הגדרות

הרשאות גישה:

א. private – נגיש רק מתוך ה-class.

ב. public – נגיש גם מתוך ה-class וגם מחוץ ל-class.

ג. protected – נגיש מתוך ה-class ומתוך מחלקות יורשות.

חוק ההורשה

זהו חוק שקובע מתי נכון לבצע הורשה ומתי לא.

החוק הוא חוק סמנטי ואומר את הדבר הבא:

מחלקה ב' יכולה לרשת מחלקה א' אך ורק אם ניתן לומר במילים שמחלקה ב' **היא גם** מחלקה א'.

דוגמאות למצבים חוקיים:

חתול הוא גם חיה. לכן מחלקה המתארת חתול יכולה לרשת מחלקה המתארת חיה כללית.

עובד הוא גם אדם, לכן עובד יכול לרשת אדם.

לקוח הוא גם אדם, לכן לקוח יכול לרשת אדם.

מכונית מרוץ היא גם מכונית, לכן מכונית מרוץ יכולה לרשת מכונית...

דוגמאות למצבים לא חוקיים:

חתול הוא כלב, לכן חתול לא יכול לרשת מכלב.

אדם הוא לא כתובת ולכן אדם לא יכול לרשת כתובת.

חנות היא לא כתובת, לכן חנות לא יכולה לרשת מכתובת.

מחלקה אבסטרקטית – Abstract Class

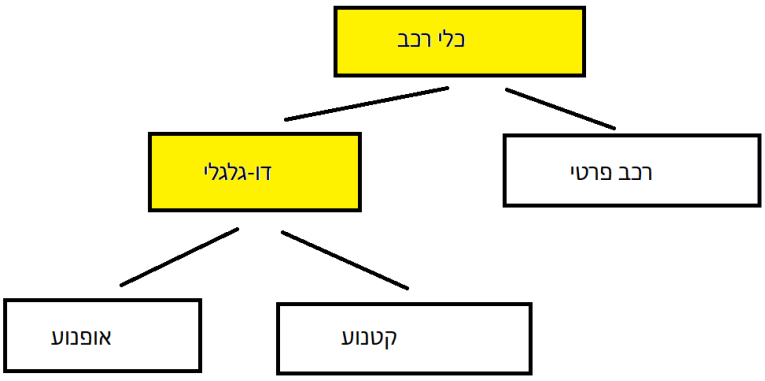
מחלקה המתארת יישות שנועדה אך ורק עבור הורשה ולא נועדה עבור יצירת אובייקט ממנה.

ממחלקה אבסטרקטית אי אפשר לבנות אובייקט, אפשר רק לרשת אותה.

```
abstract class Animal { … }
```

לרוב, מחלקת הבסיס של עץ הורשה הינה אבסטרקטית.

מחלקה אבסטרקטית יכולה לרשת מחלקה אחרת:



כלי רכב ודו-גלגלי הינן מחלקות אבסטרקטיות

פונקציה אבסטרקטית – Abstract Method

חוק: אם לכל היורשים דרושה פונקציה ספציפית, היא חייבת להגיע מהבסיס שלהם.

לדוגמה: אם בקטנוע ואופנוע דרושה פונקציה הבודקת אם יש קסדה, היא חייבת להגיע מדו-גלגלי שהינו הבסיס.

לדוגמה: בקטנוע, אופנוע ורכב פרטי דרושה פונקציה המציגה מהירות, היא חייבת להגיע מכלי רכב שהינו הבסיס.

אם באופנוע דרושה פונקציה שלא קיימת באף מחלקה אחרת, יש לבנות אותה רק באופנוע.

פונקציה אבסטרקטית היא פונקציה שאנו לא יכולים לממש כי אנו לא יודעים מה המימוש שאמור להיות לה.

זה קורה אך ורק במחלקה אבסטרקטית.

פונקציה אבסטרקטית יכולה להופיע רק במחלקה אבסטרקטית.

מחלקה רגילה (שאינה אבסטרקטית) חייבת לדרוס ולממש את כל הפונקציות האבסטרקטיות שהיא קיבלה דרך הורשה.

ממשק – interface

הגדרה של הצהרות על פונקציה אחת או יותר.

לממשק יש שם.

לכן אפשר להתייחס אליו כאל **יכולת** מסוימת המכיל פונקציה אחת או יותר.

הערה: יש שפות תכנות שנהוג להתחיל שם של interface באות I. לדוגמה IPlayer, לדוגמה IWorker וכו'.

ב-TypeScript אין מוסכמה כזו. ב-#C יש.

ב-TypeScript ממשק יכול להכיל גם הצהרות על משתנים. ברוב השפות זה לא ניתן לביצוע.

Interface מייצג משהו מבחינה דקלרטיבית (הצהרתית)

מחלקה יכולה לרשת מחלקה אחת בלבד (ברוב השפות), אך יכולה לממש ממשקים בלי הגבלה.

מחלקה שמממשת ממשק חייבת לדרוס ולממש את כל הפונקציות והמאפיינים שיש לו, אלא אם כן היא אבסטרקטית.

נהוג להתייחס לממשק כאל "חוזה" – Contract כי כאילו חתמנו על חוזה שחייב להיות לנו מה שמוגדר בממשק.

שאלה נפוצה בראיון עבודה: מה ההבדל בין מחלקה אבסטרקטית לבין ממשק. תשובה:

הבדלים טכניים:

- ניתן לרשת מחלקה אחת ומספר רב של ממשקים
- במחלקה אבסטרקטית ניתן להגדיר משתנים ולממש פונקציות, בממשק אפשר רק הצהרות
- במחלקה ניתן להגדיר הרשאות גישה שונות, בממשק לא ניתן כי הכל תמיד public

הבדלים לוגיים:

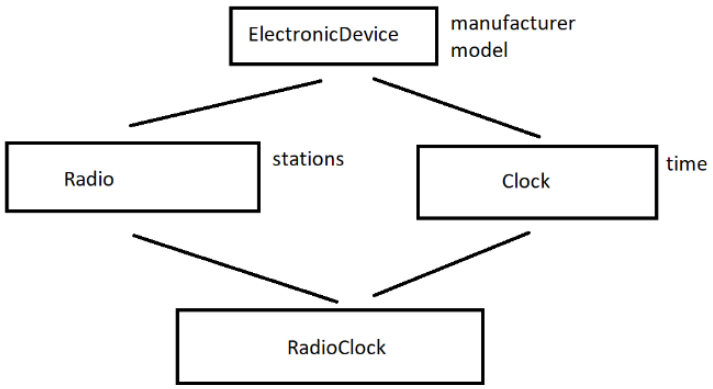
- מחלקה אבסטרקטית נועדה להיות בסיס של עץ הורשה
- ממשק נועד לתאר יכולת אותה אפשר לממש בכל מקום בעץ ההורשה

הורשה מרובה – Multiple Inheritance

זהו מצב בו מחלקה יכולה לרשת מספר מחלקות.

זה קיים בשפות ישנות, לדוגמה בשפת C++.

דבר זה יצר בעיה הנקראת הורשת יהלום:



הבעיה היא שכעת ל-RadioClock יש שני משתנים בשם manufacturer ושני משתנים בשם model כאשר RadioClock אמור להיות יצרן אחד ומודל אחד.

Type (תזכורת)

סוג הנתונים ממנו בנוי משתנה. לדוגמה string, לדוגמה number, לדוגמה Cat.

Value Type

זהו טיפוס נתונים שמשתנה ממנו מכיל את המידע שלו בתוך עצמו.

יש רק שלושה כאלו ב-JavaScript או ב-TypeScript:

א. number

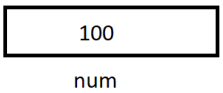
ב. boolean

ג. undefined

Code:

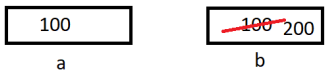
```
let num: number;
num = 100;
```

Memory:



דוגמה לשימוש ב-Value Types:

```
let a: number
a = 100;

let b: number;
b = a;   // Copy a value (100) into b

b = 200;
```



Reference Type

טיפוס נתונים שמכיל מצביע (Pointer/Reference) לאובייקט שבתוכו המידע יושב.

כלומר המידע לא יושב ישירות בתוך המשתנה, אלא בתוך אובייקט שהמשתנה מצביע עליו.

Reference

תא זיכרון שיכול להכיל כתובת של אובייקט.

זהו לא אובייקט!

new (תזכורת)

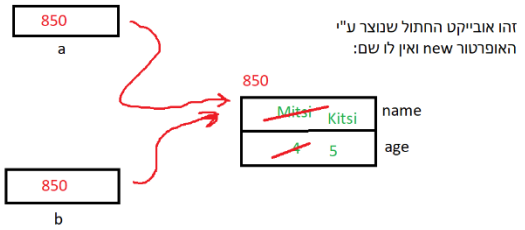
אופרטור שיוצר אובייקט חדש בזיכרון ומחזיר את כתובת האובייקט שנוצר. לרוב כתובת זו נכנסת למשתנה Reference.

דוגמה לשימוש ב-Reference Types:

```
let a: Cat;  // a is a Reference, not an object!

a = new Cat("Mitsi", 4);
```

850

```
let b: Cat;

b = a;

b.name = "Kitsi";
b.age =  5;
```



שאלה נפוצה בראיון עבודה: מה ההבדל בין Value-Type לבין Reference-Type?

תשובה: Value-Type הינו טיפוס נתונים שמשתנה ממנו מחזיק את ה-Value בתוכו. לכן השמה בין משתנים כאלו תגרום להעתקה של ה-Value לעותק חדש, כנ"ל שליחה לפונקציה. לכן שינוי משתנה אחד לא ישפיע על משתנה אחר גם אם היתה השמה ביניהם כי אלו שני משתנים אחרים עם עותקים שונים של ה-Value.

Reference-Type זה טיפוס שמשתנה ממנו הינו אובייקט בזיכרון, אך קיים Reference שמכיל את כתובת האובייקט ולכן מצביע עליו. גישה לאובייקט מתבצעת ע"י ה-Reference שלו. לכן השמה של Reference אחד לאחר תגרום להעתקה של כתובת האובייקט ולא של המידע שאובייקט עצמו, כך שלשני ה-References יש כעת את אותה הכתובת. לכן שינוי של האובייקט ע"י Reference אחד יגרום לשינוי אותו אובייקט ששני ה-References מצביעים עליו ולכן שינוי אותו האובייקט גם עבור ה-Reference השני (שהוא אותו האובייקט בדיוק).

Polymorphism – רב צורתיות

היכולת ליצור Reference מטיפוס בסיס אך להכניס לתוכו אובייקט מטיפוס נגזרת.

Casting – המרה

הפעולה של התייחסות למשתנה מטיפוס A כאילו הוא מטיפוס B באופן זמני – רק למשך פקודת ה-Casting בלבד.

Down Cast

זהו Casting מסוג מיוחד: זהו המרה של אובייקט מסוג בסיס לאובייקט מסוג נגזרת שלו באופן זמני – רק למשך הפקודה בלבד.
אם אין ב-Reference את האובייקט אליו ביצענו Casting אזי תתרחש קריסה בזמן הריצה, או שנקבל undefined, בכל מקרה זו שגיאה כלשהי. לכן אם אנו מבצעים Down Cast זו האחריות שלנו לוודא שאכן זה נכון לביצוע.

Down Cast מבוצע ע"י האופרטור as. לדוגמה: (myAnimal as Cat).color = "White"

האופרטור instanceof

מאפשר לבדוק האם בתוך Reference יושב אובייקט מסוג מסוים. לדוגמה:

```
if(myAnimal instanceof Cat) {
	// Here we can down cast myAnimal to a Cat object because it contains Cat object.
}
```