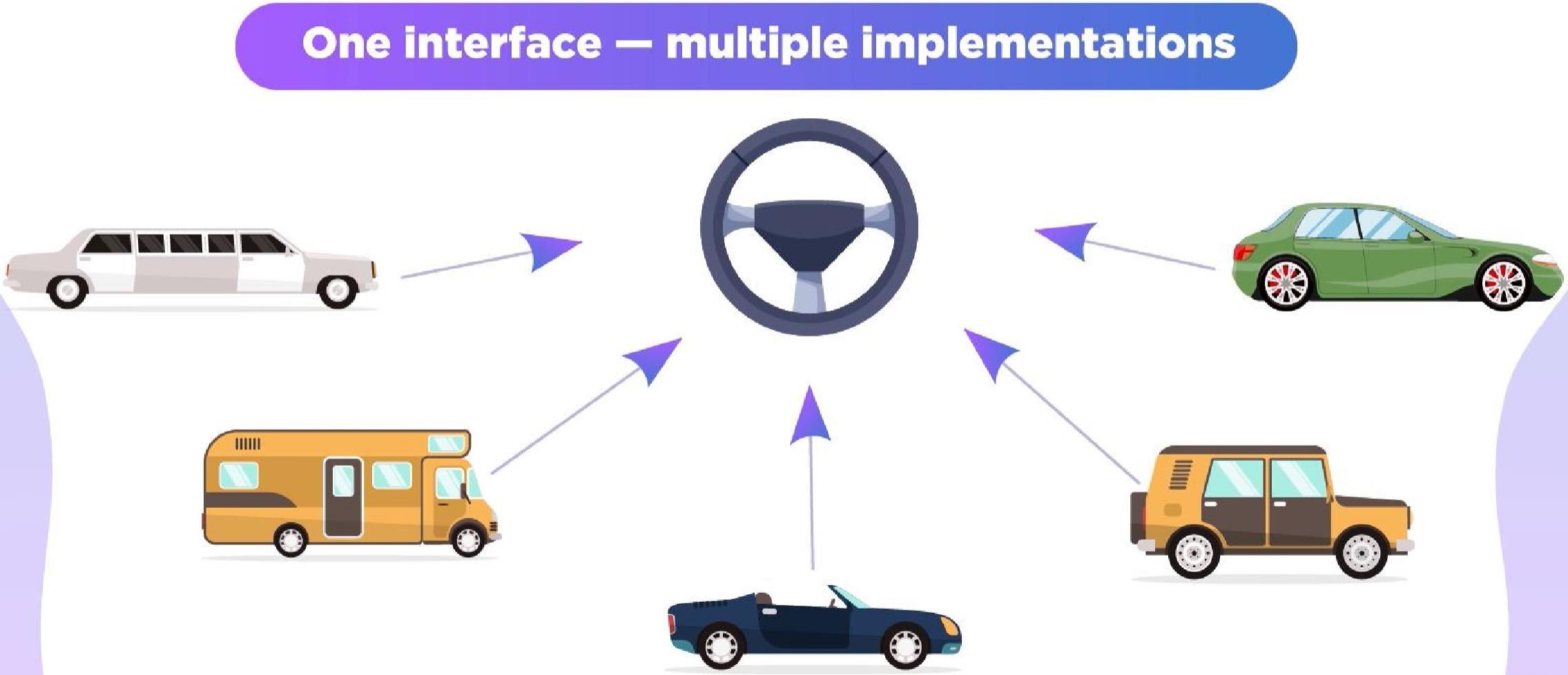
תכנות מונחה עצמים
תרגול 4

- Polymorphism
  - Overloading
  - Overriding
- עקרונות S.O.L.I.D

Polymorphism

Polymorphism



POLYMORPHISM

מוטיבציה

כאשר מסתכלים על בן אדם, ניתן להסתכל עליו מכמה זוויות שונות כמו אבא, סטודנט, בוס ועוד.

פולימורפיזם מאפשר לנו להסתכל על אותו הדבר בכמה צורות שונות בהתאם לסיטואציה הנוכחית, ולבצע פעולה אחת בדרכים שונות.

נרצה להמיר את זה לעולם התכנות, ולאפשר לנו להשתמש במתודות בצורה שונה, בהתאם לאובייקט הנתון או לצורה בה הם נקראות.

POLYMORPHISM

<u>הסבר</u>

פולימורפיזם מאפשר למתכנתים להשתמש במתודות או בפונקציות בצורה שונה בהתאם לסוג או לטיפוס של האובייקט הנתון.

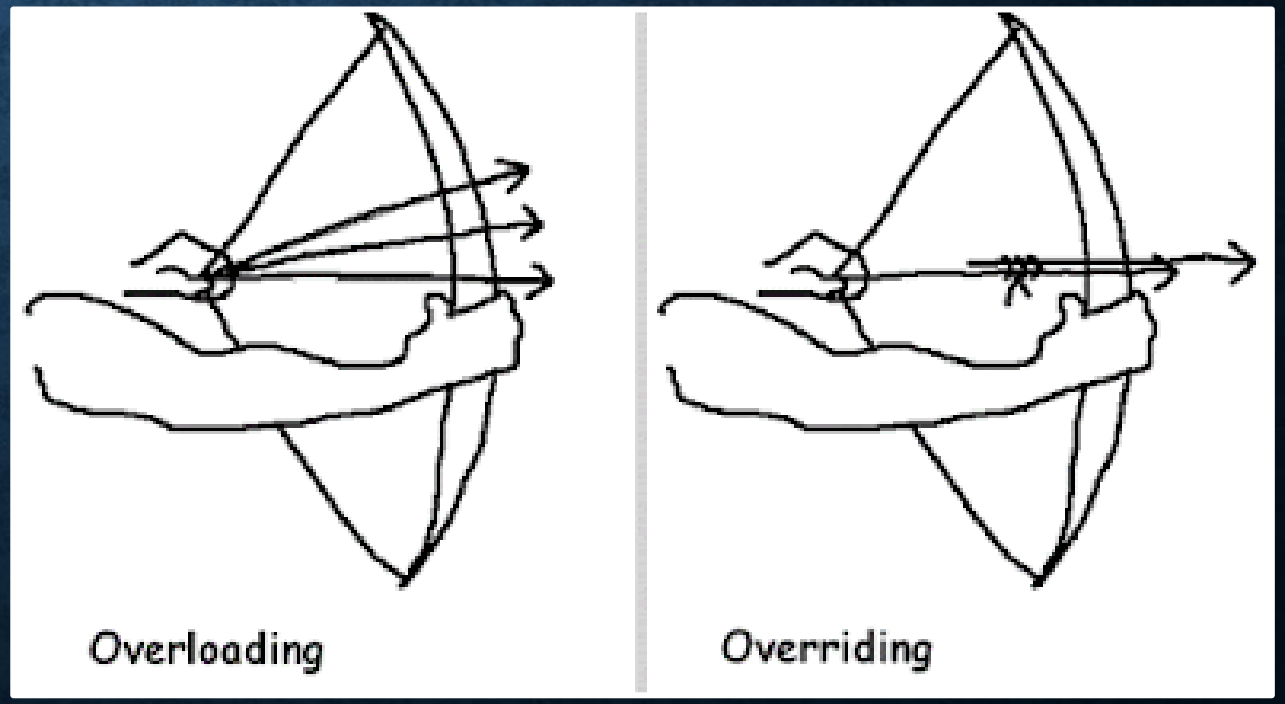
הוא מספק גמישות גדולה והפשטות בעיצוב קוד, ומאפשר יצירת קוד נקי וקל לתחזוקה.

זה מאפשר להתייחס לאובייקטים ממחלקות שונות כאובייקטים של מחלקת-אב משותפת.

POLYMORPHISM

ישנם 2 סוגים עיקריים של Polymorphism:

1. Overloading
2. Overriding



OVERLOADING

העמסת מתודות

- העמסת מתודות היא שיטה המאפשרת להגדיר כמה מתודות תחת אותו שם אך עם פרמטרים שונים (כל שינוי בסוג או בכמות הפרמטרים תופס)
- כאשר נקרא לפונקציה, יתברר בזמן הקומפילציה לאיזו פונקציה בדיוק קראנו בהתאם לפרמטרים שהבאנו לה

```
public static double avg(int[] arr){
    return Arrays.stream(arr).
            mapToDouble(x -> (double)x).sum() / arr.length;
}

public static double avg(double[] arr){
    return Arrays.stream(arr).sum() / arr.length;
}
```

OVERLOADING

העמסת מתודות

- מתודה המבצעת העמסת מתודות חייבת להיות עם כמות שונה של פרמטרים או עם סוג פרמטרים שונה
- ערך ההחזרה של הפונקציה לא משפיע בעת העמסת מתודות, כלומר:

1) גם אם 2 פונקציות עם אותו שם ופרמטרים שונים מחזירות ערך שונה עדיין הם נחשבות כהעמסת מתודות אחת של השנייה
2) אם הפרמטרים זהים ב2 הפונקציות, תהיה שגיאת קומפילציה איפלו אם ערך ההחזרה שונה

OVERRIDING

דריסת מתודות

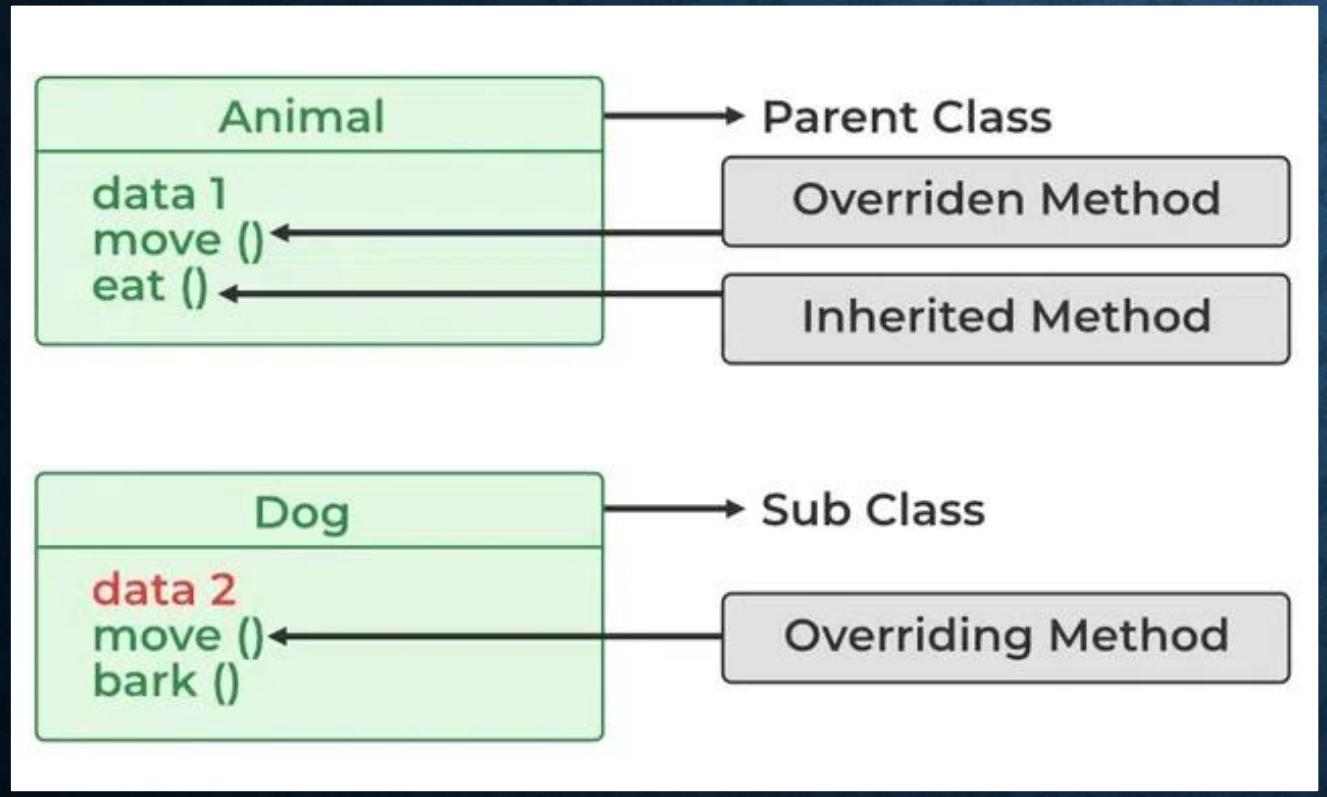
- דריסת מתודות היא שיטה המאפשרת לממש מתודה שכבר מומשה במחלקה שממנה ירשנו.
- כאשר נקרא לפונקציה, יתברר בזמן הריצה מהו סוג האובייקט שעליו קראנו את הפונקציה ובהתאם לכך תיקרא הפונקציה המתאימה לו בשרשרת הירושות.

```
@Override
public double getArea() {
    return 0;
}
```

הערה: מתודות Override חייבות להיות בעלות אותה הצהרה בדיוק (שם, פרמטרים והערך שהפונקציה מחזירה)

OVERRIDING

דריסת מתודות



- מחלקת הבן Dog היורשת ממחלקת האב Animal.
- ניתן לראות בתת-המחלקה Dog הגדרה מחדש (Overriding) של המתודה move. המתודה החדשה מחליפה את המתודה המקורית המוגדרת במחלקת האב Animal בעת יצירת אובייקט מסוג Dog.

OVERRIDING

דריסת מתודות

ישנם כמה סוגים של מתודות שמחלקת הבן לא יכולה לדרוס:

- מתודות המוצהרות כ-static
- מתודות המוצהרות כ-final
- מתודות המוגדרות כ-private

ישנם סוגים של מתודות שמחלקת הבן חייבת לדרוס:

- כל המתודות של interface
- מתודה המוצהרת כ-abstract במחלקת האב (אלא אם מחלקת הבן גם תוצהר כ-abstract)

Polymorphism

נניח שיש לנו מערך שמכיל כמה סוגים של צורות, כיצד נוכל לדעת מה הסוג המדויק של האובייקט בכל אינדקס במערך?

```
Shape[] arr = new Shape[10];
// ...
// fill the array
// ...
for (Shape shape: arr){
    if(shape instanceof Circle)
        System.out.println("It's a circle!");
    else if (shape instanceof Rectangle)
        System.out.println("It's a rectangle!");
}
```

פתרון: **(לא מומלץ)**

נשתמש במילה השמורה instanceof באמצעותה נוכל לברר מהו הסוג המדויק של האובייקט.

SOLID

SOLID

<u>סימפטומים של קוד לא מתוכנן טוב:</u>

- קוד ספגטי – AKA.
- כל שינוי בקוד משפיע על הרבה חלקים בקוד.
- שינוי בקוד משפיע על אזורים לא קשורים בקוד.
- קוד לא פריק - לא ניתן להשתמש בקוד שכבר כתבנו בהקשרים אחרים מאלו שלשמם נכתב הקוד במקור.

האופי המרכזי של הבעיות האלו הוא יותר מידי תלות בתוך הקוד.

עקרונות SOLID באים לתת קווים מנחים שיגרמו לנו להימנע מלכתוב קוד עם הבעיות הנ"ל.

S - Single Responsibility Principle

למחלקה צריך להיות תחום אחריות אחד

Bad

```
class Invoice {
    void calculateTotal() {
        // Logic to calculate total
    }

    void printInvoice() {
        // Logic to print invoice
    }

    void saveToDatabase() {
        // Logic to save invoice to DB
    }
}
```

Good

```
class Invoice {
    void calculateTotal() {
        // Logic to calculate total
    }
}

class InvoicePrinter {
    void printInvoice(Invoice invoice) {
        // Logic to print invoice
    }
}

class InvoiceRepository {
    void saveToDatabase(Invoice invoice) {
        // Logic to save invoice to DB
    }
```

O - Open/Closed Principle

מחלקה צריכה להיות פתוחה להוספות וסגורה לשינויים

Bad

```java
class DiscountCalculator {
    double calculateDiscount(String customerType, double amount) {
        if (customerType.equals("Regular")) {
            return amount * 0.1;
        } else if (customerType.equals("VIP")) {
            return amount * 0.2;
        }
        return 0;
    }
}
```

Good

```java
interface Discount {
    double calculate(double amount);
}

class RegularDiscount implements Discount {
    public double calculate(double amount) {
        return amount * 0.1;
    }
}

class VIPDiscount implements Discount {
    public double calculate(double amount) {
        return amount * 0.2;
    }
}

class DiscountCalculator {
    double calculateDiscount(Discount discount, double amount) {
        return discount.calculate(amount);
    }
}
```

L - Liskov Substitution Principle

פונקציות המשתמשות במשתנים מסוג מחלקת אב, חייבות להיות מסוגלות לפעול בצורה תקינה גם על כל סוגי האובייקטים מסוג הבן מבלי להיות מודעות לסוג האובייקט בפועל

Bad

```java
class Bird {
    void fly() {
        System.out.println("Flying");
    }
}

class Penguin extends Bird {
    @Override
    void fly() {
        throw new UnsupportedOperationException("Penguins can't fly");
    }
}
```

Good

```java
interface Bird {
    void eat();
}

interface FlyingBird extends Bird {
    void fly();
}

class Sparrow implements FlyingBird {
    public void eat() {
        System.out.println("Eating");
    }

    public void fly() {
        System.out.println("Flying");
    }
}

class Penguin implements Bird {
    public void eat() {
        System.out.println("Eating");
    }
}
```

I - Interface Segregation Principle

יש לדאוג לממשקים מצומצמים - לא לאלץ מחלקה לממש ממשק

שאין לה צורך מלא בו

(כלומר, לדאוג לכימוס מרבי של מידע)

Bad

```java
interface Worker {
    void work();
    void manage();
}

class Developer implements Worker {
    public void work() {
        System.out.println("Coding");
    }

    public void manage() {
        throw new UnsupportedOperationException("Developer doesn't manage");
    }
}
```

Good

```java
interface Worker {
    void work();
}

interface Manager {
    void manage();
}

class Developer implements Worker {
    public void work() {
        System.out.println("Coding");
    }
}

class TeamLead implements Worker, Manager {
    public void work() {
        System.out.println("Coding");
    }

    public void manage() {
        System.out.println("Managing team");
    }
}
```

Bad

```java
public class Contact
{
    String name;
    String email;
    String address;
    int telephone;

    public Contact(String name, String email, String address, int telephone)
    {
        this.name = name;
        this.email = email;
        this.address = address;
        this.telephone = telephone;
    }
```

```java
public class Emailer {
    public void sendMsg(Contact c, String msg)
    {
        //..sent message to c.getEmail() ...
    }
}
```

```java
public class Dialler
{
    public void makeCall(Contact c)
    {
        //make call to c.getTelephone() ...
    }
}
```

Good

```java
public interface IEmailable
{
    public String getEmail();
}
```

```java
public interface IDiallable
{
    public String getTelephone();
}
```

```java
public class Contact implements IEmailable, IDiallable
{
    String name;
    String email;
    String address;
    int telephone;

    public Contact(String name, String email, String address, int telephone)
    {
        this.name = name;
        this.email = email;
        this.address = address;
        this.telephone = telephone;
    }
```

```java
public class Emailer {
    public void sendMsg(IEmailable c, String msg)
    {
        //..sent message to c.getEmail() ...
    }
}
```

```java
public class Dialler
{
    public void makeCall(IDiallable c)
    {
        //make call to c.getTelephone() ...
    }
}
```

D - Dependency Inversion Principle

מחלקות high level לא צריכות להשתמש באופן ישיר

במחלקות Low level

Bad

```java
class EmailService {
    void sendEmail(String message) {
        System.out.println("Sending email: " + message);
    }
}

class Notification {
    private EmailService emailService = new EmailService();

    void notifyUser(String message) {
        emailService.sendEmail(message);
    }
}
```

Good

```java
interface MessageService {
    void sendMessage(String message);
}

class EmailService implements MessageService {
    public void sendMessage(String message) {
        System.out.println("Sending email: " + message);
    }
}

class SMSService implements MessageService {
    public void sendMessage(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

class Notification {
    private MessageService messageService;

    public Notification(MessageService messageService) {
        this.messageService = messageService;
    }

    void notifyUser(String message) {
        messageService.sendMessage(message);
    }
}
```

Bad

```java
public class WritingManager
{
    HP_Printer printer;

    WritingManager(HP_Printer printer)
    {
        this.printer = printer;
    }

    public void doWriting(String str)
    {
        printer.print(str);
    }
}
```

```java
public class HP_Printer
{
    public void print(String str)
    {
        // print the string ..
    }
}
```

Good

```java
public interface ICanWrite
{
    public void write(String str);
}
```

```java
public class WritingManager
{
    ICanWrite writable;

    WritingManager(ICanWrite writable)
    {
        this.writable = writable;
    }

    public void doWriting(String str)
    {
        writable.write(str);
    }
}
```

```java
public class HP_Printer implements ICanWrite
{
    @Override
    public void write(String str)
    {
        // print the string ..
    }

}
```

```java
public class Scodix_Printer implements ICanWrite
{
    @Override
    public void write(String str)
    {
        // print the string ..
    }

}
```