הסיבה שנרצה להשתמש ב Strategy Pattern

יש לנו מחלקה אבסטרקטית animal ושתי מחלקות שיורשות ממנה dog ו bird

אני רוצה לתת לאובייקטים שלה את האפשרות לעוף

שיטה 1

קוד לא טוב

פשוט לעשות מתודה של fly ב animal

```
public void fly(){

   System.out.println("I'm flying");

}
```

זה לא טוב בגלל שאנחנו לא רוצים שכל מחלקה שתירש מ animal יהיה את המתודה fly.

שיטה 2

אפשר לדרוס את המתודה בכל מחלקה שאני ארצה, לדוגמא במחלקת dog אפשר לעשות

```
@Override
public void fly(){

   System.out.println("I can't fly");

}
```

אבל זה גם לא קוד טוב.

שיטה 3

אפשר ליצור בכל מחלקה של חיה שאנחנו רוצים שתעוף את המתודה אבל זה סתם קוד כפול.

גם לעשות ממשק שיחייב את כל החיות לממש את המתודה זה הרבה מאוד קוד כפול סתם

```
public interface Flys {

    String fly();

}
```

שתי כללים שחשוב לזכור:

1. אנחנו רוצים להימנע מקוד כפול
2. אנחנו רוצים להימנע מטכניקות שגורמות למחלקה אחת להשפיע על מחלקות אחרות בצורה לא נכונה

Strategy

אנחנו נישאר עדיין עם ממשק שהגדרנו מקודם אבל הפעם ניצור שתי מחלקות שיורשות מאותו ממשק – מחלקה אחת שמייצגת חיות שיכולות לעוף וממחלקה שנייה שמייצגת חיות שלא יכולות לעוף

```
package designpattern.strategy.goodcode;

// The interface is implemented by many other
// subclasses that allow for many types of flying
// without effecting Animal, or Flys.

// Classes that implement new Flys interface
// subclasses can allow other classes to use
// that code eliminating code duplication

// I'm decoupling : encapsulating the concept that varies

public interface Flys {

    String fly();

}

// Class used if the Animal can fly

class ItFlys implements Flys{

    public String fly() {

        return "Flying High";

    }

}

//Class used if the Animal can't fly

class CantFly implements Flys{

    public String fly() {

        return "I can't fly";

    }

}
```

עכשיו נשתמש בממשק הזה בתור מופע של במחלקה של animal, ואז נוכל לשנות דינאמית את המצב שלו – הוא יוכל להיות או מסוג CantFly או מסוג ItFlys

זה טוב כי הרבה מאוד מחלקות עם יכולות טיסה שונות יוכלו לרשת את ה Flys ואז אנחנו יכולים ליצור הרבה סוגי טיסה שונים מבלי להשפיע על המחלקה של animal או כל אחת מהמחלקות שיורשות אותה

זה גם קומפוזיציה –

במקום הורשת יכולות באמצעות מימוש ממשק או הורשה, המחלקה מורכת עם אובייקטים עם היכולות הנכונות לאותו אובייקט.

קומפוזיציה גם נותנת לנו את היכולות של האובייקטים שלי בזמן ריצה – לדוגמא אם התחלתי עם אובייקט שבהתחלה הוא לא יכל לעוף אבל עכשיו אני רוצה לתת לו את היכולות אז אני יכול לשנות בצורה דינאמית את המשתנה של המחלקה.

```java
public class Animal {

    private String name;
    private double height;
    private int weight;
    private String favFood;
    private double speed;
    private String sound;

    // Instead of using an interface in a traditional way
    // we use an instance variable that is a subclass
    // of the Flys interface.

    // Animal doesn't care what flyingType does, it just
    // knows the behavior is available to its subclasses

    // This is known as Composition : Instead of inheriting
    // an ability through inheritance the class is composed
    // with Objects with the right ability

    // Composition allows you to change the capabilities of
    // objects at run time!

    public Flys flyingType;

. . .


// Animal pushes off the responsibility for flying to flyingType

public String tryToFly(){

    return flyingType.fly();

}

// If you want to be able to change the flyingType dynamically
// add the following method

public void setFlyingAbility(Flys newFlyType){

    flyingType = newFlyType;

}
```

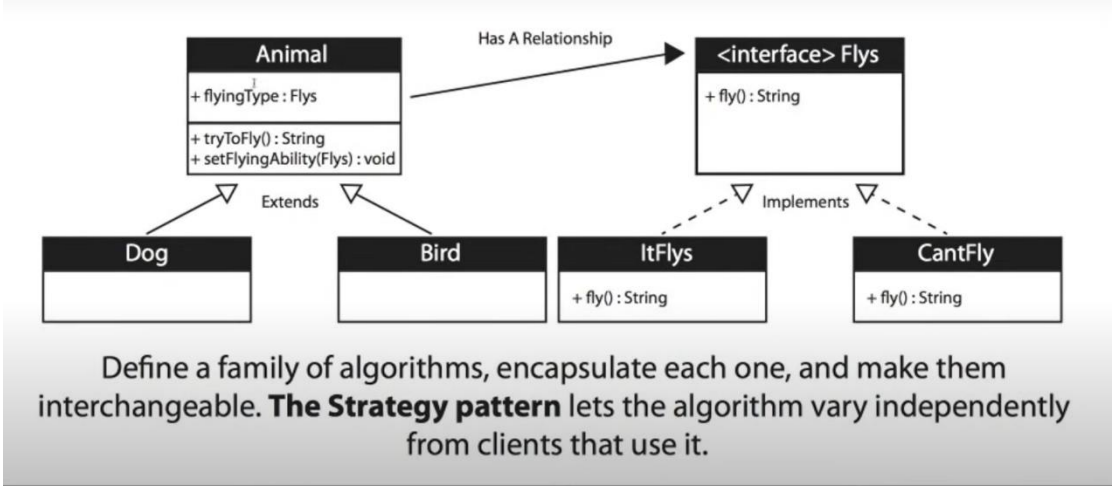
עכשיו אפשר פשוט לאתחל את המשתנה לפי איך שנרצה

```
public class Bird extends Animal{

    // The constructor initializes all objects

    public Bird(){

        super();

        setSound("Tweet");

        // We set the Flys interface polymorphically
        // This sets the behavior as a non-flying Animal

        flyingType = new ItFlys();

    }

}
```

```
public class Dog extends Animal{

    public void digHole(){

        System.out.println("Dug a hole");

    }

    public Dog(){

        super();

        setSound("Bark");

        // We set the Flys interface polymorphically
        // This sets the behavior as a non-flying Animal

        flyingType = new CantFly();

    }
}
```

דוגמאת הרצה + שינוי בזמן הרצה של התוכנה

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

    Animal sparky = new Dog();
    Animal tweety = new Bird();

    System.out.println("Dog: " + sparky.tryToFly()); // Dog: I can't fly

    System.out.println("Bird: " + tweety.tryToFly()); // Bird: Flying High

    // This allows dynamic changes for flyingType

    tweety.setFlyingAbility(new CantFly());

    System.out.println("Bird: " + sparky.tryToFly()); // Bird: I can't fly

}
```



לסיכום:

אנחנו רוצים להשתמש ב Strategy Pattern כשאר:

- אתה רוצה להגדיר מחלקה שתהיה לה התנהגות אחת שדומה להתנהגויות אחרות (חיות שעפות ולא עפות, חיות שעפות מהר ולאט וכו'), בעזרת Strategy אנחנו יכולים ליצור מתי שאנחנו רוצים שיטה חדשה של תעופה ולא נצטרך לשנות הרבה קוד.
- כשנצטרך להשתמש באחת מהתנהגויות רבות באופן דינאמי

יתרונות:

- מפחית שימוש בתנאים
- מפחית שכפול קוד
- מונע משינויי מחלקות לאלץ שינויים במחלקות אחרות
- להסתיר קוד מסובך או סודי ממי שמתממש בו

חסרונות:

- יותר סוגים של מחלקות/ אובייקטים