תכנות מונחה עצמים

בשפת Python-ו Java

מחברת סיכומים לקורס תכנות מונחה עצמים 2021

Java

Python

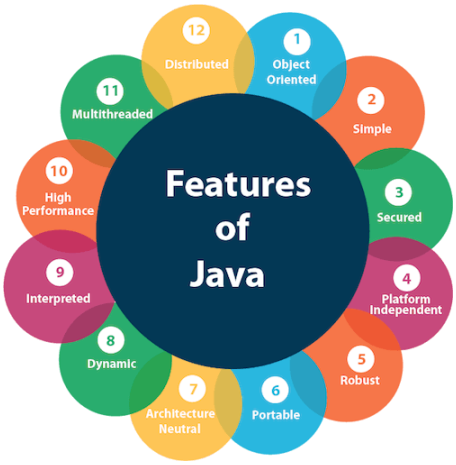
Git & GitHub

מה זה Java

ג'אווה היא שפת תכנות מונחית עצמים.
לרוב עוברות תוכניות ג'אווה הידור (**קימפול**) ל־Java bytecode, שפת ביניים דמוית שפת מכונה, שאותה מריצה מכונה וירטואלית (Java Virtual Machine; **JVM**).

ג'אווה היא שפה בעלת טיפוסיות **סטטית חזקה**, כלומר לכל ביטוי בשפה מתאים טיפוס יחיד, תקינות ביטויים נבדקת בזמן ההידור (קימפול), במידת האפשר. כאשר אין זה אפשרי, תתבצע בדיקה בזמן ריצה.
לכל טיפוס פרימיטיבי בג'אווה יש מחלקה עוטפת, אליה וממנה מתבצעת המרה אוטומטית.
למשל, לטיפוס הפרימיטיבי int קיימת מחלקה עוטפת Integer.

שפת ג'אווה כוללת גם **ניהול זיכרון אוטומטי**.
המתכנת פטור מן ההכרח לשחרר זיכרון המוקצה לאובייקט ברגע שאין עוד משתנים המצביעים עליו.
במקום זאת, סביבת זמן הריצה כוללת מנגנון "איסוף זבל" (**garbage collector**), המבצע זאת אוטומטית.
ג'אווה מאפשרת **ירושה יחידה בלבד**, וזאת על מנת למנוע בעיות דו־משמעות הנוצרות מירושת יהלום.
כדי לאפשר גמישות דומה לזו של ירושה מרובה, קיים בג'אווה מנגנון **ממשק** (**interface**) המגדיר רשימה של מתודות, המהווה חוזה עם המתכנת.
כך כל מחלקה יכולה **להרחיב** (**extends**) לכל היותר מחלקה אחת, **ולממש** (**implements**) מספר בלתי מוגבל של ממשקים.
על המחלקה לממש במפורש כל מתודה בכל ממשק כזה, וכך לא קיימת בעיה של כפל משמעות.
ג'אווה תומכת **בתכנות גנרי** (**Generics**), מספר משתנה של פרמטרים לפונקציה ו**טיפוסי מניה** (**Enums**).



רשימת מושגים ב-Java

1. **תן הגדרה לאובייקט - (עצם / Object)**
אובייקט הוא משתנה מטיפוס המחלקה, אנו בונים אובייקט, ע"י אתחול משתני העצם של המחלקה.
אובייקט הוא יחידת תוכנה, שמספקת שירותים (methods) מסוימים.

2. **תן הגדרה למחלקה (Class)**
קבוצה של עצמים מאותו סוג, כלומר שמספקים את אותם שירותים באותה הצורה.
מבנה לוגי המאגד בתוכו פונקציות ומשתני עצם תחת שם אחד, ממחלקה ניתן ליצור אובייקטים,
שם המחלקה הוא שם של טיפוס חדש, המוגדר ע"י משתמש.

3. **אובייקטים מול מחלקות**
האובייקטים הם מופעים (instances) של המחלקה.
המחלקה היא הישות הסטטית בקוד המקור והאובייקט הוא הישות הדינמית בזמן הריצה.

4. **מהו ההבדל בין שיטה לפונקציה**
פונקציה - זה אומר פונ' סטטית שהיא לא יכולה להשתמש במשתני עצם של מחלקה, אין צורך באובייקט כדי להפעיל פונ' סטטית, מספיק לכתוב את שם המחלקה, נקודה ושם הפונקציה.
שיטה - מופעלת על אובייקט ולשיטה יש גישה לכל משתני עצם של המחלקה שאליה היא שייכת.

5. **מהו ההבדל בין עצם המחלקה לבין משתנה סטטי**
   a. משתנה עצם נוצר בכל פעם שיוצרים אובייקט ומשתנה סטטי נוצר פעם אחת לפני הפעלת תוכנה ומשותף לכל אובייקטים מטיפוס המחלקה.
   b. ניתן לגשת למשתנה סטטי גם מבלי ליצור אובייקט אלא ע"י שם המחלקה ונקודה.
   משתנה עצם (תלוי בהרשאות) ניתן לגשת אליו רק דרך אובייקט.

6. **מהו ההבדל בין private לבין public**
private- רק שיטות המחלקה שאנו נמצאים בה יכולה לגשת למשתני העצם/שיטות(בשימוש פשוט ללא שיטות כמו reflection).
public - כל המחלקות יכולות לגשת למשתני העצם/שיטות.

7. **תן הגדרה ל-protected**
ניתן לגשת למשתני protected רק ממחלקות שיורשות ממנה, מהמחלקה עצמה, ולמחלקות הנמצאות באותו חבילה (package).

8. מה התפקיד של constructor

ליצור אובייקט חדש ולאתחל משתנה עצם של המחלקה.

9. מהו constructor default

בנאי ללא ארגומנטים "שכאילו" נכתב אוטומטית ע"י הקומפיילר רק אם לא נכתב אף בנאי אחר למחלקה. הבנאי הדיפולטיבי מאתחל את משתני העצם של המחלקה לערכים דיפולטיביים (0 למספרים, null לרפרנס לאובייקטים וכו').

10. הסבר מושג של הורשה

דרך להגדיר מחלקה על ידי מחלקות כלליות יותר. המחלקה היורשת מקבלת את כל התכונות והשיטות של מחלקת האב המוגדרות כ-public/protected.

11. הסבר מושג של overloading

שתי פונקציות עם אותו שם אבל עם ארגומנטים שונים/כמות ארגומנטים שונה.

12. הסבר מושג של overriding

בהורשה אם אנו נכתוב פונ' בעלת אותו שם כמו אצל מחלקת האב, פונ' זו תדרוס את התוכן של הפונ' במחלקת האב.

13. תן הגדרה למשתנה final

משתנה שאי אפשר לשנות את הערך שלו בזמן ריצת התוכנה.

14. תן הגדרה לשיטה final

פונ' שלא ניתנת לדריסה ע"י יורשים

15. הגדרה ל-polymorphism

התייחסות לאובייקטים מטיפוסים שונים בצורה אחידה.

16. מהו ההבדל בין interface לבין class abstract

ב-Interface אנו רק מצהירים על השיטות ללא מימוש שלהם וללא שדות
לעומת class abstract שיכולה להכיל שדות ומימוש של שיטות.
בנוסף מחלקה יכולה לממש כמה Interfaces אבל לרשת רק ממחלקה אחת.

17. מהו ההבדל בין class לבין class abstract

מ-class abstract אין אפשרות ליצור אובייקטים מסוג הclass abstract לעומת class שאנו יוצרים אובייקטים מסוג הclass.

18. הסבר מושג של exception

מנגנון שמאפשר להודיע על שגיאה מבלי לעצור את התהליך.

19. מהי משמעות של מילה super

רפרנס לחלק האובייקט ממחלקת האב שעליו מופעלת השיטה.

20. מהי משמעות של מילה this

רפרנס לאובייקט שעליו מופעלת השיטה.

21. תן הגדרה ל-THREAD

בתוך תהליך אחד ניתן להפעיל כמה תת תהליכים שעובדים במקביל או ב"כאילו" מקביל.

22. תן הסבר קצר למצבים הבאים של TREAD-ים

a. **new** - קיים אובייקט של ה-thread , אך הוא לא רץ.
b. **runnable** . מצב בו ה-thread רץ.
c. **blocked** - ה-thread מחכה, נעצר ע"י thread אחר.
d. **waiting** - מחכה עד ההודעה שהוא יכול להמשיך(notify).
e. **waiting timed** - כמו מקודם אבל מחכה זמן מסוים.
f. **terminated** - ה-thread סיים את חייו, סיים לעבוד.

23. מהו מצב של DEADLOCK

מצב בו שני thread-ים חוסמים אחד את השני לדוג' כאשר אחד תפס אובייקט א' ומנסה לתפוס אובייקט ב', והשני תפס' אובייקט ב' ומנסה לתפוס את א'. אנו תקועים!

24. Serialization

תהליך המרת אובייקט לזרם של Bytes.

25. Synchronized

המילה השמורה synchronized שמופיעה בראש בלוק פקודות מבטיחה שבכל עת רק thread יחיד יוכל להפעיל את הבלוק על אובייקט ספציפי, במילים פשוטות, כל עוד הבלוק לא מסתיים כל ההפעלות האחרות של אותו בלוק ע"י thread-ים אחרים ימתינו לסיום עבודתו של הבלוק על אובייקט ספציפי.

אובייקטים ומחלקות

אובייקטים (Objects) הם ישויות בעלות:

1. **תכונות** - המאוחסנות במשתנים, כלומר האובייקטים יודעים להחזיק מידע (***data members***).
2. **התנהגות** - המתממשות במתודות/שיטות (***methods***).

משתני מופע (instance variables): אוסף המשתנים המתארים את האובייקט.
מתודות מופע (instance methods): אוסף המתודות המיועדות לפעול על משתני המופע באובייקט.

הקוד של אובייקט נכתב ומתוחזק באופן עצמאי ובלתי תלוי באובייקטים אחרים, ולכן גם אפשר להעבירו בקלות ממקום למקום במערכת. נוסף על כך, האובייקט יכול להכיל מידע פרטי ומתודות פרטיות, היכולות להשתנות מבלי להשפיע על אובייקטים אחרים התלויים בו.

מחלקה (Class) : כל טיפוס של אובייקט נקרא מחלקה. השימוש במחלקות מאפשר ליצור אובייקטים רבים מאותו סוג מבלי שיהיה צורך לכתוב מחדש את הקוד עבור כל אובייקט.

```
Song imagine = new Song();
```

כאשר Song הוא המחלקה (הטיפוס) ו-imagine הוא האובייקט.
מחלקה היא תבנית, המגדירה את אוסף המשתנים והמתודות המשותפים לכל האובייקטים מאותו הסוג. המחלקה אינה מגדירה אובייקטים, היא רק מתארת את המבנה וההתנהגות שלהם. ממחלקה ניתן ליצור אובייקטים, כלומר כדי ליצור אובייקט יש ליצור מופע (***instance***) של המחלקה.
בעת יצירת המופע מוקצה זיכרון עבור משתני האובייקט.

משתני מחלקה (class variables): משתנים המשותפים למחלקה כולה.
מתודות מחלקה (class methods): מתודות המשותפות למחלקה כולה.

משתני המחלקה אינם נוצרים מחדש עבור כל אובייקט ואובייקט, אלא יש מהם עותק אחד עבור כל האובייקטים של אותה המחלקה.

מחלקה מכילה **בנאים (Constructors)** - כלומר, כלי ליצירת אובייקט.

דוגמה -

אובייקט מטיפוס "**מוצר**", שיכיל נתונים לגבי המוצר: שם, מחיר, תיאור, כמות, ותאריך תפוגה. אובייקט מטיפוס מוצר יתמוך בכמה פעולות: יצירה של מוצר, עדכון של מחיר המוצר, עדכון כמות המוצר, והדפסה של פרטי המוצר.

אובייקט מטיפוס "**מלאי**", שיכיל אוסף של מוצרים. אובייקט מטיפוס מלאי יתמוך בפעולות משלו: הכנסה של סוג מוצר למלאי, הוספת והחסרת מוצרים מהמלאי, חישוב של כלל ערך המוצרים והדפסת רשימת מלאי.

קל לראות שקיימים קשרים בין האובייקטים: בעת הדפסת רשימת מלאי, למשל, פונה אובייקט המלאי לפונקציית ההדפסה של כל מוצר במלאי. בעת חישוב סך ערך המלאי ישתמש אובייקט המלאי, כחלק מהחישוב, בנתוני המחיר והכמות שמכיל כל מוצר.

הצהרה של מחלקה

```
[modifiers] class ClassName [extends name] [implements name] { ..
```

`modifiers` האפשריים הם:

1. public - המחלקה גלויה לכל האובייקטים, גם לכאלה שהמחלקות שלהם מוגדרות ב-package אחר.
2. final - מגדיר את המחלקה כסופית, שהיא מחלקה שאי אפשר לרשת ממנה.
3. abstract - מגדיר את המחלקה כמחלקה מופשטת, כלומר מחלקה שלא כל השיטות שלה ממומשות, ולכן אי אפשר ליצור ממנה אובייקטים באופן ישיר והיא משמשת כבסיס ליצירת תת-מחלקות.

`extends` המציין את שם המחלקה שממנה יורשים.

`implements` המציין את רשימת הממשקים שהמחלקה מממשת.

הצהרה של משתנים

```
[accessSpecifier] [static] [final] [transient] [volatile] type variableName;
```

`accessSpecifier` הם מאפייני גישה: public, package, protected, private. הם מגדירים מי רשאי לגשת למשתנה.

static - מציין כי הוא משתנה מחלקה (class variable) בניגוד למשתנה (instance variable). משתני מחלקה נוצרים פעם אחת עבור המחלקה, כאשר המערכת טוענת לראשונה את המחלקה.

final - מאפשר להגדיר קבועים, שהם משתנים שאי אפשר לשנות אותם.

volatile - מאפיין שניתן למשתנים המשמשים כמה תהליכים במקביל. הוא מציין לקומפיילר שאין לבצע עליו אופטימיזציה, כלומר כל גישה למשתנה כזה חייבת להיעשות ישירות מהזיכרון, ולא מעותק מקומי על המחסנית.

type - כמו המוכרים: int, double, char ...

ירושה

זהו המנגנון ב- Java באמצעותו מחלקה אחת מאפשרת לרשת את התכונות (שדות ושיטות) של מחלקה אחרת.

- **מחלקת על (Superclass)** - המחלקה שתכונותיה עוברות בתורשה.
- **תת-מחלקה (Subclass)** - המחלקה היורשת את מחלקת העל. היא יכולה להוסיף שדות ושיטות משלה בנוסף לשדות של שיטות-העל.
- **שימוש חוזר (Reusability)** - השימוש בירושה מאפשר ליצור מחלקות חדשות תוך כדי שימוש חוזר. כאשר אנו רוצים ליצור מחלקה חדשה וכבר יש מחלקה הכוללת חלק מהקוד הרצוי לנו, אנו יכולים לייצר את המחלקה החדשה שלנו מהמחלקה הקיימת. בכך אנו משתמשים שוב בשדות ובשיטות של המחלקה הקיימת.

המשתנה השמור עבור שימוש בירושה הוא `extends` .

דוגמה עבור **ירושה יחידה**:

```
class one {
    public void print_geek() {
        System.out.println("Geeks");
    }
}
class two extends one {
    public void print_for() {
        System.out.println("for");
    }
}
```

דוגמה עבור **ירושה עם היררכיה**:

```
class one {
    public void print_geek() {
        System.out.println("Geeks");
    }
}
class two extends one {
    public void print_for() {
        System.out.println("for");
    }
}
class three extends two {
    public void print_geek() {
        System.out.println("Geeks");
    }
}
```

במקרים רבים נרצה לבצע בתת- מחלקה את הדברים הבאים:

1. להוסיף משתנים לצורך ייצוג תכונות נוספות שלא היו קיימות במחלקת-העל.
2. להגדיר בתת-מחלקה בנאים משלה. בנאים אלה מפעילים תחילה את הבנאים של מחלקת-העל באמצעות קריאה ל- ***super()*** המאתחלים את המשתנים המוגדרים במחלקת העל.
3. להגדיר מחדש (***override***) מתודות שהתת מחלקה ירשה ממחלקת העל שלה, כל שיתאימו לצרכים שלה. אפשר לראות כיצד מתודה זו מבצעת קריאה באמצעות הקידומת ***super.***

שימוש ב-***super*** עבור **משתנים**: תרחיש זה מתרחש כאשר המחלקה היורשת ומחלקת העל כוללים את אותם שמות משתנים. עבור הדוגמה הבאה נקבל את ההדפסה `"Maximum Speed: 120 "` :

```
class Vehicle {
       int maxSpeed = 120;
}
class Car extends Vehicle {
       int maxSpeed = 180;
       void display() {
              System.out.println("Maximum Speed: " + super.maxSpeed);
       }
}
class Test {
       public static void main(String[] args) {
              Car small = new Car();
              small.display();
       }
}
```

שימוש ב-***super*** עבור **מתודות**: זה משמש כאשר אנו רוצים לקרוא לשיטה ממחלקה ממנה ירשנו.
כך שבכל פעם כשיש למחלקה הנורשת ולמחלקה הירושת את אותו שם המתודה, אנו משתמשים במילת מפתח סופר כדי להבדיל בין המתודות. לדוגמה:

```
class Person{

    void message() {
        System.out.println("This is person class");
    }
}

class Student extends Person {

    void message() {
        System.out.println("This is student class");
    }


    void display() {
```

```
        message();
        super.message();
    }
}

class Test {
    public static void main(String args[]) {
        Student s = new Student();
        s.display();
    }
}
```

נקבל את ההדפסה:

```
This is student class
This is person class
```

שימוש ב-***super*** עבור **בנאים**: ניתן להשתמש במילת מפתח סופר כדי לגשת לבנאי של המחלקה ממנה ירשנו.
דבר חשוב נוסף הוא ש-'super' יכול לקרוא גם לבנייה פרמטרית וגם לבנייה לא פרמטרית, בהתאם למצב.

```
class Person {
       Person() {
              System.out.println("Person class Constructor");
       }
}

class Student extends Person {
       Student() {
              super();
              System.out.println("Student class Constructor");
       }
}

class Test {
       public static void main(String[] args) {
              Student s = new Student();
       }
}
```

נקבל את ההדפסה:

```
Person class Constructor
Student class Constructor
```

הערה חשובה - קריאה ל- ()super **חייבת** להיות ההצהרה הראשונה ביותר בבנאי המחלקה היורשת.

- ניתן לגשת למשתני **protected** רק ממחלקות שיורשות ממנה, מהמחלקה עצמה, ולמחלקות הנמצאות באותו חבילה (package).

פולימורפיזם

פירוש המילה פולימורפיזם הוא בעל צורות רבות.
פולימורפיזם מאפשר לנו לבצע פעולה אחת בדרכים שונות.

פולימורפיזם במציאות

ל-"בן-אדם" יכול להיות מאפיינים שונים בו זמנית. כמו ש-"בן-אדם" בו זמנית הוא אבא, בעל וגם עובד.
אז אותו "בן-אדם" בעל מאפיינים שונים במצבים שונים. - זה נקרא פולימורפיזם.

פולימורפיזם ב-Java

ב-Java יש 2 סוגים של פולימורפיזם:

1. סוג Overloading.
2. סוג Overriding.

העמסת מתודות (Overloading)

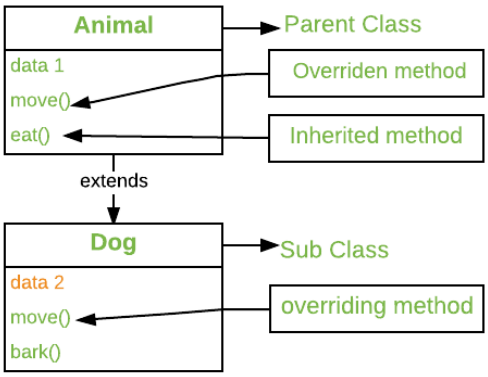
העמסה מאפשרת להגדיר באותה המחלקה כמה מתודות באותו השם בתנאי שחותמת המתודה שלהם שונה.
לדוגמה:

```
public class Sum {

    // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y) {
        return (x + y);
    }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z) {
        return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double parameters
    public double sum(double x, double y) {
        return (x + y);
    }

}
```

יתרון : אנחנו לא צריכים ליצור ולזכור שמות שונים למתודות שעושות את אותו הדבר.

הגדרה מחדש של מתודות (Override)

הגדרה מחדש של מתודה בתוך תת-מחלקה מתבצעת באמצעות כתיבת מתודה חדשה בעלת חותמת וטיפוס מוחזר הזהה למתודה המקורית שהתת-מחלקה ירשה ממחלקת- העל.



באיור Dog יורש מ-Animal ואפשר לראות בתת-המחלקה Dog כי הוא מגדיר מחדש (override) את המתודה move המתודה החדשה מחליפה את המתודה המקורית המוגדרת במחלקת- העל Animal .

לדוגמה:

```
// Base Class
class Employee{
    void salary() {
      System.out.println("100$");
    }
}
// Inherited class
class Clerk extends Employee {
    @Override
    void salary() {
      System.out.println("200$");
    }
class Manager extends Employee {
    @Override
    void salary() {
      System.out.println("1000$");
    }
}
```

ישנם שני סוגים של מתודות, שתת המחלקה לא יכולה להגדיר מחדש:

- מתודות המוצהרות עם static.
- מתודות המוצהרות עם final.
- מתודות המוצהרות עם private.

מתודות המוצהרות כ-abstract במחלקת העל הן חסרות מימוש, ולכן תת-מחלקה חייבת לממש אותם, אלא אם כן תת המחלקה היא גם abstract בעצמה.
מתודות override חייבות להיות עם אותו return type.

מחלקות ושיטות אבסטרקטיות

הפשטת נתונים היא המאפיין שמכוחו רק הפרטים החיוניים מוצגים בפני המשתמש.
לדוגמא: מכונית נתפסת כמכונית ולא כמרכיביה האישיים.

הפשטת נתונים עשויה להיות מוגדרת כתהליך של זיהוי המאפיינים הנדרשים בלבד של אובייקט תוך התעלמות מהפרטים הלא רלוונטיים. המאפיינים וההתנהגויות של אובייקט מבדילים אותו מאובייקטים אחרים מסוג דומה ומסייעים גם בסיווג / קיבוץ האובייקטים.

אבסטרקטיות במציאות

איש נוהג במכונית. האיש יודע רק שלחיצה על הגז תגביר את מהירות המכונית או הפעלת בלמים תעצור את המכונית, אך הוא לא יודע כיצד בעת לחיצה על הגז המהירות עולה, הוא אינו יודע על המנגנון הפנימי של המכונית או המימוש של הגז, בלמים וכו 'במכונית - זה הכוונה לאבסטרקטיות.

אבסטרקטיות בשפת ג'אווה

1. מחלקה מופשטת היא מחלקה המוצהרת עם מילת מפתח `abstract`.
2. מתודה מופשטת היא מתודה המוצהרת ללא מימוש.
3. במחלקה מופשטת יכולות להיות או לא יכולות להיות כל המתודות המופשטות.
4. יש להכריז על כל מחלקה המכילה שיטה מופשטת אחת או יותר באמצעות מילת מפתח `abstract`.
5. לא ניתן ליצור באופן מיידי מחלקה מופשטת.
6. במחלקה מופשטת יכולים להיות בנאים פרמטרים.
7. בנאי דיפולטיבי תמיד יהיה קיים במחלקה מופשטת.

מחלקות מופשטות (abstract) הן מחלקות שאי אפשר ליצור מהן אובייקטים באופן ישיר.
מחלקות אלה מגדירות אוסף של תכונות המשותפות לכמה מחלקות, והן משמשות רק כבסיס ליצירת מחלקות קונקרטיות (מחלקות שיש להן מימוש מלא לכל המתודות שלהן).

מתודה מופשטת היא מתודה שהמימוש שלה אינו משותף לכל התת- מחלקות, ולכן המחלקה המופשטת מספקת אותה בלי מימוש (בלי גוף).
מחלקה מופשטת יכולה להכיל משתנים ומתודות שאינן מופשטות.
תת- מחלקה של מחלקה מופשטת חייבת לספק מימוש למתודות המופשטות, אלא אם כן התת- מחלקה עצמה מוגדרת כמחלקה מופשטת.

לדוגמה המחלקה האבסטרקטית Shape עם מתודה אבסטרקטית draw:

```
import java.awt.*;
public abstract class Shape{
   private int x,y;

   public Shape(int x, int y){
      this.x = x;
      this.y = y;
   }

   public abstract void draw(Graphics g);
}
```

והמחלקה Rectangle מרחיבה את המחלקה Shape וממשת את המתודה האבסטרקטית שלה, בנוסף המחלקה Rectangle בונה את עצמה בעזרת ירושת הבנאי הפרמטרי של Shape באמצעות super.

```
import java.awt.*;
public class Rectangle extends Shape{
   private int width,height;

   public Rectangle(int x, int y, int width, int height){
      super(x,y);
      this.width = width;
      this.height = height;
   }

   public void draw(Graphics g){
      g.drawRect(getX(),getY(),width,height);
   }
}
```

ממשק - interface

כמו מחלקה, בממשק יכולות להיות שיטות ומשתנים, אך השיטות המוצהרות בממשק הן כברירת מחדל מופשטות (חתימת שיטה בלבד, ללא גוף).

ממשק (**interface**) הוא אוסף של קבועים ומתודות (בלי מימוש) שאפשר להוסיף למחלקה כלשהי כדי להרחיב את היכולות שלה. מחלקה המצהירה שהיא מממשת (**implements**) את הממשק, חייבת לממש את כל המתודות המוגדרות בממשק. בצורה זו, ההתנהגות המוגדרת על- ידי הממשק, ממומשת על- ידי המחלקה.

ממשק הוא בעצם כלי, המאפשר להוסיף יכולות משותפות למחלקות שונות שאינן נמצאות בהיררכיית ירושה משותפת. הגדרה של ממשק, כמו הגדרה של מחלקה, מורכבת משני חלקים: הצהרה וגוף. ההצהרה כוללת את שם הממשק, והגוף כולל הצהרות של קבועים ומתודות (בלי גופים).

לדוגמה, הממשק שלפניכם מגדיר קבוע וכמה פעולות הקשורות בטיפול באוספים של נתונים, כגון: מחסניות, וקטורים, רשימות מקושרות ועוד:

```
interface Collection {
   int MAX = 500;
   void add(Object obj);
   void delete(Object obj);
   Object find(Object obj);
   int currentCount( );
}
```

הערות חשובות

- אם מחלקה מיישמת ממשק ואינה מספקת את כל השיטות שצוינו בממשק, יש להכריז על המחלקה כמופשטת, אחרת - שגיאה.
- במידה ובממשק קיימת הצרה של מתודה מסוג 'default', לא נהיה חייבים לממש את המתודה הזאת במחלקה המממשת.

ההבדל בין מחלקה אבסטרקטית לממשק

- מחלקה יכולה להרחיב רק מחלקה מופשטת אחת אבל יכולה ליישם ממשקים רבים יחדיו.
- מחלקה אבסטרקטית מאפשרת לך ליצור מתודות שתתי-מחלקות יכולות ליישם או לעקוף ואילו ממשק רק מאפשר לך לציין מתודות אך לא מיישם אותם.

תכנות גנרי

הרעיון הוא לאפשר לסוג (מספר שלם, מחרוזת, וכו'.. וסוגים המוגדרים על ידי המשתמש) להיות פרמטר לשיטות, מחלקות וממשקים. באמצעות Generics ניתן ליצור מחלקות שעובדות עם סוגי נתונים שונים וגם עם מתודות שמחזירות כמה סוגים שונים של נתונים.

- Object הוא מחלקת העל של כל המחלקות האחרות והתייחסות ל-Object יכולה להתייחס לכל סוג אובייקט.

מתודה גנרית

מאפשרת להגדיר טיפוס אחד או יותר כפרמטרים של המתודה. פרמטרים אלה מוגדרים בכותרת המתודה. אזור זה תחום בסוגריים <..>.

```
class Test {
       static <T> void genericDisplay (T element) {
              System.out.println(element.getClass().getName() + " = " + element);
       }
       public static void main(String[] args) {
              genericDisplay(11);
              genericDisplay("GeeksForGeeks");
              genericDisplay(1.0);
       }
}
```

הדפסה:

```
     java.lang.Integer = 11
java.lang.String = GeeksForGeeks
     java.lang.Double = 1.0
```

מחלקה גנרית

מחלקות המגדירות טיפוס או טיפוסים כפרמטרים של המחלקה. הטיפוסים מוגדרים באזור המיועד להגדרת פרמטרים גנריים, המופיע בצורה <..>.

```
class Test<T> {
    T obj;
    Test(T obj) {
      this.obj = obj;
    }

    public T getObject() {
      return this.obj;
    }
}
```

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

        Test <Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        Test <String> sObj = new Test<String>("GeeksForGeeks");
        System.out.println(sObj.getObject());
    }
}
```

הדפסה:

```
15
GeeksForGeeks
```

היתרונות של תכנות גנרי

1. שימוש חוזר בקוד: אנו יכולים לכתוב שיטה / מחלקה / ממשק פעם אחת ולהשתמש בכל סוג שנרצה.
2. סוג בטיחות: גנריות גורמות לשגיאות בהופעת זמן הקומפילציה מאשר בזמן הריצה (תמיד עדיף לדעת בעיות בקוד בזמן קימפול ולא לגרום לקוד להיכשל בזמן הריצה).
3. יישום אלגוריתמים גנריים: על ידי שימוש בגנריות, אנו יכולים ליישם אלגוריתמים שעובדים על סוגים שונים של אובייקטים.

מחלקות מקוננות

מחלקה מקוננת היא מחלקה המוגדרת בתוך מחלקה אחרת.
ישנם שני סוגים של מחלקות מקוננות:

1. מחלקה מקוננת סטטית (static nested class).
2. מחלקה פנימית (inner class).

מחלקה מקוננת סטטית

```
public class Outer{
   ...
   public static class Inner{
      ...
   }
}
```

המחלקה **Inner** יכולה להתייחס אך ורק למשתנים ולמתודות של המחלקה שלה ולא יכולה להתייחס למשתנים ולמתודות של המחלקה **Outer** , אלא אם היא יוצרת מופע של Outer בתוכה.

מחלקה פנימית

```
public class Outer{
   ...
   public class Inner{
      ...
   }
}
```

מחלקה פנימית **Inner** יכולה להתייחס לכל המשתנים והמתודות של המחלקה המכילה אותה **Outer** , כולל למשתנים ולמתודות המוגדרות כפרטיות.

אם נרצה ליצור מופע של המחלקה הפנימית מחוץ למחלקה **Outer** (נניח ממחלקה אחרת) אז נצטרך :

```
Outer out = new Outer()
Outer.Inner in = out.new Inner();
```

השימוש ב- **this** בתוך המחלקה הפנימית מתייחסת לאובייקט של המחלקה הפנימית.
מתוך המחלקה הפנימית אפשר להתייחס ל-**this** של המחלקה המכילה באמצעות שימוש בשם המחלקה כקידומת: ***' Outer.this '*** .

מחלקות אנונימיות ולמבדות

מחלקה אנונימית

מחלקה פנימית ללא שם. מחלקה פנימית אנונימית יכולה להיות שימושית בעת יצירת מופע של אובייקט עם "תוספות" מסוימות, כגון שיטות overloading של מחלקה או ממשק, מבלי שיהיה צורך לסווג מחלקה ממשית.
בדוגמה הבאה אפשר לראות את המחלקה האנונימית בתוך Thread t = new Thread(**HERE**):

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

            Thread t = new Thread( new Runnable() {
                  public void run() {
                        System.out.println("Child Thread");
                  }
            }); // end of declaring Thread t = ..
            t.start();
            System.out.println("Main Thread");
      }
}
```

ההבדלים בין מחלקה אנונימית למחלקה רגילה

1. מחלקה רגילה יכולה ליישם כל מספר ממשקים אך מחלקה פנימית אנונימית יכולה ליישם רק ממשק אחד בכל פעם.
2. מחלקה רגילה יכולה לרשת מחלקה וליישם את כל מספר הממשקים בו זמנית. אך מחלקה פנימית אנונימית יכולה לרשת מחלקה או יכולה ליישם ממשק אך לא את שניהם בכל פעם.
3. במחלקה רגילה אנו יכולים לכתוב כמה בנאים, אך למחלקה פנימית אנונימית אנחנו לא יכולים לכתוב בנאי למחלקה הזאת מכיוון שהמחלקה חסרת שם, והדרישה הבסיסית לבנאי היא לציין את שם המחלקה בהצהרת הבנאי.

למבדה (lambda)

למבדה היא פונקציה ללא שם שניתן ליצור מבלי להשתייך לשום מחלקה, ונכתבת כך:

כאשר באדום, זה הארגומנט (כמו פונקציה רגילה), החץ זה הכניסה אל מימוש הפונקציה.
וכמו בפונקציה רגילה, נשתמש בארגומנטים לביצוע הפונקציה.

המחלקה Object והטיפוס enum

המחלקה Object

היא ה-"אמא" של כל המחלקות. המחלקה הזו מגדירה את ההתנהגות הבסיסית של כל אובייקט בשפה Java וכוללת את המתודות הבאות:

1. **equals()**: מקבלת Object כפרמטר ובודקת אם הפרמטר שווה לאובייקט שעליו הופעלה המתודה.
המתודה תחזיר true רק כאשר שני האובייקטים מצביעים לאותו האובייקט.
אפשר לדרוס את המתודה הזאת (override) ולהתאים אותה לתנאים שלנו כמו למשל להגדיר שהאובייקטים שווים אם במקרה הערך של השדה int data = 50 בשניהם וכו'...

2. **hashCode()**: מחזירה מספר שלם המייצג ערך hash, המשמש ל-Hash Table. הוא מחזיר ערך המבוסס על כתובת האובייקט בזיכרון.

3. **toString()**: מחזירה מחרוזת המייצגת את האובייקט, לרוב מוגדרת מחדש כדי להציג מידע נוסף.

הערות

1. גם מערכים נחשבים כאובייקטים ולכן אפשר להפעיל עליהם את המתודות של Object שהוזכרו.
2. כאשר מגדירים מחדש את equals(), רצוי להגדיר גם את hashCode() כדי שאובייקטים המוגדרים כשווים יחזירו ערך hash זהה.

הטיפוס enum

טיפוס בן מנייה המייצג קבוצה של קבועים.

הצהרת Enum יכולה להיעשות מחוץ למחלקה או בתוך המחלקה, אך לא בתוך מתודה.

```
enum Color {
      RED, GREEN, BLUE;
}

public class Test {
      public static void main(String[] args) {
            Color c1 = Color.RED;
            System.out.println(c1);
      }
}
```

מדפיס:

```
RED
```

Comparator & Comparable

Comparable

ממשק המגדיר אסטרטגיה של השוואת האובייקט עם אובייקטים אחרים מאותו סוג.

```
public class Player implements Comparable<Player> {

    @Override
    public int compareTo(Player otherPlayer) {
        return Integer.compare(getRanking(), otherPlayer.getRanking());
    }
}
```

סדר המיון נקבע על ידי ערך ההחזר של שיטת compareTo().

The *Integer.compare(x, y)* returns -1 if *x* is less than *y*, returns 0 if they're equal, and returns 1 otherwise.

Comparator

ממשק ה-Comparator משמש להשוואת אובייקטים של מחלקות שהוגדרו על ידי המשתמש.

אובייקט זה מסוגל להשוות בין שני אובייקטים משני סוגים שונים. בפונקציה הבאה יש השווה obj1 עם obj2:

```
public class PlayerRankingComparator implements Comparator<Player> {

    @Override
    public int compare(Player firstPlayer, Player secondPlayer) {
       return Integer.compare(firstPlayer.getRanking(), secondPlayer.getRanking());
    }
}
```

Comparator מול Comparable

1. לפעמים אנחנו לא יכולים לשנות את קוד המקורי של המחלקה אם נרצה למיין את האובייקטים שלה, ובכך לא נעשה שימוש ב- Comparable.
2. השימוש ב- Comparator מאפשר לנו להימנע מהוספת קוד נוסף לשיעורי התחום שלנו.
3. אנו יכולים להגדיר מספר אסטרטגיות השוואה שונות ב-Comparator אשר אינן אפשריות בשימוש Comparable.

Iterator & Iterable

Iterator משתמש ב- Collection של Java .

ממשק גנרי המאפשר לסרוק איברים של אוסף באמצעות הפעולות הבאות:

1. **hasNext()** - מחזיר true במידה וקיימים איברים לסריקה.
2. **next()** - מחזיר את האיבר הבא בסריקה מטיפוס Object.
3. **remove()** - מוציא מהאוסף את האיבר האחרון שהוחזר על ידי האיטרטור.

חיסרון

1. אפשר לסרוק רק מכיוון אחד - קדימה.
2. לא ניתן להוסיף או להחליף איברים בסריקה בעזרת איטרטור.

```
class Test {
      public static void main(String[] args) {
            ArrayList<String> list = new ArrayList<String>();
            list.add("A");
            list.add("B");
            list.add("C");
            Iterator iterator = list.iterator();
            while (iterator.hasNext())
                  System.out.print(iterator.next() + " ");
      }
}
```

מדפיס:

```
A B C
```

Iterable

הממשק Collection יורש מהממשק Iterable המגדיר מתודה המחזירה Iterator המאפשר לבצע איטרציה כל איברי האוסף, כלומר לקבל את איברי האוסף בזה אחר זה. אם ה-collection הוא iterable, ניתן לאתחל אותו באמצעות איטרטור (וכתוצאה מכך ניתן להשתמש בו עבור כל לולאה).

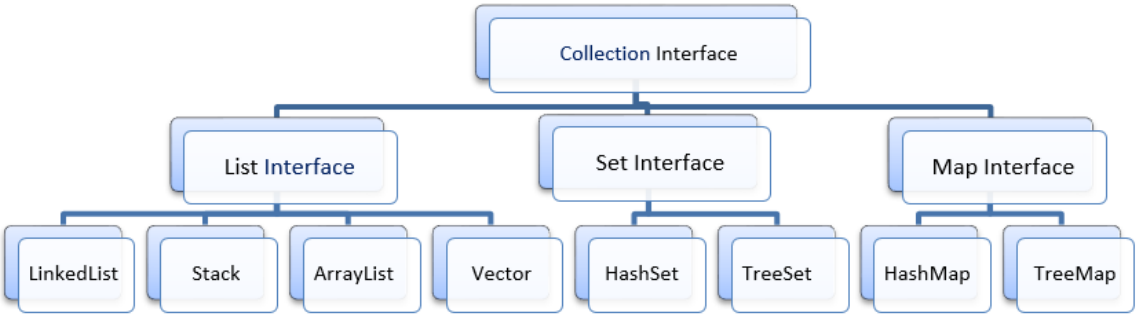
```
class SomeClass implements Iterable<String> {
    ...
}
class Main {
   public void method() {
      SomeClass someClass = new SomeClass();
      .....
      for(String s : someClass) {
      //do something
      }
   }
}
```

Java Collection

כוללת מבני נתונים מוכרים, כגון: תור, מחסנית, טבלת Hash וקבוצה שמומשו בצורה סטנדרטית.
מבני נתונים אלה נקראים **אוספים** מכיוון שהם מכילים אוספים של אובייקטים מאותו הסוג.
האוספים מוגדרים באמצעות ממשקים ומחלקות גנריות המאפשרים להגדיר את טיפוס איברי האוסף בעת היצירה.
בראש היררכיית האוספים קיים ממשק גנרי הנקרא Collection<E>, המגדיר את הפעולות הבסיסיות שאפשר לבצע על אוסף כלשהו. פעולות אלה כוללות לפי הטבלה הבאה

Method	Description
add(Object)	This method is used to add an object to the collection.
clear()	This method removes all of the elements from this collection.
contains(Object o)	This method returns true if the collection contains the specified element.
isEmpty()	This method returns true if this collection contains no elements.
iterator()	This method returns an iterator over the elements in this collection.
remove(Object o)	This method is used to remove the given object from the collection. If there are duplicate values, then this method removes the first occurrence of the object.
size()	This method is used to return the number of elements in the collection.
stream()	This method is used to return a sequential Stream with this collection as its source.

היררכיה של ממשקים ומחלקות



מבנים מסוג List

מבנה מסוג זה הוא מחלקה אשר מתארת מבנה נתונים המאפשר להתייחס אל הנתונים של המנה כנתונים שמוחזקים בסדר מסוים, ובאופן אשר מאפשר לאותו נתון **להופיע יותר מפעם אחת.**

מחלקות מסוג List הן: LinkedList, Vector, Stack ArrayList.

מבנים מסוג Set

מבנה מסוג זה מחזיק את כל אחד מן האובייקטים שלו כעותק אחד בלבד.

כל אובייקט במבנה נתונים מסוג **Set לא יוכל להופיע יותר מפעם אחת.**

מחלקות מסוג Set הן:

HashSet- מחלקה זו מתארת מבנה נתונים המיישם את **Set** וממומש באמצעות המחלקה **HashTable.**

הסיבוכיות של add, remove, contains בדרך כלל היא $O(1)$.

נחשבת כ-$O(1)$במקרים שבהם אותו ערך של hashCode מתקבל לשני ערכים שונים הוא די נדיר, כאשר ב-hash function טובה ו- load factor סביר. במקרה הגרוע היא $O(n)$.

מבנים מסוג Map

מבנה מסוג זה מחזיק לכל אלמנט – אלמנט נוסף המשמש עבורו כ-key (מפתח, שם מזהה).

לכל אלמנט משתייך key ייחודי אחד בלבד.

מחלקות מסוג Map הן:

HashMap - ממומש באמצעות HashTable, האלמנטים אינם ממוינים, מאפשר למפתח ערך **null.**

סיבוכיות של get, put בדרך כלל היא $O(1)$. נחשבת כ-$O(1)$ במקרים שבהם אותו ערך של hashCode מתקבל לשני ערכים שונים הוא די נדיר, כאשר ב-hash function טובה ו- load factor סביר. במקרה הגרוע היא$O(n)$.

חריגות

המחלקה Exception - היא מחלקת כל החריגים.
חריגות נתפוס ונזרוק שגיאה בצורה הבאה:

```
try{
...
}
//some code which may raise an exception
catch(Exception e){
// partial handling
    throw new Exception("....",e);
}
```

אם יש מצב חריג בבלוק ה-try , הוא נתפס ומטופל בבלוק ה-catch.
בהמשך הטיפול זורקים מצב חריג חדש, הכולל בתוכו את המצב החריג המקורי.
המחלקה Exception יורשת מ-Throwable מתודות כמו printStackTrace().
חריגים מובנים הם החריגים הקיימים בספריות Java ונועדו להסבר על מצבי שגיאה מסוימים:

סוג	הסבר
ArithmeticException	זורק שגיאה של פעולות אריתמטיות שגויות
ArrayIndexOutOfBoundsException	נזרק כאשר מצביעים באינדקס שמחוץ לטווח המערך
ClassNotFoundException	נזרק כאשר לא נמצא מחלקה
FileNotFoundException	נזרק כאשר לא ניתן לפתוח קובץ כלשהו או שלא בגישת המשתמש
IOException	נזרק בזמן פעולות input-output
InterruptedException	נזרק בזמן שימוש בThreads כמו wait, sleep ועוד...
NoSuchFieldException	נזרק כאשר לא קיים שדה כזה במחלקה בה אנו נמצאים
NoSuchMethodException	נזרק כאשר לא קיים מתודה כזאת
NullPointerException	נזרק כאשר אנו מצביעים על ערך null
NumberFormatException	נזרק כאשר לא ניתן להמיר String לערך מספרי
RuntimeException	זורק כל שגיאה כלשהי במהלך זמן הריצה

קלט ופלט I/O

תרגום המילה **stream** לעברית הוא **זרם**, ואכן, האובייקטים שמהווים streams מתארים זרימה של bytes אשר מהווה, למעשה, זרימה של נתונים.
את ה-streams שמיוצגים על ידי המחלקות ששייכות ל**java.io** -package **-** ניתן לחלק לשתי קבוצות:

input streams
אלה הם מקורות שניתן לקלוט/לקרוא מהם בתים (**byte**). למקורות אלה לא ניתן לכתוב/לשלוח בתים (**byte**) בחזרה.

output streams
אלה הם יעדים שניתן לשלוח/לכתוב אליהם בתים(byte). מיעדים אלה לא ניתן לקרוא/לקבל בתים(byte) חזרה.

בנאי המשמש ליצירת אובייקט חדש מטיפוס **FileInputStream** אשר יקושר לקובץ ששמו נשלח כ-String אל הבנאי. הקובץ חייב להיות קיים, וניתן לקריאה.

בנאי ליצירת אובייקט **FileOutputStream** חדש שמקושר לקובץ ששמו name. אם השם שניתן הוא שם של ספריה ולא שם של קובץ, או שהקובץ לא קיים וגם לא ניתן לייצור אותו, או שהקובץ לא ניתן לפתיחה מכל סיבה שהיא אז נזרק FileNotFoundException.

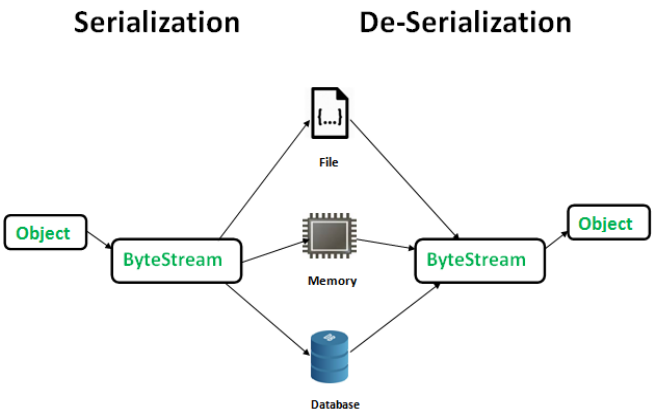
המחלקות ObjectInputStream ו- ObjectOutputStream
שתי מחלקות אלה משמשות לכתיבת/קריאת **אובייקטים** וגם לכתיבת/קריאת טיפוסים בסיסיים (פרימיטיביים).
כדי שניתן יהיה לכתוב אובייקט ל-ObjectOutputStream על המחלקה שלו (או אחת המחלקות שמעליו בהיררכית ההורשה) ליישם את הממשק: **Serializable**.
האובייקטים שנכתבים ל-ObjectOutputStream חייבים להיקרא בסדר שזהה לסדר כתיבתם.

Serialization & Deserialization

מנגנון העוסק בכתיבה ובקריאה של אובייקטים בתהליך הנקרא serialization הנשמר כולל את פרטי המחלקה ואת ערכם של שדות המופע של האובייקט.

Serialization הוא תהליך שאפשר לבצע רק על אובייקטים שהמחלקה שלהם מממשת את הממשק **Serializable** (implements Serializable). ממשק זה אינו מכיל מתודות, והוא רק מציין שהמחלקה מאפשרת serialization.

serialization לאובייקט פירושו להמיר את מצבו לבתים (bytes).
נשמור את האובייקט לפי ההמרה בקובץ וכך נוכל לשחזר את האובייקט בדיוק כפי שהיה לפי מצב הבתים שלו.



המושג **Deserialization** הוא התהליך ההפוך הממיר את זרם הבתים הסדרתי בחזרה לאובייקט בזיכרון המכונה.

```
class Demo implements java.io.Serializable {
      public int a;
      public String b;

      public Demo(int a, String b)
      {
            this.a = a;
            this.b = b;
      }

}
```

```
class Test {
	public static void main(String[] args) {
		Demo object = new Demo(1, "geeksforgeeks");
		String filename = "file.ser";

		// Serialization
		try{
			//Saving of object in a file
			FileOutputStream file = new FileOutputStream(filename);
			ObjectOutputStream out = new ObjectOutputStream(file);
			// Method for serialization of object
			out.writeObject(object);
			out.close();
			file.close();

		 System.out.println("Object has been serialized");
		}
		catch(IOException ex) {
			System.out.println("IOException is caught");
		}
		Demo object1 = null;

		// Deserialization
		try{
			// Reading the object from a file
			FileInputStream file = new FileInputStream(filename);
			ObjectInputStream in = new ObjectInputStream(file);

			// Method for deserialization of object
			object1 = (Demo)in.readObject();
			in.close();
			file.close();

			System.out.println("Object has been deserialized ");
			System.out.println("a = " + object1.a);
			System.out.println("b = " + object1.b);
		}
		catch(IOException ex) {
			System.out.println("IOException is caught");
		}

		catch(ClassNotFoundException ex) {
			System.out.println("ClassNotFoundException is caught");
		}

	}
}
```

קבצי JSON

JSON - **J**ava**S**cript **O**bject **N**otation הוא פורמט החלפת נתונים, מבוסס טקסט, שאינו תלוי בשפה, שקל לקריאה ולכתיבה לבני אדם ולמכונות.
JSON יכול לייצג שני סוגים:

אובייקט הוא אוסף לא מסודר של אפס או יותר זוגות שם / ערך.

```
{
   "Class":{ "name":"Peter", "age":20, "score": 70.05}
}
```

מערך הוא רצף מסודר של אפס או יותר ערכים. הערכים יכולים להיות מחרוזות, מספרים, בוליאנים, null או מערכים נוספים ואובייקטים.

```
{
   "arr" : ["black", "red", " Green"]
}

{
   "collection" : [
            {"one" : 100},
            {"two" : 200},
            {"three" : 300}
    ]
 }
```

מחלקות עיקריות ב-JSON	
JSONParser	נועד לניתוח מחרוזת JSON, הבנאי מקבל String או java.io.Reader
JSONObject	יורשת מ-HashMap ומאחסנת (ערך,מפתח).
JSONArray	יורשת מ-ArrayList ומייצגת אוסף
JSONValue	נועד להמרת מחרוזות JSON באובייקטים של Java.

כתיבה (write) לקובץ JSON

```
public class JSONWriteExample {
    public static void main(String[] args) throws FileNotFoundException {
        // creating JSONObject
        JSONObject jo = new JSONObject();
        // putting data to JSONObject
        jo.put("firstName", "John");
        jo.put("lastName", "Smith");
        jo.put("age", 25);

        // for address data, first create LinkedHashMap
        Map m = new LinkedHashMap(4);
        m.put("streetAddress", "21 2nd Street");
        m.put("city", "New York");
        m.put("state", "NY");
        m.put("postalCode", 10021);

        // putting address to JSONObject
        jo.put("address", m);

        // for phone numbers, first create JSONArray
        JSONArray ja = new JSONArray();
        m = new LinkedHashMap(2);
        m.put("type", "home");
        m.put("number", "212 555-1234");

        // adding map to list
        ja.add(m);

        m = new LinkedHashMap(2);
        m.put("type", "fax");
        m.put("number", "212 555-1234");

        // adding map to list
        ja.add(m);

        // putting phoneNumbers to JSONObject
        jo.put("phoneNumbers", ja);

        // writing JSON to file:"JSONExample.json" in cwd
        PrintWriter pw = new PrintWriter("JSONExample.json");
        pw.write(jo.toJSONString());

        pw.flush();
        pw.close();
    }
}
```

קריאה (read) מקובץ JSON

```
public class JSONReadExample {
    public static void main(String[] args) throws Exception {
        // parsing file "JSONExample.json"
        Object obj = new JSONParser().parse(new FileReader("JSONExample.json"));

        // typecasting obj to JSONObject
        JSONObject jo = (JSONObject) obj;

        // getting firstName and lastName
        String firstName = (String) jo.get("firstName");
        String lastName = (String) jo.get("lastName");

        System.out.println(firstName);
        System.out.println(lastName);

        // getting age
        long age = (long) jo.get("age");
        System.out.println(age);

        // getting address
        Map address = ((Map)jo.get("address"));

        // iterating address Map
        Iterator<Map.Entry> itr1 = address.entrySet().iterator();
        while (itr1.hasNext()) {
            Map.Entry pair = itr1.next();
            System.out.println(pair.getKey() + " : " + pair.getValue());
        }

        // getting phoneNumbers
        JSONArray ja = (JSONArray) jo.get("phoneNumbers");

        // iterating phoneNumbers
        Iterator itr2 = ja.iterator();

        while (itr2.hasNext()) {
            itr1 = ((Map) itr2.next()).entrySet().iterator();
            while (itr1.hasNext()) {
                Map.Entry pair = itr1.next();
                System.out.println(pair.getKey() + " : " + pair.getValue());
            }
        }
    }
}
```

ספריית Gson

ספריית Java שגוגל פיתחה כדי להמיר אובייקטים של Java לייצוג JSON.

למה Gson?

1. קל לשימוש - ממשק ה- API של Gson מספק חזית ברמה גבוהה לפשט את מקרי השימוש הנפוצים.
2. מספק serialization אוטומטי לרוב האובייקטים.
3. מהיר ומתאים לגרפים או מערכות עצמים גדולים.
4. JSON נקי - Gson יוצר תוצאה JSON נקייה וקומפקטית וקל לקריאה.
5. ספריית קוד פתוח.

שלב 1 - יצירת Gson באמצעות GsonBuilder:

```
GsonBuilder builder = new GsonBuilder();
builder.setPrettyPrinting();
Gson gson = builder.create();
```

כאשר `setPrettyPrinting` מאפשר קריאה נקייה יותר של קובץ JSON.

שלב 2 - קריאת קובץ JSON לאובייקט (Deserialization):

```
Student student = gson.fromJson(jsonString, Student.class);
```

מקבל מחרוזת של JSON או כתובת מקום של קובץ JSON , ואת סוג האובייקט אליו נבצע את הקריאה.

שלב 3 - שמירת קובץ JSON מאובייקט (Serialization):

```
jsonString = gson.toJson(student);
```

מקבל את האובייקט ומחזיר מחרוזת JSON, ומכאן יש רק להמשיך עם השלבים בעמוד הקודם עבור שמירת קובץ JSON.

תהליכים - Threads

תוכנית רגילה מתבצעת בדרך כלל בצורה סדרתית. במילים אחרות, בכל נקודה בזמן מתבצעת, לכל היותר, פקודה אחת. - תוכנית כזאת נקראת **תוכנית סדרתית**.

בשונה מתוכנית סדרתית, **תוכנית מרובת-תהליכים (multi-threaded)** מאפשרת לבצע כמה פעולות במקביל.

תהליך (**thread**) מוגדר כסדרת פקודות, המתבצעת באופן סדרתי.
בתוכנית מרובת תהליכים אפשר להגדיר כמה תהליכים המתבצעים **באותו הזמן**.

- המשתנים שבהם משתמש תהליך יכולים להיות משתנים פרטיים או משתנים משותפים.
  - להבדיל ממשתנים פרטיים, למשתנים משותפים יכולים לגשת כמה תהליכים.

- מהירות הריצה של תהליכים שונים יכולה להיות שונה. לכן, פרק הזמן שבו תהליכים מבצעים פקודות (אפילו פקודות זהות) משתנה מתהליך לתהליך, ועקב כך לתכנית מרובת-תהליכים יש מספר רב של ריצות אפשריות.

- תכנית מרובת-תהליכים יכולה להתבצע על מחשב שבו יש מעבד CPU יחיד או מחשב שיש לו מעבדים אחדים.
  - במקרה הראשון, כל התהליכים יתבצעו על אותו המעבד, והפקודות שלהם יתבצעו במשולב.
  - במקרה השני, תהליכים שונים עשויים להתבצע על מעבדים שונים בו- זמנית.

יתרונות של תוכנית מרובת תהליכים

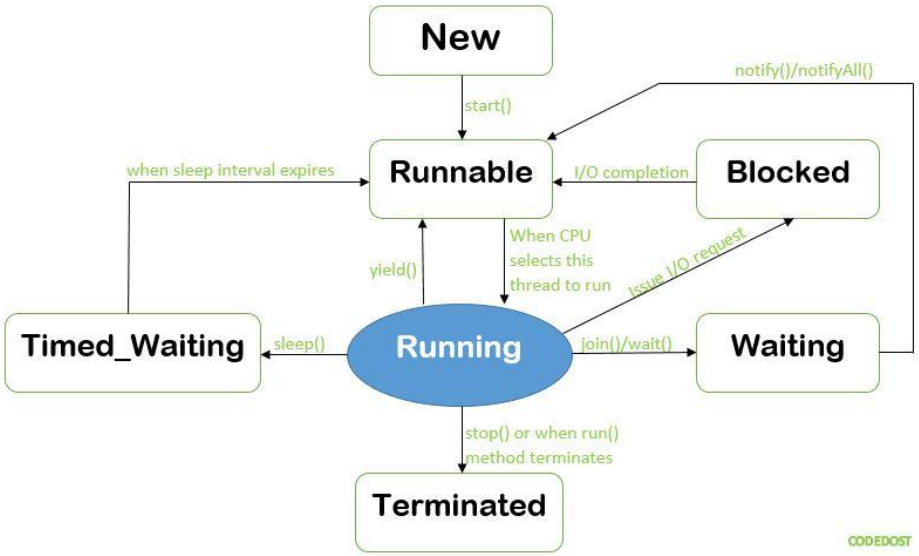
1. זמן תגובה מהיר לקלט בזמן אמת.
2. פתרון "טבעי" יותר של בעיה באמצעות חלוקת הבעיה לתת-בעיות, שבהן כל אחת ממומשת על-ידי תהליך אחר.
3. שיפור זמן הריצה של אלגוריתם (בפרט במחשב שיש לו כמה מעבדים).

חסרונות של תוכנית מרובת תהליכים

1. קשה יותר לתקן טעויות (debugging).
2. הצורך בסנכרון יכול לגרום במקרים מסוימים להאטה בזמן הביצוע.

מחזור החיים של תהליכים

כל תהליך (thread) בג'אווה נמצא במהלך חייו באחד מהמצבים הבאים:



1. **New :** תהליך שזה עתה נוצר, לאחר שנוצר התהליך יש להפעיל אותו באמצעות **`start()`** כדי להעביר אותו למצב של Runnable.
2. **Runnable :** התהליך מגיע למצב זה לאחר שהוא הופעל, למצב הזה יש שני תתי-מצבים:
   a. **Ready** : התהליך מוכן לריצה ברגע שיקבל מעבד פנוי.
   b. **Running** : תהליך שרץ כרגע בפועל.

   המעבר בין תתי-המצבים מתבצע על-ידי המתזמן של מערכת ההפעלה. רוב מערכות ההפעלה מקצות פרק זמן לכל תהליך. בחירת התהליך הבא נקבעת לפי רמה עדיפות של התהליך - **`yield()`**.
3. **Waiting** : הוא מבקש להמתין **`join()`** / **`wait()`** עד שתהליך אחר יבצע פעולה כלשהי. יציאה ממצב זה וחזרה למצב Runnable מתבצעת בעקבות קבלת איתות **`notify()`** מתהליך אחר.
4. **Timed Waiting** : תהליך נכנס למצב זה כאשר הוא מבקש להמתין זמן קצוב -**`sleep()`**. יציאה ממצב זה וחזרה למצב Runnable מתבצעת כאשר נגמר הזמן הקצוב.
5. **Blocked** : תהליך נכנס למצב זה כאשר הוא מנסה לבצע פעולה שלא יכולה להתבצע במיידי ועליו להמתין עד שהפעולה תסתיים:
   a. פעולת קלט/פלט: כלומר כאשר תהליך מבקש לבצע פעולה זו, מערכת ההפעלה חוסמת את התהליך עד שהפעולה מסתיימת. בסיום הפעולה חוזר למצב Runnable.
   b. ניסיון לבצע קטע קוד המוגן מפני ביצוע בו-זמני: קורה כאשר הקטע תפוס על-ידי תהליך אחר, התהליך נחסם עד שהתהליך האחר משחרר את הקטע התפוס.
6. **Terminated** : תהליך שסיים את המשימה שלו או שסיים בעקבות מצב חריג. תהליך שמגיע למצב כזה לא יכול להחיות את עצמו ולחזור למצב Runnable.

מימוש Thread

ניתן ליצור Thread באמצעות שני צורות מימוש:

1. Thread חדש נוצר בעזרת ירושת המחלקה java.lang.Thread.
2. מימוש הממשק Runnable.

עבור צורה 1:

- ה-thread מתחיל את חייו בשיטת ה- ()**run** שנמצא במחלקת Thread, המחלקה היורשת שלנו תדרוס **MultithreadingDemo** את השיטה ()**run**.
- ניצור אובייקט של **MultithreadingDemo** במחלקה החדשה שלנו **Multithread** ונקרא לשיטה ()start כדי להפעיל את ה-thread שמימשנו.
- השיטה ()start תפעיל את השיטה ()**run** שנמצא במחלקה **MultithreadingDemo** .

```
class MultithreadingDemo extends Thread {
	public void run() {
		try {
			// Displaying the thread that is running
			System.out.println(Thread.currentThread().getId());

		}
		catch (Exception e) {
			// Throwing an exception
			System.out.println ("Exception is caught");
		}
	}
}

// Main Class
public class Multithread {
	public static void main(String[] args) {
		int n = 8; // Number of threads
		for (int i=0; i<n; i++)  {
			MultithreadingDemo object = new MultithreadingDemo();
			object.start();
		}
	}
}
```

קריאה לפונקציה:

```
Thread.currentThread().getId()
```

תציג לנו את מספר התהליך בכל קריאה לשיטה ()run.

עבור צורה 2:

- ניצור מחלקה חדשה **MultithreadingDemo** שמתממש את הממשק java.lang.Runnable ותדרוס את השיטה ()**run**.
- לאחר מכן, ניצור מחלקה חדשה **Multithread** שתיצור אובייקט חדש של **MultithreadingDemo** ככמות הפעמים של ריצת הלולאה ותפעיל ()start בכל פעם.

```
class MultithreadingDemo implements Runnable {
	public void run() {
		try {
			// Displaying the thread that is running
			System.out.println(Thread.currentThread().getId());

		}
		catch (Exception e) {
			// Throwing an exception
			System.out.println ("Exception is caught");
		}
	}
}

// Main Class
class Multithread {
	public static void main(String[] args) {
		int n = 8; // Number of threads
		for (int i=0; i<n; i++) {
			Thread object = new Thread(new MultithreadingDemo());
			object.start();
		}
	}
}
```

ההבדלים בין 2 הצורות

1. אם אנו יורשים את מחלקת Thread, המחלקה שלנו לא יכולה לרשת אף מחלקה אחרת מכיוון ש-Java אינה תומכת בירושה מרובה. אבל אם אנו מיישמים את ממשק ה- Runnable, המחלקה שלנו עדיין יכולה להרחיב מחלקות בסיס אחרות.
2. אנו יכולים להשיג פונקציונליות בסיסית של Thread על ידי ירושת מחלקת Thread מכיוון שהיא מספקת כמה שיטות מובנות כמו yield, interrupt וכו' שאינן זמינות בממשק Runnable.

צורה נוספת ומהירה יותר ליצירת Thread היא בעזרת מחלקה אנונימית חדשה שתממש את הממשק Runnable:

```
public class anonymousInit {
   public static void main(String[] args) {
       for(int i = 0 ; i < 5 ; i++) {

           Thread thread = new Thread(new Runnable() { // anonymous class
               public void run() {
                   System.out.println(Thread.currentThread().getId());
               }
               } // end ..able() { ...
           ); // end new Thread( ...

           thread.start();
       } // end for

   } // end main
} // end class
```

עדיפויות Threads

תהליך חדש new שמוכן ריצה יחליף את התהליך הנוכחי שרץ ב-Runnable אם הוא בעל עדיפות גבוהה יותר.

במחלקת Thread מוגדרים שלושה קבועים לעדיפות:

1. עדיפות מקסימלית: `Thread.MAX_PRIORITY = 10` .
2. עדיפות מינימלית: `Thread.MIN_PRIORITY = 1`.
3. עדיפות נורמלית: `Thread.NORM_PRIORITY = 5`.

במצבים מסוימים thread בעל עדיפות נמוכה יותר יכול להתבצע למרות שקיים thread בעל עדיפות גבוהה יותר בגלל שיקולים של מערכת ההפעלה, וזאת כדי למנוע מצב של **deadlock** או **starvation** .

```
class Thread3 implements Runnable {
   String name;

   public Thread3(String name) {
       this.name = name;
   }

   public void run() {
       System.out.println(this.name);
   }

}
```

```
public class priorityThreads {

   public static void main(String[] args) {

       Thread thread1 = new Thread(new Thread3("thread1"));
       Thread thread2 = new Thread(new Thread3("thread2"));

       thread1.setPriority(Thread.MIN_PRIORITY);
       thread2.setPriority(Thread.MAX_PRIORITY);

       thread1.start();
       thread2.start();
   }
}
```

פלט:

```
thread2
thread1
```

אפשר לראות שלמרות והפעלנו את thread1 לפני thread2 , בגלל של-thread2 יש עדיפות מקסימלית אז היא תופעל ראשון עם ()start ורק אחרי שתסיים, התהליך הבא בתור העדיפות, thread1 יופעל.

שינה של Thread

השיטה **Thread.sleep()** נועדה כדי "להקפיא" את התהליך לאורך זמן מוקצב במילישניות.

1. משהה את התהליך הנוכחי שרץ בבוא הרגע.
2. במידה ותהליך פעיל משבש תהליך יישן, יזרק השגיאה InterruptedException.
3. כדי להשתמש בשיטה זו יש לסגור אותה בתוך try n catch .

```
for(int i = 0 ; i < 5; i++) {
   Thread thread = new Thread(new Runnable() {
       public void run() {
           System.out.println(Thread.currentThread().getId());
       }
   }); // end anon class
   thread.start();
   try {
       Thread.sleep(2000); // 2 seconds
   }
   catch (InterruptedException e) {
       e.printStackTrace();
   }
} // end for-loop
```

השיטה ()join

מתודה זו – בהפעילנו אותה על אובייקט מטיפוס Thread נגרום לכך שה-thread שפעל(בעת הפעלתה) ישהה את פעולתו, ויחכה עד אשר ה-thread שמיוצג על ידי אותו אובייקט Thread (האובייקט שממנו הופעלה ()join) יסתיים. ניתן לשלוח אל המתודה הזו ערך מספרי שיבטא באלפיות השנייה את משך הזמן המקסימלי שבו ה-thread ישהה את עצמו. במקרה כזה, אם ה-thread האחר, אשר עליו הופעלה המתודה join , לא יסיים את חייו (לאחר שיעבור פרק הזמן האמור) אז השהייתו של ה-thread שהושהה תופסק.

```
try {
   thread.join();
}
catch (InterruptedException e) {
   e.printStackTrace();
}
```

שיטות נוספות	
isAlive()	פונקציה בוליאנית מחזירה אמת אם ה-thread רץ (במצב runnable) אחרת מחזירה שקר.
yield()	מתודה סטטית הגורמת לתהליך שקרא לה לוותר באופן זמני על המעבד, לטובת תהליכים אחרים בעלי עדיפות זהה או גבוהה יותר.
interrupt()	מתודה המשמשת לביצוע interrupt לתהליך.
interrupted()	מתודה סטטית המאפשרת לתהליך לבדוק אם ביצעו לו interrupt.
currentThread()	מתודה סטטית המחזירה מצביע לתהליך הרץ כרגע.

תהליכים מסוג Daemon

אפשר להגדיר כל תהליך ב-Java כתהליך מסוג Daemon.

- כדי להגדיר תהליך מסוג זה, יש לקרוא למתודה setDaemon עם פרמטר true.
- אפשר להשתמש במתודה isDaemon כדי לבדוק את סוג התהליך.
- לתהליכים אלה יש את העדיפות הנמוכה ביותר מתהליכים מסוג אחר.

תהליך מסוג Daemon מספק, בדרך-כלל, שירותים להתהליכים אחרים.
הגוף שלו (המתודה run) נכתב בדרך כלל, כלולאה אין-סופית, שבה ממתין התהליך לבקשות מתהליכים אחרים. תכנית מרובת תהליכים **מסתיימת** כאשר כל התהליכים שאינם מסוג Daemon מסתיימים.

תהליכי Daemon שימושיים למשימות תומכות ברקע כגון Garbage Collector, שחרור זיכרון של אובייקטים שאינם בשימוש והסרת רשומות לא רצויות מה-cache. רוב התהליכים ב- JVM - Java Virtual Machine

הם תהליכי Daemon.

ThreadPool

היא שיטה לבצע הפשטה בין כמות התהליכים הנדרשים למימוש האלגוריתם, לבין כמות התהליכים שרצים בזמן אמת על המחשב בגלל אילוצי משאבים (לרוב זיכרון, משאבי חישוב ליבות פיסיות וכו'). וכך המתכנת יכול להגדיר כמה תהליכים שצריך בעוד שהמערכת (המחלקה Threadpool) דואגת שכמות התהליכים שרצים באמת בצורה מקבילית תואמת את הגדרות "משאבי המערכת".
עיקר עניין בנושא הוא למחזר תהליכים שסיימו את השימוש שלהם, במקום לזרוק אותם וליצור אחד חדש.

ה- JVM יוצר יותר מדי תהליכים בו זמנית ויכול לגרום למערכת לנצל את כל הזיכרון עד הסוף.
זה מחייב את הצורך להגביל את מספר התהליכים שנוצרים כדי להימנע ממקרים כאלה.
Java מספקת את ממשק ה- Executor, את ממשק המשנה שלה -ExecutorService ואת
class-ThreadPoolExecutor, המיישמת את שני הממשקים הללו.
כדי להשתמש במאגרי Thread, ראשית אנו יוצרים אובייקט של ExecutorService ומעבירים אליו קבוצת משימות. מחלקת ThreadPoolExecutor מאפשרת להגדיר את הליבה ואת גודל ה-pool המרבי. הריצות המופעלות על ידי חוט מסוים מבוצעות ברצף.

```
public class ThreadPool implements Runnable {

   @Override
   public void run() {
       System.out.println(Thread.currentThread().getId());
   }

   public static void main(String[] args) {

       int MAX_THREADS = 5;
       ExecutorService executorService = Executors.newFixedThreadPool(MAX_THREADS);

       for(int i = 0 ; i < MAX_THREADS; i++) {
           executorService.submit(new ThreadPool());
       }
       executorService.shutdown();
   }

}
```

סנכרון תהליכים

אפשר לבצע סנכרון בין תהליכים בשתי צורות:

1. שימוש במוניטורים (**monitors**) ובמאפיין - `synchronized`.

האפיון `synchronized` (מילה שמורה בג'אווה) מציין שמתודה כלשהי משתמש כמה תהליכים הרצים במקביל. במקרה כזה, חשוב להגן על המתודה מפני קריאה בו בזמן של כמה תהליכים, כדי לוודא שהנתונים שהמתודה משתמשת בהם עקביים. לדוגמה, לא ייתכן מצב שבו ביצוע אחד של המתודה ישתמש בנתונים בזמן שביצוע אחר של המתודה משנה אותם.

2. שימוש במנעולים המממשים את הממשק Lock.

```
public class mySynchronized {
   private static int count = 0;
   public static synchronized void counting() {
       count++;
   }

   public static void main(String[] args) {

       Thread thread1 = new Thread(new Runnable() {
           public void run() {
               for(int i = 0 ; i < 10000; i++) {
                   counting();
               }
           }
       });
       Thread thread2 = new Thread(new Runnable() {
           public void run() {
               for(int i = 0 ; i < 10000; i++) {
                   counting();
               }
           }
       });
       thread1.start();
       thread2.start();
       try {
           thread1.join();
           thread2.join();
       }
       catch (InterruptedException e) {
           e.printStackTrace();
       }
       System.out.println(count); // prints 20000
   }
```

```
}
```

אם לא היינו מצרפים את `synchronized` היינו מקבלים כל פעם תוצאה אחרת עבור count וזאת מאחר כי פעולת הספירה (count++) איטית יותר מקצב הריצה של התכנית.
המונה יכול להגן על עצמו מפני גישה בו-זמנית. ברגע שתהליך כלשהו יבצע עליו עדכון, הוא "**ינעל**" את עצמו ולא יאפשר לתהליך אחר לבצע עליו עדכון עד אשר התהליך הקודם יסיים את העדכון.
בזמן שתהליך יבצע את המתודה לבדו, התהליכים האחרים יושהו עד סיום הביצוע, ואחר- כך ייבחר תהליך אחר שיורשה לבצע את המתודה וכו'.

הערה - הנעילה היא ברמה של אובייקט (לא של מחלקה).

בכל אובייקט קיים משתנה דגל בשם "**flag lock**". המילה השמורה `synchronized` מאפשרת שליטה מסוימת במשתנה זה. משתנה זה – כאשר הוא מודלק במסגרת פעולתו של thread נתון (וכל עוד אותו thread נתון לא כיבה אותו), הגישה אליו מתוך threads אחרים (אשר גם מנסים לגשת אליו תחת השפעתה של המילה השמורה synchronized) **לא מתאפשרת.**

מוניטור והמתודות wait() ו-notify()

נהוג לקרוא לאובייקט שאחת או יותר מהמתודות שלו מאופיינות כ- `synchronized` בשם **monitor**.
בזמן הריצה יש לכל מוניטור שתי קבוצות, שבהן **מחכים** תהליכים עד שיתפנה המוניטור.
קבוצות אלה נקראות **קבוצת הנכנסים (Lock Pool) וקבוצת המחכים (Waiting Pool).**

notify()	wait()
כאשר המתודה נקראת על ידי thread המחזיק את מנעול של המוניטור, הוא מודיע שה-thread בקרוב ימסור את המנעול.	כאשר המתודה נקראת על ידי thread המחזיק את מנעול של המוניטור, הוא משחרר את המנעול ונכנס לקבוצת המחכים (**Waiting Poo**l).
אחד מה-thread-ים הממתינים בקבוצת המחכים (Waiting Pool) יבחר באופן אקראי ויודיע על זאת. ה-thread הנבחר יוצא מקבוצת המחכים ונכנס לקבוצת הנכנסים (**Lock Pool**) בו הוא ממתין עד שהתהליך הקודם משחרר את המנעול וכאשר זה יקרה, התהליך הנבחר יקבל את המנעול של המוניטור. ברגע שהוא רוכש את א, המנעול, הוא נכנס למצב הפעלה בו הוא ממתין לזמן המעבד (CPU) ואז הוא מתחיל לפעול.	יכולים להיות כמו תהליכים במצב ההמתנה בכל פעם.

הערות

- מתודות אלה מוגדרות במחלקה Object (ולא במחלקה Thread!).

- אפשר לקרוא למתודות האלה רק מתוך מתודה המאופיינת כ-`synchronized` .

בעיות בעבודה עם Threads

1. **Racing** - כאשר התוצאה של פעולה מסוימת תלויה בתזמון פעולה ב- thread אחר, תלות במשאב משותף. אם אין סנכרון יכולה להיווצר בעיה.
2. **Starving** - כאשר thread מסוים אינו מקבל זמן לריצה ע"י ה- CPU . יכול לקרות בגלל בעיה בעדיפויות של ה- thread'ים.
3. **Deadlock** - כאשר 2 thread'ים אינם יכולים להמשיך בפעילותם מאחר וכל אחד ממתין לשני.

Volatile

מילה שמורה `volatile` משמשת כדי לציין שניתן להשתמש ולשנות את הערך של המשתנה ב-threads שונים:

- ערך המשתנה מעולם לא ישמר בזיכרון לוקאלי של ה- thread . כל הפעולות של write/read יתבצעו בצורה ישירה בתוך זיכרון "ראשי", כלומר באותו מקום שבו נמצא משתנה העצם של המחלקה.
- גישה למשתנה דומה לגישה למשתנה מסונכרן, כלומר למשתנה שנמצא בבלוק synchronized, המסונכרן לפי משתנה זה. צריך לציין כי java מאפשרת ל-thread לשמור משתנים משותפים למספר threads בזיכרון פנימי של ה-thread למען יעילות טובה יותר. עותקים זמניים אלה חייבים להיות מסונכרנים עם משתנה המקורי.

הבדלים בין `volatile` לבין `synchronized:`

תכונה	synchronized	volatile
סוג המשתנה	רק אובייקטים	אובייקטים או משתנה פרימיטיבי
האם אפשר ערך null?	לא	כן
מתי מתבצע הסנכרון	בכניסה לבלוק שלו	בכל גישה למשתנה

בדוגמה הבאה נראה את השימוש **ב-volatile** :

VolatileExample.java

```
public class VolatileExample {

   private volatile static int value = 0;

   public static void main(String[] args) {
       new DataReader().start();
       new DataWriter().start();
   }

   static class DataReader extends Thread {

       public void run() {
           int local_value = value;
           while (local_value < 5) {
               if (local_value != value) {
                   System.out.println("Global value has changed to: "+ value);
                   local_value = value;
               }
           }
       }
   }

   static class DataWriter extends Thread {

       public void run() {
           int local_value = value;
           while (value < 5) {
            System.out.println("Incrementing global value to "+ (local_value + 1));
               value = ++local_value;
               try {
                   Thread.sleep(500);
               } catch (InterruptedException e) {
                   e.printStackTrace();
               }
           }
       }
   }
}
```

נקבל את הפלט הבא:

```
Incrementing global value to 1
Global value has changed to: 1
Incrementing global value to 2
Global value has changed to: 2
Incrementing global value to 3
Global value has changed to: 3
Incrementing global value to 4
Global value has changed to: 4
Incrementing global value to 5
Global value has changed to: 5
```

אם לא היינו מצרפים את ה-volatile היינו מקבלים את הפלט הבא:

```
Incrementing global value to 1
Incrementing global value to 2
Incrementing global value to 3
Incrementing global value to 4
Incrementing global value to 5
```

השתקפות - Reflection

Reflection הוא מנגנון המאפשר לקבל מידע על התוכנה בזמן ריצה. Reflection מאפשרת לקבל מידע על משתני עצם, מתודות ובנאים של המחלקות. Reflection גם מאפשרת לבצע פעולות על מתודות ומשתני עצם של מחלקות.

באמצעות ממשק API Reflection Java ניתן :

- לקבוע לאיזה מחלקה האובייקט שייך.
- לקבל מידע על הרשאות של מחלקה, משתנה עצם, מתודות, בנאים ומחלקת האב.
- לקבל משתנים קבועים ומתודות של ממשק.
- יצירת אובייקט של מחלקה ששמה אינו ידועה עד זמן ריצה.
- לקבל ולקבוע תכונות האובייקט
- לקרוא למתודה של אובייקט

פעולות אטומיות

פעולות אטומיות מהוות לעתים תחליף יעיל יותר לשימוש במנעולים, ועל כן אלגוריתמים חסרי נעילות נדרשים לעשות שימוש בפעולות אטומיות.

להמחשת הצורך בפעולה אטומית, נסתכל על הפשטה של הפקודות שמרכיבות את פעולת הקידום (הגדלה ב-1 של ערך בזיכרון, increment):

1. התהליך קורא מהזיכרון את הערך שבמקום X
2. התהליך מגדיל את הערך ב-1
3. התהליך כותב למקום X את הערך החדש

כעת נסתכל על 2 תהליכים שמבצעים פעולת קידום על אותו מקום בזיכרון. כיוון שיש לנו CPU יחיד, הרי שכדי ששני התהליכים יבצעו את הפעולה, חייבת להיות החלפת הקשר (החלפת 2 תהליכים באמצעות מעבד אחד)

תרחיש תקין

הערך גדל פעמיים: פעם על ידי תהליך א', ולאחר מכן פעם נוספת על ידי תהליך ב'. זו התוצאה הרצויה:

1. תהליך א' קורא מהזיכרון את הערך שבמקום X
2. תהליך א' מגדיל את הערך ב-1
3. תהליך א' כותב למקום X את הערך החדש

מתבצעת החלפת הקשר

4. תהליך ב' קורא מהזיכרון את הערך שבמקום X
5. תהליך ב' מגדיל את הערך ב-1
6. תהליך ב' כותב למקום X את הערך החדש

תרחיש לא תקין

הערך גדל לבסוף רק פעם אחת: השינוי שביצע תהליך ב' נדרס על ידי הכתיבה לזיכרון של תהליך א'. במידה ופעולת ההגדלה הייתה פעולה אטומית הרי שהחלפת ההקשר שהתבצעה אחרי שלב 2 לא הייתה אפשרית, והשגיאה הייתה נמנעת.

1. תהליך א' קורא מהזיכרון את הערך שבמקום X
2. תהליך א' מגדיל את הערך ב-1

מתבצעת החלפת הקשר

3. תהליך ב' קורא מהזיכרון את הערך שבמקום X
4. תהליך ב' מגדיל את הערך ב-1
5. תהליך ב' כותב למקום X את הערך החדש

מתבצעת החלפת הקשר

6. תהליך א' כותב למקום X את הערך החדש

בעיבוד מקבילי הסיכון גדול עוד יותר כיוון שמספר מעבדים פועלים במקביל ועובדים על זיכרון משותף, על כן חשיבות השימוש בפעולות אטומיות גדלה.

פעולות אטומיות נפוצות

מעבדים שונים במחשבים שונים מאפשרים פעולות אטומיות שונות. הפעולות הנפוצות הן:

- קידום - הוספת 1 (increment) או החסרת 1 (decrement) מיחידת זיכרון.
- השמה - החלפת ערך ביחידת זיכרון לערך חדש.
- השווה והחלף - החלפת ערך ביחידת זיכרון לערך חדש רק במידה ויש ערך מסוים ביחידת הזיכרון לפני ההחלפה. פעולה זו יכולה להיכשל.
- פעולות אריתמטיות - הפעלת פעולה אריתמטית על יחידת זיכרון (כגון הוספה, החסרה, הכפלה וכדומה).
- פעולות ביטים - הפעלת פעולת ביטים על יחידת זיכרון (כגון and, or, xor וכדומה).

רוב הפעולות האטומיות מחזירות את הערך הקודם שהיה ביחידת הזיכרון לפני הפעולה.

S.O.L.I.D

SOLID היא גישה עיצובית מובנית שמבטיחה שהתוכנה שלך היא מתוכננת טוב, קלה לתחזוקה ומובנת.

- זה מצמצם את התלות כך שניתן לשנות קטע קוד מבלי להשפיע על קטעי הקוד האחרים.
- העקרונות נועדו להפוך את העיצוב לקל יותר ומובן יותר.
- באמצעות העקרונות, המערכת ניתנת לתחזוקה, ניתנת לבדיקה, להרחבה, לניתוח ולשימוש חוזר.
- מונע עיצוב רע של התוכנית.

Single Responsibility Principle - “Just because you can, doesn’t mean you should”.

כל מחלקה חייבת לבצע תפקיד אחד ויחיד.
מימוש של כמה תפקידים במחלקה אחד יכול לבלבל את הקוד ובמידה ונדרש שינוי כלשהו זה עשוי להשפיע על כל המחלקה. דוגמה למחלקה **שלא** עומדת בעיקרון:

Student.java

```
public class Student {
   public void printDetails(){ ... }
   public void calculatePercentage() { ... }
   public void addStudent() { ...}
}
```

פתרון - כדי להשיג את מטרת העיקרון, עלינו ליישם מחלקה נפרדת המבצעת פונקציונליות אחת בלבד.

Student.java

```
public class Student {
   public void addStudent() { ... }
}
```

PrintStudentDetails.java

```
public class PrintStudentDetails {
   public void printDetails() { ... }
}
```

Percentage.java

```
public class Percentage {
   public void calculatePercentage() { ... }
}
```

Open-Closed Principle

כל מחלקה צריכה להיות פתוחה להוספות , אך לא לשינויים.
יש לנו מחלקה בשם Person עם כמה פונקציות:

Person.java

```
public class Person {
   public String firstName { get; set; ....  }
   public String secondName { get; set; .... }
   public int age { get; set; ... }
}
```

כעבור כמה חודשים הקוד שלך גדל באופן הדרגתי ובמקרה כלשהו אתה מתמודד עם כמה נתונים הקשורים לאנשים שמתעסקים בהנדסה (Engineer), אם נוסיף עוד כמה פונקציות למחלקת Person, שקשורות למהנדסים, אנחנו נעשה שימוש חוזר במחלקה Person וכאן אנחנו מפרים את העיקרון.

פתרון - נצור מחלקה חדשה שנקראת Engineer שיורשת ממחלקת Person:

Engineer.java

```
public class Engineer extends Person {
   public String domain { get; set; ... }
   public int yearsOfExperience { get; set; ... }
}
```

ולכן לא שינינו את המחלקה Person, כלומר לא שינינו את המחלקה עצמה כי לא הוספנו את הפונקציות החדשות אליה, אלא יצרנו מחלקה שיורשת את המאפיינים שלה ומוסיפה אליהם תכונות שרלוונטיות לצרכים שלה.

Liskov Substitution Principle - "If it looks like a Duck, quacks like a Duck, but needs Batteries - You probably have the wrong abstraction"

אם מחלקה A היא יורשת את המחלקה B, אז נוכל להחליף את B ב- A מבלי להפריע להתנהגות התוכנית.

דוגמה שמפרה את העיקרון:

Rectangle.java

```
public class Rectangle {
    private double height;
    private double width;
    public void setHeight(double h) { height = h; }
    public void setWidth(double w) { width = w; }
    ...
}
```

Square.java

```
public class Square extends Rectangle {
    public void setHeight(double h) {
        super.setHeight(h);
        super.setWidth(w);
    }
    public void setWidth(double h) {
        super.setHeight(h);
        super.setWidth(w);
    }
}
```

המחלקות מלמעלה אינן מצייתות לעיקרון מכיוון שלא ניתן להחליף את מחלקת הבסיס Rectangle במחלקה היורשת אותה Square.

במחלקת Square יש מגבלות נוספות, כלומר הגובה והרוחב חייבים להיות זהים.

לכן, החלפת המחלקה Rectangle במחלקה Square עלולה לגרום להתנהגות בלתי צפויה.

Interface Segregation Principle - “Don’t put your pineapple on my pizza!”

אנחנו לא צריכים לאלץ את הלקוח להשתמש בשיטות בהן הוא לא רוצה להשתמש.
אל תכריח אף לקוח ליישם ממשק שאינו רלוונטי עבורו.

דוגמה שמפרה את העיקרון:

Vehicle.java

```
public interface Vehicle {
    public void drive();
    public void stop();
    public void refuel();
    public void openDoors();
}
```

Bike.java

```
public class Bike implements Vehicle {

    // Can be implemented
    public void drive() {...}
    public void stop() {...}
    public void refuel() {...}

    // Can not be implemented
    public void openDoors() {...}
}
```

כפי שאפשר לראות, זה לא הגיוני שהמחלקה Bike תיישם את שיטת openDoors כי לאופנוע אין דלתות!

פתרון - כדי לתקן את זה, העיקרון מציע כי הממשקים יתפרקו למספר ממשקים מגובשים קטנים, כך שאף מחלקה לא תיישם ממשק שלא רלוונטי אליה.

Dependency Inversion Principle

מחלקות ברמה גבוהה לא צריכים להיות תלויים במחלקות ברמה נמוכה.
שניהם צריכים להיות תלויים באבסטרקטיות (כמו למשל ממשקים).

דוגמה שמפרה את העיקרון - יש לנו מחלקה Car התלויה במחלקה Engine (רכב תלוי במנוע) :

```
public class Car {
    private Engine engine;
    public Car(Engine e) {
        engine = e;
    }
    public void start() {
        engine.start();
    }
}
public class Engine {
   public void start() {...}
}
```

הקוד יעבוד, בינתיים, אבל מה אם נרצה להוסיף סוג מנוע נוסף, נניח מנוע דיזל? זה ידרוש "שיקום" של המחלקה שלנו Car.

פתרון: במקום שהרכב יהיה תלוי ישירות במנוע, בואו נוסיף ממשק EngineInerface:

EngineInerface.java

```
public interface EngineInterface {
    public void start();
}
```

כעת אנו יכולים לחבר כל סוג של מנוע המיישם את ממשק המנוע למחלקת הרכב, וככה הרכב (המחלקה ברמה הגבוהה), לא תלויה במנוע (המחלקה ברמה הנמוכה).

```
public class Car {
    private EngineInterface engine;
    public Car(EngineInterface e) { engine = e; }
    public void start() { engine.start();}
}
public class PetrolEngine implements EngineInterface {
   public void start() {...}
}
public class DieselEngine implements EngineInterface {
```

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

תבניות עיצוב - Design Patterns

Design Pattern נותן פתרון לשימוש חוזר כללי לבעיות הנפוצות המופיעות בתכנון תוכנה.
התבנית מראה בדרך כלל קשרים ואינטראקציות בין מחלקות או אובייקטים.
הרעיון הוא לזרז את תהליך הפיתוח על ידי מתן תבנית פיתוח / תכנון מוכחת ומוכחת היטב.
תבניות עיצוב הם אסטרטגיות בלתי תלויות בשפה לתכנית לפתרון בעיה נפוצה.

דוגמה במציאות

יכול להיות רק נשיא פעיל אחד של המדינה בכל פעם ללא קשר לזהות האישית. זה נקרא תבנית Singleton.

מטרה

- שילוב של קטעים בנויים היטב ומנוסחים מראש בעלי שם מפורסם וידוע, גורם להבנה מה נמצא במערכת המידע.
- שימוש חוזר בתבנית שעברה הרבה מערכות לפני שאתה השתמשת בה, עשוי לחסוך כתיבת קוד חדש ודבר זה לא יוסיף באגים.

סוג תבנית	הסבר	תבניות מוכרות
Creational	תבניות הדואגות לפישוט התהליך יצירת אובייקטים ממחלקות ומנגנונים נכונים ליצירה של אובייקטים.	Factory, Singleton and Prototype
Structural	תבניות הדואגות לפישוט יצירת מבנים אחידים, גדולים, מורכבים ומסובכים, כך שהמבנה הנוצר יהיה פשוט, קל להבנה ולתחזוקה ומעוצב בצורה מבנית.	Adapter, Composite and Proxy
Behavioral	תבניות הדואגות לחלק המעשי-אלגוריתמי, ליצירת אחידות בשימוש באלגוריתמים נפוצים.	Iterator, Observer, State, Strategy and Visitor

גישות לבניית מחלקות

1. **DbC - Design by Contract (חוזה) -**

- **מה זה?** - גישה לעיצוב תוכנה, הוא מבטא "חוזה" בין המפרט ליישום, כלומר, הסכם בין שני צדדים: לקוח וספק. המפרט == הדרישות .
- **למה?** - הוא מאפשר תיעוד אמין, תמיכה חזקה באיתור באגים ומקל על שימוש חוזר בקוד.
- **סוגי מצבים ב-DbC:**
  - **preconditions -**
    - תנאים שחייבים להיות נכונים **לפני** קריאה לשיטה.
    - אין תנאים נוספים לאחר קריאה .

```
/**
* @require (obj != null) < -- Precondition
*/
public void remove(Object obj);
```

  - **postconditions -**
    - תנאים שחייבים להיות נכונים **לאחר** קריאה לשיטה.
    - אין תנאים אחרים לפני קריאה.

```
/**
* @ensure obj is not contains anymore < -- Postcondition
*/
public void remove(Object obj);
```

  - **invariants (מצבים קבועים)-**
    - דברים שחייבים להיות נכונים **לפני ואחרי** כל שיטה שמופעלת (תמיד).
    - חייב להיות נכון מיד לאחר הבנייה.

```
/**
* @invariant size() >= 0 always! < -- Invariant
*/
public interface Queue {
```

2. **DP - Defensive Programming (הגנתי) -**

- תכנות הגנתי דורש בדיקות של כל המצבים האפשריים וטיפול בהם.
הסכנה הגדולה ביותר היא בקלט של משתמש שהוא בלתי צפוי ואפילו יכול להיות זדוני.
תכנות הגנתי היא טכניקה שבה המתכנת מצפה לגרוע ביותר מכל קלט בכלל.
פירוש הדבר שהתוכנית מבצעת אימות כלשהו בכדי להבטיח כי נתונים לא נכונים לא יעובדו.
- שלושה כללים לתכנית הגנתית:

- הקוד צריך להיות כמה שיותר פשוט.
- לעולם אל תניחו דבר.
- השתמש בסטנדרטים מקובלים וקבועים.

סוגי תבניות

1. Singleton

מתוך תבניות בייצור עצמים (Creational Pattern).
תבנית זו מיועדת ליצירת מחלקה (כזו שאפשר לייצר ממנה אובייקטים) שיוצרת אך ורק אובייקט אחד ממנה. המחלקה נותנת גישה גלובלית לעצם היחיד.
מטרת הסינגלטון היא לשלוט ביצירת אובייקטים, להגביל את המספר לאחד אך לאפשר את הגמישות ליצור אובייקטים נוספים אם המצב משתנה. מכיוון שיש רק מופע אחד של סינגלטון, בכל זמן נתון, סינגלטון יתרחש פעם אחת בלבד במחלקה, בדיוק כמו שדות סטטיים.

דרישות:

- לכל היותר עצם אחד ממחלקה יהיה בתוכנית. משמעות הדבר שייתכן מצב בו לא יהיה גם עצם אחד – במידה ולא נעשה שימוש בו בריצת התכנית.
- יש להבטיח שלא יווצר יותר מעצם יחיד בתכנית אף לא כתוצאה משגיאה של המתכנת.
- יש לאפשר גישה לעצם היחיד מכל מקום בתכנית.

דוגמה:

```
public class Server {
   private static Server onlyInstance = null;

   private Server() { }

   public static Server getInstance() {
       if (onlyInstance == null) {
           onlyInstance = new Server();
       }
       return onlyInstance;
   }
}
```

יתרונות:

- יכול לממש ממשקים.
- יכול להיות בירושה.
- הוא מספק נקודה אחת אם גישה למופע מסוים, כך שהוא קל לתחזוקה.

2. Observer

מתוך קבוצת התבניות Behavioral Pattern.

Observer – הגדרת תלות בין אובייקט (הנקרא Subject או Observerable) למספר אובייקטים (הנקראים Observers), כך שכאשר אובייקט אחד משנה את מצבו כל האובייקטים התלויים בו מיודעים (Notify או Update) ומתעדכנים אוטומטית, בדרך כלל על ידי קריאה אחת השיטות שלהם.

דוגמה: סוכנות חדשות יכולה להודיע לערוצים כאשר היא מקבלת חדשות.
קבלת חדשות היא זו שמשנה את מצבה של סוכנות החדשות, והיא מיידעת (Notify) את הערוצים.

ניצור את המחלקה NewsAgency:

NewsAgency.java

```
public class NewsAgency {
    private String news;
    private List<Channel> channels = new ArrayList<>();

    public void addObserver(Channel channel) {
        this.channels.add(channel);
    }

    public void removeObserver(Channel channel) {
        this.channels.remove(channel);
    }

    public void setNews(String news) {
        this.news = news;
        for (Channel channel : this.channels) {
            channel.update(this.news);
        }
    }
}
```

NewsAgency הוא נצפה (observable), וכאשר החדשות מתעדכנות, המצב של NewsAgency משתנה. כאשר השינוי מתרחש, NewsAgency מודיע למשקיפים (observers - הערוצים) על זה על-ידי קריאה לשיטה שלהם: update() .

האובייקט הנצפה צריך להחזיק הפניות למשקיפים, ובמקרה שלנו, זה המשתנה `channels`.
הצופה (observer) , מחלקת NewsChannel, יש את שיטת ()update שמופעלת כאשר המצב של NewsAgency משתנה:

NewsChannel.java

```
public class NewsChannel implements Channel {
   private String news;

   @Override
   public void update(Object news) {
       this.setNews((String) news);
   }

   public String getNews() {
       return news;
   }

   public void setNews(String news) {
       this.news = news;
   }

}
```

לממשק Channel יש רק הצהרה אחת:

Channel.java

```
public interface Channel {
    public void update(Object o);
}
```

כעת, אם נוסיף מופע של NewsChannel לרשימת המשקיפים ונשנה את מצב NewsAgency, המופע של NewsChannel יעודכן (נקבל את הפלט "GOT IT!") :

Main.java

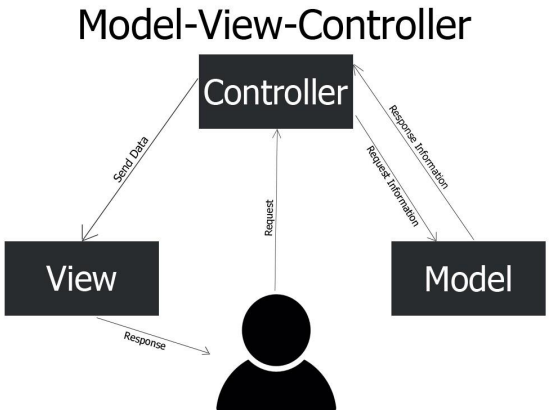
```
public class Main {
   public static void main(String[] args) {
       NewsAgency observable = new NewsAgency();
       NewsChannel observer = new NewsChannel();

       observable.addObserver(observer);
       observable.setNews("news");

       if(observer.getNews() == "news")
```

```
            System.out.println("GOT IT!");
    }
}
```

3. MVC - Model View Controller

תבנית זו היא תבנית עיצוב בהנדסת תוכנה המשמשת להפשטת יישום כלשהו.
התבנית מתארת טכניקה לחלוקת היישום לשלושה חלקים:

a. **מודל** - מכיל רק את נתוני היישום הטהורים, הוא לא מכיל לוגיקה המתאר כיצד להציג את הנתונים למשתמש.



b. **תצוגה או ממשק המשתמש** - מציג את נתוני המודל למשתמש. התצוגה יודעת לגשת לנתוני המודל, אך היא לא יודעת מה משמעות הנתונים הללו או מה המשתמש יכול לעשות כדי לתפעל אותם.

c. **בקר** - קיים בין התצוגה למודל. הוא מאזין לאירועים המופעלים על ידי התצוגה (או מקור חיצוני אחר) ומבצע את התגובה המתאימה לאירועים אלה. ברוב המקרים, התגובה היא להפעיל שיטה על המודל. מכיוון שהתצוגה והמודל מחוברים באמצעות מנגנון התראות, התוצאה של פעולה זו באה לידי ביטוי אוטומטית בתצוגה.

לעתים, המודל עשוי להודיע על שינויים המתחוללים בו לצדדים שלישיים נוספים, בדרך כלל באמצעות יישום של תבנית Observer.

4. Composite

מתוך קבוצת התבניות Structural Pattern.
גיבוש אובייקטים למבנה היררכי של עץ, אפשור התייחסות לאובייקטים כאובייקטים עצמאיים או באופן אחיד לצירוף של אובייקטים.

לתבנית יש 4 משתתפים:

a. **Component** - מגדיר את הממשק עבור אובייקטים בהרכבה. מממש התנהגות ברירת מחדל לממשק המשותף לכל המחלקות. מגדיר ממשק לגישה וטיפול בילדים.

b. **Leaf** - מייצג אובייקט שהוא עלה בהרכבה של אובייקט מורכב. אין לו ילדים, כלומר הוא יחידה בסיסית במבנה ההרכבה. מגדיר התנהגות עבור אובייקטים בסיסיים בהרכבה.

c. **Composite** - מגדיר התנהגות עבור אובייקטים שיש להם ילדים. מורכב ממרכיבים פשוטים יותר שהם הילדים שהוא מאחסן. מממש את הפעולות שקשורות לתפעול ילדים מתוך הממשק של Component.
d. **Client** - מתפעל אובייקטים בהרכבה דרך הממשק שמגדיר ה-Component. זהו למעשה התוכניתן. התוכניתן אינו צריך להיות מודע אילו אובייקטים מורכבים ואילו בדידים!!!

מתי נשתמש בתבנית הזאת?

נשתמש כאשר נרצה שהלקוחות יוכלו להתעלם מההבדלים בין הרכבה של אובייקטים לאובייקטים בודדים, ויתייחסו לשניהם באופן אחיד.

יתרונות:

- פשטות.
- פישוט עבור התוכניתן.
- אין צורך לשנות שום דבר בקליינט בעת הוספת Component חדש!

חסרונות:

היחיד הוא שתבנית זו עלולה להפוך את האפיון שלנו ליותר מדי כללי.
כאשר רוצים שיהיו ב- Composite רק סוגים מסוימים של Components .
הדרך היחידה לעשות זאת היא על ידי בדיקות שנעשות בזמן ריצה- למשל ע"י בדיקת של instanceof.

דוגמה מהמציאות

בחברה מסוימת, יש מנהלים כלליים ותחת מנהלים כלליים, יכולים להיות מנהלים ותחת מנהלים יכולים להיות מפתחים. כעת נוכל להגדיר מבנה עץ ולבקש מכל צומת לבצע פעולה נפוצה כמו getSalary().

דוגמה פרקטית

נניח שאנחנו רוצים לבנות מבנה היררכי של מחלקות בחברה.

Department.java - (זה ה- Component - רכיב הבסיס)

```
public interface Department {
   void printDepartmentName();
}
```

נגדיר שני רכיבי עלה: **FinancialDepartment** ו- **SalesDepartment** :

FinancialDepartment.java - (רכיב עלה ראשון)

```
public class FinancialDepartment implements Department {
   private Integer id;
```

```
    private String name;

    public void printDepartmentName() {
        System.out.println(getClass().getSimpleName());
    }

    // standard constructor, getters, setters
}
```

SalesDepartment.java - (רכיב עלה שני)

```
public class SalesDepartment implements Department {

    private Integer id;
    private String name;

    public void printDepartmentName() {
        System.out.println(getClass().getSimpleName());
    }

    // standard constructor, getters, setters
}
```

שתי המחלקות מיישמות את שיטת ()**printDepartmentName** ממרכיב הבסיס **Department**,
שם הן מדפיסות את שמות המחלקות עבור כל אחת מהן.
כמו כן, מכיוון שהם מחלקות עלה, הם אינם מכילים אובייקטים אחרים של המחלקה.

HeadDepartment.java - (זה ה-Composite)

```
public class HeadDepartment implements Department {
    private Integer id;
    private String name;
    private List<Department> childDepartments;

    public HeadDepartment(Integer id, String name) {
        this.id = id;
        this.name = name;
        this.childDepartments = new ArrayList<>();
    }

    public void printDepartmentName() {
        for(Department dep : childDepartments) {
            dep.printDepartmentName();
        }
    }
```

```
    public void addDepartment(Department department) {
        childDepartments.add(department);
    }

    public void removeDepartment(Department department) {
        childDepartments.remove(department);
    }
}
```

זוהי מחלקה מורכבת שכן היא מחזיקה אוסף של רכיבי מחלקה, כמו גם שיטות להוספה והסרה של אלמנטים מהרשימה.

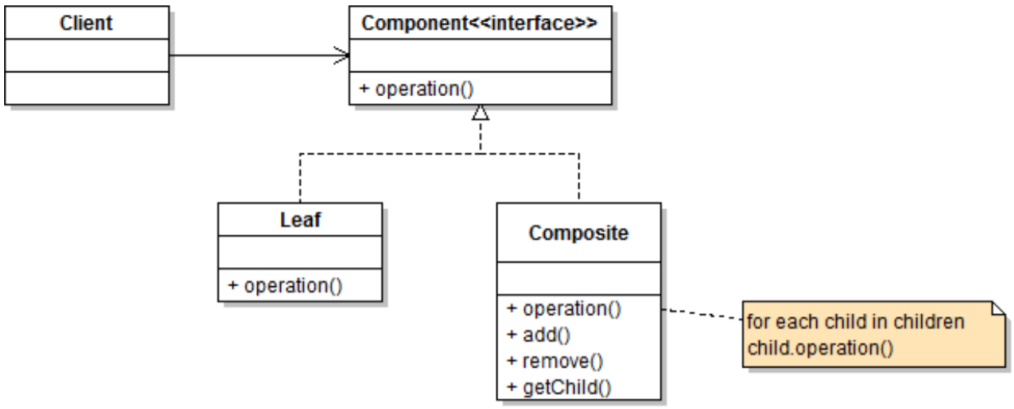
השיטה המורכבת ()**printDepartmentName** מיושמת על ידי מחרוזת ברשימת אלמנטים העלים והפעלת השיטה המתאימה לכל אחד.

CompositeDemo.java

```
public class CompositeDemo {
    public static void main(String args[]) {
        Department salesDepartment = new SalesDepartment(1, "Sales
        department");
        Department financialDepartment = new FinancialDepartment(2,
        "Financial department");

        HeadDepartment headDepartment = new HeadDepartment(3, "Head
        department");

        headDepartment.addDepartment(salesDepartment);
        headDepartment.addDepartment(financialDepartment);

        headDepartment.printDepartmentName();
    }
}
```

פלט:

```
SalesDepartment
FinancialDepartment
```



5. Iterator

תבנית איטרטור היא דפוס עיצוב פשוט יחסית ונמצא בשימוש תכוף.
יש הרבה מבני נתונים / אוספים זמינים בכל שפה שעובדים עם איטרטור.
לכל אוסף יש איטרטור המאפשר לו לסרוק את עצמיו ויחד עם זאת לוודא כי לא נחשף המימוש שלו.

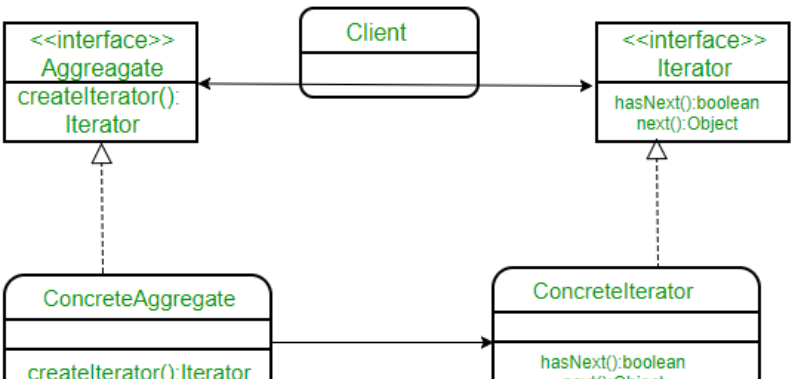
נניח שאנו בונים יישום המחייב אותנו לערוך רשימת התראות.
בסופו של דבר, חלק מהקודים שלנו ידרשו לבצע איטרציה על כל ההתראות.
אם היינו מיישמים את אוסף ההודעות שלנו כמערך, היינו מתייחס אליהם כ:

```
for (int i = 0; i < notificationList.length; i++)
     Notification notification = notificationList[i]);
```

ואם היה אוסף אחר כמו Set, עץ וכו 'דרך החזרה הייתה משתנה מעט.
עכשיו, מה אם נבנה איטרטור המספק דרך כללית לחזור על אוסף שאינו תלוי בסוגו?

```
// Create an iterator
Iterator iterator = notificationList.createIterator();

// It wouldn't matter if list is Array or ArrayList or
// anything else.
while (iterator.hasNext()) {
    Notification notification = iterator.next());
}
```

תבנית איטרטור מאפשר לנו לעשות בדיוק את זה. באופן רשמי הוא מוגדר כך:
דפוס האיטרציה מספק דרך לגשת לאלמנטים של האובייקט המקשר (Aggreagate) מבלי לחשוף את הייצוג הבסיסי שלו - כלומר, לא קיבלנו גישה למימוש של הנתונים באוסף כנדרש.



נציג תרגיל לדוגמה של מימוש איטרטור משלנו על רשימה מקושרת וגנרית (לפי ההנחיות אין צורך לממש את המתודה remove באיטרטור, אך בכל מקרה זה מופיע במימוש לידע כללי).

Node.java

```
public class Node<T> {

   private final T data;
   private Node<T> next;

   public Node(T data) {
       this.data = data;
       this.next = null;
   }

   public void setNext(Node<T> nextNode) {
       this.next = nextNode;
   }

   public Node<T> getNext() {
       return this.next;
   }

   public T getData() {
       return this.data;
   }

}
```

LinkedList.java

```
public class LinkedList<T> implements Iterable<T> {

   private Node<T> head;
   private Node<T> tail;
```

```
private int size;

public LinkedList() {
    this.head = this.tail = null;
    this.size = 0;
}

public void add(T data) {
    Node<T> newNode = new Node<>(data);
    if(head == null) {
        head = newNode;
        tail = newNode;
    } else {
        tail.setNext(newNode);
        tail = newNode;
    }
    size++;
}

public Node<T> getFirst() {
    return head;
}

public Node<T> getLast() {
    return tail;
}

public boolean remove() {
    if (head != null) {
        Node<T> current = head;

        while (current.getNext() != tail) {
            current = current.getNext();
        }

        tail = current;
        size--;
        return true;
    }
    return false;
}


public int size() {
    return size;
}
```

```
    @Override
    public Iterator<T> iterator() {
        return new LinkedListIterator<T>(this);
    }
}
```

LinkedListIterator.java

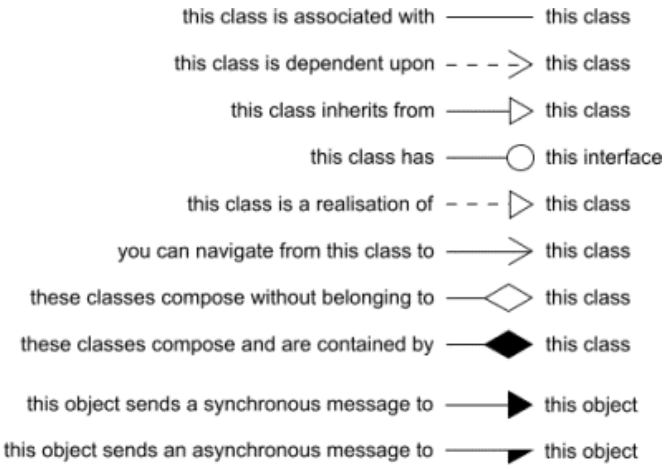
```
public class LinkedListIterator<T> implements Iterator<T> {

    private Node<T> current;
    private Node<T> previous;

    public LinkedListIterator(LinkedList<T> linkedList) {
        this.current = linkedList.getFirst();
        this.previous = null;
    }

    @Override
    public boolean hasNext() {
        return current != null;
    }

    @Override
    public T next() {
        if(!hasNext()) {
            return null;
        }

        T data = current.getData();
        this.previous = current;
        this.current = current.getNext();
        return data;
    }

    @Override
    public void remove() {
        if(current != null){
            if(previous != null) {
                previous.setNext(current.getNext());
                current = current.getNext();
            } else {
                current = current.getNext();
```

```
            }
        }
    }
}
```

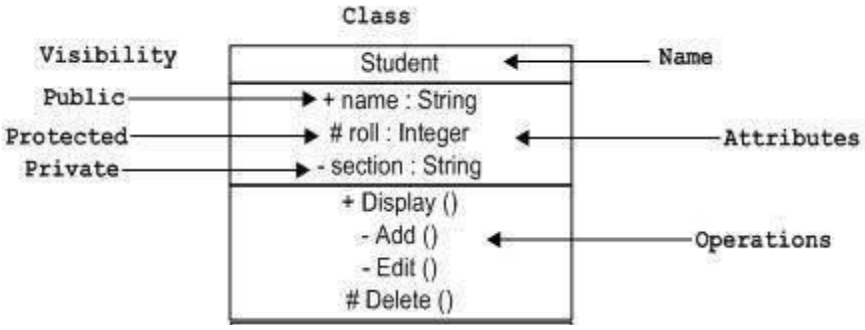
דיאגרמת מחלקות - UML

תרשים סטטי המתאר את מבנה המערכת על ידי הצגת מחלוקתיה, תכונותיהן והקשרים בין המחלקות.

משמעות החצים בדיאגרמות



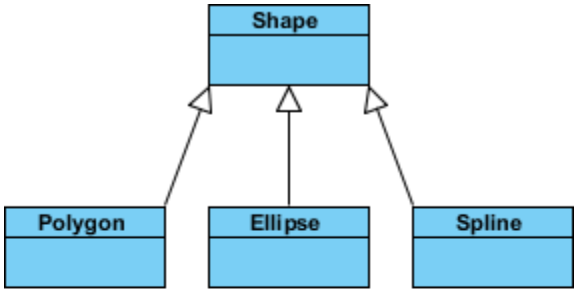
צורת מחלקה



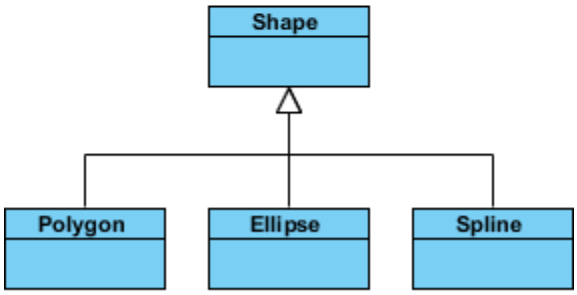
סיווגי גישה למחלקה

public	+	anywhere in the program and may be called by any object within the system
private	-	the class that defines it
protected	#	(a) the class that defines it or (b) a subclass of that class

דוגמה לדיאגרמת מחלקות ירושה

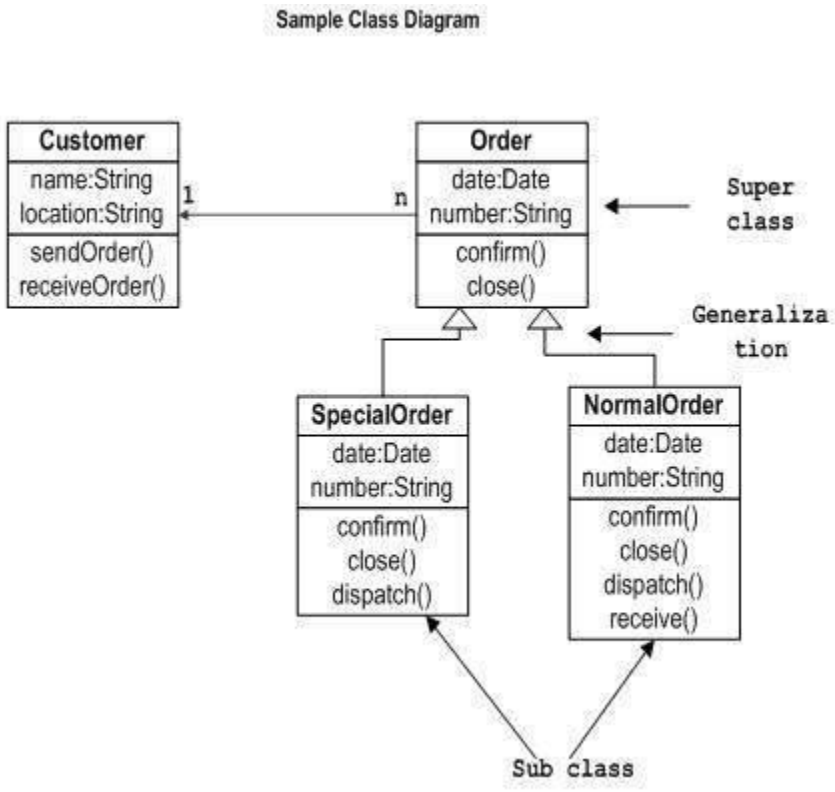


Style 1: Separate target



Style 2: Shared target

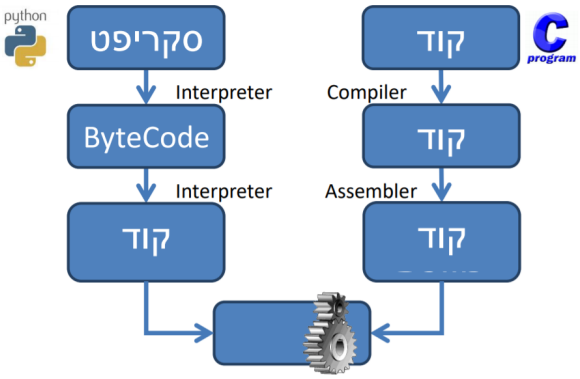
דוגמה כללית


Sample Class Diagram
Customer
name:String
location:String
sendOrder()
receiveOrder()
1
n
Order
date:Date
number:String
confirm()
close()
Super class
Generalization
SpecialOrder
date:Date
number:String
confirm()
close()
dispatch()
NormalOrder
date:Date
number:String
confirm()
close()
dispatch()
receive()
Sub class

מה זה Python

פייתון היא שפת תכנות ברמה גבוהה. פייתון היא שפת תכנות מונחית עצמים שיש לה תמיכה עצומה בספריות והופכת מימוש של תוכניות ואלגוריתמים לקלים יותר.

שפת פייתון פותחה **כשפת סקריפטים.** כדי להבין מהי שפת סקריפטים נצטרך להבין קודם כל כיצד עובדת שפה שאינה שפת סקריפטים, לדוגמה שפת C. כל שפת תוכנה צריכה להפוך בדרך כלשהי לשפת מכונה, כדי שהמעבד של המחשב יוכל להריץ אותה. **ההבדל** בין שפת סקריפטים לשפה שאיננה שפת סקריפטים הוא במסלול שעובר הקוד עד שהוא הופך לשפת מכונה. קוד שנכתב בשפת C צריך לעבור שני שלבים לפני שהמעבד של המחשב יכול להריץ אותו: **השלב הראשון** נקרא קומפילציה והוא מבוצע על ידי תוכנה שנקראת קומפיילר. הקומפיילר ממיר את הקוד משפת C לשפת אסמבלי . **השלב השני** מבוצע על ידי תוכנה שנקראת אסמבלר. האסמבלר ממיר את הקוד משפת אסמבלי לשפת מכונה.



שפת סקריפטים (פייתון) לעומת שפת C

על סקריפט פייתון פועלת תוכנה שנקראת **interpreter "פרשן".**

ה-interpreter עובד בצורה אחרת לגמרי מאשר הקומפיילר והאסמבלר בהם משתמשים כדי לתרגם את שפת C לשפת מכונה. הוא אינו יוצר קובץ אסמבלי וגם אינו יוצר קובץ הרצה. במקום זאת, כל פקודה שכתבנו בשפת פייתון מתורגמת לשפת מכונה רק בזמן הריצה. תוך כדי תהליך הפירוש, נוצר קובץ עם סיומת pycשמכיל bytecode – הוראות שונות של ה-interpreter – אך כאמור **זה אינו קובץ בשפת מכונה**, כלומר, מעבד לא מסוגל להריץ את הקובץ הזה.

מה אפשר להסיק ממה שלמדנו? קודם כל, ששפת פייתון היא הרבה יותר "סלחנית" לשגיאות מאשר שפות אחרות. לכן, הדעה הרווחת היא שקל יותר ללמוד לכתוב קוד בשפת פייתון. עם זאת , גם לשפת C יש יתרונות על פייתון: **ראשית**, אם נכתוב קוד לא זהיר בשפת פייתון הוא יתרסק תוך כדי ריצה . **אין** מנגנון כמו הקומפיילר של C ,שמונע מאיתנו לכתוב קוד שלא עומד בכללי התחביר של השפה ולכן הסיכוי לבעיות בזמן ריצה הוא קטן יותר. התרסקות של קוד תוך כדי ריצה **היא חמורה** לאין שיעור מאשר שגיאת קומפילציה, אותה ניתן לגלות ולדבג לפני ההרצה. **שנית**, העובדה שקוד בשפת C מתורגם לשפת מכונה לא שורה אחר שורה אלא כקובץ אחד, מאפשרת לבצע תהליכי ייעול (אופטימיזציה) של הקוד, כך שהוא עשוי **לרוץ יותר מהר** ולצרוך פחות זיכרון **מאשר** קוד מקביל בשפת פייתון. זה דבר חשוב למי שרוצים להריץ אפליקציות "כבדות", כגון גרפיקה מורכבת או הצפנה.

הבדלים חשובים בין פייתון לג'אווה		
	פייתון	**ג'אווה**
קוד	**פחות** שורות קוד	**יותר** שורות קוד
Framework	בהשוואה לג'אווה, לפייתון יש **פחות** Frameworks, כלומר, הוא פחות מבוסס על חלקים שונים שנבנים אחד על השני (היררכיה) כדי ליצור מערכת אחת מגובשת. (כמו Collections בג'אווה).	**הרבה יותר**.
סינטקס	**קל** לזכור, מאוד דומה לשפה אנושית.	**מורכב**, זורק שגיאה על כל עניין קטן בין אם פספסת ';' או '}'.
תכונות עיקריות	פחות שורת קוד, תכנות מהיר והקלדה דינמית.	ניהול זיכרון עצמי, חזק, בלתי תלוי בפלטפורמה אחרת.
מהירות	פייתון **איטי יותר** מכיוון שהוא משתמש במתורגמן (**interpreter**) וקובע גם את סוג הנתונים (הטיפוסים) ממש בזמן הריצה.	ג'אווה **מהירה יותר** בהשוואה לפיתון, יש לג'אווה קומפיילר לעומת פייתון שאין לו, הוא משתמש במתורגמן.
טיפוסים	מוקלד דינמי, כלומר, אנחנו לא **מחוייבים** לציין את הטיפוס לפני כל משתנה.	מוקלד באופן סטטי, כלומר אנחנו **מחוייבים** לציין את הטיפוס לפני כל משתנה.
הרצה	פייתון היא שפה **ללא קומפיילר**, יש לה Interpreter.	ג'אווה היא שפת קימפול וגם Interpreter ולכן היא מהירה יותר מפייתון.
בלוק	פייתון משתמש ב-Tab כדי להפריד קוד לבלוקים נפרדים	שפת ג'אווה משתמשת בסוגריים מתולתלים כדי להגדיר את ההתחלה והסוף של כל פונקציה והגדרת מחלקה
ממשקים וירושות	פייתון תומך בירושות בודדות ומרובות ולא תומך בממשקים (אין לו interface).	ללא ירושות מרובות, כלומר לא יכולה להרחיב יותר ממחלקה אחת.
ניידות ונוחות הרצה	תוכניות Python זקוקות להתקן של ה-**interpreter** במחשב כדי לתרגם קוד Python. פייתון פחות נייד.	תוכנית Java יכולה לפעול בכל מחשב או מכשיר נייד המסוגלים להריץ את המכונה הווירטואלית של של ג'אווה - JVM.

משתנים וטיפוסים בסיסיים

שמות המשתנים מכילים אותיות מספרים וקו-תחתון.
הם יכולים להתחיל עם תו או עם קו-תחתון אבל לא יכולות להתחיל עם מספר.
למשל אתה יכול לקרוא למשתנה בשם $planet_1$ולא בשם 1_planet.

מחרוזות

בפייתון אפשר להשתמש ב-Strings בצורה הבאה: "hi" או עם 'hi'.
גמישות זו מאפשרת לך להשתמש במרכאות במחרוזות שלך:

```
'I told my friend, "Python is my favorite language!"'
"The language 'Python' is named after Monty Python, not the snake."
"One of Python's strengths is its diverse and supportive community."
```

כאשר נרצה להדפיס מחרוזת נשתמש בפונקציה ***print()*** למשל:

```
name = "planet mars"
print(name)
```

נקבל את הפלט:

```
planet mars
```

הפונקציה ***title()*** תדפיס לנו כותרת רשמית עבור כל מחרוזת:

```
name = "planet mars"
print(name.title())
```

כלומר נקבל את הפלט הבא:

```
Planet Mars
```

במקום $planet\ mars$.

הפונקציות ***upper()*** ו-***lower()*** ידפיסו לנו את המחרוזת באותיות גדולות או קטנות:

```
name = "planet mars"
print(name.upper())
print(name.lower())
```

פלט:

```
PLANET MARS
```

```
planet mars
```

במצבים מסוימים, תרצה להשתמש בערך של משתנה בתוך מחרוזת.
לדוגמה, ייתכן שתרצה שני משתנים מייצגים שם פרטי ושם משפחה בהתאמה, ואז רוצה לשלב ערכים אלה כדי להציג את שמו המלא של מישהו:

```
first_name = "Michael"
last_name = "Jackson"
full_name = f"{first_name} {last_name}"
print(full_name)
```

פלט:

```
Michael Jackson
```

כדי להכניס משתנה לתוך מחרוזת, יש להציב את האות 'f' ממש לפני פתיחת המרכאות.
על כל משנה במחרוזת יש לשים {}.

מספרים

עבור מספרים יש את האריתמטיקה המוכרת: +,-,\,* וכו'...

```
a = 5
b = 5
print(a+b)
print(a-b)
print(a/b)
print(a*b)
```

נקבל את הפלט הבא:

```
10
0
1.0
25
```

בפייתון נכתוב חזקה כך: 2**2 ונקבל את הפלט 4 במידה ונדפיס.

רשימות

בפייתון, הסימון [ ] המציין רשימה ובה אלמנטים המופרדים בפסיק (,).

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
print(planets)
```

הפלט:

```
['Earth', 'Mars', 'Venus', 'Jupiter']
```

כדי לגשת לתאים במערך נציין כך: $print(planets[0])$ונקבל את הפלט Earth.
כמובן שגם כאן תקף פונקציות המחרוזות כמו למשל $print(planets[0].upper())$ נקבל EARTH.
בהתבקש לגשת לערך באינדקס 1-, פייתון תמיד תחזיר את הערך האחרון ברשימה.
כלומר במקרה שלנו עבור: $print(planets[-1])$ נקבל את הפלט Jupiter.

הפונקציה ***append()*** תוסיף לנו ערכים חדשים לסוף הרשימה, למשל:

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
planets.append('Neptune')
print(planets)
```

נקבל את הפלט:

```
['Earth', 'Mars', 'Venus', 'Jupiter', 'Neptune']
```

אפשר גם להוסיף אלמנט לרשימה בכל מיקום שתרצה בעזרת הפונקציה ***insert()***, למשל:

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
planets.insert(0,'Neptune')
print(planets)
planets.insert(3,'Saturn')
print(planets)
```

נקבל את הפלט הבא:

```
['Neptune', 'Earth', 'Mars', 'Venus', 'Jupiter']
['Neptune', 'Earth', 'Mars', 'Saturn', 'Venus', 'Jupiter']
```

אם אתה יודע את המיקום של הפריט שאתה רוצה להסיר ברשימה, אתה יכול להשתמש בהצהרת ***del***.

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
del planets[3]
```

נקבל את הפלט:

```
['Earth', 'Mars', 'Venus']
```

הפונקציה ***pop()*** תמחק ותחזיר את האיבר במקום האחרון ברשימה.

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
popped = planets.pop()
print(planets)
print(f"{popped} has popped from the list")
```

הפלט:

```
['Earth', 'Mars', 'Venus']
Jupiter has popped from the list
```

בנוסף אפשר להשתמש ב-pop לכל מקום ברשימה, למשל:

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
popped = planets.pop(1)
print(planets)
print(f"{popped} has popped from the list")
```

הפלט:

```
['Earth', 'Venus', 'Jupiter']
Mars has popped from the list
```

לפעמים אנחנו לא יודעים את האינדקס של הערך אבל אנחנו יודעים מהו הערך, לכן נשתמש במקרה זה בפונקציה ***remove()***. למשל:

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
planets.remove('Venus')
print(planets)
```

הפלט:

```
['Earth', 'Mars', 'Jupiter']
```

הפונקציה ***sort()*** תמיין לי את הרשימה בסדר אלפבתי :

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
planets.sort()
print(planets)
```

הפלט:

```
['Earth', 'Jupiter', 'Mars', 'Venus']
```

באפשרותינו גם למיין את הרשימה בסדר הפוך בצורה ***reverse=True*** כך:

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
planets.sort(reverse=True)
print(planets)
```

הפלט:

```
['Venus', 'Mars', 'Jupiter', 'Earth']
```

הפונקציה ***sorted()*** תציג לך את הרשימה ממוינת אבל לא באמת תשנה אותה, למשל:

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
print(sorted(planets))
print(planets)
```

הפלט:

```
['Earth', 'Jupiter', 'Mars', 'Venus']
['Earth', 'Mars', 'Venus', 'Jupiter']
```

הפונקציה ***reverse()*** תהפוך את הסדר של הרשימה (לא מיון הפוך) באופן הבא:

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
print(planets)

planets.reverse()
print(planets)
```

הפלט:

```
['Earth', 'Mars', 'Venus', 'Jupiter']
['Jupiter', 'Venus', 'Mars', 'Earth']
```

הפונקציה ***len()*** תחזיר לנו את אורך הרשימה שלנו, למשל:

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
print(f"the length of this list is: {len(planets)}")
```

הפלט:

```
the length of this list is: 4
```

פעולות עם רשימות

על כל לולאה צריך לציין 'for', שם יחידון ואחריו 'in' המתייחס לרשימה כלשהי,
בנוסף בפייתון בשונה משפות תכנות אחרות, ה-"בלוק" לא מוגדר כך {} אלא לפי tabs, אפשר לראות בדוגמה הבאה שפונקצית ה-print שלנו נמצא בתוך הבלוק של הלולאה:

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']

for planet in planets:
   print(planet)
```

פלט:

```
Earth
Mars
Venus
Jupiter
```

הפונקציה ***range()*** תציג לנו סדרת מספרים למשל:

```
for number in range(1,5):
   print(number)
```

פלט:

```
1
2
3
4
```

ניתן גם ליצור סדרה ולשמור אותה בתור משתנה של סדרה בצורה הבאה:

```
numbers = list(range(1,6))
print(numbers)
```

פלט:

```
[1, 2, 3, 4, 5]
```

נראה דוגמא לרשימות ולולאות:

```
squares = []
for value in range(1, 11):
   square = value ** 2
   squares.append(square)
print(squares)
```

נקבל את הפלט:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

סטטיסטיקות פשוטות על רשימות מספרים עם הפונקציות ***sum() ,max() ,min :***

```
digits = [1,2,3,4,5,6,7,8,9]
print(min(digits))
print(max(digits))
print(sum(digits))
```

נקבל את הפלט:

```
1
9
45
```

ניתן גם להכריז על רשימה חדשה ובמקביל כבר לאחסן אותה בנתונים שנרצה באופן הבא:

```
squares = [value ** 2 for value in range(1, 11)]
print(squares)
```

פלט:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

כדי לעבוד רק עם **חלק** מרשימה נעבוד באופן הבא:

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
print(planets)
print(planets[1:3])
```

פלט:

```
['Earth', 'Mars', 'Venus', 'Jupiter']
['Mars', 'Venus']
```

נשים לב שאינדקס 3 לא כלול כאן, כלומר האינדקס האחרון לא שייך לחיתוך.

ניתן גם לכתוב בצורה הזאת כלומר מאינדקס 0 עד 2 (לא כולל):

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
print(planets)
print(planets[:2])
```

פלט:

```
['Earth', 'Mars', 'Venus', 'Jupiter']
['Earth', 'Mars']
```

ובצורה הבאה נעבוד עם רשימה חלקית כך שאנו משתמשים מהערך האחרון ברשימה ועד לאינדקס שהגדרנו (כולל האינדקס הזה):

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
print(planets)
print(planets[2:])
```

פלט:

```
['Earth', 'Mars', 'Venus', 'Jupiter']
['Venus', 'Jupiter']
```

כדי לבצע **העתקה עמוקה** של רשימות נציין את הסימון [:] עבור רשימה חדשה באופן הבא:

```
planets = ['Earth', 'Mars', 'Venus', 'Jupiter']
otherPlanets = planets[:]
print(planets)
print(otherPlanets)

planets.append('Neptune')
otherPlanets.append('Uranus')
print(planets)
print(otherPlanets)
```

פלט:

```
['Earth', 'Mars', 'Venus', 'Jupiter']
['Earth', 'Mars', 'Venus', 'Jupiter']
['Earth', 'Mars', 'Venus', 'Jupiter', 'Neptune']
['Earth', 'Mars', 'Venus', 'Jupiter', 'Uranus']
```

Tuples/טאפלים

בשונה מרשימה, את הטאפל אנחנו נגדיר עם סוגריים עגולים ().
היכולת לשנות רשימות חשובה במיוחד כאשר אתה עובד עם רשימת משתמשים באתר או רשימת דמויות במשחק. עם זאת, לפעמים תרצה ליצור רשימה של פריטים שלא יכולים להשתנות.
Tuples מאפשרים לך לעשות בדיוק את זה. פייתון מתייחס לערכים שאינם יכולים להשתנות כ-immutable, ורשימה שאינה ניתנת לשינוי נקראת ***Tuple***.

לדוגמא, אם יש לנו מלבן שתמיד צריך להיות בגודל מסוים, אנו יכולים להבטיח שגודלו לא ישתנה על ידי הכנסת המידות לטאפל:

```
dimensions = (200, 50)
print(dimensions[0])
print(dimensions[1])
```

פלט:

```
200
50
```

בואו נראה מה יקרה אם נשנה ערך בטאפל:

```
dimensions = (200, 50)
dimensions[0] = 250
```

פלט שגיאה:

```
TypeError: 'tuple' object does not support item assignment
```

כמובן שנוכל "לדרוס" טאפל עם טאפל חדש, כלומר:

```
stars = ('Earch','Mars')
print("Original stars")
for star in stars:
   print(star)

stars = ('Venus','Jupiter')
print("New stars")
for star in stars:
   print(star)
```

פלט:

```
Original stars
Earch
Mars
New stars
Venus
Jupiter
```

תנאים ואופרטורים

השוואה	
a==b	a <= b
a != b	a > b
a < b	a >= b

פעולות לוגיות		
$x < 5 \; and \; x < 10$	מחזיר True אם התנאי נכון	and
$x < 5 \; or \; x < 4$	מחזיר True אם אחד התנאים נכון	or
$not(x < 5 \; and \; x < 10)$	מחזיר את שלילת הפסוק	not

פעולת זהות		
$x \; is \; y$	מחזיר True אם המשתנים הם מאותו סוג אובייקט	is
$x \; is \; not \; y$	מחזיר True שני המשתנים לא מאותו סוג אובייקט	is not

פעולת הכלה		
$x \; in \; y$	מחזיר True אם ערך ספציפי נמצא בסדרת אובייקט כלשהו	in
$x \; not \; in \; y$	מחזיר True אם ערך ספציפי לא נמצא בסדרת אובייקט	not in

למשל:

```
companies = ['nasa', 'space x']
for company in companies:
   if company == 'nasa':
       print(company.upper())
   else:
       print(company.title())
```

פלט:

```
NASA
Space X
```

בפייתון התנאי else-if מיוצג בצורה ***elif***, למשל:

```
age = 25
if age < 4 :
   print("what a baby!")
elif age < 18 :
   print("you are still young!")
else:
   print("watch your hair!")
```

פלט:

```
watch your hair!
```

כדי לבדוק אם רשימה ריקה (בדומה לפונקציה המוכרת ***isEmpty*** בשפות אחרות), ניתן פשוט לכתוב את שם משתנה של הרשימה בתנאי ה-if, למשל:

```
companies = []

if companies:
   for company in companies:
       if company == 'nasa':
           print(company.upper())
       else:
           print(company.title())
else:
   print("the list is empty!")
```

פלט:

```
the list is empty!
```

Dictionaries/מילונים

מילון נגדיר עם {} כאשר כל זוג מופרד ב- : (נקודותיים) משמאל זה המפתח ומימין זה הערך.
נעבוד על משחק המציג כוכבי לכת, לכל כוכב צבע ומספר ירחים.
נתבונן במילון הפשוט הבא:

```
earth = {'color': 'blue', 'moons': 1}
print(earth['color'])
print(earth['moons'])
```

פלט:

```
blue
1
```

ניתן גם להוסיף למילון קיים בצורה הבאה:

```
earth = {'color': 'blue', 'moons': 1}
print(earth)
earth['number of people'] = '7 Billion'
earth['number of days'] = 365
print(earth)
```

פלט:

```
{color': 'blue', 'moons': 1'}
{color': 'blue', 'moons': 1, 'number of people': '7 Billion', 'number of days': 365'}
```

ניתן גם לאתחל מילון ריק ואחר כך להוסיף אליו זוגות מתי שנרצה וכמה שנרצה:

```
earth = {}
earth['number of people'] = '7 Billion'
earth['number of days'] = 365
print(earth)
```

פלט:

```
{'number of people': '7 Billion', 'number of days': 365}
```

אם נרצה למחוק זוג מהמילון שלנו נשתמש ***ב-del*** למשל:

```
earth = {'color': 'blue', 'moons': 1}
print(earth)
del earth['moons']
print(earth)
```

פלט:

```
{color': 'blue', 'moons': 1'}
{'color': 'blue'}
```

שימוש ב-keys בסוגריים מרובעים כדי לקבל את הערך שאתה מעוניין במילון עלול לגרום לבעיה אפשרית אחת: אם המפתח שאתה מבקש אינו קיים, תקבל שגיאה.
במילונים, באופן ספציפי, תוכלו להשתמש בשיטת ***get()*** כדי להגדיר ערך ברירת מחדל שיוחזר אם המפתח המבוקש לא קיים.

שיטת ***get()*** דורשת מפתח בארגומנט הראשון. בארגומנט האופציונלי השני, אתה יכול להעביר את הערך שיוחזר אם המפתח לא קיים:

```
earth = {'color': 'blue', 'moons': 1}
points = earth.get('points','No points value assigned.')
print(points)
```

פלט:

```
No points value assigned.
```

מעבר שלם על מילון באמצעות לולאות for :

```
earth = {
   'color': 'blue',
   'moons': 1,
   'population': '7 billion'
}

for key, value in earth.items():
   print(f"\nKey: {key}")
   print(f"Value: {value}")
```

פלט:

```
Key: color
Value: blue

Key: moons
Value: 1

Key: population
Value: 7 billion
```

הדפסת כל המפתחות במילון:

```
earth = {
   'color': 'blue',
   'moons': 1,
   'population': '7 billion'
}
for key in earth.keys():
   print(f"Key: {key}")
```

פלט:

```
Key: color
Key: moons
Key: population
```

כדי לעבור על כל הערכים במילון נבצע אותו מימוש רק שהפעם נכתוב ()earth.values.

כשנרצה לממש מילון בתוך מילון נוסף, נפעל בצורה הבאה:

```
planets = {
   'earth': {'color': 'blue', 'moons': 1, 'population': '7 billion'},
   'mars': {'color': 'red', 'moons': 2, 'population': '0'}
}

for stars, star_info in planets.items():
   print(f"planet: {stars.title()}")
   color = star_info['color']
   moons = star_info['moons']
   population = star_info['population']
   print(f"color: {color}, moons: {moons}, population: {population}\n")
```

פלט:

```
planet: Earth
color: blue, moons: 1, population: 7 billion

planet: Mars
color: red, moons: 2, population: 0
```

מחלקות

יצירת מחלקה

```
class User:
```

יצירת מופע (instance) / עצם (object)

```
user1 = User()
```

הגדרת פעולת `__str__` עבור מחלקה מסוימת מאפשרת לנו להמיר מופעים למחרוזות בצורה טבעית.

הפעולה `__init__` היא הבנאי של המחלקה

מחלקה (Class)

תבנית, או שבלונה, שמתארת אוסף של תכונות ופעולות שיש ביניהן קשר.

המחלקה מגדירה מבנה שבעזרתו נוכל ליצור בקלות עצם מוגדר, שוב ושוב.

לדוגמה: מחלקה המתארת משתמש ברשת חברתית, מחלקה המתארת כלי רכב, מחלקה המתארת נקודה במישור.

מופע (Instance)

נקרא גם עצם (Object).

ערך שנוצר על ידי מחלקה כלשהי. סוג הערך ייקבע לפי המחלקה שיצרה אותו.

הערך נוצר לפי התבנית ("השבלונה") של המחלקה שממנה הוא נוצר, ומוצמדות לו הפעולות שהוגדרו במחלקה. המופע הוא יחידה עצמאית שעומדת בפני עצמה. לרוב מחלקה תשמש אותנו ליצירת מופעים רבים.

לדוגמה: המופע "נקודה שנמצאת ב־(5, 3)" יהיה מופע שנוצר מהמחלקה "נקודה".

תכונה (Property, Member)

ערך אופייני למופע שנוצר מהמחלקה.

משתנים השייכים למופע שנוצר מהמחלקה, ומכילים ערכים שמתארים אותו.

לדוגמה: לנקודה במישור יש ערך x וערך y. אלו 2 תכונות של הנקודה.

נוכל להחליט שתכונותיה של מחלקת מכונית יהיו צבע, דגם ויצרן.

פעולה (Method)

פונקציה שמוגדרת בגוף המחלקה.

מתארת התנהגויות אפשריות של המופע שייווצר מהמחלקה.

לדוגמה: פעולה על נקודה במישור יכולה להיות מציאת מרחקה מראשית הצירים.

פעולה על שולחן יכולה להיות "קצץ 5 סנטימטר מגובהו".

שדה (Field, Attribute)

שם כללי הנועד לתאר תכונה או פעולה.

שדות של מופע מסוים יהיו כלל התכונות והפעולות שאפשר לגשת אליהן מאותו מופע.

לדוגמה: השדות של נקודה יהיו התכונות x ו־y, והפעולה שבודקת את מרחקה מראשית הצירים.

פעולה מיוחדת (Special Method)

ידועה גם כ־dunder method (double under, קו תחתון כפול) או כ־magic method (פעולת קסם).

פעולה שהגדרתה במחלקה גורמת למחלקה או למופעים הנוצרים ממנה להתנהגות מיוחדת.

דוגמאות לפעולות שכאלו הן `__init__` ו־`__str__`.

פעולת אתחול (Initialization Method)

פעולה שרצה עם יצירת מופע חדש מתוך מחלקה.

לרוב משתמשים בפעולה זו כדי להזין במופע ערכים התחלתיים.

משתני מחלקה (Class Variables)

משתנים המוגדרים ברמת המחלקה ונגישים עבור כל המופעים שנוצרו ממנה.

בדרך כלל אלו משתנים קבועים שהמופעים משתמשים בהם לצורכי קריאה בלבד.

לדוגמה: משתנה בראש מחלקת "תיבת דואר" שמגדיר את נפח האחסון המירבי ל־5 ג'יגה בייט.

הכלה (Containment)

מצב שבו נעשה שימוש במופע שנוצר במחלקה A בתוך מופע של מחלקה B.

לדוגמה: בתוך מופע של מחלקת "הודעת דואר", תכונת הנמען והמוען יהיו מופעים של מחלקת "משתמש".

כימוס (Encapsulation)

איגוד תכונות ופעולות תחת מחלקה, וצמצום הגישה של המשתמש במחלקה למצב הפנימי של המופעים שנוצרו ממנה. בפועל, ימומש על ידי הגבלה לגישה ולעריכה של תכונות ופעולות מסוימות, כך שיתאפשרו רק מקוד שנכתב בתוך המחלקה.

עם זאת, החלטה על הסתרת תכונות רבות מדי עלולה ליצור קוד ארוך ומסורבל, ובפייתון נהוג להשתמש ברעיון בצמצום.

תכונה/פעולה מוגנת (Protected Attribute) או תכונה/פעולה פרטית (Private Attribute)

תכונה שהגישה אליה יכולה להתבצע רק מתוך המחלקה.

טכנית, פייתון תמיד מאפשרת למתכנת לשנות ערכים – אפילו אם הם מוגדרים כמוגנים או כפרטיים.

מעשית, עליכם להימנע בכל דרך אפשרית משינוי של משתנה שמוגדר כמוגן או פרטי, כל עוד אתם יכולים.

פעולת גישה ושינוי (Accessor/Mutator Method)

פעולה שמטרתה לגשת (Accessor) או לשנות (Mutator) ערך של תכונה מסוימת, בעיקר בהקשרי תכונות מוגנות או פרטיות.

מטרת ה־Accessor היא לאחזר את ערך התכונה המבוקשת בצורה מתאימה, לעיתים אחרי עיבוד מסוים.

מטרת ה־Mutator היא לוודא ששינוי הערך תקין ולא מזיק למופע או משנה אותו למצב לא תקין.

פעולות אלו נקראות גם getters ו־setters.

ירושה ואבסטרקטיות

הרעיון של מחלקה מופשטת כה נפוץ, שהמודול הפייתוני abc (קיצור של Abstract Base Class) מאפשר לנו להגדיר מחלקה כמופשטת.

- המחלקה המופשטת שלנו תבצע ירושה מהמחלקה ABC שבמודול abc.
- מעל כל פעולה מופשטת (כזו שמשמשת רק לירושה ולא עומדת בפני עצמה) נוסיף `@abstractmethod`.

מחלקה מופשטת (Abstract Class)

מחלקה שלא נהוג ליצור ממנה מופעים.

המחלקה המופשטת תגדיר פעולות ללא מימוש שנקראות "פעולות מופשטות".

המחלקות שיורשות מהמחלקה המופשטת יממשו את הפעולות הללו.

ירושה מרובה (Multiple Inheritance)

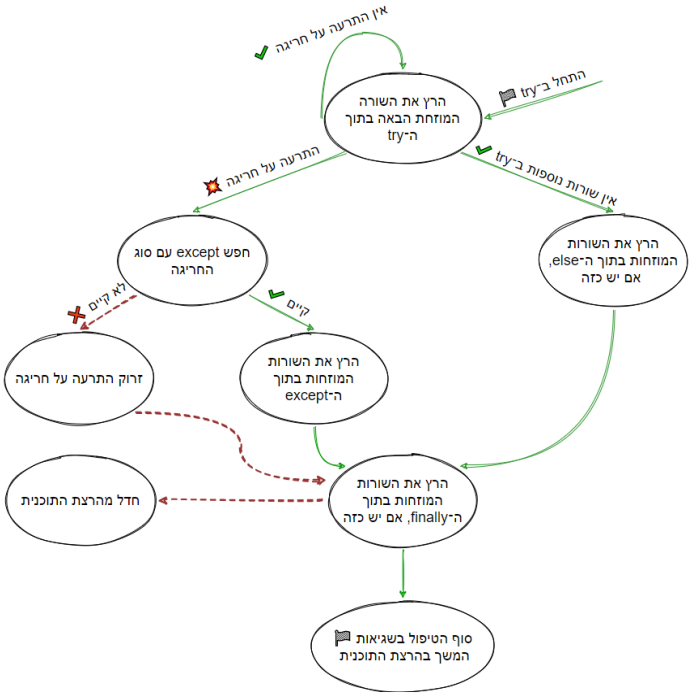
ירושה שמחלקה מבצעת ממחלקות רבות, במטרה לקבל את תכונותיהן ופעולותיהן של כמה מחלקות־על.

חריגות

התחביר של except ... try הוא:

1. נתחיל עם שורה שבה כתוב אך ורק try:.
2. בהזחה, נכתוב את כל מה שאנחנו רוצים לנסות לבצע ועלול לגרום להתראה על חריגה.
3. בשורה הבאה, נצא מההזחה ונכתוב except ExceptionType:, כאשר ExceptionType הוא סוג החריגה שנרצה לתפוס.
4. בהזחה (שוב), נכתוב קוד שנרצה לבצע אם פייתון התריעה על חריגה מסוג ExceptionType בזמן שהקוד המוזח תחת ה־try רץ.

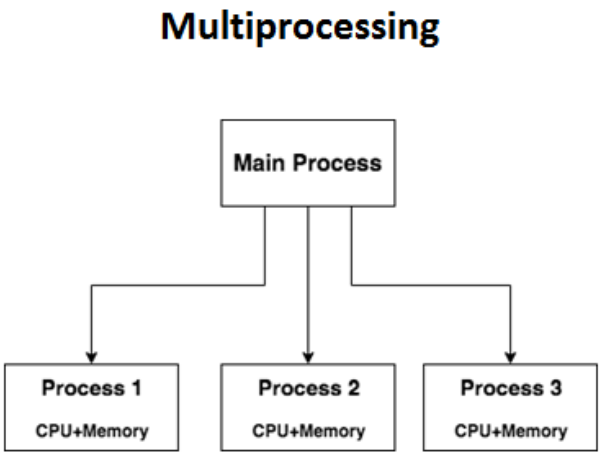
```
def get_file_content(filepath):
try:                                   נסה לבצע...
    with open(filepath) as file_handler:   קוד שעשוי לגרום להתרעה על חריגה
        return file_handler.read()
except FileNotFoundError:              אם הייתה התרעה על חריגה מסוג FileNotFoundError
    print(f"Couldn't open the file: {filepath}.")   בצע במקום זאת את הפעולות הבאות
    return ""
```

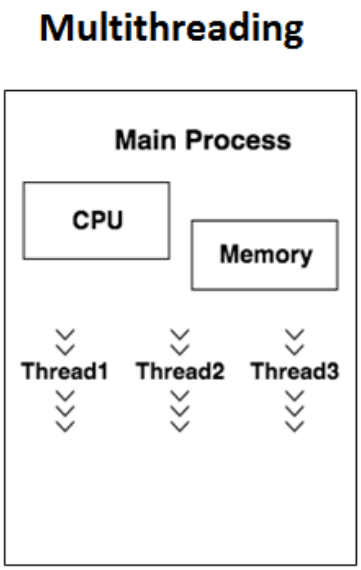


תהליכים ופרוססים - Threads & Processes

Global Interpreter Lock - GIL : נעילת ה-interpreter הגלובלית של Python היא נעילה המאפשרת רק ל-Thread אחד להחזיק את השליטה של מתורגמן (ה-interpreter) בפייתון.

פייתון לא תומך בעבודה עם ליבות מרובות. כלומר כל התהליכים יעבדו על ליבה אחת.
זה כי בפייתון יש GIL המתרגם (ה-interpreter) לא תומך בעבודה עם יותר ממעבד אחד.
כדי "לעקוף" את זה ולעבוד עם כמה ליבות, נעבוד עם פרוססים.





Multithreading היא טכניקה שבה מספר רב של threads נוצרים על ידי תהליך לביצוע משימות שונות, באותו הזמן בערך, בזה אחר זה. זה נותן לך אשליה שה-threads פועלים במקביל, אבל הם למעשה מתנהלים בצורה בו זמנית. בפייתון, נעילת המתורגמן הגלובלית (GIL) מונעת את הפעלת ה-thread-ים בו זמנית.
Threads יש להם זיכרון משותף ו-CPU משותף אחד ויחיד.

Multiprocessing היא טכניקה בה מושגת מקבילות בצורתה האמיתית ביותר. תהליכים מרובים מנוהלים על פני מספר ליבות מעבד, שאינם חולקים את המשאבים ביניהם. לכל תהליך יכולות להיות threads רבים הפועלים במרחב הזיכרון שלו. בפייתון, לכל תהליך יש מופע משלו של ה-interpreter שעושה את העבודה של ביצוע ההוראות.
לכל פורסס יש CPU משלו וזיכרון משלו (תלוי כמה CPU יש במחשב שלך). - כלומר כל פרוסס עובד בנפרד.

מה ההבדל בין Multithreading ל-Multiprocessing ?

Threads יש להם זיכרון משותף ו-CPU משותף אחד ויחיד.
לכל פורסס יש CPU משלו וזיכרון משלו (תלוי כמה CPU יש במחשב שלך). - כלומר כל פרוסס עובד בנפרד.
לכל פורסס אני אתן interpreter משלו וככה בעצם "עקפנו" את ה-GIL.

כל פעם שנעבוד עם threads נשאל את עצמנו את השאלה, האם המצב הוא CPU bound או I/0 bound?

1. אם המצב הוא **I/O bound** אז **threads** יכול לטפל בבעיה, כי אם לדוגמה, אני טוען תמונות, אני לא צריך CPU נוסף מכיוון שהמעבד לא עושה שום דבר חוץ מלחכות כל פעם שהתמונה תטען.
2. אם המצב הוא **CPU bound** אז **process** לדוגמה, אם נעשה פעולות מורכבות מבחינה חישובית, נצתרך כמה ליבות. במילים אחרות, אם נרצה לעשות חישוב של מספרים ופעולות יותר חישוביות נשתמש בפרוססים ולא בthreads.

החיסרון בפרוסס לעומת thread הוא שכדי להפעיל פרוסס, זה פעולה מאוד **יקרה (כבדה),** כי צריך לפתוח לו intrepeter וזיכרון משלו, זה בעיה כי זה יכול להאט את התוכנית שלנו.

מימוש פשוט עבור thread בפייתון, נייבא את threading, התכתיב מאוד דומה לג'אווה.
במימוש יצרנו 6 תהליכים שנשלחים בכל פעם לשיטה heavy_func כאשר args מייצג את הארגומנט שנכנס לפונקציה, כלומר מתבצע sleep כל קריאה עבור 1 שניות, 2 שניות , ... 5 שניות.
נשים לב שגם כאן הפעלנו start ואת join בדומה לג'אווה.

```
import threading
import time

def heavy_func(sleep_time):  # simulates I/O
   print(f"heavy sleep time start: {sleep_time}")
   time.sleep(sleep_time)
   print(f"heavy sleep time end: {sleep_time}")

def simple_thread():
   threads = []

   for i in range(6):
       thread = threading.Thread(target=heavy_func, args=[i])
       thread.start()
       threads.append(thread)
   for t in threads:
       t.join()

if __name__ == '__main__':
   simple_thread()
```

Thread Pool in Python

למה כדאי לנו להשתמש ב-Thread Pool לעומת Thread רגיל?
אם Thread רגיל מסתיים, ייווצר Thread אחר שיחליף אותו, למה לא למחזר את הThread שלנו?
אם Thread Pool מסתיים, ניתן לעשות בו שימוש חוזר.

Executor הוא מחלקה מופשטת בפייתון. לא ניתן להשתמש בו ישירות ועלינו להשתמש באחת מתתי המחלקות הבאות שלו: **ThreadPoolExecutor** ו- **ProcessPoolExecutor**.

מימוש Thread עם **ThreadPoolExecutor :**

```
import time
import concurrent.futures

def heavy_func(seconds):
   print(f"heavy_func sleeping for {seconds}... ")
   time.sleep(seconds)
   return f"heavy_func Done sleeping for {seconds}..."


def thread_pool_example():
   with concurrent.futures.ThreadPoolExecutor() as pool:
       result = pool.map(heavy_func, range(6))  # active the func for 6 threads

       for r in result:
           print(r)

if __name__ == '__main__':
   thread_pool_example()
```

מימוש Process עם **ProcessPoolExecutor:**

```
import time
import concurrent.futures

def heavy_func(seconds):
   print(f"heavy_func sleeping for {seconds}... ")
   time.sleep(seconds)
   return f"heavy_func Done sleeping for {seconds}..."

def process_pool_example():
   with concurrent.futures.ProcessPoolExecutor() as pool:
       result = pool.map(heavy_func, range(6))  # active the func for 6 processes
```

```
        for r in result:
            print(r)

if __name__ == '__main__':
   process_pool_example()
```

אם נרצה **לסנכרן** בין threads נפעיל את Lock מתוך threading.

באופן דומה ל-wait ו-notify שב-Java, כאשר נשתמש ב-Lock אנחנו נגדיר איזה שלב בקוד לנעול ואיפה לשחרר את הנעילה. יש 2 שיטות לנעילה ושחרור בעזרת Lock:

שיטה ראשונה

```
lock.acquire() # locking the below part
shared_int += 1
lock.release() # releasing the above part
```

שיטה שניה

```
with lock:
   shared_int += 1
```

חשוב - 2 השיטות שקולות ומבצעות אותם פעולות.

מימוש לסנכרון תהליכים:

```
import threading

shared_int = 0
def inc_times(lock: threading.Lock, num: int):
   global shared_int

   for i in range(num):
       with lock:
           shared_int += 1

if __name__ == '__main__':
   lock = threading.Lock()
   t1 = threading.Thread(target=inc_times, args=[lock, 1000000])
   t2 = threading.Thread(target=inc_times, args=[lock, 1000000])
   t1.start()
   t2.start()
   t1.join()
   t2.join()
   print("shared_int: ", shared_int)
```

מה זה Git

מתוך המדריך

Git היא מערכת לניהול גרסאות. שימוש ב-Git מאפשר לנו לעבוד בצוות על אותו הפרויקט, לצפות בשינויים ולנהל גרסאות באופן חכם, כלי חובה לכל מפתח.

Git (גיט) היא מערכת חכמה **לניהול גרסאות קוד.**
Git הוא מעין מאגר העוקב אחר השינויים בקבצי הקוד של הפרויקט שלנו.
כל שינוי המבוצע מעודכן במאגר וביחד עם השינוי, גם תיאור קצר המפרט על אותו שינוי, וכמובן מי המפתח האחראי על השינוי. המונח המקצועי של כל שיגור שכזה הוא "**Commit**".
Git מאפשרת לנו לעבוד בצוות, כל אחד על חלקו ולשגר את השינויים למאגר אחד המכיל את כל השינויים של כולם, שמו המקצועי של המאגר הוא "**Repository**".
אז מדוע להשתמש ב-Git? ביום שהפרויקט שלכם ישתבש, תוכלו לחזור לגרסאות ישנות יותר במהירות ובפשטות עזרת ה-commits שעשיתם.
השימוש ב-Git יעזור לכם לעבוד כיחיד ובפרט כצוות באופן מסודר וטוב יותר.

יצירת Repository

Git מנהלת מעקב אחר הפרויקט שלכם. כל התהליך הזה מתרחש בתיקייה שבדרך כלל תהיה נסתרת ועם שם – git. על מנת ליצור תיקייה זו, בשמה המקצועי – Repository, עלינו "לכוון" את Git להסתכל על התיקייה שבה נרצה לנהל את ה-Repository שלנו.

ניתן לעשות זאת על ידי הפקודה cd:

```
C:\Users\dazao>cd Desktop/my_project
```

כעת, נוכל ליצור את ה-Repository שלנו על ידי הפקודה **git init**:

```
C:\Users\dazao\Desktop\my_project>git init
Initialized empty Git repository in C:/Users/dazao/Desktop/my_project/.git/
```

כפי שאתם רואים, מיד קיבלנו הודעה שיצרנו Repository בתיקיה my_project.

העלאת קבצי הפרויקט שלנו למאגר של Git בפעם הראשונה

כרגע יש לנו מאגר "ריק" שרק אתחלנו אותו על ידי הפקודה **git init** אבל הוא איננו "עוקב" אחר הפרויקט שלנו. על מנת שיעשה זאת, עלינו להוסיף את הקבצים.

לצורך הדוגמא, נאמר והפרויקט שלנו מכיל כרגע 4 קבצים של java.

באפשרותנו להבין מה קורה ב-Repository שלנו על ידי הפקודה **git status**:

```
C:\Users\dazao\Desktop\my_project>git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        CEO.java
        Manager.java
        Organization.java
        Worker.java

nothing added to commit but untracked files present (use "git add" to track)
```

מה בעצם אנחנו רואים כאן? Git מודיעה לנו שיש לנו קבצים שהיא "לא מכירה" ("Untracked files").

כדי להוסיף קובץ ספציפי נבצע את הפקודה **git add** ואז את שם הקובץ:

```
C:\Users\dazao\Desktop\my_project>git add Worker.java
```

מה נעשה כאשר הפרויקט שלנו יכיל מאות/אלפי קבצים? בטח שלא נרשום כל קובץ בנפרד?
פשוט נעשה git add ואחריו נכתוב ' . ' (הנקודה == הכל).

```
C:\Users\dazao\Desktop\my_project>git add .
```

אפשר לראות לפי הסטטוס כעת:

```
C:\Users\dazao\Desktop\my_project>git status
On branch master
No commits yet
Changes to be committed: (use "git rm --cached <file>..." to unstage)
        new file:   CEO.java
        new file:   Manager.java
        new file:   Organization.java
        new file:   Worker.java
```

כעת כל ארבעת הקבצים בתיקייה my_project מזוהים כ-"הועלו" **(בירוק)** אבל עדיין לא עשינו commit.

הקובץ gitignore.

סביר להניח כי ברוב הפרויקטים שלנו, ישנם קבצים/תיקיות שנרצה ש-Git לא תעקוב אחריהם והם יועלו ל-Repository שלנו. קובץ gitignore. הוא הפיתרון.

אם אנחנו מתחרטים על קובץ מסויים שעשינו עליו add נבצע את הפקודה git rm הבאה:

```
C:\Users\dazao\Desktop\my_project>git rm --cached Manager.java
rm 'Manager.java'
```

יצירת Commit

על מנת לבצע Commit, נשתמש בפקודה git commit -m ההערה בכחול זה תיאור הקומיט לבחירתינו:

```
C:\Users\dazao\Desktop\my_project>git commit -m "init commit"
[master (root-commit) 51eb25d] init commit
 4 files changed, 161 insertions(+)
 create mode 100644 CEO.java
 create mode 100644 Manager.java
 create mode 100644 Organization.java
 create mode 100644 Worker.java
```

אם נרצה לצפות בקומטים קודמים שעשינו נבצע את הפקודה הבאה:

```
C:\Users\dazao\Desktop\my_project>git log
commit 51eb25d4acbac9eec122fe99a461c98504d3ea5b (HEAD -> master)
Author: Dor Azaria <46644036+DorAzaria@users.noreply.github.com>
Date:   Sun Jan 31 16:42:45 2021 +0200
```

במידה ויהיו לנו המון קומיטים, סביר להניח שלא פעם נרצה לקבל רק מספר קומיטים מצומצם, למשל את ה-10 האחרונים: `git log -n 10`

מחיקה והעברת קבצים

על מנת למחוק קובץ נשתמש בפקודה git rm :

```
C:\Users\dazao\Desktop\my_project>git rm Worker.java
```

על מנת להעביר קובץ לתיקיה אחרת נשתמש בפקודה git mv:

```
C:\Users\dazao\Desktop\my_project>git mv Worker.java other_folder/Worker.java
```

מה זה GitHub

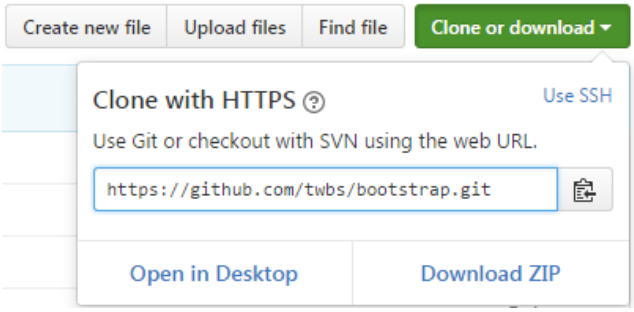
GitHub הוא מעין רשת חברתית של Git, הוא מאפשר לנו לשתף את הפרויקטים שלנו עם הציבור הרחב וכמובן לנהל את המאגר שלנו בצורה נפלאה.
לאחר רישום קצרצר לאתר, נוכל ליצור Repository מרוחק ולדחוף אליו את ה-Repository המקומי שלנו.

פקודת Clone

פקודת **Clone** מאפשרת לנו ליצור עותק של Repository מרוחק על המחשב שלנו.
בדוגמא זו המאגר הוא של Bootstrap והוא נמצא ב-GitHub.
על מנת לקבל העתק של המאגר נפתח את חלון ה-Bash ונשתמש בפקודה הבאה:

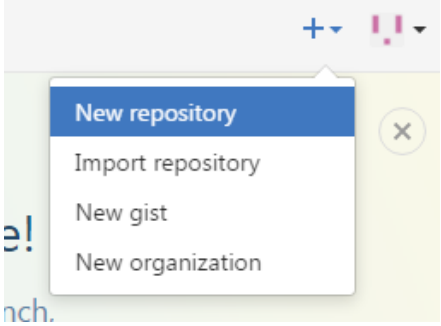
```
git clone https://github.com/twbs/bootstrap.git
```

את הכתובת של ה-Repository העתקתי מהעמוד של המאגר ב-GitHub.



יצירת Repostiory ב-GitHub

לאחר ההרשמה ל-GitHub נוכל ליצור מאגר חדש על ידי הקישור הייעודי לכך:



במסך שיפתח לנו נמלא את הפרטים הבאים:
Repository name – השם של המאגר שלנו. מאחר ומאגר זה הוא תחת השם משתמש שלנו נוכל לבחור איזה שם שנרצה.
Description – כמה מילים על המאגר שלכם – אופציונלי.
Public/Private – מאגר ציבורי או פרטי. מאגר פרטי הוא בתשלום.
שליחת הטופס תיצור את המאגר.

פקודת Push

אז יש לנו מאגר מקומי ומרוחק, על מנת לדחוף את המאגר המקומי למאגר המרוחק נצטרך להגדיר את **ה-Remote** שלנו על ידי הפקודה הבאה:

```
git remote add origin https://github.com/YourGitHubUserName/YourGitHubRepositoryName.git
```

Origin הוא השם של ה-Remote שלנו.

על מנת לדחוף את המאגר המקומי שלנו למאגר המרוחק נשתמש בפקודה הבאה:

```
git push -u origin master
```

Master זהו השם של הענף הראשי שלנו.

ענפים - Branches

ענף הוא "גרסה" של ה-Repository שלנו.

לצורך דוגמא, נאמר ופיתחנו מערכת כלשהי, הענף הראשי נקרא **master** והוא "מכיל" המערכת בגרסתה "הקלאסית".

אם נרצה להעתיק את המערכת ולהוסיף לה יכולות מיוחדות/מותאמות אישית ללקוח כלשהו או בכלל לעבוד על גרסה חדשה למערכת, כדאי לנו ליצור ענף חדש. לאחר סיום העבודה ניתן גם למזג (**merge**) בין הענפים.

על מנת ליצור ענף חדש, נשתמש בפקודה הבאה:

```
git branch newbranch
```

על מנת לקבל את רשימת הענפים שיש ברשותינו:

```
git branch
```

הענף המסומן בכוכבית - זהו הענף שאנו עובדים עליו כרגע.

על מנת לעבור לענף אחר:

```
git checkout newbranch
```

על מנת למחוק ענף:

```
git branch -D newbranch
```

פקודת Pull

לאחר שלמדנו על עבודה עם שרת מרוחק וענפים, נשאלת השאלה מה יקרה אם ביצענו **clone** למאגר מרוחק, ביצענו שינויים כלשהם אבל הענף המקורי (זה שבשרת המרוחק) עודכן?
אנו בוודאי נרצה "**למשוך**" אלינו את השינויים.

פקודת **git pull** תעשה זאת:

```
git pull origin master
```

חלאס.