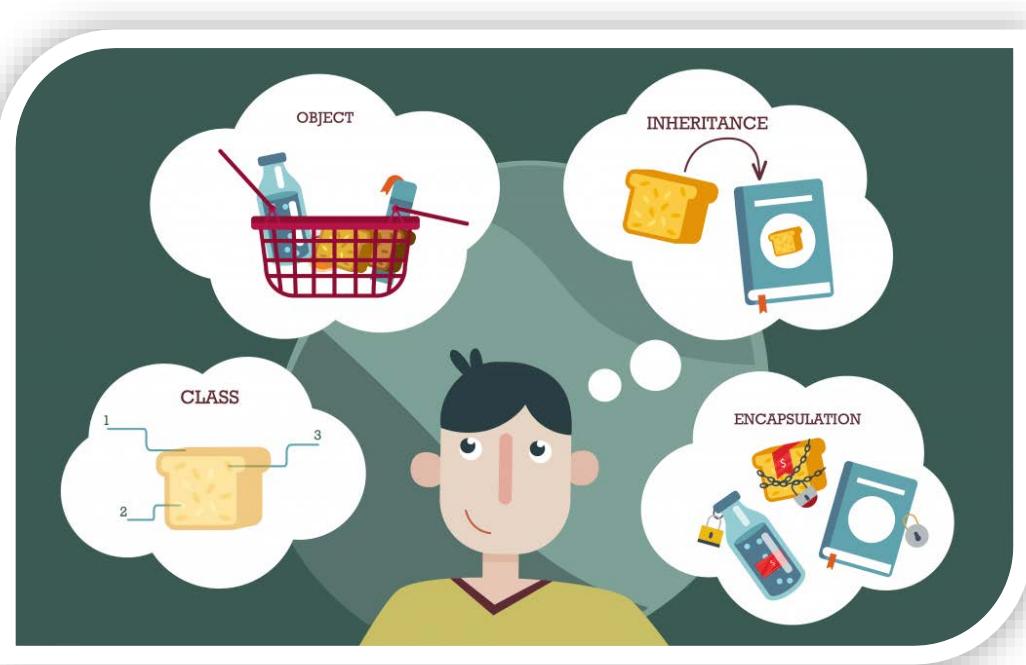


מבוא לתוכנות מונחה עצמים (67125)



מסכם:

יחיאל מרツבן

3	הקדמה לתוכנות מונחה עצמים
5	טיפוסים
9	משתנים וمتודות סטטיות
10	Minimal API
13	כימוס (קפסולה) – Encapsulation
13	ירושה ופולומורפיזם
19	Abstract Classes
20	ממשקים – Interfaces
23	מנגנוני שימוש מחדש בקוד – Reuse Mechanisms
24	Casting
25	Design Patterns
27	Collection
32	Exceptions
34	Packages – חבילות
35	מחלקות מקווננות – Nested Classes
38	מודולריות
40	בחזורה ל-Factory – מחלקת Design patterns
41	Strategy Design Pattern
42	Streams
44	Decorator – מקטש
44	Generics
50	Regular Expressions
55	Serialization – סיריאלייזציה
58	Java Reflections

הקדמה לתוכנות מונחה עצמים

מה هي תוכנה טובה?

1. עברור המשתמש :

- א. לעבוד בצורה טובה.
- ב. קל לעובוד ולהשתמש בה.
- ג. תוכנה מהירה ויעילה (Fail-Safe).
- ד. תוכנה המגנה על טעויות שלנו (Full-Safe).
- ה. תוכנה המגנה علينا ממשתמשים זדוניים.
- ו. תוכנה תאיימה – המותאמת לטביעות הפעלה שונות.

2. עברור המתכונת :

- א. תוכנה שקל לקודד אותה.
- ב. תוכנה שקל לדבג אותה.
- ג. תוכנה שקל להבין אותה.
- ד. תוכנה שקל לשדרוג אותה.

מדוע אם כן علينا להשתמש במונחה עצמים? נשאמש בתכנות כזו, במקרה בו ישן מערכות גדולות הדורשות הרבה חיבורים בין הפרטיהם.

על מנת לדבר על תוכנות מונחה עצמים, נזכיר את המונחים הבאים בקצרה :

- 1. מחולקות ואובייקטים.
- 2. אנקפסולציה – כימוס.
- 3. ירישה.
- 4. פולימורפיים.
- 5. גנריות.

בנוסף נתמקד בנושאים המתעסקים בעיצוב :

- 1. מודולריות
- 2. Desgin-Patterns

באופן כללי, תוכנות מונחה עצמים הוא פרידגמת תוכנות, בו התוכנה מוגדרת כסט של אינטראקציות בין אובייקטים. מדובר על תוכנה השונה מתוכנות פרוצדורלי, כלומר רצף של פקודות. פרידגמה זו קיימת בשפות שונות, וזו ייתכן של שפות כמו Java ו-C#.

מהו אובייקט?

אם נרצה, נוכל לומר שבאופן כללי בעולם, לכל אובייקט יש סט של מצבים המגדיר אותו וכל אובייקט יש סט של התנהוגיות. כך גם אובייקטים בתוכנה, יודעים להחזיק מידע (Data-Members) או לעשות פעולות חיצונית (Methods).

מה ההבדל בין תוכנות מונחה עצמים ותוכנות פרוצדורלי? בתוכנות פרוצדורלי, נבצע פעולות על מתודות, למשל :

```
get_name(child)
wag_tail(dog)
get_length(string)
equals(dog1, dog2)
```

בתכנות מונחה עצמים, הדברים נראים דומה, אבל הם שונים :

- child.getName()
- dog.wagTail()
- string.getLength()
- dog1.equals(dog2)

בתכנות פרוצדורלי, נדר מתודה מתאימה עבור כל חיה, ואילו בתכנות מונחה עצמים נדר מתודה באותו שם עבור כל אחד מהאובייקטים, כך למשל :



מדובר באותו הקוד, אבל הקוד הימני קל יותר לשדרוג ולעדכו – למשל להוסיף חיה נוספת.

מהן מחלקות?

המוטיבציה להגדרת מחלקות הינה שאובייקטים מסוימים דומים אחד לשני. לאובייקטים שונים, אך מאותו הטיפוס, קיימים הבדלים בפרטים, אך יש להם פעולות משותפות.

מחלקות הן יחידות תוכנה שמאפשרות להגדיר קבועות של אובייקטים, כוללם יש אותו סוג של מצבים פנימיים, וכלם יש אותה המתודה. אובייקטים של מחלוקת מסוימים נקראים מופעים (או Instance) של המחלוקת.

Data-Members הם למעשה משתנים המוגדרים על ידי המחלוקת. למעשה, למשל לכל האובייקטים מסווג אופניים למשל, יש תכונה של 'מה הסוג שלהם'. אמנם, כל אופניים הם מסווגים שונים.

שיטות - Methods – הן פונקציות הקשורות למחלוקת מסוימת. למשל, כל האופניים יודעים 'להחילף הילוך'. החילוק הפרוצדורלי של Java נמצא בתוך המתוודות.

אם נרצה, יוכל לראות דוגמה:



ובדוגמה קוד, של אופניים :

<pre>Bicycle.java</pre> <pre>class Bicycle { /* Data members */ int speed = 0; int gear = 1; String brand; // Methods void changeGear(int newValue) { gear = newValue; return; } }</pre>	<pre>// Other methods int speedUp(int increment) { speed = speed + increment; return speed; } void break() { speed = 0; return; } ...</pre>
---	--

בשונה מפייתון, ב-Java נדר את הסוג של המשתנה.

על מנת ליצור מօպעים של מחלקה מסוימת, נדרש להשתמש במבנה (Constructor). בנהו הוא למעשה הדרך שלנו לתת סוגים ספציפיים לאובייקטים (למשל – דגם של אופניים). לבנאים ישם מספר תכונות מעניינות:

1. המתודת שהיא הבניי נקראת באותו שם של המחלקה.
2. הן אינן מחזירות שום דבר.
3. בדומה למетодות, גם הם יכולים לקבל פרמטרים.

כך למשל:

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

ואם נרצה להשתמש במבנה עצמו:

```
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);
    }
}
```

למעשה, יוצרים מօպעים של מחלקה האופניים, כשהיכנסנו להם פרמטרים ספציפיים. לאחר מכן, לקחנו את האובייקטים הללו, וביצענו עליהם פעולה, אז קיבל את הייצוא הבא:

```
Output:
2, 10
3, 10
2, 0
```

当他提到“我们可以在一个类中创建两个不同的自行车对象”，他指的是在`main`方法中创建了两个`Bicycle`对象：`Bicycle bike1 = new Bicycle("BMX", 1);` 和 `Bicycle bike2 = new Bicycle("newbike", 2);`。然后，通过调用它们的方法（`speedUp`和`changeGear`）来执行操作，并打印出结果。

טיפוסים

כל משתנה ב-Java יכול להיות משני סוגי: הפניה לאובייקט מסוים, או משתנה פרימיטיבי. משתנים פרימיטיביים מחזקים את סוגי המידע הנפוצים ביותר. למשל:

- .1 – `int` – מספר שלם.
- .2 – `double` – כל המספרים העשרוניים.
- .3 – `char` – כל מה שניתן ליצג על ידי תו אחד.
- .4 – `boolean` - משתנה מיוחד שיש לו שני מצבים – `true` או `false`.

מחרוזת מיוצגת על ידי מחלקה ב-Java. כל הפרימיטיבים מאותו הסוג, דורשים את אותה כמות זיכרון, אך זה לא נכון לגבי אובייקטים.

למעשה הפניות (Reference) לאובייקטים, זהו ח' שמננה לאובייקט קונקרטי. לכל הפניה יש את הטיפוס שלו, שהיא הסוג של המחלקה.

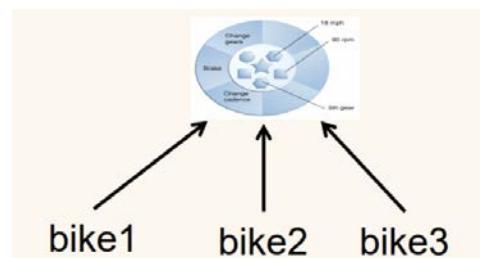
הבדלים בין תוכן והפניה

ראינו שב-Java נוכל להגדיר (1).Bicycle bike1=new Bicycle(1). החלק הראשון הוא למעשה הגדרת הפניה – הגדרכנו משתנה חדש מסוג Bicycle. החלק השני הוא יצרת אובייקט קונקרטי, באמצעות הבניי שלו.

כעת נסתכל על דוגמה לייצירת הפניה:

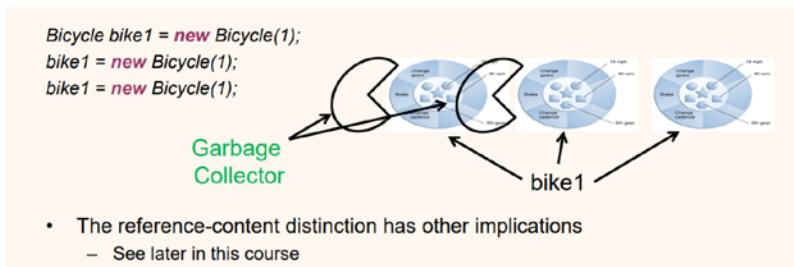
- *Bicycle bike1 = new Bicycle(1);*
- *Bicycle bike2 = bike1;*
- *Bicycle bike3 = bike1;*

בשורה הראשונה יצרנו אובייקט חדש, ובשורות לאחר מכן, הפניות חדשות מצביעות על האובייקט הקיים:



למעשה, אם נשנה את האובייקט, הוא ישנה בשלושת הפניות. בנוסף, יש לדבר זה השלכה לגבי כמה מהוזיכרנו בה אנו משתמשים – הגדרת מצביעים הינה 'זולה' יותר מאשר הגדרת אובייקטים חדשים.

אם נגדיר בכל פעם מחדש את הבניי, אנחנו יוצרים בכל פעם אובייקט חדש:



ה-Garbage Collector תפקידו 'לאכול' את כל האובייקטים הנמצאים ללא הפניה – משחרר את הזיכרונו לשימוש התוכנה. מאידך, לא כדאי לבנות עלי שחרור המידע הזה.

String

היא המחלקה הנפוצה ביותר ב-Java, ואיננו צריכים להשתמש בהנאי על מנת להגדיר אותה, אלא אפשר להשתמש בסימן "=":

String myString = "hello";

ל-String יש מספר מוגדרות שימושיות:

- Length – מזיאת אורך.
- charAt – אות במיקום מסוים.

אמנם, מחלוקת ה-String היא immutable, ככלומר בלתי ניתנת לשינוי.

קבועים – Constans

בתכנות מונחה עצמים ישנה יותר מדרך אחת לפטור בעיה. המטרה היא הרי לכתוב 'תוכנה טובה', אבל לצערנו אין פיתרון יחיד. הדרך בה נבחר לפטור את הבעיה, הינה Desing – עיצוב.

על מנת להתחיל להבין את הרעיון של העיצוב, ניגע במושג קבועים.

ב-Java קיימות אפשרויות ליצור משתנים שאינם משתנים בזמן הריצה והם למעשה קבועים. דבר זה עושים באמצעות המילה השמורה **final**.

מדוע שנרצה להשתמש בקבועים? ישן תכוונות שלא נרצה שישתנו, למשל, שם של אובייקט, או מאפיינים קבועים אחרים. לכן, כשנשתמש ב-design נחשוב קודם כל אליו מאפיינים נרצה שישתנו ואילו לא. בדוגמה **ספרטיפית**, אפשר לראות זאת כך:

```

Dog.java
class Dog {
    /* Data Members */
    final String name;
    int nSiblings;

    /* Constructors */
    Dog (String dogName, int nDogSiblings) {
        name = dogName;
        nSiblings = nDogSiblings;
    }
}

public static void main(String args[]) {
    Dog myDog = new Dog("pluto", 5);
    myDog.nSiblings = 3;           // ok
    myDog.name = "goofy";         // Error!
}

```

הזכירנו כי לא ניתן לשנות את String, אך נשים לב כי קיים הבדל בין immutable ובין קבועים. ה-**immutable** היא תכוונה **שהמחלקה עצמה** מגדרה. לעומת זאת, אפשר להגיד את המחרוזות בצורה שונה. למשל, דבר זה יעבוד:

– String s = "hello";

– s = "goodbye";

אך אם נגדיר זאת בתור final, אין זה יעבוד:

– final String s = "hello";
– s = "goodbye"; // Error

עדין עלינו לשאול, מדוע שמנע מהמשתמש לשנות ערכיהם מסוימים? למעשה, התשובה לכך היא שבשימוש חלק מהפתרונות הינו בימינעה ולא בטיפול. תפקידנו בתור מתכנתים למניעת שימוש מהמשתמש לבצע פעולות מסוימת.

Scope

מהם משתנים מקומיים? משתנים המוגדרים בתוך מתודה, או בתור בлок `if` וכן הלאה. משתנים אלו יכולים להיות מסוגי טיפוסים שונים. למעשה, משתנים מקומיים דומים מאוד ל-`.data members`.

אם כך, מהו scope? כל חלק של קוד התחום באמצעות סוגרים מסוולסים. משתנים מקומיים אינם נגישים מסקופים חיצוניים יותר, כך למשל:

```
if ( ... ) {
    int internalNum = 5;

    ...
}

System.out.println(internalNum);
```

השורה הזאת אינה חוקית, כיון שאנחנו ניגשים לסקופ פנימי יותר.

מайдן, אם ניגש למשתנה בסקופ חיצוני יותר, זה יעבד:

```
int externalNum = 5;
if ( ... ) {
    System.out.println(externalNum);
}
```

באופן כללי, נדע שנדרצה להגדיר משתנה בסקופ הפנימי יותר, כיון שקשה יותר להבין אותו ומילא לתזוזו:

```
String myStr = null;
while (...) {
    if (...) {
        for (...) {
            myStr = ...;
            System.out.println(myStr)
            ...
        }
    }
}
```

למעשה, ההגדרה של הקוד מעלה מיותרת. לכן, נרצה להגדיר את המשתנה בסקופ הפנימי ביותר:

```
while (...) {
    if (...) {
        for (...) {
            String myStr = ...;
            System.out.println(myStr)
            ...
        }
    }
}
```

משתנים וمتודות סטטיות

Static זהה למילה שמורה, הגורמת לחתה שהמידע מוקשור למחלקה עצמה ולא לאובייקט ספציפי שלה. למשל, אם נרצה לספר כמה פעמים יצרנו מופעים של אובייקט ממחלקה מסוימת:

```
class Dog {  
    // Count the number of dogs. This counter is not specific to some  
    // Dog instance, but to the Dog class  
    static int nDogs = 0;  
    String name;  
    int nSiblings;  
  
    Dog(...) {  
        // Dog.nDogs is increased each time a new Dog is created  
        Dog.nDogs += 1;  
        ...  
    }  
    ...  
}
```

בתוך הבנייה של המחלקה, נעלם את מספר הכלבים ב-1.

שיטות סטטיות אינן פועלות על אובייקט ספציפי, אלא לכל המחלקה. בעקבות כך, אי אפשר להשתמש במסתנים של המופיע, אלא רק לגשת לשדות סטטיים אחרים.

אם נרצה, נוכל להסתכל על דוגמה קונקרטית גם לכך:

```
class Dog {  
    // Count the number of dogs  
    static int nDogs = 0;  
    String name;  
    int nSiblings;  
  
    Dog(...) {  
        // Dog.nDogs is increased  
        // each time a new Dog is  
        // created  
        Dog.nDogs = Dog.nDogs + 1;  
        ...  
    }  
    ...  
}  
  
// Get number of Dogs  
static int getDogsCounter() {  
    return Dog.nDogs;  
}  
...
```

המודול הסטטי מחזירה את המשתנה הסטטי.

מדוע שנרצה להגיד מתודות סטטיות? כשאנו משייכים מתודה לאובייקט מסוים, דבר זה אומר שהוא דורשת גישה למשתני המופיע שלו. אם אף אחד מהתנאים האלו אינם מתקיים, מדובר באינדקציה טובה לכך שהמתודה צריכה להיות סטטית.

כעת נחשוב על אלטרנטיבה למודול סטטי, ונרצה להבין את הבעייתיות בה:

```
class Dog {  
    public static void main(String[] args) {  
        int nDogs = 0;  
        Dog dog1 = new Dog();  
        nDogs += 1;  
        System.out.println("noOfInstances (dog1): " + nDogs);  
        Dog dog2 = new Dog();  
        nDogs += 1;  
        System.out.println("noOfInstances (dog2): " + nDogs);  
    } // main  
} // class
```

הבעיה כאן היא קודם כל המבנה הלוגי – הגיוני יותר לספר כלבים בתוך המחלקה עצמה. בנוסף, תיתכן בעיה של חוזריות על הקוד. מעבר לכך, תיתכן בעיה של הרשות.

במקרים מסוימים, נרצה מחלקה מסוימת רק של מתודות סטטיות בלי שימוש במסתני מופיע בכלל. הדוגמה הקלאסית היא מחלקה של אופרציות מותמטיות:

```

/*
 * A library of mathematical methods.
 */
class Math {
    // Computes the sine of a given angle.
    static double sin(double x) { ... }

    // Computes the natural logarithm of a given number.
    static double log(double x) { ... }
    ...
}

```

Minimal API

הרעיון שעומד מאחורי API הינו הרצון לשתף קוד ולשתף אותו עם אחרים או בתוכנות שונות.

על מנת שנוכל להוציא לפועל את החלוקה של הקוד, עליו להיות אטרקטיבי, דהיינו שיהיה ידידותי למשתמש. נוכל לעשות זאת באמצעות API - Application programming interface , שמנדרט למשתמש מה השימוש במחלקות ומה היחס בין המחלקות השונות.

ההגדרה הקשורה לשירותים API הינה Minimal API . כיוון שהקוד הינו ארוך ובשל המורכבות שלו, אנו שואפים לספק כמה שפותות פרטימס, דהיינו API מינימלי.

נזכיר רגע לשאלת ששאלנו בהקשר אחר – למה לא לחלוק? למה לספק פרטימס מינימליים? יש לכך שתי תשובהות עיקריות : בהרבה פעמים, יותר זה פחות, כיוון שהוא הופך למסורבל מדי – מיעוט פרטימס הופך את הקוד לאטרקטיבי. מעבר לכך, ברגע שנספק יותר מידע על הקוד, הוא הופך לקשה יותר לעדכו ולשינוי. דהיינו, כל פרט מידע שנכתב ב-API הוא משחו שאנו מתחייבים אליו.



נכזה לראות כמה דוגמאות. למשל, אם נסתכל על שלושת השעונים הבאים :

לשלאם יש את אותו API למעשה.

בצורה פורמלית יותר, נסתכל בקוד הבא :

```

/* A time class. Represents time of day. Allows comparison between times. */
class Time {
    // time of day
    int hour, minute, second;
    // A constructor that sets the current time of day
    Time() { ... }
    // Is other time before this time? This method uses the convert() method
    boolean isBefore(Time other) { ... }
    // A helper method: converts time to num of seconds from start of day
    int convert() { ... }
}

```

אם נסתכל על API נוכל לומר שהמחלקה הינה להשוות בין זמנים. מדובר אם כן על המשמש לדעת על hour, minute, second ? באופן דומה, מדובר علينا להכיר את המתוודה convert אם כל השימוש שלו הוא פנימי? למעשה, זהו מידע שאין בו צורך והוא מקשה לנו.

אם נניח שבדקנו את המחלקה, והפכנו את הקוד, ונרצה לשדרג את המערכת שלנו ולעשות בה שינויים שאינם משפיעים על הפונקציונליות הבסיסית של הקוד (אלא משפיעה על הייצוג הפנימי בלבד). במצב כזה, על המשתמשים "לשוכח" את API הישן, ומעוד לפורענות. הדרך לפתרור זאת הינה באמצעות כתיבת API הרלוונטי. במצב זה, אין את members data ואין את המתוודות. דהיינו, איןנו יוצרים פרטים הקטנים.

Information Hiding

הרעיוון של Information Hiding הינו לספק דרך פורמלית לאפשר להגיע לminimal API הרעיון של Information Hiding כדי להגיע לרעיון זה, אנו משתמשים בקונספט של Modifiers, דהיינו – מדירים. גיאויה מאפשרת לנו להשתמש בשתי צורות:

- Public – המתוודות והמשתנים גלויים לכלם, ואפשר לקרוא לכל מותוודה.
- Private – המתוודות והמשתנים גלויים רק לאובייקטים במחלקה. כאשר אובייקטים ממחלקות אחרות מנסות לגשת אליהם, זה נגמר בשגיאה.

שתי שאלות שנענה עליהם בהמשך הין :

1. מה קורה כאשר אין Modifier?
2. מודיעו לנו מדירים מחלקות בתור Public ולא Private ?

נוכל להתבונן בדוגמה הבאה :

```

/*
 * A time class. Represents time of day. Allows comparison between times.
 */
public class Time {
    // time of day
    private int hour, minute, second;

    // A constructor that sets the current time of day
    public Time() { ... }

    // Is other time before this time? This method uses the convert() method
    public boolean isBefore(Time other) { ... }

    // A helper method: converts time into num of seconds from start of day
    private int convert() { ... }
}

```

כאמור, לפי API לעיל, אנו מעוניינים בכך שgas Time will be used במחלקות אחרות ולכן הן Public, בשונה משתי המתוודות האחרות, שאנו מעוניינים שיהיו private, ככלומר נגשים רק למחלקה. אם כך, ה- API Minimal Shallow, הוא לפי המתוודות והמשתנים שקבעו ב- Public.

נוכל לראות שימוש ב- API באמצעות השימוש הבא :

Using the Time Class

```
/* A tester for the time class */
public class TimeTester {
    public static void main(String args[]) {
        Time t1 = new Time();           // ok (Constructor is public)
        Time t2 = new Time();           // ok (Constructor is public)

        System.out.println(t1.isBefore(t2)); // ok (isBefore() is public)

        System.out.println(t1.hour);      // Compilation error.
        t2.second = 2;                  // Compilation error.
        int converted = t2.convert();   // Compilation error.
    }
}
```

אם כך, מה כדי להגיד כ-**Public**? ומה כדי להגיד כ-**Private**? או ראשית, באופן כללי על ה-**data members** להיות **private**. מקרה יוצא דופן הוא כאשר מדובר ב-**final static**. לגבי מתודות, علينا להגיד יותר בשלב ה-**design**, ועל כל שאר המתודות להיות פרטיות.

Getters and Setters

אחרי שהגענו למסקנה שכל ה-**data members** צריכים להיות פרטיים. אך מה קורה כאשר סביבה מסוימת צריכה לקבל את המשתנה הזה או לשנות אותו? הפתרון לכך הוא להשתמש ב-**get** ו-**set**, שמאפשרים גישה לערך ולשינויו. האתחול הראשון הינו ב-**constructor**. אנחנו מוכנים "לשלים" על שתי מתודות נוספות, ובלבן שלא 'נחשוף' את המשתנים.

ניתן לראות זאת בדוגמה הבאה :

```
/* A person class. */
public class Person {
    // A person's name
    private String name;

    // A constructor that gets the
    // person's name
    public Person(String personName) {
        name = personName;
    }

    // Name getter
    public String getName() {
        return name;
    }

    // Name setter
    public void setName(String newName) {
        name = newName;
    }
} // end Person class
```

אם כך, מדוע חשוב לנו להגיד כל כך getters ו-setters ?

- כאשר אין מתודת **set**, אין דרך לשנות את המשתנה
- באמצעות **getter** וה**setter** אפשר לעשות כל מיני בדיקות ומיניפולציות שונות
- העבודה שאיננו חושפים את ה-**data member**, מאפשרת לנו לשנות את הקוד בהמשך (למשל **להחליף** **(type)**)

כאשר אנחנו מדברים על **Information Hiding** אנחנו נעדיף לחסוך כמה שפחות מידע, אפילו בשם המתודה, כי יתכן שהדרך שבה נבצע את המתודה תשתנה גם בהמשך. בנוסף, באמצעות אי חשיפת המידע, נוכל לשנות את מבני הנתונים בהם השתמשנו בבניית המחלקה.

עלינו להבחין כי המילה **Private** אינה משמעotta ! המידע הזה אינו מוגן. ישנו מספר מנגנוןים שמאפשרים לעקוף את הרעיון של **private**, גם ב-**java**, עצמה, ולכך דבר זה אינו נכון. אם כך, מדוע משתמשים במילה **Private**? בעיקר בשביל הטוב יותר.

כימוס (קפסולה) – Encapsulation

הרעיון ב- Encapsulation הינו שכאר שניקח קבוצה של רעיונות ונאגד אותה ליחידה אחת. המטרה של Encapsulation הינה לחסוך זיכרונו של מחשבים ובני אדם, על ידי הכנסת רעיון מורכב לשימוש פשוט יותר, שקל לעובד אותו. אם נחזור לדוגמה שראינו, של השעונים, זהו למעשה כימוס – הפיכה של חומרה ותוכנה מורכבת לكونספט שנקרא "שעון".

דוגמא נוספת פשוטה יותר, הינה הדוגמה הזאת:



הכפטור הזה, הינו הדוגמה הפשוטה ביותר לאנקפסולציה, זהו למעשה כפטור אחד עם שני מצבים. הכימוס הקשור באופן ישיר ל- Information Hiding, כי במקרה זה אנחנו יוצרים סט הוראות קטן, וזהו למעשה אנקפסולציה.

ירושה ופולומורפיזם

ראשית, נשאל בכלל מהי מחלקה. אנחנו יכולים לדבר על מחלקה בתור שהוא שאנו יכולים לתאר ב-2 מילים. בנוסף, אנו רוצים לכל מחלקה תהיה איזו פונקציונליות נוספת, מעבר למחלקה אחרת. למשל, אין הצדקה לבנות מחלקה חדשה עבור השם של הכלב. מאידך, הזנב של הכלב, יתכן שהגינוי יותר שתהיה לו מחלקה משל עצמו.

בנוסף, נרצה שמחלקה תתאר משהו כללי, שאפשר ליצור instance ספציפיים שלו. כאשר אנחנו מדברים על משהו שהוא מופיע של המחלקה, אנחנו רוצים שהוא יהיה מופיע ספציפי של המחלקה.

השאלות האלה מביאות אותנו להגדיר את הרעיון הבא.

Single Responsibility Principle

מהי משמעות הרעיון הזה? אנחנו שואפים שלכל מחלקה תהיה מטרת אחת ויחידה. כלומר, יהיה תפקיד שאפשר להגיד אותו בפני עצמו. כל הדברים שבתוך המחלקה יהיו קשורים באופן ישיר למחלקה הזאת.

נוכל לראות דוגמה נגדית כך:

Counter Example

- Consider a class that both **reads** a text file and **counts** the number of words in it

```
public class ReaderAndCounter {  
    // Read a text file  
    public void read() { ... }  
  
    // Count the number of words  
    public void count() { ... }  
}
```

הבעיה בקוד הזה, הינו שהוא מלא שני תפקידים. וכך יש סיכון רב שנרצה בעתיד לשנות את הקוד, דבר שעלול לגרום לבאגים, ודוחרים עבודה נוספת. וכך, נעדיף שהשינויים יהיו ביחידות קטנות ככל היותר.

מה נוכל לעשות, למשל, בקוד שהבנו מוקדם? ברור לנו שرك בעולם מופשט כל אובייקט יכול לעשות פעולה אחת, ויש צורך במחלקה שעשויה פועלות מסוובכת יותר. על מנת לפתור בעיה זו, נוכל להגיד管理 manager מנהל המרכיב את החלקים הקטנים אחד על גבי השני למשועה. כאשר ה-`api` שתפקידו היינו למשועה הנהל. המנהל מרכיב את החלקים הקטנים אחד על גבי השני למשועה. מינימלי, השינויים משפיעים על מחלוקת אחת בלבד, כך למשל בדוגמה הבאה:

```
/** * A manager class.
 * Reads a file and counts its words. */
public class Manager {
    // Manage reading and counting
    public void manage() {
        Reader reader = new Reader(...);
        String[] lines = reader.read();
        ...
        Counter counter = new Counter(...);
        counter.count(lines);
    }
}

/** * A class that reads a text file. */
public class Reader {
    // Read a text file
    public String[] read() { ... }
}

/** * A word counter class. */
public class Counter {
    // Count number of words
    public void count(String[]){ ... }
}
```

ישנה מחלוקת עבור הקריאה, מחלוקת אחת עבור הקריאה, ומחלוקת אחרת שתפקידה לשימוש במחלוקות האחרות.

– ירושא Inheritance

ירושא מאפשרת לנו להגיד מערכת יחסים בין מחלוקות שונות. מהי בכלל מערכות יחסים בין מחלוקות? ניתן לחושב על מערכות יחסים שונות, ונitin להגדיר יחסים אלו כך:

- – יחס זה נקרא גם composition – חכלה. היחס הזה בא לידי ביטוי כאשר לאובייקט א' יש מושג ב' – לדוגמה לכל בנאדם יש שם, ולכל אופניים יש גלגולים. היחס הזה בא לידי ביטוי בדרך כלל ב-`data members`.
 - – מחלוקת א' היא סוג של מחלוקת ב'. למשל, מחלוקת student היא דוגמה של מחלוקת Person – כל סטודנט הוא גם בנאדם (וממילא יש לו אותן תכונות של בנאדם), ויש לו תכונות מסוימות.
 - עקרון זה בא לידי ביטוי בירושא – inheritance. ב-`java`, נאמר שחלוקת א' יורשת את מחלוקת ב' על ידי שימוש במילה extends (extends). המחלוקת המקורית נקראת super-class והחלוקת הירושא נקראת sub-class.
- ונכל לראות זאת בדוגמה הבאה:

```
/* A person with a name and
a mother */
public class Person {
    private String name;
    private Person mother;

    public String getName() {
        return this.name;
    }
    ...
}

/** Student: A person with student
ID that can take exams */
public class Student
    extends Person {
    private int id;

    /** Take an exam */
    public void takeExam() { ... }
    ...
}

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();
```

בצורה כזו, ניתן לגשת גם למוגדרות של סטודנט, וגם למוגדרות של בנאדם.

- – Instance-of Relation – חשוב לא להתבלבל בין סוג זה ובין Is-a Relation. למשל, פלוטו הוא סוג ספציפי של כוכב, ולא מחלוקת כללית.

עלינו לשים לב שיורשה היא רקורסיבית – אובייקט א' יורש את ב' שיורש את ג'. בנוסף ירושה היא טרנזיטיבית, אם א' יורש מב', וב' יורש מג', אז א' יורש מג'. וכך, אין צורך לספק מחדש את המילה extends.

נקודה נוספת שחווב לזכור היא שככל מחלוקת יכולה להיות מחלוקת אב של מספר לא מוגבל של מחלוקות.
מצד שני, מחלוקת בן יכולה להיות יורשת ממחלוקת אחות בלבד.

Object Class

אם לא משתמשים במילה extends, אז הערך הדיפולטיבי הוא ירושה ממחלוקת Object. דבר זה יכול להיות באופן ישיר באמצעות extend, או באמצעות עקיפה, בלי לכתוב כלום. אף כאשר אנו יורשים ממחלוקת אחרת, אנחנו באופן טרנסיטיבי יורשים מextends Object. כלומר, למעשה המחלוקת היחידה שלא יורשת מextends Object היא המחלוקת Object עצמה.

מה המאפיינים של המחלוקת זו?

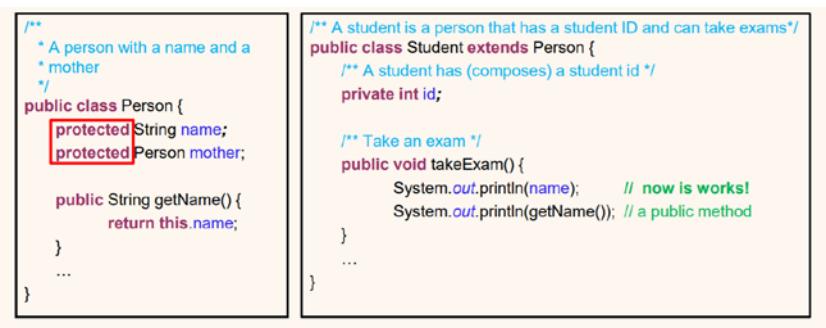
- toString() שמאפשרת להחזיר ייצוג של האובייקט בתווך סטרינג.
- equals() שמאפשרת להשוות בין שני אובייקטים שונים.

.private Data and Inheritance

הקשר בין Private ובין ירושה לא ברור. מצד אחד המחלוקת ירושת את כל המethodות של המחלוקת המקורית, ומצד שני המחלוקת היורשת לא יכולה לגשת למethodות של המחלוקת המקורית. חשוב לציין כי למשל, לכל סטודנט יש שם, אך אין לו גישה אליו.

Protected Modifier

בדומה ל-public ו-private, שיטות ו-data members יכולים לקבל גם את המודיפייר protected שנזכר. דבר זה מאפשר גישה למשתני private רק עבור מחלוקות יורשות, למשל, בדוגמה הבאה:



על פניו, נראה שprotected נותן לנו פיתרון לעוביה שהצינו קודם, אמנים נשתדל שלא להשתמש בו כיוון שיש לו לא מעט חסרונות. בסופו של דבר, כשאנו הוציאים ביטויו כלשהו protected אנחנו כן הוציאים אותו לחלק מהוז api בסופו של דבר, על כל המשתמע מכך. לכן כדאי תמיד להשתמש בprivate.

Overriding

כאן, מדובר על אחות התוכנות השימושיות כshedulerם על ירושה. למעשה, לוקחים מחלוקת ומרחיבים אותה – משנים את התנהלות שלה כך שתבצע דבר אחר. השימוש יחסית פשוט – משתמשים באותו השם במחלוקת המקורית, וממשים אותה מחדש. ברגע שנזכיר למתודה מהמחלוקת היורשת – מקבל קראיה לקוד החדש, אם נזכיר למתודה במחלוקת המקורית – מקבל קראיה לקוד המקורי.

ונכל לראות זאת בדוגמה הבאה:

Student Example Revised

<pre>/** A parent class*/ public class Parent{ public void foo() { System.out.println("P"); } ... }</pre>	<pre>/** A sub class*/ public class Child extends Parent { // Override foo() public void foo() { System.out.println("C"); } ... }</pre>	<pre>Parent p = new Parent(); p.foo(); Child c = new Child(); c.foo(); Output: P C</pre>
---	---	---

כשמדוברים על הcker את המילה `super`, כשרצה לזרוס מותודה, אך עדין להשתמש בモודה הקיימת, כפי שניתן לראות כאן :

<pre>/** A person with a name and a mother */ public class Person { protected String name; protected Person mother; public Person(String name){ this.name=name; } public String getName() { return this.name; } ... }</pre>	<pre>/** 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; } }</pre>	<pre>/** Modify the behavior of getName() to return the name twice */ public String getName() { return super.getName() + " " + super.getName(); } ... } // end of Student class</pre>
---	---	---

נכל לראות זאת בטעט הבא שנבנה :

<pre>/* * A tester for the student class */ public class StudentTest { public static void main (String[] args) { Student stud = new Student("OOP stud", 12345); Person pers = new Person("normal person"); System.out.println(stud.getName()); System.out.println(pers.getName()); } }</pre>	<div style="border: 1px solid blue; padding: 5px; display: inline-block;"> Output: OOP stud OOP stud </div>
---	---

נקודה עדינה שחייב super היא של `super` הינו `super` של `constructern` תמיד נקרא, בין בזורה הישירה ובין הזרה העקיפה. כלומר, תמיד נדרש לקרוא ליצר את הנתונים של האב על מנת ליצור את הנתונים של הבן.

- קראיה זו יכולה להיות בזורה ישירה באמצעות `super`.
- יכולה להיעשות גם בזורה לא מפורשת, וזה נקרא למחלקה המקורית ללא פרמטרים. במידה ואכן אם יש לאב קונסטרקטור שאינו מקבל פרמטרים או שיש פרמטרים דיפולטיביים, אז זה יעבד. אחרת, מתקבל שגיאה.

נראה דוגמה קטנה לכך :

<pre> public class A { public A() { System.out.println("A"); } ... } public class B { public B(int b) { System.out.println("B"); } ... } </pre>	<pre> public class C extends A { public C() { // No super() – implicit // call to A's default ctor System.out.println("C"); } ... } C c = new C(); Output: A C </pre>	<pre> public class D extends B { public D() { // No constructor to B: // compilation error System.out.println("D"); } ... } </pre>
--	--	--

Polymorphism

הרעיון של פולימורפיזם, הוא המוטיבציה המרכזית לשימוש בירושה. מדובר באחד העקרונות הבסיסיים בתכנות מונחה עצמים. פירוש המילה הינו ריבוי צורות. בעולם של התוכנות מונחה העצמים, משמעות הדבר היא יכולת של אובייקטים מסוימים להיות מסווגים שונים. יכולת זאת באה לידי ביטוי בהרחבת מחלקות.

לשם החידוד, נראה דוגמה שתבהיר לנו זאת היבט:

<pre> 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() { ... } } </pre>	<pre> public class Cow extends Animal { public String speak() { return "moo"; } public void getMilk() { ... } public void eat(int calories) { System.out.println("YamYam"); } } </pre>
--	---

כעת, ננסה את הדוגמאות הבאות:

```

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

```

למעשה, הסתכנו על פרה בפריזמה של חייה. ואז מתרחשות התופעות הבאות:

- מחד, חייה יודעת לדבר, ולכן המתודה תעבוד. מצד שני, נקרא למתודה של הפרה.
- בוודאי שנקרה לפרט בפריזמה של פרה, אז הפרה תדע לדבר.
- מצד שני, שנקרה לחיה לקבל חלב, היא לא תדע לעשות זאת, כי לאובייקט של חייה אין את האופציה הזאת.
- כאשר אין דרישת המתודה, אז מסתכלים על המתודה של האב.

ונכל להבחן שהמתודה הינה דבר שימושי ביותר :

The diagram shows a Java code snippet and its output. The code defines a function `makeAnimalSpeak` that takes an animal object and makes it speak. It then creates three animal objects: a cow, a dog, and another cow, and calls the function on each. The output shows the cow making 'moo' and the dog making 'woof' twice.

```
/** A function that get an animal argument of any type and make a sound. */
public void makeAnimalSpeak(    ???    ) {
    ???    speak();
}

Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = new Cow();

makeAnimalSpeak(myCow);
makeAnimalSpeak(myDog);
makeAnimalSpeak(myAnimal);
```

It's the concrete object that counts!

moo
woof
moo

אפשר לראות שהקוד שרצ' הוא הקוד הקונקרטי בהכרח. נראה גם דוגמה נוספת, בה הכלליות של הקוד מאפשרת לנו לעבוד בתור רשימות וכו' :

The diagram shows a Java code snippet and its output. The code defines a function `makeAnimalsSpeak` that takes an array of animals and makes each one speak. It then creates an array of three animals and calls the function on the array. The output shows the dog making 'woof' and the two cows making 'moo'.

```
/** Get an array of animals and makes each of them make a sound. */
public void makeAnimalsSpeak(    ???    ) {
    ???
}

Animal[] animals = new Animal[3];
animals[0] = myDog;
animals[1] = myCow;
animals[2] = myAnimal;

makeAnimalsSpeak(animals);
```

Reminder:
Dog myDog = new Dog();
Cow myCow = new Cow();
Animal myAnimal = myCow;

woof
moo
moo

Shadowing

דיברנו על זה שבפולימורפיזם, ה-*object type* קובע לנו מה מותר להריצ', וה-*reference type* קובע לנו מה יירוץ. כשדבר על *data members* ועל מתודות סטטיות, הדבר מתקיים בצורה שונה. במקרה כזה, רק references קובע, כפי שעולה בדוגמה הבאה :

The diagram shows Java code demonstrating shadowing. Class `A` has a static method `staticFoo` and a data member `myInt`. Class `B`, which extends `A`, also has a static method `staticFoo` and a data member `myInt`. When an object of `B` is created and its methods are called, the outputs show the values 1 and A respectively.

```
public class A {
    public int myInt = 1;
    public static void staticFoo() {
        System.out.println("A");
    }
}

public class B extends A {
    public int myInt = 2;
    public static void staticFoo() {
        System.out.println("B");
    }
}

A a = new B();
System.out.println(a.myInt);
a.staticFoo();
```

1
A

זה הגיוני, כי כשהאנחנו מדברים על *data members* איןנו מכוונים לאובייקט ספציפי.

אנחנו מעדיפים לא להשתמש ב*shadowing* כיון שהוא יוצר בלבול.

Polymorphism and Extensibility

חשיבות של פולימורפיזם היא בכך שהיא תורמת להרחבה התוכנה. בנוסף, יש לנו את היכולת לשנות את ההתנהוגות של הקוד בזמן ריצה. דהיינו, אם הגדרנו אובייקט מסווג חייה בתור פרה, נוכל להגדיר אותה לאחר מכן בתור כלב.

חשוב שנבחן שפולימורפיזם תומך ל-`minimal api`. כאשר פולימורפיזם תומך בReLU של "תכונות למשק ולא למימוש". נרצה שהמתודות שלנו יעבדו עם אובייקט בדרגות גבוהות ככל היותר בסולם ההיררכיה. דבר זה חשוב מכיוון :

- הוא כללי ואיננו מותאים לסוג ספציפי בלבד.
- סוג זה ניתן להרחבה בקבלה.
- סוג זה מאפשר לנו למנוע מהמשתמשים לדעת באילו אובייקט אנו משתמשים, דבר שמאפשר לנו לעשות שינויים בקוד.
- קל לשימוש.

Abstract Classes

נתחיל במוטיבציה קטנה לReLU של מחלקות אבסטרקטיות. נניח שיש לנו מחלוקת של חיים, וכל אחד יש מאפיינים מסוימים. בדוגמה למה שעשינו קודם, היינו יכולים לבנות היררכיה של מחלקות. אם למשל נרצה קוד כללי של "דיבור", תהיה לנו בעיה, כי הרי כל חייה 'מדברת' בצורה שונה, וממילא הקוד הספציפי שלה ידרוש את הקוד הכללי. כיצד ניתן לפתור זאת? ננסה לחשב על שני פתרונות, ונציג את החסרונות שלהם:

1. מכיווןSCP של חייה מדברת בצורה שונה, נימנע מיצירת מחלוקת כללית כזו.
- פתרון זה יוצר סרבול, והימנע מיצירת API כללי.
- בצורה כזו בלתי אפשרי לייצור `polymorphism`.
2. ניצור את המתודה "דיבור", אך היא תהיה מתודת ריקה. וכך כל חייה יתדרש את המחלוקת הריקה.
- במידת ואחת המחלוקות 'שוכחות' לדروس את המתודה הזו, החייה לא משמשה قول.
- מבחינה קונסיסטואלית, אין משמעות לкриיאת ריקה.

פתרון נוסף הוא **מחלקות אבסטרקטיות**. מדובר למעשה על מחלקות רגילות, עם המילה השמורה `instance`, שבلتאי אפשר לייצר `abstract` של המחלוקת הזאת. מדובר כבר על סוג של פתרון לבעה שהציגנו קודם לכן, שכן בדרך זו אנו אכן לא יכולים להתייחס לחייה בפני עצמה, אלא מדובר ביצור אבסטרקטי.

מאפיינים של מחלקות אבסטרקטיות :

- בעזרת המחלוקת האבסטרקטית, אפשר לייצור גם מתודות ללא למימוש, כלומר ללא תוכן.
- ולמעשה ללא קוד :

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

למעשה, בצורה כזו, כל מחלוקת צריכה 'לדרש' את המתודה הקיימת, כפי שניתן לראות בדוגמה הבאה :

```

public class Dog extends Animal {
    // Implementing the abstract speak() method.
    public void speak() {
        System.out.println("haw");
    }
}

```

- במידה ולא נכתב את המתודה הזו, קיבל שגיאת קומפיילציה.
- מחלקה אבסטרקטית יכולה לירש מחלקה אבסטרקטית אחרת, אך היא אינה צריכה 'לדרוס' את המתודות הקיימות.
- ההבדל היחיד בין מחלקה אבסטרקטית למחלקה רגילה, היא שלא ניתן ליצור instance שלהם.
- מתודות סטטיות לא יכולות להיות אבסטרקטיות.
- כאשר נשתמש במתודות super על מתודה שהיא אבסטרקטית, ככלומר ננסה לקרוא לקוד של מחלקה האב, כשהוא לא קיים, קיבל שגיאת קומפיילציה.
- מחלקות אבסטרקטיות לא יכולות להיות מוגדרות private.
- השתמש במחלקות אבסטרקטיות, כאשר אין הגיוני להגדיר instance של המחלקה הזאת, כמו במקרה ונרצה להגדיר חיה, או כאשר נרצה לכפות API על סט מסוים של מחלקות.

– ממשקים – Interfaces

ממשק, הוא דבר דומה מאד למחלקה, שיכול להכיל שני דברים :

- קבועים (בעלי המילה השמורה final)
- מתודות אבסטרקטיות.

בדומה למחלקות אבסטרקטיות, לא ניתן לצור אובייקטים ספציפיים מהם, אלא רק :

- לרשות אוטם.
- לממש אוטם.

ראשית, נסתכל על דוגמה קונקרטית :

```

/* An interface for printable objects. */
public interface Printable {
    // A print method
    public void print();
}

```

כעת, נסתכל על מחלקה שimplements את הממשק זהה :

```

public class Document implements Printable {
    // Implementing the Printable.print() method
    public void print() {
        ...
    }
    ...
}

```

כלומר, כתת-Ano נתונים מימוש קונקרטי לקוד הנמצא במסמך.

מה יקרה כאשר נרצה ליצור אובייקט קונקרטי של ממש? נקבל שגיאת קומPILEZIA.

מה המוטיבציה לממשקים? מדובר למעשה בסוג של חווים, שמחקוקות מקובלות על עצמן. אפשרות לחושב על כמה אפשרויות לממשקים:

- בר הדפסה.
- בר השוואה למחוקות אחרות.
- בר יכולת שכפול.

נשים לב שמשקדים מדברים על 'מה' ולא על 'איך'. בצורה אחרת, נוכל לבדוק כי למעשה מגדירים למעשה API.

עלינו לשים לב כי ממשקים יכולים להגדיר רק متודות מסווג `public`, וגם יכולים להגדיר קבועים בלבד. דברים אלו היגייניים, כאשר נדבר על כך שמשקדים מגדירים למעשה API.

נקודה נוספת שעליינו לשים לב אליה, היא היכולת להרחיב ממשקים. דבר זה שימושי כאשר יש 'היררכיה של חווים'. דהיינו, כאשר יש דרישות בסיסית, וככל שיורדים בהיררכיה, הדרישות הולכות ומחמירות. ניתן להראות דוגמה קטנה כאן:

```
public interface MyInterface {  
    public void superFoo();  
}  
  
public interface MySubInterface extends MyInterface {  
    public int subFoo();  
}  
  
public class MyClass implements MySubInterface {  
    public void superFoo() { ... }  
    public int subFoo() { ... }  
}
```

מחלקה שמשמשת את החוזה שהתחייבת אליו, יכולה להוסיף מגבלות, אך לא להחסיר מהקיימות. מאידך, ניתן שבממשק מסוים יידרשו אי אלו דרישות קדמ, והמחלקה יכולה להסתפק בדרישות נוספות, אך לא יותר. אם נרצה, נוכל לחושב על כך בתור חוזה גם בחימם הקיימים: במידה ונרצה לקנות במקולת, המוכר יכול להוסיף למה שהתחייב אליו, אך לא פחות; בצורה דומה, המוכר יכול לדרש פחות כסף מהקונה, אך לא לגיטימי מבחינתי לדרש יותר מהtag מחיר הקיים. במובן זה, ממשקים דומים לצד של המוכר.

נראה שתתי דוגמאות לכך:

```
public interface FactorFinder {  
    /* @return a > 1 factor of the given positive integer n. Return n iff n is prime */  
    public int factorOf (int n);  
}  
  
public class SmallestFactorFinder implements FactorFinder {  
    /* @return the smallest prime factor of the integer n */  
    public int factorOf (int n) {  
        for (int i = 2 ; ; ++i)  
            if (n%i == 0)  
                return i;  
    }  
}
```

Offer more
(smallest factor)

זה דוגמה ל'החמרה של החוצה'.

מײַידֶן, נראָה דוגמה ל'דרישות מוקלוט יוטרי':

```
public interface ArrayManipulator {  
    /** Perform some manipulation on array. @param array – a non empty array */  
    public int manipulate(int[] array);  
}  
  
public class ArrayPrinter implements ArrayManipulator {  
    /** Print array. Do nothing if array is empty. */  
    public int manipulate(int[] array) {  
        for (int i: array)  
            System.out.println(i);  
    }  
}
```

Fewer pre-conditions
(array may be empty)

בשונה מהמשמעות, המחלקה הממשית יכולה להיות ריקה.

עלינו לשים לב כי בשונה מירושה, אין היררכיה, ככלומר, כל מחלקה יכולה לרשת כמה ממשקים שהוא רוצה. וכך, על כל מחלקה ניתן להסתכל במספר צורות – ככלומר כל סט המשאים שהוא ממשית. דבר זה מאפשר לנו להסתכלשוב על polymorphism, שהרי על כל מחלקה ניתן להסתכל בזווית שונה. נראָה דוגמה:
לכֹּךְ:

ראשית, נגדיר את המחלקה ואת המשאים שהוא יורשת:

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

לאחר מכן, נוכל לבצע את הפעולות הבאות:

All the following are legal:

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

נדגיש כי כמו שראינו בירושה, בלתי אפשרי לקרוא למגוון אחריות שלא קיימות ממשקים או במחלקות האב.

מה הקשר בין ממשקים ומחלקות אבסטרקטיות? משניהם אי אפשר ליצור instance ושניהם מגדירות API.
אם כך, متى נשתמש בכל אחד מהסוגים?

- אם מחלקה אחת היא סוג של מחלקה אחרת, למשל סטודנט הוא סוג של בנאדם וכו', אז כדאי להשתמש בירושה (אבסטרקטית או רגילה)
- אם מחלקה אחת היא בסוג של חוצה עם המחלקה האחראית ולא יורשת אותה, נעדייף להשתמש ביחס של מימוש ולא ירשה – למשל מסמך הוא 'בר הדפסה', ולא סוג של הדפסה.

אמנם, לא תמיד קל לבחור, ולכן כאשר אין העדפה ברורה, יש להעדיף ממשקים, שכן ניתן למשם כמו ממשקים.

כעת, נראָה כמה דוגמאות למשקים:

- **interface** `java.util.Collection`
 - A general purpose data structure
 - `add()`, `remove()`, `size()`, ...
- **interface** `java.util.List extends Collection`
 - A collection that allows access by index
 - `get(index)`, `set(index, value)`, ...

הסיבות collections מעט מבלבלות – אפשר לחשב שרשימה מקוורת היא סוג של רשימה. אך עליינו להבחן כי ממשקים מדברים יותר על ה'ימה' ולא על ה'איך'. בנוסף, כאשר נדבר על ממשקים, יש פחות דוגמאות שעולות על פני השטח.

מנגנון שימוש חדש בקוד - Reuse Mechanisms

מהי הדרך הטובה ביותר לשימוש חוזר בקוד? עד כה השתמשנו במנגנון של ירושה, שהינו חלק משפט `java` עצמה. אולם, ישנה דרך אחרת, באמצעות `composition` – להחזיק אובייקט שנעשה את הפעולה הזו.

דבר על השיטות האלו בהרחבה יותר :

1. בירושה, אנחנו משתמשים במחלקה אחת באמצעות מחלקה אחרת. דהיינו, אנחנו מרחיבים מחלקה קיימת. לעיתים, ירושה נקראת 'קופסה לבנה', מכיוון שאנו משתמשים במחלקה שירים אותה, אנחנו יכולים לכל מיני מותודות ואלמנטים במחלקה המקורית.
2. בהכללה – `composition`, אנחנו יכולים לגשת לפונקציונליות של האובייקט. דרישת הקדם ל'הכללה', הינה שה-`API` יהיה ברור לנו היטוב. ברגע לירושה, הכללה נקראת 'קופסה שחורה', כי אנחנו קוראים למותודות שלו בלבד, ולא יודעים מה מתרחש באובייקט עצמו.

מה היתרון והחסרונות של כל אחת מהם :

- **היתרון של ירושה :**
 - א. חלק מהשפה – ברור לחלווטין.
 - ב. פולימורפיזם.
 - ג. מוגדרת באופן סטטי בתחילת הריצה ואייננה יכולה להשתנות (יש לו גם חסרונות).
 - ד. קל יותר למימוש של המחלקה היורשת באמצעות overriding.
- **החסרונות של ירושה :**
 - א. העובדה שהיא מוגדרת באופן סטטי, גורמת לכך שמדובר בסוג של 'חתונה קתולית'. לא ניתן לשנות את הקוד במהלך הריצה.
 - ב. במובן מסוים היא שוברת את האנרכטולציה, מכיוון שאנו נחשפים למידע שיש במחלקות האב.
 - ג. המחלקה היורשת מחויבת למימוש של האב – ייתכן שהמחלקה היורשת לא תרצה למשמש את כל המותודות של האב.
 - ד. אפשר לרשת רק מחלקה אחת.
- **היתרון של 'הכללה' :**
 - א. מוגדרת באופן דינמי בזמן ריצה (יש לו חסרונות, בהתאם), דבר שמאפשר לנו גמישות.
 - ב. איןנו שוברים את ה-`encapsulation` בשום צורה.
 - ג. איןנו יוצרים תלות בין האובייקטים.

- ד. ב'הכליה', כל מחלקת מוגדרת למשימה אחת.
ה. מחלוקת יכולה להגדיר כמה אובייקטים שהוא רוצה.

- החסרונות של 'הכליה':

- אין אפשרות לפולימורפיזם.
- יש הרבה אובייקטים ולעתים קשה להבין את הקוד.
- העובדת שהקוד דינמי יכולה לגרום לטיעויות.

אם נסכים, יוכל לומר שCDATA לשימוש בירושה, כאשר הוא סוג של `a`. אם זה לא המצב, אז נראה שירושה זה לא המצב הנכון. מאידך, אם נרצה רק למחזר קוד, אז הכליה צריכה תהיה הקוד הנכון. למעשה, אם נרצה רק פולימורפיזם, יוכל לשימוש ב-interface.

Casting

מדובר למעשה על הרעיון הבסיסי העומד ב-polymorphism – הרעיון של שימוש באובייקט אחד בתורה אובייקט אחר. הדבר הזה נקרא casting. בדומה פורמלית יותר, כאשר עוסקים בסולם ההיררכיה, דבר זה נקרא Up-casting. למעשה, Up-casting יש שני סוגים עיקריים:

א. הסוג שראינו עד כה, שנקרא implicit-up-casting – שאינו מפורש. במקרה זה, ה-compiler מחליט מהו הסוג של ה-casting.

ב. Explicit-up-casting – בקרה מפורשת בכך כלל, ה-compiler מזוהה את ה-casting על סמך זה. במקרה בו מושך ערך מסוג זה משליך על casting אחר, שנראה כתה.

Down-casting הוא למעשה casting בכיוון ההפוך, התנייחות לאובייקט 'במורד העז'. למעשה, בעקבות כך, על ה-casting להיות מפורש. למשל:

`Cow c = (Cow) animal;`

דבר זה יכול להצליח, אך גם יכול להיכשל, במקרה הזה למשל:

- But it can also fail
 - Animal animal = new Dog();
 - Cow c = (Cow) animal;
 - Cow c2 = (Cow) new Integer(5);
- animal is actually a Dog. It cannot be cast to Cow.
- Cow is not a sub-class of Integer. This operation can never succeed. Compilation error

למעשה, איןנו אוחבים down-casting, מכיוון שכפי שראינו קודם לכן, ניתן בו שגיאות בזמן הריצה, דבר בעייתי במיוחד. בנוסף, מדובר בעיה במגוון – שהרוי אנחנו לוקחים קוד כללי והופכים אותו לספקיבי, בניגוד לגישות שראינו עד עכשיו.

InstanceOf

- אופרטור של Java שמאפשר לנו בזמן ריצה לבדוק האם אובייקט הוא מסווג מסוים. instanceof

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

Returns true

לכארה, דבר זה פותר את הבעיה של down-casting, אך למעשה זה פיתרון גרווע, מכיוון שגם הוא מועוד לבאגים. בנוסף, גם פיתרון זה אינו גמיש, כפי שנראה כאן:

```

class AnimalMover{
    public void moveAnimal(Animal animal) {
        if (animal instanceof Fish) {
            ((Fish)animal).swim();
        } else if (animal instanceof Horse) {
            ((Horse)animal).ride();
        }
        ...
    }
}

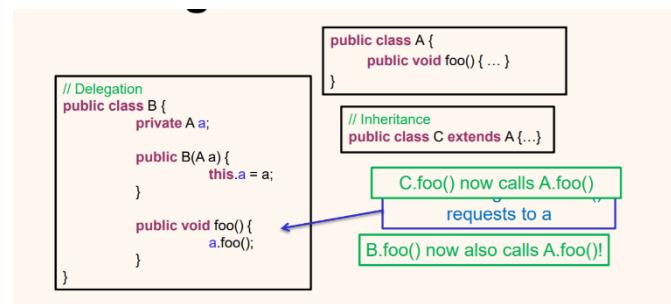
```

במידה ונרצה להוסיף חיה חדשה, הקוד כבר לא יעבד. אם כך, מהו הפתרון? שימוש ב-API כללי.

Design Patterns

רעיון זה הגיע מעולם האדריכליות, כאשר חיפשו פתרונות לבניות מבנים. הפתרונות האלה היו מספיקים כלליים וגם מספיקים קונקרטיים. עולם התוכנה אימץ הרעיון הזה, והגיע ל-design patterns. המשמעות הינה שאפשר להתמודד עם בעיות ב-design, שלפעמים חזורות על עצמן. הוא למעשה מציע לנו פתרונות אלגנטיים וجمישים. בנוסף הוא מאפשר לנו לדעת מה יהיו ההשלכות של ה-design שלנו, ואינו תלוי בשפה או מקרה ספציפי.

על מנת להבין את המוטיבציה לשימוש ב-design patterns, נסתכל על הבעיה הבאה. למשל, אם נניח שנרצה מחלקה שעושה אותו דבר כמו מחלקה אחרת, ללא שימוש בירושה. נוכל להשתמש ב-delegation, שהוא סוג : design patterns של



כלומר, אנחנו מנסים להשתמש בירושה, דרך הרעיון של קומפוזיציה למעשה. דבר זה נקרא דلغזיה.

מדוע שמשתמש בReLU זה?

- דבר זה אפשר לנו להחליף את אובייקט a באובייקט d למשל.
- בנוסף, נוכל לקבל אפשרות של ירושה ממחלקה אחרת.
- tabnition שמתאימה להרבה הקשרים שונים ולא נקודתיים.

ברמה הפורמלית, אלו הדברים שמופיעים : design pattern

- שם בפני עצמו – לכל רעיון אמרור להיות שם משלו, על מנת ליצור שפה משותפת.
- הגדרת הבעיה – הרעיון צריך להגיד מה הבעיה שונמדת, ובallo תנאים עליו לעמוד.
- פתרונות עצמו – מהם הרכיבים השונים של הפתרון ומהם הקשרים ביניהם. לא מדובר על פתרון קונكريטי, אלא על סכמה.

ד. ה歇לכות שלו – כיצד ישפייע על שאר המערכת והמרכיבים של הקוד.

למעשה, קיימים שלושה סוגים pattern :

א. Creational patterns – רעיון שעסקים ביצירה של אובייקטים. למשל:

.Factory. (1)

.Singleton (2)

ב. Structural patterns – רעיון שעסקים במבנה. למשל:

.Delegation. (1)

.Façade (2)

.Decorator (3)

ג. Behavioral patterns – רעיון שעסקים בחתנהגות. למשל:

.Iterator (1)

.Strategy (2)

Façade

רעיון זה מתעסק במבנה. השם שלו מגיע מצרפתית והמשמעות שלו היא לפנים. גם רעיון זה מגיע מאדריכלות, והכוונה היא לצד של הבניין. בעולם התכונות, הכוונה היא לקחת רעיון מורכב ולהסתכל עליו דרך חלון צר.

א. השם של ה-pattern – דיברנו על כך בפסקה הקודמת.

ב. הבעיה שקיים בפנינו היא כאשר ישנה מערכת גדולה עם המון מחלקות שונות, ואז נוצר מושך.

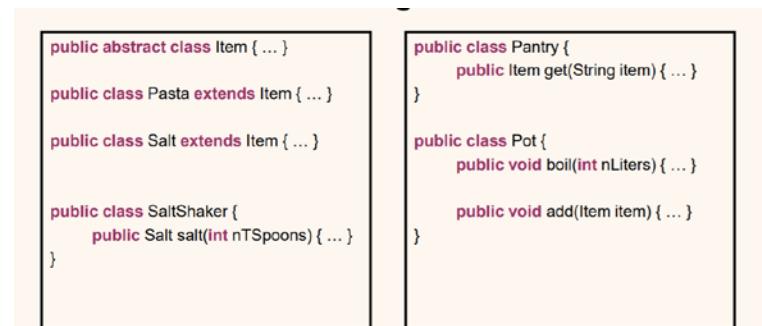
ג. הרעיון הינו לבנות API פשוט דרך בנית מחלקה אחרת, שעשויה את הפעולות המורכבות של ה-API הקשה. בrama הפורמלית:

(1) בניית מחלקה חדשה.

(2) נגדיר למחלקה זו יש API פשוט יותר.

(3) נממש אותו בכך שהוא ייקח את המערכת הגדולה, ויצא את המערכת הקטנה.

נסתכל על דוגמה, למערכת מטבח מורכבת:



כעת נבנה מחלקה שימושת בモרכבות של המטבח ובונה פתרון קטן של יצרת 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);
    ...
}
```

ד. מה היתרונות של רעיון זה?

- לקוחות רואים רק את המערכת הקטנה.

- חוסר מודעות לשינויים הקטנים.

- לקוחות מנוסים יכולים להשתמש במערכת המקורית.

- הוא אינו מוסיף מצד עצמו.

- אפשר לחת API מורכב ווופך אותו לפחות.

נראה דוגמה נוספת לשימוש ברעיון זה:

```
public class ComplexPlayer {
    public void play(String fileName,
                    int leftvolume,
                    int rightvolume,
                    double 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);
    }
}
```

Collection

אוסף הוא למעשה אובייקט תכוני שמחזק הרבה אובייקטים אחרים. אפשר גם לקרוא לו 'מבנה נתונים'. ב'מבנה נתונים' קודם כל מאחסנים מידע וגם אפשרות כל מיני מניפולציות על מידע זה.

אוסף יכול לייצג דברים קונקרטיים, או דברים מופשטים.

ב-Java יש את הרכיב הנקרא collections framework המכיל שלושה רכיבים מרכזיים:

- .Interfaces -
- .Implements -
- .Algorithms -

על מנת להשתמש בהם, צריך לייבא את המחלקה, באמצעות import java.util.

דבר כעת על כל אחד מחלקים:

א. Interfaces – מגדר לו כל מיני מבנים כללים של מבני נתונים בלי מימוש ספציפי כמו map, list, set, וכו' להלאה.

treemap, hashset, - Implementations מבני הנתונים הללו. למשל -

.linkedlist

ג. – Algorithms – אפשרות פעולה מסוימת כמו חיפוש לינארי, ערבות וכו'.

מה החשיבות של Collections ?

א. שימושי ומוריד זמן עבודה. אין צורך להמציא מחדש את הכלל.

ב. חבילת תוכנה איקוונית. יש פחות באגים.

ג. אפשר ל-API שונים לעבוד אחד עם השני באמצעות ה-Collections .

מערכות

מערכות היא הדרך הכי פרימיטיבית ב-Java להחזיק נתונים. אבל יש להם הרבה בעיות:

- הגדרת הגודל מראש.

- אין דרך לשנות את ההתנהגות של הערכים.

Genericity

עד כה התעסקנו בציר הנע בין האבסטרקטי לקונקרטי, ואילו Genericity מתעסק בשאלת מה מחזיקות המחלקות הללו. Genericity מאפשר לנו להגדיר פרמטרים מסוימים לתוךן שלהם. למשל, אם נתבונן בדוגמה הבאה `List<E> E List.get(int index)` ניתן לקבל איבר מסווג E. בכל פעם שנרצה ליצור איבר מהמחלקה List אנחנו למשה יוצרים מחלקה חדשה, בפעם אחת מקבל רשימה של מחרוזות, ובפעם אחרת מקבל רשימה של מספרים וכו'.

בפועל, משתמשים בדבר זה כך:

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

אם נרצה, יוכל לראות דוגמה גדולה יותר:

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

// A list of doubles
LinkedList<Double> list2 = new LinkedList<Double>();
list2.add(new Double(2.71));
Double d2 = list2.get(0);
list2.add("hello");             // Compilation error
```

אחרי שדיברנו קצת על Genericity נוכל לומר שלמעשה כל מבני הנתונים שהסתכלנו עליהם ב-collections הינם גנריים.

מהו בעצם ההבדל בין Implementation ובין Interface ? מתי נדע להשתמש בכל אוסף?

ישנן מספר שאלות שעוזרות לנו לענות על איזה ממשק לבחור:

א. האם אפשר להכניס כל איבר פערמיים?

ב. האם הוא ממויין או לא?

ג. האם יש לו מפתחות או רק ערכים?

בנוגע למימוש, علينا לשאול מספר שאלות :

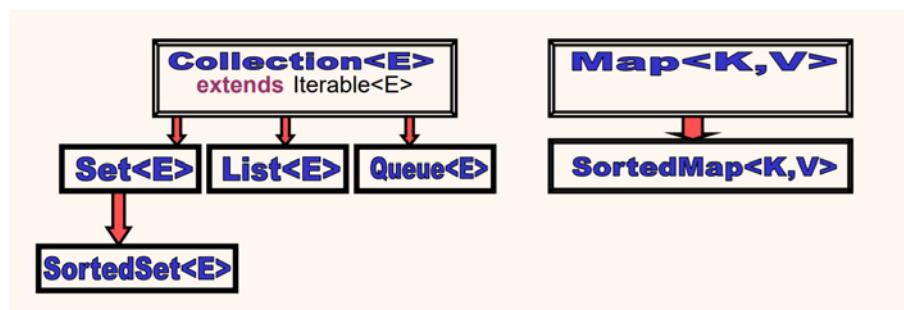
- מה הסיבוכיות?
- מה הדרישות?

בגدول, ישנו 7 אוסףים שונים :

1. הממשק הבסיסי הוא `collection`.`Iterable`, שהינו כל מבנה נתוני שאפשר לעשות עליו איטרציה. מתוך `collection` יורשים שלושת האוספים הבאים :

 2. `Set`
 3. `List`
 4. `Queue`
 5. מ-`set` יורש `map` וממנו יורש הממשק הבא :
 6. ממשק נוסף הוא `map` וממנו יורש הממשק הבא :
 7. `Sortedmap`

לסיכום קצר :



מה שմבדיל בין שני הצדדים הוא ה-`generics`.

אם נתבונן ב-`javadoc` נוכל לראות את ה-API הבא :

```

public interface Collection<E> extends Iterable<E> {
    // Bulk operations
    boolean containsAll(Collection<?> c);
    // Optional Bulk operations
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
  
```

כעת ניגש לעסוק בממשקים שיורשים מ-`collection`.

ה-interface הראשון שניגש אליו הוא `List`. מדובר במבנה נתונים שבו יש סדר (לאו דווקא ממויין). בגלל שיש איבר ראשון שני וכן הלאה, אז ניתן לגשת לאיבר ספציפי. האיברים יכולים להיות אותו הדבר בדיקוק.

ה-interface השני הוא `queue` – תור. עליו לדעת מי עומד בראש התור (`peek`) ומצד שני להכניס איבר חדש (`push`), ולהוציא את ראש התור (`pop`). איך קובעים מי ראש התור? בדרך כלל מקובל לקבוע לפי FIFO – זה הינו, `first-in-first-out`. במקומות אחרים, מסדרים תור בצורה אחרת – דבר זה נקרא Priority queues – למי יש עדיפות גדולה יותר.

ה-`interface` השלישי נקרא `set`. מדובר ב'קבוצה מתמטית' ללא סדר ועם הופעה יחידה. בדומה לכך קיימים שנקרא `sortedset` שבו דוחוקא יש סדר.

עד כה דיברנו על היעי של `collection`, וכעת נלקח ליעי השני, של ה-`interface` שנקרא `map`, שמקבל שני ערכים גנריים. דבר זה מאפשר למפות בין ערכים למפתחות. גם בפתחות כל ערך יכול להופיע פעם אחת, אך ערכים יכולים להופיע כמה פעמים. בדומה לכך, `sortedmap` זה `map` עם סדר.

Collection Implementations

ראשית, علينا לשים לב שימושיים לא יכולים להגיד בנאים. אך ככל זאת, ישנה קונבנצייה לגבי מבנה הבנייה של מחלקות אוסףים:

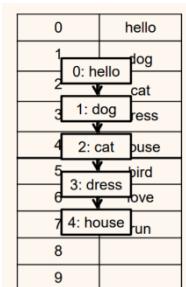
- א. `Builder` – בניית אוסף ריק.
- ב. `Copy-constructor` – מקבל אוסף אחר, ומכו尼斯 את כל האיברים שבתוכו, בסדר מסוים, לתוך הבנייה הנוכחי.

חשוב לציין שככל זאת אין דרך לכפות את הקונבנצייה הזאת.

כעת נעבור על כל אחד מהIMPLEMENTATIONS במשקדים שראינו קודם לכן.

1. `List` יש שניimplementations עיקריים:

- א. `ArrayList` – 배열 סדר מסוים. ניתן להוציא (`get`) או לשנות (`set`) איבר מסוים. או לקבל מידע האם האיבר נמצא (`contains`), מה האינדקס שלו (`indexOf`) ולמחוק אותו (`remove`). כל הפעולות הללו לוקחות $O(n)$. פעולה ההוספה (`add`) לוקחת $O(1)$ ולעתים לוקחת $O(n)$.
- על מנת להבין זאת נסתכל על עומק המימוש של `ArrayList`:



כאשר יש מקום במערך, זה י Mishik להיות $O(1)$ ואם אין מקום במערך, זה לוקח $O(n)$.

ב. `LinkedList` – רשימה מקוישת. כל הפעולות הללו – `get()`, `set()`, `contains()`, `indexOf()` – לוקחות $O(n)$.

`remove()` – לוקחות $O(n)$. זה קורה כי علينا לעבור על כל הרשימה עד שנגיע לאיבר הרצוי.

רשימה מקוישת לוקחת פחות מקום בזיכרון.

2. `Set` יש שניimplementations סטנדרטיים:

- א. `TreeSet` – למעשה מדובר במימוש של `sortedSet`, דברים שמסודרים בצורה בינארית וכל הפעולות הבאות: `add()`, `remove()`, `contains()` לוקחות למעשה $O(\log n)$.

ב. `HashSet` – מדובר במימוש של `hash table`. בדרך כלל כל הפעולות שדיברנו עליהם לוקחות $O(1)$.

בדומה ל-`Map` יש שניimplementations מתאימים למימושים של `set` ו-`TreeMap` – `set` של `HashMap`

לסייעים, נוכל להסתכל בטבלה הבאה:

	Add	Remove	Get by index	Contains	Iteration
ArrayList	O(1)*	O(N)	O(1)	O(N)	O(N)
LinkedList	O(1)	O(N)	O(N)	O(N)	O(N)
HashSet	O(1) avg	O(1) avg	—	O(1) avg	O(T)**
*TreeSet	O(logN)	O(logN)	—	O(logN)	O(N)

* amortized constant time, that is, adding n elements requires $O(n)$ time

** see later

הספריות מאפשרות לנו כל מיני מניפולציות, כגון searching, counting וכן הלאה. נוכל לראות זאת לשיכום בטבלה הבאה:

Collection algorithms:	Collection factories:	Collection Wrappers:
<ul style="list-style-type: none"> - min / max - frequency - disjoint List algorithms: <ul style="list-style-type: none"> - sort - binarySearch - reverse - shuffle - swap - fill - copy - replaceAll - indexOfSubList - lastIndexOfSubList 	<ul style="list-style-type: none"> - EMPTY_SET - EMPTY_LIST - EMPTY_MAP - emptySet - emptyList - emptyMap - singleton - singletonList - singletonMap - nCopies - list(Enumeration) 	<ul style="list-style-type: none"> - unmodifiableCollection - unmodifiableSet - unmodifiableSortedSet ... - synchronizedCollection - synchronizedSet - synchronizedSortedSet ... - checkedCollection - checkedSet - checkedSortedSet ...

Hash Table

טבלת גיבוב משמשת עבור חיפוש ומחיקה ייעילים. למשל, כיצד נוכל למצוא האם איבר מסוים נמצא ברשימה בצורה יעילה? באמצעות טבלאות גיבוב, נוכל לעבור מ-(n) O ל-(1).

פונקציה Hash זה פונקציה שמחזירה לנו מספר בטוחם מסוימים. בעזרה הפונקציה הזו, אפשר לבצע בטבלאות הגיבוב חיפושים יעילים.

נסתכל למשל על הדוגמא הבאה:

	car			goodbye		hello		dog	
0	1	2	...	k	...	j	...	N - 2	N - 1

אם נרצה להוסיף את המילה hello, נמפה אותה עבור j מסוים, ובתא ה-j נכניס את האיבר הזה. כך גם לגבי בדיקה אם goodbye נמצא. נוכל לעשות זאת באמצעות ה-k המתאים. בהמשך נדון לגבי התנשויות. בוגר לאיתרציה על טבלת גיבוב, מדובר למעשה על O(n) כאשר n זה גודל המערך.

איטרטורים

איטרטור הוא למעשה אובייקט שלוקח מבנה נתונים ועובד על כל אחד מהאיברים שלו. האופרטור מגדר שתי פעולות עיקריות:

.1 - האם יש עוד איבר קדימה.

.2 - בהינתן השאלה הקודמת הייתה נכון, נקרא לאיבר הבא.

```
List<String> myList = ...;
Iterator<String> myIterator = myList.iterator();
while ( myIterator.hasNext() ) {
    String next = myIterator.next();
    System.out.println(next);
}
```

Iterators are also generic
(Parameter must match
the collection parameter)

סיבות לשימוש באיטרטורים :

1. יוצר הפרדה בין המידע ובין הדרך בה עוברים עליו.
2. אותו איטרטור יכול להתאים לכמה מבני נתונים שונים.
3. אפשרות להגדרת דרכי שוננות לעבור על המידע.
4. במידה ואין סדר טבעי, האיטרטור מגדיר אותו.
5. איטרטורים יכולים להיות יעילים יותר.

Exceptions

שגיאות זה דבר שעליינו להתמודד איתו. עד כה הכרנו שגיאות קומפלציה, שנוח לנו לעבוד איתן כי הקומפילר בעצמו מספק מידע לגבי עצם השגיאה. הסוג המוסף יותר של השגיאות נקרא **שגיאות זמן ריצה**, והוא בעיתי יותר. שגיאות אלו יכולות להתקיים בתוכאה מבאים בקוד, או כתוצאה מקלט בעיתוי של המשתמש.

כיצד נוכל לטפל בשגיאות זמן ריצה? בעבר דיברנו על כך שתוכנית טובה מסוגלת להתמודד עם שגיאות, וכך במקום המתאים לכך.

חריגה – **Exception**, זו הודעה המודיע לנו שהתרחשה תקלה. (חריגות אלו עומדות מול 'ערכי החזרה') כאשר התרחשה שגיאה, המתוודה 'תזורך' חריגה. ב-Java חריגות הם למעשה אובייקטים. נסה לראות קוד לדוגמה :

```
int get(int index) throws ListException{
    if (list.isEmpty()) {
        throw new ListException();
    }
    //...
}
```

עלינו לשים לב כמה דברים :

- למילה השמורה **throw** שדווגת לזריקת השגיאה.
- למילה **throws** שיש בתחילת המתוודה. שימושה לב שמדובר במקרה מה-API של המחלקה שנוטן לנו מידע על סוג השגיאה שי יכול להתקיים במתוודה זו. למשל, אילו הן שגיאות אפשריות במתוודה זו. חשוב אף לתעד אותן ב-Javadoc.

שים לב שהסוג של השגיאה המוצג כאן הוא **ListException**.

נוכל לראות דוגמה נוספת להתמודדות עם שגיאה :

```
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
```

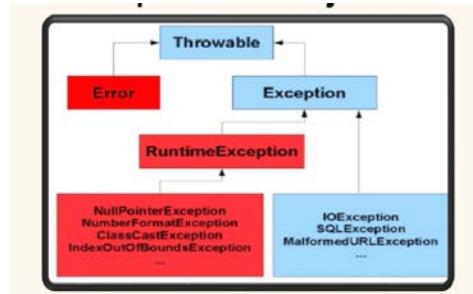


משמעות המילה **catch** הינה 'لتפוס' את השגיאה, במידה והיא מתקיימת. במקרה כזה, אנחנו למעשה יוצרים אובייקט של **ListException** כשהמשתנה יהיה **e**.

אפשר ליצור חריגות גם באופן עצמאי, ללא בлок של **try-catch**. במצב כזה נדרש להגדיר את המילה **throws** ששמורה בשם של המתוודה עצמה.

סוגי חריגות

ראשית, נتبונן בהיררכיה של החריגות:



- בקורס לא נ עסק בסוג הראשון של בעיות, כיון שלאו שגיאות מערכת.
- הסוג השני (האדום) הוא **unchecked errors** – קלומר בעיות שלא אמורות להתறחן ונוצרן לטפל בהם.
- הסוג השלישי הוא סוג שיכול להתקיים, והתוכניות צריכות למצוא פתרונות לבעיות אלו.
- נרחבicut על שני הסוגים האחרונים.

בדרך כלל, **checkedException** נובעת משגיאות של המשתמש.

1. בرمאה הטכנית, כשמתוודה מחייבת לזרוק חריגה שכזו, היא חייבת לכלול **throws**.
2. חייבים ליצור את החריגה באופן יוזם.

לעומת זאת, **unchecked errors** נובעות מבעיות בעת כתיבת הקוד. שגיאות אלו עלולות להתறחן במקרים שונים. (בנוגע ל-**NullPointerException**, בಗל שזו שגיאה כל כך נפוצה, אין צורך לכלול את **throws**).

ישנן שלוש סיבות עיקריות לשימוש בחיריגות:

1. ליזור הפרדה בין הקוד הרגיל לקוד שמתפל בשגיאות.
- נוכל להסתכל על בעיה פשוטה, של קריית מידע:

```

public int readData {
    if (!ask the data size)
        return -1;
    if (!allocate memory)
        return -2;
    if (!read data)
        return -3;

    return 0;
}
  
```

באמצעות חריגות, הקוד הזה יהיה אלגנטיבי יותר:

```

public void readData {
    try {
        ask for data size;
        allocate required memory;
        read the data from user into memory;
    }
    catch (DataSizeException e) { doSomething; }
    catch (OutOfMemException e) { doSomething; }
    catch (ReadDataException e) { doSomething; }
}

```

במצב זה ישנה הפרדה ברורה בין החלק המבצע את המשימה והחלק שמתפל בשגיאות.

2. לפעוף שגיאות במעלה 'מחסנית הקראיה'.
אם למשל ישנו רצף של קריאות לפונקציות, רק ב'פונקציה האخונה', תתקיים השגיאה:

foo₁() → foo₂() → ... foo_n() → error!

זה מה שונעשה, באמצעות חריגות, לעומת זאת:

```

foreach 1 <= i < n:
public void foon throws SomeException() {
    // Some foon() code
    if (error)
        throw new SomeException();
    // Some more foon() code
}
public void fooi throws SomeException() {
    // Some fooi() code
    fooi+1();
    // Some more fooi() code
}

```

אם קורתה שגיאה באחת המתוודות הקודומות, פשוט עברו הלאה.

3. לאחד בין סוגים של שגיאות ולהפריד בין סוגים אחרים.

עלינו לשים לב שהחריגות לא אמורים להחליף פעולות וגילות של תכנות! מנגנון של לולאות יעל הרבה יותר מנגנון של באגים, ובנוסף, המנגנון הזה אינו צפוי ועלול להחביא בתוכו גם באגים.

שימושים לא נפוצים בחריגות:

1. חזרה מקראית عمוקה של פונקציות.D – Timeout – כתיבת קוד לזמן ריצה מסוים.
2. –UnsupportedOperationException – ראיינו בעבר כי בכל הממשקים של Collection ישן מתודות אופציונליות, שנitin למש במקרים מסוימים. הדך למש את המתודות האופציונליות האלו, היא באמצעות חריגות.

Packages – חבילות

חבילה היא דרך לחלק את הקוד לחולקות הגיוןיות. אפשר להסתכלAnalogy על תיקיות במחשב. דבר זה מאפשר לחלק את הקוד למודולים שונים. זה הגיוני כאשר נרצה ליצא את הקוד, למשל, או לאפשר הראות מסוימות.

ונכל לראות דוגמה לחברה את Collection framework אשר נמצא בתוכה Java.util. הרכיבים של החבילה הוזת שייכים לה, וחולקים תכונה סימנטית – כולם עוסקים במבנה נתונים.

נתבונן בכמה דוגמאות של משתנים מסוג חבילה:

1. בלי הרשאה למחלקה A וצריך ליבא את החבילה :

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

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

Modifier	Class	Package	Subclass	World
default(=package)	Y	Y	N	N

Error!

2. ו-B באותה החבילה ולכן הרשאה מוצלחת :

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

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

Modifier	Class	Package	Subclass	World
default(=package)	Y	Y	N	N

3. בדומה לדוגמה הקודמת, אך עם - גישה מוצלחת :

```
package pack1;
public class A {
    protected int protectedInt;
}

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

Modifier	Class	Package	Subclass	World
protected	Y	Y	Y	N

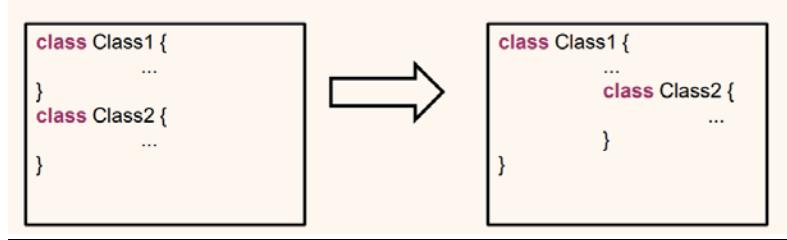
ניר שnochן ליצור מחלקות ללא modifier, דבר זה גורם לכך שרק מחלקות מאותו החבילה יכולות לגשת למחלקה זו. למעשה, מחלקות ה-public מדיריות את ה-API של החבילה.

הקשר בין חבילות לחריגות

באופן כללי, עדיף לשים כל חריגה בחבילה הרלוונטית שלה, ולא בתורן בחבילה אחרת.

מחלקות מקווננות – Nested Classes

כאשר אנו באים לדבר על מחלקות מקווננות, אנחנו מתיכוונים למחלקה בתוך מחלקה. עד כה, כל מחלקה הוגדרה בתוך קובץ עצמאלה. מחלקות מקווננות מאפשרות להגדיר מחלקה אחת בתור אחרת :



מתארים את המחלוקת החיצונית בתור Outer Class או Wrapping Class Enclosing Class או המחלוקת הפנימית נקראת Nested Class.

מה הוצרך במחלקות מקוונות :

1. קיבוץ לוגי של מחלקות – אם למשל מחלוקת רלוונטי רק בהקשר של מחלוקת אחרת, אין צורך להגדיר אותה בתור Public.
2. הגדלת הancock솔ציה – כאשר אנחנו מגדירים מחלוקת אי בתור מחלוקת ב', אנחנו מאפשרים גישות למשתנים הפרטיים, ודבר זה אפשר לעשות דרך מחלוקת המקוונת. יתכן שגם המחלוקת הפנימית תהיה Private, וזה ממשיך עם החבאת המידע של Hancock솔ציה.
3. קוד שסביר יותר עצמו – אמנס המחלוקות המקוונות מעט מסורבלות, אך בעניינים זרות ניתן להבין את העיצוב טוב יותר (עצם הפנימיות של המחלוקת מעיד על המהות שלה).

מהם התנאים לשימוש במחלקות פנימיות :

1. מחלוקת קטנה – אם מחלוקת גדולה מדי, היא הופכת את העניין למסורבל.
2. מחלוקות Private – לא בהרשאת Public.

מהם ההרשאות למחלוקות פנימיות?

- למחלוקת פנימית יש גישה לכל המשתנים של המחלוקת העוטפת, וכן להיפך.
- המחלוקת הפנימית עצמה יכולה לקבל כל אחד מה- Modifiers : Public, Private, package, Protected

סוגים של מחלוקות פנימיות :

- הסוג המרכזי נקרא static Nested Class. מדובר למעשה במחלקה שהייתה יכולה להיות מוגדרת בקובץ נפרד, אבל מצרכי 'חבילות', החלתו לשים אותה באותה מחלוקת. אך אין קשר בין ה- Instance של המחלוקת החיצונית והפנימית. על מנת שנוכל לגשת למשתנים של המחלוקת העוטפת, יש לתת instance ספציפי, כפי שנוכל לראות בדוגמה הבאה :

```

public class EnclosingClass {
    private int dataMember = 7;
    private static class NestedClass {
        private int nestedDataMember = 8;
    }
    public static public void main(String[] args) {
        // No need to create an instance of EnclosingClass
        NestedClass nested = new NestedClass();
        System.out.println(nested.nestedDataMember);
    }
}

```

- מחלקת פנימית שאינה סטטית. מדובר במחלקת שמקושרת ל-`instnace` ספציפי של המחלקה העוטפת. בגלל שהיא מקושרת למחלקה ספציפית, היא לא יכולה להגדיר משתנים סטטיים.

- הסוג הכי נפוץ של Inner Class נקרא Member Class. עליו קודם כל ליצור של המחלקה החיצונית, אז יוכל ליצור Instance של המחלקה הפנימית :

```

public class EnclosingClass {
    private class MemberClass {
        private int memberClassField = 8;
    }
    public void foo() {
        // a MemberClass object is created with reference
        // to a specific instance of OuterClass
        MemberClass innerObj = new MemberClass();
    }
}

```

- האם לשים חריגות בתוך מחלקות מקווננות? עדיף שלא, כיון שהריגות הן בדרך כלל חלק מה-.API

- מדבר על מחלקת בתוך מתודה. בדומה למשתנים מקומיים, גם מחלקת נגישה רק בתחום אותה מתודה. متى משתמש בה? כאשר מחלקת רלוונטית רק ביחס למתודה מסוימת. למשל, כמו בדוגמה הבאה :

```

public class StringLinkedList {
    private Node head;
    ...
    public Iterator<String> elements() {           // This method creates and returns an Iterator object
        class ListIterator implements Iterator<String> {           // Definition of the local class
            Node current;
            ListIterator() { ... } // inner class Constructor
            public boolean hasNext() { ... }
            public String next () {...}
        } // end of local class ListIterator
        return new ListIterator();
    }
}

```

- אפשר לשים לב שرك המתודה יכולה לעשות שימוש במחלקה זו. היא מחזירה למעשה 'איטרטורי' של `String`. מחלקות פנימיות יכולות לגשת למשתנים במתודה, רק כאשר המשתנים האלו מוגדרים כ-`Final`.
- הגבלות על מחלקות מקומיות :

1. אפשר לראות אותה רק בתחום המתודה המגדירה אותה.
2. בغالל שהיא מוגדרת בתחום מתודה, אין משמעות ל-`Modifier` שלה.
3. כמו כל המחלקות הפנימיות, היא אינה יכולה להגדיר משתנים ספציפיים.

4. אי אפשר להגדיר ממשקים להיות מחלקות מקומיות.
 – מדבר במחלקה ללא שם, מכיוון שהיא ספציפית לחלק מסוים בקוד.
- דוגמה לשימוש במחלקה אונומית :

```
String[] filelist = dir.list(
    new FilenameFilter() { // Creating an instance while
        // implementing the class
        public boolean accept(File dir, String s) {
            return s.endsWith(".java");
        }
    } // end of class declaration
); // end of the statement of calling dir.list()
```

שימוש לב שיצרנו Instance של Interface, אך למעשה יצרנו מחלקה אונומית שמשמשת תהו
המשמעות הזה.

Closures

Closure הינו בлок של קוד, שיש לו קשר לא טריוויאלי עם משתנים שלו. בדרך כלל כשאנו מדברים על Closure אנחנו מדברים על פונקציה בתוך פונקציה, למשל, כמו בדוגמה הבאה (בפייתון) :

```
def counter():
    # A local variable
    x = 0

    # A nested function
    def increment(y):
        # Access non-local
        # variable
        nonlocal x
        x += y
        print(x)

    return increment
```

כאנלוגיה, אפשר לומר שמחלקה מקומית היא כמו Closure. אמנם, מחלקות מקומיות יכולות רק למשתנים הקבועים, ולכן הם לא בדוק כמו Closure. בגרסת Java 8 קיימת אפשרות דומה יותר ל-Closure של Java.

מודולריות

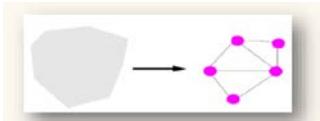
הרענון של 'מודולריות', הינו לפרק את התוכנית שלנו לחלקים קטנים, שנקראים מודולים.

ה יתרונות של אפין מודולרי :

1. קל לתזוזק.
2. פירוק בעיות מורכבות לעביעות קטנות.
3. פירוק בעיות מורכבות לצורות שונות באופן בלתי תלוי.

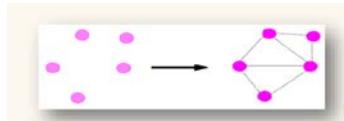
במודולריות קיימים ארבעה עקרונות משמעותיים :

1. **פריקות – Decomposability**: פירוק בעיה מורכבת לתתי בעיות קטנות, וחיבור באמצעות מבנה פשוט.



דבר זה מוביל לאפשרות לחלוקת עבודה.

2. Composability – הרכבה. עיצוב שمبוסס על הרכבה, הוא עיצוב שבו אנחנו יוצרים רכיבים שונים לקחת אותם ולשלב אותם כדי לבנות יחידות מורכבות יותר.



על הרכיבים ברגע זה להיות עצמאיים. רעיון זה מתקשר לעניין של 'מחזור קוד' (אם הם עצמאיים וקטנים, ניתן להשתמש בהם שוב ושוב). כabilות תוכנה הן דוגמה טובה לרגעון של הרכהבה.

- לכארה, העקרונות של פריקות והרכבה סותרים, אך לפחות ניתן לשלב בין שני הרענוןות האלו.

3. Understandability – מובנות. הרעיון שבעיצוב מודולרי הינו שכל אחד יוכל להסתכל על כל אחד מהמודלים בנפרד. (דרך לבדוק האם הוא אוטונומי – האם אנחנו יכולים לתאר אותו בכמה מילויים?)

- מובנות איננה דומה לקריאות. קריאות היא ברמת הקוד, מובנות היא ברמת העיצוב.
- רציפות. במידה ויש שינוי בדרישות, נרצה שהוא ישנה מודול אחד בלבד.

עקרונות נוספים המתקשרים למודולריות:

1. Open-Closed – פתוח-סגור. רכיבי תוכנה צריכים להיות סגורים לשינויים ופתוחים להרחבה. על מנת שלא נפגע בקוד, נרצה קוד שלא משתנה, ושנינו לכתוב לו קוד חדש המרחיב אותו. אם נרצה, יוכל להסתכל על מערכת שיצרת צורות. נழוד על היתרונו של הגישה הזאת, כשהנגיף אותה לשיטה הפראזדורליית הבאה:

```
void drawAll(Shape[] list) {
    for (int i=0; i < list.length; i++) {
        Shape s = list[i];
        int type = getType(s);
        switch (type){
            case SQUARE:
                drawSquare((Square)s); break;
            case CIRCLE:
                drawCircle((Circle)s); break;
        }
    }
}
```

קוד זה אינו ניתן להרחבה והוא מועד לצורות. לעומת זאת, פיתרון עם שימוש במונחה עצמים משמשו יותר נוח יותר:

```

public interface Drawable{
    public void draw();
}
public class Square implements Drawable {
    public void draw() {...}
}
public class Circle implements Drawable {
    public void draw() {...}
}
public void drawAll (Drawable[] list) {
    for (Drawable drawable: list)
        drawable.draw();
}

```

קיבלוינו עיצוב שהינו פותח להרחבות מחד, ומצד שני, הפונקציה של הציגו בעצמה סגורה לשינויים. Single-Choice – מזעור כמות השינויים שנעשה במודלים – אם נדרש לשנות, נעשה את השינוי במקום אחד. זה עיקרונו המתכתב באופן ישיר עם עקרון הפתוח-סגור.

בחזורה ל-Factory – מחלקת Design patterns

Factory הוא למעשה סוג של מפעול. הרעיון המרכזי הוא ליזור הפרדה בין 'הקוד שמייצר את האובייקטים', לבין 'הקוד שימוש בהם'. רעיון זה כМОובן מתקשר למודולריות.

המפעלי יכול לייצר את האובייקטים בצורה דינמית, או מראש (Objects pool), ולמעשה לא מדובר בשם ספציפי אלא בקורספט כללי שמננו נגזרים כל מיני Design patterns. נתבונן בדוגמה קטנה של שימוש במפעלי – בייצור מראש :

```

public class ShapeFactory {
    private static final SQUARE_OBJECT = new Square();
    private static final CIRCLE_OBJECT = new Circle();

    public Drawable[] loadAll (String[] list) {
        Drawable[] drawables = new Drawable[list.length];
        for (int i = 0 ; i < list.length ; ++i) {
            if (list[i].equals("Square")) drawables[i] = SQUARE_OBJECT;
            else if (list[i].equals("Circle")) drawables[i] = CIRCLE_OBJECT
            ...
        }
        return drawables;
    }
}

```

. Single-Choice גם הוא ל-

Singelton

בדומה למפעלי, מדובר על pattern של יצרה, אך אמנים במקרה זה אונחנו רוצחים יישות אחת ויחידה בלבד של אותו אובייקט, ושתיהיה גישה נוחה אליו. הפיתרון לכך הוא להחזיק משתנה סטטי, שהבנאי של המחלקה זו יהיה Private, ולהחזיק מתודה סטטית שנקראת instance, שתמיד מחזירה את אותו משתנה סטטי :

```

public class Singleton {
    private static Singleton single =
        new Singleton();

    private Singleton () { ... }

    public static Singleton instance() {
        return single;
    }
}

```

נשים לב שכיוון שהבנייה הוא Private, אין מחלוקת שיכולה לרשת ממנו. סיבה לכך היא שבעצם כל מחלוקת היורשת היא חלק מהאב, ודבר זה אינו אפשרי במקרה שלנו.

אלטרנטיבית לSingleton, היא יצירת מחלוקת של متודות סטטיות. החיסרונו בReLU זה הוא שהמתודה ב-Singleton היא לא סטטית, ומעבר לכך היא יכולה לרשת מחלוקות או למשמש משקדים – פולימופרפיים והחבהת מידע.

Strategy Design Pattern

כאשר המערכת שלנו ישנה מערכת של אלגוריתמים או התנהוגיות המדברים על תוכן המערכת, ונרצה להפריד אותה מהמערכת עצמה.

פורמלית, הביעה ב-Strategy, היא שיש לנו מטרה מסוימת עם אלגוריתמים שונים אפשריים (למשל, מיון). ביעיה נוספת היא סט של התנהוגיות המשתנה ביחס לקלט של המשתמש.

הפיתרונו לכך הוא הגדרת API (באמצעות מחלוקת מופשטת או ממשק), וכל מחלוקת מימוש את אותו ה-API. מהפיתרונו הזה מקבלים כי את ההתנהוגות המורכבת אנחנו מעבירים לחלקים קטנים. למעשה, אנחנו יוצרים את ההתנהוגות באמצעות factory. נוכל לראות דוגמה לכך כאן:

<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>
--	---

עקרונית, הינו יכולים להשתמש בחבילות שיורשות את כל אחד מהמיונים, אך יש כאן מספר בעיות:

1. חבילה היא לא סוג של המילון.
2. אין מודולריות.
3. אין הסתרת מידע.
4. בעיות של שימוש מחדש בקוד.
5. אין אפשרות לשינוי בזמן ריצה.
6. מחלוקות ירושות רק ממחלוקת אחת.

נבחן כי גם Strategy משתמש בReLU של פתווח-סגור ושל בחירה-יחידה.

מה הקשר בין Strategy-factory, singleton ו-singleton? נבחן בדוגמה פשוטה:

<pre>public class QuickSort implements SortStrategy { private static QuickSort instance = new QuickSort(); private QuickSort () { ... } public static QuickSort instance() { return instance; } public void sort(Comparable[] data) {...} }</pre>	<pre>public interface SortStrategy { void sort(Comparable[] data); } public class SortStrategyFactory { public static SortStrategy select(...){ if (cond1) return QuickSort.instance(); else if (cond2) return MergeSort.instance(); } }</pre>
--	--

Streams

הרעיון של stream אומר למעשה כי התוכנה שלנו מקבלת מידע ושולחת מידע, לגורמים שונים. למעשה, ישנו מקורות מידע שונים, ובעקבות לכך עליינו לנסוט להתאים את התוכנה לכל אחד ממקורות המידע האלו. אפשר לפעול בצורה נאיבית, אך עדיף למצוא את המשותף בין המקורות ובכך להתמודד עם בעיה זו. המשותף הוא למעשה – קבלת מידע מה-stream – צינור המידע, ושליחת מידע אליו.

ספריית Java Stream מאפשרת לקבל מידע ממוקורות שונים, בלי להבחין בסוג המידע הזה (דבר המאפשר גם אנקטולציה). אם נרצה, נוכל כאנלוגיה לחשב על ברז – איננו יודעים על מקורם, אבל בסופו של דבר מים זורמים בו. בצורה דומה, ניתן לחשב על זרימות המים בכירור.

פורמלית, אלו הפעולות שנctract:

- .1. יצירת צינור המידע – .Create
- .2. כתיבה למקור המידע – .Write
- .3. קריאה ממקור המידע – .Read
- .4. סגירת מקור המידע – .Delete

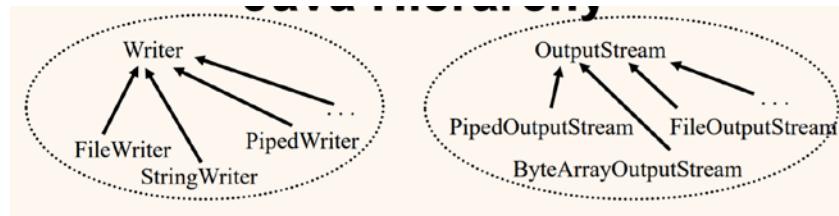
לפי שנדבר על קריאת או כתיבת מידע, ניגע קודם כל בשאלת מהו מידע. ישנו למעשה שני סוגי של מידע:

1. מידע טקסטואלי – רצים של אותיות. קבצי Word למשל, או קבצי XML.
2. מידע בינארי – רצים של ביטים (8 ביטים, ככל בית יכול לקבל 0 או 1). לא מדובר דזוקא על טקסט, אלא ישנו סוגים שונים.

כאשר אנחנו עובדים עם Streams, אנחנו יוצרים תקשורת מסוימת עם מישחו, וכך עליינו ליצור שפה משותפת (קיזוז) של המידע. ההחלטה הראשונה היא האם מדובר במידע טקסטואלי או במידע בינארי.

בrama הפרקטיית, קיימת ב-Java חיבור הנקראת `java.io`, ובתוכה שתי מחלקות, Reader ו-Writer לעבודה עם טקסט ושתי מחלקות, InputStream ו-OutputStream לעבודה עם מידע בינארי. מדובר במחלקות אבסטרקטיות, וקיימות מספר מחלקות שיורשות מהן.

בצורה סכמטית, ניתן לראות זאת כך:



קונקרטיות, נוצר ליצור Instance של אחת מהמחלקות האלו ולבוד אותו. כך למשל (הסימות לא הכרחית) :

```

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

```

במבט על, נבחן במחלקות הקיימות ב-Java לקריאה ולכתיבה של קבצים :

I/O Type	Streams
Memory	<i>CharArrayReader/Writer</i> <i>ByteArrayInput/OutputStream</i>
Files	<i>FileReader/Writer</i> , <i>FileInput/OutputStream</i>
Buffering	<i>BufferedReader/Writer</i> , <i>BufferedInput/OutputStream</i>
Data Conversion	<i>DataInput/OutputStream</i>
Object Serialization	<i>ObjectInput/OutputStream</i>
Filtering	<i>FilterReader/Writer</i> , <i>FilterInput/OutputStream</i>
Converting between bytes and characters	<i>InputStream/OutputReader</i>

נتابון בדוגמה של העתקה קובץ א' לקובץ ב' :

```

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");
}

```

מדובר בבלוק של Try ו-Catch על מנת לטפל בעיות של IO. במידה וקרויה שגיאה במהלך אחת הפעולות, הקובץ איננו נסגר! ולכן, ב-Java 7 ומעלה, יוצרים את האפשרות הבאה:

```

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!)

```

באמצעות אפשרות זו והעובדת שהמחלקה הזו מימוש משק הנקרא AutoCloseable, המחלקות נסגורות אוטומטית, גם במקרה של שגיאה.

– מDecorator

כשכתבו ל-stream כלשהו, נעדיף כתוב פחרות, לשמור מקום בדיסק ומסיבות נוספות. על מנת לצמצם במידע, נרצה לכוזץ אותו. אמנם, לעתים מדובר במקורות מידע שונים, שקשה לכוזץ אותם באופן אחד, מבחינות חוזרות של קוד וכן הלאה. עיה נוספת שללה לקרותה הינה שקריאת וככיתה של כל המידע בקובץ איננה יעילה בכלל – המנגנון של קריאת וככיתה במערכת ההפעלה איננו יעיל. העובדה שפעולות המערכת buffer הפעלה שוות ביעילותן למגוון COMMANDS הנדרשת, יוצרת לנו פיתרון. כמובן, יוכל כתוב פתרון מוקומי, כשיתמאל נכתוב אותו במתוך אחת, דבר שהוא יעיל יותר. אמנם, ישנו עיכוב בין כתיבת המידע לתוך הימיכל הזה, ובין כתיבת המידע לדיסק עצמו. פיתרון זה מתאים לכל המקורות השונים.

נגיד את הבעיה הזאת כתעתת בדינה פורמלית. קיימים לנו אוסף של Streams, ואנחנו רוצים להוסיף להם יכולות נוספות רבות.

על מנת לתאר את הפיתרון, נזכיר ברעיון של דילגציה, שמאפשר מחזר קוד ללא ירושא.

בrama הפורמלית, אנחנו רוצים לקחת B ולהוסיף לה יכולות נוספות של מחלקה A. מחלוקת A תירש למעשה את מחלקה B ויש לה את אותו ה-API. מכאן, היא מחזיקה אובייקט מסווג B ומעבירה אליו בקשנות. העובדה ש-A יורש מ-B מאפשרת ליצור Decorator נוסף בסוף ל-A.

בצורה גרפית, זה נראה כך:



וכך בצורת הקוד:

```
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

ולגבי כיווץ:

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

נבחן כי המחלקות המKeySpecות לא מקבלות מידע בעצמן. בcontra מידע, המחלקות שמקבלות מידע אינם מKeySpecים, לא בrama הרעיוןית ולא בrama הפרויקטית.

Scanner

מדובר על מחלוקת נפוצה ב-Java המשמשת לקרוא מידע טקסטואלי, והיא מקבלת אבן InputStream. היא מעבירה מחלוקות לבקשה זו, ויש לה מתודות שימושיות לניטוח טקסט. אמנם, היא איננה נחשבת למKeySpec, כיון שהיא לא יורשת את המחלוקת InputStream.

Generics

בחלק זה נדבר על הפשטה מעל טיפוסים שאינם פרימיטיביים. יוכל לדבר על שני שחקנים מרכזיים:

1. מחלקות.

2. פרמטר כללי – שאוטו יכולות להחליף מחלקות או מערכים.

המטרה המרכזית של Generics הינה להגדיר Type safety – תכונה שהופכת את הקוד לחסין מבאים. מאפשר קריאות של הקוד. נבחן כי לא משתמש ב-Generics Collection רק ב-Generics על מנת להבין את Generics טוב יותר, נדבר באופן כללי על 'מניעה' במקום 'טיפול'. למעשה, ככל שנזהה טעות בשלב מוקדם יותר, 'ישלים' עלייה פחותה. למעשה, מאפשר לנו לזהות בשלב מוקדם יותר – בפרט שגיאות טיפוס.

נזכיר כי קיימים שלושה סוגי טיפוסים :

1. פרימיטיביים.

2. מחלקות.

3. מערכים.

מחלקות יכולות להיות למעשה גנריות, וכך גם מערכים יכולים להיות כאלה.

על מנת להבין מהי שגיאת טיפוס, נתבונן בקוד הבא :

```
String s1 = new Integer();
```

Compile-time type error

דבר זה נקרא 'שגיאת טיפוס', כי אנחנו מנסים להתאים טיפוס לא מתאים לטיפוס כלשהו. דבר זה טריוואלי, אבל הפעולה הבאה טרייוואלית פחותה :

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

run-time type error

בעקבות העבודה שאנו מנסים לעשות Down-cast לאובייקט שאינו מותאים, תהיה לנו שגיאת ריצה (שהקומpileר לא יידע לנו). שגיאה מורכבת למדי. Generics מאפשר לנו למנו שגיאות כלשהוא ולזהות אותם בזמן קומPILEציה – זאת למעשה ההגדלה של Type safety Generics. אם כך, מהוות את הדריך להגעה ל- Type safety .

על מנת להבין זאת טוב יותר, נתבונן בדוגמה קונקרטית :

```
/** My stamp collection. Contains only  
 * Stamp instances. */  
private final Collection stamps = ... ;  
  
// Erroneous insertion of coin into  
// stamp collection  
stamps.add(new Coin( ... ));
```



```
// Now a raw iterator type - don't do this!  
for (Iterator i = stamps.iterator(); i.hasNext(); ) {  
    Object o = i.next();  
    Stamp s = (Stamp) o; // Throws ClassCastException  
    ... // Do something with the stamp  
}
```

לפנינו שונעbor ל- Generics. נתבונן בדוגמה מוכרת לנו יותר, בדרך בטוחה יותר :

```

/** My stamp collection. Contains only
 * Stamp instances. */
private final Collection stamps = ...;

// Erroneous insertion of coin into
// stamp collection
stamps.add(new Coin(...));


```

```

for (Iterator i = stamps.iterator(); i.hasNext(); ) {
    Object o = i.next();
    if (o instanceof Stamp) {
        Stamp s = (Stamp)o; // Safer: no
                            // ClassCastException
        ...
        // Do something with the stamp
    } else {
        throw new SomeException(...);
    }
}


```

אך אמם, נשים לב כי אנחנו משתמשים כאן ב-`InstanceOf`, שהסבירנו כי נעדיף שלא להשתמש בהם. בנוסף, הקוד הזה אינו קרייא וגם קיימת בו חזרתיות קוד. מעבר לכך, עדין מדובר בשגיאת זמן ריצה (אנו יודעים לחתמודד אותה, אבל עדין מעדיפים שלא הגיעו לכך).

הפתרון לכך הוא :

```

/** My generic stamp collection. Contains only
 * Stamp Instances. */
private final Collection<Stamp> stamps = ...;

// Erroneous insertion of coin into stamp collection
stamps.add(new Coin(...));
// Compilation error:
Test.java:9: add(Stamp) in Collection<Stamp>
cannot be applied to (Coin)
    stamps.add(new Coin());
^


```

```

// for-each loop over a parameterized collection –
// type-safe
for (Stamp s : stamps) {
    // No cast ...
    // Do something with the stamp
}


```

✓Short
✓Readable
✓Type-Safe!

הבעיה כבר מוצגת בזמן הקומPILEציה, וגם קיימות לנו הדרכים לטפל בה. במצב זה קיבלו קוד קצר וקרייא יותר.

כעת נסח לכתב מחלוקת של :

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>

כעת, על מנת להשתמש בקוד, נגיד רצית את כך :

<pre> Node n = new Node("hello"); ... String s = (String) n.data(); </pre>	<pre> Node<String> n = new Node<String>("hello"); ... String s = n.data(); </pre>
--	---

אם נרצה, יוכל להתבונן בדוגמא נוספת :

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

נעיר כמה הערות לגבי פרמטרים גנריים :

- נגידר פרמטר גנרי באטען סוגרים משולשים עם T בamu.
- כתה, השתמש ב-T שהוא הגדרנו.
- דבר זה יכול להיות גם וקורסיבי.

נבחן כי Generics הם אינווריאנטיים. כלומר, אם יש רשיימה מקוועת של String, ורשיימה מקוועת של Object, אי אפשר לעשות Cast ביןיהם. אולם, אין דבר זה אומר שאי אפשר להוסיף מחרוזות לרשימה מקוועת של אובייקט.

על מנת להתמודד עם ה'בעיה' של האינווריאנטיות, הומצא המושג הבא.

WildCards

כאשר נרצה לכתוב מתודה, שעובדת עם רשיימות גנריות אך אין חשיבות לסוג הספציפי, נוכל להשתמש ב->? . איננו יכולים להניח שום דבר על הסוג האמתי, מלבד העובדה שהם יורשים מ-Object. לרשיימה זו נוכל להכניס null בלבד.

מדוע נוכל להשתמש באפשרות זו? בדוגמה הבאות :

1. כשרק מספיק :

- Object is enough

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

2. כאשר Object אינו נדרש :

```
public boolean isEqualSizes(List<?> c1, List<?> c2) {
    return c1.size() == c2.size();
}
```

3. להפוך את ה-null לפיזיר :

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

4. כשנרצה להחזיר רשימה גנית, בלי שנדע מה הסוג :

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

למה לא נרצה להשתמש ב-`List<Object>`? האופציה הראשונה תעבוד במקרה זה, אבל בעקבות האינווריאנטיות, לא יוכל להדפיס רשיימה של מחזוזות. אם נרצה להשתמש ברשיימה שאיננה גנריית, קיבל רשיימה שאיננה Type-Safe.

ניגע בצורה עמוקה יותר בתכונות של `List<?>`:

- כיון שאינו יודעים מהו הסוג הגרפי, לא יוכל להוסיף משהו מעבר ל-`null`.
- יוכל לשולף רק `Object`, כיון שאינו יודעים דבר על הפרמטר הגנרי, מלבד העובדה שהוא יורש מ-`Object`.
- רק יכולים להשתמש בו:

- `LinkedList<?> c1 = new LinkedList<String>(); // Legal`
- `LinkedList<?> c2 = new LinkedList<?>(); // Illegal`

UpCasting ו-WildCards -

`List<? Extends Animal>`

נשתמש במקרה זה, כשרצהנו אובייקטים שאנו יודעים שכולם יורשים מ-`Animal`. על מנת שנוכל לעשות סוג של UpCasting, יוכל לבצע את הדבר הבא:

`LinkedList<? extends Animal> myList = new LinkedList<Dog>();`

באמצעות אפשרות זו, יוכל להשתמש ב-Casting, עבור מחלקות גנריות:

1. ביצוע פעולות ספציפיות למחלקה:

```
void printAnimals(List<? extends Animal> c) {
    for (Animal e : c) {
        e.printEars();
        e.printNose();
    }
}
```

2. שימוש גם עבור ממשקים (`extends` עובד גם עבור ממשקים):

```
Void makeAllJump(List<? extends Jumpable> c) {
    for (Jumpable e : c)
        e.jump();
}
```

לא יוכל להשתמש ב-`List<Animal>` כי האינווריאנטיות לא תאפשר זאת.

לסיכום של תכונתו:

1. יכול להיות מואתך עם אובייקט ספציפי.
2. ניתן להוסיף רק `null`.

Erasur

זו למעשה הדריך בה Generics ממומש ב-Java. למעשה, בתחילת הקומpileציה שתרחש ב-Java, מתרחשים הרבה פעולות מאחור הקלעים, שמטרתם להפוך את ה-`List<String>` ל-`List`, דרך מיימושים שונים. מדובר בסוג של תרגום. היתרונות של הדריך זה הוא שהוא מאפשר גם גרסאות ישנות של Java.

אפשר לראות זאת בדוגמה הבאה:

Generic Node

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

: Erasure

Code after Erasure

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

אם כך, מה צריך בכל התהליך שעשינו עד כה? נבחן כי שתי המטרות, Type-Safe וקריאות-קוד, עדיין קיימות למורota Erasure.

מהן ההשלכות של תהליך Erasure?

- כשרצה להשוות בין 'קבالت מחלקות', נקבל מחלקות שוות, גם אם מדובר ב-Generics שונים.
- אין צורך לשאול על instanceof, בגלל תהליך זה.
- אין טעם לעשות List<String> DownCast, מאותה סיבה.

הקשר בין Generics ומערכות :

מערכות הם למעשה Covariant, כלומר מערך של אובייקט מסוים, יירוש מערך של אביו של אותו האובייקט. דבר זה עלול לגרום לביעיות, למשל כך נוצרת שגיאת ריצה :

```
String[ ] strArray = new String[10];  
Object[ ] objArray = strArray;  
objArray[0] = new Integer(5);
```

ולכן מסיבה זאת איננו יכולים להכניס מערכים ברשימות עם Generics.

Generics איננו משפיע לנו על הקוד – רק הופך אותו לבטוח יותר. קוד גנרי איננו קוד כלל-יוטר, שהרי האפשרות האחראית היא לעבוד עם Object, שהוא כמעט כללי הרבה יותר. מאידך, נבחן כי קוד גנרי איננו מוסיף פונקציונליות לקוד שלנו.

Regular Expressions

אחד הכלים השימושיים בגיאוח, הוא ביטויים רגולריים. מדובר בתחום שיש מתחוריו הרבה תיאוריה, אך אנחנו נדבר על המובן הפרקטני שבענין.

מבוא

נניח ונרצה לכתוב מתודה שמקבלת מחרוזת, ובזוקת האם היא בפורמט של תאריך. כיצד נוכל לעשות זאת?

```
/*
 * A method that returns true iff the given string is a legal
 * date of the format dd/MM/yyyy
 */
boolean isDate(String s) {
    ???
}

• 01/05/2012
• 15/15/1214
• Hello
• 1a/02/2011
• 111/01/2012
```

ניתן למעשה להגיע לתשובה אחת שטעה על דבר זה.

מהם ביטויים רגולריים? מדובר בסוג של תבנית שנitin להשוות למבנה, ולשאול האם הטקסט מותאים לביטוי או לא.

אם התשובה היא שהtekst מותאים, אפשר לשאול אילו חלקים מותאים.

השימושים הנפוצים לביטויים רגולריים: חיפוש בטקסט, אפליקציות של שפה, יצירות דפי ווב.

ראשית, נראה כיצד ביטויים אלו נראים:

```
[A-Za-z]+ (or)? .*$  
[^A]+a+[^a]+ (.*) .+ (.*)  
^ABC]+d*$  
Cats? and rats?  
([a-z]{1,4}?[A-Z1-9] (d+)_ (d+).(d+)
```

סינטקס בסיסי

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	[cb]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 aaa, caaa,
...		...

עוד דבר שnochל לדבר עליו, הוא כתמים, שבודקים כמה פעמים לשים כל איבר:

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

אם כך, נבדוק למשל את כל המופיעים של המילה `rabbit`, שניתן להכפיל כל אחת כמה פעמים. הדרך פשוטה היא באמצעות האופרטור `+`:

סוגי חיפוש

כדי להבין על מה מדובר, ניקח את הביטוי הרגולרי `[a-z]+` שימושו הינה אותן כלשהיא מבין הטוווח הנ"ל שמופיעה פעם אחת או יותר. נבדוק את הטקסט "Now is the time" - הביטוי הזה כולל איננו מתאים לטקסט, כי בו אותיות גדולות ורווחים. לכן, הדרך הנוחה יותר היא להריץ את הטקסט באופן סדרתי – לחפש תתי מחרוזות שמתאימים.

נוכל להתבונן בעוד דוגמא – חיפוש מחרוזות שבهم סוף הביטוי מכיל את `rabbit`. למעשה, הרגיקס המתאים הוא:

`[A-Za-z]*[Rr][Aa][Bb][Bb][Ii][Tt]`

הרכבות של ביטויים

ישנן שתי דרכים להרכיב שני ביטויים אחד על גבי השני. האחד – באמצעות הסימן "||" – קוו אנכי המפריד בין שני ביטויים – למשל `abcxyz` – לחפש אחד משני הביטויים.

דרך אחרת, היא לשים את שני הביטויים אחד ליד השני. למשל `[0-9][A-Za-z]+`.

ניתן להשתמש בשתי דרכים אלו בצורה רקורסיבית. האופרטור השימושי שמאפשר זאת הוא אופרטור `ה"סוגרים"`. למשל `c(b)` ייתן את ביטוי `a` או `b` ואחריו את ביטוי `c`.

נראה בעת מעט קיצורי דרך לביטויים רגולריים:

.	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\v\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

למעשה, מדובר בביטויים שמשלימים אחד את השני.

אם נרצה דוגמא, למשל לחפש את:

"1/one rabbit" or "X rabbits"

כאשר `X` הוא מספר כלשהו.

נוכל להשתמש ברגיקס הבא:

(1|one) rabbit|\d+ rabbits

קודם לכן ראייתי את הקיצורי דרך לביטויים כלליים, ועשינו נדבר על מהهو משמעותי יותר.

מסמני גבולות

נניח ונרצה למצוא מהهو שנמצא בתחילת מילה, נוכל לעשות זאת? או כיצד נוכל למצוא דברים בסוף המחרוזת? הטבלה הזאת מצינה זאת.

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

ו'אפשר לנו להפריד בין ביטויים שהם לא אוטיות לביטויים שהם אוטיות. למשל, בדוגמה הבאה:

Example:

REGEX: "**\brabbit\b**"

INPUT: "Some rabbits plays in the yard."

Output: No match found.

בניגוד לדברים שראינו עד עכשו, כתעת אנחנו מחפשים מיקום. מה היתרונו על פני שימוש ברוחח? דבר זה יכול להוועיל במקרה בו אנחנו נמצאים **בסוף משפט** (רווח לא יယוב), או במקרה בו יהיה רווח **בדוחץ** בין שני ביטויים רצויים.

כמתים

נניח ונמצא למצוא במקרה פעמים בדיקוק נמצא נמצאת ביטוי מסוים, כיצד נוכל לעשות זאת? כיצד נוכל למצוא בטווה מסוימים, או לומר שהביטוי אופציונלי?

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)
X*	X occurs zero or more times
X+	X occurs one or more times

הכמתים הללו מופיעים אחרי הביטוי.

אם נרצה להתבונן בביטוי rabbit שראינו, שככל אותן תופיע שני פעמים, נוכל לעשות זאת כך:

r{1,2}a{1,2}b{2,4}l{1,2}t{1,2}s?

אחרי שדיברנו על כמתים, נרצה לבדוק כיצד הם מתנהגים. ישנן שלוש גישות"

1. כמת חמדני – שמחפש כמה שיותר.
2. כמת מינימליסטי – על מנת לעשות זאת, علينا להוסיף ? לכל אחד מהביטויים.
3. צורה רכושנית – דומה לחמדנית, אבל שונה מעט. היא מתבצעת באמצעות הוספה +.

הצורה הרגילה היא חמדנית. למשל:

- Input is: xgaxxxga

– REGEX is: .*ga

• Searching for .*

• Searching for ga

• Going back one letter

• Searching for ga

• Going back another letter

•

• Found ga!



בצורה שונה, ה-reluctant ימצא כך:

- Input is: xgaxxxga

– REGEX is: .?ga

• Searching for .

• Searching for ga

• Grabbing another character

• Searching for ga

• Searching for . again

• Searching for ga again

• Grabbing another character

•

• Found ga!



רוחחים

חשוב לשים לב כי רוחחים ברג'קס הם ביטוי לכל דבר. וכך נדרש להתייחס אליהם ככ אלו.

כתיבת ביטויים רגולריים

ישנם המונ שקיולו של כתיבת ביטויים רגולריים (קיצור באמצעות טוווח וכו'). נוכל למצוא מספר כללי אכבע לרמת הקריאות והיעילות:

- יש לכתוב ביטויים פשוטים יותר. למשל, עדיף לכתוב {1,2}[ab] מאשר albaaabbaabb.
- עדיף לכתוב ביטויים ייעילים יותר. ביטויים יכולים להופיע המונ פעמים בתוכנה, וכך דבר זה חשוב. חשוב לשים לב לכשلونות.
- עדיף להשתמש בביטויים מדויקים יותר. למשל +[6-0] עדיף מאשר +..
- עדיף להשתמש בביטויים התלויים במיקום.
- אלטרנטיבות של | מאייטים את הביטוי. וכך עדיף להשתמש בחישיבות לסדר.
- פיצול חלקיים משותפים יכול להאיץ את התוכנה. למשל עדיף לכתוב "(cdleff)abc" מאשר {abcdabef}.
- שימוש בסוגרים של Capturing-Groups מאוד מאט את התוכנה.
- שימוש בביטויים לא עיל מבחינות ביצועים, וכך אם אפשר, עדיף להשתמש בכמות הרקורסיבי.

עקרונית, ביטויים רגולריים הם לא מאוד טבעיות. מדובר בסוג של שפת תכנות חדשה.

ביטויים רגולריים ב-Java

ישנה חבילה שנקראת java.util.regex שעליינו ליבא אותה. לאחר מכן, עליינו ליצור אובייקט Pattern שיחזיק את הביטוי הרגולרי. כך למשל – ("+[a-z]+").Pattern pat = Pattern.compile("Now is the time").matcher(pat).find()

icut, אפשר להשתמש בביטוי הזה כדי למצוא אובייקט ממחלקת אחרת. כך למשל – ("Now is the time").Matcher matcher = patt.matcher("Now is the time").find()

לשטי ממולאות אלו אין בניין public ונתן שנייהם יוצרים בתור מתודת סטטית. ברגע שיצרונו Pattern אחד, אפשר ליצור הרבה Matcherים ולכן זה דבר זה יעיל יותר.

באמצעות מתודת matches, שייכת ל-Matcher, ניתן לבדוק האם הביטוי מתאים לכל המחרוזות.

באמצעות מתודת find ניתן לבדוק האם ביטוי מסוים מתאים לחלק מהמחרוזות.

אחרי חיפוש מוצלח ניתן למצוא התקבל הערך :

- מתודת start() מחזירה לנו את האינדקס של הערך הראשון במחuzeות שנמצאה.
- מתודת end() מחזירה לנו את האינדקס של הערך האחרון במחuzeות שנמצאה.

הם מתאימים למתודה substring של גיאוה. אם ננסה לקרוא לביטויים אלו מבלי שהפעלו את הרג'קס, נקבל IllegalStateException, וכיוון שמדובר ב-try-catch, אין צורך ב-try-catch.

נראהicut דוגמה מלא לשימוש ברג'קס :

```

import java.util.regex.*;

public class RegexTest {
    public static void main(String args[]) {
        String patternString = "[a-zA-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()) + " ");
    }
}

```

Output: now_is_the_time

נראה כתה שלוש מетодות רלוונטיות לרג'יקס, הקיימות ב-`String` – מетодת `matches` שבודקת האם הביטויים מתאימים, מетодת `split` שמקצת את הביטוי לפי הרג'יקס, ומетодת `replaceAll` שמחליפה בהתאם לרג'יקס :

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

מבחינת יעילות של גיאוה – חשוב לשים לב כי `Pattern.compile` היא פחרות יעילה. בעקבות כך גם `String.matches`() פחרות יעיל, כי מדובר בקמפול מחודש בכל פעם.

אם נרצה לחזור לשאלת שראינו בתחילת השיעור, כיצד למצוא ביטויים שמתאים לפורמט של תאריך, נקבל :

```

/**
 * A method that returns true iff the given string is a legal
 * date of the format dd/MM/yyyy
 */
boolean isDate(String s) {
    return s.matches("\\d{2}/\\d{2}/\\d{4}");
}

```

סיריאלייזציה - Serialization

על מנת שנוכל להבין מהו `Serialization`, נתחיל במודיבציה. נניח ונרצה לחת אובייקט ולשכפל אותו למקום – בדומה למקומות אחרים במחשב, אפשר לעשות זאת גם בגיאוה. יש לכך שימושים, על מנת ליצור גיבוי או טמפליטים וכו'.

סבירת העבודה שבה עוסדק הינה `Stream` – נסתכל על אובייקט מסווג של מידע.

על מנת להבין זאת, נתבונן בדוגמא פשוטה, של לוקוט, למשל בchnerות :

```

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

```

מה זה אומר "לשמר את האובייקט"? עליינו לחת את כל אחד מהמרכיבים. כאמור, כמשמעות ב-`String` זה יותר פשוט, אבל אם נדרש לעبور על רשיימה, נדרש לבצע תחילה וקורסיבי. בנוסף, ישנו סיבוכים

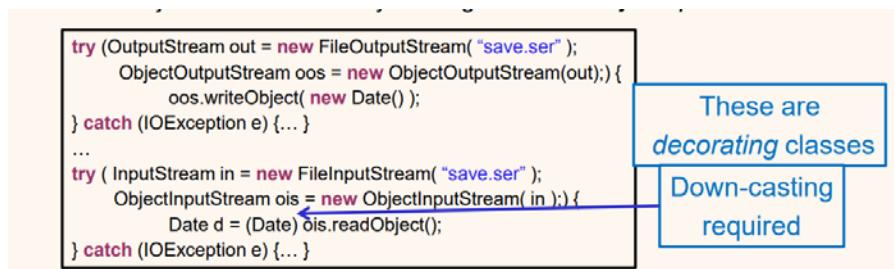
אפשריים, כמו למשל אובייקטים מסוימים, או שני אובייקטים שחולקים פרטנר מסוים (שני בני זוג שחולקים חשבו בנק).

Serialization – סיריאלייזציה הוא תהליך שבו אנו לוקחים אובייקט והופכים אותו לסוג של מידע. כפי שראינו, התהליך זה אכן מתבצע בצורה רקורסיבי, עד שנגיע למיצוי (משתנים פרימיטיביים או משתנים ללא שדות). התהליך ההפוך נקרא **deserialization** ובו מתבצעת הפיכת מידע לאובייקט.

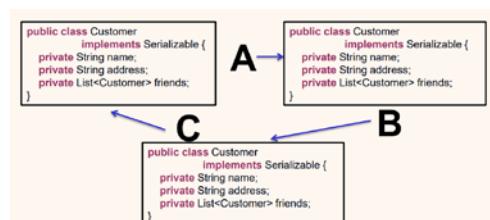
ב-Java, על מנת שנוכל לשמר אובייקט, עליו להיות בעל "יכולות שמירה", או **Serializable**. הרעיון הוא שלפעמים המתכונת לא ירצה שימושו אחר ישמר את האובייקט שלו, למשל. למעשה, כל מחלקה של Java כביררת מבדל לא מאפשרת שימושו אותה, ועליה למשמש **marker interface** שנקרא **Serializable**. השם הכללי של ממשקים כאלה הוא **marker interface** – קלומר מזובר במשמעות שכל מטרתה היא לסמן. כיון שמדובר בתהליכי רקורסיבי, כל אחד מהשדות הללו צריך להיות Serializable או פרימיטיבי.

תהליך הכתיבה נעשה באמצעות **ObjectOutputStream** ותהליך הקריאה נעשה באמצעות **ObjectInputStream**. מדבר במחוקות שמצוות "קיישוט" String – גם מרחיבות את String וגם מקבלות אותו כארוגומנט.

נוכל לראות דוגמה פשוטה לכך כאן:



מה קורה אם בתהליכי הסיריאלייזציה אנחנו נתקלים באובייקט שנתקלנו בו בעבר? נשמר מצביע לאובייקט המקורי ולא נשמר אותו מחדש. נוכל להבין את החשיבות של דבר זה דרך דוגמא. נניח שללקוחות יש לנו מוקודם יש רשימת חברים, בקרה כזו, אם היינו שומרים כל אובייקט מחדש ולא שומרים מצביע לאובייקט המקורי, היינו נכנסים לתוך לולה אינסופית:



אפקט מעניין של שמיירת האובייקט פעמי אחת, בא לידי ביטוי בהרצתה הבאה:

```

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

```

אנחנו למשה מגדירים את ה"מצב" של האובייקט בכל פעם מחדש. מה יקרה עבור ההגדירה של האובייקט בפעם הראשונה? האם ישנה לו ה"מצב"?

עלינו לשים לב כמה נקודות:

- האובייקט שנכתב ראשון נקרא ראשון.
- פועלות השמירה למשה איננה שומרת אותו שוב אלא שומרת מצבו לאובייקט הראשון.

לכן, אם נבצע את הדפסה הבאה:

```
ObjectInputStream in = new ObjectInputStream(...);  
obj = (MyClass)in.readObject();  
System.out.println(obj); // prints the state of the obj  
obj = (MyClass)in.readObject();  
System.out.println(obj);
```

נקבל הדפסה כפולה של 100, כיון שהאובייקט של ה-200 לא מוכר.

על מנת לעקור את התהליך הזה, אפשר לסגור את ה-stream, או לקרוא למトודה בשם `out.reset()` שישוכחתי את מה ששנשמר עד כה. המחיר של האתחול הינו בזבוז משאבים.

שדות transient

המילה `transient` זו מילה שמורה של java, שמשמעותה הינה להוראות כי בתהליכי הסיריאלייזציה, לא נשמר משתנים אלו. בתהליכי הסיריאלייזציה, מוגלים על שדות אלו, ובתהליכי הדחסיריאלייזציה הן מקבלות את הערך הדיפולטיבי. יש שימוש לכך למשל בעת שמירת `String` עצמו. יש לשאל את עצמנו איזה שדות חשובות לשימירה. בנוסף, אין משמעות לשימירת שדה סטטי בתהליכי הסיריאלייזציה, כי הרוי השדה לא שייך לאובייקט אלא מחלקה.

אובייקטים פרימיטיביים אינם יכולים לעبور תהליכי סיריאלייזציה, אבל אנחנו בכל זאת רוצח לבצע להם תהליכי דומה, אפשר לעשות זאת דרך הממשק `DataOutput` שמאפשר מתודות כמו `writeInt` וכו'. יש לשים לב שאין הכוונה למשתנים פרימיטיביים – אלו יכולים לעبور תהליכי סיריאלייזציה ללא בעיה.

במידה ושמרנו אובייקט מסוים, והאובייקט המקורי השתנה, האם אנחנו יכולים לעורוך ולשנות את האובייקט השמור בהתקאה? ישנו שינויים בלתי אפשריים – סוג של שדה, או שינוי היררכיה. אמנם, חלק מהשינויים אפשריים – למשל מחיקת שדה מסוים שפיסטו תגרור התעלומות באובייקט השמור.

על מנת להתמודד עם הבעיה שחלק מהשינויים אפשריים וחלק לא הוא הגדרת `SerialVersionUID` שהוא משנה סטטי. הרעיון הוא כי אנו, בתור כותבי המחלקה, יודעים איזה שינויים אפשר 'לסבול' ואילו לא. בכל פעם ששמרנו אובייקט של גיאוה, נגידר את השדה הזה. במידה ונבצע שינויים שבلتאי אפשר לעדכן, נעדכן את הגרסה. במקרה זה, במידה ונרצה לשנות את האובייקט השמור, התהליך ייכשל. רק במקרה בו הגרסאות תואמות, השינויים יתבצעו.

נשים לב שהשדה הזה אינו הכרחי, ועקרונית java מגדירה זאת באופן עצמאי. אמנם, java איננה יודעת להבדיל בין שינויים 'נסבלים' ובין שינויים 'בלתי נסבלים', ולכן בכל שינוי היא מעדכנת את הגרסה. לכן, הדבר המומלץ הוא להגדיר את השדה בעצמו, ולהחליט באופן מודע אילו שינויים דורשים עדכון גרסה ואילו לא.

באופן כללי, תמיד כדאי להגיד שדה זה כמשמעותם בסיריאלייזציה.

– שכפול – Cloning

נניח ונרצה לייצר שכפול של האובייקט, כיצד נוכל לבצע זאת? נשים לב כי ישנו שני סוגי של העתקה :

- העתקה שטוחה (Shallow Copy) – כל השינויים באובייקט המקורי מתרחשים גם באובייקט החדש (שני האובייקטים מצביעים לאותו המיקום).
- העתקה عمוקה (Deep Copy) – שני האובייקטים מצביעים למקומות שונים, ולכן השינויים אינם משפיעים אחד על השני.

בדומה לシリאליזציה, גם אובייקט דורש אפשרות של שכפול – Cloneable. על מנת שיישכפו את האובייקט, יש צורך להשתמש במתודה clone(). כבירותת מחדל clone() יוצר שכפול שטוח, והאחריות שלנו לדروس את clone() כראות עינינו.

על המחלקה למשם את המשק Cloneable, ואם היא לא משתמשת ומשתמשים במתודה clone() יזרק שנקרא CloneNotSupportedException. היא משתמשת שכפול שטוח ולכן גם מערכים, שימושים את המתודה clone() מבצעים העתקה שטוחה. נראהicut דוגמה לקוד שמבצע העתקה :

<pre>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; } }</pre>	<pre>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 }</pre>
---	--

אמנם, הדרך המומלצת לבצע שכפול אינה clone() אלא copyConstructor(). נראה זאת באמצעות דוגמה :

<pre>class Pet implements Cloneable { 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(); } }</pre>	<pre>Pet myPet = new Pet(); myPet.setType("Dog"); Pet myPet1 = new Pet(myPet); Pet myPet2 = new Pet(myPet); myPet1.setName("Woofi"); myPet2.setName("Goofi");</pre>
---	---

בצורה זו, אנחנו מביצים את הבנייה המקורי, ואת האובייקטים שאנו רוצים לבצע שכפול عمוק, אנחנו משכפלים בצורה עצמאית.

Java Reflections

למעשה, אנחנו מדברים כאן על מגנון שמאפשר לתת לתוכנה 'מודעות מסוימת' – לשנות חלקים בתוכה. מדובר באחד הכלים החזקים ביותר שנלמדים בקורס, וממילא גם אחד הכלים המסוכנים.

עד כה, דיברנו על שני סוגי אובייקטים :

- .Reference Type -
- .Primitive Type -

לכל אובייקט מסווג מואתחל סוג של מחלקה שנקרא `.java.In.calss`. כיצד ניתן לגשת למחלקה זו? באמצעות השורה הבאה, למשל : `Class cls = Class.forName("MyClass")`. דרך נוספת היא למצוא זאת באמצעות() כאשר `Class cls = myObj.getClass()` הוא מעשה מופיע של האובייקט. אם מדובר במחלקה שלא נמצא, תזרוק השגיאה `.ClassNotFoundException`.

באמצעות האובייקט `class` ניתן לבצע מספר פעולה :

- לקבל את רשימת הבנים שלו, באמצעות השורה הבאה :
 - `Constructor[] ctorlist = cls.getDeclaredConstructors()`
 - ליצור אובייקט חדש באמצעות הבניי :
 - `.Object retobj = ctorlist[i].newInstance(arglist)`
 - לקבל רשימה של כל המethodות במחלקה, באמצעות :
 - `.Method[] methlist = cls.getDeclaredMethods()`
- אנחנו אפילו יכולים לקבל את כל **הmethodות הפרטיות** (איך זה מסתדר עם כל מה שראינו בקורס)?
- להריץ את המethodות, באמצעות :
 - `.methlist[j].invoke(obj, arglist)`
 - לקבל שדות, לשנות אותו ועוד (ואפילו פרטיות). למשל :
 - `.Field[] fieldlist = cls.getDeclaredFields()`

נראה דוגמה כללית לשימוש ב-`Reflections` :

```
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));
    }
}
```

מדוע ששימוש ב-`Reflections`? מדוע בכך שמאפשר להרחיב את הגישות. למשל, אנחנו יכולים לקבל מחלקה כללית וליצור על פיה המונם דברים (אין צורך ב-SingelChoice). לעומת זאת זה מאפשר יצירת קוד כללי לחלווטין. בנוסף, `Reflections` מאפשר לנו כלי חזק יותר לדיבוג. למעשה, תהליך הסיריאלייזציה מתבצע באמצעות `Reflections` (כי יש צורך לגישה למethodות וכו').

מצד שני, יש לדבר זה הרבה חסרונות, שהרי הוא פוגע משמעותית באפקטולוציה ובהחבות מידע. בנוסף, דבר זה יכול ליצור כל מיני תלויות שהמתכונת לא תכנן מראש. מעבר לכך, תוכניות שימושísticas בכך הופכות לאיטיות יותר.

נשים לב כי דבר זה מתחבר למה שראינו בתחילת הקורס כי `private` איננו מוצפן. המטרה העיקרית של `private` הינה ליצור עיצוב טוב יותר.