

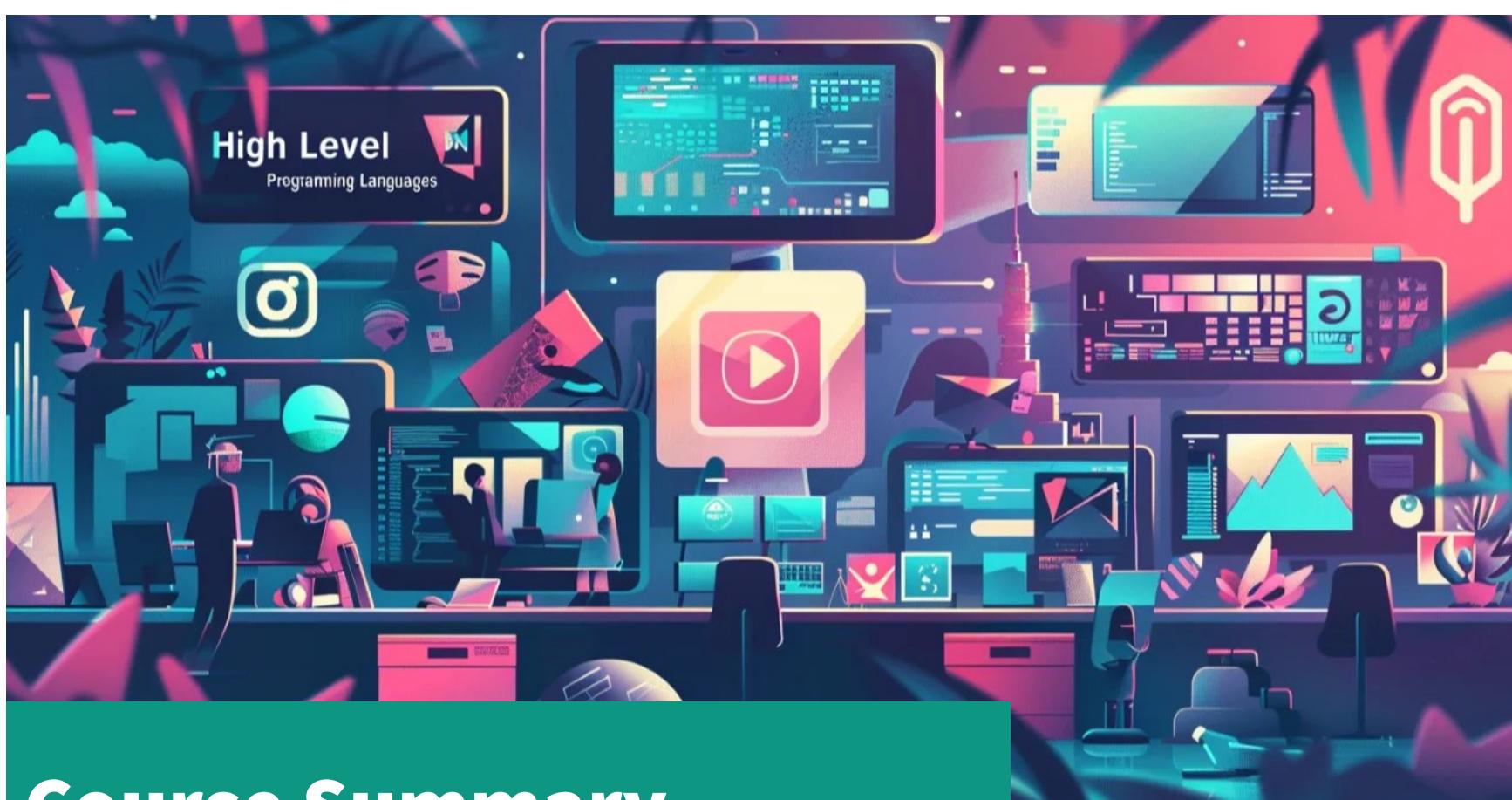
SUMMARIZED BY NOAM KIMHI
noam.kimhi@mail.huji.ac.il

Introduction to OBJECT ORIENTED PROGRAMMING

YEAR 2

SEMESTER A

2024-2025



Course Summary

COURSE NUMBER 67125

פרק 1 – מבוא לעצמים ויג'אוות

1.1 מנגנוןים בתוכנות מונחה עצמים: עצמים ומחלקות

מושגים בסיסיים

עקרונות בתוכנות מונחה עצמים

1. עצם object – מופע.
2. מחלוקת class – אוסף עצמים כאלה סוג.
3. תכונות attributes – הפעולות והמאפיינים של כל עצם במחלוקת.
4. שיטות methods – פועלות שהמחלקה מגדרה.

מילים בעלות משמעות זהה

Member == fields & methods	-
== == שיטה, מתודה, פונקציה	-
המצב של העצם השתנה == ערכי השדות של העצם השתנה	-
== משתנה מחלוקת/תכונה/שדה	-

מבנה של מחלוקת

יצירת מחלוקת ב-Java

```
Bicycle.java
class Bicycle {
    /* Data members */
    int speed = 0;
    int gear = 1;
    String brand;

    // Methods
    void changeGear(int newValue) {
        gear = newValue;
        return;
    }

    // Other methods
    int speedUp(int increment) {
        speed = speed + increment;
        return speed;
    }

    void break() {
        speed = 0;
        return;
    }
}
```

יצירת מופע

מתבצע באמצעות constructor. מתודת הבנייה נקראת כמו שם המחלוקת והוא לא מחזירה כלום, על כן לא>Returns בינה return.

```
Bicycle.java
class Bicycle {
    /* Data members */
    int speed = 0;
    int gear;
    String brand;

    /* Constructor */
    Bicycle(String myBrand, int newGear) {
        brand = myBrand;
        gear = newGear;
    }
}
```

שימוש בבנייה נעשה באמצעות המילה השמורה new, והדפסה באמצעות `System.out.println()`. דוגמה:

```

BicycleDemo.java
class BicycleDemo {
    public static void main(String[] args) {
        // Create two different Bicycle objects
        Bicycle bike1 = new Bicycle("BMX", 1);
        Bicycle bike2 = new Bicycle("newbike", 2);

        // Invoke methods on those objects
        bike1.speedUp(10);
        bike1.changeGear(2);
        System.out.println(bike1.gear+" "+bike1.speed);
        bike1.changeGear(3);
        System.out.println(bike1.gear+" "+bike1.speed);
        bike2.speedUp(10);
        bike2.break();
        System.out.println(bike2.gear+" "+bike2.speed);
    }
}

```

bike1 and bike2 belong to the Bicycle class

Output:
2, 10
3, 10
2, 0

לכל מחלקה קיימים עותק אחד בזיכרון של ג'אווה, אבל כל מופע של המחלקה תופס זיכרון משל עצמו. כאשר יוצרים אובייקט הוא צריך להיות מקשר למחלקה אחת (לא מודיק, יורחב בהמשך הקורס).

רפרנסים ופרימיטיבים

כל משתנה בג'אווה יכול להיות רפרנס לאובייקט אחר, או משתנה פרימיטיבי (כמו int, double, char, boolean).

רפרנסים/מצבייע לאובייקט

לא אובייקט אמיתי, אלא חז שמצויע לאובייקט קונקרטי. לכל רפרנס יש type שהוא שם המחלקה.

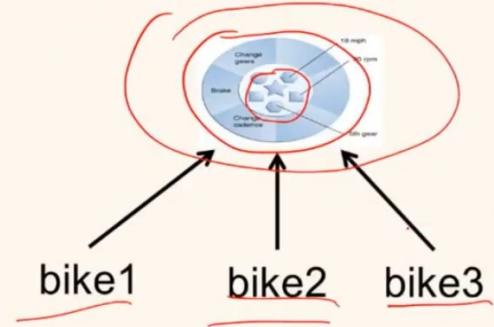
Content

Bicycle bike1 = new Bicycle(1);

החלק משמאלי לסימן = מגדיר רפרנס לאובייקט מטיפוס Bicycle, החלק הימני מגדיר אובייקט קונקרטי, ועוד bike1 מצוויע לאובייקט קונקרטי מטיפוס Bicycle.

- The creation of new references doesn't waste much memory

- Bicycle bike1 = new Bicycle(1);
- Bicycle bike2 = bike1;
- Bicycle bike3 = bike1;
- ...



שינויים שמבצעים לאובייקט דרך bike1 משפיע גם על bike2 ועל bike3.

אם מגדירים ל-bike1 אובייקט חדש להצבוע אליו, יבוא טיפוס שנקרא Garbage Collector ויאסוף את האובייקט הקונקרטי שאינו אלא רפרנס:

Bicycle bike1 = new Bicycle(1);

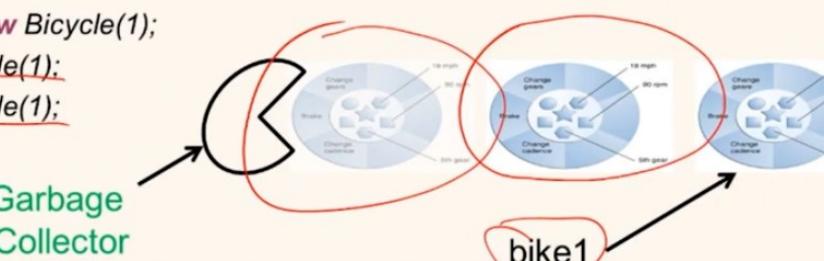
bike1 = new Bicycle(1);

bike1 = new Bicycle(1);

Bicycle bike1 = new Bicycle(1);

bike1 = new Bicycle(1);

bike1 = new Bicycle(1);

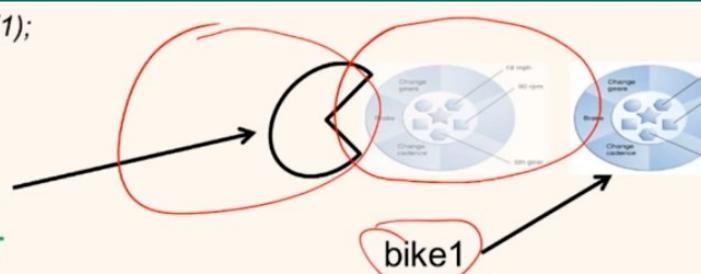


Bicycle bike1 = new Bicycle(1);

bike1 = new Bicycle(1);

bike1 = new Bicycle(1);

Garbage
Collector



By reference vs. By value

מתואר מקרה:

```
int cookieCount = 2;
assignOne(cookieCount);
System.out.println(cookieCount);
```

```
void assignOne(int param) {
    param = 1;
}
```

במקרה זה הפונקציה `assignOne` תפקידה ללקח משתנה ולשים בו את הערך 1. האם כשנדפיס את `cookieCount` נקבל 1 או 2?
תשובה: נקבל 2, כי מה בקריאה לפונקציה `assignOne` נוצר תא חדש עבור `param` שמקבל עותק של מה שהלחנו אל הפונקציה, שבתחלתה ערכו 2, ואז המתודה משנה אותו ל-1. בסיום המתודה המשתנה `param` נעלם, ולמעשה לא שינו את `cookieCount` ולכן יודפס 2.

ברפרנס

עם רפרנס קורה דבר שונה, למשל במקרה:

```
Box mysteryBox = new Box();
openBox(mysteryBox);
System.out.println("Is mystery box open? " +
                    mysteryBox.isOpen());
```

```
void openBox(Box box) {
    box.open();
}
```



יצרנו מופע של המחלקה `Box` וכשהעבರנו אותו למתחודה נוצר עוד רפרנס למופע, ולכן כל שינוי שבוצע בתחום המתודה נשמר גם אחרי שהרפרנס למופע נמחק.

לעומת זאת:

```
Box mysteryBox = new Box();
openBox2(box);
System.out.println("Mystery box is open? " +
                    mysteryBox.isOpen());
```

```
void openBox2(Box box) {
    box = new Box();
    box.open();
}
```



במקרה הזה לא ישנה דבר, כי בתחום המתודה אנחנו יוצרים מופע חדש של המחלקה, שלא משפיע על המופע המקורי.

לסיכום, בგ'אזהו בשולחים לשיטה פרימיטיב אנחנו שולחים רק עותק שלו והשיטה לא יכולה לשנות אותו בצורה שתשתקף אצל מי שקרה לה, ולכן שפרימיטיבים נשלחים `value by` לומר הערך שלהם נשלח אבל בתובתם לא. לעומת זאת רפרנס אנחנו כן יכולים לשנות אותו, והאובייקט עצמו מעבר `reference by`, העברנו מצביע לאובייקט. הרפרנס עצמו הוא רק עותק, אך אם משנים את הרפרנס להצביע לאובייקט אחר – זה לא ישתקף מוחז למתודה.

null

בעצם, `null` הוא כלום, אפשר ליצור מצביע לאובייקט ממשי, או לא להצביע לכלום; `null`:

אולם אם ננסה לגשת לאייר של העצם כך: `example.gear;`

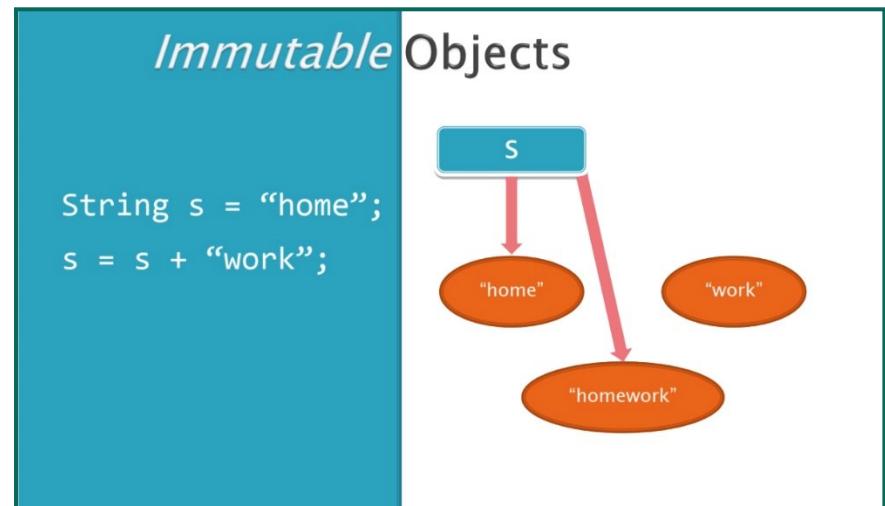
נקבל שגיאת זמן ריצה בשם `NullPointerException`.

כasher מגדרים מצביע חדש כך: `Bike example;`

ה מצביע יגידך באופן דיפולטיבי לכלום, ולכן ערכו יהיה `null`.

[String](#)

מחלקה נפוצה בJAVA, שאינה פרימיטיב אבל עדין אפשר לאתחל מופע שלה באמצעות סימן '=' ולא בקונסטרוקטור. יש לה הרבה Method'ות כגון `length()`, `charAt()` ועוד. ברגע שיצרנו מופע אי אפשר לשנות את התוכן שלו. המופעים שלה לא ניתנים לשינוי, וכך גם `String` הוא Immutable. כלומר לא ניתן לשנות את התוכן שלו.



בדוגמה זו לא מייצרים אובייקט `String` שערךו "home" ולאחר מכן משנים אותו לערך "homework" אלא גורמים לו-z להציגו לאובייקט חדש מסוג String שערךו "homework".

1.2 ג'אווה סטרים: ChatterBot[main, IO, import](#)

ב-Java יש שני שלבים, שלב של קימפול (בו הקומpileר של Java מודיא שבכבודו קוד נכוון תחבירית) ושלב הריצה.
[קבלת קלט ב-Java](#)

קבלת קלט ב-Java מתבצעת באמצעות מחלקת `Scanner`. ניצור מופע של המחלקה כך:

```

import java.util.Scanner;

class Chat {
    public static void main(String[] args) {
        System.out.println("What should I say?");
        Scanner scanner = new Scanner(System.in);
        String textToPrint = scanner.nextLine();
        System.out.println(textToPrint);
    }
}
    
```

מייבאים את `java.util.Scanner` ועוד כתובים את השורות כמו שמתואר.

[Class, default constructor, while](#)

כאשר לא מגדירים בנאי ב-Java, מאחריו הקליעים מוגדר בנאי בירית מחדר, שלא מקבל כלום ולא עושה כלום. ברגע שנגדר בנאי הוא יחליף את בנאי בירית המחדל.

[לולאת while](#)

```

import java.util.Scanner;

class Chat {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String statement = scanner.nextLine();

        ChatterBot bot = new ChatterBot();

        while(true) {
            String reply = bot.replyTo(statement);
            System.out.println(reply);
            statement = scanner.nextLine();
        }
    }
}
    
```

שימוש בתנאים ובMETHOD של String

```
class ChatterBot {
    String replyTo(String statement) {
        String reply = "";
        if(statement.startsWith("say ")) {
            return statement.replaceFirst("say ", "");
        }
        //if reached here, than the request is ILLEGAL
        //handle illegal request
        return "what";
    }
}
```

קבועים

למשל בדוגמה שבתמונה לעיל, אם למשל נרצה להחליף את "say" למשהו אחר אנחנו יכולים לשכוח להחליף או ליצור טעויות. במקרה זה מוטב להשתמש בקבוע.

כדי להגדיר קבוע נכתב:

```
static final int NUM_OF_LEGS_PER_DOG = 4;
```

```
static final String REQUEST_PREFIX = "say ";
```

- static – אומר שהקבוע לא מקשר למופיע, אלא באופן כללי.

- final – הערך לא ישנה ברכישת התוכנית.

מחלקה Random ואתחול שדות

נניח שנרצה להוסיף למחרוזת נוספת באופן אקראי, ניצור מופיע של מחלקת Random אחרי שייבאנו Random.java.util.

עדיף ליצור מופיע של Random פעם אחת ואז להמשיך וואז להציג מופיע בכל פעם חדשניים שהוא באופן אקראי.

```
class ChatterBot {
    static final String REQUEST_PREFIX = "say ";
    static final String RESPONSE_TO_ILLEGAL_REQUEST = "what ";

    Random rand = new Random();

    String replyTo(String statement) {
        if(statement.startsWith(REQUEST_PREFIX)) {
            return statement.replaceFirst(REQUEST_PREFIX, "");
        }
        return respondToIllegalRequest(statement);
    }

    String respondToIllegalRequest(String statement) {
        String reply = RESPONSE_TO_ILLEGAL_REQUEST;
        if(rand.nextBoolean()) {
            reply = reply + statement;
        }
        return reply;
    }
}
```

מערכות והמילה השמורה thisמערכות

כאשר לא מאתחלים מערך, כמו למשל לכתבו

```
int[] arr;
```

זה שקול לכתבו

```
int[] arr = null;
```

כל עוד לא שמים ערך זה מאותחל ל-arr, בנגד לקרה של; num int num שボ ממש מאותחל תא עם ערך 0.

```
int[] arr = new int[5];
int num = arr[arr.length - 1];
```

ברגע כל אברי המערך מאותחלים להיות 0.

כל הדריכים הבאות תקינות:

```
String[] repliesToIllegalRequests = { "what ", "say I should say " };  
ChatterBot bot = new ChatterBot(repliesToIllegalRequests);
```

```
String[] repliesToIllegalRequests = new String[2];  
repliesToIllegalRequests[0] = "what ";  
repliesToIllegalRequests[1] = "say I should say ";  
ChatterBot bot = new ChatterBot();
```

המילה השמורה this

בשאנו רוצה לעדכן שדה של מופע שהוא בעל שם זהה לפרמטר של מתודה, נוכל להשתמש ב-`this` כדי להבדיל ביניהם (ניתן גם לשנות שם, אבל `this` היא הדרך המקובלת יותר). דוגמה:

```
class ChatterBot {  
    static final String REQUEST_PREFIX = "say ";  
    static final String RESPONSE_TO_ILLEGAL_REQUEST = "what ";  
  
    Random random = new Random();  
    String[] repliesToIllegalRequests; ←—  
  
    ChatterBot(String[] repliesToIllegalRequests) {  
        this.repliesToIllegalRequests = repliesToIllegalRequests;  
    } ←—
```

לולאות for

סינטקס זהה לשפת C

```
ChatterBot(String[] repliesToIllegalRequests) {  
    this.repliesToIllegalRequests = new String[repliesToIllegalRequests.length];  
  
    for(int i = 0 ; i < repliesToIllegalRequests.length ; i = i + 1) {  
        this.repliesToIllegalRequests[i] = repliesToIllegalRequests[i];  
    } ←—
```

קונבנציות

הזהות (אינדנטציה), סגירים מסולסים באותה שורה שבה נפתח scope חדש, קבועים באותיות גדולות, עבר מחלקות ושם הקובץ שלהם, שמות מתחילהם באות קטנה.

פרק 2 Encapsulation and API – 2

2.1 עיצוב תוכנה: הגדרת ממשק למתכנת

אנקפסולציה

אנקפסולציה (בימוס) מורכבת משני חלקים: ראשית, **לאגד- ליחידה אחת את כל הפונקציות והמשתנים שעוסקים באותו תחום אחריות.** החלק השני הוא להסתיר מידע, כל עצם מפריד בין מה שמשרת יחידות אחרות, לבין מה שנועד לשימוש פנימי.

מודיבציה להסתרת מידע

גורמת לאנקפסולציה, פיתוח אינקרמנטלי (אפשרות לפתח את הקוד הלאה), מקל על התמצאות, פחות באגים ויתר קל לדיבג אותם.

מגדרי נראות

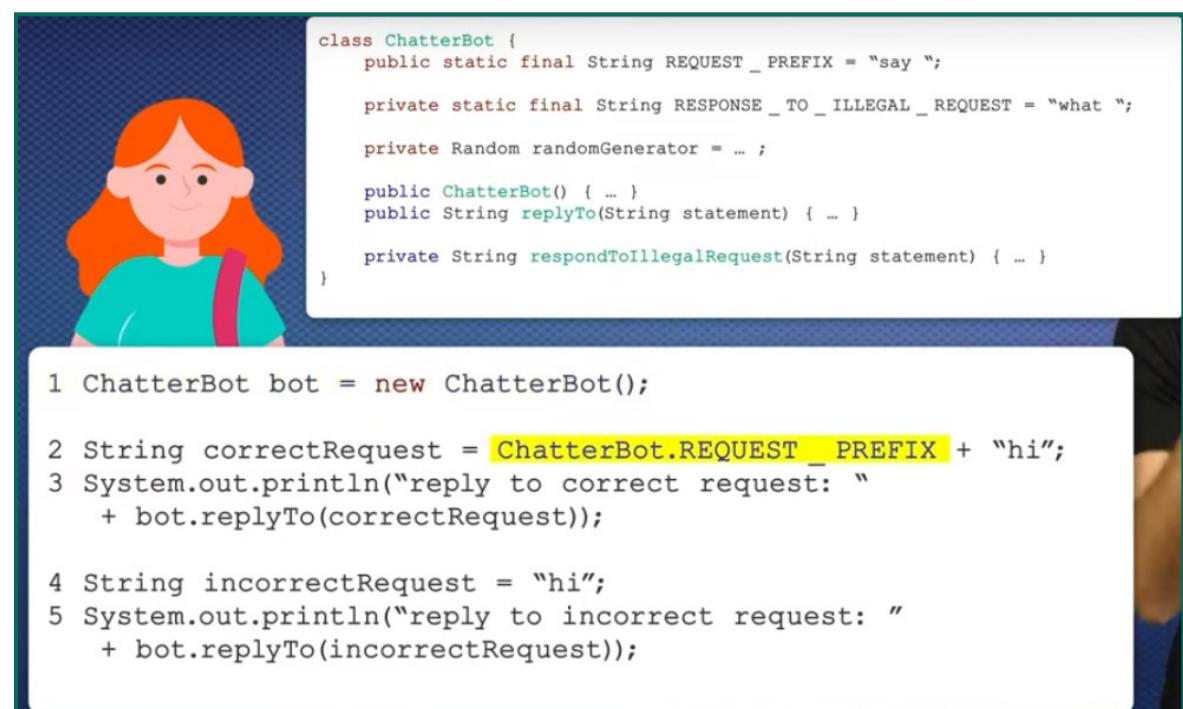
מגדרי נראות

בשפת Java ישן שני מיללים שמורים – `private`, `public`. איברים (שודות או שיטות) שמסומנים כ-`public` זמינים מחוץ לעצם, ואילו ניסיון גישה לאיברים שמסומנים ב-`private` יכשל בקומpileציה.

```
class MyClass {  
    public void publicMethod() { ... }  
    private void privateMethod() { ... }  
}  
...  
MyClass obj = new MyClass();  
obj.publicMethod(); //compiles  
obj.privateMethod(); //line will not compile: "privateMethod" is declared private
```

קובעים בדרך כלל יופיעו ראשונים במחלקה, קודם הפומביים ואז הפרטיים.

הערה חשובה – נשים לב שכאן הגישה לקבוע `REQUEST_PREFIX` נעשית דרך המחלקה ולא דרך הרפרנס, כי הקבוע לא שייר למופע מסוים אלא לכל המחלקה:



האיברים הפומביים – API/ממשק, Application Programming Interface, במחלקות ארכות אפיו נהוג לסמן כך:

```
class MyClass {
    // ====== public constants ======
    public static final int PUBLIC_CONSTANT1 = ...;
    public static final int PUBLIC_CONSTANT2 = ...;
    public static final int PUBLIC_CONSTANT3 = ...;
    public static final int PUBLIC_CONSTANT4 = ...;

    // ====== private constants ======
    private static final int PRIVATE_CONSTANT1 = ...;
    private static final int PRIVATE_CONSTANT2 = ...;
    private static final int PRIVATE_CONSTANT3 = ...;

    // ====== fields ======
    private int field1;
    private int field2;
    private int field3;

    // ====== public methods ======
    public void publicMethod1() { ... }
    public void publicMethod2() { ... }
    public void publicMethod3() { ... }
    public void publicMethod4() { ... }

    // ====== private methods ======
    private void privateMethod1() { ... }
    private void privateMethod2() { ... }
}
```

כשאנחנו ניגשים לכתב קוד חדש, נעבד בשלושה שלבים:

- (א) **ankepsoltsia** – נחלק את העצים לקפסולות לפי תחומי אחריות.
- (ב) **קביעת API של עצם** – חוצה של מחלקה עם הלקוחות שלה.
- (ג) **ימוש** – כל האיברים שנוסיף בשלב זה הם פרטיים.

Minimal API

עקרון הממשק המינימל אומר דבר פשוט – **KEEP IT SIMPLE**.

האובייקט צריך לעשות כל מה שמצופה ממנו, ורק מה שמצופה ממנו.

אבסטרקציה

אבסטרקציה היא התרחקות מהשימוש והתקינות לצורך של הלוקה. בהינתן API אבסטרקט, הוא לא משתנה גם בשימושים שונים בימוש, ואך שינוים בקוד לא מתפשטים.

אבסטרקציה מפירה בין מה לאיך. מאפשרת לקבל ממוק נוח יותר לлокה ומוגן משינויים.

Private != Secret

צריך לעשות הפרדה בין `private` לבין `secret`. יש נטייה לחשב שאפשר לשומר `private` מידע רגיש כמו סיסמות, אבל יש בಗ'ואה דרכים לעקוף את הגישה הפרטית – וכך אסור לעשות זאת.

Mini JavaStream: A little Bit More Java 2.2

מערכות DO-MANDIM

אם נרצה מערכת של מערכות של `float` נגידו: `float[][] pic;` בדומה לשפת C, יש לנו מערכת שבשבבנה הראשונה יש לנו מצביעים, כל אחד מהם מצביע למערך של `float`. נאתחל באחת הדרכים הבאות:

```
float[][] pic = new float[3][];
```

```
class Arr2d {
    public static void main(String[] args) {
        float[][] pic = {
            { 0, 1, 0, 1 },
            { 1, 0, 1, 0 },
            { 0, 1, 0, 1 }
        };
    }
}
```

Enums

enum מתפקיד תחבירית כמו המילה השמורה `class`, מדובר בטיפוס מסווג enum שיכל להכיל ערכים.
דוגמה:

```
enum PieceColor { BLACK, WHITE }

class Piece {
    private PieceColor color;

    public Piece(PieceColor color) {
        this.color = color;
    }
}
```

גישה ל-enum מתבצעת כך כמו גישה קבועה, באופן הבא:

```
public static void main(String[] args) {
    Piece whitePiece = new Piece(PieceColor.WHITE);
}
```

מערכות enums

// נספף מידע חדש מהותי

אופרטורים לוגיים

```
boolean cond1 = true;
boolean cond2 = false;

if(!cond1) {}

if(cond1 && cond2) {}

if(cond1 || cond2) {}

boolean cond3 = !cond1 && cond2;
```

ג'אווה תעבור ביעילות בבדיקה תנאים, למשל אם יש פעולה יקרה ואני ארצה לבדוק לבודק `cond2 && expensiveOperation`, מכיוון ש-`cond2` הוא לא תיבדק כלל, כי במקרה תוצאה הגיומם תיתן `false`. בדומה עם אופרנד `|`.

פרק 3 – ממתקים ופולימורפיזם

3.1 מנגןונים בתכנות מונחה עצמים: ממתקים ופולימורפיזם

מוטיבציה לממתקים

דוגמה לממתק:

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

ממתק הוא לא כמו מחלקה, ולא ניתן ליצור מופעים שלו. הממתק רק מגדיר איזה שיטות יש לעצם מחלקה בלבד. ולרוב אין בו בכלל קוד. המחלקה גם צריכה להציג שהיא ממשת את הממתק. למשל:

```
class FurFly implements Carriable {
    ...
    public void attachTo(Child child) {
        ...
    }

    public void unAttach() {
        ...
    }
}
```

נשים לב שכאשר מחלקה ממשת ממתק בלבדו, צריך לוודא שהיא מכילה את כל השיטות שיש בממתק (במקרה זה `attachTo`-ו `unAttach`). על השיטות האלה להיות פומביות.

dagsh – אפשר להוסיף מетодה `reply` למשת מחלקה `Child`. יcollה למשת מחלקה `Child` `reply` שנמצאת בממתק `Carriable` גם ביל שזכהיה שהמחלקה `Child` implements `Talkable` והקוד יתאפשר וריאץ, אבל ההפך לא עובד. כלומר אם כתבנו: `FurFly` אבל לא שמננו במחלקה את המתודה `reply` הקוד לא יתאפשר.

מדוע כדאי להשתמש בממתקים?

משמשים להגדרת חוזה שמחלקה לוקחת על עצמה. למשל ממתקים כגון: `Comparable`, `Printable`, `Clonable`. אינטראפיזיסם מדברים על מה עושים ולא על איך עושים.

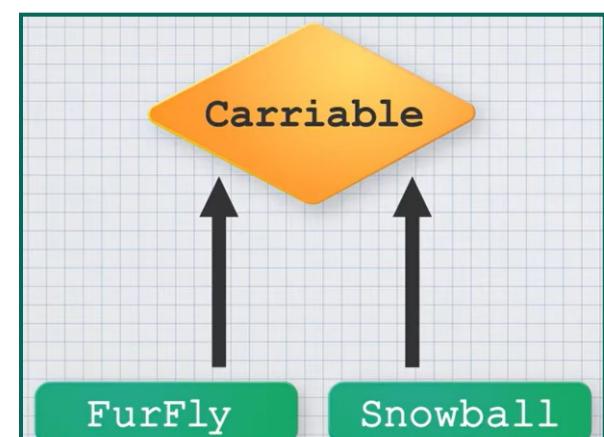
המרה בלפי מעלה

Upcasting

בדוגמה לעל, מופיע של `Child` יכול להרים עצמים עם המתודה `attachTo` וכרגע היא מצפה לקבל רק `Snowball`. אבל אנחנו רוצים שילד יוכל להרים כל עצם שמשת את הממתק `Carriable`. כיצד נעשה זאת? באמצעות `c`. מדובר ברפרנס בלי מסווג הממתק, שמצוין על עצם מסווג של מחלקה קונקרטית.

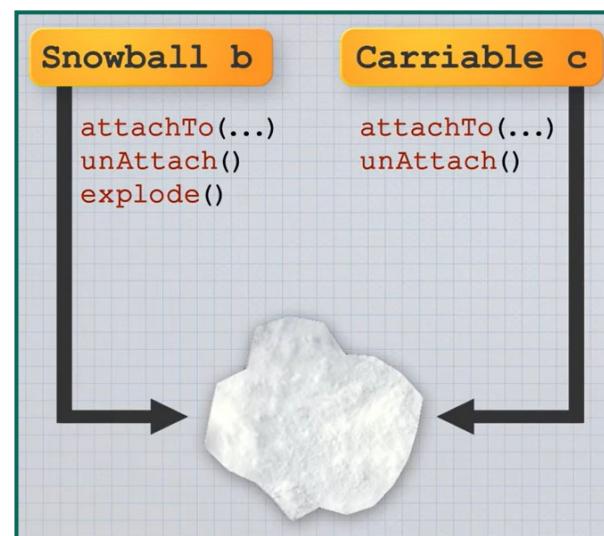
```
Carriable c = new FurFly();
```

זה מותר למרות שהטיפוס מצד ימין לא שונה מהטיפוס שבצד שמאל. זה מתאפשר רק בגין ההצהרה במחלקה `Furfly`.



ברגע הטיפוס `FurFly` הוא על הטיפוסים `Snowball`-ו `FurFly`, וזה נקרא המרה בלפי מעלה או `Upcasting`.

יכול להיות של FurFly ו-Snowball יש שיטות נוספות, ואלו אפיו שיטות משותפות לא דרך Carriable, אבל במשמעותם עליון עם פרנס מסוג Carriable, יוכל לקרוא רק לשיטות משמעותן ב-Carriable.



הערה: קיימת גם המירה כלפי מטה הנקראת Downcast, אך היא הרבה פחות נפוצה ויורח במעלה בהמשך.

מימוש מספר ממשקיים

אין מניעה עקרונית שעצם עם מספר התחנויות ימשך מספר ממשקיים, אך כדי לראות יותר משלבים.

ממשקיים בהסתכבות שונה

// לא נוסף מידע חדש מהותי

פולימורפיזם

המשמעות של פולימורפיזם היא שני עצמים המשווים לממשק (למשל Carriable) יכולים להיות מחלקות שונות, אבל למתכנת זה בכלל לא משנה. בשnoch ליאתיכס אליהם כ-*Carriable* ובל' בכלל להכיר את המחלוקת אליהם משתייכים. האפשרות להיות אדיש למחלוקת הקונקרטית היא אכן יסוד בתוכנות מונחה עצמים.

בדוגמה נסתכל למשל על הממשק :

```

Interface Shape {
    double perimeter();
    double area();
}
  
```

מחלקות Circle-Square מימוש את הממשק בדרךים שונות, ריבוע בפונקציה של הצלע, ועיגול בפונקציה של הרדיוס:

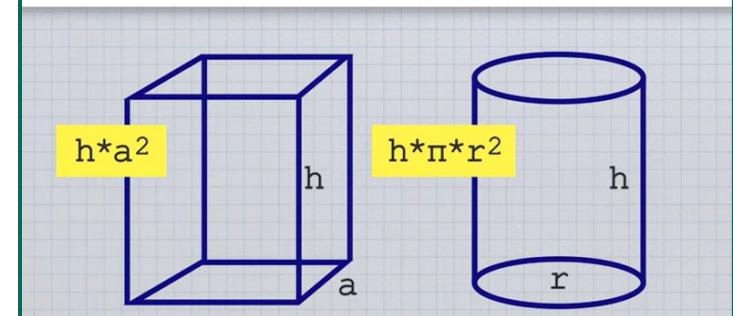
```

class Square implements Shape {
    private double edgeLength;
    ...
    public double perimeter() {
        return 4 * edgeLength;
    }
    public double area() {
        return edgeLength * edgeLength;
    }
}

class Circle implements Shape {
    private double radius;
    ...
    public double perimeter() {
        return 4 * Math.PI * radius;
    }
    public double area() {
        return Math.PI * radius * radius;
    }
}
  
```

אם נרחיב את התוכנית, ונכניס גם מחלקה בשם Building Building בעלת בסיס מצורה כלשהי ולה יש מתודה המחשבת את נפח המבנה, בפועל המתודה volume מבצעת חישובים שונים לגמרי בתולות בצורה הבסיס – אבל הקוד נקי לחולטי if statements – אם זה מאפשר באמצעות פולימורפים:

```
class Building {
    private double height;
    private Shape baseShape;
    ...
    public double volume() {
        return height * baseShape.area();
    }
}
```



אבסטרקציה ופולימורפיזם הם הבסיס של כתיבת קוד כללי.

מבנה למשك

מבנה למשק ולא למימוש – תמיד נשאף לכתוב קוד עבור הטיפוס הגבולה ביותר, עם המטרה הכללית ביותר. מבנים למשק אומר לשאל את עצמו האם לשוב כדור שלג מנוקודה א' ל-ב' זו המטרה הכללית ביותר? לא, אנחנו רוצים לשוב כל Carriable מ-א' ל-ב'. מונע במקרה בפל קוד, משרת דברים שעוד לא חשבתי עליהם.

מימוש שונה במחלקות שונות

אם למשל נרצה שסחיבת של FurFly תגרום לכדור הפרווה לגרגר, בעוד שסחיבת של Bird תגרום לה לנפנוף בכנפיים. עשויו כן לצורך הפריד לשתי מטריות שונות במחלקה Child שמרימה שהוא מושה Carriable? התשובה היא לא. ובאן נכנס העניין של אנקפסולציה. למחלקה Child לא אכפת אופן התגובה של מה שהוא הרימ, וזה באחריות המחלקות FurFly ו-Bird למשם מה קורה לשמורים עצם המשיך אליו. ברגע שאנו מרגיש שאין צורך לעשות משהו עבור מחלקת אחת ומהו אחר עבור מחלקת שנייה, ככל הנראה שברתי אנקפסולציה. אם שברנו מבנים למשק, סביר ששברנו גם אנקפסולציה.

3.2 עיצוב תוכנה: Factory

ישור קו: השוואת מחרוזות ב'גואה'

בහינתן קטע הקוד:

```
String x = "string object";
String y = "string object";
```

ההשוואה בין הרפרנסים `y==x` בודקת אם שני המצביעים מצביעים על אותו עצם, היא משווה בין המצביעים. על מנת לבדוק האם שתי המחרוזות שוות, גם אם הן שמורות בשני עצמים נפרדים, משתמש בשיטה `equals` של המחלקה String ב-`(y).equals(x)`.

מבנה עיצוב

מבנה עיצוב – תריחסים נפוצים בהרבה תוכנות ופתרונותים שמתאים להם שפותחו בעבר, מונע ממי לפתח מחדש ביעות קטנות שכבר יש להן פתרון. אחת מtabיות העיצוב האלו נקראת מפעל – Factory.

מotifyה למפעל

אם למשל נרצה לכתוב מחלקה שמקבלת שנקראת BuildingPlacer Building שמקבלת מהמשתמש שם של צורה. אומנם המחלקה Building נכתבה למשק ולא למימוש, אך לא אכפת לה מה הצורה שבבסיס שלה – אבל באמצעות הקוד נוצר לבעיה למחרוזת שקיבלנו מהמשתמש אל הצורה המתאימה וזה יראה כך:

```
shapeName = scanner.next();

Shape baseShape;
if(shapeName.equals("square"))
    baseShape = new Square(...);
else if(shapeName.equals("circle"))
    baseShape = new Circle(...);
else if(shapeName.equals("triangle"))
    baseShape = new Triangle(...);
...

Building building = new Building(baseShape, 100);
```

מה יקרה בשנרצה להוסיף עוד צורות? זו צורת כתיבה שתלויה במימושים ולא רק במשמעות, אנחנו מנסים להימנע מכתיבת צו.
את החלק ה"גוע" באופן ספציפי למימוש אנחנו רצים לבודד, ואת זה נעשה עם **מפעל**.

מפעל

ניקח את החלק של יצירה העצם שמחייבת אותנו לנקוב בשם **ShapeFactory** של מופע, וניצא אותו למחלקה חיצונית ש"טספוג את האש". למחלקה בוצאת קוראים **מפעל**. למשל בדוגמה לעלה, נקרא למפעל הספציפי זהה **ShapeFactory**. כשייהו שינויים בראשית הצורות המפעל מבונן יצטרך לשנתנות, אבל הוא יהיה יחיד שzierיך לשנתנות.

איך נכתב מפעל את המתודה בתחום המפעל?

```
public Shape buildShape(String shapeName, double[] shapeParameters) {  
    if(shapeName.equals("square"))  
        return new Square(...);  
    else if(shapeName.equals("circle"))  
        return new Circle(...);  
    else if(shapeName.equals("triangle"))  
        return new Triangle(...);  
    ...  
    return null;  
}
```

נשים לב שהשימוש לא מטפל בכל הקלטים, אבל זה יורחב בהמשך. בעצם המחלקה תראה כך:

```
shapeName = scanner.next();  
double[] shapeParameters = ...  
ShapeFactory factory = new ShapeFactory();  
Shape baseShape =  
    factory.buildShape(shapeName, shapeParameters);  
Building building = new Building(baseShape, 100);
```

מפעל הוא עוד תצורה של הסורתה מיידע, אבל בסקרה של מחלוקת. האו גם מסיע באבstraction, כי הוא לא משתף את המשתמש אין הוא מייצר את הצורה, אלא רק מבטיח לו שיקבל צורה בהינתן קלט תקין.
הערה: כמה מחלוקות שמלאות **יחד** שירות מאוחד עברו לקוחות חיצוניים נקראת **מודול**.

עקרון הבחירה היחידה

מבנה העיצוב "מפעל" מתכנתת עם עקרון עיצובי ידוע: עקרון הבחירה היחידה (The Single Choice Principle). לפי עקרון הבחירה היחידה, אם יש משה בתוכנית שלנו עבורי תיתכנה מספר אפשרויות, אז הרשימה המלאה של האפשרויות הללו תופיע רק במקום אחד.

3.3 ג'אווה סטרים: ChatterBot 2.0

JShell וחלוקת שלמים

JShell

אפשר לבתו בטרמינל `jshell` ולקבל `jshell` אינטראקטיבי בשפת ג'אווה.

`vars` / יציג לי את כל המשתנים שהגדרתי

אפשר גם להגיד שיטות לא בתחום מחלוקת, למשל:

```
<press tab again to see synopsis>  
jshell> int factorial(int n) {  
...>     if(n <= 1) return 1;  
...>     return n * factorial(n-1);  
...> }  
|   created method factorial(int)
```

חלוקת שלמים

אם נרצה להכניס משתנה מטיפוס `double` לתוך משתנה מטיפוס `int`, נדרש להודיע לקומפיילר ישירות שאנו מתוכונים לעשות זאת, כי אחרת יעצור אותנו מחשש לטעות:

```
jshell> double d = 5.3;  
d ==> 5.3  
  
jshell> int n = d  
| Error:  
| incompatible types: possible lossy conversion from double to int  
|   int n = d;  
|   ^
```

```
jshell> int n = (int) d;
n ==> 5

jshell> d = 5.7
d ==> 5.7

jshell> int n = (int) d;
n ==> 5
```

המשתנה `d` קיבל רק את החלק השלם של ה-`double` ויזרוק את החלק השברי.
אם בכלל זאת נרצה לבצע עיגול אפשר להשתמש בזה כך: `n = (int) Math.round(d)`.
תוצאה של חלוקה `t / n` היא תמיד שלמה, אבל בשימושים פעולה עם שני טיפוסים מורכבים שונים, התוצאה של האופרציה תהיה מטיפוס המורכב מבין השניים (למשל אופרציה על `int` ועל `double` תיתן תוצאה מסוג `double`).

```
jshell> 37/10
$9 ==> 3

jshell> 37.0/10
$10 ==> 3.7

jshell> 37D/10
$11 ==> 3.7

jshell> (double)37/10
$12 ==> 3.7

jshell> (double)(37/10)
$13 ==> 3.0

jshell> 37/(double)10
$14 ==> 3.7
```

הגדרת מפעל קש Kushner

Command Line Arguments

אנו מקבלים במתודת `main` רשימה של ארגומנטים שאפשר להשתמש בהם בהתאם לדרישות התוכנית. למשל אם נרצה לẤת מערך של `Chatter` עם סוגים שונים של בוטים, נוכל לקבל את סוג הבוטים הנדרשים בהתאם:

```
class Chat {
    public static void main(String[] args) {

        Chatter[] chatters = new Chatter[args.length];
        for(int i = 0 ; i < chatters.length ; i++) {
            chatters[i] = null; //init according to the cmd args
        }
    }
}
```

Factory & String Format

כמו שלמדנו, קטע קוד שמתפל במופעים ספציפיים של מחלקות י יצא למפעלי:

```
class ChatterFactory {
    Chatter build(String arg) {
        if(arg.equals("echo")) {
            return new EchoChatter(new String[] { "what " });
        }
        /*
        else if(arg.equals("other kind of chatter")) {
            return other chatter
        }
        */
        return null;
    }
}
```

בעת נחזיר למחלקה שלנו, ניצור מופע של המפעל ואז ניצור מערך של `Chatter` בהתאם לארגומנטים שקיבלנו:

```
ChatterFactory factory = new ChatterFactory();
Chatter[] chatters = new Chatter[args.length];
for(int i = 0 ; i < chatters.length ; i++) {
    chatters[i] = factory.build(args[i]); //init according to the cmd args
    if(chatters[i] == null) {
        System.err.println( String.format("Failed to create chatter %s.", args[i]) );
    }
}
```

נשים לב לאופן הטיפול בקלט לא חוקי. נעשה בכך שימוש ב-`String.format`, בעוד אנחנו יוצרים יצרנים משפט ו-`%s` מהוות placeholder לארגומנט הבא, במקרה זה `[i].args[i]`.

המשך מפעיל

switch

פעמים רבים בכתייה מפעיל נטויים למקרים שונים, ובמקרה לבצע אותו עם `switch if,...,else if,...,else` אפשר לעשות זאת עם

```
class ChatterFactory {
    Chatter build(String arg) {
        Chatter chatter = null;

        switch(arg) {
            case "echo":
                chatter = new EchoChatter(new String[] { "what " });
                break;
            case "other chatter":
                chatter = new OtherChatter();
                break;
        }
    }
}
```

היה אפשר גם לאותל `Chatter chatter;` ואז לשים מקרה דיפולטיבי עמו-

```
switch(arg) {
    case "echo":
        chatter = new EchoChatter(new String[] { "what " });
        break;
    case "other chatter":
        chatter = new OtherChatter();
        break;
    default:
        chatter = null;
        break;
}
}
```

אפשר גם לחת בתאוטו קיס כמה ערכים שונים, למשל `"echo"`, `"Echo"` אבל זה חייב להיות hardcoded ולא להכיל איזשהו משתנה (אלא אם כן הוא `final static private`). השינוי הזה תקין החל מ-`Java 14`.

עד על switch ועל scopes

לא ניתן לשנותים בתוך `case` כי בשנתייחס אליהם מחוצה לו, הסקופ החיצוני לא יזהה אותם.

Response to Illegal Requests

```
private String[] getStringsFromUser(String stringsDescription) {
    //prompt user
    System.out.print(String.format("Enter %s (with %s as separator): ", stringsDescription, STRING_SEPARATOR));
    String line = scanner.nextLine();
    return line.split(STRING_SEPARATOR);
}
```

הוספת קש Kushnerim חדש

humanChatter, continue, break

המילה השמורה `continue` תدلג על קטע הקוד שמשמעותה אחרת, אם זו לולאת `for` או יבצע את התנאי של סיום האיתרציה, ויעבור לאיתרציה הבאה. כך למשל אם נרצה שהבוטים ידברו עד שמתקיים `EXIT_STATEMENT` נעשה זאת כך:

```
Scanner scanner = new Scanner(System.in);

String statement = "hi";
for(int i = 0 ; !statement.equalsIgnoreCase(EXIT_STATEMENT) ; i = (i + 1)%chatters.length) {
    String newStatement = chatters[i].replyTo(statement);
    if(newStatement.isEmpty())
        continue;
    statement = newStatement;
    System.out.print(String.format("%s: %s", args[i], statement));
    scanner.nextLine(); //wait for "enter"
}
```

טיפול: המתודה `equalsIgnoreCase` מתעלמת מהבדלים של case.

SilentChatter

אם למשל נרצה בוט ששותק באחוז מסוים של המקרים, ובשאר המקרים אומר איזשהו statement statement געזה זאת עם random באופן הבא:

```
import java.util.Random;

class SilentChatter implements Chatter {
    private String statement;
    private int percentageToSayStatement;
    private Random random = new Random();

    public SilentChatter(String statement, int percentageToSayStatement) {
        this.statement = statement;
        this.percentageToSayStatement = percentageToSayStatement;
    }

    public String replyTo(String statement) {
        if(random.nextInt(100) < percentageToSayStatement)
            return this.statement;
        return "";
    }
}
```

Toddler

Toddler בסיסי

אין מידע חדש, חוץ מהמתודה contains של String שבודקת האם מחרוזת כלשוי נמצאת בתוך מחרוזת אחרת (לא משנה באיזה חלק שלה).

Toddler רקורסיבי

אם למשל נרצה שה-toddler יתנהג בסיכוי של 50% כמו בוט אחר בהתאם לבחירת המשתמש, אז הוא צריך לקבל לבנאי שלו כמו מי להתנהג – בהתאם למה שהוא משתמש ב蹊. במקרה זהה, יוכל לבצע בתוך build בפעולת קרייה רקורסיבית build-�build בתוך המקרה של Toddler באופן הבא:

```
case "toddler":
    System.out.print("enter a phrase the toddler likes: ");
    String likedPhrase = scanner.nextLine();
    System.out.print("enter a chatter name to use if phrase not found: ");
    String chatterName = scanner.nextLine();
    Chatter chatterIfPhraseNotFound = build(chatterName);
    chatter = new Toddler(likedPhrase, chatterIfPhraseNotFound);
    break;
```

Decorator

דקורטור ("מקשט") זו תבנית עיצוב שבה מחלקה מממשת ממשק כלשהו בעל מתודה:

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

ומעבירה קראיות ל-method method של אובייקט אחר המשתמש באותו ממשק:

```
class Decorator implements SomeInterface {
    private SomeInterface innerInstance;

    public Decorator(SomeInterface innerInstance) {
        this.innerInstance = innerinstance;
    }

    @Override
    public void method() {
        innerInstance.method();
    }
}
```

המתודה יכולה להוסיף פונקציונליות נוספת למתודה method ובכך "מקשטת" את הפונקציונליות 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 ...
    }
}
```

פרק 4 – ירושא

4.1 מנגןונים בתכונות מונחה עצמים: ירושא

מבוא והעמסה

העמסה – Overloading

הרעיון של העמסה הוא שניתן לכתב כמה בנים לאותה מחלוקת אבל עם רשימת פרמטרים שונה, וגם כמה שיטות עם אותו שם כל עוד הן נבדלות ברשימה הפרמטרים שלהם.

```
class Foo() {  
    public Foo() { ... }  
    public Foo(int n) { ... }  
    public Foo(Foo other) { ... }  
  
    public void method() { ... }  
    public void method(double d) { ... }  
    public void method(String s) { ... }  
}  
  
class AutoPlaylist {  
    public AutoPlaylist(int startYear, int endYear) {...}  
    public AutoPlaylist(String genre) {...}
```

לעומת זאת, לא ניתן להבדיל פונקציות לפי ערך החזרה – זה יוביל לשגיאת קומpileציה:

```
public void method(int num) { ... } X  
public int method(int n) { ... } X
```

היררכיית ירושא

הכללה וירושא

בין מחלוקות יכולם להיות סוגים שונים של יחסים:

Has-a Relation - Composition

במקרה בו למחלוקת א' יש רכיב מסווג מחלוקת ב'. נקרא גם יחס הכללה. ממומש ב-Java על ידי Data members. למשל לרכיב יש הaga.

Is-a Relation

במקרה בו למחלוקת א' היא סוג של מחלוקת ב', למשל סטודנט הוא סוג של בנאדם. במקרה זה סטודנט יכול את כל התכונות והשדות שיש לבנאדם, אבל לא הפוך. זה ממומש ב-ג'אווה על ידי קונספט של ירושא. למחלוקת היורשת (סטודנט) נקרא sub-class, ולמחלקה העל ממנה ירושים (בנאדם) נקרא .super-class

<pre>/** A person with a name and * a mother */ public class Person { private String name; private Person mother; public String getName() { return this.name; } ... }</pre>	<pre>/* Student: A person with student * ID that can take exams */ public class Student extends Person { private int id; /* Take an exam */ public void takeExam() { ... } ... }</pre>
---	--

המחלקה Student ירשת מהמחלקה Person את כל המתודות שלה, ואפשר להוסף לה שדות ומетодות נוספים.

לאחר יצירת אובייקט Student, אפשר לקרוא על האובייקט למתודות ששייכות ל-Person:

```
Student myStud = new Student( ... );  
  
// Running a method of the parent class  
(Person)  
System.out.println(myStud.getName());  
  
// Running a method of the sub-class  
(Student)  
myStud.takeExam();
```

הערה: לא להתבלבל בין instance-of לבין a-is, האחד הוא מופע קונקרטי של המחלוקת, והשני הוא טיפוס חדש.

היררכיות ירושה

ירושה היא רקורסיבית, מחלקה A יכולה לרשת מחלקה B שירושת מחלקה C.. יש טרנסיטיביות – במקרה של מחלקה A ירוש מחלקה C. כל מחלקה יכולה להיות super-class של מספר לא מוגבל של מחלקות. כל מחלקה בગ'אווה ירושת מהמחלקה `java.lang.Object`, פרט אליה עצמה.

איברים פרטיים של מחלקה האב

מחלקה ירושת לא יכולה לגשת לשדות הפרטיים של מחלקה האב שלה, גם אם הם השdots הפרטיים שלה (למשל אם המחלקה `Student` ירושת מהמחלקה `Person` שבה יש שדה `name`, אז `Student` לא יוכל לגשת ישירות לשדה `name` שלו. שdots המסומנים `public` מאפשרים גישה כמפורט).

```
/**  
 * A person with a name and a  
 * mother  
 */  
  
public class Person {  
    private String name;  
    private Person mother;  
  
    public String getName() {  
        return this.name;  
    }  
    ...  
}  
  
/** A student is a person that has a student ID and can take exams*/  
public class Student extends Person {  
    /** A student has (composes) a student id */  
    private int id;  
  
    /** Take an exam */  
    public void takeExam() {  
        System.out.println(name); // error (name is a private  
                                // field of the parent class)  
        System.out.println(getName()); // a public method  
    }  
    ...  
}
```

נראות ודרישה

Protected Visibility

בדומה ל-`public` ול-`private` ישmodifier שנקרא `protected` ומאפשר גישה למחלקות ירושות (sub-classes). לכן, הדוגמה הבאה תעבוד:

```
/**  
 * A person with a name and a  
 * mother  
 */  
  
public class Person {  
    protected String name;  
    protected Person mother;  
  
    public String getName() {  
        return this.name;  
    }  
    ...  
}  
  
/** A student is a person that has a student ID and can take exams*/  
public class Student extends Person {  
    /** A student has (composes) a student id */  
    private int id;  
  
    /** Take an exam */  
    public void takeExam() {  
        System.out.println(name); // now is works!  
        System.out.println(getName()); // a public method  
    }  
    ...  
}
```

עם זאת, למגדיר `protected` יש גם הרבה חסרונות, כי שדה המסומן `protected` הוא עדין חלק מה-API של התוכנית, בעוד ששדה `private` הוא לא. זה יכול לגרום להסתמכות מסויימת על הימצאותו של שדה, בעוד שלא קיים צורך אמיתי בכך. בוגדול – אם לא חייב להגדיר משוחה כ-`protected` להגדיר אותה `private`.

דרישה

באנגלית Overriding זו הפעולה של ליקחת מחלקה ולרשת ממנה, אבל לשנות התנהגות מסוימת שלה (שינוי מתודה). זה מתרחש על ידי ליקחת חתימת המתודה (פרמטרים, ערך החזרה) ומיושש שהיא חדשה במחלקה היורשת. בשנקרא למתחודה מאובייקט של המחלקה היורשת, יוזץ הקוד מתוך המחלקה היורשת.

המילה השמורה `super` מאפשרת לנו לגשת למתחודה של מחלקה האב מתוך המתודה הדורשת במחלקה היורשת. זה שימושי כאשר נדרש שינוי קטן, ולא לדחוס את כל המתודה. זה יתבצע על ידי `(super.method)` או באמצעות הכנסת ערך לבניית `super`. דוגמאות:

```
/** A person with a name and a  
 * mother */  
public class Person {  
    private String name;  
    private Person mother;  
    public Person(String name){  
        this.name=name;  
    }  
    public String getName() {  
        return this.name;  
    }  
    ...  
}  
  
/** A student is a person that has a  
 * student ID and can take exams*/  
public class Student extends Person {  
    private int id;  
  
    /** A constructor that receives the  
     * student's name and id. */  
    public Student(String name,int id) {  
        // Call parent constructor  
        super(name);  
        this.id = id;  
    }  
}  
  
/** Modify the behavior of  
 * getName() to return the  
 * name twice */  
public String getName() {  
    return super.getName()  
    + " " + super.getName();  
}  
...  
} // end of Student class
```

כמה שיטות foo מופיעות בתוכן?



3

```
class A {
    public void foo() {
        System.out.println("A");
    }
}
class B extends A {
    public void foo() {
        System.out.println("B");
    }
}
class C extends B {
    public void foo() {
        System.out.println("C");
    }
}
```

בנאיםמילה המפתח super ובנייה

קריאה ל-super מתרחשת בכל פעם שאנו יוצרים מופע של תת-מחלקה. הקריאה זו יכולה להתבצע בצורה מפורשת בעזרת הקראיה ל-super, או בצורה לא מפורשת, ואז מאחריו הקלעים תהיה קראיה ל-super בלבד. אם לבניית האב יש פרמטרים – זה יוביל לשגיאה.

דוגמאות:

```
public class A {
    public A() {
        system.out.println("A");
    }
    ...
}

public class B {
    public B(int b) {
        system.out.println("B");
    }
    ...
}
```

```
public class C extends A {
    public C() {
        // No super() – implicit
        // call to A's default c'tor
        system.out.println("C");
    }
    ...
}

C c = new C();
Output:
A
C
```

```
public class D extends B {
    public D() {
        // No super() and no default
        // constructor to B:
        // compilation error
        system.out.println("D");
    }
    ...
}
```

ירושה ופולימורפיזם

Upcasting ופולימורפיזם עובדים בירושה בבדיקה כמו שהם עובדים עם ממשקים.

דוגמאות:

```
public class Animal {
    public String speak() { return "ha?"; }
    public void eat(int calories) {
        System.out.println("Yummy");
    }
}

public class Dog extends Animal {
    public String speak() {
        return "woof";
    }
    public Person getOwner() { ... }
}
```

```
public class Cow extends Animal {
    public String speak() {
        return "moo";
    }
    public void getMilk() { ... }

    public void eat(int calories) {
        System.out.println("YamYam");
    }
}
```

במקרה הבא, איזה קוד יוציא? הקוד של הרפרנס כלומר של החיה?

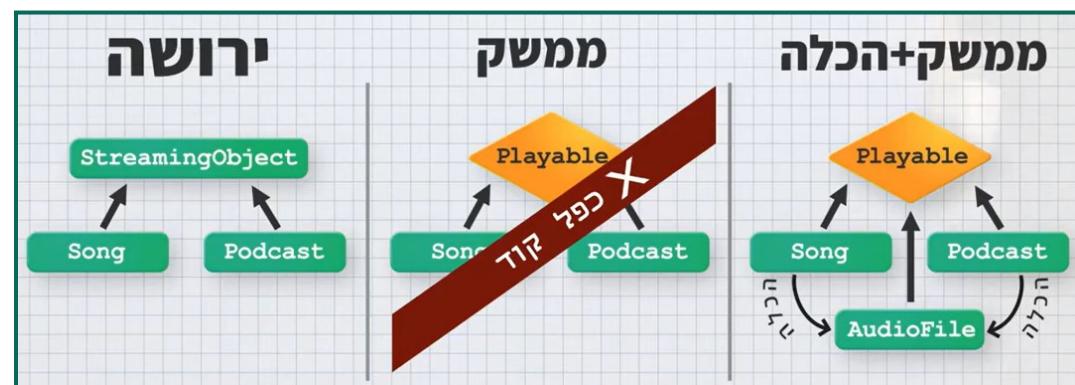
```
Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = myCow;
myAnimal.speak();
```

הקוד שיוציא הוא של הפרה. צריך להסתכל מראש על זה שהוא פקודה חוקית, Animal הוא בעל מתודה speak. בכלליות – השאלה איזה קוד אפשר להריץ נקבעת על ידי הרפרנס (Animal). אבל מה הקוד שיוציא נקבע על ידי התוכן (Cow). ברגע שהשורה (Cow myAnimal.getMilk()) תוביל לשגיאת קומPILEZA, כי למרות שבבסיס Animal myAnimal מצביע לתוכן של פרה, זו לא פקודה חוקית עבור אובייקט מטיפוס Animal.

ירושה לעומת שימוש במשך

דוגמאות ליתרונות של ירושה

יש מספר דרכי לעצב את אותו המימוש:



- **ירושה** – מחלוקת אב ושתי תת-מחלקות. מחלוקת האב הכילה את הקוד המשותף ושתי תת-מחלקות ירשו אותו.
- **משך** – כאן הקוד המשותף הופיע בשתי המחלוקות, מימוש זה גורם לכפל קוד עבור קטיעים שאמורים להישאר זהים בין שתי המחלוקות.
- **משך + הכלה** – משך משותף לשתי המחלוקות מימושו, אבל את הקוד המשותף ייצאנו לחלוקת נוספת נספת ששתה המחלוקות מכילות מהתרשים רואים בклות שעבור המקהלה זהה, המימוש הפשטוט ביותר הוא ירושה. הירשה שימשה כאן גם לשתי מטרות:

- 1) מנענו כפל קוד ויכולנו למחזר קטיעים דומים
- 2) פולימורפיזם, הנחנו ש-`StreamingObject` יהיה בשימוש על ידי הלקוח `Song` וגם על ידי הלקוח `Podcast` בלי לדעת על איזה סוג קונקרטי מדובר.

אם היינו צריכים רק מחזר קוד היה אפשר להסתפק באופציה של הכלה, ואם היינו צריכים רק פולימורפיזם היה אפשר להסתפק במשך. لكن ירושה נועדה למקהלה בו יש צורך בשניהם.

Casting

שימוש ב-up-casting כאשר סוג הרפנס (משמאלי לסימן =) הוא מחלקת האב או המשך של האובייקט הקונקרטי (מימין ל-=).
Up-casting יכול להיות implicit כמו למשל
`Animal myAnimal = new Dog()`
יכול להיות explicit כמו למשל
`Animal myAnimal = (Animal) new Dog()` (בדרא'ב לא נרצה לבתוב כר).

לעומת זאת, down-casting זו הפעולה של לחת אובייקט השיר למחלוקת האב ולהוריד אותו בהיררכיה הירשה בר:

`Animal animal = ...`

`Cow c = (Cow) animal`

במקרה זה נהיה חייבים להשתמש ה-explicit. עם זאת, צריך להיזהר כי הקומpileר לא תמיד ידע לזהות טעויות כמו למשל:

`Animal animal = new Dog()`

`Cow c = (Cow) animal`

וכאן תהיה שגיאת ריצה. (לעומת זאת אם ננסה לעשות `(Cow) new Integer(5)` הקומpileר בן יודע לזהות את הטעות).
ב-down-casting נוכל להשתמש רק אחריו שביצמנו לאובייקט up-casting.

הבעיות בהמרה בלפי מטה

עדיף להימנע משימוש בהמרה מטה מכמה סיבות:

- 1) טעויות בשימוש בה מתגלו לעיתים רק בזמן ריצה, מקשה על דיבוג..
- 2) גמישות – המרה מטה לוקחת קוד כללי והופכת אותו ליותר ספציפי, דבר שאנחנו מנסים להימנע ממנו.

instanceof

האופרטור `instanceof` מאפשר לנו לבדוק עבור אובייקט כלשהו בזמן ריצה האם הוא מסווג מסוים (תקף גם למחלוקת אב וגם למשך).

```
Animal animal = new Cow()
if (animal instanceof Cow) {
```

זה יחזיר true.

לבאורה, זה יכול להגיד לנו מפני שגיאות בזמן ריצה. אולם השימוש בו ברוב המקרים נחטא bad practice.

נסתכל על דוגמת הקוד הבאה:

```

Animal a; Cow c; Dog d;

d = new Dog(); \\ we are creating a new dog that d is referencing to.

a = new Cow(5); \\ this is an implicit up-casting that turns Animal to a Cow.

a.speak(); \\ this is okay, speak is method of animal, "Moo" will be printed.

a = d; \\ this is okay, implicit upcasting and it is legal because a is of type Animal.

a.speak() \\ also okay, "woff" will be printed.

d = (Dog) a; \\ explicit down-casting that will work since d is supposed to hold the type dog.

d = new Cow(3); \\ won't work - compile error. This up-casting is not legal.

d = a; \\ this is implicit down-casting, which is not an option.

c = (Cow)a; \\ this will compile because a is Animal which is super-class of cow, but a is holding
               \\ a dog, and so we cannot make it reference to a cow. Runtime error.

if (a instanceof Cow){
    c = (Cow) a; \\ Even though we won't get here, this isn't recommended.
}

```

toString-equals

בامור, כל מחלקה בג'אווה ירשה מהמחלקה `Object` שיש לה 2 שיטות שימושיות:

- **toString**

להציג ייצוג מחרוזתי של העצם. נctrיך לדרס את המתודה הדיפולטיבית של המחלקה `Object` כך:

```

class Song {
    private String name;
    private String artist;
    private String genre;
    private int year;
    public String toString() {
        return artist + " - " + name;
    }
}

```

ואז בקריאה להדפסה נקבל את הפורמט שבחרנו, לעומת זאת צירוף שמורכב משם המחלקה ותווים נוספים כמו `cd109e7`

- **equals**

להשוות תוכן של שני עצמים ולהציג האם הם שווים. למה לא להשתמש באופרטור `==`? כי בג'אווה האופרטור זהה משווה בין תוצאות בזיכרון, כלומר שני רפרנסים שונים מצביעים לאותו מקום בזיכרון או על עצמים שונים.

```

String oop1 = new String("OOP");
String oop2 = new String("OOP");
oop1.equals(oop2); // returns true
oop1 == oop2; // returns false!

```

ובמקרה של המחלקה `Song` נדרס את `equals` ככזה:

```

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

```

שmagdirah שני שירים בשים אם יש להם את אותו שם וmbozim על ידי אותו אמן, אלה טיפוסים של `String` لكن נשווה אותם עם `equals`, ואם יש להם את אותה שנת פרסום (טיפוס פרימיטיבי שאפשר להשוות עם `==`).

לשים לב, אנחנו מקבלים כפרטטור אובייקט מסוג `Object` ולכן צריך קודם קודם לבדוק שהעצם שקיבלנו הוא `instanceof` של האובייקט שאנו חenso מיפוי לקבל.

ירשות ממשקים

לממשק אפשר להגיד תתי ממשקים באמצעות המילה `extends` בדומה למחלקות. זה שימושי במקרים בהם אנחנו רוצים להגיד כמה סוגי של חזים והנהגיות בעלי מתודות מסוימות. מחלקות המממשות את תת הממשק מחויבות למשם גם את המתודות שמוגדרות בממשק האב.

חבילות

Package זו דרך לחלק את הקוד לחולקות היגייניות שיעזר על שמירות סדר. כמו כן, מסייע במודולריות של הקוד ובהוספת הרשותות. דוגמה לחבילה היא Java Collection Framework שנמצאת בתחום `java.util`. הרכיבים של החבילה (ממשקים, אלגוריתמים, מימושים וכו') שייכים לאוטו ה-Package.

על מנת ליצור חבילה נשתמש במילה השמורה `package` ואז שם החבילה, ואם נרצה להשתמש בחבילה נctrurk ליבא אותה:

```
package pack1;
public class A {
    int packageInt;
}
```

```
package pack2;
import pack1.A;
class B {
    A a = new A();
    System.out.println(a.packageInt);
}
```

אחר שהמחלקה B נמצאת ב-`pack2` והמחלקה A נמצאת ב-`pack1`, אין אפשרות זו את זו וכן הצורך ביבוא.
נשים לב שימושה במחלקה A שנקרה `int packageInt` אין שם מגדר נראות, ולכן הוא בהרשה `default` או `package`.
הדוגמה לעיל תגרום לשגיאת קומpileציה כי A לא נמצא באותה חבילה כמו B, אך B אין גישות להרשותות Package. נתקן:

```
package pack1;
public class A {
    int packageInt;
}

package pack1;
class B {
    A a = new A();
    System.out.println(a.packageInt);
}
```

גם מחלקות ניתן להגדיר ללא `modifier`, וכך המחלקה עצמה היא בהרשות `Package`, כלומר רק למחלקות אחרות שנמצאות באותה חבילה. כך למעשה המחלקות ה-`public` מגדירות את ה-API של החבילה.

סיכום מגדרי נראות

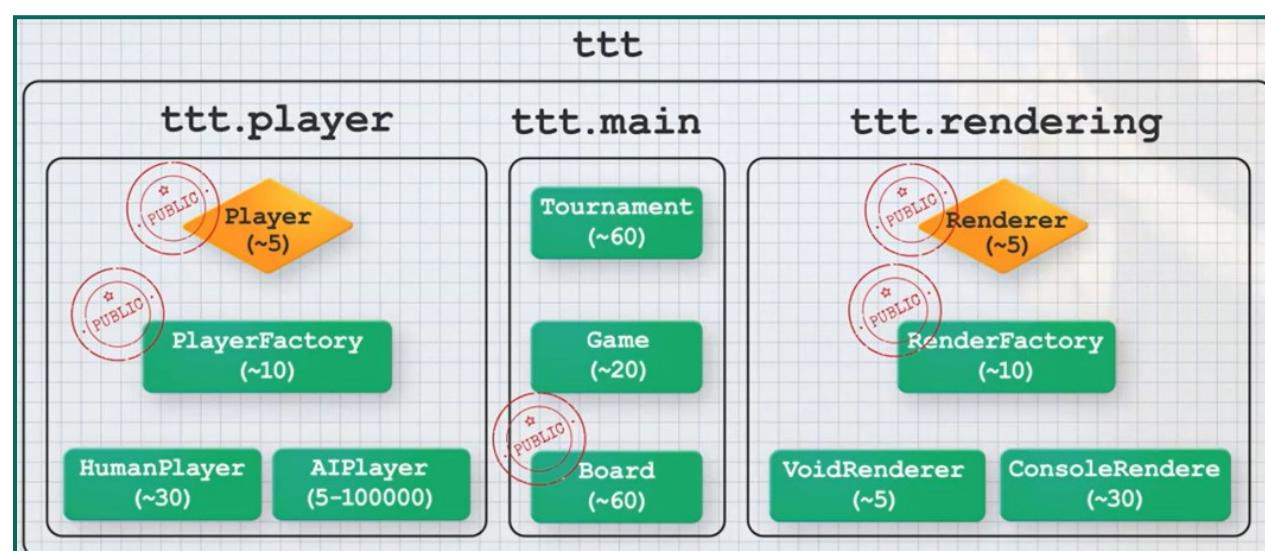
מתודה או שדה יכולים להיות – `private`, `public`, `protected`, רק למחלקות בתוך החבילה).

מחלקות וממשקים יכולים להיות או `public` או `private`, ולא `protected`.

נראות של איברים במחלקה מבוסנת חסומים על ידי המחלקה עצמה. אם המחלקה כוללת בראותות חביבה, לא ניתן לראות אותה מחוץ לחבילה – גם אם היא פומבית!

מודולים

מה קורה בפרויקטים בהם יש עשרות או אלפי מחלקות? למקרים אלו מועד `Module` או בעברית רכיב תוכנה, שהוא אוסף של מחלקות וממשקים. בדומה למחלקה גם למודול יש API מסוימנו והוא חלק מהמחלקות והמשקים שבתוכו. בתוכניות גדולות יותר קודם כל נבצע אנקטולציה למודולים, ובתוך כל מודול נבצע אנקטולציה לטיפוסים שלו. למשל חלוקה למודולים בפרויקט איקס עיגול שלו תראה כר' :



פרק 5 – מנגנוני מחזור קוד

5.1 מנגנונים בתכנות מונחה עצמים: שיטות אבסטרקטיות

מחלקות אבסטרקטיות

מושביצה למחלקות אבסטרקטיות

אם יש לנו מחלקה אב של חיות עם מетодת () `Animal.speak()` שכל תת מחלקה מימושה באופן שונה, () `Cat.meow()`, () `Dog.bark()` – נשאלת השאלה איך מימושים את המטודה במחלקה האב? אם לא נממש מטודה זאת בכלל נפגע בפולימורפיזם ונהפרק את הקוד שלו למסורבל, ואם נממש באופן ריק אנחנו יכולים להוסיף עוד מחלקה, לשוכח מהעובדת צריכה לעשות `Override` למטודה () `speak()` ונשאר עם מטודה שלא עשויה כלום.

מחלקות אבסטרקטיות

הפתרון שיוצע לבעה שהוצגה לעיל היא שימוש **במחלקות אבסטרקטיות**. מדובר במחלקה שמכרידים עליה כך: `public abstract class Animal` ומה שמיוחד בהן הוא שאו אפשר ליצור מופעים שלהן (יגרום לשגיאת קומpileציה). במחלקות כאלה ניתן להגדיר **מטודות אבסטרקטיות**, מטודות שאין להן מימוש:

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

נשים לב שאחרי המטודה אין {} אלא רק ; זה אומר שמחלקה (לא אבסטרקטית) שמעוניינת לרשת את המחלקה `Animal` חייבת למש את המטודה `speak`.

ירושה בין מחלקות אבסטרקטיות

עד על מחלקות אבסטרקטיות

מחלקה אבסטרקטית יכולה לרשת מחלקה אבסטרקטית אחרת. במקרה שמחלקה אבסטרקטית ירושת מחלקה אבסטרקטית אחרת, היא לא נדרשת למש את המטודות האבסטרקטיות של מחלקה האב, אבל היא כן יכולה למשם אם היא רוצה.

מחלקות אבסטרקטיות יכולות להחזיק **data members** ומethods ברגיל, כולל בנאים. ההבדל הוא שלא ניתן ליצור מופעים שלהן. איזו מתי השתמש במחלקה אבסטרקטית?

- 1) מצב שבו אין היגיון להגדיר אובייקט קונקרטי מהמחלקה הזאת (כמו להגדיר `Animal` ביל הסוג שלו - Cat, Dog, ..).
- 2) לבסוף API על סט של מחלקות.

פולימורפיזם של ירושה ומשהקם

הمرة מעלה לטיפוסים גבוהים

הקשר בין אינטראפיסים לירושה מרובה – באינטראפיסים הם לא חלק מהיררכיית המחלקות. מה שמאפיין ירושה זה שכל מחלקה יכולה לרשת רק מחלקה אחת אחרת, בעוד שמחלקה אחת יכולה למשם כמה אינטראפיסים שהיא רוצה. למשל עבור אובייקט של המחלקה הבאה:

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

כל שורות הקוד הבאות תקיןות:

```
MyClass myObj = new MyClass();  
MyParentClass myParentObj = myObj;  
Object obj = myObj;  
Printable myPrintableObj = myObj;  
Clonable myCloneableObj = myObj;
```

במה types יש למופע של המחלקה `MyClass` הבא?

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

✓

6

תשובה

MyClass, MyParentClass, Bable, Printable, Clonable, Object; נסכל

לגביה בחרה בין ממתק לירושה - שניהם מקיימים יחס *a-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*"). בהינתן הבחירה בין ירושה למחלקה למימוש ממתק, נעדיף ממתק.

דוגמאות לממשקים ומחלקה אבסטרקטית מספרית ג'אווה

דוגמה לאינטראפיס בג'אווה היא חבילת-*collections* של ג'אווה, שהיא חבילת מבני נתונים. יש בתוכה אינטראפיס שנקרא *Collection* שהוא *List* *data structure* נלי שיש לו מתודות שונות כמו *size*, *add*, *remove* (עבור גודל מבנה הנתונים). יש גם אינטראפיסים יותר ספציפיים כמו *Number* (עבור מספרים מהאינטראפיס *Collection*), ומיצג מבנה נתונים עם חשיבות לאינדקס. דוגמה נוספת היא המחלקה *Number*:

- **abstract class** *java.lang.Number*
 - A general number class
 - *intValue()*, *floatValue()*, ...
 - Subclasses: *Integer*, *Double*, ...

שיטות בירית-מחלול ופרטיות בממשקים

החל מגרסת 8 של ג'אווה, יש אופציה חדשה בממשקים שמאפשרת לנו לבתוב מימוש לשיטה בבר בממשק עצמו. שיטות אלו כותבים עם המילה השמורה *default* לפני שם השיטה, ומחלקה שמממשת את הממשק יכולה, אף לא חיבת, לדרוס את השיטה:

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

למה זו אופציה? לצורך עדכונים עתידיים לממשקים קיימים, אם החלטנו לעדכן את כל מי שמשמש ממתק מסוים, הינו נאלצים לבתוב מחדש הקוד אצל כל מי שמשמש את הממשק הזה – אם זה לא קורה כי מי שתכנת את המחלקות האלו למשל בבר לא עובד עליון יותר, הקוד פשוט לא יתפרק. סיבה נוספת היא מעמעי נוחות – אם אנחנו מעריכים שהרבה אנשים ירצו להשתמש באבストראקטיות של הממשק בדרך מסוימת, אפשר להוסיף שיטה דיפולטיבית שתעשה זאת.

ונכל גם בממשק לבצע העמשה של השיטה, ובכך לספק למשתמשים גרסה דיפולטיבית אם לא בחרו ממש אותה בעצם:

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

בממשקים ניתן גם למשש שיטות פרטיות שມטרתן להיות עזר לשיטות הדיפולטיות, למשל כדי למחזר קוד. באופן כללי, נשתדל להשאיר ממשקים כמה שיותר נקיים מקום, ברגע שאנחנו מתחילה להריגש צורך גדול מדי בתוכו מתחילה לבתוב מתחודשת בתוך ממתק – צריך לשאול את עצמנו האם זה מקרה שמדובר במקרה לירושה. מחלקותណן בשביל המימוש, וממשקיםណן ביעדו לקחת את המימוש ולעשות לו אבסטראקטיה לרמת החוזה.

מגבילות של ירושה

הבעיה מתחילה כשהשאלה *בש망זנחים אנקפסולציה לטובה ירושה. ירושה היא כל – לא עקרון מנחה*.

5.2 עיצוב תוכנה: הכללה ומחדור קוד

ירושה לעומת הכללה

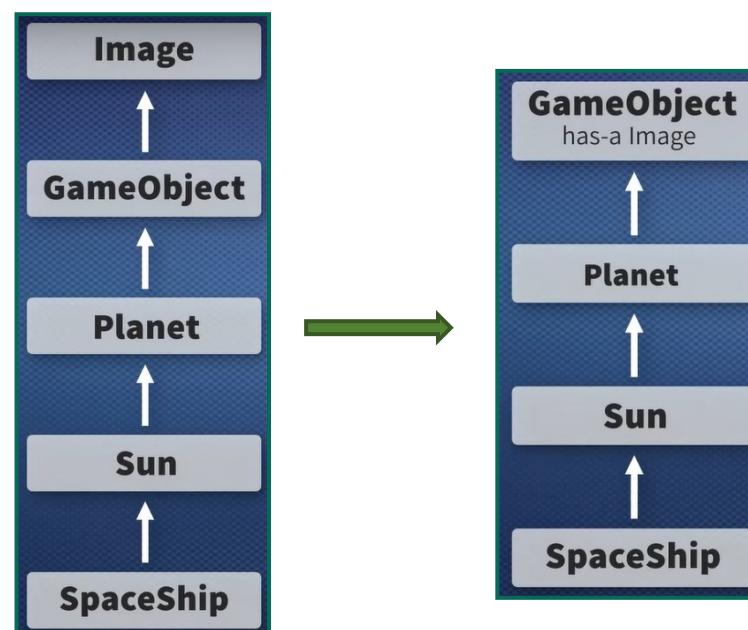
מה הדרך הטובה ביותר להשתמש שוב בפיסת קוד שככבנו?

- **ירושה** – דרך מובנית בג'אווה לשיתוף קוד. יתרונות: ברורה לשימוש, פולימורפיזם, מוגדרת באופן סטטי בזמן קומpileציה (אולי גם חישור?)
- **הכללה** – באנגלית *Object Composition*, שלפיה אובייקט מכל *instance* של אובייקט אחר. יתרונות: מוגדר דינמית בזמן ריצה (אפשר להחליט בזמן ריצה לרשת מחלוקת אחרת), מסיע בשימוש על *single purpose*, ומחלקה יכולה להחזיק במה אובייקטים שהוא רוצה – לא מוגבלת לירושה בלבד.

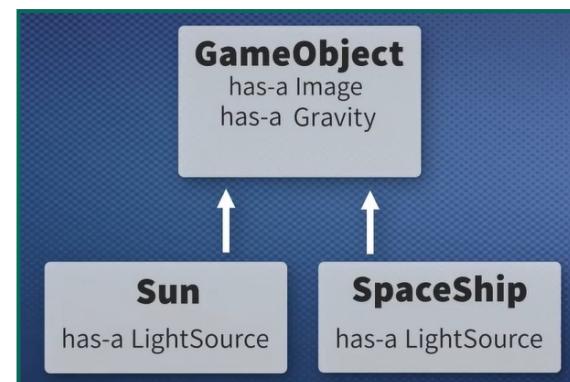
```
public class B {  
    public void foo() { ... }  
}  
  
public class A {  
    private B b;  
    public A(B b) {  
        this.b = b;  
    }  
    public anotherFoo(...) {  
        ...  
        // A uses the foo() code by calling b.foo()  
        this.b.foo();  
        ...  
    }  
}
```

דוגמת שימוש בהבלה על פני ירושה

במקרים בהם זה עדיף, אפשר לשבור קשר של ירושה ולהפיכו אובייקט רלוונטי. למשל אם עבור משחק צריך רק עוקם, במקום ששייטת `Image` מירש ממנה, יותר הגיוני ש מופע של `GameObject` יוביל `Image` וכל המתודות הרלוונטיות יופעלו דרכו.



עקרון זה נקרא Composition over Inheritance, הbhala על פני ירושה. אם נסתכל שוב על התרשימים החדש, נבין שהסיבה היחידה שרצינו `Planet` הוא לצורר גוף בעל בבידה. נחלף אותו ב-`Gravity`. והירושה של `Spaceship` מ-`Sun` לא הגיונית כי ממש לא מתקיים "spaceship is a sun" והסיבה היחידה שרצינו את הקשר הזה הוא כי שניהם מפיצים אור. מכאן מגייע עיצוב חדש:



זה עדין לא עיצוב מיטבי, כי נדמה כרגע שככל שינוי קטן לקוד יגרור שינויים מהותיים בעיצוב התוכנה. לשם כך נועדה אסטרטגיה.

עיצוב תוכנה: אסטרטגיהמבנה העיצוב אסטרטגי

זה מצב שבו מערכת שלנו יש משפחה של אלגוריתמים או התנהלות שמדוברים על מה בעצם המערכת עשויה, והינו רצויים במובן מסוים להפריד אותם מהמערכת עצמה. למשל בתלות בקלט מהמשתמש נוכל לחתך רכיב אחד במקומות האחר וזמן ריצה להחליף בין התנהלותות שונות. דוגמה לכך היא בהתאם לקלט מהמשתמש להחליט איזו שיטה מיען עדיפה (...bubble sort, merge sort, quick sort).

לבעיות מסווג זה נועד `Strategy` design patterns API שזו איזושהי מחלוקת אבסטרקטית או ממשק, והוא יגדיר מה האלגוריתם עשויה. לכל אחת מהדרכים השונות תהיה מחלוקת שתשתמש את ה-`API` זהה. 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>
---	---

נשים לב שיש אלגוריתם שהמערכת צריכה להכיר ולהשתמש בו, אבל במקרה שהוא תכיר את הפרטים שלו בעצמה ותעשה שימוש בקטעי קוד שלא רלוונטיים עליו, היא עשויה סוג של יבוא מחלוקת חיצונית שמממשת API כללי ובעזרת `factory` עם המethode `select` יכולה לבחור מה המיען הרלוונטי עבורה על פי קלט מהמשתמש.

בדוגמה קודם, נשים לב שהיה פחותה הגינוי לשימוש בחלוקת אב שנקראת `Collection` וממנה ירשו מחלקות כמו `Collection` או `MergeSomeCollection`. `QuickSortSomeCollection` כי בוודאות אין בכך קשר של `a-is`. מעבר לכך, לאסטרטגיה יש יתרון של מודולריות – אנחנו רוצים להפריד כמה שאפשר בין אלגוריתמים בקוד לבין המשמש. יתרון נוסף הוא הסתרת מידע – המשמש לא צריך להציג יישורות באיזה אלגוריתם או התנהגות הוא מעוניין לשימוש, זה שהוא צריך לזכור לאחר מכן. אם בהמשך יעלה שימוש חדש ל-`QuickSort` הוא יהיה נועל במסגרת מחלוקת ויצטרך למשוך אותה בעצמו כדי לשימוש בה. יתרון נוסף הוא שינוי התנהגות בזמן ריצה שמתאפשר בעזרת אסטרטגיה לעומת ירושא.

פרק 6 – עוד סוגים API

6.1 מונחה עצמים – שדות סטטיים

שדות סטטיים

מה זה שדה סטטי?

שדה שלא מייצג תכונה של המופע, יש רק עותק אחד שלו עבור כל המופעים של המחלקה. משתנים סטטיים לא `final` הם בדרך כלל רעילים לא מוצלח.

בכל זאת, דוגמה לשימוש:

```
class Citizen {  
    private static double treasury = 0;  
    private double savings;  
  
    public void getPaid(double salary) {  
        double tax =  
            calculateTax(salary);  
        savings += salary - tax;  
        treasury += tax;  
    }  
}
```

כאן למשל חלק מהחישובן של מופע של `Citizen` הולך לקופת המדינה `treasury`. זה לא שלבל מופע של אזרח יש קופת מדינה אחרת, כולל מעבירים מס לאותה קופת וכן הגיוני שהמשתנה הזה יהיה `static`. `Citizen.treasury` ניתן לגשת לממשתנה הזה בעזרת סטטי. `Class variable` קוראים לעתים לממשתנה מחלקה `treasury`.

המילה השמורה `final`

שדות סופיים

המגדיר `final` אומר לממשתנה מקבל ערך רק פעם אחת ברגעיה של התוכנה ואי אפשר לעדכן אותו יותר. הוא לא חייב להיות סטטי, הוא יכול גם לייצג ערך סופי של מופע בודד:

```
class Person {  
    private final int id;  
    private final int birthYear;  
  
    public Person(int id, int birthYear) {  
        this.id = id;  
        this.birthYear = birthYear;  
    }  
}
```

כאן למשל תעוזת זהות של אדם ונתת הלידה שלו לא אמורים להשתנות, וכן עברו אותו מופע הנתונים האלה סופיים `final`, אבל ממשתנים בין מופע למופע שכן לא נסמן אותם `static`.

נקודות חשובות:

1. שדה סופי חייב לקבל ערך או בשורת האתחול או במבנה, ולא בשנייהם.
2. מטרת של שדות מופע סופיים פרטיים – אם הם פרטיים, מפני מה בעצם אנחנו מגנים? מעטינו. זה מנגנון בטיחות עבורנו שבמהלך כתיבת התוכנה לא ביצענו שינוי בממשתנה שאמור להיות סופי.
3. אם נגד (`) new Diary =` `private final Diary`, מה ש-`final` כאן הוא המצביע ליום, כלומר בשום שלב לא נפתח יומן חדש, אבל היומן אליו אנחנו מצביעים מבוגן עוד ישתנה. لكن השדה `final int birthYear`, בעוד שהוא מצביע אליו מצביעים `mutable diary`.

קבועים – קבוע הוא שדה סטטי סופי שמקבל את הערך שלו בשורת ההצהרה.

שאלה – האם השדה (`) new Diary diary =` `public static final Diary diary` קבוע? לא, מצביע לעצם `mutable` לא קבוע קבוע, כי עצמים אחרים יכולים לעשות בשדה זהה שינוי ואחרים יהיו מושפעים מהשינוי.

העשרה – לא ניתן לרשף מחלקה `Math` כי היא מוגדרת {...}. `final` כאשר משמעות `final` כאן היא זו שלא מאפשרת יהושה ממנה, וזה גם הסיבה שלא ניתן ליצור מופעים של `Math` כי הבנאי שלו פרט. באופן דומה שיטה שמוגדרת `final` לא ניתנת לדרישה על ידי מחלקה הבטה.

בעיות במילוטבליות משותפת

בכל שימוש, נרצה להימנע ממילוטבליות משותפת בין הרובה חלקים בקוד, כי קשה לעקוב אחריו הערך שלו במצב זהה. לבן נהפוך שודות שאין הכרח שיינוי מילוטבליים לאי-AMILITBLIIM. עם משתנה פרימיטיבי זה מתבצע פשוט על ידי הוספה `final`.

מה לגבי שודות סטטיים מילוטבליים? זהobar נרצה אפיו יותר קשה לטעוקב, כי כל מופע של המחלקה יכול לשנות את השדה. למשל בדוגמה:

```
Class instance;
instance = new Class();
instance.print(); → 0
instance = new Class();
instance.print(); → 1
```

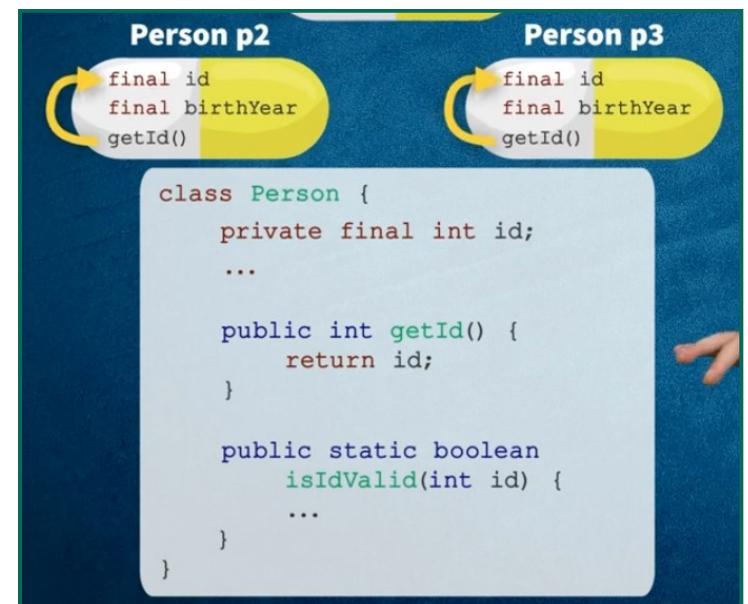
הינו מצפים שיאזדפס אותו דבר, אבל אם מאחרוי הקלעים ההדפסה הראשונה גרמה לשינוי משתנה סטטי, זה מקשה עליו לדbeg ולהבין מה קורה במחלקה מאחרוי הקלעים. איך בכל זאת נוכל לנצל את זה?

אפשר להחליף בຄלות שדה סטטי שמשתieur לכל המופעים במופע של מחלקה אחרת לקבוצה הרלוונטי. למשל בדוגמה של המחלקה `Citizen` אם בהמשך נרצה להוסיף עוד מדינות, יצא שכל מופע של `Citizen` משלם לאוותה קופת מדינה למגוון שם מדינות שונות. נסיף מחלקה `Government` שבו יש שדה מחלקה treasury, ואז כל אוצר יהיה משoir למופע של `Government`.

6.2 מונחה עצמים: שיטות סטטיותמבוא לשיטות סטטיותמהו שיטות סטטיות?

מה עשו במאובן שאני צריך מתודה שלא קשורה למופע בודד של המחלקה? למשל שיטה כמו `isValidId(int id)` שאמורה לקבל תעודת זהות ולהגיד אם היא תקינה בסטנדרטים של המחלקה? לא הגיוני לשירות את השיטה זו למופע של המחלקה, כי אני לא אוצר אדם תקין ואז אצטריך לבדוק אם תעודת זהות של...

שם כך יש שיטות סטטיות, אשר לא משוויכות לאף עצם אלא למחלקה כולה על פי עקרון discoverability, לשימוש את השיטה במקום שבו הייתה מוחפש אותה.



כאן השיטה `(int id)isValidId(int id)` מתייחסת למופע של המחלקה, והשיטה `(int id)getId()` מתייחסת למחלקה כולה. איך מבצעים קרייה למחלקות סטטיות מתוך המחלקה? כמו קרייה לכל שיטה, פשוט משתמשים בשם המתודה. אם היא פומבית ורוצים לקרוא לה מבחן, זה יעשה באמצעות `Person.isValidId(id)`.

שיטה סטטית אין גישה `-this` ולאיברי המופע (וגם אין צורך בכך), אבל כן יש לה גישה לשודות סטטיים של המחלקה.

<input type="radio"/> כשדה מופיע, כי השפה הרצויה תלויה בשחקן.
<input type="radio"/> כשדה סטטי, כדי שהשיטה תוכל לנשנות לשדה.
<input type="radio"/> קבוע (סטטי סופי), כי השפה לא משתנה עבור גרסה נתונה.
<input checked="" type="radio"/> שאלת מכם! <code>printMessage()</code> צריכה להיות שיטת מופיע.

אנו ממחשים משחק שחמט של שני שחקנים שמשחקים על גבי האינטרנט. המשחק מכיל שני מופעים של המחלקה `Player`. השיטה הסטטית `printMessage()` מדפסה הודעה לשחקן, כשפה ההודעה (אנגלית, רוסית וכו') תלויה בערך השדה `language`. איך לדעתם כדי להגדר את השדה?

```
class Player {
    ... definition of "language" ...

    public static void printMessage() {
        switch(language) {
            ...
        }
    }
}
```

סמן את התשובה הנכונה ביותר:

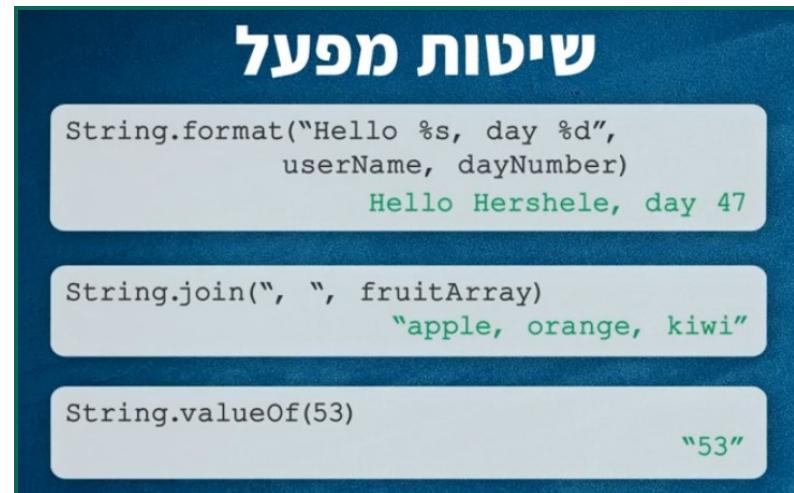
אמרנו שישיטה סטטית נשים תחת המחלקה שהבי' קשורה אליה, אבל אם זה הבי' מתאים למשחק – אפשר לשים אותה גם שם (לא נפוץ, אבל אפשר).

שיטה סטטית במשחק היא שיטה לכל דבר, אבל אין לה ממש קשר לתפקיד של משחק בפולימורפים.

מתי נשתמש בשיטה סטטית?

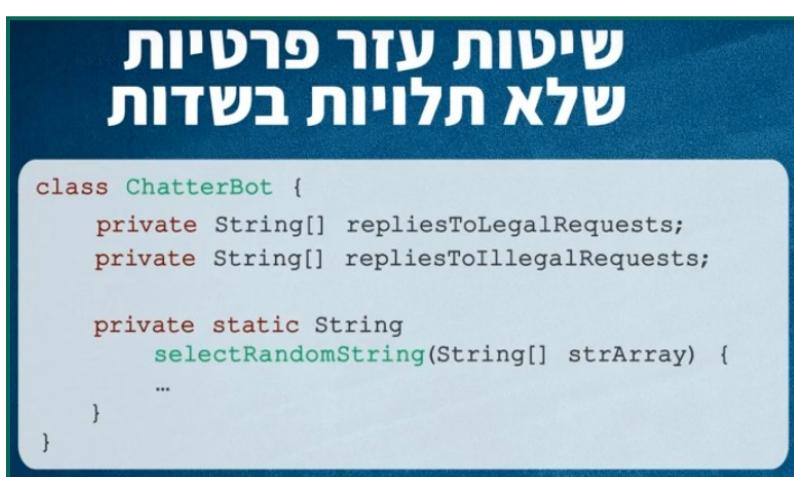
- אחד השימושים הנפוצים של שיטה סטטית פומבית הוא בשיטה שמייצרת מופיע של המחלקה שהיא נמצאת בה, ושיטה זו נקראת "שיטה מפעל".

דוגמא לכך אפשר לראות במחלקה `String`:



למה לא להשתמש פשוט במבנה? פשוט כי ב-API נוח יותר לתתשמות אינטואיטיביים למתחודות, במקום בנאי אחד עם המונח סוג העמסות.

- שיטות עזר פרטיות שלא תלויות בשדות.



באן למשל יש שיטה של המחלקה `ChatterBot` שבוחרת מחרוזת רנדומלית בין זו שהביאו לה בארגומנט (אם היא חוקית, מתוך החוקיות ואם לא אז מתוך הלא חוקיות). למה זה צריך להיות `static`? זה יעבוד גם בלי.. אבל הגישה צריכה להיות אחרת – אם משהו יכול להיות סטטי, עדיף לעשות אותו זהה.

אם אני הופך מתחודה לסטטית, זה מבהיר לקרוא לשיטה לא נוגעת לשדות (וגם לא יכולה לגעת בהם).

אפשר להסתכל על זה כבה `final` עוזר לנו לצמצם את מספר החלקים שנעים בתחום המבונה. `static` עוזר לנו לצמצם את המרחב שבו החלקים הנעים יכולים לעמוד (ולמנוע התנגשויות מיותרות).

- שיטות עזר לשיטות סטטיות – למשל השיטה `chowdah` חייבת להיות סטטית, כי היא נקראת עוד לפני שיש עצמים בתוכנית ולכן אין שדות כלל. אם פונקציית `chowdah` הפכה להיות ארכובה ואני צריך שיטות עזר, הן כולן חייבות להיות סטטיות – כי שיטה סטטית לא יכולה לקרוא לשיטה לא סטטית.

סיכום:



אם יש לנו אוסף של מетодות שלא קשורות למופע ספציפי של המחלקה, נרצה להפוך אותו ל-`static` כי הוא לא קשור לאף מופע.

מגבלות של שיטות סטטיות

הצללה

דברינו על זה שבפולימורפיזם ה-`type` Reference type קובע לנו מה מותר לנו להריצ', וה-`Object type` קובע לנו מה ירצה בפועל. בשאנחנו מדברים על `data members` ועל מетодות סטטיות זה לא מתנהג באותו אופן, ורק הרפרנס קובע. דוגמה:

<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>
---	---

מחלקה B מרחיבה את מחלקה A, ומגדירה מחדש את המשתנה `myInt`. המethode `staticFoo()` במחלקה B היא לא overriding של המethode `staticFoo()` במחלקה A, אלא נקראת shadowing שלה. במקרה הבא:

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

יצרים אובייקט חדש מבוסס על פולימורפיזם, מסתכלים על אובייקט מסווג B דרך רפרנס מסווג A. במקרה זהה המethode תדפיס את הערך 1 ולא 2, כי ירצה `myInt` של ה-`reference`, למרות שהאובייקט הקונקרטי הוא B. באופן דומה המethode `a.staticFoo()` תדפיס "A".
למה זה הגיוני? כי אנחנו מסתכלים על מethodות-`members` שהם סטטיים, שכן אין משמעות לאובייקט הספציפי. במקרה זהה עדיף להשתמש ב-`(().staticFoo()` או `B.staticFoo()`.
אנו שמים לב שהוא מבלבל, ועדיף שלא להשתמש זהה אם אפשר.

6.3 עיצוב תוכנה – תבנית ה-Façade

הציג התבנית: Façade

תבנית העיצוב Façade

מילה מצרפתית שמשמעותה פנים, אבל מגיעה מעולם הארכיטקטורה ומשמעותו צד של בניין.
Façade רלוונטי באשר יש לנו API מורכב שקשה לעבוד איתו, אבל יש מקרים בהם לקוות לא באמצעות ציריכם לדעת את כל המרכיבות של המחלקה, אלא מספיק לספר להם רק חלק קטן ממנה או איזשהו API פשוט יותר.
השימוש Façade זה רעיון פשוט שלוקח מחלקה מורכבת ובונה מחדש מחלקה חדשה שנקרהת Façade עם API פשוט יותר, כבה שהמחלקה עצמה תעשה את העבודה של להכיר את הקשרים ותספק לך דרך פשוטה יותר לעבוד איתו.

דוגמא ל-Façade

נסתכל על מערכת המחלקות הבאה:

<pre>public abstract class Item { ... } public class Pasta extends Item { ... } public class Salt extends Item { ... } public class SaltShaker { public Salt salt(int nTSpoons) { ... } }</pre>	<pre>public class Pantry { public Item get(String item) { ... } } public class Pot { public void boil(int nLiters) { ... } public void add(Item item) { ... } }</pre>
--	---

נניח שאנחנו רוצים להשתמש במערכת כדי להכין פשוטה. הפתרון להתמודד עם מערכת זאת היא לבנות מחלקה שנקרהת Façade שתתמודד עם המרכיבות של המטבח ותוביל את הלוגיקה של הכנת פשוטה.

```

/** 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);
    ...
}


```

המחלקה `Chef` מimplements את ה-`makePasta()` שנקרא `Façade design pattern`. למחלקה זו יש את כל מה שנדרש להבנת פסנה, ויש לה מתחודה אחת פשוטה שנקראת `?Façade`.

- מצד הלוקח, הוא צריך להתמודד עם מערכת פשוטה יותר.
- אם יש שינויים במורכבות המערכת, הלוקחות לא בהכרח מודעים לשינוי (בהנחה שתכננו את מערכת ה-`Façade` כך שלא תושפע מהשינויים). הלוקח לא צריך ללמידה API חדש.
- לא מאבדים שום דבר, מי שרצה להשתמש ב-API המורכב יותר, עדין יוכל לעשות זאת.

Façade ופולימורפיזם

בהתאם בין טיפוסים Façade

Façade מאפשר לנו לחתוט API מורכב ולתת לו "חלון" ל-API פשוט יותר. דוגמה:

```

public class ComplexPlayer {
    public void play(String fileName,
                    int leftvolume,
                    int rightvolume,
                    int bass) { ... }

    ...
}

public interface SimplePlayer{
    public void play(String fileName);
}

public class ComplexPlayerFacade
    implements SimplePlayer {
    private ComplexPlayer player;
    public ComplexPlayerFacade () {
        this.player = new ComplexPlayer();
    }
    public void play(String fileName) {
        player.play(fileName, 50, 50, 0.5);
    }
}


```

יש לנו מחלקה `ComplexPlayer` שבה אפשר לשלוט על עצמת הנגינה בכל צד, עצמת בסים, וממשק `SimplePlayer` שפשוט מנגנת ביליה היכנס להולצות המורכבות. המחלקה `ComplexPlayerFacade` לוקחת את המחלקה המורכבת ומפשיטה אותה לעבוד עם הממשק `SimplePlayer` על ידי שימוש במתודה `play()` עם ערכים דינטטיביים. הקפיצה בין ה-API המסובך לפрост נעשית על ידי מחלקה ה-`Façade`.

פרק 7 – Generics ומבנה נתונים שימושיים

7.1 מבני נתונים שימושיים

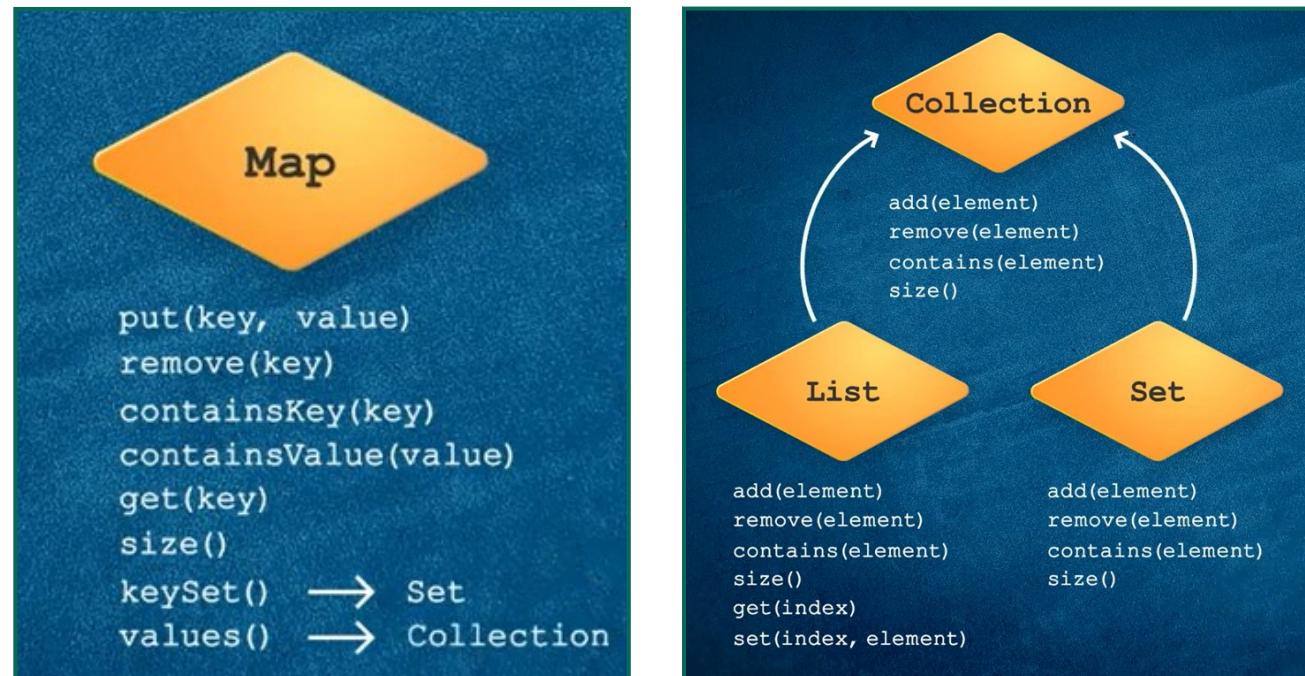
הצגה של מבני הנתונים השימושיים ביותר

היבשות עם Set, Map, List

- List מדבר על רשימה עם סדר, בລומר יש חשיבות מי האיבר הראשון, מי השני... יתכונו בפיזיות בין ערכיהם.
- Set מודל קבוצה מתמטית, בລומר רצף איברים ללא סדר ולא יכול להופיע אותו איבר פעמיים.
- Map מאפשר לנו להגדיר מיפוי בין מפתחות לערכים. לא יכול להיות שאותו מפתח יופיע פעמיים, אבל כן יתכן שני מפתחות ימפו לאותו הערך.

The Collections Framework

סיכון פועלות עיקריות שהמשקדים האלואפשרים למם לעשות:



قطع הקוד הבא לא מתקפל:

```
List list = new List();  
  
if(list.size() == 0) {  
    System.out.println("Being on the worst dressed list is  
better than not being on any list at all!");  
}
```

זה אחר ש-List הוא ממשק, ואנחנו צריכים ליצור מופע ספציפי שלו למשל LinkedList או ArrayList.

שימושים נפוצים לממשקים List, Set, Map

ArrayList

שימוש של List המבוסס על מערך. המחלוקת מחייבת מערך בגודל התחלתי, ובאשר 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.

LinkedList

שימוש של List שהוא רשימה מקושרת. מורכבת מאובייקטים נפרדים המצביעים אחד לשני. למחלוקת רק לתחילת הרשימה ולבסוף הרשימה. הגישה לאיבר לפי אינדקס היא בעבר על כל האיברים מתחילה הרשימה עד האיבר הרצוי. מימוש זה מתאים לשימוש בסיסי עבור מבני נתונים אחרים כמו מחסנית.

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 ומ剔ה של איברים הן יעילות בגלל הגישה הישירה למקום בטבלה המתאים לערך הגיבוב. ל-*java* יש מנגנוןיעיל לטיפול בתנשיות ו גם במקרה של של איברים עם אותו ערך גיבוב היעילות נשמרת.

הערה: לכל אובייקט בא-*java* יש מימוש של הונקציה hashCode במחלקה Object ואפשר גם לדחוס את הונקציה זו כמו שנראה בהמשך.

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 key. ניתן לקרוא עוד בתיעוד הרשמי.

הציגת מנגנון ה-Generics

הjęונספט של Generics מתעסק בינה המחלקות מחזיקות, למשל רשימה של ספרים? של אנשים?Generic ביג'ואה מאפשר לנו להגדיר פרמטר אחד או יותר לכל מחלקה או ממשק. כך בעצם בכל פעם שאנו יוצרים יוצרים Collection שמחזיק טיפוס אחר אנחנו בעצם יוצרים מחלקה חדשה, אבל זה מאפשר בלי שבאמת נשים לב להבדל. מה יקרה אם ננסה להכין ArrayList טיפוס שונה מהטיפוס שהגדרנו ביצורה? שגיאת קומפליציה.

לגביו שלים – נניח שאנחנו רוצים מערך של משתנים פרימיטיביים. אם משתמש במערך, יוכל ליצור מערך [] או, אבל מנגד בשבייל להשתמש ב- ArrayList הטיפוס שלו נדרש להיות Integer. במקרה זה המערך הפנימי בתוך ArrayList לא יהיה של ints אלא של מצביעים לעצמים מסווג Integer, שכל אחד מהם עטוף int יחיד. לכן במקרה של פרימיטיבים, מערך פשוט בן עיל יותר מסוג ArrayList. אבל זה רלוונטי רק לפרימיטיבים ורק במקרים בהם 1) הרבה מהקוד סובב סביב פעולות על המערך, ו-2) בשימוש במערך ספורדי ו/או כאשר לנו צורך ליעיל את הקוד, מבני הנתונים האלויים שקיים אפקטיבית מבחינת ביצועים, בזמן ש-ArrayList מספק יותר נוחות.

השיטה hashCodeהכנסה וחיפוי בטבלת גיבוב

אתה משתמשת של המחלקה Object (ממנה כל עצם ביג'ואה יורש) היא השיטה hashCode של התפקיד המרכזי של השיטה היא שמבנה נתונים יכולים להשתמש בה כדי לקבל ערך מסווג את העצם. למה זה עוזר? באשר נבצע בדיקה על Set וונפעיל את Method contains על mySet. Set לא יחפש בכל הסט, אלא יבודק רק במקומות אליו הם היה ממפה את העצם – אילו היה בתוך הסט.

מימוש השיטה hashCode

למשל במקרה הבא יש התנהגות לא צפוייה:

```
Set<Person> hershelia = new HashSet<>();
Person hersheleThe1st =
    new Person("Israel", 1234, "hershele");
hershelia.add(hersheleThe1st);
System.out.println(
    hershelia.contains(hersheleThe1st) );

Person hersheleThe2nd =
    new Person("Israel", 1234, "hershele");
System.out.println(
    hershelia.contains(hersheleThe2nd)) ?!
```

אם הגדרנו את Method equals לבודק מדינה, ת.ז. שם – הרי שהמתודה תחזיר true עבור hersheleThe2nd והרשלהThe1st – מדוע? זה מאחר ששני העצמים האלה ירשו את המימוש של hashCode מהמחלקה Object שמחשבת לפי כתובות זיכרון. לאחר ששניהם יושבים באותה מקום בזיכרון, לפעמים ימומנו אותו תא בסט ולפעמים לא. כלל אכבר –

אם דרשו את equals – צריך לדרשו גם את hashCode. והמימוש החדש של hashCode על אותם שדות כמו equals. למשל:

```
@Override
public boolean equals(Object other) {
    if(!(other instanceof Person))
        return false;
    Person otherPerson = (Person) other;
    return
        otherPerson.country
        .equals(country)
        && otherPerson.id == id;
}
@Override
public int hashCode() {
    return Objects.hash(country, id);
}
```

יש מתודה שעושה את
הבדיקה בשביילנו

```
@Override
public boolean equals(Object other) {
    if(!(other instanceof Person))
        return false;
    Person otherPerson = (Person) other;
    return
        otherPerson.country
        .equals(country)
        && otherPerson.id == id;
}
@Override
public int hashCode() {
    return country.hashCode() + id;
}
```

סיבוכיות של מבני הנתונים

סיבוכיות במבנה הנתונים

- Magidor פועלות של `get` ו-`set` קבוע, אבל פועלות שבודק האם איבר קיים ברשימה או הוספה של איבר מתבצעות ב- $O(n)$.
- LinkedList מבע הוספה של איברים ב-(1) $O(1)$ אבל כל שאר הפעולות לוקחות $O(n)$.
- HashSet מימוש של קונספט שנקרא Hash Table. פועלות הסרה חיפוש ובדיקה שיעות מתבצעות בתוכלת ב-(1). אין הבטחה לגבי סדר האיברים. לכל איבר יש מצביע למפתח שלו.

	Add	Remove	Get by index	Contains
ArrayList	$O(1)^*$	$O(N)$	$O(1)$	$O(N)$
LinkedList	$O(1)$	$O(N)$	$O(N)$	$O(N)$
HashSet	$O(1)$ avg	$O(1)$ avg	-	$O(1)$ avg

בונוס: בתיבת טיפוסים גנריים

מחלקה גנרטית לנומת מחלקה שעובדת עם Object

Node before Generics	Node with Generics
<pre>public class Node { private Object elem; public Node(Object elem) { this.elem = elem; } public Object data() { return this.elem; } ... }</pre>	<pre>public class Node<T> { private T elem; public Node(T elem) { this.elem = elem; } public T data() { return this.elem; } ... }</pre>

Node before Generics	Node with Generics
<pre>Node n = new Node("hello"); ... String s = (String) n.data();</pre>	<pre>Node<String> n = new Node<>("hello"); ... String s = n.data();</pre> <div style="border: 1px solid red; padding: 5px; margin-left: 20px;"> Notice that you don't need to Respecify the type (String) in the instantiation </div>

.down-casting לא היו צריכים לעשות Generics עם

זה מאפשר לנו לאתר טעויות בקומpileציה, לא בritchah!

Node before Generics	Node with Generics
<pre>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</pre>	<pre>LinkedList<String> list = new LinkedList<>(); list.add("hello"); String s = list.get(0); list.add(new Integer(5)); // Compilation error</pre>

Node before Generics	Node with Generics
<pre>public class Node { private Object elem; private Node next = null; public Node(Object elem) { this.elem = elem; } public Object data() { return elem; } ... }</pre>	 <div style="position: absolute; top: 0; left: 0; width: 100%; height: 100%;"> <p>Not the same meaning of <T></p> </div> <pre>public class Node<T> { private T elem; private Node<T> next = null; public Node(T elem) { this.elem = elem; } public T data() { return elem; } ... }</pre>

סוגי שגיאות

תמיד נעדיף לתפוס שגיאות במה שיותר גבוה בהיררכיה זו. שגיאות קומפילציה מגנות עליהם והן מאפשרות לנו להבין מה הבעיה איפה היא נמצאת. שגיאות שתוכנה מזזה, אנחנו יודעים מה קרה, ולעתים אבל לא תמיד גם איפה זה קרה. באגים ידועים – אנחנו לא יודעים לבדוק מה קרה (אלא רק למה זה הוביל), וגם לא יודעים איפה זה קרה. באגים לא ידועים – לא יודעים עליהן בכלל, כי אנחנו לא מודעים אליהם.

שינוי סוג השגיאה

באגים לא ידועים – זה תחום חדשני טטימי, כדי להגדיל את מספר התרחישים שהתוכנה רצתה עליהם כדי להגיע לכיסוי במה שיותר גדול ובטקווה להמיר את סוג הבעיה לבעיה מסדר שלישי או סדר שני. שיטות דומות אפשר לישם גם עבור סדר שלישי ושני. איך גורמים לקומפイルר לזהות יותר בעיות? כלומר להמיר בעיה מסדר רביעי/שלישי/שני לבעיה מסדר ראשון? אזהרות של הקומפイルר הן בדרך כלל מאוד חשובות. אפשר גם להשתמש בשפות סטטיות בהם מצהירים על סוג המשתנים (כמו Java לעומת Python) והדבר האחרון הוא השלמת קוד חכמה.

Assert

אפשר להשתמש ב- `assert` כדי לקבל תנאי, ולהקritis את התוכנית אם הוא לא מתקין. כאשר מרים את התוכנית אפשר לבחור האם להתעלם מ- `asserts`. השימוש בה מטבח לצרכים פנימיים, לא נרצה שהשימוש בה יקritis את התוכנית אצל לקוחות.

מבוא לחריגות

חריגה – זו הودעה שימושה השتبש, על ידי שימוש במתודה שנתקלה בעיה ואז צריכה להחזיר אחרת הודעה למי שקרה לה. בג'אווה `exceptions` הם `objects` והודעות שהמתודות הללו יזרקו למי שקרה להן יהיו אובייקטים של ג'אווה.

מהצד שזרק exceptions

למשל בקוד (`new ListException()`) `throw` זורקים אובייקט מסווג `ListException`. מה קורה בשיש שגיאה? גם אם יש קוד אחרי השגיאה, הוא לא יירוץ, אין שם ערך החזרה מהפונקציה. כאשר אנחנו מוסיפים ל-API של מתודה את המילה השמורה `throws` אנחנו אומרים שבפונקציה המתודה זו יכולה לזרוק את סוג/חריגת/ות שציגנו. כאשר אנחנו זורקים שגיאה אנחנו צריכים לטעד ב-`javadoc` איזה סוג שגיאה מරוקת ובעקבות מה.

מהצד שקורא למתודה הבועיתית:

```
try {
    //get index from user
    int element = list.get(0);
    ...
} catch(ListException e) {
    // handle list errors
} catch(OtherException e) {
    // handle other errors
}
// Rest of program
```

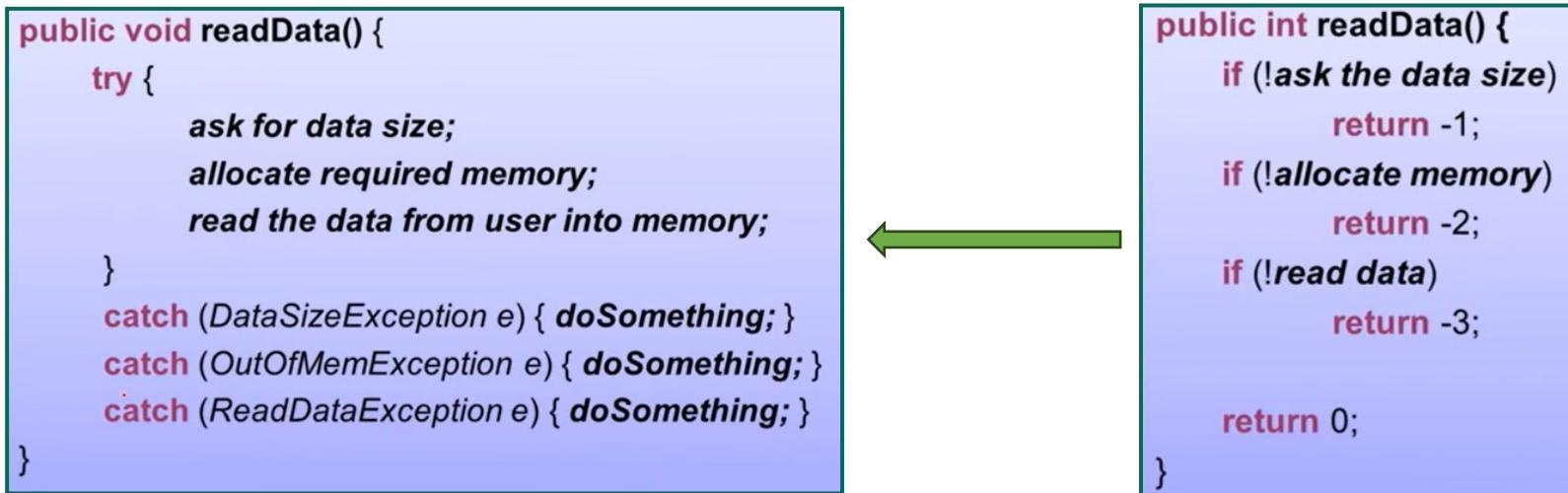
אנחנו מנסים לקרוא לפונקציה בתחום בлок `try`, ואז תופסים סוג שגיאות שונות. כאשר נכנס לבlok ה-`catch` הרלוונטי, תפנסו אובייקט למשל מסווג `ListException` והגדרנו את השגיאה בשם `e`. אפשרות נוספת היא להציג חריגה נוספת זורקת חריגה מאותו טיפוס של המתודה שבה היא משתמשת, ובכך "לגלגל" את האחריות הלאה לפונקציה הבאה שקרוות לה.

```
public void foo() throws ListException() {
    //get index from user
    int element = list.get(0);
    ...
}
```

חריגות ש קופצת מ-main

כאשר יש חריגה בקוד של `main` ה-`exception` מזרק אלינו היוזרים כי אין מישחו אחר שיתופס את השגיאה.

мотיבציה לשימוש בחיריגות



קיבלנו הפרדה בין החלק בקוד שמבצע את המשימה אם לא הייתה שום תקלת, לבין החלק שאחראי לטפל בשגיאות.

גם בקריאה מורכבות לפונקציות נקלט קוד נקי יותר:

Without Exceptions:	With Exceptions:
<pre> public boolean foo_n() { // Some foo_n() code if (error) return false; // Some more foo_n() code return true; } </pre>	<pre> foreach 1 < i < n: public boolean foo_i() { // Some foo_i() code if (!foo_{i+1}()) return false; // Some more foo_i() code return true; } </pre>

```

public void foo_n throws SomeException() {
    // Some foo_n() code
    if (error)
        throw new SomeException();
    // Some more foo_n() code
}

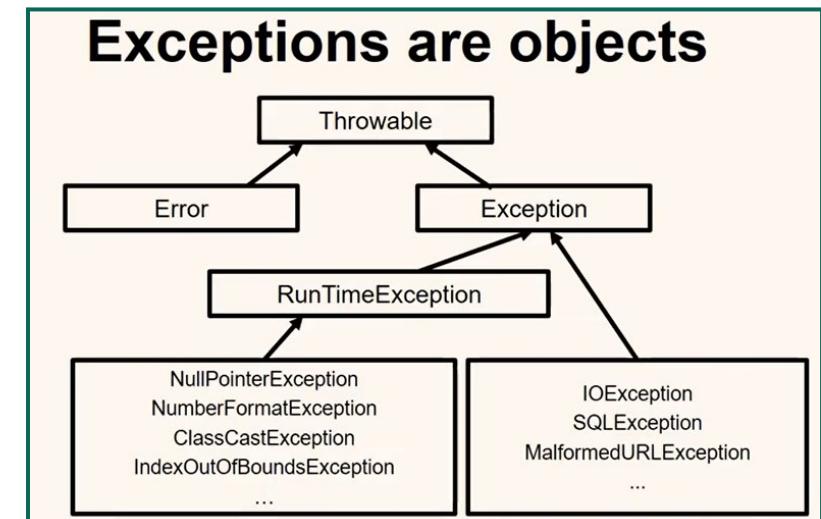
foreach 1 < i < n:
public void foo_i throws SomeException() {
    // Some foo_i() code
    foo_{i+1}();
    // Some more foo_i() code
}

```

כל המתוודות בין 1 ל-n לא מתעסקות בלבד בבדיקה האם הקוד שלו שעד או לא, אלא פשוט יקרו למtodoה הבאה וימשיכו להריצ. אם הייתה שגיאה, הקוד ישבור בכוונה שקופה למtodoה הראשונה.

סוגי חריגות

Checked and Unchecked Exceptions



- במעט ולא נתעמק בקורס. שגיאות של מערכת הפעלה למשול.
- נקרא לסוג זהה `Unchecked Errors` והם בדרך כלל נובעים מבאגים בתוכנה שלנו.
- נקרא `Checked Errors` והן בדרך כלל נובעות מקלט לא נכון מהמשתמש.

Checked

ירוש מהמחלקה `Exception`, ובובאים משגיאה בשימוש של המשתמש. הרבה פעמים ספציפיים לתוכנית מסוימת. במקרה של שימוש בשגיאה כזו נהייה חייבות להשתמש ב-`try/catch`. הצד שקורא למtodoה שזרקת `Checked Exception` או לעשות `throws statement`.

Unchecked

בדרך כלל נובעים מטעות של המתכנת ולא מטעות של המשתמש, כמו למשל גישה ל-pointer `null` או אינדקס גדול מדי ברשימה. יכול להופיע בכל סוג של תוכנית. لكن במקרים האלה אנחנו מוחתרים על הדרישה לציין `throws` או להתמודד עם השגיאה, אבל אם השגיאה לא טריוויאלית – מומלץ לעשות זאת בכלל אופן.

יצול ההיררכיה של חריגה

סיבה נוספת להשתמש בחיריגות היא היכולת לאחד ביחיד סוגים דומים של חריגות ולהפריד ביניהם במידת הצורך:

```
class ListException extends Exception{...}

class EmptyListException extends ListException {...}
class InvalidIndexException extends ListException {...}

long get(int index) throws ListException{
    if (list.isEmpty())
        throw new EmptyListException();
    if (list.size() <= index)
        throw new InvalidIndexException();
    //...
}
```

כלומר המתודה זורקת שגיאות מסווג `ListException`, אבל בתוכה נזרקות תת-השגיאות (שיורשות ממנה) בהתאם למה שקרה בפועל. זה גם עקרון של פולימורפיזם. הטיפול בשגיאה הזאת גם יכול לנצל את העקרון, ולבחר לטפל במקרה מיוחד או במקרה כללי ללא קשר לאיזה סוג פרטי זורק:

```
try {
    long l = myList.get(5);
} catch (EmptyListException e) {
    // handle empty lists
} catch (InvalidIndexException e) {
    // handle invalid index
}

try {
    long l = myList.get(5);
} catch (ListException e) {
    // handle any list error
}
```

סיכום הסיבות להשתמש בחיריגות:

- 1) ליצור הפרדה בין הקוד שמבצע פעולות לבין הקוד שמתפל בשגיאות.
- 2) דרך לפעוף שגיאות במעלה מחסנית הקריאה.
- 3) לאחד סוגים שונים של שגיאות ולהפריד סוגים אחרים.

Lambdas & Callbacks – פרק 8

8.1 ג'אווה סטרים – למבודת

מחלקות מקומיות ואנונימיות

לטהילך של יצירת מ-`int` את `Integer` קוראים Boxing, וההפך – לחתת מותך ה-`wrapper class` את הטיפוס הפרימיטיבי נקרא Unboxing. מחלוקת בთוך שמויפה בתחום שיטה נקראת מחלוקת מקומית או `Local Class`.

```
public class Lambdas {  
    public static void main(String[] args) {  
        List<Integer> list = List.of(1,2,3,4,5,6,7,8);  
  
        class IsEvenIntPredicate implements IntPredicate {  
            @Override  
            public boolean acceptNumber(int num) {  
                return num % 2 == 0;  
            }  
        }  
  
        processNumbers(list, new IsEvenIntPredicate());  
    }  
}
```

אבל בהרבה מקרים זה עדין לא הצורך הכני מינימלי שנדרצה, כי לעיתים נדרש רק רפרנס אחד של השיטה ההז, ובמקרה זה ג'אווה מציעה לנו את הפתרון של **מחלקה אונימית**. למחלוקת אונימית יש בנאי ריק, ולא נתאר את השם של המחלוקת, ויוצרים מופע שלה באמצעות סוג של Upcasting.

```
public class Lambdas {  
    public static void main(String[] args) {  
        List<Integer> list = List.of(1,2,3,4,5,6,7,8);  
  
        IntPredicate predicate = new IntPredicate() {  
            @Override  
            public boolean acceptNumber(int num) {  
                return false;  
            }  
        };  
  
        processNumbers(list, predicate);  
    }  
}
```

בלומר `IntPredicate` זה ממשק, ואנו יכולים ליצור מופע שלו עם `new` בתנאי שימושם את המתודה שמצוינה במשק.

למבודת וממשקים פונקציונליים

למבודת

בתמונה לעיל, אם יש לממשק `IntPredicate` רק שיטה אחת, אז מיותר לציין את שמה, ואת ערך ההחזרה שלה, ואת מגדר הנראות.. והאם צריך `new`? לאט לאט נשארים עם כמעט שום דבר שצריך לכתוב, זה בגודל למבודה.

```
IntPredicate predicate = num -> num % 5 == 0;
```

נשארכו עם הביטוי הקצרני הזה שאומר: יש לנו משתנה `predicate` שהוא רפנס מסווג `IntPredicate`, כלומר נעשה עליו Upcasting מMOVUP של מחלוקת אונימית שemmמשת את `IntPredicate` וממשת את השיטה `acceptNumber` אשר מקבלת `num` וממירה אותו לביטוי הבוליאני `num % 5 == 0`.

אם כך, כבר לא צריך להזכיר על המשתנה כי משתמשים בו רק במקום אחד, וכך מקבלים:

```
processNumbers(list, num -> num % 5 == 0);
```

ממשקים פונקציונאליים

מה הם התנאים לממשק שאפשר למשה על ידי ביטוי למבודה? לממשקים באלה יש שם – **ממשקים פונקציונליים**, כי בסופו של דבר הם לא סובבים סביר מחלוקת אלא סביר פונקציונליות בודדת, ולכן נדרש מהם שתהיה להם רק שיטה אחת שצריך לממשק. לממשק זה נשים את התגית `@FunctionalInterface`, שבדומה לתוויות `@Override` לא מדרשת מבחינת הקומפיאילר, אבל היא עוזרת לקריאות.

אבל, אם לממשק יש מתודה עם מימוש דיפולטיבי (שכחוך – כתובים אותו כבר בתחום המשק), זה בסדר, כי אנחנו לא נדרש מMOVUP שemmמש את המשק להגדיר איך הוא מMOVMS את המתודה. למשל:

```

@FunctionalInterface
interface IntPredicate {
    boolean acceptNumber(int num);
    default void foo() {
        boo();
    }
    private void boo() {}
}

```

java.util.function

למענה כל הממשק שהגדכנו קודם נמצא בברכ-`java.util.function.Predicate`.
בsein מופיע גם `consumer` שביצם לוקחת את כל מי שעובר תנאי מסוים ומבצעת עליו פעולה שנבחר, למשל להדפיס את המספר עם רווח:

```

public class Lambdas {
    public static void main(String[] args) {
        List<Integer> list = List.of(1,2,3,4,5,6,7,8);
        processNumbers(list, num -> num % 4 == 0, num -> System.out.print(num + " "));
    }

    private static void processNumbers(
        Iterable<Integer> numbers,
        Predicate<Integer> predicate,
        Consumer<Integer> consumer) {
        for(int num : numbers) {
            if(predicate.test(num)) {
                consumer.accept(num);
            }
        }
    }
}

```

רפרנסים לשיטות

מה קורה אם אני רוצה בתוכה הלמבדה לעשות חישוב מעט יותר מורכב? למשל אם המספר ראשוני ולא רק אם הוא מתחלק ב-4 ללא שארית? זה כבר מתחילה מעט לאבד את המטריה של למבודה. נשים לב ש-`Predicate` מצפה לקבל פונקציה שמקבלת מספר ומחזירה `boolean`, וזה הרו בדיק מה שהשיטה שאנו חנכו מחפשים צריכה לעשות. במקרה הזה נגדיר שיטה בשם `isPrime` ונכתוב באיזה מחלוקת היא ממוקמת, בך:

```

public class Lambdas {
    public static void main(String[] args) {
        List<Integer> list = List.of(1,2,3,4,5,6,7,8);
        processNumbers(list, Lambdas::isPrime, num -> System.out.print(num + " "));
    }

    private static void processNumbers(
        Iterable<Integer> numbers,
        Predicate<Integer> predicate,
        Consumer<Integer> consumer) {
        for(int num : numbers) {
            if(predicate.test(num)) {
                consumer.accept(num);
            }
        }
    }
}

```

נשים לב שלא כתבים `Lambda.isPrime` כי זה דומה מדי לקרוא לפונקציה, ומה שקרה כאן זה לא קריאה כי `isPrime` הוא מוחזר `boolean` אבל מaptive שנשלח פונקציה ולא `boolean`. לכן משתמשים באופרטור : שזה מצביע לפונקציה.

עבשו, נשים לב שהפרמטר השלישי הוא `consumer` כתום מצפה לקבל פונקציה שמקבלת פונקציה ומחזירה כלום, אז אפשר פשוט לשלוח לה את המתודה `:print`:

```

public class Lambdas {
    public static void main(String[] args) {
        List<Integer> list = List.of(1,2,3,4,5,6,7,8);
        processNumbers(list, Lambdas::isPrime, System.out::println);
    }

    private static void processNumbers(
        Iterable<Integer> numbers,
        Predicate<Integer> predicate,
        Consumer<Integer> consumer) {
        for(int num : numbers) {
            if(predicate.test(num)) {
                consumer.accept(num);
            }
        }
    }
}

```

זה נראה דומה, אבל יש כאן שתי הצבעות לשיטות מסוגים שונים. הראשונה זו שיטה סטטית והשנייה שיטה מופע. חוץ `print` היא לא שיטה סטטית, היא שיטה של האובייקט `this`. בשני המקרים זה עובד על ידי שימוש באופרטור : : פעם אחת אנחנו אומרים באיזה מחלוקת נמצאת השיטה הסטטית, ופעם אחרת אנחנו אומרים באיזה אובייקט השיטה נמצאת.

נניח שיש לנו אובייקט בשם `factory` שנותן לקרוא ממנה לשיטה עם החתימה הבאה:

```
MyClass create(String type)
```

מה מה הבאים יתאפשר?

נקודות (עם ציון) 1/1

`Supplier<MyClass> supplier = factory::create;`

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

`Consumer<String> consumer = factory::create;`

`BiFunction<String, MyClass> biFunc = factory::create;`

`Supplier<MyClass> supplier = factory.create();`

`Supplier<MyClass> supplier = factory.create;`

`Supplier<MyClass> supplier = Factory::create;`

הסבר:

`Supplier<MyClass> supplier = factory::create;`

Won't compile because a supplier expects to get no parameters, but `factory.create` expects to get a `String`.

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

This is exactly what `Function` expects to get, an input: `String` and an output: `MyClass`

`Consumer<String> consumer = factory::create;`

`Consumer` returns `void`, while the method `create` returns a `MyClass` object.

`BiFunction<String, MyClass> biFunc = factory::create;`

A `BiFunction` takes 2 arguments, while the method `create` expects only one.

`Supplier<MyClass> supplier = factory.create();`

This is a try to call the method `create` without any parameters, while it expects to get a `String`.

`Supplier<MyClass> supplier = factory.create;`

This is a try to access `create` as if it was a field of the object `factory`.

`Supplier<MyClass> supplier = Factory::create;`

This is trying to access the method `create` as if it is a static method of the class `Factory` (which doesn't even exist in this context).

Effectively Final

משתנה שימושים בו בביטוי למבודה (שהוא לא פרט של הפונקציה אבל בן מוכר על ידי השיטה החיצונית ולבן אפשר להשתמש בו בimboda) נדרש שהוא יקבל ערך אחד והוא לא ישנה שוב, לא משנה אם כתוב עליו final או לא. אגב, גם אם הוא משתנה אחר שכבר השתמשו בו בביטוי למבודה, Java לא מאפשר לנו להריץ את התוכנית. יש כאן פרצה, כי אפשר בעקרון להשתמש במצביע שהוא סופי, לאובייקט קונקרטי שכן משתנה.

8.2 **עיצוב תוכנה: Callbacks**Callbacks

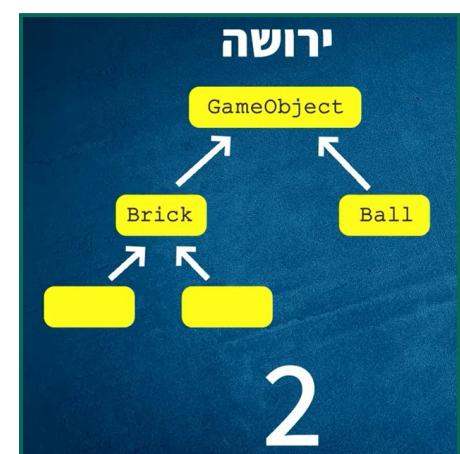
הכוונה לפרטט שהתיפוס שלו הוא פונקציה. זו פונקציה שמיישה אחר שלח לנו כדי שנರיץ אותה מתי שנŻטורך בהתאם לדרישה.
למשל נוכל לבקש מאובייקט במשחק להפעיל פונקציה באופן הבא:

```
GameObject obj = new GameObject(...);
obj.renderer().fadeOut(1,
    ()->windowController.closeWindow() );
```

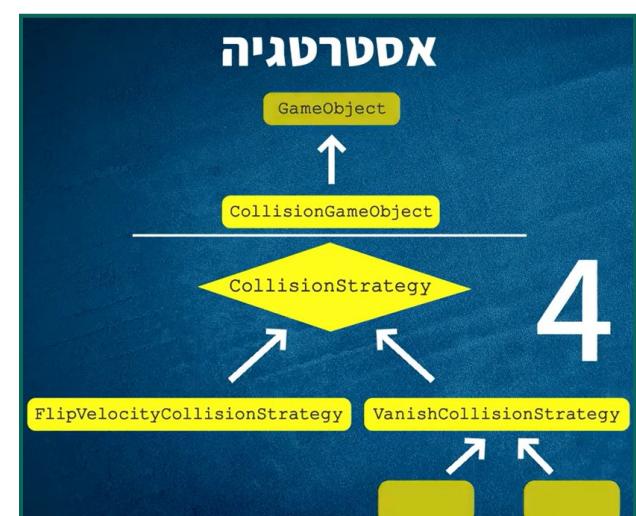
השיטה fadeOut מקבל זמן עד לתחלת ה-fadeOut ופונקציה לביצוע בסיום ההשיה. כאן שלחנו לה למבודה שלא מקבלת כלום ולא מחזירה כלום, וסגורת את חלון המשחק בסיום ההשיה. בגלל העובדה שהממודה לא מקבלת כלום ולא מחזירה כלום היה אפשר גם לכתוב כאן windowController::closeWindow, אבל זה מקרי ולא בהכרח יראה כך תמיד.

Callback Strategiesירושה לעומת אסטרטגייה: בחינה מחודשת

כשמיישנו את המשחק של שבירת לבנים, היינו לנו כמה גישות אפשריות להסתכל על האובייקטים במשחק. גישה ראשונה באמצעות ירושה:



גם הבחירה וגם הלבנה ירושה מהמחלקה GameObject, ועבור כל לבנה עם התנוגות מסוימת נגדיר מחלוקת שתיריש ממנה. וכך נדרשנו להגדיר 2 טיפוסים, ועוד טיפוס נוסף לבנייה חדשה שנוסף.
אפשרות נוספת היא באמצעות אסטרטגייה:



תהייה מחלוקת כללית שתיריש מ-GameObject שתקרא CollisionGameObject, שתתקבל איזה CollisionStrategy ותתמשכו באותו בתעת התנגשות. וכך נדרשנו להגדיר 4 טיפוסים חדשים, ועוד טיפוס נוסף לבנייה תחת אסטרטגייה.

Callback Strategy

```
class CollisionGameObject extends GameObject {
    private BiConsumer<GameObject, Collision>
        collisionCallback;

    // constructor
    // setter for collisionCallback

    @Override
    public void onCollisionEnter(
        GameObject other,
        Collision collision) {

        super.onCollisionEnter(
            other, collision);
        if(collisionCallback != null)
            collisionCallback.accept(
                other, collision);
    }
}
```

בأن מופיע יימוש ל-`CollisionGameObject` בצורה שמקבלת `BiConsumer` – ממשק פונקציוני שמקבל שני פרמטרים ויש לו שיטה בשם `accept` שמקבלת אותם ולא מחזירה כלום – בדיק מה שאנו צריכים עבור `onCollisionEnter`.

Callback Strategy

```
CollisionGameObject brick =
    new CollisionGameObject( ... );

brick.setCollisionCallback(
    (other, collision) ->
    gameObjects().removeGameObject(brick) );
```

ובשוו איך נראה הקשר בין האובייקטים?

Callback Strategy



זהו, המשק של האסטרטגיה התחלף במשק פונקציוני מובנה, והמחלקה שמסממת את האסטרטגיה התחלפה בбиוטי למבדה. אבל איך נגדיר אסטרטגיה חדשה בגישה זו? באמצעות ביטוי למבדה אחר, באמצעות `method reference` עם האופרטור `::`, ותמיד יוכל להשתמש גם במחלקה שמסממת את המשק הפונקציוני.

Callback Strategy

יתרונות

- שינוי בזמן ריצה
- גמישות
- ריבוי אסטרטגיות
- קוד קצר וברור
- מודולריות

API צריך להיות מצומצם, אבל לעיתים לשם הנוחות עדיף לתת עוד כלים שהופכים את השימוש בעצם של המחלקה לקלים יותר. אבל, יש לו מהירות אם בעצם יש הרבה אסטרטגיות או callbacks האתחול שלו ייקח לא מעט שורות קוד, בצורה שנראית שהיא כבר קצת מושתלת על המטרה של המחלקה שבה הוא הוגדר.

לשם כך אפשר לפתח מחלקה כמו בדוגמה הבאה:

```
public class SpecializedObj {
    public static Class<?> create() {
        //initialize strategy 1
        //initialize strategy 2
        specializedObj = new Class(strategy1,
            strategy2);
        specializedObj.method4(
            () -> {
                ...
            });
        specializedObj.setCallback( () -> ... );
        specializedObj.addStrategy(new Strategy3());
        return specializedObj;
    }
}
```

כלומר כל האתחול של העצם י יצא למחלקה.

פרק 9 Wildcards, Erasure – 9

Intro To Generics 9.1

מה זה ?Generics

הפשתה מעל טיפוסים לא פרימיטיביים (מחלקות/מערכות). יש שני שחקנים – המחלקה והפרמטר. למשל <E> Collections במקורה זה Collections או המחלקה, ו-E הפרמטר. מטרת ה-Generic הוא להגדיר type safety, שהופך את הקוד שלנו ליותר חסין לבאגים.

שגיאות טיפוס

- `String s1 = new Integer();`

Compile-time type error

את השגיאה הזאת הקומpileר ידע לזהות בזמן קומPILEציה

- `Object o = new Integer(5);`
- `String s2 = (String)o;`

run-time type error

השגיאה הזאת תאותר (במקורה הטוב) בזמן ריצה.

Generics יעדתו לנו להפוך את השגיאות לסוג הראשון, שקל יותר לזריהו. תוכנית שמודדרת להיות **type safe** היא תוכנית שבה שגיאות טיפוס מתגלות בזמן קומPILEציה ולא בזמן ריצה.

כך למשל, כאשר משתמש-ב-Collection למשול עבור Collection של איברים מסווג Stamp (מחלקה שיצרנו), נגיד `<Collection<Stamp>` ואז לא יוכל בטעות להוסיף איברים שאינם stamp לתוכו האוסף (למרות שבאופן כללי Collection יכול להכיל טיפוסים שונים), ואז כשרצה לעשות איטרציה על איברי האוסף אנחנו יכולים להיות בטוחים שככל האיברים בתוכו הם מאותו סוג (להימנע שימוש-ב-instanceof ובדיקות שונות).

Writing Generic Classes 9.2

בתיבת מחלקה גנרטיב

Node before Generics

```
public class Node {  
    private Object elem;  
    public Node(Object elem) {  
        this.elem = elem;  
    }  
    public Object data() {  
        return this.elem;  
    }  
    ...  
}
```

Node with Generics

```
public class Node<T> {  
    private T elem;  
    public Node(T elem) {  
        this.elem = elem;  
    }  
    public T data() {  
        return this.elem;  
    }  
    ...  
}
```

בכל מקום בו היה לנו Object החלפנו בפרמטר T.

Node before Generics

```
Node n = new Node("hello");  
...  
String s = (String) n.data();
```

Node with Generics

```
Node<String> n = new Node<String>("hello");  
...  
String s = n.data();
```

נשים לב שימוש-ב-Generics חסר לנו את ה-down casting שהואינו צריכים לעשות.

השימוש בפרמטר עובד גם בצורה תורשתית, אפשר בכך המחלקה הכלכלית להגדיר משתנה שימושה באותו טיפוס, לדוגמה:

Node before Generics	Node with Generics
<pre>public class Node { private Object elem; private Node next = null; public Node(Object elem) { this.elem = elem; } public Object data() { return elem; } ... }</pre>	<pre>public class Node<T> { private T elem; private Node<T> next = null; public Node(T elem) { this.elem = elem; } public T data() { return elem; } ... }</pre>  <p>Not the same meaning of <T></p>

פעם אחת הגדרנו את `<T>` בתוך הפרמטר שבו אנחנו רוצים להשתמש, אבל בתוך המחלקה מדובר בבר בטיפוס עצמו אליו `next` מצביע. דגש- או אפשר לרשות בין טיפוסים באופן הבא:

- This means, for example, that `LinkedList<String>` doesn't extend `LinkedList<Object>`
 - `LinkedList<Object> myObjList = new LinkedList<String>(); // Compilation error`

אפילו ש `String` מ- `Object`.

Wildcards 9.3

מה זה Wildcards?

נותן לנו בסיסי פולימורפיים ל-`Generics`. באמצעות הגדירה של `List<?>` במקרים בהם אנחנו לא יודעים או שלא אכפת לנו מה הטיפוס הספציפי של הפרמטר. מה בכל זאת אנחנו צריכים לעשות שימושים בשימושים ב-`<?>`:

- אנחנו לא יכולים להניח כלום על האובייקטים שהרשימה מקבל, מלבד העובדה שהם יירושים מ-`Object`.
- אנחנו לא יכולים להכניס שום אלמנט לרשימה בלבד וזה, כי אנחנו לא יודעים מה יכנס לשם. לכן נוכל רק לשולף אובייקטים או להוסיף אותם.

מתי זה שימושי?

במקרים שבהם מספיק לדעת שהאובייקטים הם מסוג `Object`:

Object is enough
<pre>void printList(List<?> c) { for (Object e : c) { System.out.println(e); } } public void removeAll(List<?> list) { Iterator<?> e = list.iterator(); while (e.hasNext()) { e.remove(); } }</pre>

מבצעים פעולה הדפסה כי לכל אובייקט יש מתודת `toString()`. `e` או מחיקה של כל הרשימה.

במקרה בו מביצעות פעולות שאן ברמת הרשימה ולא האובייקט שבתוכה:

Even object isn't needed
<pre>public boolean isEqualSizes(List<?> c1, List<?> c2) { return c1.size() == c2.size(); }</pre>

אם נרצה להציג את הרשימה כבה שיכלן להוסיף אליה רק `null`, נוכל לבתוב כך:

```
/** This method does not add anything to c1 (except null), and only retrieves Objects from it. */  
public void addOnlyNullMethod(List<?> c1) { ... }
```

נכלי להחזיר טיפוסים שונים בהתאם למקרה:

Return a list of unknown parameter type

```
private List<?> generateListFromUserInput(String str) {  
    switch (str) {  
        case "String": return new LinkedList<String>();  
        case "Integer": return new LinkedList<Integer>();  
        default: return new LinkedList<Object>();  
    }  
}
```

במקרה זה אנחנו לא יודעים מראש מה נחריר, אבל אנחנו חייבים לציין את זה בחתימת המתודה, שכן אפשר להשתמש ב-`List<?>`.

אלטנטיבות ל-Wildcards

למה שלא משתמש למשלב-`List<Object>`? אם היינו רוצים ליצור רשימה של מחרוזות, אנחנו נהיה מוגבלים רק לרשימות של אובייקטים, זה מבהיר רשימה מוקורת של מחרוזות היא לא רשימה מוקורת של אובייקטים (אינוריאנטיות).

למה לא להשתמש ברשימה שהיא לא-גנרטית? כי זה לא `List<?>` מתחורי הקלים יש טיפוס שלא ידוע לנו, ולכן הוא.type-safe.

List<?>

- למה אי אפשר להוסיף אליו כלום?

מאחר שאנחנו לא יודעים מה הטיפוס האמתי מאחורי-`?` לכן אי אפשר לבנות רשימה אובייקט שיטאים לכלום.

- למה אנחנו יכולים לשולף ממנו רק `Object`?

אנחנו לא יודעים כלום על הטיפוס שהוא משווה בוודאות יורש מ-`Object` (מחלקה, ממתק, מערך..)

- רק פרנסים יכולים להשתמש ב-Wildcards

- A concrete object must have an *actual* parameter

- `LinkedList<?> c1 = new LinkedList<String>();` // Legal

- `LinkedList<?> c2 = new LinkedList<?>();` // Illegal

- up-casting עובד ביחס עם Wildcards

- Wildcards and up-casting work together

- `LinkedList<T> implements List<T>`

- `LinkedList<String> implements List<String>`

- `List<?> is applicable to List<String>`

- `List<?> c3 = new LinkedList<String>();` // Legal

שימוש לב למשברים בין `List` ל-`LinkedList`

שימוש בירושה בתוך Wildcards

- As generic classes are **invariant**, generic parameters cannot be up-cast

- `LinkedList<Animal> myList = new LinkedList<Dog>();` // Compilation error!

- In order to up-cast, we must use **wildcards**

- `LinkedList<? extends Animal> myList = new LinkedList<Dog>();` // Now it works!

זה אומר לנו שיש ל-`LinkedList` מכיל פרמטר גנרי, אבל אנחנו יודעים שהוא עומד לרשות מחלוקת `Animal`. כך יוכל לעבור על רשימה בזאת ולהשתמש בפקודות שאנחנו מכירים מכך למשהו שיורש מחלוקת `Animal`. זה עובד באופן דומה גם עם ממתקים, גם שם משתמש ב-`extends`.

עוד נקודות לזכור:

- Can be initialized with any parameter that **extends** `Animal` (**including** `Animal`)

- `myList = new LinkedList<Animal>();` // Ok

- `myList = new LinkedList<Dog>();` // Ok

- `myList = new LinkedList<Creature>();` // Compilation error

- Can only add **null**

- `myList.add(new Animal());` // Compilation error

- `myList.add(new Dog());` // Compilation error

כלומר:

(1) לא יוכל להגדיר את הרשימה בר' שתקבל את טיפוס האב שלו (Creature).

(2) יוכל להוסיף רק null.

אבל מצד שני, בשנשלוף איבר מתוך הרשימה:

- Can retrieve up to animal

```
• Animal a = myList.get(0);           // Ok
  • Creature c = myList.get(0);         // Ok
  • Dog d = list.get(0);               // Compilation error
```

אפשר להגדיר אותו בתור הטיפוס שלפי הוגדרה הרשימה – Animal, או בלטיפוס גבוה יותר כמו Creature.

Erasurement 9.4

מה זה Erasure

הדרך שבה Generics ממומשים ביג'ואה. Erasure מוחק את כל הטיפוסים הgenerics בשלב הקומpileציה, בעצם זה תהליך שבו כל מה שהמלכנו לא לעשות באופן יידי מבוצע באופן אוטומטי (כמו אתחול רשימה בלי ציון של הטיפוס שהוא מכילה, או לבצע down-casting). זה מאפשר לקובץ חדש וישן לעבוד אחד עם השני. דוגמה:

Generic Node	Code after Erasure
<pre>public class Node<T> { private T elem; public Node(T elem) { this.elem = elem; } public T data() { return this.elem; } ... }</pre>	<pre>public class Node { private Object elem; public Node(Object elem) { this.elem = elem; } public Object data() { return this.elem; } ... }</pre>

מה לגבי type-safe

הקוד שנותר לא חשוף למתקנת. הקוד נשאר type-safe וקריא גם לאחר תהליכי Erasure ועדין חשוב להשתמש ב-Generics כדי לוודא שהקוד שלנו יהיה type-safe לפניה שאנחנו מרכיבים אותו, זה לא משתנה לאור ה-Erasure.

- What is the output of the following code?

- ArrayList <String> l1 = new ArrayList<String>();
- ArrayList<Integer> l2 = new ArrayList<Integer>();
- System.out.println(l1.getClass() == l2.getClass());

- true

- getClass() returns “ArrayList” for both lists

זה קורה ברגע תהליכי Erasure, כי מאחורי הקלעים 1 ו-2 איבדו את המאפיין של הטיפוס שהם מכילים, ושניהם פשוט נחשבים ArrayList. מואתיה סיבה אין טעם להשוות instanceof או לבצע down-casting.

קוריאנטיות יצורת בעיות

ציינו ביחידות מקודם ש-Generics הם אינטראנטיים (כלומר רשיימה של Objects הם לא אבא של רשיימה של String). במערכות זה לא המצביע כי הם קוריאנטיים, אם מחלקה Child יורשת ממחלקה Parent, אז [] Child הוא בן של [] Parent. לכן, במקרה הבא תעלת בעיה שתתגלה רק בזמן ריצה:

- Consider the following code:

- String[] strArray = new String[10];
- Object[] objArray = strArray; // Allowed due to covariance: String extends Object
- objArray[0] = new Integer(5); // Allowed – objArray is an array of objects, and // Integers are objects

- The last line will result in a runtime error

לבאורה לא ביצענו כאן באופן אקטיבי down-casting ולכן קשה לדחות שקיימת פה בעיה. בעקבות זה ומאחר ש-Generics נועד למנוע שגיאות טיפוס, לא ניתן להשתמש ב-List גנרי:

- For this reason, and since generics were designed to prevent type-errors, it is illegal to define an array of generic objects

- List<String>[] list = ...; // Compilation error

האם Generics משפיע לי על הקוד?

לא! זה פשוט משנה את הדרך בה אנחנו כתבים את הקוד, אבל בפועל הואעובד באותה צורה.

קוד לא גנרי עדין יכול להשתמש בספריות גנריות.

נקודה לשים לב אליה – קוד שנכתב עם Generics הוא **לא כללי יותר**, במובן מסוים הוא אפילו **ספציפי יותר**.

פרק 10 Regular Expressions – 10

Intro to Regular Expressions 10.1

מה הם ביטויים רגולריים?

תבנית שאפשר להשווות אותה ולבזוק אותה מול טקסט, ולהגיד האם התבנית מתאימה לביטוי. כל פיסת טקסט יכולה להתאים או לא להתאים לביטוי רגולרי. אם זה כן מתאים, אפשר לשאול כל מין שאלות כמו אילו חלקים מתאימים, אילו חלקים מתאימים לאילו חלקים וכדומה.

סינטקס בסיסי של ביטויים רגולריים

Char	Usage	Example
a,b,c,...	Regular text	abc matches abc
.	Matches any single character	.at matches cat, bat, rat, 1at...
[...]	Matches any single character of the ones contained	[cbr]at matches cat, bat, rat.
[^...]	Matches any single character except for the ones contained	[^bc]at matches rat, sat..., but does not match bat, cat.
[a-z]	Matches any character in the range a-z Also works for A-Z, 0-9, and in the negative form (with ^)	- [f-l]aaa matches faaa, gaaa,...,laaa - [^a-f]aaa matches gaaa, 5aaa, &aaa, but does not match aaaa, caaa,
...		...

- טקסט רגיל כמו abc הוא בעצם ביטוי רגולרי.
- מסמן כל תו שהוא.
- [...] מסמןתו בלשונו מבון האפשרויות שמצוינו בין הסוגרים.
- [...] שלילה של מה שמעל.
- [^...] טווח תאים בטוחה, עובד גם בגרסת השלילה עם ^.

במתים

Char	Usage	Example
*	Matches zero or more occurrences of the single preceding character	- .*at matches everything that ends with at: at, hat, 123\$_&treat... - <[^>]*> matches <...anything...>
+	Matches one or more occurrences of the single preceding character	0+123 matches 0123, 00123, 000123...

- הביטוי <[^>]*> אומר בעצם שאינו רצאה > פתוח, ואז כל תו שהוא לא תו סגורה, ובסוף תו סגור >.
- + מתייחס לכל הופעה של מה שהוא לפניו, כמה פעמים שנרצה. לא מתייחס לביטוי 123 כלומר חייב לפחות פעם אחת. אם נרצה תמייה ב-0 נשים במקומות 0#123.

איך למשל נקבע Regex שיאפשר לכתוב את המילה rabbit עם הכפלה של כל אות כמה פעמים שנרצה? r+a+b+b+i+t+.

נשים לב שעល ה-t בגל שהיא מופיעה מראש פעמיים, ורק על אחת מהן מספיק להגיד שהיא יכולה להופיע לפחות פעם אחת או יותר ואין צורך על שתיהן.

Search Structure 10.2

גישה חיפוש ביטויים רגולריים

נראה לדוגמה את הביטוי "Now is the time" עם התבנית "+[a-z][a-z]"

- 1) חיפוש על כל הקטל - במקרה זהה כמובן נכון, כי הביטוי מכיל גם N וגם רווחים שלא נכללים בטוחה z-a.
- 2) חיפוש על כל תות מהירות, במקרה זהה נמצא התאמה עבור ws ועוד עבור time, is, the, time, is וזה נכון (ונגמרה המחרוזת).

דוגמא: איך ניתן לרשום את המילה rabbit עם הקדמה שכוללת את כל האותיות (גדלות או קטנות) ובמילה עצמה נאפשר שכל האותיות יהיו גדולות או קטנות פרט ל-t בסוף? כמו למשל: MyRabbit, HISRABBiT, RabBit.

תשובה: t[iI][A-Za-z]*[Rr][Aa][Bb][Bb][Ii]

চৰৰত কোৰ্সিভ

আপুনি হাতে ব্যবহৃত কোনো অক্ষরের মধ্যে একটি পৰিশেষণা। এই পৰিশেষণাটো আল্টৰনেটিভ। যদি একটো অক্ষর নথি হয়ে থাকে তবে এটো পৰিশেষণা কোনো ফল দেব।

Predefined Character Classes

.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\r\f\v]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

উদাহৰণ: ব্যৱহৃত কোনো অক্ষরের মধ্যে একটো পৰিশেষণা কোনো ফল দেব। যদি একটো অক্ষর নথি হয়ে থাকে তবে এটো পৰিশেষণা কোনো ফল দেব।

মাত্ৰ একটো অক্ষর নথি হয়ে থাকে তবে এটো পৰিশেষণা কোনো ফল দেব।

Boundary Matchers

^	The beginning of a line
\$	The end of a line
\b	A word boundary
\B	A non-word boundary

- অপৰ্যাপ্ত অক্ষরের মধ্যে একটো পৰিশেষণা কোনো ফল দেব। যদি একটো অক্ষর নথি হয়ে থাকে তবে এটো পৰিশেষণা কোনো ফল দেব।
- মাত্ৰ একটো অক্ষর নথি হয়ে থাকে তবে এটো পৰিশেষণা কোনো ফল দেব।
- মাত্ৰ একটো অক্ষর নথি হয়ে থাকে তবে এটো পৰিশেষণা কোনো ফল দেব।
- মাত্ৰ একটো অক্ষর নথি হয়ে থাকে তবে এটো পৰিশেষণা কোনো ফল দেব।
- মাত্ৰ একটো অক্ষর নথি হয়ে থাকে তবে এটো পৰিশেষণা কোনো ফল দেব।

অক্ষরের মধ্যে একটো পৰিশেষণা কোনো ফল দেব। যদি একটো অক্ষর নথি হয়ে থাকে তবে এটো পৰিশেষণা কোনো ফল দেব।

অক্ষরের মধ্যে একটো পৰিশেষণা কোনো ফল দেব। যদি একটো অক্ষর নথি হয়ে থাকে তবে এটো পৰিশেষণা কোনো ফল দেব।

- Assume X represents some pattern
 - X may be a **single character**, a **range of characters** or an **expression wrapped by parentheses**
 - $X\{n\}$ X occurs exactly n times
 - $X\{n,\}$ X occurs n or more times
 - $X\{n,m\}$ X occurs at least n but not more than m times
 - $X?$ X is optional (it occurs once or not at all) $\Leftrightarrow X\{0,1\}$
 - X^* X occurs zero or more times $\Leftrightarrow X\{0,\}$
 - X^+ X occurs one or more times $\Leftrightarrow X\{1,\}$
- These are **postfix operators** (coming **after** the operand)

שלוש האפשרויות הראשונות נועדו לציין טווח של במות הפעמים, למשל:

$a\{3\}$ → a appears exactly 3 times.

$b\{2,\}$ → b appears twice or more.

$c\{4,9\}$ → c appears at least 4 times, but no more than 9.

$d\{,6\}$ → d appears no more than 6 times.

$r\{1,2\}a\{1,2\}b\{2,4\}i\{1,2\}t\{1,2\}s?$ → the word rabbit where each word can be duplicated at most 1 more time, and there's a possible "s" in the end.

סוגי כמתים

כמתים חמדניים

מנסה לחפש כמה שהוא רק יכול, וחוזר אחורה אם הוא צריך (mbzutg backtracking) כמו למשל:

{,n} X {,n} X+ X* X? X

כמתים מינימלייסטיים (reluctant)

מתנהגים ההפך מהם. כדי להגיד להם לחפש את המינימום בטווח שנთנו, נוסיף עוד סימן שאלה כר:

?{,n} X {,n} ? X+ ? X* ? X? ? X

כמתים רכושניים (possessive)

מחפשים כמה שהם רק יכולים, מבליל לבצע backtracking. כדי ליצור במת דה, נוסיף +:

+{,n} X {,+} X++ X*+ X?+ X

דוגמאות:

נתונה המחרוזת "xgaxxxxxga"

1) הכתת החמדן ga^* . מתחילה לחפש כל קטע שהוא (*). אך ימשיך וימשיך כי הכל מתאים, עד שיגיע לסוף המחרוזת. עבשו הוא צריך לחפש ga אבל הוא בסופו. במקרה זה, הוא יודע לבצע backtracking. יתקדם אות אחרת ויש לו ברגע a , זה לא מתאים. יתקדם עוד אות אחרת ועבשו יש לו ga – מתאים למה שהוא מחפש. לכן יחזיר את כל הביטוי, כלומר [8:0].

2) הכתת המינימלייסטי $ga^?$. מתחילה לחפש את המינימום האפשרי עבור *. וזה המחרוזת הריקה, לכן סיים עם שלב זה. מחפשים ga אבל מצאנו א, שכן ברגע זה נכשל. מתקדם עוד אות ואז יש gx , עדין נכשל. נתקדם עוד אות ועבשו יש לנו את המחרוזת gax שעונה על התנאים – והיא המינימלייסטית. אחר כך ימשיך לחפש וימצא גם את המקטע $gaxxxx$. לכן יש 2 הצלחות [2:0] או [3:8].

3) הכתת הרכושני ga^+ . מתחילה לחפש *. על המחרוזת, ומאחר שהכול מתאים הוא ימשיך וימשיך עד שיגיע לסוף המחרוזת. בשונה מהחמדן, הוא לא יעשה backtracking (טיפ של לזכור את זה: הוא רכושני, מה שהוא לוקח להחזיר). לכן הוא פשוט יכשל ויחזר על כל החיפוש `false`.

Greedy vs. Possessive

מתי משתמש בחMAND? כשאנו רוצים לעבור על טקסט כמו `toingXXX` הרכושני תמיד יכשל, כי אם ננסה `toing++[z-a]` נבחן כי ה-`toing` יכול בחלק של `[z-a]` ומאחר שררכושני אין `backtracking` הוא לא ידע לחזור אחרה ויכשל. לעומת זאת החMAND יצליח במקרה זה (וגם המינימליסטי).

מתי משתמש ברכושני? Computer science logins בדוגמה 67125.

הMAND עבר טקסט זה והוא `"[z-a][z-0]+[z-a]"` עם המחרוזת `abcdefgab`. החMAND במקרה זה יהיה בזבזני, כל הקלט יתפס עבור החלק `+[z-a]` ועוד הוא יגיע לסוף, ובגלל שביקשנו לחפש גם ספרות יתחל לחזור אחרה, בשבוףועל אין טעם. הרכושני לעומת זאת, יש לה " `"[z-a][z-0]+[z-a]"` יהיהiesel בהרבה, כי הוא לא ינסה לחזור אחרה.

לסיכום, משתמש בחMAND כדי שיפפה מסויימת בין החלק הראשון לחלק השני (כמו עם ה-`toing` לעומת `z-a`). כאשר חפיפה, עדיף להשתמש ברכושני.

Greedy vs. Reluctant

מתי משתמש בחMAND לעומת במינימליסטי? שניהם רלוונטיים רק במקרים שהרכושני לא רלוונטי, אחרת משתמש בו.

מה שאנו צריכים לשאול את עצמנו זה כמה צפוי להיות אורך החלק הראשון לעומת השני?

אם החלק הראשון צפוי להיות קצר, עדיף להשתמש במינימליסטי. אם הוא צפוי להיות ארוך, משתמש בחMAND.

למשל אם נחפש את התבנית `toing+[z-a]` כאשר אנחנו צופים קלט כמו `"toingxxxxxxxxx"` חבל להשתמש במינימליסטי, כי אחרי כל תוו נבדוק אם מגע אחריו `toing` ונגלה שלא. עדיף לחתוך עם החMAND בכל מה שאפשר, ואז לחזור אחרה רק עם `toing`.

דוגמה לקרה בו עדיף את המינימליסטי הוא התבנית `"xxxxxx+z+a"` כשהאנו מצלמים שהקלט יראה כמו `"xxxxxxa"`.

More on Regular Expressions 10.4

Backreferences

נניח שאנחנו רוצים לחפש אותן שמות פעמיים ברצף כמו ... , CC, bb, aa מה לגבי `{2}{Z-zA-a}`? זה לא עובוד. כי הוא יזהה גם Z'a ולא בהכרח זוג אותיות זהות.

Capturing groups

סוגרים עושים capture לכל דבר שמתאים לחלק של התבנית. דוגמה: `(* [0-9]) (* [Z-zA-a])`

אם נמצאה הצלחה, 1) מחזיק את החלק שמתאים לאותיות - 2) מחזיק את החלק שמתאים למספרות. אפשר בהמשך לגשת לבב אחד מהחלקים. בק 1(`(Z-zA-a)`) מתחאים לאות כפולה כמו במילה `letter`.

אם נרצה למשל לתפוס מבנה כמו `rabbit` שלפני ואחרי המילה מופיעה בדיקת הספרה, נחפש התבנית: `1\b(rabbit)\d+`

איך עובדת פתיחת סוגרים ?

מספר הסוגרים מתבצע לפיו זמן הפתיחה שלהם, דוגמה בבייטוי הבא:

`(()_2 A) (_3 B _4 C)`

`\1 = ((A)(B(C))),`
`\2 = (A)`
`\3 = (B(C))`
`\4 = (C)`

איך כתבים נסוכן בייטויים רגולריים

בתיבת בייטויים פשוטים בכלל האפשר

בייטויים רגולריים אפשריים לנו קיצורים שמקלים על הכתיבה. {2,1}{ab} עדיף על `bb|ba|ab|b|aa`.

בתיבת בייטויים עליים

- בייטויים רגולריים יכולים להיות בשימוש אלףים ועשרות אלפי פעמים בקוד, אפילו שיניינים מינוריים בהם יכולים להיות עליים יותר. באופן כללי, בישלון יקר יותר מהצלחה, "כמעט הצלחה" היא היקרה ביותר. לכן נרצה להקטין את מרחב החיפוש ולפודא שנכשלנו בשלב כמה שיותר מוקדם. גם אם אנחנו יודעים שצריכים להיות בקלט רק מספרים וכך אפשר לבקש התבנית +. ולקיים תוצאה דומה, עדיף `+p` שיהפוך את הבדיקה לעיליה יותר.

אם אנחנו יודעים שהביטוי צריך להופיע בתחילת/בסוף מחרוזת – מאוד עיל לשימוש ב-`^` וב-`$`.

• אופרטור '`|`' הוא איטי, איך בכלל יוכל ליעל אותו?

הסדר משנה: למשל `dast|oop|intro` יחפש קודם קודם `intro`. אפשר לשים ראשון את המחרוזת בשכיחות הגבוהה יותר. גורם משותף: אם אפשר להוציאו משוה שמשותף לבבויותם, נעשה זאת. למשל: “`(ab|cd|ef)ab`” עדיף על - “`(ab|cd|ef)cd`” • תמיד נשתמש בຄמת הרקורסיבי בשנובל. פעולה backtracking היא יקרה, ולכן נשתmdl להימנע מכמת חמדני/מינימלייטוי.

- מוטב להימנע שימוש ב-capturing groups בולמי סוגרים. אם אנחנו רוצים סוגרים, אבל רק בשביל הפרדה ולא בשוביל להשתמש בהמשך בחלקים שלהם (כلم בר-1 ובר-2 והלאה) אז נוכל לרשום `(cd|ab:?)` ואז זה בן יחיד אותם, אבל לא יזכיר אותם כ-capturing group.

Regular Expressions in Java 10.5

AIR מושתמשים בביטויים רגולריים ב-Java?

כדי לעבוד עם ביטויים רגולריים נייבא: `*.Pattern`. כדי ליצור `pattern` לא נקרא לבנאי, אלא נשתמש במתודה סטטית שנקראת `compile` ונונתנים לה ביטוי רגולרי:
`Pattern patt = Pattern.compile("[a-z]+");`
 עבשו ניצור `Matcher` למחרוזת ספציפית באופן הבא:
`Matcher matcher = patt.matcher("Now is the time");`

Pattern and Matcher

לשנים אין בכאי פומבי, יוצרים אותו באמצעות המחלקה `Pattern`. אובייקט מסוג `Matcher` מוביל מידע גם על ה-`pattern` שבו נרצה להשתמש וגם על הטקסט עליו נרצה להפעיל אותה. לאחר קומpileציה, `Pattern` אחד יכול לשמש ליותר מ-`Matcher` אחד.
 בהינתן `m` `Matcher` נרצה לדעת האם הביטוי מתאים לטקסט.
`((m.matches() == true) ? "m matches" : "m does not match")`
`((m.find() == true) ? "m finds" : "m does not find")`
 בסוף בshell ישארו עוד, יחזיר `false`.

אם נרצה לקבל חזרה את האינדקסים בו מתחילה ונגמר החלק המתאים, נשתמש ב-`(m.start()...m.end())`. אינדקסים אלו מתאימים לאינדקסים שנדרשים בארגומנטים במתודה `substring`. `str.substring(m.start(), m.end())` ובכה נוכל לקבל לידיינו את המחרוזת המתאימה: `((str.substring(m.start(), m.end()) == "Now is the time"))`

אם לא ניסינו למצוא התאמה, או אם היא לא הייתה מוצלת (החזרה `false`) ניסינו לקרוא `l(-)start()` או `l(-)end()` יזרוק `IllegalStateException` (שגיון זמן ריצה, לא חייב לתפוס אותה).

דוגמה מלאה:

```
import java.util.regex.*;

public class RegexTest {
    public static void main(String args[]) {
        String patternString = "[a-z]+";
        String text = "Now is the time";
        Pattern pattern = Pattern.compile(patternString);
        Matcher matcher = pattern.matcher(text);
        while (matcher.find())
            System.out.print(text.substring(matcher.start(), matcher.end()) + "_");
    }
}
```

התוצאה בגין תחיה: `ow_is_the_time:_`

<u>Return value</u>	<u>Method name and description</u>
boolean	matches(String regex) Tells whether or not this string matches the given regular expression
String []	split(String regex) splits this string around matches of the given regular expression
String	replaceAll(String regex, String replacement) Replaces each substring of this string that matches the given regular expression with the given replacement

שימוש

הפעולה ()`pattern.compile()` יחשיט יקרה, שכן אם נרצה עבור טקסטים שונים להשתמש באותו ביטוי רגולרי, אפשר להשתמש באותו מופיע. מסיבה זו המethode ()`String.matches()` פחות יעיל, כי הוא בכל פעם מקנ赔ל מחדש. לכן אם אנחנו יודעים שנצטרך לחפש את אותו ביטוי מספר פעמים, נעשה זאת בעצמינו.

הערה: בשבוע 10 יש דוגמאות שונות לשימוש ב-`Regex`.

דוגמה מורכבת – מה משמעות הביטוי הרגולרי הבא: "`\d+.?|abc|[xyz][abc]\s*`"

`\d+` → one or more digits ("1234")

`.?` → zero or one occurrence of any character ("a")

`|` → "OR"

`abc` → abc ("abc")

`|` → "OR"

`[xyz][abc]` → matches two characters, first is from xyz second from abc ("yb")

`\s*` → matches zero or more whitespace characters (" ")

אילו טקסטים יתאים למבנה?

"1234", "12a", "abc", "xa", "yb "

Streams – 11

Intro to Streams 11.1

?Stream

זה הרעיון שהתוכנה שלנו עומדת במרכז נתיב של אינפורמציה שזורם מידע אליה וממנה. מפשטים את הרעיון הזה באמצעות `Read` ו-`Write`.

Java Stream Library

אפשרות העבודה של כל תוכנה עם מגוון מקורות אחרים שמקבלים ממנו אינפורמציה ומחזירים אליה אינפורמציה. זה מומש בצורה שמחביה הרבה מהמקורות או הבודות - עקרון של אנקפסולציה.

אלו מתודות אנחנו צריכים?

Create, Write, Read, Delete, **ncstr 4 מתודות עיקריות:**

פרוצדורה של עבודה עם Streams

- (1) פתיחה ה-stream.
- (2) כל עוד יש עוד data (לכתוב אליו/ לקרוא ממנו).
- (3) סגירת ה-stream.

Textual Data vs. Binary Data

ישנם 2 סוגי מידע – טקסטואליים ובינאריים:

קבצים טקסטואליים – מכיל רצף של תווים (קראים לאדם) כגון קבצי טקסט, XML, APILO קבצי Java.

קבצים ביןאריים – מכילים רצפים של Bytes, ואין בהם הכרח מתורגמים בטקסט. דוגמאות לכך הן קבצי zip, jpeg, mp3, .class.

קידוד מידע

העברית מידע מ/אל תוכנית זה סוג של תקשורת ועל כן שבי הצדדים צריכים להסכים על קידוד מסוים, האם המידע טקסטואלי/ביןארי, מה משמעות הדברים שנכתבים..

Streams in Java 11.2

Java.io

בג'ואה קיימת החבילה `java.io` שם יש מחלקות `InputStream`-`Reader` ו-`OutputStream`-`Writer` לעבודה עם קבצים טקסטואליים, ו-`Binary`.

עבודה עם מידע טקסטואלי

מחלקה `Reader` – נותנת API ומימוש חלקו למחלקות של קריאת טקסט. `Writer` דומה אך עבר בתיבת טקסט. `InputStream` ו-`OutputStream` באופן דומה, רק עבר קבצי מידע ביןארי.

עבודה עם מחלקה קונקרטית

לאחר שבחרנו איזה סוג ספציפי של מחלקה אנחנו צריכים, נאתחל באופן הבא:

```
Writer writer = new FileWriter("mail.txt");
writer.write('a');
writer.close();
```

נשים לב בשורה הראשונה, הסיומת `txt`. אינה מחייבת, זו קונברנץיה. מה שקובע זה התוכן שכותבים (אפשר לחשב על זה כך – גם אם היו רושמים סיומת `gç`). זה לא היה הופך את זה לתמונה).

I/O Type	Streams
Memory	<i>CharArrayReader/Writer</i> <i>ByteArrayInput/OutputStream</i>
Files	<i>FileReader/Writer, FileInputStream/OutputStream</i>
Buffering	<i>BufferedReader/Writer, BufferedInputStream/OutputStream</i>
Data Conversion	<i>DataInputStream</i>
Object Serialization	<i>ObjectInputStream</i>
Filtering	<i>FilterReader/Writer, FilterInputStream/OutputStream</i>
Converting between bytes and characters	<i>InputStream/OutputStream</i>

דוגמה בסיסית – העתקת קובץ

```

try {
    InputStream input = new FileInputStream(args[0]);
    OutputStream output = new FileOutputStream(args[1]);
    int result;
    // Reading the file
    while ((result = input.read()) != -1) {
        output.write(result);
    }
    output.close(); input.close(); //Cleanup
} catch (IOException ioErrorHandler) {
    System.err.println("Couldn't copy file");
}

```

בأن יש דוגמה לפונקציה שמעתקה את קובץ א' לקובץ ב'. [0] args הוא המקור, [1] args זה הקובץ היעד. מתחילה מלייצר `InputStream` באמצעותו אפשר לקרוא מה-`InputStream`, ו-`OutputStream` באמצעותו אפשר לכתוב אל היעד. הפעולה `(input.read())` מחזירה את הביטוי של הדאטה או -1 אם הגענו לסוף-ה-stream.

נשים לב שהכל מתבצע בתוך בлок של `try`-`catch`. זו דרך סטנדרטית לטפל בשגיאות IO. זה יכול להיות מסיבות שונות בגין קושי בפתיחת הקבצים או סגירתם, או קושי בקריאה תוכן קובץ המקורי וכתיבה לקובץ היעד (אם מישחו משנה אותן במקביל למשול).

אבל, נשים לב שבקיים שלמו אם השגיאה נזרקה באמצע הכתיבה, לא הגיעו לחלק בו סגרמו את הקבצים – זהה בעייה.

החל מ-`Java 7` אפשר לכתוב כך:

```

try (OutputStream output = new FileOutputStream(args[1]);
     InputStream input = new FileInputStream(args[0])); {
    int result;
    while ((result = input.read()) != -1) {
        output.write(result);
    }
} catch (IOException ioe) {
    System.err.println("Couldn't copy file");
} // No need to close streams! (AutoCloseable interface rocks!)

```

הפעם פתחנו את ה-streams בתוך הבלוק ה-`try`. הכתיבה זו דואגת לכך שברגע שנמצא מה-`try` או מה-`catch` הוא סגור את הקבצים בעצם (ممמש ממש שנקרא `AutoCloseable`). אנחנו מעדיפים לכתוב בצורה זו.

Decorator: Motivation 11.3

אין סואיציה

בעיה: בשאנו בותבים ל-stream, לפעמים אנחנו צריכים לכתוב כמה שפחות (לשמור על מגבלות זיכרון בדיסק, להגביל שימוש ברשת אינטרנט).

פתרון אפשרי: לדחוס את המידע – `.compress`.

אבל, אנחנו נרצה להיות מסוגלים לדחוס דאטा בשאנו עובדים עם סוגי שונים של מבשרי `input`-`output`, והינו רוצים לפתור הכל באותה צורה. איך עושים את זה? האם נרחיב כל מחלקת IO כך שתתאים?

`CompressedFileOutputStream` extends `FileOutputStream`

`CompressedPrinterOutputStream` extends `PrinterOutputStream`

וכן הלאה.. זה מרגיש לא נכון, המון חזרה על קוד.
בעיה נוספת: קריאה של קובץ גדול ביבט-בייט היא לא יעילה, כי הרি קריאת 1,000 בטים שקופה בערך לקריאה ביבט אחד.
פתרון: לקרוא קטעים גדולים של `data` לתוכן `buffer` מוקומי, ובסיום להיפטר מה-`buffer`.
היתרון הוא שאנו צריכים לא צריכים להיפטר בכל פעם מביבט אחד, אבל החיסרונו הוא שלפעמים אנחנו ממתעכבים בין הזמן שבו אנחנו בותבים את הדטה לאפר בין הזמן שבו זה יכתב לדיסק.

אבל בשילוב 2 הבעיה נראה שאנו רוצים גם – גם לכתוב פחות מידע על ידי דחיסה, וגם לעשות את זה מהר יותר עם `buffer`.

הבעיה העיצובית

היום רוצים לשפר את ה-streams עם עוד יכולות. הבעיה הנישת הרבה יכולות שהיינו רוצים לעשות: דחיסה, buffering, הצפנה.. אבל אז נשים עם המון קוד בפול.

Decorator: The Solution 11.4

תזכורת

בහינתן מחלוקת A ו-B
A מכיל את B כאשר הוא מכיל `instance` של B.
A delegates B כאשר הוא מכיל את B, וגם מעביר לו בקשות דרך המתודות שלו.

Decorator Design Pattern

כדי להעצים את B שהוא `class` שהוא ה-`InputStream`, נבנה את Class A (BufferedInputStream) שמקיים:
.A extends B
B delegates A – שולח בקשות לריביב B ומבצע פעולות נוספות.
עבדשו כשהם בעלי API מסווג, זה מאפשר למקשטיים להיות מוכנים.

קריאה וכתיבה עם Buffer



בקוד:

```

Reader inFile = new FileReader("my_file.txt");
Reader inBuffer = new BufferedReader(inFile);

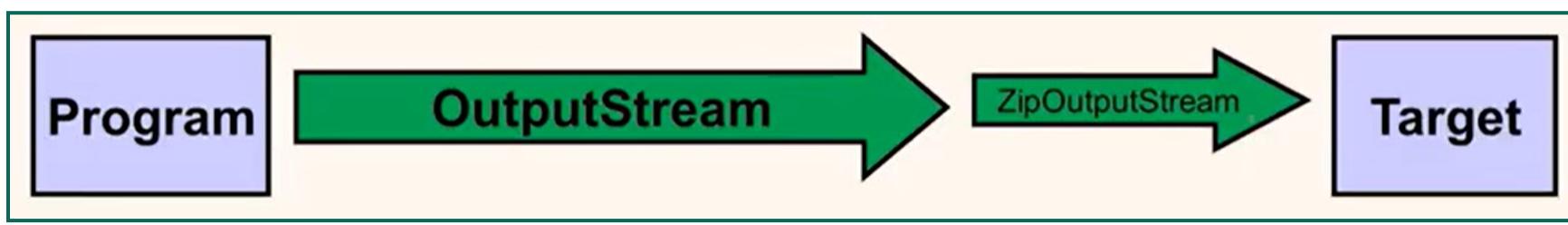
Writer outFile = new FileWriter("my_file.txt");
Writer outBuffer = new BufferedWriter(outFile);
  
```

Decorator classes

Source classes

ונשים לב ש-`BufferedWriter`-`outFile` עטוף `FileReader`, והוא דבר עם `BufferedReader`-`inFile`.

קריאה וכתיבה עם ZipOutputStream



בקוד:

```

OutputStream basic = new FileOutputStream("myfile.dat");
ZipOutputStream advanced = new ZipOutputStream(basic);
  
```

בזכות האופן בו הגדרנו את המחלקות, אפשר עבשו לעטוף כמה פעמים את ה-`InputStream`. אם נרצה גם לעבודה עם `buffer` וגם ביזוי:

```
// Base stream – a FileInputStream
InputStream inStream= new FileInputStream("myfile.dat");

// Efficient reading enhancement - BufferedInputStream
InputStream inBuffered = new BufferedInputStream(inStream);

// Compressed reading enhancement - ZipInputStream
ZipInputStream inZipped = new ZipInputStream(inBuffered);
```

דגשים נוספים על Decorators

מחלקות `Decorator` לא מחזיקות מידע משל עצמן. הן מעבירות בקשות למחלקות בהן יש מידע קונקרטי. באופן דומה, גם מחלקות של `Streams` וכך אין דקורטורים כי הן לא קונספטוואליות, הן מייצגות מחלקות מידע ולא פונקציונאליות. גם באופן שבו הן פועלות, הן לא עושות `delegation` לאובייקט `stream` אחר.

סיכום:

- Let A,B be 2 classes
 - A **Composes** B
 - A **holds an instance** of B (as a data member or a local variable)
 - A **Delegates** B
 - A **composes** B and **forwards requests** to the composed instance's **methods**
 - **Code reuse** alternative to **inheritance**
 - A **Decorates** B
 - A **delegates** B and **extends** B
 - Add a set of **functionalities** to a set of classes

Scanner

מחלקה שמשמשת לקרואת מידע טקסטואלי, ומחזיקה רכיב מסווג `InputStream`. מעבירה בקשות לרכיב זהה ומנתחת טקסט. שימושית בשאנהנו רצים לבצע אנהיזה של טקסט, והוא משתמש ב-`buffer` קטן (קטן יותר מאשר `BufferedReader`).

Scanner משתמש ב-`delegation`, הוא מכיל רכיב `InputStream` ומשתמש בו לבצע את הפעולות שלו. ואולם, `decorator` אין Scanner כי הוא לא עונה ל-`extend`.

פרק 12 Serialization, Cloning, Reflection – 12.1

מה זה Serialization

מוניביציה

זה הרעיון של לחת אובייקט ולשכפל אותו להעתיק אותו מקום למקום, בדומה להעברה של קבצי סאונד או טקסט במחשב. זה עוזר להעביר את התוכנה שלנו עם state מסוים למישחו אחר, או לגבות מצב מסוים..

עבודה עם Streams

דברנו על צימורות דרכם עבור מידע. אנחנו נסתכל על האובייקטים שלנו כמו מידע, ונעביר אותו מהתוכנה שלנו אל יעד מסוים, או מקור מסוים אל התוכנה שלנו.

דוגמאות

```
public class Customer {  
    private String name;  
    private String address;  
    private List<BankAccount> accounts;  
}
```

נניח שנרצה לשמר לקוחות בנק, לומר לחת את האובייקט וכל אחד מהערכיו שלו, ולשמור על דיסק או על טורים. עבור השם והכתובת, פשוט נכתב את המידע ל-stream כי מדובר במידע טקסטואלי. העניין מסתבר בשאלה לשמר משהו כמו רשימת חשבון בנק, ואז נכנים לסייע רקורסיבי כי רוצים לשמר את שמות הלקוחஆן עבור כל אחד נדרש את רשימת הכתובות, אז את כל אחד מחשבונות הבנק, וכל חשבון בנק יתבצע עוד שודות.. מה קורה אם יש שני לקוחות שונים עם אותו חשבון בנק? יתבצע גם התנגשויות. מה קורה אם נשמר אחד ואז בשני היה שינוי ביתריה? אלו שאלות שהנושאים אלה עונה עליהם.

Serialization

התהליך של העברת אובייקט כללי ולהעביר אותו בזיכרון מידע. התהליך הזה עובד רקורסיבית, עבורים על כל שדה של האובייקט ועל כל שדה להפעיל את פעולת הסריאלייזציה וכן הלאה.. עד שmagics לאובייקט פרימיטיבי או אובייקט חסר שודות.

התהליך ההופך של serialization הוא Deserialization, התהליך של לחת אובייקט שנשמר ב-stream ולקראתו ולחזר את האובייקט המקורי. מונחים שקולים שאנו עושים להיתקל בהם: Serializing in ו-Serializing out.

Java Serialization Requirements

בג'אווה על מנת לחת אובייקט ולשמור אותו ל-stream האובייקט צריך להסכים לזה שימושו אותו, בולמר מפתחים יכולים להחליט שהם לא רוצים שהאובייקטים שהם יתבצעו ניתנים לשימירה (מידע רגיש למשל). בברירת מחדל כל מחלקה של ג'אווה לא מסבינה לשימרו אותה. כדי שיוכלו לשמור אותה, היא צריכה למשוך interace Serializable, שהוא ממשך ריק ללא מתודות, הוא רק מסמן שモתר לעשות לו סריאלייזציה (מחלקות אלה נקראות marker).

אחר שהתהליך רקורסיבי, כל אחד מהשדות של המחלקה, והשדות של השדות.. כולל צרכיים להיות או פרימיטיביים או Serializable בעצםם.

איך נראה התהליך?

התהליך serialization בעזרת כתיבה נעשה באמצעות ObjectOutputStream.

התהליך deserialization בעזרת קריאה נעשה באמצעות ObjectInputStream.

שתייה מחלקות Stream באמצעות Decorating. דוגמת קוד:

התהליך כתיבה:

```
try (OutputStream out = new FileOutputStream( "save.ser" );  
     ObjectOutputStream oos = new ObjectOutputStream(out);) {  
    oos.writeObject( new Date() );  
} catch (IOException e) { ... }
```

התהליך קריאה:

```
try ( InputStream in = new FileInputStream( "save.ser" );  
     ObjectInputStream ois = new ObjectInputStream( in );) {  
    Date d = (Date) ois.readObject();  
} catch (IOException e) { ... }
```

לשימוש Downcasting, ב-Object readObject

מה קורה אם מנסים לשמר שדה ששמרנו?

בהתליך הסריאלייזציה, יתכן שנפגש שוב בשדה שכבר שמרנו. אנחנו לא נשמר אותו שוב, אלא נשמר מצביע על השדה ששמרנו בפעם הראשונה. מה החשיבות של הנושא זהה? אם אנחנו שומרים רשימת לקוחות, שלב כל אחד מהם רשימת חברים שבל אחד מהם לquoach, הינו יכולים להיבנס לולאה אינסופית במצב בו יש למשל 3 חברים שבולם חברים אלו של אלו.

דוגמה – מה ידפס בשננסה לבדוק את ה-states בין לבין?

<pre>MyClass obj = new MyClass(); ObjectOutputStream out = new ObjectOutputStream(...); obj.setState(100); out.writeObject(obj); obj.setState(200); out.writeObject(obj);</pre>	<i>// must be Serializable</i> <i>// state – a data member of MyClass</i> <i>// saves object with state = 100</i> <i>// saves object with state = ?</i>
---	--

<pre>ObjectInputStream in = new ObjectInputStream(...); obj = (MyClass)in.readObject(); System.out.println(obj); obj = (MyClass)in.readObject(); System.out.println(obj);</pre>	<i>// prints the state of the obj</i>
---	---------------------------------------

התוכנית זו תדפיס:

100

100

מדוע? ראשית, האובייקטים נשמרים בשיטת `first in first out`, מי שנכתב ראשון נקרא ראשון. נקודה שנייה, פעולה השמירה שומרת את אותו `obj` פעמיים, וזה ששינו את השדה לא אמר נשמר שוב, אלא נשמר מצביע. יש דרכים לעקוף את התהליך הזה כמו למשל לסגור את האוטופוט בין כתיבה לכתיבת, או לקרוא למתודת `reset` ואז הפעם השניה תשמר מחדש.

transient and static Fields

אלו מילים שמורות בג'אווה. המילה `transient` מאפשר לשים אותה לפני ה-`member` `data` וזה יגיד לתהילך deserialization שדות המסומנים `transient` יקבלו ערך דיפולטיבי (שדות נומריים ל-0, מצביעים ל-1111). מתי זה שימושי? עבור שדות שאין טעם לשמור את המצב שלהם. כמו כן, שדות טיטיים שייכים למחלקה ולא לאובייקט, ולכן בתהליך serialization אין טעם לשמור אותם.

לסיכום – בתהליך הסריאלייזציה נשמר שדות שאינם מסומנים ב-transient או ב-static.

אובייקטים פרימיטיביים

פרימיטיבים לא עובדים בתהליך הסריאלייזציה כמו אובייקטים אחרים, למשל: `Out.writeObject(5);` – זו שורה לא חוקית. לעומת זאת, המחלקה `DataOutput` שimplements `DataOutput` מאפשר `out.writeInt(5)` `out.write` לשמר גם `.int`. הערכה: הכוונה היא לא ל-`data members` פרימיטיבים, למשל אם לאובייקט יש משתנה גיל מסווג `char` הוא ישמר ברגיל בתהליך סריאלייזציה, מדובר על שדה שהוא ממש מספר.

Modifying a Class

מה קורה אם שמרנו מחלקה ואז נעשו בה שינויים ונרצה לשמר אותה שוב? יש שינויים שבוצעו במחלקה שהפכו את זה לבליי אפשרי – כמו למשל שדות חדשים, שינויים להיררכיית מחלקות..

מצד שני, שינוי כמו הורדה של `data member` `z` אפשר לנו לעדכן, פשוט נתעלם מהשדה ששמרנו בפעם הראשונה.

מה הפתרון? `version` – שדה טיטי שנקרא `SerialVersionUID` ששמור את הגירסה של המחלקה. אנחנו יכולים לשנות אותו בשאנחנו במצבים שינויים קritisטים בקוד. `SerialVersionUID` זה שדה טיטי `z` שנשמר בಗירסה. אם ביצענו שינויים שאינם מהותיים, לא נשנה את מספר הגירסה ועודן לעדכן את השינויים בהצלחה.

לא חייב להגיד את השדה הזה, כי ג'אווה תגידור אותו עבורינו. הבעיה עם זה, שלא ג'אווה לא יודעת איזו גרסה דורשת שינויים ואיזה לא, לכן היא בכל פעם שבוצע שינוי תעדרן את מספר הגירסה. מומלץ להגיד את השדה הזה בעצמנו ולהחליט מתי שינוי נדרש גרסה ומתי לא.

מה זה Cloning

הפעולה של לחת אובייקט ולשכפל אותו כמה פעמים בתוך התוכנה שלנו, בלי להוציא אותו ל-streams אחרים. למשל:

```
public class Customer {
    private String name;
    private String address;
    private List<BankAccount> accounts;
}
```

נניח שיש לנו לקוחות ונקח רוצים עותק זהה שלו. יהיה לו אותו שם ואוותה בתובת, אבל נשאלת השאלה האם אנחנו רוצים שחשבונות הבנק יהיו מוקושרים? שניינו באחד יכול גם בשני? כאן נבנאים שני סוגי העתקה.

Shallow vs. Deep Copy

בחעתקה שטוחה, אם למחילה יש שדה לא פרימיטיבי – הרפרנסים אליום מועתקים (ולא האובייקטים עצמם). שינוים באובייקט המקורי יגרמו לשינויים באובייקט החדש.

בחעתקה עמוקה, שדות לא פרימיטיביים מועתקים בצורה רקורסיבית גם הם, ונוצרים מחדש השווה לאובייקט המקורי. השדות של האובייקט המקורי והמודעך מצביעים לאובייקטים שונים בזיכרון.

שכפול בג'אווה

כדי שייהי מותק לשכפל מחילה, היא צריכה למש את הממשק הריך (marker) `Cloneable` שנקרא `shallow copy` כמוה כן, צריך לדרכו את המתודה `clone()` של המחלקה `Object`, אם לא נעשה זאת, מקבל `shallow copy` של השדות.

`Object.clone()`

בודקת שהמחלקה שקוראת לה מימוש את `Cloneable` (אחרת יזרק `CloneNotSupportedException`) ובצורה דיפולטיבית תבצע העתקה שטוחה. מערכים מימושים את `Cloneable`, אבל השכפול שהם עושים הוא שטוח. דוגמת קוד:

```
class Pet implements Cloneable {
    private Date birthDate;
    public Object clone()
        throws CloneNotSupportedException {
        // First – creating a shallow copy.
        Pet pet = (Pet) super.clone();
        // Cloning date for deep copy.
        pet.birthDate = (Date) birthDate.clone();

        return pet;
    }
    ...
}
```

אם נרצה שדקה תאריך הלידה יהיה בחעתקה עמוקה נקרא למתחד `clone()`.

```
try {
    Pet myPet = new Pet();
    myPet.setType("Dog");
    Pet myPet1 = (Pet) myPet.clone();
    Pet myPet2 = (Pet) myPet.clone();
    myPet1.setName("Woofi");
    myPet2.setName("Goofi");
    ....
} catch (CloneNotSupportedException e) {
    e.printStackTrace(); // Checked Exception
}
```

לשים לב ל-`upcasting` בغالל שחוזר מ-`Object` טיפוס `Pet`.

אלטרנטיבה ל-Cloning

יש אלטרנטיבה לשכפל שנקרא `copy constructor`. מנגנון פשוט יותר שמאפשר לשכפל אובייקט גם לטיפוס שונה משל עצמו, כמו למשל לשכפל `ArrayList`-ל-`LinkedList`.

```
class Pet {
    private Date birthDate;
    public Pet(Pet other) {
        this(); // First – calling default ctor.

        // Date class doesn't have a copy constructor
        // Use cloning instead
        this.birthDate = other.birthDate.clone();
    }
    ...
}
```

```
Pet myPet = new Pet();
Pet.setType("Dog");
Pet myPet1 = new Pet(myPet);
Pet myPet2 = new Pet(myPet);
myPet1.setName("Woofi");
myPet2.setName("Goofi");
```

ובמשו יוצרים חיית מחמד חדשה באמצעות חיית מחמד קיימת, ואז מבצעים שינוי.

Reflections 12.4

?Java Reflections

מנגן שמאפשר לתוכנית ג'אווה להסתכל על עצמה ולענות על שאלות מה היא, לשנות חלקיים בתוכנה, כמו למשל להדפיס את כל השדות שלה. מדובר בכללי חזק מאוד בג'אווה, וצריך להשתמש בו ב זהירות.

The Class class

כל אובייקט בג'אווה הוא או רפנס (מחלקה, מערך, ממתק) או טיפוס פרימיטיבי (`int`, `char`, `double`...).
כל אובייקט שהוא רפנס מאותחלת מחלוקת שנקראת `java.lang.Class`.
דרך אחת: (`"MyClass"`)
`Class object cls = Class.forName("MyClass")`
דרך נוספת היא ללבת לאובייקט מסוים `myObj` ולבצע: `Class cls = myObj.getClass();` כולם לשמור את המחלוקת אליה שייר.

מה אפשר לעשות עם אובייקט Class

בහינתן מחרוזת המשמש, שמייצגת שם של מחלוקת. יוצרים `instance` של המחלוקת, מה יוכל לעשות אותו?

- אפשר לקבל את רשימת הבנאים שלו: (`Constructor[] ctorlist = cls.getDeclaredConstructors()`)
ואז אפשר דרכה לקרוא למתחודה `newInstance` וליצור אובייקט חדש (עם רשימת הארגומנטים המתאימה).
- אפשר לקבל רשימת מethodים של המחלוקת: (`Method[] methlist = cls.getDeclaredMethods()`)

זה כולל גם **מетодות פרטיות**. עם כל אחת מהmethodות אפשר לשאול איזה מחלוקת הגדרה אותה, מה רשימת הפרמטרים שלהן, מה ערך ההחזרה. אפשר גם כמובן להריץ את המethodות (`methlist[i].invoke(obj, arglist)`).

- אפשר לקבל את רשימת השדות של המחלוקת: (`Field[] fieldlist = cls.getDeclaredFields()`)
אפשר לקחת את רשימת השדות ובעזרת `Field.set` לעדכן את השדה של האובייקט לערך חדש. בצורה דיפולטיבית לא ניתן גישה לשדות פרטיים, אבל אפשר לשנות את זה.

```
import java.lang.reflect.*;

public class DumpMembers {
    public static void main(String args[]) throws ClassNotFoundException, InstantiationException,
        IllegalAccessException, IllegalAccessException, InvocationTargetException {
        Class cls= Class.forName(args[0]);           // args[0] is a class name
        Field[] fields = cls.getDeclaredFields();       // Get class fields
        Constructor[] ctors = cls.getDeclaredConstructors(); // Get class constructors
        Object obj = ctors[0].newInstance();           // Create new instance of the input class
        for (Field field:fields)                      // Traverse object fields
            if (Modifier.isPublic(field.getModifiers())) // Print public ones
                System.out.println(field.getName()+" "+field.get(obj));
    }
}
```

למה בעצם?

- (1) בעצם **Java Reflections** הוא כלי ויעיל לאפשר גמישות ויכולת הרחבה של התוכנית שלנו. למשל – בהתאם לקלט מהמשתמש ליצור ושירות אובייקט של המחלקה (בסגנון דרך הפעולה של מפעל). עם רפלקציות אנחנו יכולים לבתוב קוד כליל לגמרי, ואם הוספנו עוד מחלקה נוספת בהתאם הרבה פעות, כי נוכל ליצור **Instance** שלא מבלי לשכתב את הקוד שמייצר אובייקטים.
 (2) מסייע ב-**Debug**
 (3) תהליכי **Serialization** ניגש לשדות פרטיים כדי לשמר אובייקטים, זה מתבצע באמצעות רפלקציה.

סיכום

מןוגד להרבה עקרונות שראינו בקורס,ऋיך להיזהר עם השימוש בהם.