OOP – הגדרות

Object Oriented Programming – OOP

תכנות מונחה עצמים

שיטה תכנותית המאפשרת ליצור תבנית המתארת יישות ולאחר מכן ליצור מהתבנית אובייקטים.

כל OOP אלו פקודות שהינן חלק משפת התכנות.

ב-JavaScript אנו יכולים ליצור אובייקטים. אובייקטים אלו נקראים Literal Objects:

```
const cat1 = { name: “Mitsi”, age: 4 };
function printCat(cat) {
        alert(cat.name + “, “ + cat.age);
}
printCat(cat1);
```

אך בעיה:

אם בהמשך ניצור אובייקט נוסף המתאר את אותה היישות (חתול), אנו יכולים לטעות במאפיינים:

```
const cat2 = { nam: “Kitsi”, age: 4 }; // ← nam is an error!!!
printCat(cat2);
```

אם היינו יוצרים ראשית תבנית ע"י OOP המתארת חתול, ורק לאחר מכן יוצרים אובייקטים מהתבנית, לא היו שום שגיאות באובייקטים.

הנ"ל זה חלק קטן ממה ש-OOP מספק לנו.

השם המקצועי עבור אותה תבנית הוא "מחלקה" – "Class" (לא קשור ל-class של CSS)

class זו מילה שמורה של שפת התכנות שיוצרת את התבנית (המתארת את היישות).

מה אנו עושים בפיתוח OOP:

א. יוצרים תבניות ע"י class-ים.

ב. יוצרים מהן אובייקטים.

פיתוח שאינו OOP נקרא תכנות פרוצדורלי (פרוצדורה זה שם נרדף לפונקציה), כלומר תכנות שאינו OOP מכיל פונקציות, פונקציות, פונקציות. הדבר המרכזי שתמיד בונים בתכנות פרוצדורלי הוא פונקציה.

בפיתוח OOP אנו בונים class-ים (יישויות). זה הדבר המרכזי שבונים.

החשיבה שלנו צריכה לכלול את אותן יישויות.

כל Class בונים בקובץ נפרד. שם הקובץ הינו שם ה-Class עם סיומת ts. (עבור שפת TypeScript)

תמיד – שם ה-Class חייב להתחיל באות גדולה!!!

שינוי חשיבתי:

פונקציה תמיד מתארת פעולה

Class תמיד מתארת יישות

לכן במקום לחשוב ישר על אלו פונקציות (פעולות) יש לנו בתוכנית, עלינו ראשית לחשוב אלו Class-ים (יישויות) יש לנו בתוכנית.

Member

כל דבר הנמצא ישירות בתוך Class

Data Member, נקרא גם Field

משתנה שנמצא ישירות בתוך Class

Method

פונקציה שאנו כותבים בתוך ה-Class.

הפונקציה מבצעת פעולה השייכת ל-Class. היא בעצם מבצעת פעילות של ה-Class.

שדות ה-Class מתארים את המידע של היישות.

פונקציות ה-Class (Methods) מבצעות את הפעולות של היישות.

ראינו שמה-Class אנו יוצרים אובייקט (אחד או יותר).

ע"י האובייקט אנו מגיעים למשתנים שלו ויכולים לכתוב אליהם או לקרוא אותם.

ע"י האובייקט אנו מגיעים לפונקציות שלו (ל-Methods) ויכולים להפעיל אותם.

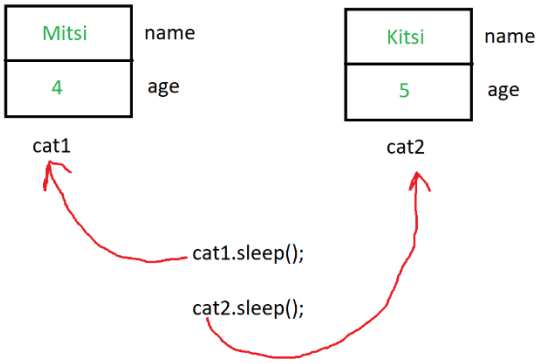
מתודות (Methods) יכולות לגשת לשדות ה-Class ע"י המילה this.

מתודה לא אמורה לקבל כארגומנט את ערכי ה-Data-Members של ה-Class שלה. יש לה גישה אליהם ע"י this.

ברור שמתודה כן יכולה לקבל ארגומנטים אחרים.

this

מילה שמורה הניתנת לשימוש בתוך Method ומתארת את האובייקט הנוכחי – זה שקרא למתודה.



שאלה מראיון עבודה:

מה ההבדל בין Class לבין Object

תשובה:

Class זו תבנית המתארת יישות, ממנה לאחר מכן אנו בונים אובייקטים בזכרון. האובייקט בנוי לפי ההגדרה של ה-Class. האובייקט מכיל את שדות ה-Class (Data Members) ויכול לבצע את הפעילויות של ה-Class (Methods).

constructor

בעברית נקרא "בנאי", בסלאנג נקרא ctor.

מתודה מיוחדת הנכתבת ב-Class, שמופעלת אוטומטית עבור כל אובייקט ברגע יצירת האובייקט.

לא ניתן לקרוא לה אחרת, לא ניתן לקרוא לה בצורה יזומה.

תפקידו: ביצוע פעולות אתחול על האובייקט. לרוב – אתחול שדות האובייקט ע"י ארגומנטים המתקבלים לסוגריים.

Constructor לא מחזיר דבר. גם לא void.

יצירת האובייקט הינה קריאה ל-constructor. הסוגריים של יצירת האובייקט זה בעצם הסוגריים של פונקציית ה-constructor.

Access Modifiers – הרשאות גישה

מילה שמורה שניתן לתת למשתנה או לפונקציה שקובעת מהיכן ניתן לגשת (לקרוא או לכתוב) למשתנה או לפונקציה.

ישנן 3 הרשאות גישה ב-TypeScript:

א. public – ברירת המחדל – נגיש מכל מקום – מתוך המחלקה ומחוץ למחלקה.

ב. private – נגיש רק מתוך המחלקה.

ג. protected – נגיש מתוך המחלקה וממחלקות יורשות (נראה בהמשך מה זה).

readonly

מילה שמורה שניתן לתת ל-Data Member המאפשר להכניס ערך ל-Data Member אך ורק בשורת ההגדרה או ב-ctor.

אין אפשר לשנות ערך זה בפונקציות אחרות במחלקה ואי אפשר לשנות אותו מחוץ למחלקה.

מנגנון Getters / Setters (נקרא גם Property)

אלו שתי פונקציות המאפשרות לחשוף מידע.

Getter – מחזירה את המידע.

Setter – מכניסה לתוכו ערך לאחר בדיקת חוקיות.

למחלקה יש מידע. המידע נמצא ב-Data Members שלה. יכול להיות שקיימת חוקיות לגבי המידע הזה. אם קיימת חוקיות, יכול להיות שמי שמשתמש במחלקה ויוצר ממנה אובייקט, יכניס ערכים שגויים לשדות המחלקה.

המחלקה עצמה חייבת להגן על המידע שלה מפני שימוש לא נכון וערכים לא נכונים.

מנגנון Property מאפשר לבנות פונקציית Getter ופונקציית Setter עבור משתנה private אך להשתמש בהן כאילו אנו משתמשים במשתנה ולא בפונקציה!!!

שיטת ההגנה מתבצעת ע"י תשלובת של Getters & Setters וכוללת שלושה שלבים:

א. יש להגדיר את המשתנה שעלול להכיל מידע שגוי ע"י הרשאת גישה private.

במצב כזה, נהוג להתחיל את שם המשתנה בקו תחתון, לדוגמה _price.

ב. בשפות שלא מכילות מנגנון Property כגון ++C, Java וכדומה, אנו כותבים פונקציה שתפקידה להחזיר ערך של משתנה private בלבד. היא לא מבצעת שום דבר אחר. זאת בכדי שניתן יהיה לקרוא את ערכו מבחוץ, כי המשתנה הינו private.

בשפות המכילות מנגנון Property כגון #C, TypeScript וכדומה, אנו גם כותבים פונקציה כזו, אך ע"י המילה השמורה get. דבר זה מאפשר לנו לקרוא לפונקציה כאילו היא משתנה ולא פונקציה – מה שיותר נוח עבור השימוש בה.

ג. בשפות שלא מכילות מנגנון Property כגון ++C, Java וכדומה, אנו כותבים פונקציה שתפקידה לקבל כארגומנט את הערך שאנו רוצים להכניס לשדה ה-private, לבדוק את חוקיות הערך, ורק אם הוא חוקי – להכניס לשדה הפרטי. אם הוא לא חוקי – להעלות חריגה.

בשפות המכילות מנגנון Property כגון #C, TypeScript וכדומה, אנו גם כותבים פונקציה כזו, אך ע"י המילה השמורה set. דבר זה מאפשר לנו לקרוא לפונקציה כאילו היא משתנה ולא פונקציה – מה שיותר נוח עבור השימוש בה.

דוגמה לשימוש בפונקציית Setter ו-Getter בשפות שאינן מכילות מנגנון Property:

```
class IceCream {

    private _price: number;
    public getPrice(): number { // Getter
        return this._price;
    }
    public setPrice(value: number): void { // Setter
        if(value < 0 || value > 500) {
            throw new Error("Illegal price! Price must be positive, up to 1000.");
        }
        this._price = value;
    }
}

const myIceCream = new IceCream();
myIceCream.setPrice(10); // Calling the set function
document.write("The price is: " + myIceCream.getPrice()); // Calling the get function
```

דוגמה לשימוש במנגנון Property – פונקציית Setter ו-Getter המאפשרות לקרוא להן כאילו הן משתנים:

```
class IceCream {

    private _price: number;
    public get price(): number { // Getter
        return this._price;
    }
    public set price(value: number) { // Setter
        if(value < 0 || value > 500) {
            throw new Error("Illegal price! Price must be positive, up to 1000.");
        }
        this._price = value;
    }
}

const myIceCream = new IceCream();
myIceCream.price = 10; // Calling the set function
document.write("The price is: " + myIceCream.price); // Calling the get function
```

שלושת המרכיבים המרכזיים של OOP

א. Encapsulation (כימוס)

ב. Inheritance (הורשה)

ג. Polymorphism (רב צורתיות)

Encapsulation – כימוס

היכולת ליצור class עם כל הכלים הזמינים לנו עבור יצירת class:

- Data Member
- Methods
- this
- Constructor
- Access Modifiers
- Read Only
- ...

Inheritance – הורשה

היכולת להעביר את כל מה שיש ב-class א', ל-class ב' ללא כתיבת קוד חוזר.

המחלקה הראשונה נקראת מחלקת הבסיס – Base Class או Super Class.

המחלקה השנייה נקראת מחלקה יורשת / מחלקה נגזרת – Derived Class או Sub Class.

שימושי כאשר אנו רוצים לבנות מחלקה אחרת ממחלקה קיימת אך מכילה את כל מה שהמחלקה הקיימת כבר מכילה + דברים נוספים. במצב כזה ניתן לרשת את המחלקה הקיימת ולהוסיף עליה עוד דברים.

ב-TypeScript מחלקה יכולה לרשת מחלקה אחת בלבד.

super

התייחסות למחלקת הבסיס שלנו. אם אנו ב-constructor של המחלקה הנגזרת, עלינו לקרוא ל-constructor של מחלקת הבסיס שלנו.
הפקודה היא: super(...). חובה לקרוא ל-super לפני שימוש ב-this.

הערה:

אם מחלקה יורשת צריכה להכיל פונקציה שמבחינה לוגית צריכה לבצע פעולה שקיימת במחלקת הבסיס – השם של הפונקציה צריך להיות זהה לפונקציה של מחלקת הבסיס. זה לא נכון להמציא שם אחר עבור אותה פעילות לוגית.

לדוגמה, לא נכון להגדיר פונקציית display במחלקת הבסיס, ופונקציה displayMore או displayWildCat או displayDetails וכו' במחלקה הנגזרת. פונקציית הצגת הפרטים בשתי המחלקות צריכה להיות באותו השם. לדוגמה display.

אם שם הפונקציה במחלקה היורשת זהה לשם פונקציה במחלקת הבסיס – זה נקרא "דריסת פונקציה"

או באנגלית: "Function Override"

override

כתיבת פונקציה במחלקה היורשת בעלת שם זהה לפונקציה של מחלקת הבסיס.

Inheritance Tree – עץ הורשה

זהו תרשים של כל המחלקות שבנינו המתאר את קשרי ההורשה שלהן – איזו מחלקה יורשת מאיזו מחלקה.

בתרשים כזה, מחלקת בסיס נמצאות מעל מחלקה נגזרת ויש חץ המפנה מנגזרת לבסיס.

לדוגמה: