תכנות מונחה עצמים ועקרונותיו בשפת ג'אווה

תכנות מונחה עצמים (OOP) הוא גישה בתכנות שמבוססת על יצירת **אובייקטים** שמייצגים ישויות או רעיונות בעולם האמיתי, עם תכונות לכל עצם (נתונים השמורים בו - משתנים) ופעולות (פונקציות הפועלות על האובייקט).

בתכנות מונחה עצמים ישנם ארבעה עקרונות שנעבור עליהם, עקרונות אלו הינם הבסיס של תכנות מונחה עצמים ומכווינים גישה לשימוש בו.

אחת הסיבות להשתמש ב OOP הינה ארגון קוד בצורה יעילה.

לדוגמה, כאשר קיבלנו משימה לשמור מידע על לקוחות נוכל לשמור אותם בקוד עם ובלי OOP, נראה את ההבדלים:

```
String Customer1_name = "man1";
String Customer1_email = "blabla@gmail.com";


String Customer2_name = "man2";
String Customer2_email = "man2@gmail.com";
```

כאן נראית עבודה עם שימוש בעצמים ללא OOP

בעזרת OOP הקוד יראה מסודר ויעיל יותר, ניצול אובייקט עבור כל לקוח

```
class Employee {
    String name;
    int age;


    Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

את האובייקט ניצור במחלקה בשם האובייקט בקובץ ג'אווה חדש (רצוי, גם בשם האובייקט).

```
class Employee {


}
```

בתוך מחלקה זו נצטרך ליצור בנאי( ) (Constructor) שמטרתו תהיה ליצור את העצם, בעצם ליצור מהקוד "מופע" של אובייקט ייחודי עם תכונות בזיכרון.

במידה ולא ניצור בנאי שהופך את הקוד מקוד מסתם ושתנים ופעולות לעצם כולל שהוא הבעלים שלהם, הJVM (מערכת מובנת בשפה) תיצור אותו באופן דיפולטיבי בצורה מינימלית. תמיד נרצה ליצור בנאי ולא להשתמש בדיפולטיבי.

פונקציית הבנאי חייבת להיות בשם האובייקט וכאשר ניצור אובייקט,

חדש מהסוג הזה הבנאי יקרא בצורה אוטומטית ליצירת האובייקט בזיכרון:

```
People p = new People(); //  שורה זו תקרא לבנאי של people
```

נרצה ליצור אובייקט של אדם, עם תכונות כמו גיל ושם (לא ניצור כאן בנאי, נשתמש בדיפולטיבי לדוגמה זו)

```
class People {
     String name = "ORI";
     int age = 19;
}
```

כעת בתוכנית ה Main נוכל ליצור אובייקט מסוג People!

```
public class Main {
```

```
     People p;
}
```

כעת יצרנו אובייקט בשם p מסוג People, כלומר במקום ליצור 4 משתנים בצורה מפוזרת ב MAIN אנו יכולים ליצור אובייקט ולגשת לתכונות (משתנים) ומתודות השמורות בו.

כעת אם ניגש לתכונה age שנמצאת בתוך האובייקט שלנו נקבל 19.

```
public class Main {
     People p = new People;
     System.out.println(p.age());
}
```

כעת נוכל ליצר אובייקט ספציפי עם התכונות שהגדרנו מראש באובייקט.

ולכאן נכנס הבנאי.

כאשר אנו "מחיים" אובייקט, כלומר הופכים אותו מסתם משתנים שמורי לאובייקט ששמור בזיכרון האובייקט נבנה בעזרת העולה הבונה.

אז כאשר נרצה ליצור אובייקט שיוכל לקבל גיל ושם נכניס את זה לבנאי שיקבל בפרמטרים שלו את התכונות הנדרשות למופע.

נשנה את האובייקט שלנו כך שיוכל לקבל ערכים ולהיווצר לפיהם:

```
class People {
     String name; //  .יצרנו משתנים כרגע ללא ערכים ספציפיים
     int age;

     Public People(String chars,int nums) {
          age = nums; // הפרמטר המתקבל יכנס למשתנה
          name = chars;
     }
```

```
}
```

כעת, כאשר נרצה ליצור אובייקט מסוג People נהיה חייבים להוסיף פרמטרים לבנאי, שייצור מופע בזיכרון לפיהם.

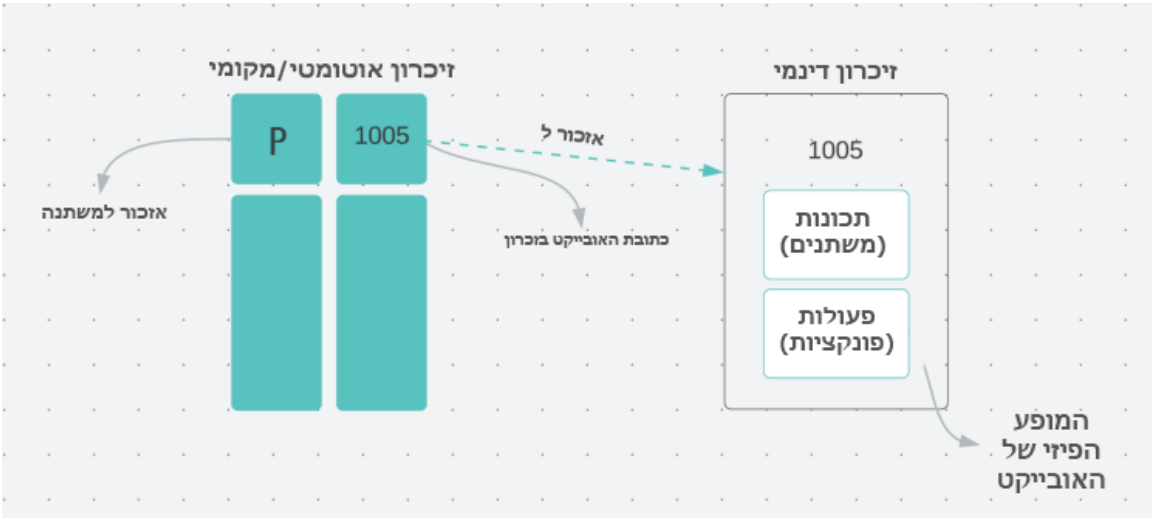
```
public class Main {
     People p = new People("Manzur",15);
     System.out.println(p.age); // נקבל 15 כפלט
     System.out.println(p.name); //  נקבל manzur כפלט
}
```

כעת כל עוד לא הגדרנו אחרת, התכונות (משתנים) באובייקט ציבוריים, כלומר ניתן לגשת ולשנות אותם מהתכנית הראשית.

ולכן כאשר נרצה לשנות את גיל האדם באובייקט, נוכל לגשת ולשנות.

```
public class Main {
     People p = new People("Manzur",15);
     p.age = 19;
     p.name = "Gil";
     System.out.println(p.age); // נקבל 19 כפלט
     System.out.println(p.name); //  נקבל Gil כפלט
}
```

ובחזרה לזכרון,



זיכרון מחסנית/אוטומטי/מקומי – stack memory	***זיכרון דינמי – Heap memory***
זיכרון מחסנית (Stack Memory) מיועד לשמירת משתנים מקומיים, כמו משתנים של פונקציה, שנמחקים אוטומטית בסיום הפונקציה. זיכרון זה מוגבל בגודל ולכן כאשר ניצור אובייקט מורכב עם פונקציות ומידע גדול נעדיף שישמר בזיכרון הדינמי. המשתנים נמחקים בסיום פונקציה והזיכרון אינו מנוהל.	זיכרון דינמי (Heap Memory) נועד לשמירת אובייקטים ומשתנים דינמיים, שנשמרים בזיכרון גם לאחר סיום הפונקציה שבה הוקצו, כל עוד יש התייחסות אליהם. זיכרון זה גמיש הרבה יותר מכיוון שמנוהל בצורה דינמית וניתן להקצות אובייקטים גדולים מאוד. הזיכרון מנוהל ע"י "גארבג' קולקטור" שמנהל את הזיכרון הזה, כאשר יש הפניה לאובייקט הוא יישאר בזיכרון, במידה ואין, הוא יהיה אחראי לשחרר אותו.

תרגול:

1 . צרו אובייקט מסוג רכב עם תכונות – מהירות מקסימלית, צבע, וחברה.

2. צרו פונקציה המקבלת מערך מסוג האובייקט ומחזירה את האובייקט עם המהירות המקסימלית הגבוהה ביותר.

תשובה:

```
class Car {
    int maxSpeed;
    String color;

    public Car(int maxSpeed, String color, String name) {

        this.maxSpeed  = maxSpeed;
        this.color  = color;
        this.name  = name;
    }
}
class Main {
    public static void main(String[] args) {
        Car c1 = new Car(200,"blue","BMW");
        Car c2 = new Car(150,"blue","mazda");
        Car c3 = new Car(320,"Red","hyundai");
        Car[] cars = {c1,c2,c3};
        System.out.println(fastestCar(cars).maxSpeed);
    }
    public static Car fastestCar(Car[] cars) {
        Car carMax = cars[0];
        for(Car c: cars){
            if(c.maxSpeed > carMax.maxSpeed) {
                carMax = c;
            }
        }
        return carMax;
    }
 }
```

בתרגיל זה ראינו שימוש בפעולה this.___

שימוש בthis לפני תכונה תאמר לגאבה שמדובר באחת מהתכונות של העצם.

למרות שלא חובה להשתמש בזה, תמיד נעדיף כדי למנוע בלבול.

כאשר אנו משנים את אחת מתכונות העצם נעדיף להשתמש בthis מכמה סיבות עקריות:

- קוד קריא יותר – נוכל לזהות בבירור את כוונת הפעולה, מתי רצינו לשנות ערך תכונתי
- נמנע משגיאות כאשר נשתמש בפעולה המקבלת משתנה בעל אותו שם
- שמירה על סטנרדט מסויים בקוד, קונבנציונליות.

כעת נתקדם לפעולות,

בנוסף לתכונות (משתנים) של אובייקט, כמו צבע, מהירות מקסימלית, מהירות נוכחית וחברה של רכב.

לאובייקט ניתן גם ליצור פעולות השייכות לו.

לדוגמה לאובייקט מסוג רכב נוכל להוסיף פעולה של לחיצת גז שתוסיף לו למהירות בו נוסע, פעולה של ברקס שתוריד מהמהירות ופעולה של צפירה.

כאשר נרצה ליצור פעולה לאובייקט, הפעולה תהיה שייכת לאובייקט בלבד ונוכל להפעיל אותה עליו, כלומר פעולה של אובייקט אינה פעולה סטטית כמו פונקציית MAIN.

מכיוון שאנו רוצים ליצור פעולה השייכת למופע מסוים של אובייקט, ואת הפעולה נצטרך לעשות עליו, לא נשתמש במילה static לפני שם הפונקציה, שמשמעותה היא פעולה השייכת למחלקה מסויימת אך לא למופע שלה.

אז ניצור פעולה כמו פונקציה סטנדרטית

```
class Car {
    int maxSpeed;
    String color;
    Int currSpeed;

    public Car(int maxSpeed, String color, String name) {

        this.maxSpeed  = maxSpeed;
        this.color  = color;
        this.name  = name;
    }

    Public speedup() {
        if(currSpeed <= MaxSpeed - 10) {
            this.currSpeed() += 10; // כל פעם שנקרא לפונקציית המהירות תעלה בעשר
        }
        else{
            currSpeed = MaxSpeed;
        }


    }
}
```

הגדרות גישה

כמו שאמרנו, בתכנות מונחה עצמים יש לנו את האופציה לשלוט בגישת התכונות והפעולות של האובייקט.

מכאן מגיע אחד מהעקרונות בתכנות מונחה עצמים –

עקרון הכימוס – Encapsulation

עד עכשיו לפני שם הפונקציה נהגנו להוסיף Public,

אך יש כמה אופציות נוספות שכדאי וחשוב לדעת,

ראשית, מהי הגדרת גישה?

הגדרת גישה הינה הגדרה בה נוכל לשלוט בגישה לתכונות ופעולות כלשהן.

אז למה בכלל שנשנה את הגישה לאובייקטים ותכונותיהם?

יש כמה סיבות עיקריות

ראשית, כאשר אנו יוצרים אובייקט מסוים אנו יודעים בדיוק מה תכונותיו יהיו, מה יוכלו לקבל, מה המגבלות שלהן וכו'..

כאשר נעבוד בעתיד על פרויקט עם עוד אנשים ואף גם על אותה מחלקה, נצטרך לשמור על גישה מבוקרת שתמנע טעויות אנושיות, חריגות גישה ובעיות נוספות

לדוגמה – אני עובד על אובייקט שיודע לתכנן מצב משלוח מסוים, כלומר האובייקט הוא חבילה, אני יודע שהאובייקט שלי יודע להתנהל כאשר תכונת הסטטוס שלו היא shipped ו not Shipped,

מחר יגיע עובד חדש שירצה לשפר את הקוד וליצור פעולה במחלקה שידעת ליצור פעולות באובייקט, ובטעות באחת מהפעולת שעשה הוא הכניס ערך שאינו נכלל באחת מן האופציות שהאובייקט יודע לעבוד איתן בפעולות אחרות שלו.

כאשר המתכנת ינסה לגשת לתכונת הסטטוס שלו, בהנחה שהיא ציבורית, הוא יוכל לעשות זאת בלי בעיה בכלל!

סיבה נוספת להגבלות גישה על אובייקטים היא **שיפור קריאות הקוד (כאשר הנחזור לקוד, או שהוא ישתנה ע"י אדם נוסף, יהיה קל יותר להבין איפה נכון לשנות, מה אפשר לשנות, ואיך)** והתאמה לקונבנציה מודרנית.

ולכן, כאשר אנו מגדירים תכונות ומשתנים באובייקט ניקח את זה בחשבון ונעדיף תמיד להשתמש בהגבלות גישה שונות.

Modifier	Class	Package	Subclasses	World
public	✅	✅	✅	✅
protected	✅	✅	✅	❌
no modifier	✅	✅	❌	❌
private	✅	❌	❌	❌

פאבליק – יוכלו לגשת למשתנים מכול מקום, במחלקה עצמה, במחלקות יורשות (נלמד בהמשך) במחלקות ב Package של הקובץ, ובמחלקות חיצוניות.

פרוטקטד – יוכלו לגשת מהמחלקה עצמה, מחלקות יורשות, ומחלקות שונות ב PACKAGE

ללא ציון הגדרה (דיפולטיבי) – רק במחלקה עצמה וב PACKAGE

פרייבט- רק במחלקה עצמה.

כאשר נגדיר תכונות של עצם כמעט תמיד נעדיף להשתמש ב – private, וכדי לגשת לתכונות של האובייקט ממחלקות שונות, מה-main וממלקות יורשות נצטרך לעבור דרך פעולות שיעזרו לנו לשנות בצורה תקינה את התכונות.

פעולות אלו נקראות getters ו – setters

כשמן –

Getter: פעולה לקבלת את תוכן התכונה.

Setter: פעולה שעוזרת לשנות את תוכן התכונה באופן תקין.

פעולת הבנאי חייבת להיות פעולה ציבורית, כדי שנוכל ליצור את האובייקט מכל מקום שנרצה, אחרת לא נוכל ליצור אובייקט משום מקום והעצם יהיה לא רלוונטי.

לדוגמה, כאשר יש לנו עצם של חשבון בנק

```
class Account{
    private int Balance;
    private int ID;

    Public Account(int ID) {
        this.ID = ID;
        Balance = 0;
    }

    Public getBalance() {
        Return Balance; // הפעולה רק תחזיר את תוכן התכונה ובכך תתן גישה לצפות בתוכן מבחוץ
    }
    Public setBalannce(int amount) {
        // if(...) // :כאן נוכל להוסיף כל תנאי שנרצה כדי שהשינוי יהיה מבוקר לדוגמה
        If( amount > 0 ) {
          Balance = amount; // .כאן נשנה את תוכן התכונה במידה והפרמטר עמד בתנאי הכרחי
        }
    }
        // כאן נעשה גם GETTERS AND SETTERS
        // לתכונת הID
 }

Public class Main {
        Public static void Main(String[] Args) {
        Account acc = new Account(200);
        acc.Balance = 500// יראה שגיאה מכיוון שאאיננו מכירים בתכונה זו – חסרי גישה אליה
        acc.setBalance(500); //.תוכן התכונה תהיה 500 כעת
        }
}
```

ובחזרה ל – STATIC

בנוסף לפעולה סטטית – ששיכת למחלקה ולא למופע ספציפי -

(כמו פונקציית המיין, ולדוגמה אם נרצה ליצור מחלקה של פעולות מתמטיות, חיבור חיסור וכו, הפעולות יהיו פעולות סטטיות שמקבלות 2 ערכים, הפעולות אינן קשורות למופע ספציפי)

ניתן להשתמש בתכונות סטטיות.

תכונה סטטית זו תכונה הקשורה לכל המחלקה ולא למופע ספציפי, כאשר ניגש אליה באמצעות שם המחלקה היא תשתנה לכלל האובייקטים ממחלקה זו.

לדוגמה – משתנה מסוג counter במחלקה הקודמת:

```
class Account{

    private int Balance;

    private int ID;

    public static int counter = 0; // מאותחל רק פעם אחת, בהתייחסות הראשונה למחלקה


    Public Account(int ID) {

        this.ID = ID;

        Balance = 0;

        counter++; // כל פעם שניצור אובייקט(נשתמש בבנאי) נקדם את תכונת המחלקה באחד

    }

}

class main(


    public static void main(String Args){

        Account acc1 = nee Account(166); // overall counter will be 1;

        Account acc2 = nee Account(50); // overall counter will be 2;

        Account acc3 = nee Account(519); // overall counter will be 3;

        //  כעת לא משנה מאיזה עצם ניגש לקאונטר הוא יהיה 3 כיוון שהוא סטטי ומשותף לכול
          העצמים במחלקה









    }


)
```

למרות שאמרנו שמשתנה סטטי שייך למחלקת האובייקט ולא למופע ספציפי, ניתן לגשת אליו דרך אובייקט אך דרך זו פחות אידאלית ונכונה ואף הקוד מחליף את מהאובייקט למחלקה

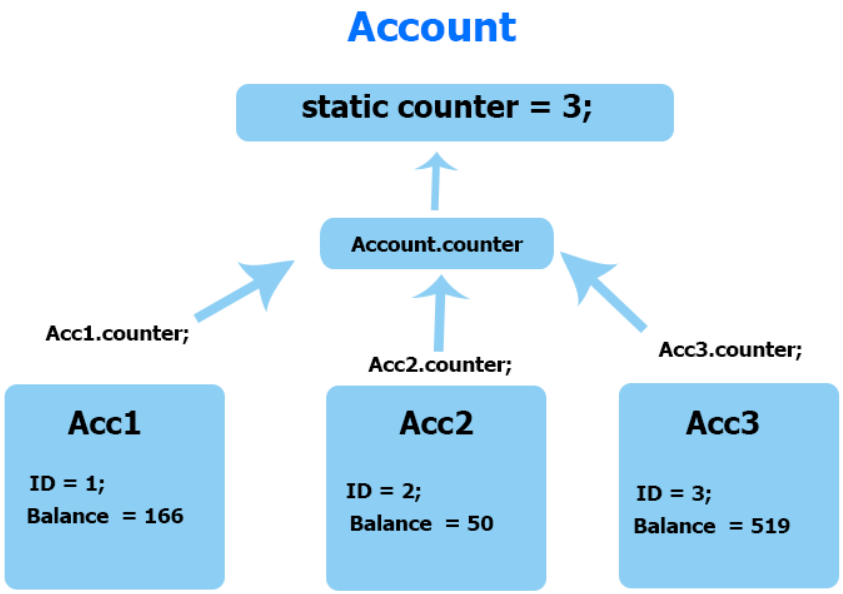
לדוגמה:

```
Acc1.counter;  → Account.counter
```

ג'אווה תבצע החלפה לבקה מהמחלקה ולא מאובייקט למרות שדרך זו תקינה מבחינת קוד.

אך תמיד נעדיף לגשת למשתנה סטטי משם המחלקה ולא מהאובייקט.

בעיקרון זה נראה כך –



העמסה – overloading

לפני נשתחיל לדבר על הורשה (העקרון הבא) נלמד מהו overloading.

מנגנון ה overloading הינו מנגנון בגאבה ובשפות שונות שמאפשר לנו לכתוב כמה פעולות (פונקציות) בעלי שם זהה אך בעלי פרמטרים זהים וגאבה תדע לזהות לאיזה פונקציה אנו קוראים לפי הפרמטרים (או חוסר בפרמטרים, כאשר נשתמש בפונקציה בעלת אותו שם פעמיים, פעם אחת עם פרמטירים ופעם אחת בלי).

לדוגמה כאשר ניצור פעולה של חיבור

```
class Main {


public static int sum(int a, int b) {

        return a+b;

}


public static double sum(double a, double b) {


        return a+b;

}


public static void main(String[] args){


  System.out.println(sum(1,3)); //כאן יודפס הפלט מהפונקציה הראשנה שמקבלת מספר שלם כפרמטר

  System.out.println(sum(3.4 , 6.6));//  כאן יודפס הפלט מהפונקציה השנייה שמקבלת מספר
עשרוני כפרמטר



}

}
```

אם נשים לב, נראה שגם הפעולה println מקבלת כפרמטר פעם אחת int ובפעם השניה double, לכן ניתן לראות שגם פונקציה זו משתמשת בעקרון ה overloading ויודעת לנתב את הפרמטרים yלפעולה הנכונה.

println(int x)

println(double x)

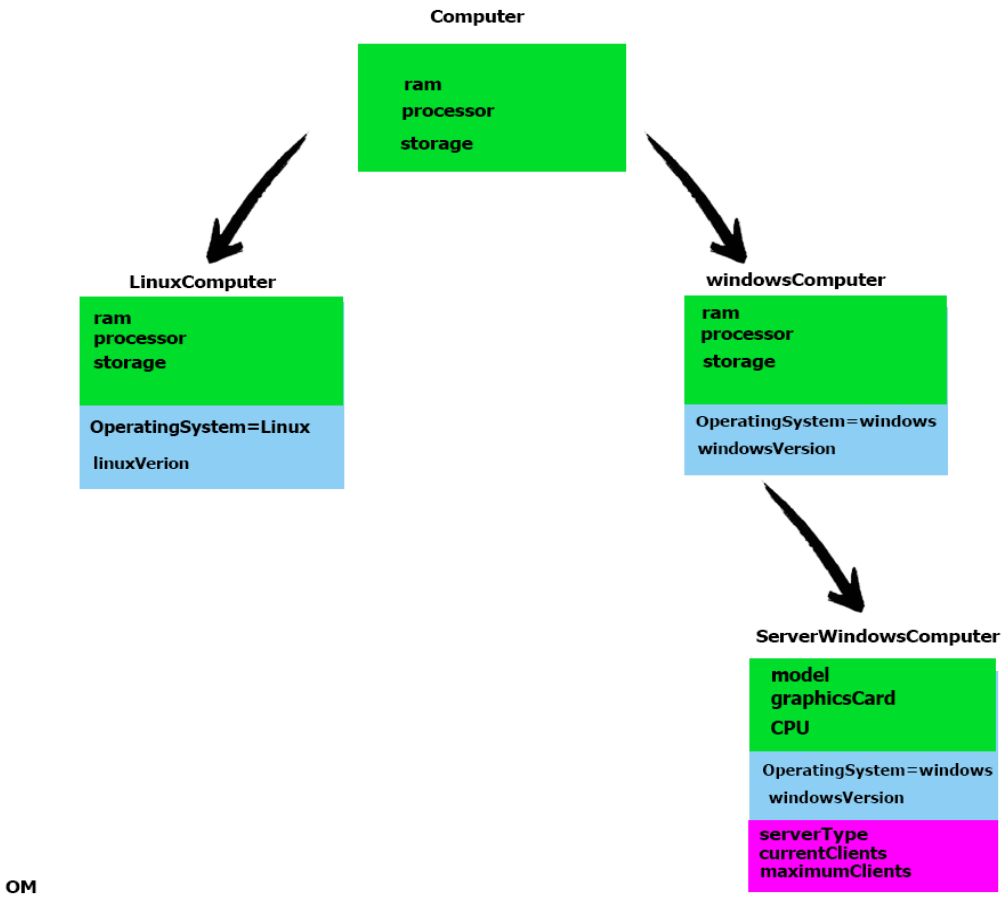
כאשר נשתמש באובייקטים נצטרך להגדיר בעצמנו איך נדפיס אותם בעזרת toString, שprintln משתמשת בה.

עקרון ההורשה – Inheritance

עקרון נוסף בג'אווה הוא עקרון ההורשה -

עקרון זה הוא גם עקרון שעוזר לנו לממש את התכנות לעולם האמיתי.

עקרון זה מאפשר לנו להוריש בתור אובייקט אב/ לרשת בתור אובייקט בן את התכונות והפעולות שלו.



בדוגמה למעלה ניתן לראות שישנו אובייקט מסוג מחשב, כלומר כרגע ניתן ליצור אובייקט מסוג רכב ולהשתמש במופע שלו.

בנוסף לאובייקט זה יצרנו אובייקט שיורש אותו בשם windowsComputer, כאשר יצרנו אותו, בחרנו לרשת את האובייקט computer כלומר כל התכונות והפעולות שלו יהיו זמינות למחלקת הבן (גישה ישירה במידה והם פאבליק/פרוטקטד, אחרת בעזרת getters&setters)

כדי לרשת ממחלקה אנו נשתמש במילה השמורה extends (להרחיב) כאשר ניצור את המחלקה.

זה יראה כך:

```
class WindowsComputer extends Computer {

       String operatingSystem = windows;

       int windowsVersion;


 }
```

כאשר אנו יורשים מאובייקט אנו צריכים לממש את פונקציית הבנאי שלו בעזרת המילה השמורה super שמפעילה את הבנאי של אובייקט האב, פעולת ה super מקבלת את הפרמטרים שהבנאי של פונקציית האב צריכה, ולכן כל פעם שניצור אובייקט של windowsComputer נצטרך לקרוא לבנאי של פונקציית האב בבנאי הנוכחי עם הפרמטרים הדרושים לו.

```
class WindowsComputer extends Computer {

       String operatingSystem = windows;

       int windowsVersion;


       public WindowsComputer(int ram,String Processor,int storage,int windowsVersion) {

         super(ram,Processor,storage); // הפרטמרים שעצם זה מקבל נשלח כאן לבנאי האב

         this.windowsVersion = windowsVersion;

       }


}
```

כאשר במחלקת האב יש בנאי שלא מקבל פרמטרים, לא נצטרך להשתמש ב super, זה יקרה באופן אוטומטי.

כאשר לא נשתמש ב super למרות שבנאי האב יהיה חייב בפרמטרים נתקל בשגיאת קומפילציה והקוד לא ירוץ – במהלך המרת הקוד לשפת מכונה הקוד יתקל בשגיאה זו ולא יריץ את התכנית.

כאשר אנו מפעילים את בנאי האב לא נוצרים שני אובייקטים למרות שאנו כן מפעילים שני בנאים.

מה קורה בפועל?

כאשר אנו יורשים מאובייקט A שדורש 8 בתים, למחלקה B שדורשת 10 בתים, גאבה יודעת ש A יורש מ B וכאשר ניצור את B היא תקצה את הזכרון ש A דורש ותוסיף את הזכרון ש B דורש לזכרון כולל של B.

כאשר נקרא לבנאי של מחלקה B, כל הפעולות יתבצעו בזכרון הכולל של B ולא ניצור מופע חדש אלא נפעל במופע הכולל הקיים והמתודות והתכונות ישתנו בקטע הזכרון הכולל.

דוגמה למחלקה שלא דורשת שימוש ב – super

```
class Point {

        int x = 1;

        int y = 1;

        public Point() {

        // super() **** הסבר למטה*****

        // גם אם הבנאי ריק הוא יכנס לבנאי הזה כדי ליצור א האובייקט

        }

}


class Main {

  public static void main(String[] Args) {

        Point p1 = new Point();

        System.out.println(p1.x);

   //(דיפולט מודיפייר) X לאחר שיצרנו את האובייקט נוכל לפנות למופע שלו בזכרון ולהדפיס את

  }

}
```

מדוע כתבנו את ה – super שם ? (נכתב בקומנט אבל למה הוא שם?)

באופן דיפולטיבי, כל מחלקה בגאבה יורשת מהאובייקט Object אוטומטית.

כלומר כל מחלקה שניצור תהיה זהה לאם נוסיף לה extends object

ולכן כאשר יצרנו את מחלקת Point, היא יורשת אטומטית מהמחלקה Object, כלומר הקוד שכתבנו שקול לקוד הבא:

```
Class Point extends Object {

        ...

}
```

אז למה זה קורה? למה באופן דיפולטיבי אנו יורשים מ – object ?

יצא לכם פעם להדפיס בטעות אובייקט ולקבל משהו כזה :0xf094c649 ?

או אולי להשתמש בפעולה A.equals(B) ?

פעולות אלו הן פעולות שאנו יורשים מהמחלקה הבסיסית ביותר object.

כאשר אנו מדפיסים אובייקט אנו פונים באופן אוטומטי לפונקציה שיירשנו מהאובייקט Object.

כלומר:

```
System.out.println(p1);
```

יהיה שקול ל:

```
System.out.println(p1.toString()); // שתילקח ממחלקת האב אם לא נגדיר אחרת
```

למה זה קורה?

אנו פונים לאובייקט System.out בעל הפעולה println , פעולה זו מחזירה לנו Null במידה והאובייקט ריק כלומר שלחנו אובייקט ששווה ל –NULL .

אחרת הוא יחזיר לנו:

Object.toString()

כלומר ישלח אותנו לפעולה הזו במחלקת Object אם לא הגדרנו אותה שוב במחלקת הבן (או באחת מהן).

]

למה בפעולה println אנו מתייחסים לפעולה Object כלומר מסתכלים על האובייקט כ Obect ולא כאובייקט הבן שקראנו דרכו לפונקצייה, אך בכל זאת אם נממש אותה שוב אצל אחד הבנים נקבל את הפלט שיצא מפעולת הבן ולא מפעולת ה – Object?

שאלה זו שולחת אותנו לעיקרון הבא – פולימורפיזם, שנרחיב עליו בהמשך, אך כרגע רק נסביר.

כמו שאמרנו כאשר אנו יוצרים מופע אנו מקבלים מקטע זכרון למופע, שהופך אותו מסתם מחלקת פעולות למופע של עצם בזכרון – ממש לישות.

כחלק מקטע זכרון האובייקט אנו מקבלים גם מצביע לטבלה הנקראת טבלת VMT (Virtual Method Table) בטבלה זו שמות המתודות שיש לעצם זה ומצביעים אליהן בזכרון. (מצביע לחלק בו הפעולות נמצאות , כמו שדיברנו על HEAP ו STACK לתכונות, כאשר אנו יוצרים

אובייקט ה JVM – מערכת מובנת בגאבה, יוצר לעצם מקום ב – STACK HEAP ו Method Area בו נשמרות כל הפעולות של האובייקט.)

כאשר יצרנו לדוגמה אובייקט, הפעולות יהיו משהו כזה:

- נשמור את כתובת הפונקציה שממנה קראנו לאובייקטים - כתובת חזרה ל MAIN
- ניצור ב HEAP את מקטע הזיכרון של העצם (תכונות וכו)
- כל הפעולות ישמרו ב METHOD AREA
- ב HEAP ישמר גם מצביע VMT (טבלת הכתובות של פעולות הפונקציה)
- ה VTM (גם ב METHOD AREA) יחזיק את שמות הפעולות וכתבותיהן כדי לא לחפש כל פעם אותן, וכדי לדעת אילו פונקציות שייכון לאותו העצם.

ה VMT של p1 בעלת מצביע לפעולה toString של הבן, כלומר אחרי שאנו דורסים את הפעולה בבן אנו נשנשה את הכתובת ב VMT לכתובת של הפעולה הדורסת.

ואז גם כאשר נסתכל על p1 כ object אנו עדיין נלך לטבלת ה VMT כדי לחפש את כתובת המתודה ב METHOD AREA ונמצא את הכתובת של הפעולה הדרוסה שהסתנתה בדריסה.

אם ככה, אז למה אנו לא יכולים להדפיס את כל פעולות הבן?

כאשר אנו מסתכלים על אובייקט בן כאב, לפני שנדפיס את הפעולה תתבצע בדיקה אוטומטית שאת הפעולה יש גם בעצם האב(פונקציה משותפת) . (אנו נרצה להסתכל על אובייקט בן כאב במקרים שיש פונקציות שמקבלות את האובייקט כאב, או לדוגמה שנרצה להבטיח שכל האובייקטים מסוג מסויים יהיהו בעלי פונקציה שיש אצל האב ונמנע משגיאות מיותרות)

בזכרון יראה משהו כזה:

Stack Memory

משתנים מקומיים נוספים ישמרו כאן.
000: Object -> 003 // מצביע לאובייקט בהיפ

001: Point -> 005

Heap Memory

003: Object
// כאן ישמרו משתנים של האובייקט
004: VTM Pointer (מסוג מצביע)(לטבלה של האב)->

005: Point
// כאן ישמרו משתנים של האובייקט
006:VTM Pointer (מסוג מצביע)(לטבלה של הבן)->

Method Area

007: toString(){
008: .... }
009:
010:toString() { דרוסה
011:.....
012: ..}
013:
014:
015:VMT Parent
016:toString -> 007
017:
018:VMT Child
019:toString -> 010
020:

בגאבה כל מחלקה יכולה לרשת אך ורק מחלקה אחת, כאשר ננסה לרשת יותר ממחלקה אחת נקבל שגיאת **קומפילציה** והקוד לא יוכל לעבור קופילציה של javac (מי שמקמפל את הקוד ל bytecode, ולאחר ב bytecode הקוד עובר ל JVM. שגיאות אחרות שקורות בזמן הריצה, כלומר לא יזוהו בקומפילציה כמו כאל שתלויות בקלט ממשתמש או קבצים במחשב, יזוהו ברמת ה JVM שיזוהו לאחר הקומפילציה, כשהם כבר ב BYTECODE **בזמן הריצה.)**

לפתרון ירושה אחת בגאווה כן יש אופציה למימוש של יותר במחלקה אחת אך לא נשתמש בהורשת מחלקה אלא בהורשת ממשק, זה עקרון שנדבר עליו בהמשך, אך מבחינת ירושה מחלקה אחת יכולה לרשת רק אחת בעזרת extends

מנגנון הדריסה - overriding

מנגנון הדריסה הוא מנגנון ירושתי (כלומר המימוש בו הוא במחלקה יורשת) כאשר אנו דורסים את פעולת האב כדי ליצור פעולת בן מתאימה. (בעלת שם ופרמטרים זהים)

נשתמש בפעולת דריסה כאשר הפעולה של האב אינה תואמת את פעולת הבן והמימוש שלה שונה.

לדוגמה כאשר יש לנו אובייקט של עובד, ויש בעצם זה פעולות ורעיון מסוים שאנו רוצים לממש במחלקת בן הנקראת מנהל.

המחלקת האב – עובד פעולה של הכנסת משכורת תכניס לעובד 30 שח עבור כל שעת עבודה, אולם אצל מחלקת הבן – מנהל, פעולה זו תכניס לעובד 50 שח עבור כל שעה.

נניח שהפעולה קיימת במחלקת האב ונראית כך:

```
class Worker {

  protected double salary = 0;

  public Worker(){


  }

  public void addPayment(double workedHours) {

    salary += 30 * workedHours;

  }

  public double getSalary() {

      return salary;


  }
}
```

כאשר נרצה לרשת את כל הפעולות של מחלקת האב כדי לחסוך בקוד, נוכל להשתמש בהן ולשנות (לדרוס) פעולות רלוונטיות לפי הצורך.

לדוגמה נרצה לשנות את השכר השעתי של מנהל ל 50 בעזרת דריסה של פעולת האב כך:

```
class Manager extends Worker {

    public Manager() {

      super();

    }

  @Override

  public void addPayment(double workedHours) {


    salary += 50 * workedHours;

  }

}
```

כעת כאשר ניצור אובייקט מ Manager אנו נדרום את הפעולה של האב שנקראת addPayment וכאשר נקרא לה אנו נתייחס לפעולת הבן.

איך זה קורה?

כאשר אנו מריצים את הקוד ה JVM יוצר את טבלת הפעולות של כל אובייקט, כאשר אנו מבצעים דריסה של אובייקט בטבלה זו מתעדכן הערך של המצביע לכתובת הפונקציה הדרוסה, לכן כאשר ניצור אובייקט בן המצביע לפונקציה תהיה על כתובת פעולת הבן ואם ניצור אובייקט אב הפעולה תשאר של האב.

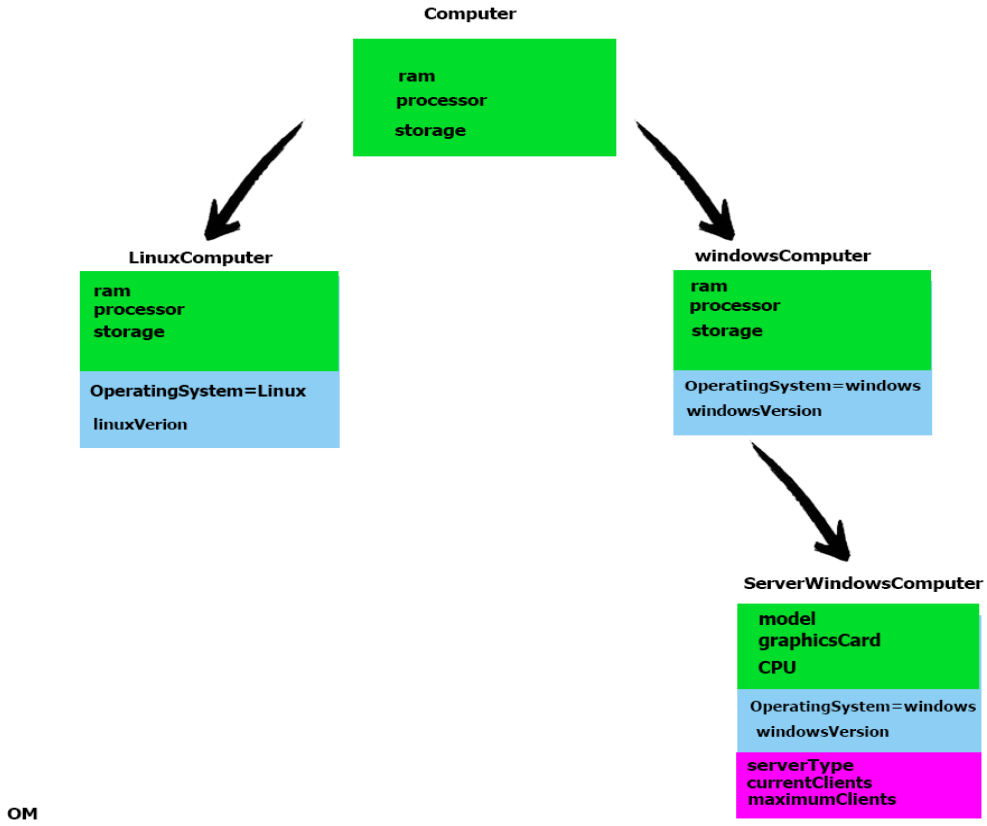
עקרון הפולימורפיזם – Polymorphism

רב צורתיות – Polymorphism

עקרון הפולימורפיזם הוא אחד מהעקרונות של תכנות מונחה עצמים, בעצם נחשב לאחד החשובים מהם.

עקרון זה נותן לנו להסתכל על אובייקט בכמה צורות, מבלי לדעת איך מחלקות הבן מימשו אותו, כלומר אני מתייחסים לאובייקט לפי פעולות האב (הפעולות והתכונות המשותפות להן בלבד) אנו יודעים שהם בעלי מימוש של הפעולות שלו (דריסה), או שימוש בהן ולכן אני יכולים להשתמש בהן מבלי לדעת איך המימוש קורה בכל אחד מהילדים, אלא בהנחה שהם נמצאים שם.

לפי הדוגמה מעקרון ההורשה



כאשר יצרנו מחלקת אב שנקראת מחשב, אנו יודעים בוודאות שלשכל האובייקטים שיורשים אותו יש את המשתנים model,GrapichsCard , CPU ואת כל הפעולות (מתודות) שההגדרנו בו.

לכן כאשר ניצור אובייקט בן מסוג שרת/מחשב ווינדאוס/מחשב לינוקס אנו יודעים בוודאות שאנו יכולים להתייחס לפעולות ולמתשנים האלו, לא אכפת לנו המימוש שלהם בפועל, אלא הידיעה שאנו יכולים להשתמש בהם ולהתסכל עליהם בצורת האב.

שימוש בפועל בפולימורפיזם קורה כאשר אנו נרצה להשתמש בתכונת בן עבור כמה אובייקטים היורשים אותו, במקום להפעיל את הפעולה עבוד כל אובייקט נוכל כל פעם לגשת לפעולת הבן דרך פעולת האב ובכך לחסוך בקוד, בנוסף איננו תלויים באיזה אובייקטים יהיו לנו, כאשר אנו בטוחים שהם יורשים מאובייקט האב הקוד גמיש ויכול להשתנות תמיד- אובייקטים יכולים להצטרף מבלי התייחסות ספציפית אליהם כאשר הם עומדים בהורשה.

)

כרגע השימוש בפולימורפיזם בהורשה של עצמים ברי מופע, בעתיד נלמד שימוש בפולימורפיזם עבוד ממשקים ועצמים אבסטרקטים – כלומר נוכל לממש אוסף פעולות ממחלקה שלא ניתן ליצור ממנה אובייקט או לממש רשימת פעולות ובכך באמת להמחיש שימור בפולימורפיזם עבור פעולות.

)

איך זה קורה בפועל בקוד?

```
  //לדוגמה יש לנו אובייקט של מחשב


class Computer { // ניצור אובייקט אב

  Public Computer() {}//  בנאי מינימלי (לא באמת חייב לעשות אותו, נוצר אטומטית)

  void start() { //פעולת הפעלה במחלקת האב

    System.out.print("Computer without an operating system is starting...");

  }

}

class WindowsComputer extends Computer{ //ניצור אובייקט של מחשב ווינדאוס היורש ממחשב

  Public WindowsComputer() {super();} //  בנאי מינימלי גם כאן לא באמת חייב ליצור אותו
                                        ולהשתמש בסופר -> יקרה אוטומטית

  void start() {

    @Override // נצהיר שמדובר בדריסה לקומפיילר - גם את זה לא באמת חייב לכתוב אנו כאן
מצהירים שמדובר בפעולת דריסה ואם יש לנו שגיאת כתיב או שאיננו ביצענו דריסה השגיאה תופיע כבר
כשנכתוב את הקוד בקומפלציה בזמן אמת שגאווה עושה לנו ונקדים את הבעיה שניצור בזמן הריצה
לשגיאת קומפילציה ונמנע מבעיות עתידיות

    System.out.println("windows computer is starting..."); // ההדפסה לאחר הדריסה

  }



}

class Main() {

  public static void main(String[] args) {

    Computer c1 = new Computer(); // כאן יצרנו אובייקט מסוג מחשב ואנו מתסתכלים עליו
              ( מסתכלים עליו כלומר המשתנה שמצביע הוא מסוג מחשב ) כאובייקט מסוג מחשב

    Computer c2 = new WindowsComputer();  כאן אנו בעצם יוצרים מופע של אובייקט הבן
ומסתכלים עליו כאובייקט האב - כלומר המשתנה ששומר אותו או המצביע שאנו מסתכלים דרכו הינו
                                                                    אובייקט האב.

כעת ניתן לראות את השימוש ברב צורתיות, אנו יכולים להסתכל על אבייקט בכמה צורות שונות.

  }


}
```

אז מה קרה כאן בעצם?

יצרנו מופע בעזרת הבנאי של מחלקת הבן, כלומר יצרנו אובייקט מסוג הבן.

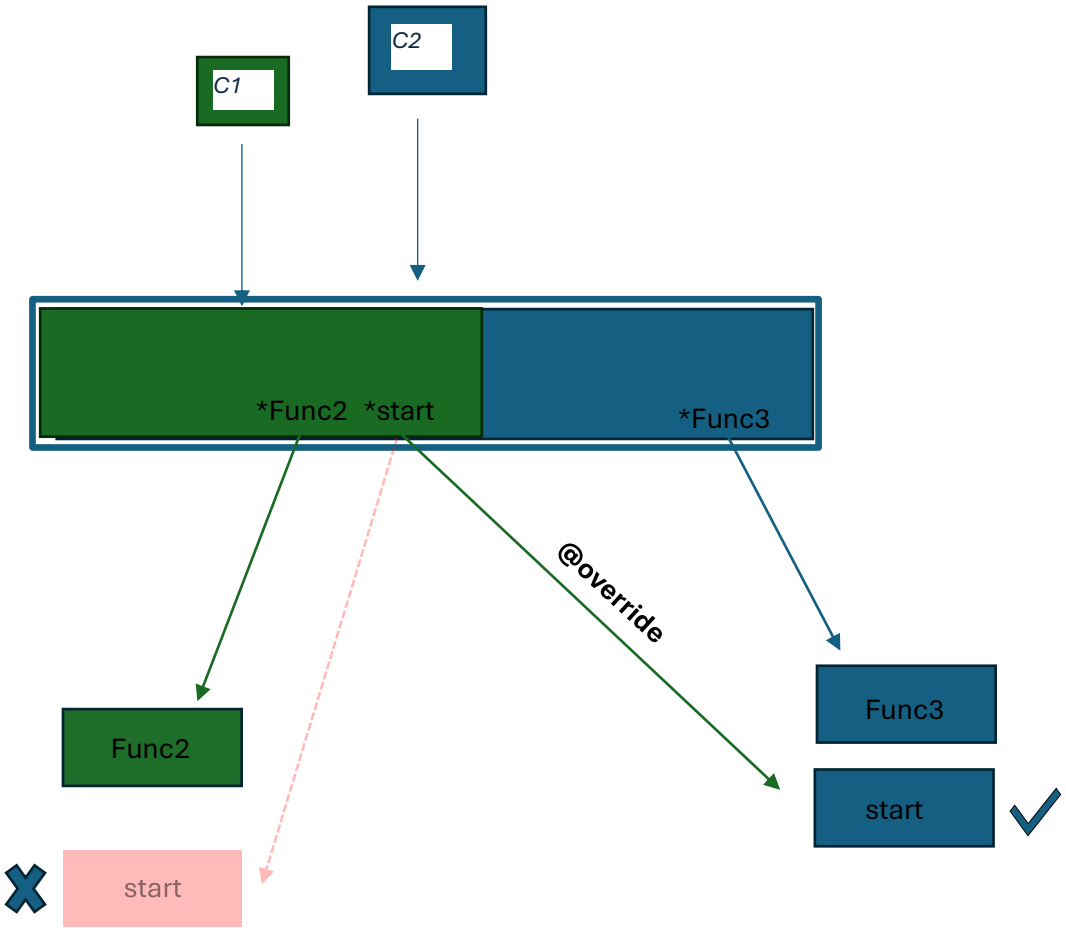
בזכרון – יצרנו בלוק של זכרון בגודל הבן, כל התכונות של הבן אותחלו, המקום בזכרון של התכונות נוצר, המצביעים של האובייקט לפעולות אלו ב JMT נוצרו, אך כרגע אנו מתייחסים לאובייקט בנקודת מבט של האב, כלומר יש לנו גישה רק לתכונות/פעולות שיש גם אצל האב.

כמובן שאם דרסנו את אחד הפעולות שיש גם באובייקט האב אנו נתייחס לפעולה הדורסת.

למה?

מה זה אומר להסתכל בנקודת מבט של אב?

כאשר אנו שומרים את אובייקט הבן במצביע לאובייקט מסוג computer אנו בעצם יוצרים מצביע שיודע להצביע בזכרון ל sizeof(compuer) , כלומר המצביע יצביע רק לבלוק הזכרון של האב בתוך המקטע הכולל של הבן, שבעל בלוק זכרון זה.



ולכן כשאנחנו מסתכלים על אובייקט בתצורת האב- אנו בעצם מוגבים לראות את אובייקט האב ומצביעיו בגודל הבלוק של הזכרון שהוא מוגבל אליו – בלוק האב, ולכן אם ננסה לגשת לפונקציות בן לא נוכל, מחוץ לגבולות המצביע.

לכן כאשר נדפיס את start, נקבל את הפעולה של ה windowsComputer גם כאשר אנו מסתכלים על האובייקט כ computer

וכמו שאמרנו, כאשר נרצה להוסיף עוד סוגים של מחשבים בעתיד ולעשות עליהם בפעולות לפי משתנים ואו פונקציות שיש לכולם, נוכל להתייחס עליהם כאובייקט האב ובכך הקוד ישאר גנרי, לא נצטרך לחזור על קוד ולקרוא לכל פונקציה באופן ספציפי לאובייקט.

כאשר אנו משתמשים באובייט עם משתנה מסוג האב (מצביע מסוג אב) איננו יכולים לגשת לפעולות ומשתנים שנוצרו/ יחודיים במחלקת הבן מכיוון שהם לא בתווך בלוק האב, ולכן כדי לגשת אליהם עלינו להתייחס חזרה לאובייקט דרך משתנה מסוגו(מצביע על כל האובייקט כולל הבן) בעזרת המרה – casting.

```
WindowsComputer c3 = new windowsCoputer(c1);
```

כעת יצרנו מצביע בגודל (windowsComputer)sizeof וכעת למצביע יש גישה לכל מקטע הזכרון האב בתוך הבן והבן.

- גם c3 וגם c1 מצביעים לאותו כתובת בזכרון, ל c1 גישה רק למקטע האב, כיוון שזה גודל המצביע ול c3 גישה לכל מקטע הזכרון, גם לתכונות ופעולות הבן.

אם נבצע :

```
system.out.println(c1 == c3); // יודפס true.
```

כמובן שתמיד לפני שנעשה קאסטינג נעדיף לבדוק האם האובייקט באמת מופע של המחלקה שאנו רוצים להמיר אליו, אחרת תתקבל שגיאת זמן ריצה שתקריס לנו את התכנית (גם שימוש בtry-catch) יכול לפתור את הבעיה אך עם שימוש יתר במשאבים כמו טיפול בשגיאה, יצירת אובייקט חריגה וכו' ולכן נעדיף לטפל בבעיה בצורה שונה.

הדרך בה נעשה את זה תהיה בעזרת הפעולה instanceof

a instanceof b תחזיר אמת כאשר aהוא חלק ממופע b, כלומר b יורש את a כלומר a יהיה אב של b

ובדוגמה שלנו לפני שנבצע קאסטינג נעשה בדיחה ש c1 נמצא במקטע הזכרון של c3 כלומר c3 יורש את c1:

```
if (c1 inctance of windowsCoputer ){
  WindowsComputer c3 = new windowsCoputer(c1);

}
```

עקרון האבסטרקציה (הפשטה) – Abstraction

עד עכשיו כשדיברנו על הורשה השתמשנו בעצם של אב, שהוא עצם בפני עצמו, כלומר ניתן ליצור ממנו מופע בזכרון. (מחלקה ללא בנאי שניתן לרשת ממנה).

בתכנות, ובעולם בכלל, אנו לפעמים נרצה להתייחס לאובייקט רעיוני מסויים, כלומר אובייטק מופשט (כללי) שאינו יכול להתקיים בפני עצמו, אלא רק בהתייחסות ספציפית יותר.

לדוגמה כאשר אנו מדברים על "בעל חיים" או " כלי תחבורה" או או אפילו "פריט לבוש" איננו באמת יכולים ליצור כלי תחבורה כעצם בפני עצמו, אלא יכולים ליצור את העצם הספציפי יותר – רכב, אוטובוס, אופניים וכו, כאשר מתייחסים לבעל חיים – כלב, חתול וכו'...

בעולם התכנות לפעמים נרצה ליצור רעיון מסויים שאיננו רוצים שיוכל להתקיים בפנו עצמו, כאשר אנו יוצרים מחלקה של כלי רכב אנחנו לא רוצים שהיא תעמוד בפני עצמה כאובייקט ותוכל להיווצר כמחלקת אב, אלא רק במחלקה שיורשים אותה כעצם ספציפי יותר כמו רכב.

אובייקט אבסטרקטי הוא לא באמת עצם כיוון שלא באמת ניתן ליצור ממנו מופע, אלא רק להשתמש בו כתבנית לאובייקטים הממשים/יורשים אותו.

אנו נשתמש במחלקה זו כאשר אנו נרצה לבנות רעיון כללי בסיסי שיכול להיות מורש לתתי מחלקות, מחלקות אלו יכולים להיות אובייקטים שניתן ליצור מהם מופע אך מהמחלקה הבסטרקית אשר נירש ממנה תהיה רעיון בסיסי/כללי של עצם שאינו בר מופע.

גם בדוגמ הקודמת שלנו, כאשר התייחסנו למחשב, היינו יכולים להגדיר כי המחלקה "מחשב" אינה יכולה להיווצר כמופע, אלא רק מחשב מסוג "ווינדאוס" או מחשב מסוג "שרת"

דוגמה נוספת:

ניצור מחלקה אבסטרקטית של אובייקט מסוג עובד (אין באמת תפקיד כזה "עובד", יש קופאי, מנהל, אחמש, גם עובד כללי זה תפקיד, ולא "עובד")

בקוד:

```
Abstract class worker {

  String name;

  Int age;// משתני מחלקה שיורשו

  void onDuty( // פעולה שתעבור ליורשים, תוכל גם להידרס

    System.out.println(name + "is now on duty");

  );

  void offDuty(name + "is now off duty");

  );

  Abstract void getRoleInfo(); //כל מחלקה יורשת תהיה חייבת בפעולת מידע על התפקיד

  Abstract double addPayment();//כל מחלקה יורשת תהיה חייבת במימוש פעולת קבלת שכר




}

Class Manager extends //גם למימוש מחלקה אבסטרקטית// Worker {

  Boolean dailyPayement = false;

  Doduble salary = 0;

  Public Manager(string name, int age) {

    This.name = name;

    This.age = age;

  }

  Void getRoleInfo() {

    System.out.println("Manager – takes care of the business overall !");

  }

  Double addPayment(){

    if(!dailyPayment){

    dailyPayment = true;

    Salary += 1500; // כאן ממשנו את פעולת האב האבסטרקטי שהיינו חייבים, בתור מנהל ממשנו
אותה בכך שהוא יקבל משכורת יומית קבועה של 1500, מימוש של עובד יכול להיות פעולה משקבלת שעות
עבודה ולפי שכר שעתי עבור אותו תפקיד תסכום למשכורת.

    }

     Else { System.out.println("payment already received!"};

    };


    setDaily(){ כאן תוכל להיות פעולה שבודקת האם באמת עבר יום לפי בדיקה של התאריך היום
יום עבר באמת אם לבדוק ולפניה האחרון השינוי של התאריך לעומת כלשהם/ימים מספר};

}
```

ממשקים – interface

באבסטרקציה - כאשר אנו יורשים אובייקט שניתן ליצור ממנו מופע אנו יוצרים שימוש של בנאי האב במקטע הזיכרון הכולל של אובייקט האב, כלומר יש לנו אובייקט שיורש פעולות ותכונות שניתן להשתמש ולדרוס באובייקט הבן.

כאשר אנו יורשים אובייקט אבסטרקטי אנו עדיין יורשים רעיון מסויים של מופע עתידי של היורשים ממנו, אלומר אובייקט בסיסי שלא ניתן ליצור ממנו מופעים אך פוטנציאלי לכך.

זממשק הוא הדרך שלנו להגיד מן חתימת פעולות מסויימת, כלומר ממשק של פעולות שמי שיממש אותה יהיה חייב ליישם את הפעולות באובייקט.

כאר אנו ניצור ממשק של פעולות אנו נדע שמי שמממש את הממשק הזה מתייחס בקוד שלו לפעולות הממשק, אמנם עצם המימוש בקוד לא רלוונטי עבורינו, אך הידיעה שאוסף הפעולות ממומש באובייקט מסויים מבהירה לנו שאנו יכולים להשתמש בהם על העצם ללא שגיאה. כלומר יש לעצם זה פעולה מגדירה מתאימה.

לדוגמה יש לנו כמה חברות רכבים שממשים ממשק מסויים של נעילת רכב

אנו יודעין שהפעולה car.lock תסגור את האוטו איכשהו, הפעלה car.windowLock תנעל את חלונות הרכב.

ידוע לנו שגם car מסוג BMW תממש את הממשק שלנו, ולכן נוכל ליצור כמה אובייקטים מסוג car שיממשו את הממשק ונוכל לדעת שאנו משתמשים ברכב שתומך בפעולות אלה ולהסתכל עליהם כממשק האב.