encapsulation – מהו מנגנון המשלב נתונים וקוד המתמרן אותם נתונים הללו, ומגן הן מפני תבלה או שימוש לרעה מבחוץ. בתכנות מונחה עצמים ניתן לשלב קוד ונתונים יחדו במקרה לו נאמר כי נוצרת מה שנקרא "קופסה שחורה" כאשר קודים ונתונים משולבים בצורה כזו, נוצר אובייקט. האובייקט הוא מה שמתמן ב-encap

polymorphism – מהו מאפיין המאפשר להשתמש באותו שם כדי לפנות שתי בעיות דומות אך יותר אך שונות מבחינה טכנית. מלבד הפולימורפיזם כפי שהיא שיושמה על תכנות מונחה עצמים, היא להשתמש בשם יחיד להגדרת פעולות נפרדות עבור מחלקה. הביצוע של כל פעולה מבוצע "קבע" על פי סוג הנתונים.

ירושה

inheritance – מהו הנהלים שבו אובייקט אחד יכול לרכוש את מאפיינים של אחר. לאחר כיוון, אובייקט יכול לרשת את מאפיינים הבסיסיים של אובייקט אחר ולהוסיף לו תכונות "חדשות". הירושה חשובה מכיוון שהיא מאפשרת במושג הסיווג ההיררכי. השימוש בהיררכיה מעמדית הופך את לרמי המינים הגבוהים לניתנים לניהול.

Scanned with CamScanner

overriding vs overloading *

- overload – שיטה עוסקת במושג שיש שתי שיטות או יותר באותה מחלקה, בשם להה אך עם טיפוסים שונים.

- override – שיטה בירושה שיש שתי שיטות עם אותם טיפוסים, אך "שומים שונים. אותן מהם יתקיים בה class parent, "אב" והשני ב derived class או "נגזרת" child class. "בן"

- override method – לה כאשר childs class עוקפת את אותה שיטה כמו parent class, עם אותם פרמטרים. לה פולימורפיזם.

- overload method – לוה' ההגדרה של מספר שיטות ב-class אחת שלוקחות מספרים וסוגים שונים של פרמטרים. בשונה הנקראות נקבעת בזמן הידור על מסך מספר וסוג' הארגומנטים.

overriding	overloading

class Subject {
public virtual void study()
{
console.writeline("Study all the sub
}
}
class Mathematics: Subject {
public override void study()
{
conside.writeline("Study Mathematics")
}
}
Subjec s = new Mathematics();
s.study();
	class Option {
	public int Sum(int x, int y) {
	return x + y;
	}
	public int Sum(int x, int y, int z) {
	return x+y+z;
	}
	}
	Option o = new Option();
	int sum1 = o.Sum(24,28);
	int sum2 = o.Sum(10,20,30);

Scanned with CamScanner

* Polimorphism – לחלק יישום שונה של אותו סוג של פעולה.

לדוגמא: interface אתן, ~~דוגמה~~ יישומים רבים
מאפשר

- Virtual method – היא שיטה שניתן לעקוף במחלקה נגזרת לשיטה כזו יכולה להיות מימוש סטנדרטי במחלקת הבסיס.

- Abstract method – היא שיטה שמחייבת ליישום אותה בירושה נגזרת יתבצע לבד, שיטה מופשטת לא יכולה לקבל יישום משלה במחלקת הבסיס (הגוף ריק), בניגוד לשיטה הווירטואלית.

- Overriding method – היא שינוי ביישום שיטה שהוגדרה כווירטואלית (במחלקה שעברה בירושה, השיטה תפעל באופן שונה ממחלקת הבסיס)

למה צריך Polimorphism?

פולימורפיזם מאפשר לך לכתוב תוכניות מופשטות יותר, הניתנות להרחבה, אותו קוד משמש לאובייקטים ממחלקות שונות וקריאות הקוד משופרת. פולימורפיזם חוסך מפתח מכתיבה, קריאה וניפוי באגים בהרבה מבני if-else / case-switch.

```
public class Animal
{

  public virtual void Say();
    Console.WriteLine("Say Animal");
  }
}
```

```
public class Dog: Animal
{
  public override void Say
  {Console.WriteLine("Gav");
   base.Say();
  }
```

```
var animal = new Dog();      // Gav
animal.Say();                // Say Animal
```

Scanned with CamScanner

abstract class *

למשל: במציאות שלנו מושגי הבנקאי אין יכולים להגדיר שני ישויות עקרי
לקוח בנק ועובד בנק. כל אחת מהישויות הללו תהיה שונה,
למשל לגבי עובד אנו מגדירים את שמו, וללקוח - סכום בחשבון
לקוח ועובד ירשו שיעורי לקוח ועובדים נפרדים.
לשני ישויות הללו יכול להיות משהו במשותף, למשל שם פרטי
ושם משפחה, פונקציונליות נפוצה אחרת עדיף להגדיר את
פונקציונליות הכללית הזו ל class נפרדת שלה Person.
כלומר, ה classes Client and Employee יורשו מ- class Person.
מכיוון שכל האובייקטים במערכת שלנו מייצגים עובד בנק
או לקוח, לכל נגזר אובייקטים ישירות מ class Person.
לכן כדי לעשות את class Person מסוג abstract.

```
abstract class Person {
    public string Name { get; set; }
    public Person(string name) {
        Name = name;
    }
    public abstract void Display();
}
class Client: Person {
    public int Sum { get; set; }
    public Client(string name, int sum): base(name) {
        Sum = sum;
    }
    public override void Display(){
        console.writeline($"{Name} quantity sum {Sum}");
    }
}class Employee: Person {
    public string Position { get; set; }
    public Employee(string name, string position): base(name) {
        Position = position; }
public override void Display() { Consolo.WriteLine($"{Position} {Name}");
```

Scanned with CamScanner

* Encapsulation – היכולת מנגנון תכנותי מאגד את הקוד ואת הנתונים שהוא מעבד, ומבטל גם הפרעות חיצוניות וגם שימוש לרעה בנתונים. בשפה מונחית עצמים ניתן לשלב נתונים וקוד לקופסה אחורה עצמאית המלווים כל הנתונים והקוד הדרושים נעצרים בתוך תיבה כזו. כאשר קוד ונתונים מקושרים לה לזה בצורה כזו, נוצר אובייקט. בעולם אמרות, אובייקט הוא אלמנט עולמי

```
class Gun {
    private bool isLoaded;
    private void Reload() {
        Console.WriteLine("Заряжаю...");   2
        isLoaded = true;
        Console.WriteLine("Заряжено!");   3
    }
    public void Shoot() {
        if(!isLoaded) {
            Console.WriteLine("Оружие не заряжено!");   1
            Reload();
        }
        Console.WriteLine("Бом Бом");   4
        isLoaded = false;
    }
}

Gun gun = new Gun();
gun.Shoot();
//print 1, 2, 3, 4
```

Scanned with CamScanner

protected internal – может получить доступ с помощью любого кода в assembly и которой объявлен, или внутри производного класса в другом assembly.

любой класс в том же assembly или в любом производном классе – даже если он находится в другом assembly.

Inheritance (הורשה) – הורשה מאפשרת לנו לבנות מחלקות

ה- reuse – שימוש חוזר בקוד ללא צורך לכתוב אותו מספר פעמים.

```
class Base {
    public Base(int x) {
        //...
    }
}
class Derived: Base {
    public Derived(int x, int y): base(x) {
        //...
    }
}
```

```
class Person {
    string firstName;
    string lastName;
public string FirstName {
    get { return firstName; }
    set { firstName = value; }
} public Person(str fN, str lN) { firstName = fN;
public Person(): this(" ", " ") { }
```

"

```
class Employee: Person {
    public double salary;
    public double Salary {
        get { return salary; }
        set { if(value < 0)
                throw new Excepti
            salary = value; } }
    public Employee { Salary = 0 }
    public Employee(str fN, str lN
        :base(fN, lN) { Salary = salary
```

Scanned with CamScanner

Creational Patterns

Abstract Factory – Создает экземпляр нескольких семейств классов

Builder – Отделяет построение объекта от его представления

Factory Method – Создает экземпляр нескольких производных классов

Prototype – Полностью инициализированный экземпляр для копирования или клонирования

Singleton – Класс из которого может существовать только один экземпляр

Structural Patterns

Adapter – Сопоставьте интерфейсы разных классов

Bridge – Отделяет интерфейс объекта от его реализации

Composite – Древовидная структура простых и составных объектов

Decorator – Динамическое добавление обязанностей к объектам

Facade – Единый класс, представляющий всю подсистему

Flyweight – Мелкозернистый экземпляр, используемый для эффективного обмена

Proxy – Объект, представляющий другой объект

Scanned with CamScanner

Behavioral Patterns

Chain of Resp. - Способ передачи запроса между цепочкой объектов

Command - Инкапсулирует запрос команды как объект

Interpreter - Способ включения языковых элементов в программу

Iterator - Последовательный доступ к элементам коллекции

Mediator - Определяет упрощенное взаимодействие между классами

Memento - Захват и восстановление внутреннего состояния объекта

Observer - Способ уведомления об изменениях ряда классов

State - Изменение поведения объекта при изменении его состояния

Strategy - Инкапсулирует алгоритм внутри класса

Template Method - Отложите точные шаги алгоритма до подкласса

Visitor - Определяет новую операцию для класса без изменений

Scanned with CamScanner