needs to write code. So, a JDK is a Java SDK.

• JRE

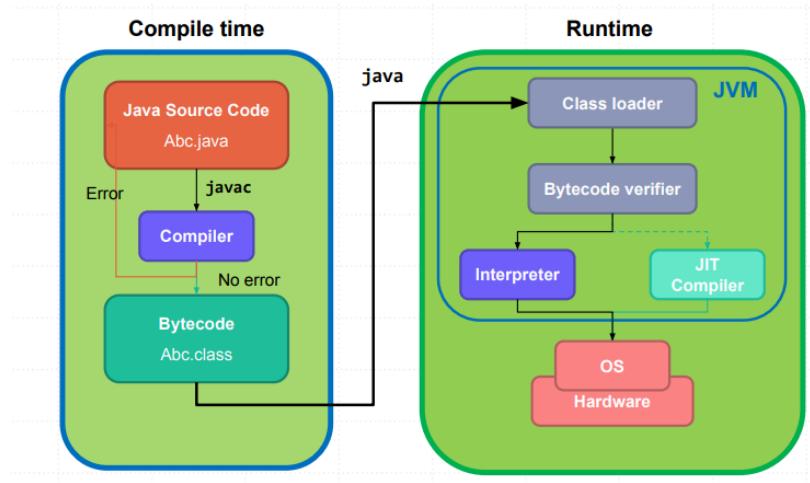
- JRE – Java runtime environment.
- A subset of programs and libraries that is included in the JDK. The Java Runtime Environment refers to all tools and programs that help execute a Java program. This includes the JVM, Java class libraries (more on this later).

• JVM

- JVM – Java virtual machine.
- A virtual machine (a program that simulates the architecture of a real computer) whose purpose is to execute Bytecode files. The JVM is part of the JRE and consists of a class loader, a bytecode verifier and a JIT (Just In Time) compiler and more



תהליך הקומpileציה בג'אווה



1. Compile time:
 - a. Java source code (.java file) goes through the Java compiler (the Java compiler is invoked using the 'javac' command, short for Java Compiler)
 - b. The Java compiler produces Bytecode.
 - c. Bytecode ends with the .class extension.
2. Bytecode:
 - a. Bytecode is called Bytecode because each command the JVM can run is the length of one byte (one byte = 8 bits, so there are $2^8=256$ possible operations in bytecode).
 - b. Bytecode isn't human readable but like assembly, each code has a mnemonic, a human readable representation.
 - c. We can check this human readable representation by using 'javap -c' on a .class file.
3. Runtime:
 - a. The 'java' command calls upon all the tools we need for runtime, it also instantiates the JVM.
 - b. We feed the JVM bytecode directly.
 - c. The JVM first loads our bytecode and some of Java's library code (Java.utils, etc.).
 - d. The class loader creates general memory for every class (we might learn more about this if we talk about reflection).
 - e. The bytecode verifier makes sure the bytecode is valid (properly formatted and made by a trusted compiler).
4. Execution Engine (in runtime):

made up of three items (depends on the JVM):

 - a. Interpreter – translates bytecode to machine code line by line to be executed immediately
 - b. JIT – Just In Time Compilation, if JVM sees a chunk of repeated code that is used over and over it can compile that chunk (during runtime) to produce more optimized code as interpreting line by line can be inefficient.
 - c. Garbage collection – more on this in a bit.
5. Java Garbage Collection
 - a. The last part of the execution engine.
 - b. Every JVM has a garbage collector which is solely responsible for clearing up memory.

- c. The GC works automatically, it clears objects that have no more references pointing to them (remember, all objects in Java are on the heap!).
- d. Similar to shared pointers in C++.
- e. The difference is that we can't control the GC and it is called whenever the JVM decides fit.

יתרונות וחסרונות

Pros	Cons
<ul style="list-style-type: none"> ▪ WORA (Write Once Run Anywhere) Bytecode works universally for all JVMs. The JVM is system specific, so it takes care of the JVM. ▪ Java is backwards compatible, we can compile using JDK8 and run it on a JVM in JDK16. 	<ul style="list-style-type: none"> ▪ Interpreting and compiling using JIT costs us in runtime performance. ▪ Automatic memory management leads to performance issues

מושגים בסיסיים בשפה

מרכיבי השפה:

- עצמים (Objects) - הייצוגים המרכיבים את התוכנית
- מחלקה (Class) - תבנית שמרכזת עצמים עם תכונות וקטגוריות מסוימות
 - עצם של מחלקה נקרא גם מופיע או `instance` של המחלקה
 - כל מחלקה מדירה אילו תכונות או `attributes` יהיו למופעים שלה.
 - הפעולות שמחלקה מדירה נקראות שיטות או `Methods`.

רכיבי מחלקה:

- טיפוס נתונים (data members) - שדות עם שם + type ששומרות ערכים במחלקה
 - `int speed = 0`
 - בנאי (constructor)
 - קונסטרוקטורים נקראים באותו השם כמו המחלקה
 - קונסטרוקטורים לא מחזירים שום דבר, כלומר אין להם ערך החזרה ואנו לא משתמשים בambilיה השמורה `return`
- מתודות
 - שיטה שמחזירה void לא מוכחה לכלול פקודת `return`.
- זיכרון:
 - זיכרון למחלקה (קוד)
 - לכל אובייקט שאנו מגדירים במחלקה אנחנו צריכים להזכיר זיכרון משל עצמו.

טיפוס נתונים:

- פרימיטיבים: `int, double, boolean, char`
- רפרנסים (reference) - מצביעים למחלקה

משחרר זיכרון פניו לשימוש עתידי - Garbage Collector

*** עמודי התוור ***

עצמים ומחלקות

אנקפולציה

API

אבסטרקציה

פולימורפיזם

ירושה

אסטרטגיה

ArrayList, HashSet, HashMap

Callbacks

בקורס השנה:

encapsulation, polymorphism, abstraction, inheritance

חלק 2 - מאנקפוזולציה עד אבסטרקציה

כימוס (מאנקפוזולציה)

מאנקפוזולציה אומרת בגודל לחלק את הקוד לקפסולות, והיא מורכבת משני חלקים:

1. **הفرد ומשל:** החלק הראשון הוא קודם כל לאגד ליחידה אחת את כל הfonקציות וגם את כל

המשתנים שעוטקים באותו תחום אחריות.

המושג עצם, object, נולד כדי לחת את הרעיון התאורטי של מאנקפוזולציה ולגלם אותו בקוד. העצם הוא הישות שמאגדת את המשתנים והפעולות הקשורות לאיזשהו תחום אחריות.

* השנה בקורס לעיקרון בזב קוראים מאנקפוזולציה, בנפרד מהעיקרונו הבא *

2. **הסתרת מידע:** החלק השני של מאנקפוזולציה הוא שכל עצם מפריד בין מה שמשרת יחידות אחרות ומה שנועד לשימוש פנימי.

חוץ מכמה פונקציות, כל עצם מסתיר מהעצמים האחרים את הfonקציות ואת המשתנים שלו כאילן הם לא קיימים.

از מאנקפוזולציה אומרת שאנו קודם כל מחלקים את הfonקציונליות בתוכנה לקפסולות שמרכזות תחומי אחריות, ובנוספַּה הקפסולות מסתירות כמה שיותר איברים.

בפרט את השדות של האובייקט, אותן משתנים הקשורות לתחום האחריות, אנחנו כמעט אף פעם לא נחשוף.

הבסיס של המתודולוגיה של תוכנות.

* השנה הקורס - המושג "מאנקפוזולציה" מתיחס להפרד ומשל", בנפרד מעיקרון הסתרת מידע.

יתרונות המאנקפוזולציה:

- הסתרת מידע מאלצת אותנו לכתוב קוד הקשור בתחום האחריות של אובייקט בתוך האובייקט.
- שינוי של השימוש במחלקה אחת לא מצריך שינוי במחלקות אחרות.
- אפשררת לתכנן את השינויים מראש ומגינה מפני שינויים לא מתוכנים.
- ניפוי שגיאות (דיבוג) – מכיוון שהקוד מחולק לחוקים לוגיים והאינטראקטיות בין העצים יותר פשוטות, אפשר לבדוק בדיקות לאיתור תקלות והדיבוג נעשה קל יותר.
- בנוספַּה, מונע באגים.
- רכיב שעומד בפני עצמו ניתן לשלב גם בתוכנות אחרות.

מגדירי נראות

שתי מילים שמורות ב'ג'אווה: public, private, ו-c-public.
איברים (שדות או שיטות) שממוסנים c-public זמינים מחוץ לעצם, ואילו nisioin גישה לאיברים שממוסנים private ייכשל בקומפיולציה.

כל מה שממוסנן c-public חשוף לגישה של עצמים אחרים,

מה שממוסנן private – מגודר, אפשר לגשת אליו רק מתוך המחלקה.

אמנם לא ניתן לגשת לממשתנה private ממחילה חיונית, אבל המשנה זהה לא באמצעות מוסתר ואף ניתן לעקוף את חסימת הגישה אליו, ولكن לא כדאי לשמור בו מידע רגיש.

המטרה של private היא לא לשמור משתנים בסוד, אלא ליצור design יותר טוב.

לכן קיימת מילה שומרה secret שהיא שונה מ-private, שנועדה להגן על משתנים המחזיקים מידע רגיש.

מגדירי נראות נוספים ודרישות בחומרת ההגבלה:

private > default (package-private) > protected > public

כל אצבע: שדות המחלקה יהיו private.

API

לאבירים הפומביים של מחלקה אנחנו קוראים ה-API שלו. API ראי תיבות של Application Programming Interface. כמובן, זה המשך של העולם החיצון עם המחלקה. במחלקות גדולות כדי להפריד את כל סוגי הטיפוסים: קבועים פומביים, קבועים פרטיים, שדות פרטיים, שיטות פומביות, שיטות פרטיות. תוכנה מונחית עצמים היא רשות של לקוחות וספק שירות. רוב העצמים משתמשים בשירות של עצמים אחרים, וכל העצמים, אויל חוץ מהעצם שמכיל את main, מספקים שירות. השירות בא לידי ביתי ב-API. כשרוצים לקבוע את ה-API של מחלקה כלשהי, תמיד נחשוב עליה מנוקדת המבט של הלקוחות שלה. המשך צריך להיות נוח, אינטואיטיבי, פשוט. שני עקרונות שעוזרים לנו להנחות את ההגדרה של ה-API של מחלקה הם מינימלי ו-absטרקטיבי.

עקרון המשך המינימלי

עקרון המשך המינימלי, Minimal API, אומר: Keep It Simple האובייקט צריך לעשות כל מה שמצופה ממנו, אבל הוא גם צריך לעשות רק מה שמצופה ממנו, וכל המידע גורע. מנוקדת המבט של הלקוח, יותר פונקציונליות זה יותר להבין, יותר לזכור ויותר מקום לטיעוויות. עדיף להוסיף מלהורייד שיטות שכבר כבר תלויה בהן.

כשאנו ניגשים לכתוב קוד חדש, נעבד בשלושה שלבים: שלב ראשון – **אנקפסולציה**, כלומר נחלק את העצמים לקפסولات לפי תחומי אחריות. השלב השני – **לקבוע מה יהיה המשך, ה-API**, של כל עצם. זה החוצה של המחלקה עם הלקוחות שלה. השלב השלישי – **מיימוש** המחלקה. בעצם, מדובר יכול להיות שבשלב השלישי, שנבוא למשתמש את החוצה הזאת, נדרש שדות או נחלק לשיטות או נוסיף קבועים, אבל כל האבירים שאנו מושפעים בשלב המיימוש הם פרטיים.

אבסטרקטיביה

כשאנו מפתח צריכים להריך את המחשבה מ"air" נפתח (מה יהיה יותר פשוט למימוש), ולהתמקד ב"מה" הלקוח רוצה וצריך, כלומר נוחות המשתמש. ההתרחקות מה"air" אל ה"מה" היא לא מבנת מלאה והיא תמיד דורשת את שני המרכיבים האלה: קודם כל להבין את הלקוח, ודבר שני, לחשב איך מתרגמים את המימוש שונה לנו אל המשך שונה לו. לתהילך הזה אנחנו קוראים **אבסטרקטיביה**.

אבסטרקטיביה היא התרחקות מהמיימוש והתקרכבות לצורך של הלקוח.

יתרונות האבסטרקטיביה: כל עוד ה-API שלנו אבסטרקטי, כלומר מורחק מהמיימוש, הוא לא משתנה אפילו אם המימוש משתנה. ואם ה-API לא משתנה, אז השינויים לא מתפостиים, ואז קל יותר לעשות טסטינג, חלוקת עבודה, דיבוג. אבסטרקטיביה עוסקת בהפרדה שבין "מה" המחלקה נועדה לספק לבין "air" שהוא עשויה את זה.

אבסטרקטיביה ואנקפסולציה הן מושגים משלים. בתכנון נכון של תוכנית, אפשר להריחיב את התוכנית, אך רצוי לא לשנות. (**עקרון open-close**)

סיכום: אנקפסולציה, הסתרת מידע וابتראקטיה

Principal	Pros	Tools
Encapsulation	<ul style="list-style-type: none"> Makes every module have a set responsibility. 	Classes
Information Hiding	<ul style="list-style-type: none"> Enforces strong encapsulation. 	Access Modifiers
Abstraction	<ul style="list-style-type: none"> Makes code less complex. Users aren't affected by changes in implementation. 	Interfaces Abstract Classes

סיכום: OOP בקטנה

נסכם את שלבי הפיתוח של תכנית מונחית עצמים קטנה, ואלו הם:

1. אנקפסולציה

לחולק את התכנית למחלקות, כך שכל מחלקה אחראית על נושא אחד מוגדר היטב.

2. קביעת ה-API

אמרנו ש-API הוא המשק התכנוני של המחלקה - application programming interface.

אנחנו מבטאים את התפקיד של המחלקה בבחירה האיברים פומביים שלה. דיברנו גם על שני עקרונות שעוזרים לנו בשלב קביעת ה-API:

a. minimal API - לא להכביר על המשק. להעדיף רשיימת שיטות פשוטה להבנה ופיענוח.

b. אבסטרקטיה: קודם כל היינו רוצים שהמחלקה תיראה מבחן. איך ניתן שירות נח.

השירות עשי להיוות (וכנראה גם כדי שייה) שונה מאוד מאיך שחשבנו למשמש את הפונקציונליות בפועל.

3. מימוש

כאן אנחנו מימושים את ה-API שהחלטנו עליו. בשלב זהה אנחנו יכולים להוסיף קבועים, או שדות,

או שיטות - לפי הצורך. בתנאי --- שהאיברים הללו פרטיים - הם לא חלק מהמשק. זהה קראנו

הסתרת מידע, וזהו חלק מהוות בשימור החלוקה שהגדרנו בשלב הראשון.

ENUM

טיפוס עברו קידוד קבועים.
ה-enum פה מתפרק לחברית כמו המילה השמורה class.
ניתן ליצור לו קונסטרוקטורים, מתודות, וטיפוס נטונים (members).
ניתן לעשות מערך של enums.
לא ניתן להמיר enum ל-int.

סינטקטו בסיסי:

```
enum Days{  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SHABBAS  
}  
  
class Main{  
    public static Days day = Days.SHABBAS;
```

מתוך:

```
enum Days{  
    SUNDAY("OOP!"), MONDAY("NAND"), TUESDAY("Probability"),  
    WEDNESDAY("OOP!"), THURSDAY("Algorithms"), FRIDAY("It's Friday!"),  
    SHABBAS("Shabbat Hayom!"); // each constant is made once and calls the constructor  
  
    Days(String dayCourse) // constructor for all the constants  
    {  
        this.dayCourse = dayCourse;  
    }  
    private String dayCourse; // private member  
    public String getCourse(){ // public getter for the member  
        return dayCourse;  
    }  
}  
  
class Main{  
    public static void main(String[] args){  
        for(Days day : Days.values()){ // for each loop  
            System.out.println(day.getCourse());  
        }  
    }  
}
```

חלק 3 - ממשקים ופולימורפיזם

ממשק **Interface**

טיפוס המהווה רשימה של שיטות (מתודות).

```
interface Carriable {  
    void attachTo(Child child);  
    void unAttach();  
}
```

מימוש - **implements**

מחלקה שמצוירה שהיא תומכת בממשק, חייבת למשתמש את כל המתודות בממשק, אחרת התוכנית לא תתאפשר. לעומת זאת, הממשק רק מגדיר איזה שיטות צריכה להיות לעצם מוצואה מחלוקת. ההצעה היא:

```
class Child implements Carriable, Talkable {  
    ...  
  
    String replyTo(String sentence) {  
        ...  
    }  
}
```

איןטרפייסים משמשים להגדרת סוג של חזירים או הסכמים שמחולקות מקבילות על עצמן. בנגד למחלקות, שמייצגות איזשהו מושהו בעולם, או בעולם התוכנה שלנו לפחות אינטרפייסים לא מגדירים מושהו קונקרטי, אלא מגדירים איזושהי תוכנה, הסכם או יכולת שמחולקות לוקחות על עצמן. נשים לב:

1. הממשק לא מגדיר את אופן המימוש.
2. לפיו מה שלמדנו עד כה, ממשק יכול להכיל רק רשיימת שיטות ללא מימוש, קבועים. על סוג איברים אחרים נדבר בהמשך.
3. ממשק הוא **צון של API** מסוים. אך אף שבממשק לא כתוב שהשיטה פומבית, הכוונה היא שבמחלקה המימושה השיטות האלה חייבות להיות פומביות.
4. מכיוון שמדובר בתנאי התנהגות מסוימת של עצם, אין מניעה עקרונית שעצם עם מספר התנהגות ימשך מספר ממשקים, אם כי נדר לראות יותר מאשר.

הרחבה - Extending interfaces

המילה השמורה **extends** מחייבת את מי שמשתמש את הממשק המורחב למשתמש גם אותו וגם את הממשק המקורי:

```
interface A {  
    void funcA();  
}  
  
interface B extends A {  
    void funcB();  
}  
  
class C implements B {  
    public void funcA() {  
        System.out.println("This is funcA");  
    }  
    public void funcB() {  
        System.out.println("This is funcB");  
    }  
}
```

```
public class Demo {  
    public static void main(String args[]) {  
        C obj = new C();  
        obj.funcA();  
        obj.funcB();  
    }  
}
```

11

כלומר בדוגמה זו -

מחלקה **শ** שמשתמש את ממשק A תצטרכ למשתמש את הפונקציה A funcA
מחלקה **শ** שמשתמש את ממשק B תצטרכ למשתמש את הפונקציה A funcB וגם את B funcB

המרה - casting, upcasting

• Examples:

- *Printable*: for classes that can be printed
- *Comparable*: for classes that can be compared to other classes
- *Clonable*: for classes that can be cloned

באו נסתכל על רפרנס מטיפוס שעוד לא ראיינו, c .Carriable

- c יכול להצביע על כל אובייקט שמשתמש את Carriable , Carriable כל מופע של מחלקה שמשתמש את Carriable .
- סוג הרפרנס הוא סוג הממשק .
- העצם הוא מופע של מחלקה קונקרטית .

דוגמה:

```
Carriable c = new FurFly();
```

מתבצע casting , המרת טיפוס , מהטיפוס FurFly לטיפוס Carriable .

זהו דוגמה ל upcast - הממשק הוא טיפוס גובה יותר מאשר שמשתמש אותו .

בגדיל, המטרה של רפרנס מסווג ממשק היא להיות מסוגל לקבל מופעים ממחולקות שמשתמשות את הממשק . כמו כן, כשמצביעים על רפרנס מסווג ממשק ניתן לקרוא אך ורק למתודות של הממשק, גם עם הרפרנס ה"מקור" היה בעל מתודות נוספות.

از Upcast או המרה כלפי מעלה, היא כשמצביעים על עצם דרך רפרנס מסווג של ממשק .

אחרי Upcast אפשר לקרוא רק לשיטות של הממשק .

Upcasting – ex. 2

- When upcast, an instance can only call methods that are part of the parent's API, but it uses its own implementation.

```

interface ParentType {
    void printType();
}

class Child implements ParentType {
    public void printType() {
        System.out.println("Child type");
    }

    void childPrint() {
        System.out.println("Child instance");
    }
}

public class Upcast{
    public static void main(String[] args) {
        Child childReference = new Child();
        childReference.childPrint(); // both valid
        childReference.printType(); // and will compile
        // upcasting to parent type
        ParentType parentReference= new Child();
        parentReference.printType(); // still OK
        parentReference.childPrint(); // Does not compile!
    }
}

```

Upcasting – ex. 3

- This example shows how a single class can be compartmentalized, without the functions knowing the exact type of the instance.



```

interface Printable {
    void Print(String strToPrint);
}

interface Faxable {
    void Fax(String strToFax);
}

interface Scannable {
    void Scan(String strToScan);
}

class CanDoAllPrinter implements Printable, Faxable, Scannable {

    @Override
    public void Print(String strToPrint) {
        System.out.println("Printing... \n" + strToPrint);
    }

    @Override
    public void Fax(String strToFax) {
        System.out.println("Faxing... \n" + strToFax);
    }
    @Override
    public void Scan(String strToScan) {
        System.out.println("Scanning... \n" + strToScan);
    }
}

```

תחביר - המרת (casting) וחלוקת מספרים

```

double d = 5.7;
int n = (int) d; // =5
int m = (int) Math.round(d); // =6

```

אם רצים שתוצאת חלוקה של שלמים תהיה שברית, יש להמיר את אחד המספרים ל double או להוסיף נקודה עשרונית למספר:

$$37.0/10 = 3.7$$

פולימורפיזם

*** עמוד תוך 5! ***

פולימורפיזם: רב-צורתיות, המצביע בו יש מספר סוגים שונים של עצמים שמשמשים את המשק (להלן: הגדרה הראשונה של פולימורפיזם).

לעתים מגדירים פולימורפיזם בצורה מעט שונה, בטור המצביע בו ניתן להסתכל על עצם של המחלקה המשמש באמצעות פרנסים מסוימים (על פי המשקאים אותם הוא ממשך), וכך לאותו עצם יש למעשה מספר "צורות" שונות. לדוגמה: לעיגול יש גם את הצורה Circle וגם את הצורה Shape, ועל כן, על פי הגדרה זו, הוא מרובה צורות (פולימורפי).

אצל כל עצם שמשמש את המשק Carriable-*Carriables*-יות עשויה לבוא לידי ביטוי בצורה אחרת. כי אם אחד יכול למשוך את השיטות של Carriable בדומה אחרת. אבל ככלום יש את השיטות של Carriable וכלום אפשר לדבר באותו אופן, כלומר לקרוא לאותן שיטות.

המשמעות של פולימורפיזם היא שניי *Carriables* (עצמים המשמשים את המשק) יכולים להיות ממחלקות שונות, אבל לי זה בכלל לא משנה. כשזה נוח לי, אני אתייחס לשנייהם כ-*Carriables* ואני אפילו לא חיב להכיר את המחלקה הקונקרטית.

* אבסטרקציה ופולימורפיזם הם הבסיס לתוכנות כלליים *

עקרון האחירות היחידה

The Single Responsibility Principle: Every class should have only one responsibility

תכנות למסך (Program to Interface

העיקרונות המנחה שלנו הוא **"תכנות למסך, ולא למימוש"**.

העיקרונות אומר שתמיד נשאף לכתוב קוד עבור הטיפוס הגבואה ביותר ועם המטרת הכללית ביותר.

יתרונות

1. הימנעות מכפל קוד - חסוך עבודה.
2. כלויות - משרות גם לקחוות לא מוכרים או שעדיין לא המצאו.
3. זה מאפשר לתוכנת המסך לשנות את המימוש - בעתיד ניתן להחליף את המימוש, והלkersה שתוכנת למסך לא יצטרך לשנות את הקוד שלו, וכך גם הלkersה שמשתמש בו, וכן הלאה.
4. הלkersה אינו מודיע לעצם הפסיכיפי שבו הוא משתמש, כל עוד העצם תומך במסך לו מצפה הלkersה, וכן ניתן לנצל את עקרון הפולימורפיים לעבודה עם מספר עצמים מסוגים שונים.
5. קל יותר לתוכנת למסך שכן המשמש אינו צריך להבין דבר מעבר למסך.

אנקפוסולציה מאפשרת תוכנות למסך

אם שברנו תוכנות למסך, כולמר ויתרנו על הכלויות ועשינו משהו אחד עבור מחלוקת קונקרטית אחת ומישהו אחר בשביל מחלוקת קונקרטית אחרת, סביר שגם שברנו אנקפוסולציה.
יש לציין שהאחריות שלנו בכתיבה המסך הוא לא לחת אחירות של עצמים אחרים ולא למש את ההתנהגות שלהם!

מבנה עיצוב

בעיצוב תוכנה, אחד הדברים שנדבר עליהם הוא תרחישים נפוצים שעולים בהרבה תוכנות, ופתרונות עיצוב שמתאים להם. לפתרונות אלה קוראים **מבנה עיצוב**.

את הפתרונותים מימושו כבר, וכך ניתן להשתמש בהם כיחידות בטור התוכנה שלנו.

יתרונות

1. בעיצוב תוכנה שלמה לא חסר על מה לחשב, אז זה מועיל במיוחד פעם אפשר להשתמש בפתרון מוקן לפחות חלק מהבעיה.
2. ללמידה פתרונות עוזר לנו ללמידה גם את הביעות.
3. כמו בכל מקצוע, למתקנים יש זרגון שעזר להם לתקשר רעימות. השמות של מבנים עיצוב הם חלק מהזרגון של תכונות מונחה עצמים ותכונות בכלל.

Factory

יש לנו ממשק פולימורפי עם כמה שימושים והיד עוד נתיה. מה שקבע זה הממשק, מה שמשתנה זה רשיימת המחלקות שמשמשות את המשתק. אנחנו רוצים להיות תלוים רק בדבר הקבוע ולבטל את התלות במימושים, אבל היירה של העצם מצrica לנקוב בשם של מחלקה קונקרטית שיוצרים.

אך אנחנו לוקחים את החלק של היירה ומיצאים אותו למחלקה ייודית, שככל תפקידה ביקום להיות זו שמייצרת את האובייקט. אם היא מייצרת את האובייקט, היא מאפשרת לאחרים לתוכנת רק למשתק.

למחלקה צו קוראים **פעעל**.

פעעל הוא עוד תצורה של הסתרת מידע, רק בסקלה של קבוצה של טיפוסים במקומם בסקלה של מחלקה. חוץ מהסתורת מידע יש כאן גם אבסטרקציה, כי המפעעל לא משתף את הלוקה באיך הוא מייצר את העיגול. כל עוד המפעעל מחזיר את מה שהלקה רצה, זה לא משנה איך הוא עושה את זה.

היתרון הגדול ביותר של תבנית עיצוב מפעעל הוא הפרדת הלוגיקה של הייצור והשימוש: הלוגיקה של ייצור המופעים נפרדת משימוש בהם.

בmarsh לדוגמה שדיברנו עליה בחלק של פולימורפים:

אנחנו מקבלים מהמשתמש את השם של הצורה (נניח שהפרמטרים המספריים מתאימים לה), אנחנו צריכים מפעעל, אך אנחנו פשוט יוצרים אותו.

נניח גם פרנסו בשם `baseShape` שיכיל את הצורה שהפעעל פולט.

כמובן, מהטיפוס הכללי `Shape`.

נשים בו את הצורה שייצר המפעעל, את זה נשלח לבנייה של `Building`.

בתכלס: מפעעל הוא פונקציה שלוקחת את המשתנים שהמשתמש רוצה ליצור מהם את האובייקט (לדוגמה מהרזה של המילה "דדור") ומהזירה את האובייקט הרצוי ("דדור").

- מומלץ למשוך עם `switch case` מפעעל:
- נרצה להשתמש בתבנית העיצוב "מפעעל":
- כאשר נרצה למשוך שיטה שתחזיר מפעעל מחלקה בין מספר אפשרויות שונות של מחלקות שהמשותף ביניהן הוא שהן ממשות את אותו ממשק או יורשות מאותה מחלקת אב.
 - כאשר נרצה להפריד בין יצירת האובייקט לבין העבודה אליו.

dekoratorim

dekorator (decorator), ובעברית "מDecorator" (design pattern) זאת תבנית עיצוב שבה מחלוקת ממשת ממשך כלשהו, SomeInterface, בעל Method :

```
class Decorator implements SomeInterface {  
  
    @Override  
  
    public void method() {}  
  
}
```

ומעבירה קראות ל-method שלה למетод method של אובייקט אחר, גם הוא ממש את SomeInterface

```
class Decorator implements SomeInterface {  
    private SomeInterface innerInstance;  
  
    public Decorator(SomeInterface innerInstance) {  
        this.innerInstance = innerinstance;  
    }  
  
    @Override  
    public void method() {  
        innerInstance.method();  
    }  
}
```

בנוסף, לפני או אחרי הקראה ל-method של המופיע הפנימי (innerInstance.method), היא מוסיף פונקציונליות נוספת (ומכאן השם, שכן היא "_decorator" את הfonקציונליות של האובייקט הפנימי).

```
class Decorator implements SomeInterface {  
    private SomeInterface innerInstance;  
  
    public Decorator(SomeInterface innerInstance) {  
        this.innerInstance = innerinstance;  
    }  
  
    @Override  
    public void method() {  
        ... additional "decorator" code ...  
        innerInstance.method();  
        ... additional "decorator" code ...  
    }  
}
```

דוגמיה מתרגול 3

Decorator Example

```
interface Fish {
    String getDescription();
}

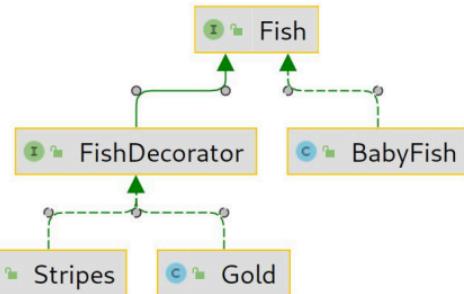
class BabyFish implements Fish {
    @Override
    public String getDescription() {
        return "fish";
    }
}

interface FishDecorator extends Fish {}

class Gold implements FishDecorator {
    Fish fish;

    public Gold(Fish fish) {
        this.fish = fish;
    }

    @Override
    public String getDescription() {
        return "Gold "+fish.getDescription();
    }
}
```



```
class Stripes implements FishDecorator {

    Fish fish;
    private String color;

    public Stripes(Fish fish, String color) {
        this.fish = fish;
        this.color = color;
    }

    @Override
    public String getDescription() {
        return color+" Stripes
"+fish.getDescription();
    }
}
```

בנייה דקורטורים אחד ע"ג השני:

```
public class DecoratorExample {

    public static void main(String[] args) {
        Fish clownFish = new BabyFish();

        clownFish = new Gold(clownFish);

        clownFish = new Stripes(clownFish, "White");

        //Alternative initialize
        //Fish clownFish = new Stripes(new Gold(new BabyFish()),"White");

        System.out.println("A Clown fish is: a "+clownFish.getDescription());
    }
}
```

סיכום

תבנית עיצוב היא פשוט פתרון עיצובי נפוץ לתרכיש נפוץ. התרכיש הנפוץ שלו דיברנו הפעם הוא שיש לנו ממשק פולימורפי, כלומר ממשק עם כמה מחלקות שימושיות, ולקחוות שימושים בו.

הלקחוות היי מעדים למסדק, ובפרט הם לא רוצים לקרוא לבניי של מחלקת קונקרטית. כל עוד הם מכירים רק את המשapk, לא צריך לעדכן אותם כשהם מוסיפים מחלוקת ממשת.

תבנית העיצוב מפעל היי דרך נפוצה לפתור את הבעיה של הלקוות. כשאנו כותבים את המשapk והמחלקות שימושיות אותו, בעסקת חבילת אונחנו מספקים גם מפעל. הלקוות אומרת למפעל מה הוא צריך, והמפעל כבר יקרה לבניי של המחלוקת הנכונה.

מה שהמפעל יחזיר זה רפנסו מסווג המשapk.

בסך הכל הלקוות יצריך להזכיר רק את המפעל ואת המשapk. רשיימת המחלוקת הקונקרטיות יכולה לשנותו, וללקוח לא אכפת.

עקרון הבחירה היחידה

tabniah העיצוב "מפעל" מתכתבת עם עקרון עיצוב ידוע: עקרון הבחירה היחידה (The Single Choice Principle). לפי עקרון הבחירה היחידה, אם יש משהו בתכנית שלו (זה יכול להיות כל דבר!) עברו תייכנה מספר אפשרויות, אך הרשימה המלאה של האפשרויות הללו תופיע רק במקום אחד.

The Single Choice Principle: Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list

המחלקה ShapeFactory שראינו היא דוגמה לכך: מקום שמספר ל��ות פוטנציאליים יכול קוד שבוחר צורה קונקרטית על סמך if-else, המפעל הופך למקומות היחיד בו מופיעה הרשימה המלאה של מחלקות כאלה. ב的日子里 אחרים, זה המקום היחיד בו נעשית בחירה בין האפשרויות (ומכאן שם העקרון). כמובן בספר למפעל, עקרון הבחירה היחידה רלוונטי גם במקרים אחרים של רשימה אלטרנטיבות לצריכה להופיע בקוד: יהיה לנו חשוב רשימה כזו תופיע אך ורק במקום אחד. אם בתרחיש הנטען זה נראה קשה, זו נורית אזהרה! שווה לחזור ולשקול אם היינו יכולים לעצב את הדברים אחרת כך שהבחירה תבודד.

חלק 4 - ירושא

העמסה Overloading

כשיש לנו צורך בשיטה מסוימת, אבל יש אפשרות לסוגי הפרמטרים שלה, ניתן להשתמש במנגנון העמסה או **Overloading**.

הweeneyון בהעמסה הוא שאפשר לכתוב כמה בנאים לאותה פעולה, אבל עם רשיית פרמטרים שונה. זה תופס גם לכמה שיטות עם אותו שם, כל עוד הן נבדלות ברשיות הפרמטרים - בפרט type הפרמטרים (השמות יכולים להישאר זהים).

יש מקרים שבהם אי אפשר להשתמש בהעמסה:

- אם השיטות נבדلات רק בערך ההחזקה, אבל יש להן אותו שם ואוותה רשיית פרמטרים.
- אם רק שמות הפרמטרים ברשימה שונים, אך לא הטיפוסים והסדר שלהם.

הכלה / composition

למחלקה א' יש רכיב מסווג מחלוקת ב' (מופע של אובייקט אחר).
משמעות ע"י **data members**.

ירושא / inheritance

מחלקה א' היא סוג של מחלוקת ב'.
משמעות ע"י המילה השמורה **extends**.

אם B שהוא מחלוקת האב שממנה יוצאה A, אז A הוא sub-class של B, ו-B הוא super-class של A. למחלוקת הירושא (the-sub-class) יש את כל הפיצרים (כולומר תכונות, Data Members, מתודות, קונסטרוקטורים...) של המחלוקת B, אבל היא גם יכולה להוסיף תכונות مثل עצמה. ניתן לבצע ירושא גם בין ממשקים, ומחלוקת שתממש את המשך המורחב תאלץ למשג גם את המתודות שלו וגם של ממשק-האב.

* יש להבדיל בין אובייקט ירוש לבין מופע של אובייקט, שכן מופע הוא שימוש בעצמו, וירושא היא הרחבה שלו.

תכונות של ירושא

- רקורסיביות - אפשר לשדרר מחלוקת שעשוות extends : מחלוקת C מרחיבה את B, מרחיבה את A ...
- טרנסיטיביות - אם A ירוש מ-B-C, אז A גם ירוש בצורה לא ישירה מ-C.
- כל מחלוקת יכולה להיות מחלוקת אב של מספר בלתי מוגבל של מחלוקות. כל מחלוקת יכולה להיות מחלוקת בן של מחלוקת אחת בלבד.
- Object או בשם המלא java.lang.Object, (نبין את המשמעות של java.lang.java.lang.Object) היא מחלוקת מיוחדת, שככל מחלוקת אחרת ב-Object ירוש ממנה. Object זו המחלוקת היחידה ב-Object, שאין לה מחלוקת אב אחרת.

מגדירים נראות וירושא

אמנם למחלוקת הירושא יש את כל השdot של האב שלה, אבל מחלוקת ירוש לא יכולה לגשת לשdot הפרטיטים של אבא שלה (אם אם הם השdot הפרטיטים שלה).

לדוגמה: Shiorsh m-Person ירוש ממנו גם את השדה הפרטוי name, אבל הוא לא יכול לשאול את עצמו מה השם שלו דרך השדה private. כולם לכל Student יש שם, אבל פשוט אין לו גישה אליו בצוואה

ישירה. הוא יוכל לקבל את השדה עם getter פומבי, במקרה שלנו getName, (כמו כל מחלקה חיצונית אחרת).

protected modifier

מגדיר נראות עם המילה השמורה **protected**.
protected שהוא מאפשר גישה למחלקות יורשות, ל-**sub-classes**, לקבל גישה למשתנים שאוטם הם יורשים (ולכל המחלקות שטרנסיטיבית ירשו מהן).
 ככלומר משתנים שקדם הגדרתו **private** ולמחלקה היורשת לא הייתה גישה אליהם, אם נגדיר אותם **protected**, עכשו כבר כן תהיה אליהם גישה.
 לעומת זאת, כל מי שלא יורש את המשתנים האלה, כל שאר העולם, יישיר להיות ממודר מהמשתנים האלה.
 אז **protected** על פניו נראה כאילו הוא נותן לנו איזשהו פתרון לעביה של גישת הירשים לשודות, אבל בעצם **protected** זה משווה שאנו נשתדל להימנע מלהשתמש בו, שכן אמן יש לו חשיבות במקרים מסוימים, אבל יש לו גם הרבה חסרונות:
 ברגע שאנו מגדירים משווה להיות חלק מה-API שלנו. **Private** זה משווה שהוא לא חלק מה-API - מי שמסתכל על המחלוקת שלנו לא רואה איזה מידע יש בה, הוא לא בכלל מודע לשודות ה-**private** והמתודות ה-**private** שיש לנו.
 מצד שני, **protected** זה כן משווהchosפים, ועם זה באים כל החסכנות של לחשוף דברים לעולם.
 למשל, זה יכול ליצור תלות מיותרת בשדה, ובהמשך יהיה קשה יותר "להחזיר אותה" את ההגדרה מ-**protected** ל-**private**.
 כלל האצבע שאנו ננסה לעמוד בו הוא תמיד להעדיף **private** או **final** (הסתרת מידע).
 בהמשך נדבר על חビルות ועל מגדירים נראות באופן כולל.

Driving Overriding

Overriding זה רעיון של לחת מחלוקת ולרשת ממנה, אבל תוך כדי שינוי ההתנהגות שלה.
 ככלומר דרישת היא קיום של מתודה עם חתימה זהה במחלוקת האב ובמחלקת הבן, עם שימוש שונה. אם קוראים למethode מאובייקט של המחלוקת היורשת (הבן), אנחנו נקבל קריאה לקוד החדש במקום לקוד הישן.
 הדרישת מאפשרת למשמש את API של **superclass**, תוך התאמת לאובייקט הנוכחי.
 הוספה "@Override" לפני חתימת הפונקציה מס'יעת בדיבוג, שכן היא מצינית שנעשה שימוש בדרישה של מחלוקת אב, ואם קיימות אי התאמות בחתימה - המחלוקת לא תתקמפל.

Overriding vs. Overloading

Provides the specific implementation of a method from its superclass.	Used to increase the readability of the program.
Occurs in two classes with an inheritance relationship .	Performed within a class .
The parameters of the overriding method are the same as the original method.	In case of method overloading, parameters must be different.
Runtime polymorphism.	Compile-time polymorphism.
Return type must be same or covariant in method overriding.	Has nothing to do with return-type.

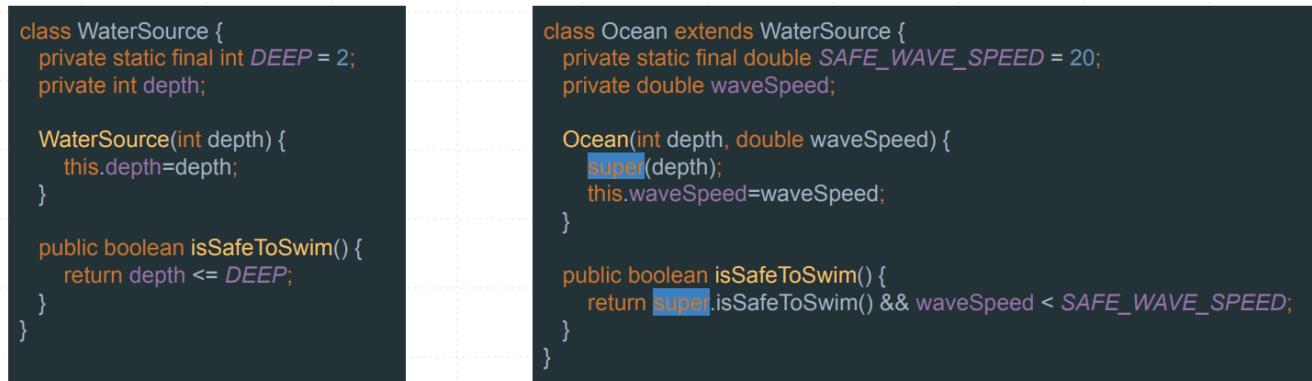
המילה השמורה super

משמשת לגישה לשדות של מחלקת האב, וקונסטרוקטור שלו.
בפרט מאפשרת לנו לגשת מethodה שאנו חווים למתחודה של
האב, ולעשות שימוש בקוד שלו.

דוגמא: פה יודפו

```
class WaterSource{  
    String name="Water Source";  
}  
  
class Ocean extends WaterSource{  
    String name="Ocean";  
  
    void printName(){  
        System.out.println(super.name);  
        System.out.println(name);  
    }  
  
    public static void main(String[] args) {  
        Ocean sea = new Ocean();  
        sea.printName();  
    }  
}
```

Water Source
Ocean



```
class WaterSource {  
    private static final int DEEP = 2;  
    private int depth;  
  
    WaterSource(int depth) {  
        this.depth=depth;  
    }  
  
    public boolean isSafeToSwim() {  
        return depth <= DEEP;  
    }  
}  
  
class Ocean extends WaterSource {  
    private static final double SAFE_WAVE_SPEED = 20;  
    private double waveSpeed;  
  
    Ocean(int depth, double waveSpeed) {  
        super(depth);  
        this.waveSpeed=waveSpeed;  
    }  
  
    public boolean isSafeToSwim() {  
        return super.isSafeToSwim() && waveSpeed < SAFE_WAVE_SPEED;  
    }  
}
```

11

פה לקחנו את הבניי של האב והרחבנו אותו: השתמשנו בsuper כדי להשתמש באתחול הבנים של בניי האב, ואז אתחלנו את השדה waveSpeed של האב והרחבנו אותו עם תנאי נוסף. בנוסף לקחנו את המethodה isSafeToSwim של האב והרחבנו אותה עם תנאי נוסף.

בנייה ללא ארגומנטים ובניה דיפולטי

קונסטרוקטור ריק נוצר באופן דיפולטיבי אם הוא לא כתוב במפורש, ואפשר גם "לדרוס" אותו ע"י כתיבת מפורשת של קונסטרוקטור ללא ארגומנטים.
כאשר מרכיבים מחלוקת, חייבים לקרוא לsuper() בשורה הראשונה של הקונסטרוקטור של הבן, אא"כ יש לבב בניי ללא ארגומנטים (או אין בניי = בניי דיפולטיבי). קלומר אם יש לבב קונסטרוקטור שלא מקבל שום פרמטר - אז הקונסטרוקטור זהה ייקרא, ואם אין לו צה - תהיה שגיאה.
באופן כללי ניתן לומר ש- super של הקונסטרוקטור תמיד נקרא או בצורה ישירה או בצורה עקיפה.

ירושא ופולימורפיזם

upcasting ופולימורפיזם עובדים בירושא בדיקן כשם שהם עובדים עם ממשקים. כשבושים **cast** מאובייקט בן לאובייקט אב, המתודות של המופע יריצו את הקוד של מחלקה הבן. לעומת זאת, לא ניתן לגשת למופיע של האב למתודות של מחלקה הבן. לעומת זאת, לא ניתן להשתמש אך ורק ב-API של מחלקה האב (זהו יתאים לקוד שהבן מימוש), ולא יהיה ניתן לגשת למתודות המורחבות של מחלקה הבן. אחרי המرة **כליי מעלה** (לטיפול של ממשק או של מחלקה אב), **סוג המצביע קובע איזה שיטות אפשר להריץ**, ואילו **סוג העצם** קובע איך הן ממומשות.

הمرة וירושא casting

זה הרעיון של להתייחס לאובייקט מסווג אחד בתור אובייקט מסווג אחר, זהה אחד השימושים המרכזיים בפולימורפיזם.

upcasting

ניקה אובייקט בן, ונתייחס אליו כאובייקט אב, ככלומר באופן כללי יותר - אנחנו עולמים בהיררכיה המחלקטית. כאמור, אחרי **Upcast** אפשר לקרוא רק לשיטות של מחלקה האב. דוגמה:

```
Animal a = new Cow();
```

ניתן להמשיך ולהחליף את **c** כך שיצביע על מחלקה-בן אחרת, לדוגמה:

```
a = new Dog();
```

הمرة נעשית פה באופן לא מפורש, שכן הקומפיילר מזהה את הצורך בהمرة.

ניתן גם לעשות המرة מפורשת באופן הבא: (**Explicit Up-Casting**) - לא מומלץ, שכן אין צורך

```
Animal cow = (Cow) new cow();
```

downcasting

הمرة כלפי מטה. **חייב להיות Explicit**, ככלומר מפורשת. בנוסף, יכולה לדרוש רק לאחר **upcasting**: **downcasting**

```
Animal animal = new Cow();
```

```
Cow cow = (Cow) animal;
```

כלומר אנחנו לוקחים אובייקט-אב ומתייחסים אליו כמורע של אובייקט-בן. החלק המודגם הוא ההمرة המפורשת, ובליידי תהיה שגיאה.

לא ניתן לעשות המرة בין שני סוגים שונים של מחלקות ירושות (בניים), או בין מחלקות לא קשורות. **הבעיה העיקרית ב-downcasting**, שבגללה הפעולה הזאת מאוד לא מומלצת, היא שלמרות שלפעמים לא יהיו כתוצאה מההمرة שגיאות קומפליציה, ההمرة יכולה ליצור שגיאות בזמן ריצה (שגיאות זמן ריצה זה רע מאוד).

סיבה נוספת היא שהمرة כזו מיפוי נתקוד לא גמיש ולא כללי - צמצמנו את הטיפוס והפכנו אותו מכללי לספקיבי. כאשר אנו מבצעים **downcasting**, אנו מטפלים בפחות מקרים.

instanceof

OPERATOR של java שמאפשר לנו לבדוק האם אובייקט הוא מופיע של טיפול או ממשק מסוים. לכוארה **instanceof** יכול לשמש אותנו כדי לבדוק לפני שאחננו עושים **Downcasting** ולודא שאין לנו שגיאות ובאגים, אבל זו אופציה מאוד לא מומלצת.

עדין יש סבירות גבוהה מאוד לבאגים, זה עדין קוד לא כללי ולא גמיש (שכן לא ניתן להריץ אותו ולהוציא מקרים נוספים שהוא לטפל בהם), ולכן נועד להימנע מזה ולתקן את הקוד שלנו בצורה טובה יותר (לפי עקרון הפילמורפיזם).

דוגמאות:

```
public void draw(Shape shape) {  
    if (shape instanceof Circle) {  
        System.out.println("Circle with radius " + ((Circle)shape).getRadius());  
    } else if (shape instanceof Square) {  
        System.out.println("Square with width " + ((Square)shape).getWidth());  
    }  
}
```

כלומר אנחנו בודקים אם הצורה שקיבלנו היא בעצם עיגול או ריבוע.
התוכנית תאלץ ללקת "למאתורי הקולעים" ולבדק אצל כל מחלקה אם היא המיושר הרלוונטי של האובייקט
שקיבלנו. בנוסף, בכל הוספה של אובייקט מסווג חדש היינו צריכים לטפל שוב בקוד זהה ולעדכן אותו.

ירושה - ממשק - הכלה

משק מאפשר לנו מיחזור קוד + פולימורפיזם
 הכלה - רק מיחזור קוד.

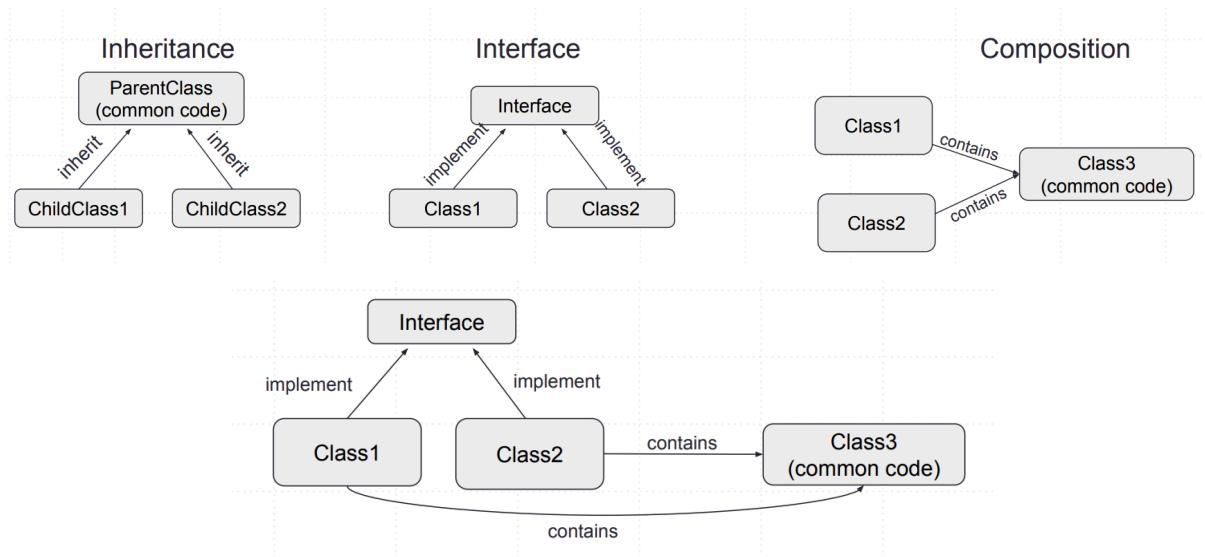
Inheritance vs Interfaces

Inheritance lets us reuse code instead of duplicating it and use Polymorphism.

- Polymorphism & Common behavior → Inheritance.
→ Always consider also Interface + Composition.
- Polymorphism without Common behavior → Interface.
- Common behavior without Polymorphism → Composition.
- Polymorphism & Common behavior → Always consider also Interface + Composition.

בקיצור:

1. אם יש קוד משותף שצורך להיות בימוש של כל המחלקות, נעדיף פולימורפיזם - שימוש בירושה.
2. אם אין קוד משותף - נעשה שימוש בממשקים.
3. כאשר יש צורך בשימוש נקודתי בקוד משותף בין מחלקות - משתמש בהכללה: ניתן ליצור מחלקה ספציפית עם הקוד המשותף, ונותרמש בה במקרה הצורך.
4. נשים לב ש Java פחות ידידותית לירושה, אך לרוב נעדיף ליצור מבנה קצר מרכיב יותר של תוכנית שאין בה שימוש בירושה.



מетодות של אובייקט

toString

הmethod `toString` קיימת באופן דיפולטיבי במחלקה `Object` (אך כל המחלקות) ומאפשרת ייצוג של האובייקט כמחרוזת.

לרוב נרצה לדרס את method זה ע"מ ליצור ייצוג אינפורטטיבי יותר של האובייקט שלנו.

equals

method המקובל אובייקט מסוג `Object` ובודקת האם הוא שקול לאובייקט הנוכחי. בעוד שמספר מיטיבים אפשר להשוות עם '`==`', השוואה זו בין אובייקטים תהיה חלולה (כמו `String`) ולכן נרצה את method `equals` ע"מ ליצור אפשרות להשוואה בין אובייקטים של המחלקה.

נשים לב שאנו מקבלים אובייקט מסוג `Object` בפורמט של השיטה, ולכן אפשר בערך לשלוח לשיטה `getName` עצמה שלא רק שם לא אותו אובייקט, הם בכלל לא מהמחלקה הנוכחי. אז בהתחלה של כל דרישת `equals` צריך לבדוק קודם כל אם האובייקט שקיבלנו להשוואה הוא באמת מהמחלקה שלנו, ולאחר מכן אפשר לבדוק אם השדות הרלוונטיים להשוואה אכן שוויים.

דוגמא:

```
public boolean equals(Object otherObj) {  
    if(!(otherObj instanceof Song))  
        return false;  
    Song otherSong = (Song) otherObj;  
    return getName().equals(otherSong.getName()) &&  
        getArtist().equals(otherSong.getArtist());  
}
```

שימוש לב שהשתמשנו ב-`instanceof`, ושלמעשה כך נראה כמעט כל שימוש של השיטה!
שימוש `instanceof` הוא לא מקור כל הרוע, במילויו כשהוא רק בודק אם עצם הוא מINSTANCE של המחלקה הנוכחי.
לעומת זאת השימוש בו הוא נורת אזהרה אדומה שימושים בו כדי לבדוק את הטיפוס הקונקרטי מבין
כמה אפשרויות במקומות להשתמש בפולימורפיזם (למשל: אם העצם הוא ציוף תעשה כך, אם העצם הוא כלב
תעשה אחרת וכן הלאה).

Packages & access modifiers

חבילות הן בעצם תיקיות - הן אוחזות את הקוד המשותף (מחלקות משותפות וממשקים). **Package** זו דרך לחלק את הקוד לחולקות שהן הגיוניות. הן תורמות לאנקיופסולציה של הקוד.

המילה השמורה **package** מגדירה את החבילה של המחלקות השונות. החבילה מכירה רק את מה שבתוך החבילה, והוא מופרدة מ"העולם החיצון".

ע"מ לגשת מוחץ לחבילה למחלקה כלשהי צריך את התנאים הבאים:

- שהמחלקה `class className` שב`package1` תהיה צריך את התנאים הבאים:
`public class className` שב`package1`
- שבמחלקה השנייה שב`package2` נויס `import package1.className`

Modifier	Description
Default	declarations are visible only within the package (package private)
Private	declarations are visible within the class only
Protected	declarations are visible within the package or all subclasses
Public	declarations are visible everywhere

Modifier	Class	Package	Subclass	World
<code>public</code>	✓	✓	✓	✓
<code>protected</code>	✓	✓	✓	✗
<code>no modifier*</code>	✓	✓	✗	✗
<code>private</code>	✓	✗	✗	✗

חbillah Difpolitybit

כאשר לא מצינימם לפני איבר אף הרשות נראות (כלומר לא כתבים `private`, `public` או `protected`), האיבר מקבל את הרשות ברירת המחדל. בג'אווה הרשות זו היא הרשות "חbillah" (package).

אנחנו גם יכולים להגיד מחלקות ללא שם `modifier`, ואז המחלקה עצמה היא בהרשות `Package`. מחלקות `public` מגדירות את ה-`API` של ה-`Package`, דומה ל-`API` של מחלקה.

סיכום: מגדירי נראות

עד כה למדנו על 4 מגדירים נראות של איברים בתוך מחלקות. איבר (שיטה או שדה) יכול להיות `private`, `protected`, `public`, וعصוי למדנו הגדרת נראות רביעית: נראות חbillah (ברירת המחדל בג'אווה). איברים בנראות חbillah נגישים, ובכך, מכל מחלקה שבתוך החbillah.

לטיפוסים עצם (מחלקות ולמשקדים), יש רק שני מגדירים נראות אפשריים. טיפוס יכול להיות מוגדר בלבד מגדיר נראות, למשל `Foo class`, ואז הוא בנראות חbillah וזמן רק בתוכה. אחרת, הוא יכול להיות פומבי, למשל `public class Foo`, ואז הוא זמין גם ממחבילות אחרות. טיפוס לא יכול להיות `private` או `protected`.

כמובן, הנראות של איברים במחלקה חסומים על ידי נראות המחלקה עצמה. למשל, אם המחלקה יכולה בנראות חbillah, בודאי שלא ניתן לראות אותה או אף איבר שלה (לרובות הפומביים) מוחץ לחbillah.

מודולים

עד עכשוו למדנו על מחלקות ועל הקשרים ביניהן, למשל באמצעות ירושה או ממשקים. בתוכניות גדולות שיש בהן מספר רב של מחלקות האדריכלות של התוכנה היא בעודה די מורכבת, אחד הכלים החשובים שעומדים לרשוטנו בתכנון של תוכנות גדולות הוא הרעיון של Module - רכיב תוכנה. מודול הוא אוסף של מחלקות וממשקים. המודול מאפשר ליצור היררכיה גבוהה יותר מחלוקת למחלקות, שכן כתת המחלקות מוחולקות למודולים.

דרך טובה ליציג מודולים היא באמצעות חבילות, Packages.

API בשימוש מודולים

כמו למחלקה, גם למודול יש API. ה-API של מודול הוא חלק מהמחלקות והמשקי שבתוכו, והחלק الآخر הוא לשימוש פנימי של המודול. הטיפוסים הפנימיים של החבילה הם לא private כמו שהיא לנו במחלקות, אין בכלל טיפוסים פרטיים, הם פשוט בלי כלום. קלומר יש לנו טיפוסים פומביים, למשל public interface, public class, public enum, או טיפוסים פנימיים, ואז כתבים רק class modifier או interface בלי modifier של נראות.

דוגמה למשחק האיקס-עיגול שעשינו, רק כתת עם חלוקה למודולים:



בעבודה עם חבילות, מומלץ שכל טיפוס יהיה חלק מ חבילה בעלת שם.

סביבת עבודה IDE וקיצורי מקלדת

IDE ראשי תיבות של Integrated Development Environment, או בעברית סביבת עבודה. אנחנו משתמשים בIntelliJ.

קיצורי המקלטים של מדריכנו:

- shift+F6 לשינוי שם
- alt+enter להצעות תיקונים מצד סביבת העבודה
- F2 לדילוג בין שגיאות הידור
- ctrl+tab למעבר בין קבצים
- ctrl+space לבקשת הלשמה אוטומטית
- sout להדפסה, ctrl+/ להערה או הוצאה מהערכה
- alt+insert להכנסת איברים
- ctrl+o למשתמש שיטת ממשך או דרישת
- ctrl+x/v ל-cut/paste של שורה
- alt+shift+up/down להזנת שורות
- ctrl+q לפתיחת תיעוד
- fori להגדרת לולאת for סטנדרטית
- shift+tab על מספר שורות כדי להזין אותן ושמאלת
- ctrl+C כדי ללכט להגדירה של המחלקה\משתנה\שיטה המסומנים
- ctrl+B כדי לתקן את המילה

משחק Bricker (תרגיל 2)

*** החלק הזה בסיכון מוקדש לתרגיל 2, וככה"נ לא רלוונטי למבון. אפשר לkippoz לחלק 5! (עמ' 35)
קיישו לסרטונים ולהסביר ההתחלתי של חלק זה (מומלץ לקרוא!)
כל הקבצים של BouncingBall קיימים שם להורדה. זה יחסור לכם הקלדות מיותרות 😊

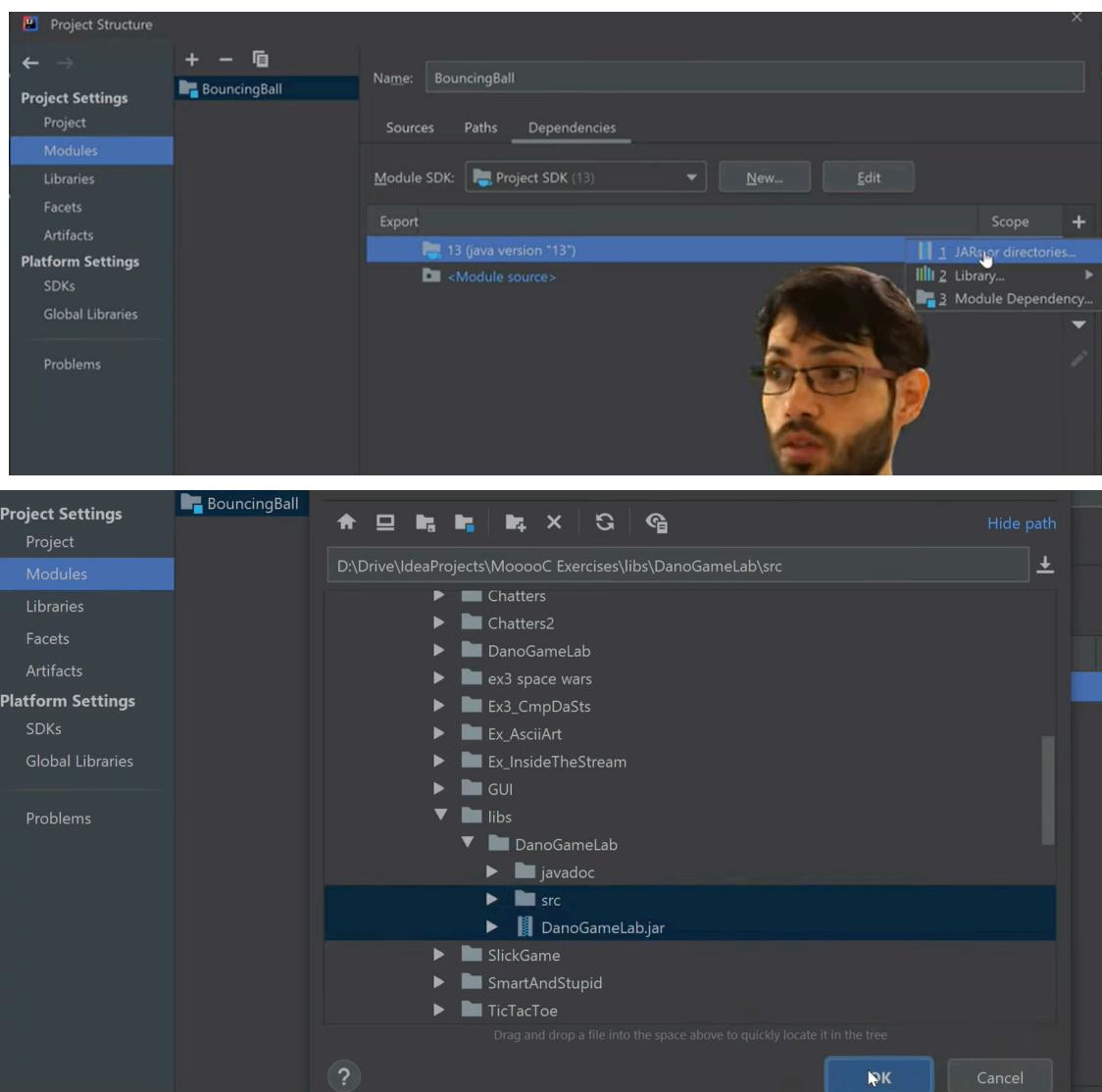
הגדירה ושימוש ב DanoGameLab

(יש הסבר מפורט במדריך ההתקנות - תכל"ס מומלץ לעבוד על פיו וולדאג לחלק הבא)

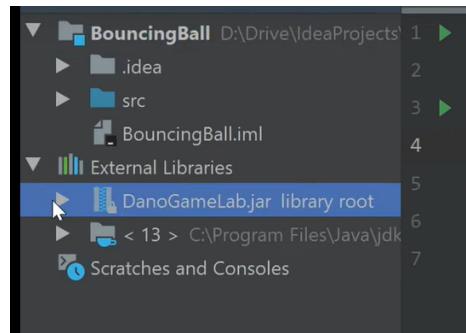
1. התקנת הסכירה ויצירת הפרויקט Bricker

2. צרו בפרויקט חבילת (package) חדשה בשם bricker.main: כפטור ימני על src, ואז בתוכה הפנימית (כפטור ימני על שם החבילה החדשה), צרו מחלקה חדשה בשם BrickerGameManager.

3. הגדרת תלות ב:DanoGameLab-3



cut מופיע בפרויקט:



דוגמה לשימוש בסיפוריה:

```
1 import danogl.GameManager;
2 import danogl.util.Vector2;
3
4 public class BouncingBallGameManager {
5
6     public static void main(String[] args) {
7         new GameManager( "Bouncing Ball",
8             new Vector2( x: 700, y: 500));
9     }
10 }
```

הגדירו גם אתם ב-BrickerGameManager שיטת main, וצרו בה מופיע חדש של GameManager מקבל את כותרת החלון, ואת המימדים שלו בצורת מופיע של Vector2. שימוש לב לשורות import. בנוסף, בשונה מהסרטון, חybim להוסיף גם קרייה לשיטה חנו או ששם דבר לא יקרה. אם הכל הילך כמורה, נפתח חלון המשחק.

*** אם זה לא עובד: קיימת אופציה פחות מומלצת - להעתיק את תיקית danogl ישירות לפרויקט.

הוספה ball & paddle

*** בעניין אפשר קודם ליצור אובייקטים של כדור/פadel אז להוסיף אותם למין (קצר והגיוני יותר) - פשוט להעתיק מתיקית BouncingBall ולשנות במקרה הצורך חלק א:

```
1 import danogl.GameManager;
2 import danogl.util.Vector2;
3
4 public class BouncingBallGameManager extends GameManager {
5
6     public BouncingBallGameManager(String windowTitle, Vector2 windowDimensions) {
7         super(windowTitle, windowDimensions);
8     }
9
10    public static void main(String[] args) {
11        new GameManager(
12            windowTitle: "Bouncing Ball",
13            new Vector2( x: 700, y: 500));
14    }
15
16 }
```

הגדירו את המשחק ב-BrickerGameManager, וצרו ב-main מופיע של המחלקה שלהם. הקראיה לשיטה run שמקורה ב-GameManager עדין הכרחית לפתיחת החלון. אגב, בגרסה שהורדתם של DanogameLab, למחלקה GameManager יש גם בניית פרמטרים (שמגדיר את המשחק להיות במסך מלא ולא שם). בשבייל להגדיר חלון שאינו במסך מלא עם כותרת כרצונכם, כדי להשתמש באותו בנאי בו נעשה שימוש בסרטון.

חלק ב: שימוש לב - מומלץ להשתמש בתיעוד המתודות כדי להבין (ctrl + q)

```
public void initializeGame(ImageReader imageReader, SoundReader soundReader, UserInputListener inputListener, WindowController windowController)
super.initializeGame(imageReader, soundReader, inputListener, windowController)

Renderable ballImage =
    imageReader.readImage( path: "assets/ball.png", isTopLeftPixelTransparency: true);
GameObject ball = new GameObject(Vector2.ZERO, new Vector2( x: 50, y: 50), ballImage);

gameObjects().addGameObject(ball);|
```

}

- דירטו את initializeGame
- הורידו את הקובץ assets.zip מהמודול, וחלצו את הקבצים לתיקיה עם השם assets.
- **שימוש לב: המיקום של התיקיה assets חייב להיות כמו בסרטון (תיקיה אחות ל-src)**
- התשתמשו בתמונה ball.png וצרו GameObject שמייצג את הכדור בגודל 20X20 פיקסלים.

- רשמו את הcador עצם-שחקן של המשחק. הכל בדיק כי שנעשה בסרטון. הcador מופיע בחולן שלכם?

חלק א:

```
WindowController windowController) {
super.initializeGame(imageReader, soundReader, inputListener, windowController)

Renderable ballImage =
    imageReader.readImage( path: "assets/ball.png", isTopLeftPixelTransparency: true),
GameObject ball = new GameObject(Vector2.ZERO, new Vector2( x: 50, y: 50), ballImage);
ball.setVelocity(Vector2.DOWN.mult(100));

Vector2 windowDimensions = windowController.getWindowDimensions();
ball.setCenter(windowDimensions.mult(0.5));
gameObjects().addGameObject(ball);
```

הגדירו גם אתם את הcador להתחיל במרכז המסך, עם מהירות התחלתית כלשהי.

שיםו לב! בסרטון נשלח הfrmater 0.5 לשיטה `mult`, שבסרטון מצפה ל-`double`. בגרסתה שהורדתם, השיטה מצפה לא ל-`double` כי אם לא ל-`float`. הfrmmitib `float` מאד דומה ל-`double`, רק תופס חז' מה מקום (בתמורה לפחות טווחן). שברים שמצוינים כ-`float` צריכים להיות מצוינים כךalla, על ידי הוספת `f` בסוף השבר, למשל: `0.5f`.

לחלופין ניתן להשתמש בביטוי מסוג `double` ולעשות לו המרה:

`(float) 0.5`

עד כדי ההבדל הזה, הקוד שלכם נראה יהיה מאד דומה לזה הסרטון.
ודאו שהcador שלכם זז.

*** (בנוסף בהמשך יהיה צריך לשנות ל `Ball` (`new Ball`))

חלק ב:

```
//create paddle
Renderable paddleImage = imageReader.readImage( path: "assets/paddle.png", isTopLeftPixelTransparency: true),
GameObject paddle = new GameObject(Vector2.ZERO, new Vector2( x: 200, y: 20), paddleImage);
paddle.setCenter(
    new Vector2( x: windowDimensions.x()/2, y: windowDimensions.y()-30));
gameObjects().addGameObject(paddle);
```

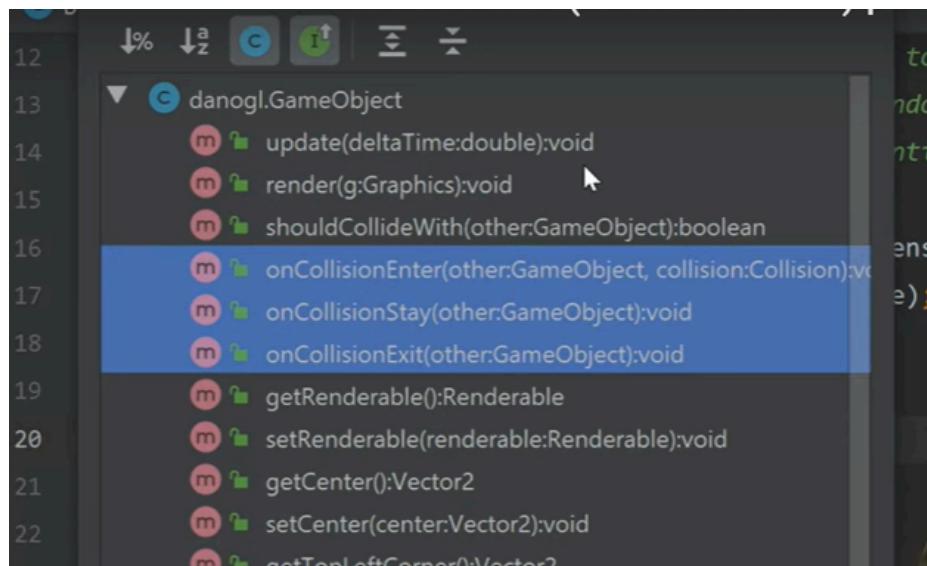
הויספו גם אתם דיסקית כמו הסרטון, בגודל 100x15.
שיםו לב: הסרטון, התמונה של הדיסקית היא צזו שבה הפינה השמאלית-עלונה לא מייצגת שקיות. בתמונה שלכם, היא כן מייצגת שקיות (הfrmater השני של `imageReader.readImage` צריך להיות `true`).

הערה: אם מה שהיינו רוצים זה כן רק מלבןצבע אחד, אפשר היה לעשות זאת גם באמצעות המחלקה `danogl.gui.rendering.RectangleRenderable` במקום `danogl.gui.rendering.Rectangle`. בידקו שהדיסקית מופיעה בחלון!

טיפול במקרה שבו המשחק לא/תקין/עמו מדי:

```
@Override  
public void initializeGame(ImageReader imageReader,  
                            SoundReader soundReader,  
                            UserInputListener inputListener,  
                            WindowController windowController)  
{  
    super.initializeGame(imageReader, soundReader, inputListener);  
  
    int targetFramerate = 80;  
  
    windowController.setTargetFramerate(targetFramerate);  
}
```

התנגשויות



נדרכו את שלושת השיטות הנ"ל כדי שיטפלו בהتانגשויות.

צרו את המחלקה `Ball` בתוך חבילה חדשה `bricker.gameobjects`. כמו בסרטון, דרכו את `onCollisionEnter` כך שכאשר כדור מתחילה התנגשות באובייקט אחר, רכיב המהירות שמקביל לנורמל יתהף.

בנוסף, הוסיפו למחלקה משתנה פרטי `collisionCounter`, המונה את מספר ההتانגשויות של הכדור, ודאגו לקדם את המונה בכל פעם שקוראים לשיטה `onCollisionEnter`. דאגו כמובן ליצור גם שיטה פומבית `getCollisionCounter` כך שהיא ניתנת לקבל את הערך שלו מחולקה. משתנה זה ישמש אותנו בהרחבה של המשחק.

אל תשכחו בזמן יצירת הכדור לקרוא לבניית המחלקה `Ball` במקום `GameObject` (בניגוד למשהו אחר). כמובן זה יצריך לעשות `import` ל-`Ball`. בדקו שהכדור חזר מהדיסקית!

גרמו לכך להשמי "בלופ!" כשהוא מתנגש במשהו והופך כיוון.
פשוט להעתיק את בטיפול בסאונד מ BounvingBall

כרגע ה-paddle שלכם הוא מופיע של GameObject. הגדרו אותו במקום להיות מופיע של מחלקת חדשה בשם Paddle (כמו בסרטון), שתוגדר בחבילה .bricker.gameobjects. עדכנו את הקוד של הדיסקית (Paddle) כך שהיא תזוז ימינה ושמאליה לפי המקשים. זכרו להגדיר את קבוע המהירות של הדיסקית float, ולא כ-double כפי שנעשה בסרטון. שימו לב! עליהם גם לדאוג שהדיסקית לא תצא מגבולות המסך בעקבות לחיצה על המקשים. תוכלם להשתמש במתודות getTopLeftCorner ו-setTopLeftCorner כדי לבדוק שהדיסקית לא חורגת מגבולות המסך, וכך למקם אותה במקרה במקרה של חריגה.

גרמו לכך שלכם להתחיל באופן אקרים לאחד מכיווני האלכסון. למה אנחנו לא מתחילה בזווית אקראית כלשהי? ניתן היה לעשות זאת באמצעות השיטה Vector2.rotate ושליחת פרמטר אקרים. אבל אנחנו לא רוצים שהכדור ינוע בכיוון שכמעט חלוטין אופקי; בחירה בין ארבעת כיווני האלכסון היא פשוט הדרך היחידה להימנע ממצב זה纯粹 ומאידך להוציא אקרים. המשך הסרטון מגדיר ומימוש את המחלקת AI_Paddle. היא לא נדרשת למשחק Bricker שלנו, ומילא מעוניין לצפות במימוש שלה מזמן לדרג ליחידה הבאה.

קירות וruk
[קישוב למדריך](#)

לבנים
[קישוב למדריך](#)

בדיקה הפוד

תנאי הפסד במשחק שלנו יהיה דומה לזה שבסרטון. כלומר אם הcador נופל לתחתית המסך זה נחשב לפסילה, אבל אצלנו התנאי יהיה קצת יותר מורכב. במשחק שלנו, לשחקן יש כמות של 3 פסילות לפני שנגמר המשחק. אם הcador נופל ונותרו פסילות, הcador מאותחל חזרה במרכז המסך, עם מהירות על אלכסון רנדומלי, ומספר הפסילות יורד באחד. אחרי 3 פעמים שהcador נופל לתחתית המסך המשחק נגמר בהפסד. משמש בקוד את הבדיקה של תנאי הפסד. דאגו לכך שהמשחק לא יהיה תלוי בכך שכמות הפסילות המשקם יימילט היא דוקא 3, ואם נרצה בעתיד לשנות את זה זה לא יצריך שינוי בקוד. אם המשחק נגמר דאגו שיוויון חלון עם ההודעה

"You lose! Play again?"

ועם כפתורים של Yes/No שבחם יבחר המשתמש אם לשחק שוב או לסגור את המשחק.

כדי שהמשתמש ידע מה כמות הפסילות שנשארה לו, נוסף את מספר הפסילות הננותרות בשתי תצורות, תצוגה גרפית ותצוגה נומרית. בחלק הזה תצטרכו לחשב בעצמכם איך למש את המחלקה/מחלקות שיציגו את כמות הפסילות. עליים, בסופו של דבר, לעמוד בדרישות הבאות:

תצוגה גרפית - יש להציג למשתמש על המסך לבבות כמספר הפסילות הננותרות. השתמשו בתמונה `heart`. שימו לב שבשלב זה הקוד שלכם צריך לתרום לכך שמספר הלבבות המוצגים ירד, אבל בהמשך התרגיל נרצה גם שמספר הלבבות יגדל במהלך המשחק עד למקסימום מסוים, דאגו שהקובד שלכם תומר בשנייהם. הנתונים המדוייקים של גודל ומיקום נתונים לבחירתכם.

טיפ: `GameObject` יכול להכיל בתוכו `GameObject` אחרים. כמו כן, כדי שאובייקט יופיע על המסך הוא חייב להיות מושך בעצמו לרשות האובייקטים של המנהל ולא מספיק שאובייקט שמכיל אותו יהיה ברשימתו.

תצוגה נומרית - יש להציג למשתמש על המסך את מונה הפסילות שנותרו לו בצורה מספרית. צבע הספרה יתחלף לפי כמות הפסילות:

- 3 ומעלה - צבע ירוק
- 2 - צבע צהוב
- 1 - צבע אדום

כדי להציג טקסט על המסך תוכלו להשתמש במחלקה `TextRenderable` ובשיטה שלה `setString` וכן תוכלו לשנות את צבע הטקסט באמצעות השיטה - `setColor` והקבועים `Color.green`, `Color.yellow`, `Color.red`. תוכלו לקבל דוגמא של איך מוצגת כמות הפסילות בצילומים המופיעים בהקדמה לתרגיל.

הערה: אין צורך לעדכן את המונחים להיות 0 במקרה של הפסד, והודעת סיום המשחק מספקת.

חלק 5 - מנגנוןן מיחזור קוד

מחלקות אבסטרקטיות

מוטיבציה

כאשר אנחנו רוצים להשתמש בירושא, וקיים מתודה שנרצה שהמיימוש שלה יהיה שונה בכל מחלוקת בת. לדוג' בימיוש מחלוקת ANIMAL ומחלוקת בת - COW, DOG, CAT, המתודה speak צריכה להיות שונה בין מחלוקת לחלוקת. מה יהיה תוכן המתודה () ?ANIMAL.speak()

מה הפתרונות האפשריים?

- לא למשם () ANIMAL.speak - ובכך לא אפשר לכל מחלוקת בת למשם את המתודה כרצונה. בעייתי כיוון שפוגע בעיצוב הקוד כך שמייצרים מתודות שונות רובות, כמו כן - פתרון זה לא מנצל את קונספט הירשה בו API אמר לשמש למזהה בין המחלוקות השונות.
- שימוש ריק ל() ANIMAL.speak ודרישה של כל מחלוקת בת לימיוש שלה. הבעיה - אם תהיה מתודה בת שלא תמשך speak אנחנו מפרים את API. בעיה נוספת היא שם אנחנו יוצרים instance של Animal וקוראים למתודה speak שלו, זה עלול ליצור התנהגוויות בלתי רצויות בקוד.

הפתרון הרצוי - מחלוקות אבסטרקטיות.
שימוש במילה השמורה **abstract** מגדיר כי לא ניתן ליצור מופע שלהן. זה מתרחש כך:

```
public abstract class Animal { ... }  
Animal animal = new Animal(...); // Compilation error
```

למעשה התפקיד של ANIMAL הוא להיות בסיס למחלוקות בת.
למחלקות אבסטרקטיות ניתן להגדיר מתודות אבסטרקטיות:

```
public abstract class Animal {  
    // An abstract speak method.  
    // To be implemented by Animal sub-classes.  
    public abstract void speak();  
}
```

המחלקה האבסטרקטית מחייבת הרבה ממנה לדרש את המתודה האבסטרקטית ולמשם אותה, אחרת מחלוקת הבת לא תעבור קומפליזיה.
בדוגמה לעילו: כל מחלוקת יורשת מ Animal חייבת למשם את המתודה speak.

ירשה בין מחלוקות אבסטרקטיות

מחלקה אבסטרקטית יכולה לרשף ממחלוקת אבסטרקטית אחרת.
במקרה שמחלקה אבסטרקטית יורשת ממחלוקת אבסטרקטית אחרת, היא לא נדרשת למשם את המתודות האבסטרקטיות של מחלוקת האב, למחרת שהיא יכולה אם היא רוצה.
מחלקה קונקרטית שמרחיבה מחלוקת אבסטרקטית חייבת לישם את כל השיטות המופשנות שעברו בירושא. במקרה אחריות: מחלוקת-נכד, ככלומר מחלוקת אחרת שתירש את מחלוקת הבן, נדרש שהיא מיימוש לכל ה-API מעלה.

שימוש במחלקה אבסטרקטית

המצבים בהם השתמש במחלקה אבסטרקטית:

1. **במצב שבו אין היגיון להגדיר אובייקט קונקרטי מהמחלקה הזאת.** למשל אמרנו שאם אנחנו מסתכלים על המחלקה האבסטרקטית Animal, אין כל כך שימושות להגדיר Animal כללי בלי בסוג שלו.
2. **אנחנו רוצים לכפות איזשהו API על סט של מחלקות.** כלומר אמרנו שככל החיות אנחנו רוצים להכיריה אותן להיות מסוגלות לדבר. אז הדרך לעשות את זה זה לשימוש במחלקה האב מתודת speak. ואז מי שרצה לבנות מחלקה שיורשת מ-Animal חייב למשוך את ה-API הזה.

פולימורפיזם של ירושה וממשקים

casting & types

interfaces - אינט חלק מההיררכיה של מחלקות. מה שמאפיין את הירושה זה שכל מחלקה יכולה לקבל רשות רק מחלוקת אחת אחרת. בממשקים המצביע הוא שונה, וכל מחלוקת יכולה להחליט למשוך כמה ממשקים שהוא רוצה. כתוצאה לכך, לכל מחלוקת ב-Java יכול להיות מספר גדול מאוד של type שונים. בעצם על כל מחלוקת אפשר להסתמך בכמה דרכים:

- ה-type שלה עצמה.
- ה-types של המחלוקת (או סט המחלוקות) שממנה היא יורשת.
- סט הממשקים שהוא מממשת.

ניתן לבצע casting לכל אחד מה-types.

דוגמאות:

```
public class MyParentClass implements Bable {....}  
public class MyClass extends MyParentClass implements Printable,  
Cloneable {....}
```

למופיע של המחלוקת MyClass יש types 6: MyClass, MyParentClass, Bable, Printable, Clonable, Object

שימוש ממושך לעומת ירושה ממחלוקת

שניהם מקיימים יחס-is/so, אבל שימוש מוצדק בירושה מצריך תנאי חזק יותר לפיו לא מספיק ש-A הוא סוג של B, כי אם המאפיין העיקרי של A הוא שהוא B ("A is first and foremost a B").

כל אצביע: אם זה יכול להיות ממושך (ולא מחלוקת אבסטרקטית), נגדיר את זה כממושך. ככל ברහמה הבחירה בין ירושה ממחלוקת למימוש ממושך, נעדיף ממושך. בהמשך נדבר על הסיבות לכך, אבל לפני כן נראה דוגמאות.

דוגמאות מספרית Java

ממשקים: ממשקים נפוצים מאוד ב-Java. דוגמה קלאסית היא חבילת **Collections** של Java, שהיא בעצם חבילת מבני הנתונים של Java.

Collection, הוא data structure כללי, יש לו כל מיני מתודות שהוא מגדר כמו למשל add ו-remove ו-size ו-get by index (גודל מבנה הנתונים). יש גם ממשקים יותר ספציפיים ב-Collection כמו למשל List שיורש מהמחלקה Collection, מייצג מבנה נתונים שיש בו מקום על אינדקס ולכן יש לו מתודות כמו get ו-set, set by index.

הסיבה ש-Collections גם ממשקים ולא מחלוקות, היא שהממשקים של Collection הרבה פעמים מדברים יותר על המה ולא על האיך.

בדוגמה של הממשק List: יש הרבה מאד דרכים למשרשרת רשימה - רשימה מוססת מערך וכו'.

לכל האינטראפיסים האלה יש את אותו API, המתכוונים של java החליטו שיותר הגיוני להגיד את ה-Collections האלה בתור אינטראפיסים - כמובן מושגים ולא מחלקות שאפשר לרשף מהן.

מחלקות אבסטרקטיות: דוגמה המחלקה Number.

זו מחלקה המייצגת איזהו מספר כללי. יש לה כל מיני מתודות שימושיות כמו intValue,intValue, floatValue, זה בעצם אומר לייצג את המספר בפורמט של integer, בפורמט של מספרשלם/עשרות. יש כל מיני מחלקות שירושות מ-Number כמו Integer ו-Double.

שיטות ממומשות בממשק

שיטות ברירת מחדל (default)

עד היום למדנו שמשתוקים מתקדים בג'אווה רק בתור חוזה. הם רק מצינים איפה שיטות יהיו למחלקה שתתמשצ אונם, וכל מחלקה שמממשת את הממשק חייבת לכתוב עצמה את התוכן של כל שיטה שמוצינת בממשק המקורי.

החל מגרסת 8 של ג'אווה, יש אפשרות חדשה בממשקים שיטת ברירת מחדל או שיטה דיפולטית, והיא לרובת הגדהמה (!) מאפשרת לנו לכתוב מימוש לשיטה כבר בממשק עצמו.

שיטות דיפולטיות כתובים באמצעות המילה השמורה default ואז אפשר לכתוב את המימוש הדיפולטי.

```
interface ExampleInterface {
    default void exampleMethod() {
        int i = 0;
        i++;
        ...
    }
}
```

מחלקה שמממשת את הממשק יכולה, אבל לא חייבת, לדרס את השיטה. מבון הטכני זה בדיק כmo שיטה במחלקה אב במקורה של ירושה.

סיבות לכך ששיטות דיפולטיות הן שימושיות:

1. לצורך עדכונים עתידיים (תמכה לאחר מכן). יכול מאוד להיות שכתבנו פעם איזה ממשק ובעקבותיו כל מיני מחלקות שונות ומשנות מממשות אותו, אבל אחרי כמה זמן אנחנו מחליטים שצריך לעדכן את הממשק ולהוסיף לו שיטה. אם הינו פשוט מוסיפים שיטה אבסטרקטית, כל מחלקה שמממשת את הממשק תצטרכ להוסיף אותה מימוש לשיטה. במצב זה אנחנו אומרים שההוספה של השיטה החדש האבסטרקטית היא לא Backwards Compatible, היא לא תומכת לאחר. אך במקומות להוסיף שיטה אבסטרקטית, נוסיף שיטה דיפולטית.
2. שיטות נוחות שימושות בשיטות האבסטרקטיות. ככלור אם אנחנו מעריכים שהרבה אנשים ירצו להשתמש בשיטות האבסטרקטיות של הממשק בדרך מסוימת ספציפית, אז אפשר בעזרת העמסה להוסיף שיטה דיפולטית שתעשה את זה.
3. כדי לאפשר ליותר ממשקים להיות "משקים פונקציונליים": ממשקים שיש להם שיטה אבסטרקטית אחת בלבד. (למה זה טוב? ולמה זה נקרא ממשק פונקציונלי? דבר על זה כשנדבר על למבודוט).

דוגמאות:

```
interface Playable {
    void play();
    default void repeat(int times) {
        for (int i = 0; i < times; i++)
            play();
    }
}

interface Renderable {
    void render(
        Vector2 position, double angle);
    default void render(Vector2 position) {
        render(position, 0);
    }
}
```

שיטות פרטיות

המטרה של שיטות פרטיות היא להיות פונקציות עזר לשיטות הדיפולטיות, במקרה שיש צורך
הן שיטות שחויפות רק בסkop של הממשק עצמו, כלומר המחלקות הממשות לא יכולות להשתמש בהן
בדיוק כמו שיטות פרטיות במחלקות.
לדוגמא: לצורך מיחזור קוד בין שתי שיטות דיפולטיות במשק

```
interface IntefaceExample {
    public int abstractMethod();
    default int defMethod1() {
        res = privateMethod(1);
        return res;
    }
    default int defMethod2() {
        res = privateMethod(2);
        return res;
    }
    private int privateMethod(int num) {
        //shared code...
    }
}
```

עקרונות מגבלים למימוש שיטות במשק

בשימוש הטהור ביותר של משק אנחנו כן נרצה להשאיר אותו כמה שיותר נקי מושיטות עם מימוש, ולהרכיב
אותו מושיטות אבסטרקטיות, אבל ראיינו גם כמה מצבים שבהם בהחלט אפשר לנצל את יכולות של שיטות
דיפולטיות ואפילו שיטות פרטיות:

כשהה מוסיף נוחות, או כשזאת הדרך היחידה שלא להרים את המחלקות שכבר ממשות אותנו.
מה שמאוד לא מומלץ לעשות עם שיטות דיפולטיות זה להוסיף כל מיני יכולות חדשות לגמרי שבכל לא
קשריות לשיטות האבסטרקטיות, פשוט כדי לשתף אותן עם המחלקות הממשות.
זה שימוש שכבר אפשר לקבל לירושא ויש סיבה טובה שמשקים הם ממשקים ומחלקות הן מחלקות:
מחלקות נועדו בשביל המימוש, וממשקים נועדו לקחת את המימוש ולבנות לו אבסטרקטיה לרמת החוזה.

חוּרְנוֹנוֹת שֶׁל יִרְשָׁה

ירשה היא כל', לא עקרון שמנחה אותנו. בגלל שהיא מאוד נוחה, צריך לשים לב שלא עושים לה "ניצול לרעה", מפרים אנקופולציה ומסבכים את הקוד.

בשימוש יתר בירשה, למחילה ירושת יש הרבה איברים - כל מה שהיא צברה מכל הטיפוסים שמעליה. כל להסתברך כשייש לה הרבה שיטות שלא היא עצמה הגדרה בתוך הקובץ שלה. הקוד מפוזר על פני כל העץ במקומם להיות מאוגד לאריזות הגיוניות.

צריך לשמור על עצ' ירושה ברור ופשוט, כך שייהי קל לעקוב אחר הלוגיקה ואחר המתוודות שכל מחלוקת ממשת.

ובקצרה, **חוּרְנוֹנוֹת שֶׁל שִׁימוֹשׁ יִתְרַח בֵּרְשָׁה:**

- שרשות ירושה ארוכות עלות לייצר API מנופח ומסורבל במחלקות הסופיות.
- לאובייקט מחלוקת סופית יהיו הרבה שיטות הפרוסות על מספר מחלקות שונות.
- קוד שקשה להתמצאו בו.

לסיום: אנחנו מעוניינים בעיצוב מונחה נוחה לקפסולות אוטומות לפי תחומי אחריות, ולא בעיצוב מונחה ירושה 😊

עיצוב תוכנה: הכללה ומיחזור קוד

ירשה לעומת הכללה

דרך נוספת למיחזור קוד בלבד ירושה - **Object composition**. כלומר להכיל אובייקט אחר, להחזיק `instance` של האובייקט הזה ולהשתמש במתודות שלו.

לדוגמה:

```
public class B {  
    public void foo() { ... }  
}  
  
public class A {  
    private B b;  
    public A(B b) {  
        this.b = b;  
    }  
    public anotherFoo(...) {  
        ...  
        // A uses the foo() code by calling b.foo()  
        this.b.foo();  
        ...  
    }  
}
```

הכללה	ירשה	
מודגר באופן דינמי בזמן ריצה - לא סטטי, נתונים גמישות תוך כדי זמן ריצה.	פונקציית <code>foo()</code> מושעת בירשה.	יתרונות פולימורפיים - ניתן להשתמש במתודה ללא תלות בסוג הספציפי של תת המחלוקת.
מייקוד - כל מחלוקת ממוקדת במשימה אחת, ותת משימות שלה ימומשו בחלוקת אחרת. ריבוי עצמים - יכולת להכין מספר עצמים.	מודגרת באופן סטטי - לא משתנה בזמן ריצה (ב吐וח יותר).	חוּרְנוֹנוֹת חסור דינמיות.

דוגמאות לירשה למול הכללה [ב讼טו](#).

הכוונה לבירה בהכללה על פני ירושה:

1. מיקוד - לחשב האם צריך שתת המחלקה תירש את כל מחלקה האב או שטיפיך שימוש במשהו ספציפי.
2. לא צריך ירושה כדי לחלק קוד בין קבצים - בניית מחלקה חדשה על מנת לחלק קוד.
3. היחס "a-to" - אם הוא לא מתקיים, לא משתמש בירושה.

עיקנון **Composition over Inheritance**: הכללה על פני ירושה. אם ניתן, במקום לרשת מחלקה, נכיל מופע שלה באופן שומר על ה-API.

תבנית עיצוב אסטרטגיה

נדבר על התבנית עיצוב חדשה ששימשתה בהכללה: אסטרטגיה (**Strategy**).
היא אחת מעמודי התווים, ובמה שירצה מדווק אסטרטגיה היא כלי מרכזי בתכנות מונחה עצמים מודרני.
בבאונו לכתוב קוד כללי, שיכל לשרת צרכים רבים.
Strategy הוא מצב בו למערכת שלנו יש משפחה של אלגוריתמים, התנהגוויות או אוסף דברים שעוסקים בבחירה התנהגוויות מסוימת של המערכת.
הינו רצים להפריד את האוסף הזה מהמערכת עצמה, על מנת שייהי פשוט להחליף בין ההתנהגוויות השונות (אפיו בזמן ריצה).
כל עוד ההתנהגוויות הן בעלות API זהה, יכולות לקבל אותו קלט ולהוציא אותו פלט, לא באמצעות חarov
לנו גם לדעת מה ההתנהגוויות הספציפית שאנו משמשים בה בכל רגע נתון. הלוגיקה של האסטרטגיה היא לא חלק אינטגרלי מהלוגיקה של המחלקה.
דוגמה: אלגוריתם מיון בעל סוגים שונים.

מימוש:

נדיר API (מחלקה אבסטרקטית או אינטראפיט) שמנדר מה האלגוריתם או ההתנהגוות עשויה.
כל אחד מהויריאנטים, לכל אחת מההתנהגוויות השונות תהיה מחלקה שתשתמש את ה-API זהה.
את הבחירה בין סוגי ההתנהגוויות ניתן לעשות באמצעות **.factory**.

יתרונות: קוד מודולרי, המורכבות שקופה למחלקה הראשית, קוד קל להרחבה ולשינויים, הפרדה בין קוד ה"לקוח" (מחלקה הראשית) למימוש סוג הספציפי של האלגוריתם, יכולת לבצע שינויים בזמן ריצה (למשל כתוצאה מקבלת קלט של המשתמש).

דוגמה:

<pre>public class SomeCollection { private Comparable[] contents; private SortStrategy sorter; public SomeCollection() { this.sorter = SortStrategyFactory.select(...); } public void sortContents() { this.sorter.sort(this.contents); } }</pre>	<pre>public interface SortStrategy { void sort(Comparable[] data); } public class QuickSort implements SortStrategy { public void sort(Comparable[] data) {...} } public class MergeSort implements SortStrategy { public void sort(Comparable[] data) {...} } public class SortStrategyFactory { public static SortStrategy select(...) {...} }</pre>
---	---

אסטרטגיה לעומת ירושא

עדיפות בימוש באמצעות אסטרטגייה: (במוקם למשתמש את קונספט האסטרטגיה בירושות מחלוקת כללית)

1. שימוש נכון בירושא - תנאי ה-"a-is" לא מתקיים במקרה של אסטרטגיה, ולכן לא ניתן בו. (לדוגמה QuickSort , MergeSort , Collection)
2. מודולריות - לשםימוש של אלגוריתם שנדרצה שיتمש בחלוקת מסוימת במקומות נפרדים עם API בלבד.
3. הסתרת מידע - מסתיר את השימוש של האלגוריתמים השונים שאנו יכולים להשתמש בהם, למשתמש בחלוקת אין צורך לדעת בבדיקה איך אסטרטגיה מסוימת מתחממת, רק על הקיום שלה.
4. שינוי בזמן ריצה - לעומת הסיטיות של ירושא כאן שינוי בזמן ריצה אפשרי.
5. מיחזור קוד - ניתן להשתמש באסטרטגיות במגוון מחלוקות ולא רק בחלוקת הספציפית שליהם.

המשך Bricker (תרגיל 2)

אסטרטגיה

בסרטונים הראשונים נראה שימוש של אסטרטגיה, לא אפרט.

במקרה הנוכחי, האסטרטגיה MovementStrategy סבבה סיבוב שיטה אחת שמקבלת GameObject ומחזירה Vector2. שום דבר בתחריר זה לא כובל אותה להיות רק אסטרטגיה לתנועה, או למעשה לאסטרטגיה כלל! די היה לנו במשחק כללי יותר שמכיל שיטה אחת עם אותה רשימה פרמטרים וערך החזרה.

במשך נראה איך אפשר היה לחסוך את הצורך המשך בהגדרת המשך באמצעות ממשקים פונקציונליים. בנוסף, נראה איך אפשר היה לחסוך גם את המחלוקות שemmמשות את המשך, באמצעות ביטוי למבדא! כשאסטרטגיה לא תחייב טיפוסים חדשים כלל, השימוש בתבנית העיצוב יפוך לכמעט מוביל, והצריך בירושא יפחח עוד יותר.

Javadoc

הkonvensia לתייעוד מחלוקות, ממשקים ושיטות.

על מנת לפתח היררכיה שנתמכת בDocumentation כתוב `**/`ENTER``, ועוד יש השלמה אוטומטית למשתנים ולערך החזרה.

ניתן ליצור מהתייעוד קובץ HTML מרכיב: `.tools -> generate Javadoc`

חלק 6 - עוד סוגים API

static final

המילה השמורה **final**

- משתנה **final** מוגדר ומואטחל בקונסטרקטור או ב declaration.
- המילה **final** מבטיחה שלא ניתן יהיה לשנות את הערך לאחר השמה ראשונית.
- פרימיטיביים - לא ניתנים לשינוי.
- רפרנסים - המצביע עצמו לא ניתן לשינוי, אבל השודות ניתנים לשינוי.
- בד"כ **private** - המטרה היא להגן על שינויים בתוך המחלקה.

דוגמאות מהתרגום:

```
public class FinalStaticExample {  
    /* final fields */  
    //    private final int a;  
    private final int a=0;  
    public final int[] arr = {1, 2, 3, 4, 5};  
  
    /* static fields & methods */  
    public static int b;  
    public static void staticPrintExample() {  
        System.out.println("Static methods are cool");  
        //        System.out.println(a);  
        System.out.println(b);  
    }  
  
    /* normal fields & methods */  
    public int c;  
  
    public void regularMethod() {  
    }  
}
```

מחלקות ושיטות **final**:

- שימושות המילה **final** ב- **public final class** היא שלא ניתן לרשות מהמחלקה.
- באופן דומה, שיטה שמוגדרת כ-**final** לא ניתנת לדרישה על ידי מחלקת הבת.

המילה השמורה **static**

- מגדיר משתנה או מתודה של מחלקה. (**class variable**)
- המשמעות של משתנה/מתודה סטטית היא שלכל המופעים של המחלקה נוצר מופיע אחד בזיכרון.
- שלכל המחלקות יש גישה אליו.
- שדה סטטי הוא שדה שלא מייצג תוכנה של המופיע.
- מתודות סטטיות לא יכולות לקרוא למתודות שאין סטטיות.
- נועד להימנע מיצירת משתנים סטטיים שאינם **final**.
- נועד להגדיר מתודה סטטית במקרה בו המתודה לא ניגשת לשודות/מתודות אחרים.

```

class FinalStaticMain {

    public static void staticMethod() {}

    public void regularMethod() {}

    public static void main(String[] args) {
        FinalStaticExample example = new FinalStaticExample();
        //           example.a = 5;
        //           example.arr = new int[5];
        example.arr[0] = 10;

        System.out.println(example.b);
        System.out.println(FinalStaticExample.b);
        FinalStaticExample.staticPrintExample();

        staticMethod();
        //           regularMethod();
        example.regularMethod();
    }
}

```

השילוב static final

משתנה static (שאינו מיטבייל) מגדיר משתנים קבועים שרלוונטיים לכל מופע של המחלקה. חיב לקל את הערך שלו בשורת הרצה. נועד לא משתנים. נועד להגדיר משתנים כאלו במרקם הרלוונטיים, כך נודע שהם לא משתנים. כל אצבע: **תמיד** נשאף לפחות את כמות ה **Moving Parts** בתוכנית, כלומר את החלקים שניתן לשנות לא לצורך. لكن איפה שניתן נרצה להגדיר משתנה **final**, ובפרט משתנה סטטי **static final**.

מיוטבליות משותפת

"אנחנו רוצים להימנע ממיטבליות משותפת מאד" - אם יש לנו שדה מיטבייל, נרצה שהוא יהיה כמו שפותחות משותף (כלומר נעדיף שהיא פרט). בזמן ששדה רגיל זמני מכל השיטות של העצם, שדה סטטי נגיש מכל השיטות של כל העצמים. לעקב אחריו השינויים שלו, לוודא שהוא תמיד עם ערך הגיוני וחוקי, ולצפות את ההשפעות שלו על הקוד, זה כבר קשה יותר. لكن נימנע ככל האפשר משדות סטטיים מיטבליים. שדה סטטי שימושי לכל המופעים אפשר להחליף מאוד במקרה אחרת שימושי לקבוצה של המופעים.

שיטות סטטיות

שיטות שלא משויות לאף עצם. הן עדין שיטות, מבוסן שהן משויות למחלקה, אבל הן לא מושפעות מ- **instance** של המחלקה, כלומר לא משויות למופעים אלא למחלקה. אז איך ואיפה נמוקם שיטה סטטית? בהתאם לעקרון ה-**discoverability**, כלומר נשים את השיטה במקום שבו היינו מחפשים אותה. **לדוגמה:** במקרה זהה נניח ש-**Person** היא המחלקה שהכי קשורה לטעות זהות. אז אני מגדירה:

```

public static boolean isValidId
    שמתקבל פרטיטן id לבדיקה.

```

קריאה לשיטה סטטית

- בטור המחלקה: עם שם השיטה עצמה. לדוגמה: בטור `Person` אפשר להשתמש פשוט בקריאה `isValid(...)`
- מוחוץ למחלקה: אם השיטה הסטטית פומבית ורוצים לקרוא לה מוחוץ למחלקה, עושים זאת דרך שם המחלקה. לדוגמה:
`Person.isValid(...)`
- לשיטה סטטית אין גישה ל-`this`, כי היא לא מקושרת למופע מסוים.
- בפרט אין לה גישה לאיברים רגילים, לא סטטיים (אברי מופע).

שימושים בשיטה סטטית

- **בטור ממשך:** שיטה סטטית במשך היא כמו כל שיטה סטטית אחרת עם מיושן, ואין לה ממש קשור לתפקיד המקורי של משך בפולימורפיים. משך הרוי מייצג חוצה שבא לידי ביטוי בשיטות האבסטרקטיות שלו.
- השיטה הסטטית שמופיעה בו כנראה קשורה באיזשהו אופן למופעים שהוא מייצג, אבל היא לא חלק מהחוצה של המשך. (שימוש פחות נפוץ, אבל קיים).
- **פונקציות פומביות** שלא שייכות לפחות עצם מסוים.
- שיטה **פומבית** שמייצרת מופע של המחלקה שהיא נמצאת בה ("שיטה מפעל", שיכולה להיות תחליף לבנייה. זה שימוש נפוץ). לדוגמה: השיטה `String.join`, שmarsחרשת כמה מחרוזות למחרוזת אחת בהינתן איזושהי מחרוזות שצריך לשים בינהן.
- נעדיף שיטה צזו במקום העמסה של בניאים, כי זה מאפשר API יותר נוח וניתן לשנות בקלות את שם השיטה.
- שיטות עזר **פרטיות** שלא נוגעות למצב עצמו, כמו למשל לא נוגעות לשדות. (שימוש נפוץ) מוטיבציה לשימוש זהה, הרוי יכולנו לкопל את הקוד גם בלי להגדיר את השיטה סטטית: כבר דיברנו על זה שסדרות מהוים מיutableיות משותפת בטור המחלקה, והמיutableיות הזה משותפת לכל השיטות הרגילים, שיטות המופע. הגדרת השיטה כ static עוזרת להבהיר לקרוא שהשיטה הזאת לא נוגעת לשדות ולא יכולה לגעת לשדות. אך אם ניתן, נגיד שיטה בטור צזו שלא נוגעת לשדות. אך אם final עוזר להקטין את מספר החלקים הנעים במכונה שלנו, אך שיטות שאפשר להגדיר כשיטות סטטיות עוזרות לצמצם את המרכיב שבו החלקים שכן נעים יכולים לנوع.
- שימוש בשיטות סטטיות **פרטיות** כדי עזר לשיטות סטטיות פומביות.
- לדוגמה: `main`, שהוא שיטה סטטית פומבית. היא חייבת להיות שיטה סטטית, כי היא נקראת בזמן שבו עוד אין בכלל עצמים בתוכנית. אם רוצים לחלק אותה לפונקציות, כי היא מאוד ארוכה למשל, אז חייבים לעשות זאת זה עם שיטות סטטיות אחרות, כי שיטה סטטית כמו `main` הרוי לא יכול לקרוא לשיטות מופע (שיטות שאינן סטטיות).

לסיכום, שיטות סטטיות הן כל'י שלא תופס מקום מאד מרכז'י בתכונות מונחה עצמים, כי הן לא משתתפות בכלל המשחקן של עצמים שקוראים לעצמים. עם זאת, יש להן שימוש.

איברים שמשמעותם יכול להזכיר (recap)

- שיטות ללא מיושן (שבמחלקה הממשת תהינה פומבית).
- שיטות ברירת מחדל (default methods), עם מיושן.
- שיטות פרטיות (עם מיושן) - שיטות עזר ששיטות ברירת המחדל יכולות לגשת אליהן.
- שיטות סטטיות (עם מיושן).
- שדות סטטיים סופיים.

מחלקה Utility

מחלקה שמחזיקה אוסף של מתודות סטטיות, בלי האפשרות או הצורך ליצור אובייקטים חדשים מהמחלקה. דוגמאות:

- Math - מחלקה של אופרציות מתמטיות, עם כל מיני מתודות כלליות כמו סינוס, לוג, אקספוננט וכו'.
- המתודות הללו לא קשורות לאובייקט ספציפי, ולכן אין היגיון בעשות אותן מתודות מופע (instance).
- Arrays - מספקת מגוון שיטות עזר שפועלות על מערכים.

import java.util.[package name]; // import command

Shadowing

הצללה (Shadowing) היא שימוש במחלקה יורשת במשתנה public בעל שם זהה למשתנה במחלקה האב, או מתודה static public בעלי שם זהה למתחודה במחלקה האב. זה מימוש שנרצה להימנע ממנו, כי הוא מבלב ומהווה מימוש לא נכון בפולימורפיזם. בפולימורפיזם, ה-type reference קובע מה מותר לנו להריץ וה-type Object קובע לנו מה ירצו. העניין הזה לא מתקיים בצורה דומה בקשר ל data members ומתחודות סטטיות.

דוגמה: נגידיר את המחלקות הבאות

<pre>public class A { public int myInt = 1; public static void staticFoo() { System.out.println("A"); } }</pre>	<pre>public class B extends A { public int myInt = 2; public static void staticFoo() { System.out.println("B"); } }</pre>
---	---

הגדירה של המתודה staticFoo היא לא Overriding, אלא היא נקראת staticFoo, וכך גם המשתנה .myInt

<pre>A a = new B(); System.out.println(a.myInt); a.staticFoo();</pre>

עת, אם נריץ את הקוד הבא:

יודפס:

1

A

בגישה ל a.myInt. אנחנו מסתכלים על אובייקט מסווג B דרך החלון של רפרנס מסווג A, אך ירצו ה-myInt של ה-type reference, יודפס 1.

בגישה Foo.a.staticFoo. גם פה זה יהיה תלוי רק ב-type reference, יודפס A. (בנוסף, זו דרך לא נכונה לקרוא למתחודה, כי עדיף שנקרא ל - Foo.a.staticFoo).

הסיבה: מתחודה סטטית לא קשורה לאובייקט ספציפי, אלא קשורה לכל המחלקה, ולכן Java קוראת למתחודה של הרפרנס (כי זה מה שהוא מכיר).

לכן כשאנו מסתכלים על a שהגדכנו, כל מה שאנחנו יודעים עליו שהוא רפרנס מסווג A, למרות שככיכול יצרנו אותו בתור B.

לסיכום:

- ה-data members נקבעים על ידי ה-type reference, והתוכן של המתודות נקבע על ידי ה-object.
- Shadowing זה משחו שהוא לא מומלץ ומבבל, ונרצה להימנע ממנו.
- אם שיטה מייצגת משך שיש לנו עניין בכך שהיא פולימורפי (שתהינה עוד צורות של מימוש עבורה), השיטה חייבת להיות שיטה מופע.

Façade

משמעותו של השם: חזית (של בניין).

המטריצה: לחת מרכיבת ולהסתכל אליה דרך חלון צר או דרך איזשהו אינטראפישן קטן וויתר פשוט.

נרצה להשתמש בתבנית זו במקרים בהם יש לנו מערכת גדולה עם API מרכיב שקשה לעבוד איתנו, כשבנוסף יש למערכת הרבה ליקוחות שלא צריכים לדעת את כל המרכיבות של המחלקה, אלא מספיק להם לדעת חלק מאוד קטן منها או איזשהו API פשוט יותר (בנוסף ה-API הפשט יכול עליהם ויהפוך את המערכת לאטרקטיבית יותר).

הפיתרון: ניקח את המערכת המורכבת ונבנה מחלקה חדשה שנקראה לה `Facade`, שהיא תבנה איזשהו API יותר פשוט עבור הלוקה. המחלקה עצמה תעשה את העבודה המולכנת של להזכיר את כל הקשרים, את

הדרך המורכבת לעבוד עם ה-API המקורי, והיא תספק לנו איזושהי דרך פשוטה יותר לעבוד איתנו. כתת הליקוחות יכולים לבצע את אותה פעולה שהם רצו לבצע קודם, כאשר עשינו בשביבם את הפעולות המורכבות של התמודד עם ה-API הקשה.

כלומר ה-`Facade` ממומש כך שהוא לוקח את המערכת הגדולה ומשתמש ב-`API` המסובך, והפלט שלו זה ה-`API` הפשט.

יתרונות:

- הליקוחות חסופים ל-`API` מינימלי ופשוט לשימוש
- הליקוחות לא מודעים לשינויים מאחוריו הקליעים, ולפעמים אפילו לשינויים ב-`API` שמאחוריו הקיימים (כל עוד מחלוקת `Facade` יודעת לתמוך בהם)
- אנחנו לא מבדים את הפיצרים המקוריים - ה-`API` של מחלוקת `Facade` אינם בהכרח תחליף ה-`API` המקורי של המחלוקות המרכיביות אותן, ולכן ישן מקרים בהם ניתן לעשות שימוש ב-`API` של המחלוקות המקוריות, ככל שיש בכך צורך.
- יכול לספק את הפונקציונליות המופשטת כמיומש של ממש בעל `API` פשוט.

דוגמה:

ה-`API` המקורי:

```
public abstract class Item { ... }

public class Pasta extends Item { ... }

public class Salt extends Item { ... }

public class SaltShaker {
    public Salt salt(int nTSpoons) { ... }
}
```

```
public class Pantry {
    public Item get(String item) { ... }
}

public class Pot {
    public void boil(int nLiters) { ... }

    public void add(Item item) { ... }
}
```

מחלקה ה-Facade שתאפשר לנו להציג פסטה בקומות:

```
/** Chef class that implements the Façade Design Pattern */
public class Chef {
    private SaltShaker saltShaker;
    private Pantry pantry;
    private Pot pot;
    public Chef () {
        this.saltShaker = new SaltShaker();
        this.pantry = new Pantry();
        this.pot = new Pot();
    }
}
```

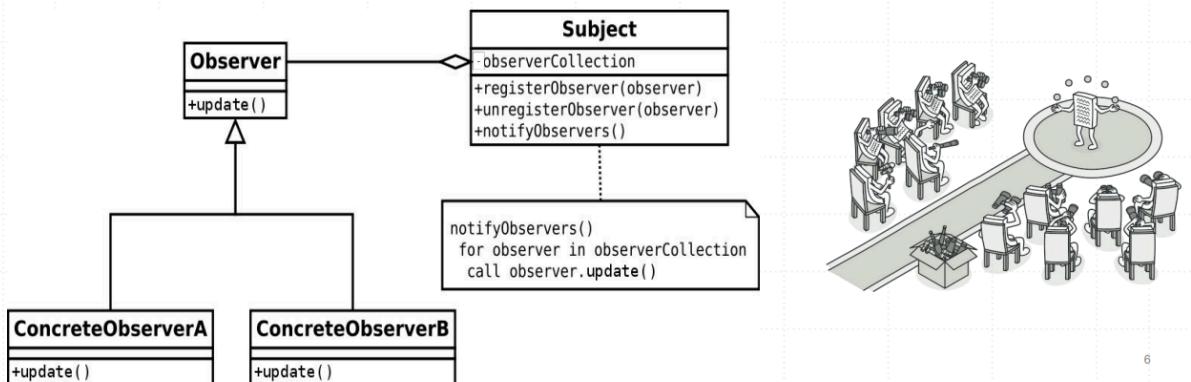
```
public void makePasta() {
    pot.boil(2);
    Item pasta = pantry.get("pasta");
    pot.add(pasta);
    Salt salt = saltShaker.get(1);
    pot.add(salt);
    ...
}
```

Observer

ניהול מעקב אחר עדכון - במקום שהרכיב ישאל בלי הפסקה האם יש עדכון כלשהו, ה observer ישלח אקטיבית עדכון.

Observer

The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Observer Design Pattern – Participants



Subject:

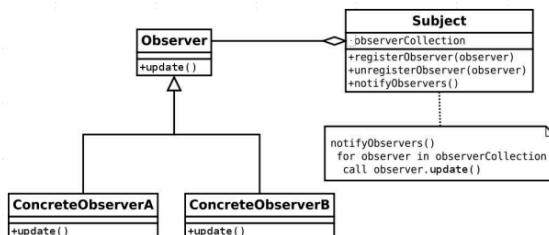
- Has a list of Observer objects.
- Can add or remove Observer objects from the list.
- Notifies the observers registered in the list when the event of interest has accrued.

Observer:

- Defines an updating interface for objects that should be notified of changes in a subject.

ConcreteObserver:

- Implements the Observer updating interface to be notified by the subject when the event of interest has accrued.



Observer Example - DanoGameLab

```
public class GameManager {  
    .  
    .  
    private GameObjectCollection gameObjects;  
    .  
    .  
  
    public void update(float deltaTime) {  
        //update all objects and look for collisions.  
        for(GameObject obj : gameObjects)  
            obj.update(deltaTime);  
        if(camera != null)  
            camera.update(deltaTime);  
        gameObjects.update(deltaTime);  
        gameObjects.handleCollisions();  
    }  
}
```

```
public class GameObject {  
    .  
    .  
    private ModifiableList<Component> components;  
    .  
    .  
  
    public void update(float deltaTime) {  
        transform.update(deltaTime);  
        transform.setAccelerationEnabled(true);  
        renderer.update(deltaTime);  
        if(components != null) {  
            for (var component : components)  
                component.update(deltaTime);  
            components.flushChanges();  
        }  
    }  
}
```

Singleton

- הסינגלטון הוא תבנית עיצוב שמגבילה את המחלקה ליצור מופע יחיד ובודד.
- שימוש התבנית חייב לעמוד בשני עקרונות: מופע יחיד של המחלקה, וגישה גלובלית.
- זה שימושי כאשר יש צורך לבדוק אובייקט אחד לביצוע פעולות במערכת, או שמספר מופעים יפגעו בלוגיקה של התוכנית.
- הבנאי יהיה פרטי והוא יחזיק רפרנס לאובייקט singleton
- תהיה给她 גלובלית לsingleton
- דוגמאות: יצירת מחלקה שמקשרת ל- DB, אובייקט logger

דוגמה בסיסית:

```
package Singleton;  
  
class Singleton {  
  
    private static Singleton singleton;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (singleton != null) {  
            return singleton;  
        }  
        singleton = new Singleton();  
        return singleton;  
    }  
}  
  
public class BasicSingleton {  
    public static void main(String[] args) {  
        Singleton singleton = new Singleton(); // ERROR! private constructor  
        Singleton singleton1 = Singleton.getInstance(); // will return the single object created  
        Singleton singleton2 = Singleton.getInstance(); // these two objects are equal  
    }  
}
```

בහינתן כל מה שדיברנו עליו, אולי ישמע מפתיע ששדה מיטביי סטטי פומבי יהיה דפוס תכונתי נפוץ, אבל "תבנית העיצוב" Singleton מתייחסת בדיק לזה. Singleton הוא התבנית שבה מגדירים עצם מיטביי שהוא זמין בכל מקום בתכונה. הנה למשל הדרך הפשוטה ביותר להגדיר Singleton:

```
class MutableClass { ...  
public static final MutableClass SINGLETON = new  
MutableClass();  
  
... }
```

כעת, מכל מקום בקוד, ניתן לכתוב `MutableClass.SINGLETON` ולגשת לשיטות של העצם.

השם Singleton מגיע מכך שגם גם המופיע היחיד של המחלקה: בדרך כלל המתכונטים יחסמו את האפשרות להגדיר מופעים אחרים של המחלקה. הסיגלטןזכה לפרסום כתבנית-עיצוב למקרים בהם נראה שיש צורך רק במופיע אחד של המחלקה, ורוצים להקל על הגישה אליו. כיום Singleton נחשב אנט-תבנית-עיצוב (קרי: דפוס תכונתי נפוץ, אבל לא לפתרון בעיות כי אם ליצירה שלהן) משומש לעיתים רבות ליצור הגבלות שלא לצורך, במצבים בהם אין צורך ממשי במופיע יחיד.

חלק 7 - מבני נתונים

בפרק זה נכיר כמה מבני הנתונים המוכרים ביותר: List, Set ו-Map (רשימה, קבוצה, ומפה/מילון). אלו מבני נתונים שלא ייחודיים לגאווה ונמצאים כמעט בכל שפת תכנות.

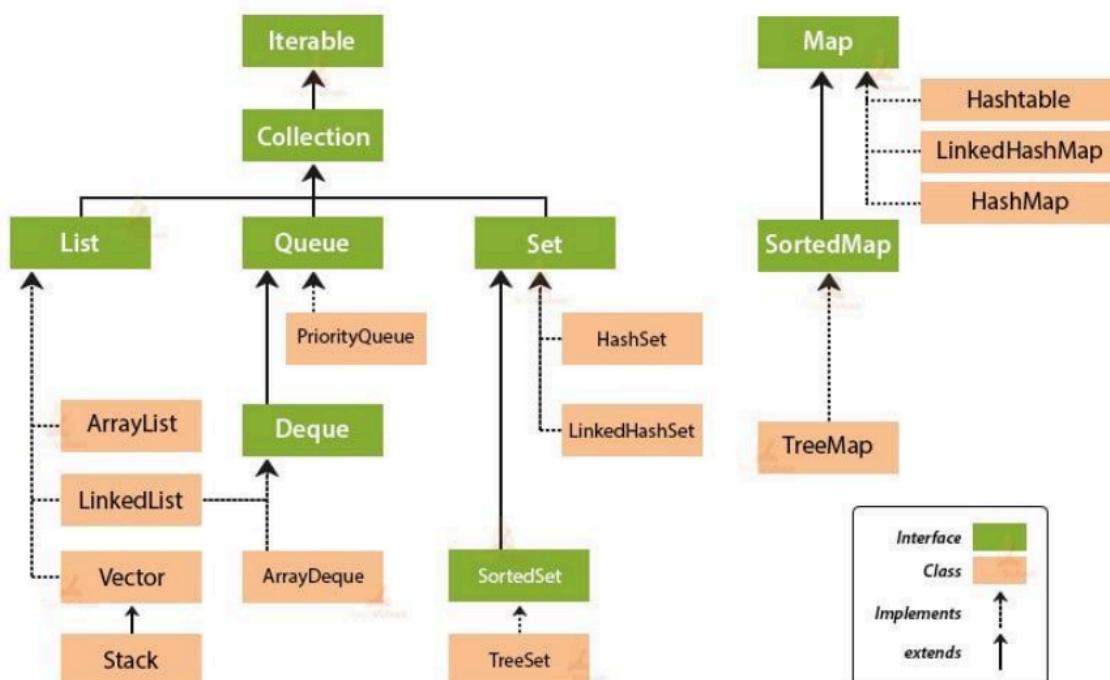
נדבר על שני נושאים מובאים: שלושת מבני הנתונים ברמה האבסטרקטית - כלומר רק מהם מייצגים אצלנו, ולאילו תרחישים מתאימים כל אחד (בכללי, לא ספציפית לJava).

לאחר מכן נכיר מעוף הציפור את הממשקים והמחלקות הרכזונטיים של גאווה.

Collection Framework

אוסף מחלקות ש- Java מספקת, על פיה ממשקים מוגדרים מראש. געסוק בעיקר בממשקים List, Set, Map ובמימושיהם שלהם. הממשק Collection מייצג אוסף - מבנה שאפשר להוסיף לו איברים, להסרר איברים, לשאל אם איבר קיים, וכמה איברים יש באוסף הזה. בהתאם לכך, Collection מכיל 4 שיטות: add, remove, contains, size. הממשקים List ו-Set בעצם יורשים מהממשק הכללי יותר Collection ומקבלים את ארבע השיטות האלה ממנו, כך שהממשק List מוסיף רק עוד שתי שיטות אבסטרקטיות, ו-Set לא מוסיף שיטות בכלל.

Collection Framework Hierarchy in Java



רשימה List

- רשימה שהאיברים בה הם בעלי סדר מוגדר (ראשון, שני, שלישי...). ניתן לגשת לאיברים ע"פ אינדקס, וכן גם לשנות אותם. אין הגבלה על חזרה של איברים. רשימה מיוצגת בગ'ואה על ידי הממשק List. מכיוון שהוא משתק, הוא לא עוסק בכך שומרם את האיברים בזיכרון, אלא רק בעולות שהרשימה מספקת למשתמש, כמו למשל לטעמם.

משתמע מכך שלא ניתן לקומפל את השורה

```
List list = new List();
```

שיטות (מתודות) לדוגמה:

- add - מוסיף איבר חדש לרשימה
- remove - מסירה איבר מהרשימה
- contains - בודקת האם האיבר קיים
- size - מחזירה את הגודל של הרשימה
- get - מקבלת אינדקס ומחזירה את האיבר באינדקס זהה
- set - משנה את האיבר באינדקס מסוים

קבוצה Set

- מודל את הקונספט של קבוצה מתמטית. רצף של איברים שאין להם סדר, ובלי כפליות (כל איבר יכול להופיע בקבוצה פעמי אחת בלבד).

מיוצגת בગ'ואה על ידי הממשק Set. גם Set כולל add, remove, contains, и size.

מפה Map

- (מפה/מילון) מייצג מבנה נתונים שמאפשר בין מפתחות לערכים. המפתחות חייבים להיות ייחודיים (מפתח יכול לקבל פעם אחד בלבד). ממשק Map לא משתמש ב借口 Collection, אלא זה משתק נפרד. אובייקט של map ימפה כולל key ו-value.

שיטות לדוגמה:

- put - מקבלת key ו-value ומוסיפה את הזוג למפה
- remove - מקבלת מפתח ומסירה אותו (אות הערך שלו) מהמפה
- containsKey - בודקת האם יש במפה זוג עם ה-key שהולחים
- containsValue - שואלת האם יש זוג עם ה-value הנתון.
- get - מקבלת מפתח ומחזירה את ה-value.
- size - מחזירה כמה זוגות יש במפה.
- keySet - מחזירה את קבוצת כל המפתחות. השיטה זו מחזירה Set, כי במפתחות אין כפליות.
- values - מחזירה את כל הערכים של המפה. מחזירה רפנסו מסוג Collection כי יתכן שיש כפליות.

שימושים של מבני נתונים

Map, Set, List הם רק מושגים, ככלור אלה רק חווים שגדירים מה יהיה צריך מבנה נתונים אמיתי לkiem. המחלקה הקונקרטית היא זו שקובעת איך החזזה זהה ממושך מאחורי הקלעים. נesson בימושים הבאים:

. **LinkedList** ו- **ArrayList** - List

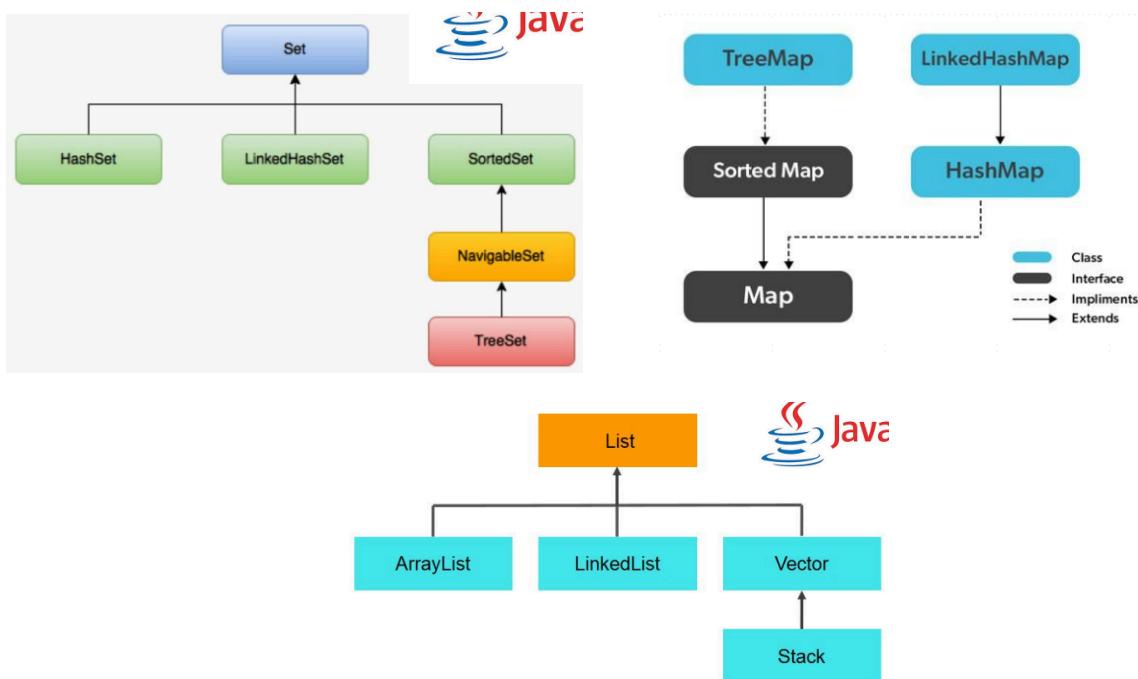
. **HashSet** - Set

. **HashMap** - Map

כל הטיפוסים האלה הם חלק ממה שמכונה ה-**Collections Framework** של ג'אווה, שהוא הרבה יותר רחב מהחלק שנesson בו בקורס.

כמה מהטיפוסים שלא ניכנס אליהם:

- Collection בעצמו ירוש מעוד ממשק בשם Iterable, שמייצג שהוא כל כך כללי שאינו אפשר אפילו להגיד כמה איברים יש בו (אין לו שיטת size). כל מה שהוא מבטיח זה שאפשר לעשות עליו איטרציה עם לולאת for each.
 - כמובן שהוא אומר שאפשר לעשות איטרציה גם על List Collection ועל Set ו-**Collection** וכל המימושים שלהם.
 - ל-**Collection** יש עוד ממשק ליד בשם Queue, שמייצג תור, שתומך בפעולות של הכנסת אחד ויציאה מהצד השני.
- יש כמובן עוד ממשקים ומימושים, אבל אלו שציינו בתחילת הפרק הם המרכזיים ביותר.



ArrayList

שימוש של List המבוסס על מערך. המחלקה מחזיקה מערך בגודל התחלתי, וכאש נגמר המקום, יוצרת מערך חדש גדול יותר בזיכרון ומעתיקה אליו את כל האיברים מהמערך הישן. יש גישה מאוד נוחה לאיברים לפי אינדקס מכיוון שהוא מערך. זו אלטרנטיבה טובה לשימוש במערכות פרימיטיביים. ניתן לקרוא עוד ב[תיעוד הרשמי](#).

LinkedList

שימוש של List שהוא רשימה מקוشرת. מורכבת מאובייקטים נפרדים המצביעים אחד לשני. למחלקה יש מצביע רק לתחילת הרשימה ולבסוף הרשימה. הגישה לאיבר לפי אינדקס היא מעבר על כל האיברים מתחילה הרשימה עד האיבר הרצוי. שימוש כזה מתאים לשימוש בסיסי עבור מבנים נתונים אחרים כמו מחסנית. ניתן לקרוא עוד ב[תיעוד הרשמי](#).

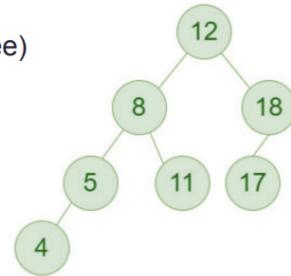
HashSet

שימוש של הממשק Set המבוסס על טבלת גיבוב (hash-table). כל איבר מוכנס לטבלה לפי ערך הגיבוב שלו ואין כפילות של איברים. כל הפעולות, הכנסה, חיפוש ומחיקה של איברים הן יעילות בגלל הגישה היררכית למקום בטבלה המתאים לערך הגיבוב. לאנגנון יעיל לטיפול בהתנשויות וגם במקרה של של איברים עם אותו ערך גיבוב היעילות נשמרת.

הערה: לכל אובייקט במאף יש שימוש של הפקציה hashCode במחלקה Object ובמחלקה HashSet ואפשר גם לדרש את הפקציה הזאת כמו שנראה בהמשך.
ניתן לקרוא עוד ב[תיעוד הרשמי](#).

TreeSet

- Implements NavigableSet (that extends Set<E>)
- Implemented using a balanced binary search tree (similar to AVL tree)
- The same expected runtime for add, remove and find - log(n)
- Can be iterated efficiently (ascending or descending order)
- In contrast to hashset:
 - requires less memory
 - has predictable runtime
 - Has more functionality, like search closest value from below/above.



HashMap

שימוש של הממשק Map המבוסס גם הוא על טבלת גיבוב של זוגות. ערך הגיבוב של כל זוג של key-value, נקבע ע"י ערך הגיבוב של ה-key.
ניתן לקרוא עוד ב[תיעוד הרשמי](#).

TreeMap

דומה לו TreeSet (כasher TreeSet is a facade of TreeMap), מממש NavigableMap<K,V> (that extends Map<K,V>)

יש לו יותר פונקציונליות מאשר HashMap (לדוגמה מетодת floorKey - חיפוש המפתח הקרוב ביותר מלמטה)

סיבוכיות של collections

זמן ריצה:

	Insert	Find	Remove	Access by index	implements
ArrayList	$O(1)^*$	$O(n)$	$O(n)$	$O(1)$	List
LinkedList	$O(1)$	$O(n)$	$O(n)$	$O(n)$	List
HashSet	$O(1)^*$	$O(1)^*$	$O(1)^*$	-	Set
TreeSet	$\log(n)$	$\log(n)$	$\log(n)$	-	NavigableSet
HashMap	$O(1)^*$	$O(1)^*$	$O(1)^*$	-	Map
TreeMap	$\log(n)$	$\log(n)$	$\log(n)$	-	NavigableMap

צריכת מקום:

על פניו, ב-ArrayList יש ברוב המקרים הקצאה של מקומות מיותרים במערך, אבל בפועל ההקצאה זהה מתגמדת ביחס לצריכת המקום של רשימה מקוشرת.
אם תחשבו על כך, כל צומת ברשימה מקושותה דורש לא רק רפנס לעצם, אלא גם מצביעים קדימה ואחוריה ברשימה. די בהם בשבייל לקבוע שעלות המקום של רשימה מקושותה תמיד גבוהה יותר, אפילו כמספר האיברים המיותרים ב-ArrayList הוא במקסימום. בפועל, כל צומת צריך לפחות עוד יותר מקום משולשה רפנסים בלבד. צריכת מקום גבוהה יותר מיתרגמת גם לתקנות זמן גבוהות יותר של ניהול זיכרון.

בחירה אובייקט

בבחירה סוג אובייקט, השיקולים שנחנו אותו:

- פונקציונליות - אובייקט שישרת את המטרות שלנו
- תחזוקה
- זמן ריצה

hashCode

כפי שŁמְדָנוּ, כל עצם ב-`Java` יורש בעצמו או טרנסיטיבית מהמחלקה `Object`, וכן יורש את השיטות שלה. אם הוא רוצה הוא דורך אותן ומחליף בימוש משלו.

כבר הכרנו שתי שיטות של `Object`:

- `toString` שמחזירה ייצוג מחרוזתי של העצם,
- `equals` שמקבלת עצם אחר ומחזירה האם הוא "שׁוֹלֵךְ" לעצם זהה לפי איזושה הגדרת שיקולות של המחלקה.

כעת נדבר על עוד שיטה של `Object`, `hashCode`.

התפקיד המרכזי של השיטה `HashCode`: מבני נתונים יכולים להשתמש בה כדי לקבל ערך מסוים שמייצג את העצם. מימוש ברירת המחדל של `Object` עבור `hashCode` הוא להשתמש בכתובת של העצם בזיכרון.

שימוש ב `contains`

אם משתמשים במתודת `contains` על טבלת גיבוב עם עצם מסוים, המתודה תקרא ל-`hashCode` של העצם, את ה-`hashCode` תמיר לאינדקס בטבלה, תלך לתא באינדקס ותראה שבאמת יש שם עצם כלשהו. זה לא אומר שהעצם שנחננו מוחפשים, שכן המתודה תקרא למאתודה `equals`, ואם היא מוחזירה `true` נגיע למסקנה שהעצם אכן נמצא בטבלה.

מימוש השיטה `hashCode`

מתהיל מיציאת עצם בטבלה שתיארנו, נסיק שאם שני עצמים שהם שקולים לפי המימוש של `equals`, אז ה-`hashCode` שלהם צריך להיות זהה, כי אנחנו רוצים ששתי רשומות זהות ימופו לאותו מקום.

לכן אם דורסים את `equals`, חיברים לדרכו גם את `hashCode`!

בנוסף, המימוש החדש של `hashCode` צריך להתבסס על אותן שדות כמו `equals`. אם להרבה עצמים שונים יש את אותו `hashCode`, זה גורם להאטה בביטויים של טבלת הגיבוב. ג'אווה פתרה לנו את השאלה של איך להגדיר `hashCode` ייחודי עבור צירוף של שדות. ישנה מחלוקת Utility בשם `Objects` עם השיטה `hash`, שמקבלת כמה שדות שתרכזו ועשו בדיק את זה.

אם לא דرسנו את `equals`, זאת אומרת ש מבחיננו שני עצמים של המחלקה שקולים רק אם הם ממש אותו עצם בזיכרון. ובמקרה זה, גם את `hashCode` לא צריך לדרכו. במילים אחרות, מימוש ברירת המחדל של `equals` ומימוש ברירת המחדל של `hashCode` כן משחקים יפה אחד עם השני.

hashCode()

- One of the `Object` class methods;
- Returns an integer value, generated by a hashing algorithm.
- Objects that are equal (according to their `equals()`) must return the same hash code.
- This method is supported for the benefit of hash tables such as those provided by `HashMap` and `HashSet`.

```
class Student {  
    private final long id;  
    private String name;  
    private String email;  
  
    @Override  
    public boolean equals(Object obj) {  
        if(!(obj instanceof Student))  
            return false;  
        Student other=(Student) obj;  
        return (other.id==this.id)  
            && other.name.equals(this.name)  
            && other.email.equals(this.email));  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(id, name, email);  
    }  
}
```

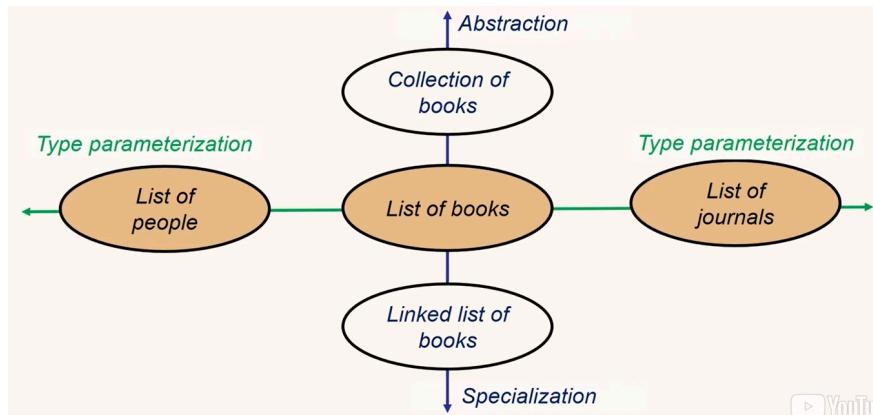
נשים לב שב `equals` אנחנו (כמעט תמיד) עושים - זה נכון כי זו הדרך היחידה למשמש את הפונקציה.

Generics

בג'אווה מבני הנתונים, פרט למערכים, לא יכולים לשמר פרימיטיבים, רק עצמים!
 לכן בג'אווה ישן מחלקות Wrapper (או מחלקות עוטפות) לפרימיטיבים: המחלקה Integer העוטפת את הפרימיטיב int, המחלקה Double העוטפת את הפרימיטיב double וכו'.

עטיפה, בהקשר זה, משמעה פשוט שלמחלקה Integer יש שדה מסטיפוס int.
 כל המחלקות הללו הן immutable, למשל Integer(new) הוא עצם שעוטף עד סוף חייו את הערך 4 (ובשביל לעוטף מספר אחר יצרים עצם אחר).

מבנה הנתונים של ג'אווה לא יכולו אם כן לשירות את הפרימיטיב int, אלא ישמרו מופעים של המחלקה Integer.



از זה מדבר לא על רמה יותר מופשטת או ספציפית של מחלקות, אלא על מה מחזיקות המחלקות האלה.

API גנרי: בכל פעם שאנו יוצרים אובייקט של מחלקה generic, נגידיר את סוג האיברים שמבנה הנתונים מכיל, וכך כביכול ניצור מחלקה חדשה. לדוגמה:

- `LinkedList<String> myList = new LinkedList<String>();`
- `HashMap<String,Double> map = new HashMap<String,Double>();`

```
String s = myList.get(0);
map.put("hello", new Double(5.7));
```

```
// A list of strings
LinkedList<String> list = new LinkedList<String>();
list.add("hello");
String s = list.get(0);
list.add(new Double(3.14));      // Compilation error – list is a list of strings
Double d = list.get(0);          // Compilation error
```

Generics

- Used to create a single class, interface, or method that can be used with different types of objects.
- Generics are considered type-safe. Errors that occur from generics happen at compile time.

Node before Generics

```
LinkedList list = new LinkedList();  
  
list.add("hello");  
  
String s = (String) list.get(0);  
  
list.add(new Integer(5));  
String s = (String) list.get(1);  
// Run-time error
```

Node with Generics

```
LinkedList<String> list = new LinkedList<String>();  
  
list.add("hello");  
  
String s = list.get(0);  
  
list.add(new Integer(5));  
// Compilation error
```

עם Generics אנחנו מגדירים את הסוג הגרפי של האובייקטים שמבנה הנתונים יכול להכיל, אנחנו לא נשתמש בכמה סוגיים שונים של איברים באותה מחלוקת או באותו Collection, אלא נגדיר את הסוג המסוים שאנו משתמשים בו.

המטרה של Generics היא להיות מנגנון שmbatich - **type-safety** - התוכנית מכירה את הסוג שאנו אמורים להכין/לקבל מבנה הנתונים.

זה ימנע **type errors** - שגיאות שנובעות מהשמה בין טיפים לא מתאימים. בשימוש עם Generics נמיר את השגיאות האלה מזמן ריצה לקומPILEZA. בפועל, המנגנון מסייע במניעת casting-down שגוי.

יעילות השימוש במחלקות העוטפות

נניח שאנו רצimos מערך של פרימיטיבים. אם נשתמש במערך, יוכל ליצור מערך `int[]`, אבל מנגד בשביל להשתמש ב-ArrayList הטיפוס שלו יצטרך להיות Integer. במקרה זה המערך הפנימי בתוך ArrayList לא יהיה של ints אלא של מצביעים לעצמים מסוג Integer, ככל אחד מהם עוטף `int` יחיד. לכן במקרה של פרימיטיבים, מערך פשוט כן יעל יותר מ-ArrayList. אבל זה רלוונטי רק לפרימיטיבים ורק במקרים בהם:

1. הרבה מהקוד סבב בעיות על המערך.
2. כשייש בעית ביצועים.

כשהשימוש במערך ספורדי או כמשמעותו לטעות את הקוד, מבני הנתונים האלה יהיו שוקלים אפקטיבית מבחינה ביצועים, בזמן ש-ArrayList מספק יותר נוחות.

המרות

ניתן להמיר בין טיפ עוטף לפרימיטיבי, ולהפוך - Boxing - התהיליך של למתוך int ולייצר ממנה Integer. Unboxing - התהיליך של למתוך Integer מהתו ה-Wrapper class, מתוך ה-Integer את ה-int.

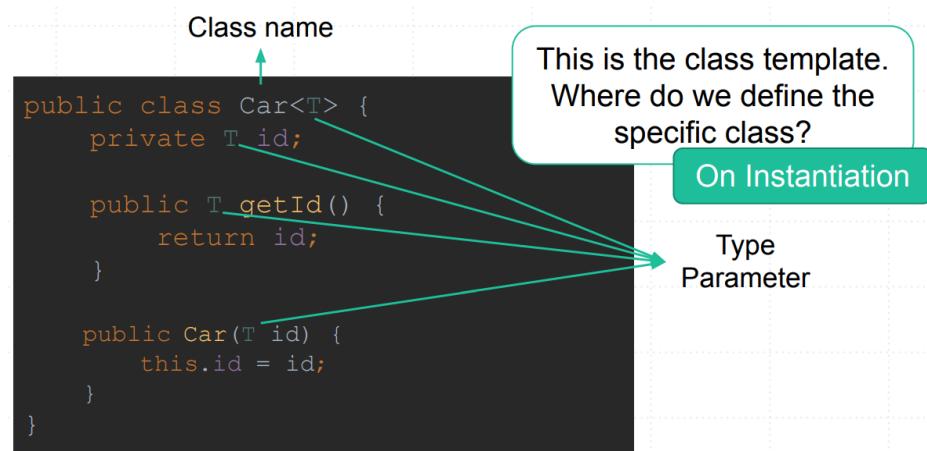
```
Iterable<Integer> numbers = ...;  
for(int num: numbers) { ... }
```

באוטומאטיותnumbers יכול להיות מסוג `[Integer]`.

לעומת זאת, נראה בהמשך שיש אינוריאנטיות בין מחלקות גנריות מסווגים שונים - אין בינהן יחס supertype/subtype ולכך לא ניתן להמיר מחלוקת אחת לאחרת (לא ניתן לעשות casting). לדוגמה, שימוש של אובייקטים היא לא מחלוקת-אב של רשימת מחזוזות.

יצירת מחלקה גנרייתית

(*** מתרגל 9)



זו צורה כללית של מחלקה ונitin לכתוב כל אות במקום T (זה רק מצין שמדובר בטיפוס גנריי במחלקה גנרטית). עם זאת, בכל מקום בו נכתב T וnochzor על האות שהגדנו, הכוונה היא שמדובר באותו type שקייבלו ממחלקה.

ביצירת מחלקה, ניתן להכניס כל משתנה בתור T, לא משנה מאייה סוג, גם בלי לציין את ה-type / העוטף שנשימוש בו:

Car is the raw type of the generic type Car<T>.

```
public static void main(String[] args) {  
    Car c1 = new Car(12345);  
    System.out.println((Integer)c1.getId() + 1);  
}
```

Compiler Warning:

Raw use of parameterized class 'Car'

כשהם מצינים type למחלקה מדובר ב-type raw (שכן היינו יכולים להציג Car<Integer>) זה שקול לכתיבה Object, שכן לא צינו את הטיפוס הגרפי. זה מתקמל אבל נחשב ל bad-practice ומכך, כי זה לא type-safe.

הפתרון: להשתמש ב-generic, זה מובן יותר ומאפשר לקוד שלנו להיות type-safe

```
public static void main(String[] args) {  
    Car<Integer> c1 = new Car<>(12345);  
    System.out.println(c1.getId() + 1);  
}
```

Returns an Integer instance

למעשה אנחנו מבצעים casting down מהתיפוס הכללי לסתמי, רק-cut ניתן להבטיח שהוא תקין.

מחלקה גנריית כמערך

לא ניתן ליצור מערך של מחלקה גנריית!

```
public static void main(String[] args) {  
    HashMap<String, Integer>[] map =  
        new HashMap<String, Integer>[5];  
}
```

Compilation error – Generic array creation.

במקום זאת נכניס את המחלקה לערך :Collection

```
List<HashMap<String, Integer>> listOfMaps =  
    new ArrayList<>();
```

נזכיר שכן ניתן ליצור מערך של טיפוסים גנריים:

```
Integer[] nums = {1, 2, 3, 4, 5}; // Initialize with array initializer
```

ירושה ממחלקה גנרטית

ניתן לרשת ממחלקה גנרטית:

```
class SubClass<T> extends SuperClass<T> {  
    // class body  
}
```

וכמו כן ניתן תור כדי כך להגדיר לה generic ספציפי וליצור מחלקה שהיא :non-generic

```
class SubClass<T> extends SuperClass<SomeType> {  
    // class body  
}
```

יחסים בין מחלקות גנריות, :type invariance

For any class (or Interface) myClass:

myClass<Type1> is neither a subtype nor a supertype of **myClass<Type2>**

For example: **LinkedList<String>** doesn't extend **LinkedList<Object>**

```
LinkedList<Object> myObjLst = new LinkedList<String>();  
-> compilation error!
```

Notice, you can still add **String** to **LinkedList<Object>**!

Upper Bound

בעזרת שימוש במילה השמורה **extends**, ניתן להגדיר שה raw-type שאנו מקבלים הוא שעל תוכנות ספציפיות:

<T extends S>

אנו מגדירים כך "חסם עליון" (upper bound) לטיפוס הגנרי T, אז ניתן להשתמש בתכונות ובمتודות של S.

דוגמא:

```

public class Car<T, D extends Iterable> {
    private final T id;
    private final D countries;

    public Car(T id, D countries) {
        this.id = id;
        this.countries = countries;
    }

    public void printCountries() {
        for (Object c: countries)
            System.out.println(c);
    }

    public static void main(String[] args) {
        Set<String> countries = new HashSet<>();
        countries.add("Israel");
        countries.add("UK");
        Car<Integer, Set<String>> car =
            new Car<>(54321, countries);
        car.printCountries();
    }
}

```

Wildcards

במחלקות המשתמשות בטיפוסים גנריים, אם לא יודעים או לא משנה לנו מה הטיפוס הגנרי, ניתן להשתמש בתו '?'-placeholder עבור הטיפוס הגנרי:

List<?> wildList;

במקרה זהה, לא ניתן לדעת שום דבר על הטיפוס שמבנה הנתונים מכיל, ונitin להוציא אליו רק null.
דוגמה:

```

public static void Count(Set<?> set) {
    int num = 0;
    for (Object o : set) {
        num++;
    }
    System.out.printf("Set has %d items", num);
}

public static void main(String[] args) {
    Set<String> countries = new HashSet<>();
    countries.add("Israel");
    countries.add("UK");
    Count(countries);
}

```

גם במקרה זה ניתן לעשות **upper-bound** עם טיפוס שנון כל לבצע פעולות על האיברים:

```
class WildCardUpperBound {  
    private static void askAnimalsToMakeSound(List<? extends Animal> list) {  
        for(Animal animal: list){  
            animal.makeSound();  
        }  
    }  
}
```

ניתן לאתחל רשימה **upper-bound** עם טיפוס ירוש מ-**Animal** (במקרה הזה **dog, cat** ירוש מ-**Animal**)

```
List<? extends Animal> list = new LinkedList<Dog>(); // Legal  
// Can only add null:  
list.add(new Animal()); // Illegal  
list.add(new Dog()); // Illegal  
list.add(null); // Legal
```

כמו כן גם במקרה זה ניתן להכניס רק null, שכן לא ניתן לדעת אם הטיפוס שנכנסו אכן יכול לעשות **upcast**, שכן?-? יכול להציג כל אחת מהמחלקות הירושות. בקבלה אובייקט, ניתן לעשות לו רק **upcast** לטיפוס של החסם העליון:

```
Animal a = list.get(0); // Legal  
Dog d = list.get(0); // Illegal
```

כמו כן ניתן להציג על **lower-bound** באמצעות המילה השמורה **:super**:

```
public static void addInteger(List<? super Integer> list) {}
```

In this example, the method may only receive a list that holds objects that are Integer or higher (Number is a super class of Integer).

We can be sure that there won't be Double, Float and the like in that list (as Integer is not a subtype of them).

ניתן לאתחל את המחלקה רק עם טיפוסים שבוגרים מהחסם התיכון. דוגמאות:

```
public static void addInteger(List<? super Integer> list) {  
    list.add(1);  
    System.out.println("number 1 added to list");  
}  
public static void main(String[] args) {  
    // Integer List is being passed  
    List<Integer> list1 = Arrays.asList(4, 5, 6, 7);  
    addInteger(list1);  
  
    // Number list is being passed  
    List<Number> list2 = Arrays.asList(4, 5, 6, 7);  
    addInteger(list2);  
}
```

- ניתן להוסיף לרשימה **Integers** או יורשים/IALIZEDים שלו (כלומר יותר גבוה פה, שכן יש המרה בינם)
- ניתן לדעת בוודאות שאין ברשימה **Double**, **Float** או כל מקביל שלהם (**Integer** is not a subtype of them)

```

class LowerBound {
    public static void addDogToList(List<? super Dog> list, Dog element) {
        list.add(element);
    }
    public static void main(String[] args) {
        List<Creature> creatures = new ArrayList<>();
        List<Animal> animals = new ArrayList<>();
        List<Animal> dogs = new ArrayList<>();
        Dog dog = new Dog("Buddy");
        addDogToList(creatures, dog);
        addDogToList(animals, dog);
        addDogToList(dogs, dog);
    }
}

```

סיכום: חסמים על wildcards

הכנסת איבר	שליפת איבר	תחול המחלקה עם	סוג ה-wildcard
רק null	רק Object	כל טיפוס	ללא חסם
רק null	מ-Object מטה, עד לטיפוס של החסם העליון T	T או יורש/משיר (subtype of T)	חסם עליון (extends T)
מכל יורש של T, מעלה עד לטיפוס של החסם התיכון T	רק Object	T או אב של T (super of T)	חסם תחתון (super T)

Erasure

במהלך הקומpileציה, java מוחקמת את כל הטיפים הגנריים, ומחליפה אותם ב-Object או בחסם העליון.
לכן אם אין חסם:

The diagram illustrates the Java erasure process. On the left, a generic `Node<T>` class is shown with private fields `T data` and `Node<T> next`, and a constructor that initializes both. A green arrow labeled "Erasure Process" points to the right, where the resulting non-generic `Node` class is shown. This class has private fields `Object data` and `Node next`, and a constructor that initializes them. It also overrides the `getData()` method to return `Object`.

```

public class Node<T> {
    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
}

public class Node {
    // PSUEDO COMPILER GENERATED CODE
    private Object data;
    private Node next;

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Object getData() { return data; }
}

```

וכאשר יש חסם עליון:

The diagram illustrates the Java erasure process for a class that extends a generic interface. On the left, a generic `Node<T extends Comparable<T>>` class is shown, which extends `Comparable<T>`. It has private fields `T data` and `Node<T> next`, and a constructor that initializes them. A green arrow labeled "Erasure Process" points to the right, where the resulting non-generic `Node` class is shown. This class has private fields `Comparable data` and `Node next`, and a constructor that initializes them. It also overrides the `Comparable.getData()` method to return `Comparable`.

```

public class Node<T extends Comparable<T>>{
    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
}

public class Node {
    private Comparable data;
    private Node next;

    public Node(Comparable data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Comparable getData() { return data; }
}

```

shawgias & Exceptions

סוגי שגיאות

מחלקות לסדרים (ע"פ היררכיה וקדימות בזיהוי):

1. **shawgias kompiltsia** - shgiasot shanchnu natkalim bahn bezman ctitat haqod, ci anchnu mpirim choki' tchbar.
2. **shawgias zaman ritsa** - shgiasot shakompilel la zihah, abel yesh azoshah tocna shmaza otun bezman ritsa.
דוגמאות:
 - a. Murekhet hahefula zihata shanchnu manosim leasot peula bli'rashot.
 - b. h-JVM shmarit at haocna shel zihah shnito legash le'atzem NULL au shnito leasot Downcast la choki'.
- (JVM - tocna makblat kalt kod bshfat bytecode moritzha otu ul'machshav)
- c. haocna shel zihata sheia uzma nishta legash laindok la choki' b'murak, ao sheia nishta legash la'kobz la'kayim.
3. **bagim yaduim** - shgiasot shanchnu matcnim yaduim upihen (l'morot shakompilel/tocna la zihah).
hcowna labag shel kritis at haocni, kolmer haocna ritsa, abel ushta dbrim la'ncovim.
htnagogot shel haocna shel haocno alia (ldog' nishta leasot chisob v'beutz chisoba la'ncov, ao shbaniyi berikr abel hdiskit nulmat).
4. **bagim la yaduim** - bagim shudin kiyimim batocna shel zihah lemorot shel zoh (b'shlavim koddim ao b'c'l).



כל סוג עדיף על הסוגים שמתוחתיו. ככל שלנו יש שליטה על זה, אנחנו מעדיפים לתפוא shgiasot coma shiutor gebava ba'irerchia zo.

giloi shgiasot v'dibog

(*** Chuk zo ha'eb maboa l'shogiasot be'olam haocna - mitan idu clili ul dibog, shcnraha cabr yadu l'cm) um lemire bin sogi shgiasot v'ululot ba'irerchia, matcnim mbcuim tssting - meritzim at haocna mspf gdol shel fuimim v'ul magon trchishim. Yesh hizuk rabh ma'od shel haocna shmetrtn le'azor um tstsing, como g'm cilim shmobinim batuk ha-IDE. Bperfet, matcniti g'avoja mstamshim b'clli b'shm zohul.

la'achr zihaii haocna yesh tahlir ni'piy shgiasot, ao bagin (nitn la'azor b'dibagur shel ha-IDE). Achat haocrim haklot v'hnpozot latar at haocm shvo haocna shgiasa, hia kodus cil lemire otoha shgiasa mahsder ha'shi, kolmer lohosi' l'kod b'dikot shigromo lo'zahot mzcivim la'rczim.

Zo ycolha lehoyt b'dika shbcl mkrha hiyta zricha lehoyt sh, l'mash v'idoa tkionot shel formetrin ao kalt. B'mkrim achrim zo ycolha la'ahar b'dika shanchnu ro'itim shtrutz rk b'zman dibagin.

Mstamshim b'miyut haocfa assert, shndbar ulia b'hamsar. Zo tchafif pesot v'ioter batoh la'adot, shlefumim mosipim b'kod cd'i lagrom lo'uzor lnu lmazoa at haocm shel shgiasa.

meuber munin zo ba' shlosot sogi shgiasot haocnanim al shgias kompiltsia, kolmer air gormim l'kompilel zihota yoter be'ut. az dz'er rasan, nitn hstcel ul azhorot haocnita, shan la'pumim hn la'fchot

אינפורטטיביות מהשגיאות. ככלא מסתכלים על האזהרות, הן בקלות הופכות להיות שגיאה מאחד הסוגים האחרים.

דרך אחרת, הרבה יותר חזקה להעביר שגיאות בזמן הידור היא לבחור בשפה סטטית. שפה סטטית היא שפה כמו ג'אווה שבה כל הישויות בין אם זה משתנים, מחלקות וכן הלאה, וגם היחסים בין הישויות, מוגדרים היטב בזמן כתיבת הקוד. שפה דינמית, לעומת זאת, היא שפה כמו פיתון או MATLAB, שבה כל המשתנים והמחלקות מוגדרים בזמן ריצה. בשפות סטטיות הקומpileר שומר עליון הרבה יותר במנני שגיאות, בעוד שפות דינמיות מבוססות על שגיאות מסדר שני ומעלה.

השלמת קוד חכמה - פיצ'ר שקיים ב-IDE. השימוש של השלמת קוד גדולה לאין שיעור כשמדבר בשפה סטטית.

מנגןון Generics הוא דוגמה מעולה למנגן שמטרתו להמיר חילך משגיאות זמן הריצה לכאלו שייעלו כבר בזמן הידור. בקוד הבא, למשל, נקבל שגיאת הידור בשורה השלישיית כבר בזמן כתיבת הקוד:

```
List<String> list = new ArrayList<>();
list.add("Hi!");
Double d = list.get(0);
```

במנגןון היישן יותר של ג'אווה, שבסיסו פולימורפיזם (List מחזיק מצביעים מסוג Object), הינו כתובים את הקוד הבא:

```
List list = new ArrayList(); //list of Objects
list.add("Hi!"); //upcast of String to Object
Double d = (Double) list.get(0); //downcasting the Object to Double
במקרה זהה שורה 3 הייתה מציפה רק בזמן ריצה, כשהיא מתגלת שההמרה לפני מטה בלתי חוקית.
```

הפקודה assert

assert (בעברית: "הנני טוענת כי...") היא פקודה שתפקידה לוודא הנחות שאנו חזו מנכחים על הקוד. מפתחים. מדובר בהנחות שאמרות להיות בהכרח נכונות, אבל אמר זה שם של דג, ואם הן לא מתקיימות, משתמש שיש באג בקוד שלנו.

דוגמה: נניח שאנו כותבים קוד שמקבל לוח של משחק איקס עיגול שהסתויים ובודק מי המנצח. כולם נכון, בשלב זה, נניח שהמשחק אכן הסתיים! אם זה במקרה לא נכון, בغالל באג, הקוד לא יפעל כראוי וכי ידוע מה יקרה בהמשך הריצה.

הפקודה assert מקבלת תנאי, ומקריסה את התכנית אם הוא לא מתקיים. למשל, נוכל לכתוב:

```
assert isGameOver(board);
```

לחילוף נוכל גם לכלול הודעה שגיאה אחריו נקודותים:
assert isGameOver(board) : "the game isn't over yet";

היתרון של assert על פni בדיקה רגילה, הוא שבבדיקות assert מorzoot רק אם אנחנו מצבים זאת מפורשות. אם נרצה קוד שמכיל assert באופן רגיל, הקוד יגיע לסופו אפילו שהתנאי ב-assert לא מתקיים. רק כאשר נוסיף את הדגל -ea ה-assert יפעל ויקרי את התכנית:

```
>>>
>>> javac Main.java
>>> java Main
completed successfully
>>> java -ea Main
Exception in thread "main" java.lang.AssertionError
at Main.main(Main.java:3)
```

```
public class Main {
    public static void main(String[] args) {
        assert false;
        System.out.println("completed successfully");
    }
}
```

assert עוזר להמיר באגים בקוד שלנו מסדר שלישי או רביעי לסדר שני.

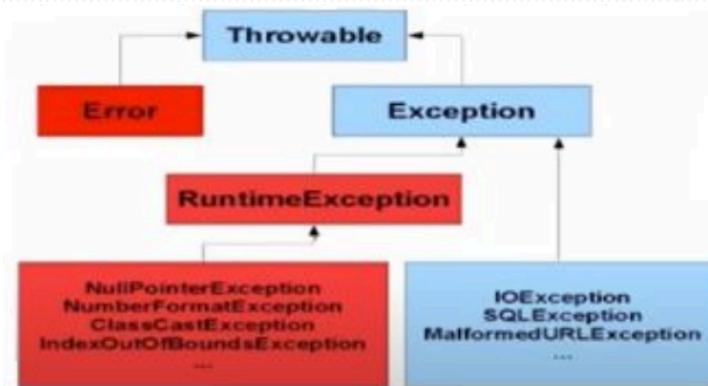
- השימוש ב-assert יכול לחסוך לנו זמן רב אם נדע לשימוש בו נכון. חשבו על המקרים הבאים:
- ישנם מקרים בהם אין כל שגיאת זמן ריצה, אלא התנאות לא רצiosa של התוכנית ואז קשה לאתר את מקור הbug. במקרים כאלה, ה-IDE הפשט לא יוכל לעזור לנו ונצרך לשימוש בכלים debug מתקדמים ואילו השימוש ב-assert הוא פשוט ויכול לחסוך את כאבי הראש האלו.
 - לפעמים, שגיאות זמן ריצה מתרחשות במקום אחר בקוד אולם מקור במוקם מוקדם הרבה יותר ולכן קשה מאוד לזהות אותן. השימוש ב-assert עוזר לנו לאתר את המוקם המדוייק בו השגיאה התרחשה.

זכרו: assert נבדלת לבדיקה רגילה בכך שהיא מיועדת לבדיקות שאנו מבצעים בזמן הדיבוג וכטיבת הקוד אבל שלא נרצה שתתרחשנה בזמן ריצה אצל המשתמש. למשל כי הבדיקה יקרה, ובמהלך הפיתוח כבר ידנו די הצורך שהתנאי תמיד מתקיים, או כי לתוצאות הבדיקה יש ערך עבורנו כמפתחים אבל אין לנו עניין שתקרויס את הריצה בשימוש אמיתי.

שגיאות נדרקות (Throwable)

יש 3 סוגים עיקריים של שגיאות כאלה:

- שגיאות (errors): אנחנו לא אמורים לטפל בהן, כי זו לא תחום האחריות שלנו (בדק' של ה-OS)
- : exceptions
- זמן-ריצה (unchecked): לא אמורים לקלוט, זהו התנאות לא תקינה של התוכנית שלנו וכאן בעיירנו גם לא אמורים לטפוש את השגיאה או לזרוק אותה (אלא לתקן את התוכנית)
- קלט/משתמש (checked): שגיאות כתוצאה מקלט לא תקין, לא ניתן להזות אותן מראש, אז علينا לטפל בשגיאות אלו בתוכנית שלנו (אפשר גם לגאלץ חזרה לשימוש, אבל בלי טיפול בתוכינה שלנו לא תפעל כראוי)



טיפול ב checked exceptions

כאשר יש שגיאה בתוכנית והמתודה שרצה כרגע לא יכולה להמשיך בגללה, המתודת תזרוק `Exception` חזרה למי שקרה לה. המתודה תעוצר איפה שזרקה השגיאה ולא תחזיר את הפלט המתוכנן שלה. `Exceptions` הם אובייקטים ב-`java`. בהמשך נגידר מחלקות מיוחדות שהן מחלקות של `Exceptions`. תוכנית טובה תתמודד עם השגיאות במקום הנכון ובצורה הנכונה.

נזרוק שגיאה במקומות בו השיטה הנוכחיית לא מתאימה לטפל בשגיאה, וצריכה לפעוף אותה הלאה כי היא לא בתחום אחוריותה (אם הקולט אינו תקין, מי שקרה לפונקציה צריכה צריך לטפל בכך). אם חלק מהגדרת האחוריות של השיטה היא לקבל ולטפל בקלט מהמשתמש, והמשתמש טעה, תיזרק שגיאה זהה בדיקת המוקם בו נרצה לתפוס את השגיאה ולטפל בה (לדוג' נבקש קלט חדש).

ע"מ לזרוק חריגה נשתמש במילה השמורה `throws`, שמנגדירה את החריגות שיכולה להיזרק מהשיטה:

תפישת שגיאה: בהתאם המתודה שקרה לשיטה שעלולה לזרוק exception נפתח בлок עם המילה השמורה `try`, ובתוכו נשים את הקריאה הבועיתית בפוטנציה. לאחר מכן אונחנו מוסיף בлок `catch` שתווסף את `Exception` שנזרק, אם אחד צה נזדקן, וטפל בו. נסיף `catch` שונה לכל היותר שעלול להיזרק:

- To catch an exception, we should surround the called method/operation with a try catch block.

```
try {
    // divide an integer by zero
    int i = 5 / 0;
} catch (ArithmaticException e) {
    System.out.printf("An exception occurred, %s\n", e);
}
```

- If needed, we can catch multiple types of exceptions

```
try {/* some code that might raise different exceptions*/}
catch (IOException eIO){/* handle IOException*/}
catch (ArithmaticException eA){/* handle ArithmaticException*/}
catch (Exception e){/* handle super type exception*/}
```

קיימות היררכיות של חריגות, ועלינו לתפoso את השגיאה המסוימת ביותר ע"מ שלא מתאפשר חריגות שלא התכוונו לטפל בהן (לדוג' חריגת זמן ריצה כשבכל רצינו לטפל בשגיאת O או קובץ לא קיים).
בנוסף, זה תורם לкриאות הקוד וכך ניתן ליצור פתרונות ספציפיים לעיבוט.
אם קיימות מספר חריגות אפשריות - נתפoso מהפרטית לכללית. ניתן גם ליצור חריגות משלנו.
במקרה בו יש היררכיה, כלומר מספר catch, נקבע איזו חריגה נזקקת ואז לבדוק בעץ החריגות היכן הפעბ הראשונה שהחריגה נתפסת. רק ה-**catch** שלבסוף תפso את החריגה יבצע טיפול בשגיאה (ירץ את הקוד בבלוג שלו).

```
try {/*trying to do something with files*/
catch (FileNotFoundException e){/**/}
catch (IOException e){/*escalating! but i know
how to treat it still*/}
catch (Exception e){/*unknown and unexpected*/}
```



דרך אחרת לטפל ב-Exception זה לא להכנסו אותו לבלוק try/catch, אלא להציגו בעצמו שאנו נזרקים את ה-Exception הזה. כמוור ניתן גם לגלל להלה את ה-exception עד שלב החוצה (למתוודה הקוראת).
עשיה זאת באמצעות המילה השמורה throws בפרקית הפונקציה.

לפיים:

הצד שuousה throw ל-Checked Exception, חייב להוסיף את ה-throws statement. הצד שקורא למетодה שזורקת Checked Exception חייב לשים בлок try/catch או לעשות throws בaczem.

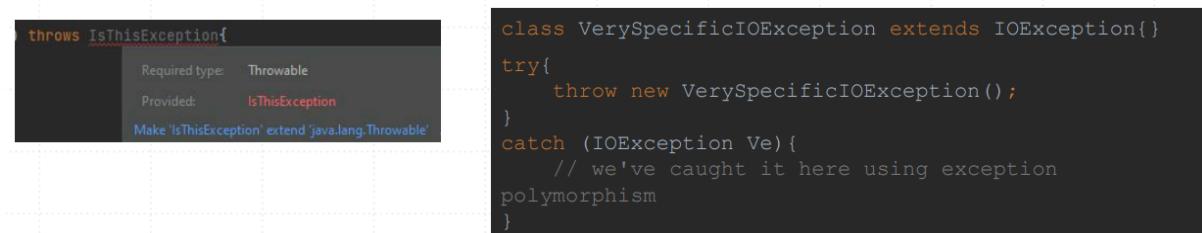
(אחרת נגרום לשגיאת קומפיילציה)
בזכיר שלמרות שניתן לעשות זאת גם עבור `unchecked exceptions`, לרוב לא נעשה זאת, אא"כ מדובר בשגיאות מעניינות לא-טריוויאליות. (לדוג' טיפול ב- `NullPointerException`)

מחלקות חריגה

כפי שציגנו, exceptions הן גם אובייקטים, ככלומר כמו מחלקות רגילות הן יורשות מ Object. אך ניתן להשתמש בפולימורפיזם ולרשת בין סוגים שונים של exceptions. צריך רק להקפיד שלא בסוף מתבצעת ירושה מהמחלקה Throwable כדי שזו אכן תיחשב כ exception.

ככלומר בפועל ניתן להגדיר Checked Exception חדש באופן הבא: נגיד מחלקה חדשה שעשויה extends Exception, או לחריגה מסוימת כלשהי (מחלקה שכבר ירושת מ-Exception). בדרך זו אפשר ליצור סוגים נוספים ספציפיים של שגיאות הקשורות לקרות.

דוגמאות:



יתרונות לשימוש ב-Exceptions

- קרייאת קוד: יש חלוקה ברורה יותר של הקוד, ניתן ליצור הפרדה בין החלק שמבצע את הפעולה הרגילה (בנהנה ואין שגיאה), לבין החלק שמטפל בשגיאות.
- אפשרות לפעוף שגיאות במעלה מחסנית הקריאה: באמצעות שימוש במילה השמורה throws בחיתמת מתודה, אנחנו מצהירים שאנו לא מטפלים בשגיאה אלא הקוד יעבור בצורה שקופה דרך המתודת. בצורה זו המתודת החיצונית לא תתעסק בלבבד בבדיקה האם המתוודה הפנימית הצליחה או לא, וזה יוצר קוד אלגנטית יותר.
- היכולת לאחד ביחיד סוגים דומים של חריגות או להפריד ביניהם במידת הצורך: כפי שראינו ניתן לרשות exception וליצור תת-סוגים של חריגות. ניתן לתபוע ולטפל בכל תת-שגיאה בנפרד, אך בנוסף מעיקרון הפולימורפיזם ניתן לתפוע יחד חריגות-בת באמצעות החריגת-אם שלhn (=מחלקת האב).

חריגות מ-main

כשה-main עצמו זורק exception אין מתודה בתוכנית שתתபוע את החריגה, כי מי שקרא ל-main זה המשמש שרירץ את התוכנה. لكن במקרה הזה החריגה נזרקת למשתמש כפלט-שגיאה של התוכנית, שבו מקבלים את שם ה-Exception.

הבדיל מהקרה בו נתפוע שגיאת קלט מהמשתמש, פונקציית main אינה מקבלת קלט מהמשתמש וכן אין זה מקומה לטפל בכך. הריצה דיפולטיבית הינה פתרון אפשרי, אך זהו אינה הדרך הנכונה לעשות זאת (בעזרת מגנון החריגות). لكن נסגור את התוכנית עם הודעת שגיאה, שכן לא ניתן להתקדם.

סיכום: תפיסת חריגות

כאשר קראנו לשיטה שעשויה לזרוק חריגות, נשאלת השאלה: לתפוע או לא?

כל האצבע הוא שתפועים רק אם אחד המקרים הבאים מתקיים:

1. אנחנו יכולים לטפל בחריגה במיקום זהה בתוכנית ולהמשיך ברכיצה.
2. ברור מעבר לכל ספק שהתוכנית לא יכולה להמשיך, ורוצים להפסיק את התוכנה בצורה שקטה עם הודעה אינפורטטיבית.
3. אם אנחנו רוצים להציג הלאה חריגה מסווג אחר.
4. אם אנחנו רוצים לפעוף, אבל לפני כן צריך להריץ קוד שמנקה את מה שעשינו בינהים.

אם אף אחד מآلה לא מתקיים, לתפוס זה רעיון רע כי לא יהיה לנו משהו מיוחד לעשות, אנחנו לא יכולים
לשים מועלינו (שאלוי עבורן אחד המקרים כן תופס) לתפוס.
בכל מקרה, אף פעם לא תופסים עם catch ריק (מטאטאים את החריגת מתחת לשטיח).

מחלקות מקווננות Nested Classes

מחלקה בתוך מחלקה. למחלקה המקווננת (הפנימית, מבוסנת הטכני ולא כפי שנגיד בorthand) יש גישה גם לרכיבים הפרטיים של המחלקה החיצונית.

```
class LinkedList {  
    Node head;  
  
    private static class Node {  
        int value;  
        Node next;  
    }  
}
```

יתרונות

- לוגית יותר נכון לקבץ מחלקות הקשורות אחת לשניה יחד, במיוחד כאשר רק המחלקה החיצונית היא זו שעושה שימוש במחלקה המקוונת.
- תורם לאנקפוסולציה:
 - מונע מהגדלת משתני `public` במחלקה החיצונית כשהכוונה היא שרק המקוונת תשתמש בהם, ולא מחלקות חיצונית אחרות.
 - מאפשר להסתר את המחלקה המקוונת ממחלקות חיצונית.
- תורם לקריאות הקוד - רכיבים משתמשים בהם יחד נמצאים באותו מקום (ולא בקבצים נפרדים)

סוגים של מחלקות מקווננות

- סטטיות
 - מחלקות מקווננות שמקשורות למחלקה הכללית ולא למופע ספציפי.
 - יכולות לגשת רק לשדות/מתודות סטטיות
 - דוגמה: `Node` (כפי שראינו לעלה)
- פנימיות
 - מקשורות למופע של המחלקה החיצונית ולא יכולות להתקיים בלי מופע כזה.
 - יכולה לגשת לכל הרכיבים של המחלקה החיצונית.
 - דוגמה: איטרטור בתוך מחסנית (כפי שנראה בorthand)

כל אזכור: כל מחלקה פנימית שיכולה להיות סטטית, נעשה אותה סטטית! (בדומה למתחוזות ושדות)

קריאה למחלקות מקווננות

דוגמיה: מחלקות מקווננות של המחלקה החיצונית stack

```
class Stack {  
    private Node head;  
  
    static class Node {  
        private int value;  
        private Node next;  
    }  
  
    void add(int val) {  
        Node newNode = new Node();  
        newNode.value = val;  
        newNode.next = head;  
        if(head == null) {  
            head = newNode;  
            return;  
        }  
        head = newNode;  
    }  
}
```

✓ Can only
access static
members of its
enclosing
instance

```
public class Iterator implements java.util.Iterator {  
  
    private Node current;  
  
    Iterator() {  
        current = head;  
    }  
  
    @Override  
    public boolean hasNext() {  
        return current != null && current.next != null;  
    }  
  
    @Override  
    public Object next() {  
        int value = current.value;  
        current = current.next;  
        return value;  
    }  
}
```

✓ Can access all
members of its
enclosing
instance

דוגמיה לסינטקס של קרייה למחלקות מקווננות: המחלקה Node היא סטטית, המחלקה Iterator היא פנימית

```
public class Nested {  
    public static void main(String[] args) {  
        Stack stack = new Stack();  
        Stack.Node node = new Stack.Node(); // can be created without outerclass instance  
        System.out.println();  
        for (int i = 0; i < 10; i++) {  
            stack.add(i);  
        }  
        // must be tied to an existing outer class instance! hence the special syntax:  
        for (Iterator iter = stack.new Iterator(); iter.hasNext(); ) {  
            System.out.println(iter.next());  
        }  
    }  
}
```

✓ Creates an instance of the enclosing class
in order to create inner class instances

Memento

תבנית שאחראית לניהול גרסאות - בפרט שמירה של מצבים ואפשרות לחזרה למצב שמור.

- Used for restore an object to its previous state.
- The memento pattern is implemented with three objects:

Originator

Some object with an internal state.

Implements:

1. `createMemento()`
2. `restore(memento)`

Memento

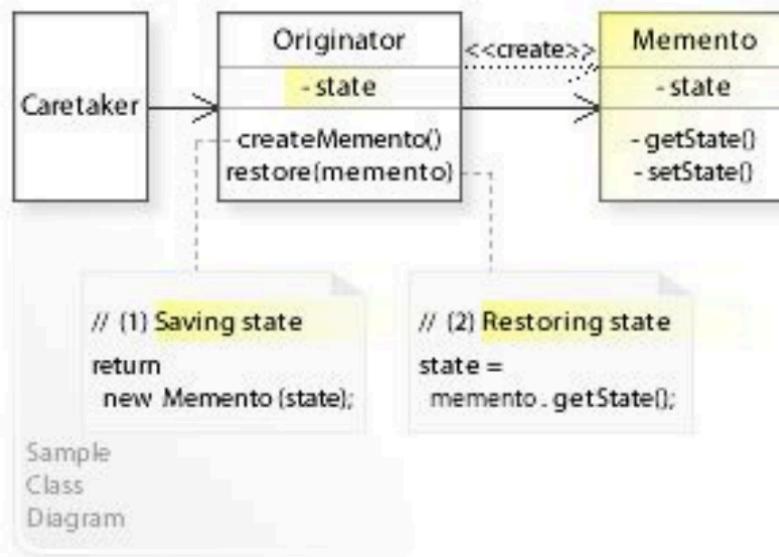
A POJO (Plain Old Java Object) that holds the state of some Originator instance.

CareTaker

A class that makes changes on the originator, but wants to be able to undo the change

Before modifying the Originator, The Caretaker asks it for a Memento object.

To restore to the state before the operations, it passes back the Memento object to the Originator.



דוגמה: מימוש מקומי של git (במעבדה 7)

Originator - CodeProject

Implements:

1. `createSnapshot()`
2. `restore(snapshot)`

Memento – Snapshot

CareTaker - Git

Implements:

1. `commit() : int`
2. `rollback(id: int)`

חלק 8 - Lambdas & Callbacks

מחלקות מקומיות ואנונימיות

דבר על דרכי שונות לארוז ולהעביר קוד ממקום למקום:

- לחת קוד ולשים אותו במשתנה
- לחת קוד ולשלוח אותו לפונקציה או לשיטה
- לחת קוד ולהציג אותו משיטה וכול'

התהילך זהה נקרא פרמטריזציה, כלומר לחת מההו שהוא קבוע hardcoded ולהכניס אותו בפרמטר.

מחלקה מקומית local class

מחלקה המוגדרת בתוך מתודה. (להבדיל ממחלקה מקוננת שמוגדרת בתוך מחלוקת) אפשר לראות אותה בתוך המתודה (scope) בכל מקום החל מרגע ההגדרה שלה, אבל לא ניתן לראות אותה מחוץ למתודה שהיא מוגדרת בה.

משתנים חיוניים שהמחלקה מreprsentת חייבים להיות **final** או **effectively final**. (נתיחה בהמשך בזמן הקומpileציה הקומpileייר למשה יוצר קובץ **class**. נפרד עבור המחלוקת הlokאלית. הן תורמות לאנקוסטציה, אבל לא מאד נפוצות).

דוגמה (מתרגול 9):

```
class Dictionary implements Iterable<TripletItem> {  
    private final ArrayList<TripletItem> dict =  
        new ArrayList<>();  
  
    void add(TripletItem item) {  
        dict.add(item);  
    }  
  
    public Iterator<TripletItem> iterator() {  
        /* local class */  
        class Iter implements Iterator<TripletItem> {  
            int index = 0;  
  
            public boolean hasNext() {  
                return index < dict.size();  
            }  
  
            public TripletItem next() {  
                return dict.get(index++);  
            }  
        }  
        return new Iter();  
    }  
}  
  
class TripletItem{  
    private final String name;  
    private final String value;  
    private final int id;  
  
    public TripletItem(String n, String v, int id) {  
    ... }  
    public String toString(){  
        return String.format("[%s, %s, %d]", name,  
value , id);  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Dictionary dict = new Dictionary();  
        dict.add(new TripletItem("#1", "A", 543));  
        dict.add(new TripletItem("#2", "B", 6542));  
        dict.add(new TripletItem("#3", "C", 875));  
        for (TripletItem tripletItem : dict) {  
            System.out.println(tripletItem);  
        }  
    }  
}
```

מחלקה אונומית

מחלקה אונומית זו מחלוקת מקוננת ללא שם.

בבת אחת מגדרים אותה, יוצרים אובייקט, ומכניםים את האובייקט לרפרנס - מבלי לציין את השם שלה.

כל מחלוקת אונומית תיצור רק באמצעות בנאי ריק, ולאחר ההגדרה ניתן להשתמש במחלוקת אונומית על ידי הרפרנס שיצרנו.

מחלקות כאלה הן שימושיות ונitin להיעזר בהן בעיקר באופן מקומי.

מחלקה אונומית תהיה מימושת (**implements an interface**) או יורשת (**extends an existing class**).

מחלקה אונומית שممמשת ממשק חייבת למשוך את כל המתודות שלו (בדיקות כמו בחלוקת מימושת ריגלה). ביצירת מחלוקת אונומית יורשת (מרחיבה מחלוקת) התחבר זיהה.

מבנה לכתיבה מחלקה אונומית:

```
interface SomeInterface {
    public void method();
}

class OuterClass {

    // defining anonymous class
    SomeInterface anonymousClass = new SomeInterface() {
        // body of the anonymous class
        @Override
        public void method() {
            // implementation
        }
    };
}
```

כasher התו ; (שהחץ מצביע עליו) מצין את סוף הביטוי (הגדרת המחלקה).

דוגמה למחלקה אונומית ממשית:

```
IntPredicate predicate = new IntPredicate() {
    @Override
    public boolean acceptNumber(int num) {
        return false;
    }
};
```

לא צינו לחלוטן את השם של המחלקה ואנחנו למשה עושים מההען Upcast.

כי מה שכתב כאן זה יוצרה של מופע חדש ממחלקה שמתממשת את IntPredicate והמופע הזה הוא של מחלקה, לא של הממשק. אי אפשר ליצור מופעים של ממשקים, אבל זה המופע של המחלקה ומכוון שכן IntPredicate כאן זה למשה לא הטיפוס של האובייקט, זה Upcast.
אי אפשר להשתמש במחלקות אונומיות בלי Upcast.

דוגמה למחלקה אונומית ירושת:

```
import java.util.*;

public class Main {

    public static void main(String[] args) {

        List<Integer> list = new ArrayList<Integer>() {
            @Override
            public int size() {
                return 0;
            }

            @Override
            public boolean isEmpty() {
                return true;
            }
        };

        list.addAll(Arrays.asList(1,2,3,4,5));
        new ListSummer().printSum(list);
    }
}
```

օլիմ: השוואת סוגי מחלקות מקווננות

	Initialization	Access	Usage
static nested class	<code>OuterClass.NestedClass obj = new OuterClass.NestedClass();</code>	All static members of the outer class	When the nested class doesn't need access to a specific instance of the outer class throughout its lifetime
Inner class	<code>OuterClass outerObj = new OuterClass(); OuterClass.InnerClass innerObj = outer.new InnerClass();</code>	All members of the outer instance.	When the nested class needs access to a specific instance of the outer class throughout its lifetime
Local class	Defined within a method and instantiated within the same method.	All members of the outer instance and local variables of the enclosing method.	When the nested class only needs to be used within the scope of a single method, and it is not necessary to expose it to the rest of the program.
Anonymous class	Defined and instantiated in a single expression.	All members of the outer instance and local variables of the enclosing method.	To create a short class that is only needed in a single context.

Functional Programming

ממשקים פונקציונליים

ממשק פונקציוני (functional interface) הוא ממשק שמכיל בדיקת מתודה אבסטרקטית אחת. קוראים לו כך כי הוא לא משתמש מחלקה, אלא את הפונקציה. הממשק יכול בנוסף להכיל מתודות אחרות, `default`, `private`, `static`, כלומר יתכן שתהינה עוד שיטות, כל עוד לא צריך למשוך אותן.

practice good: כדי לסמן את הממשק בתגית `@FunctionalInterface` כדי שהקומpileר יבטיח שהממשק מכיל בדיקת מתודה אבסטרקטית אחת.

```
@FunctionalInterface
interface PriceComponent {
    double getCost();

    static double freeRide() {
        return 0;
    }
}
```

ביטויי למבדה Lambda Expressions

למבדה היא ביטוי מקוצר למחלקה אוניברסלית (sugar syntax), וזה דרך פשוטה למשוך ממשק פונקציוני. למבדה לוקח את התחביר של מחלקה אוניברסלית, וזרקת ממנו כל מה שהוא יכול להנש או להבין בלבד, ונשארים עם משאו תמצית.

הסיבה לדרישה שהממשק של למבדה יהיה פונקציוני, כלומר יכול רק מתודה אבסטרקטית בודדת, היא CD' שלביוטי הלמבדה יהיה ברור לאיזה מהשיטות הוא מתייחס. אם קיימות שיטות אחרות הן מומושות כבר.

סינטקטוס:

- Made up of a parameter list, followed by the arrow token and a body:
`(parameter list) -> {statements}`
- For example, the following lambda receives two ints and returns their sum:

```
interface MathOperation {
    float operation(int a, int b);
}

class LambdasExample {
    MathOperation sum = (a, b) -> {a + b;};
}
```

בהמשך לדוגמה הקודמת, במקום להגדיר מחלקה אוניברסלית "מלאה", ניתן באופן 쉬ול לכתוב:
`IntPredicate predicate = num -> num % 5 == 0;`

כמו כן ניתן לקחת את הביטוי המושם (מודגש) ולשים אותו ישירות בתור פרמטר בקריאה למетодה שמקבלת משתנה מסווג `IntPredicate`.

از יש לנו משתנה `predicate`, הוא רפנס מסווג `IntPredicate`, כלומר נעשה עליו Upcasting ממופיע של מחלקה אוניברסלית שמסמכת את `Int`, וממשת את השיטה `acceptNumber` בצוරה זו:
מקבלת `num`, והחץ אומר אינטואיטיבית: ממירה את הקולט, שהוא פרמטר `num`, אל הביטוי הכלולاني `num%5==0`.
ובקצרה: `IntPredicate predicate` הוא רפנס ממופיע של מחלקה אוניברסלית הממשת את הממשק `Int`.

למבدا מימוש ממשק המתאר פונקציה (אבסטרקטית בודדת), אך השם של ממשקים שניתן לתאר אותם על ידי למבדות הוא ממשק פונקציוני (functional interfaces).

java.util.function

חבילת ממשקים פונקציונליים - כשם שג'ואה מספקת לנו מבני נתונים שאנו צריכים למשב עצמנו, כך היא גם חשבה מראש על ממשקים פונקציונליים נפרדים וממשה אותם עבורנו.

במהשך לדוגמה, יש מימוש קיים של Predicate של java.util.function.Predicate. בדוגמה זו עם משתנה גנרי כרצוננו.

از Predicate של טיפוס כלשהו T (לדוג' `Predicate<Integer>`), זה ממשק שמייצג פונקציה שמקבלת T, במקרה זהה `Integer`, ומחזירה `boolean`, עבורו או לא עבר איזהו תנאי. באופן שקול יותר ניתן להשתמש ב `IntPredicate` (יש גרסאות דומות עבור הפרימיטיבים האחרים).

ממשקים שימושיים נוספים: (ממשקים כלליים)

- `Consumer` - ממשק שמייצג את הפונקציה שמקבלת טיפוס T, ו"בולעת" אותו, מחזירה `void`.
- משמש עבור שליחת מתודות `void`-יות כפרמטר, לדוגמה הדפסה.
- `Supplier` - ממשק שמייצג את הפונקציה שמקבלת כלום ומחזירה טיפוס R .
- `Function` - ממשק המיציג פונקציה שמקבלת T ומחזירה R.

ולסיכום

```
Predicate<T>      ( T -> boolean )
Supplier<T>        ( void -> T )
Consumer<T>        ( T -> void )
Function<T,R>      ( T -> R )
```

Interface	Its abstract method	Description
<code>Function<T,R></code>	<code>R apply(T t)</code>	Represents a function that accepts one argument and produces a result.
<code>BiFunction<T,U,R></code>	<code>R apply(T t, U u)</code>	Represents a function that accepts two arguments and produces a result. This is the two-arity specialization of Function.
<code>Consumer<T></code>	<code>void accept(T t)</code>	Represents an operation that accepts a single input argument and returns no result.
<code>Predicate<T></code>	<code>boolean test(T t)</code>	Represents a predicate (boolean-valued function) of one argument.

*** Bi__ - gets 2 types.

```
public static void main(String[] args) {
    /* Use the Function<T, R> functional interface to Define a function
     * that takes a string and returns the number of "a"s in the string */
    BiFunction<String, Character, Integer> charInString = (str, character) -> {
        int count = 0;
        for (int i = 0; i < str.length(); i++) {
            if (str.charAt(i) == character) {
                count++;
            }
        }
        return count;
    };

    // Use the function to count the number of "a"s in a string
    String phrase = "Strawberries and Banana!";
    System.out.println("The number of 'a's in the string is: " + charInString.apply(phrase, 'a'));

    /* Use Predicate<T> to define a function that checks if a string is longer than 10 characters */
    Predicate<String> largeStr = (str) -> str.length() > 10;

    System.out.println("The string is empty: " + largeStr.test(phrase));
    System.out.println("The string is empty: " + largeStr.test("short str"));
}
```

רפרנסים למתודות

בדומה לשילוח למבוזת כפרמטר למתודה אחרת, ניתן לשולח מתודות או להתייחס אליהן כ reference.
יתרונות על פניו שימוש בלבודה: רפנס לשיטה זאת דרך קצר יותר מפורשת וישראל להגיד מה אני רוצה,
ממושך בצוורה קצר יותר עיליה.
בשביל להבדיל בין קרייה לפונקציה (method reference), משתמשים באופרטור ":".

ניתן לשולח רפנס למתודה סטטית באופן הבא:

ClassName::methodName

(בלי סוגרים!)

במקרה זה האופרטור מצין באיזו מחלוקת השיטה הסטטית נמצאת.

בצורה דומה ניתן לשולח רפנס לשיטת מופע.

למשל אם אנחנו רוצים לשולח את מתודת ההדפסה למתודה שרצואה לקבל Consumer, נשלח כפרמטר System.out::println

במקרה זה האופרטור מצין באיזה אובייקט שיטת המופיע נמצאת.

דוגמה נוספת:

נניח ויש לנו אובייקט בשם factory בשם שנין לקרוא ממנו לשיטה עם החתימה הבאה:

MyClass create(String type)

אפשר ליצור רפנס לשיטה באופן הבא:

Function<String, MyClass> func = factory::create;

יש עוד סוגים של הצבעות לשיטות, קצר יותר אוטומטיים (רלוונטיים במקרים קצר פחות שכיחים).

```
public class MethodReferenceExample {  
    public void printWithExclamation(String message) {  
        System.out.println(message + "!");  
    }  
  
    public static void main(String[] args) {  
        MethodReferenceExample example = new MethodReferenceExample();  
        List<String> messages = List.of("OOP", "IS", "COOL");  
        messages.forEach(example::printWithExclamation);  
    }  
}
```

OOP!
IS!
COOL!

סיכום: השוואה בין מחלוקת אונומית ללבודה

	Anonymous Class	Lambda Expression
Definition	An inner class without a name.	For declaring independent methods without a name.
From the compilers view	The compiler creates a separate '.class' file for every anonymous class.	Lambda expressions aren't converted to .class files. ⇒ saves time and memory.
when to use	When there is more than one abstract methods to implement.	For implementing functional interfaces.
Implementation	We need to write the class body.	We only need to provide the function body .

*** חידוד: נשים לב שלפי הנלמד בקורס השנה, למבודה **איננה** מחלוקת אונומית, היא איננה מחלוקת כלל.
למבודה **היא** מימוש אונומטי למתודה של ממש פונקציונלי.

```

public class AnonymousVsLambda {
    public static void main(String[] args) {
        /* Anonymous class */
        Printable a = new Printable() {
            @Override
            public void print(String s) {
                System.out.println("Using Anonymous class");
                System.out.println(s);
            }
        };

        /* Lambda expression */
        Printable b = s -> {
            System.out.println("Saving memory and run-time with Lambda Expression");
            System.out.println(s);
        };

        a.print("One may prefer Anonymous classes");
        System.out.println();
        b.print("Others prefer Lambdas");
    }
}

```

Anonymous Class Vs. Lambda Expressions

סיכום: Functional Programming vs. OOP

Functional Programming:

- When the output of functions depend on the input and not on the state of any object.
- Provides efficient, modular and clean code.

Summary:

- **OOP** ⇒ when there are many operations to be performed on entities, and you intend to extend the application by adding more entities.
- **Functional programming** ⇒ when the program consist mostly of operations, and only a few entities.

דוגמא: חישוב עלות נסיעה
לפni תכנות פונקציונלי:

```

class CalculateMyRide {
    public static void main(String[] args) {
        MillageCost millageCost = new MillageCost();
        TollRodeCost tollRodeCost = new TollRodeCost();
        CongestionCost congestionCost = new CongestionCost();
        ParkingCost parking = new ParkingCost();

        SplitTripCost calc = new SplitTripCost (millageCost, tollRodeCost, congestionCost, parking, 3);

        System.out.printf("The total Ride cost was: %.2f", calc.getTotalCost());
        System.out.printf("Each participant owes the driver: %.2f", calc.getSingleParticipantCost());
    }
}

```

אחרי:

Functional Programming

```
class CalculateMyRide2 {  
    public static void main(String[] args) {  
        double literPerKm = 0.06, kms = 37, shekelPerLiter = 8;  
        PriceComponent millageCost = () -> literPerKm*kms*shekkelPerLiter;  
  
        double kmsOnToll = 4, shekelPerKm = 6;  
        PriceComponent tollRodeCost = () -> kmsOnToll*shekkelPerKm;  
  
        LocalTime time = LocalTime.of(9,4);  
        PriceComponent congestionCost = () -> congestionPricing.floorEntry(time).getValue();  
  
        boolean free=true;  
        double minutes = 186, costPerMinute = 0.2;  
        PriceComponent parking = () -> {  
            if(free) {  
                return PriceComponent.freeRide();  
            }  
            return minutes*costPerMinute;  
        };  
  
        SplitTripCost2 calc = new SplitTripCost2 (millageCost, tollRodeCost, congestionCost, parking, 3);  
  
        System.out.printf("The total Ride cost was: %.2f",calc.getTotalCost());  
        System.out.printf("Each participant ows he driver: %.2f",calc.getSingleParticipantCost());  
    }  
}
```

effectively final

ג'אווה מאפשרת לנו להשתמש במחלקות לוקליות (ובפרט בתוך למבדות או במחלקות אונונימיות), במשתנים שהשיטה החיצונית מכירה, בין אם זה משתנים שהוא הגדרה או פרמטרים שהוא מקבל. במובן זה ג'אווה תומכת בסוג מסוים של מה שנקרא בשפות אחרות closure.

כשמדובר במחלקה לוקלית או למבודה, ניתן היה להשתמש במשתנים scope-חיצוני שהוגדרו לוקלית. כאשר יש שימוש כפול במשתנה, יכול להיווצר מצב שבו אם המשתנה משנה את ערכו, לא ניתן יהיה לדעת באיזה ערך התכוונו לשימוש - המוקורי או החדש.

בשביל למנוע את הבלבול, המשתנה החיצוני שבו משתמש מחלוקת לוקלית אמרור להיות final או effectively final.

כלומר אם יש משתנה שהוא לא פרמטר של הפונקציה, אבל היא מכירה אותו, נדרש שהוא מקבל ערך אחד ולא משתנה יותר - משתנה שלא משתנה. לא משנה אם Can/לא כתוב עליו את המילה final, הוא צריך להתנהג כמו final.

לקומפיילר לא משנה אם מה שמספר את final, הערך השני או השלישי שהוא מקבל, קורה לפני שמדריכים את המחלוקת (לדוג' predicate) או את הלמבדה או לאחר מכן. ברגע שהמשתנה כבר לא effectively final, אין אפשרות להכניס אותו למלבדה או למחלוקת לוקלית.

נשים לב שהצורה על המשתנה בתוך scope פנימי, בפרט בתוך for, היא כהצהרת משתנה חדש (בכל איטרציה נוצר משתנה חדש). אומנם לכל המשתנים האלה קוראים באותו שם, אבל אלה משתנים נפרדים, והם effective final.

נשים לב שגם המשתנה לא פרימיטיבי אלא רפנס שמצויע לאובייקט, הקומפיילר לא יזהה אם האובייקט משנה את המצביע שלו, כי המצביע עצמו עדין נחשב אפקטיבית final. במקרה הזה הלמבדה תנסה את ההתנהגות שלא לפי המצביע של האובייקט.

דוגמה

```
interface Employee {  
    void employeeData(String name);  
}  
  
class LambdaEffectivelyFinalTest {  
    public static void main(String[] args) {  
        int points = 100;  
        /* lambda expression */  
        Employee employee = name →  
            System.out.printf("The employee %s has %s points: ", name, points);  
            points++;  
            employee.employeeData("Sami");  
    }  
}
```

Variable used in lambda expression should be final or effectively final



שינו את המשתנה points, لكن יש שגיאת קומפיילציה. כדי לפתור זאת נמחק את השורה הבאה:
points++;

Callbacks

פרמטר שהטיפוס שלו הוא פונקציה. אם קיבלנו פרמטר Callback ניתן לקרוא לה מתישהו, לפי הצורך. כלומר, Callback היא פונקציה שקיבלנו ממשיהו אחר.

בג'אווה Callbacks ממומשות באמצעות ממשקים פונקציונליים - וכך שראינו פרמטר של למבודה או Callback method reference הוא

ממשק בילט-אין המתואג כממשק פונקציונלי, כי יש בו רק שיטה אבסטרקטית אחת. השיטה האבסטרקטית הזאת היא חוץ, והוא לא מקבלת כלום ולא מחזירה כלום. ככלומר Runnable מייצג פונקציה, כל פונקציה, שלא מקבלת כלום ולא מחזירה כלום.

Callback Strategies

עד עכשיו ראיינו שני סוגים מיוחדים של עצמים במשחק: ירשה (לדוג' לבנה יורשת מ GameObject ודורשת

מתודת onCollisionEnter) ואסטרטגיה (לבנה יורדת מאסטרטגיית העلمות).

עת נראתה אסטרטגיה שמשמעותה באמצעות CollisionGameObject, Callback Strategy, שנכנה במקומם להציג פונקציה, נקל פונקציה שמטפלת באסטרטגיה ונקראה לה.

בדוגמה שלנו: ניקח את אותו פתרון של האסטרטגיה עם המחלקה CollisionGameObject שיורשת GameObject ו록 מושיפה איזושהי התנהלות שתקרה כשהעצם יתנגש במשהו, אבל במקום להגיד שהוא מכילה מופע של אסטרטגיה שמטפלת בהתנגשות collisionStrategy collisionStrategy היא תכיל פונקציה שמטפלת בהתנגשות.

כלומר, זה Callback שקיים מימיiso אחר, כדי שיציר את העצם. נקרא לה collisionCallback. במקרה של אסטרטגיה להעלמת לבנה: נגידir vanishCallback ששווה לביטוי למבדה. ביטוי הלמבדה מקבל collision和其他, שאינו צריך לציין מפורשות את הטיפוסים שלהם, והתוכן של הלמבדה הוא לגרום לעצם brick להיעלם.

שני הבדלים בין הגישה הזאת לבין אסטרטגיה:

1. במקום להגדיר ממשק חדש של אסטרטגיה נשתמש בממשק פונקציונלי מובנה. (בדוגמה שלנו במקום CollisionStrategy נשתמש ב BiConsumer).
2. במקום שהאסטרטגיה תהיה מופע של מחלקה, נחשוב עליה ועל פונקציה. המחלקה שסමממת את האסטרטגיה התחלפה בביטוי למבדה או method reference למבודה או

יתרונות:

- נוחות - קיבלנו פתרון עם כל היתרונות של אסטרטגיה, ולא רק שהוא פחות מסורבל מאסטרטגיה, הוא פחות מסורבל אפילו מירושה.
 - ניתן לשנות את האסטרטגיה בזמן ריצה.
 - אין הגבלה על ידי ירושה נוקשה.
 - ניתן להגדיר כמה Callbacks לאותו GameObject, כל אחת מטפלת באיזושהי סוגיה אחרת.
 - קוד קצר וברור יותר.
 - מודולריות שבה אני שולט בנפרד באסטרטגיות ובנפרד GameObject.
 - ניתן גם לשלוח את אותן Callbacks לעצמים מסוימים אחרים.
- החדש הוא בעיקר בהפחתה של כמות הקוד, API מובן יותר.

זהירות בשימוש callbacks

בעולם של Callbacks אנחנו צריכים לחזור לשורשים ולהזכיר ב-API Minimal. Callbacks יכולות לחתוך אותנו מהכיוון המינימלי אל הנוח והמרוח, וזה פשוט טרייד-און שצריך לא להפריז בו. נקיטת משנה זהירות מהוספת פונקציונליות מיותרת היכולה לפגוע במקום להוסיף. בנוסף, אם המחלקה שאנו יוצרים אכן מופע שלה מאפשרת הרבה קוסטומייזציה באמצעות למבודות

שהן Callbacks וסטרטגיות, ונחנו מנצלים את קויסטומיזציה הזאת, אז הייצור של עצם אחד יכול לקחת בקהלות עשרות שורות קוד. כל זה נמצא בתוך איזושה' מחלוקת ראשית שיש לה עוד עשרים אתחולים כאלה על הראש, כל אחד עם קויסטומיזציה אחרת לגמר.

אז בשביל לשמר על הסדר הטוב, ובשביל לשמר על אנקפוסולציה, מומלץ בכל זאת להפריד את האתחול של האובייקט למחלוקת משלה בקובץ משלו.

LOSEIM

בקצרה, למדנו שהאנו זו אחליה שפה והיא שימושית בסך הכל! רק אל תעצמנו את הקומפיילר 😊

