

OOP

تتقسم OOP الى اربع مفاهيم أساسية :

- التغليف Encapsulation
- الوراثة Inheritance
- اشكال متعددة Polymorphism
- التجريد Abstraction

التغليف Encapsulation

معنى التغليف هو التأكد من إخفاء البيانات "الحساسة" عن المستخدمين. لتحقيق ذلك ، يجب عليك:

- تعريف Attributes الخاصة بالكلاس على أنها private
- توفير طرق الحصول العامة وتعيينها للوصول إلى قيمة المتغير الخاص وتحديثها

تعرفنا في السابق على Modifiers ومن أجل تحقيق مفهوم Encapsulation سنحتاج الى جعل جميل Attributes من نوع private كم في المثال التالي

```
class Animal{
    private String name;
    private int age;

    Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

ثم سوف نقوم بتوفير وسيلة للحصول على طريقة للوصول للعناصر الخاصة بهذا الكلاس وتسمى هذه الطريقة ب Getters/Setters كما في المثال التالي:

```
class Animal{
    private String name;
```

```

private int age;

Animal(String name, int age) {
    this.name = name;
    this.age = age;
}

public String getName() {
    return name;
}

public int getAge() {
    return age;
}

public void setName(String name) {
    this.name = name;
}

public void setAge(int age) {
    this.age = age;
}
}

```

- دوال getters

- قمنا بتعريف `getName` وظيفتها هي الحصول على قيمة `name`
- قمنا بتعريف `getAge` وظيفتها هي الحصول على قيمة `age`

- دوال setters

- قمنا بتعريف `setName` وظيفتها هي تغيير قيمة `name`
- قمنا بتعريف `setAge` وظيفتها هي تغيير قيمة `age`

من المهم ان يكون Modifiers الخاص بهذه الدوال من نوع `public` لنتمكن من الوصول له من اي موقع في مشروعنا

الوراثة Inheritance

في Java ، من الممكن أن ترث Attributes و methods من كلاس إلى كلاس آخر.

مفهوم Inheritance يندرج تحته ثلاث مفاهيم أساسية

- الكلاس الفرعي subclass (الطفل) - الكلاس الذي يرث من كلاس آخر
- الكلاس الأساسي superclass (الأصل) - الكلاس الموروث منه
- لنرث صفات كلاس نستخدم كلمة `extends`

في المثال أدناه ، ترث كلاس Car (الكلاس الفرعي) attributes و methods من فئة Vehicle (الكلاس الأساسي):

```
class Vehicle{
    void moving(){
        System.out.println("The Vehicle is moving");
    }
}

class Car extends Vehicle{
}
```

كما في المثال الأعلى يمكننا استخدام جميع methods الخاصة بكلاس Vehicle عند انشاء object من نوع Car

```
public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();
        car1.moving();
    }
}
```

المخرج

```
The Vehicle is moving
```

رغم اننا لم نعرف اي method بإسم moving في الكلاس الخاص ب Car لكننا استطعنا استخدام دالة moving و السبب لأنها موروثة من كلاس Vehicle ونستطيع استخدامها.

استخدام super

عند وجود Constructor في الكلاس الأساسي و اردنا وراثه هذا الكلاس فإننا سنحتاج الى استخدام كلمة سوبر في Constructor الخاص بالكلاس الفرعي كما في المثال التالي

```
class Vehicle{
    String speed;
    int yearOfManufacture;
    public Vehicle(String speed, int yearOfManufacture) {
        this.speed = speed;
        this.yearOfManufacture = yearOfManufacture;
    }

    void moving(){
        System.out.println("The Vehicle is moving");
    }
}

class Car extends Vehicle{
    String company;
    public Car(String speed, int yearOfManufacture, String company) {
        super(speed, yearOfManufacture);
        this.company = company;
    }
}
```

يمكننا باستخدام super اسناد القيم الخاصة attributes الكلاس الأساسي في الكلاس الفرعي

اشكال متعددة Polymorphism

مفهوم Polymorphism او تعدد الأشكال، ويحدث عندما يكون لدينا العديد من الكلاسات التي ترتبط ببعضها البعض عن طريق الوراثة.

كما حددنا في الفصل السابق؛ نتيج لنا الوراثة وراثته Attributes و methods من الكلاس الاساسي الى الكلاس الفرعي. يستخدم Polymorphism هذه methods لأداء مهام مختلفة. هذا يسمح لنا بأداء عمل واحد بطرق مختلفة.

على سبيل المثال ، فكر في كلاس اساسي اسمه Animal و يحتوي method تسمى animalSound(). يمكن أن تكون الكلاسات الفرعية من كلاس Animal عبارة عن كلاس Cat و كلاس Dog ولديهم أيضًا method خاصة بنفس الاسم تسمى animalSound():

```
class Animal {
    public void animalSound() {
        System.out.println("The animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    public void animalSound() {
        System.out.println("The dog says: bow wow");
    }
}

class Cat extends Animal {
    @Override
    public void animalSound() {
        System.out.println("The dog says: meow meow");
    }
}
```

يتم استخدام `@Override` للإشارة ان هذه method موجودة مسبقا في الكلاس الأساسي و لكن تم تغييرها بداخل الكلاس الفرعي.

تكمّن قوة مفهوم Polymorphism عند تعريف object لهذه الأنواع كما في المثال التالي:

```
public class Main {
```

```

public static void main(String[] args) {
    Animal animal = new Animal();
    Animal cat = new Cat();
    Animal dog = new Dog();
    animal.animalSound();
    cat.animalSound();
    dog.animalSound();
}
}

```

قمنا بتعريف ٣ متغيرات من نوع Animal و لكن القيم المعطى لها تختلف

- في المتغير الأول تم اسناد له قيمة من نوع Animal
- في المتغير الأول تم اسناد له قيمة من نوع Cat
- في المتغير الأول تم اسناد له قيمة من نوع Dog

وهنا تم استخدام مفهوم Polymorphism بحيث انه نوع Animal يقبل اشكال متعددة للقيم و لكن يجب ان تكون كل القيم عبارة عن كلاسات فرعية لكلاس نوع المتغير.

عند القيام بمناداة دالة `animalSound()` سوف نلاحظ اختلاف القيمة المطبوعة و السبب يرجع الى نوع القيمة المسندة لكل متغير.

```

The animal makes a sound
The dog says: meow meow
The dog says: bow wow

```

التجريد Abstraction

مفهوم التجريد هو اخفاء البيانات الخاصة بشكل كامل و اظهار method فقط، يتم ذلك عن طريق استخدام احدى هذين الفهومين `interface` او `abstract classes`.

استخدام abstract classes

- مفهوم `abstract class`: هي كلاس لا يمكن استخدامه لإنشاء `objects` (للاوصول إليها ، يجب أن يتم وراثته الى كلاسات فرعية).

- مفهوم abstract method: هي method فارغة يتم تعريفها في abstract class ولكن يجب كتابة الكود الخاص فيها في الكلاس الفرعي الذي يرث abstract class

في المثال التالي تم تعريف abstract class باسم Animal وتم تعريف abstract method باسم moving و نلاحظ انها فارغة ولكن تم تعريف هذه method في الكلاسات التي ترث من abstract class Animal

```
abstract class Animal {  
    public void animalSound() {  
        System.out.println("The animal makes a sound");  
    }  
    abstract void moving();  
}  
  
class Dog extends Animal {  
    @Override  
    void moving() {  
        System.out.println("The Dog is Moving");  
    }  
}  
  
class Cat extends Animal {  
    @Override  
    void moving() {  
        System.out.println("The Cat is Moving");  
    }  
}
```

يتم استخدام abstract class بنفس الطريقة الموجودة في Polymorphism ولكن يجب ان ننتبه انه لا يمكن تعريفه باستخدام نوع abstract class

```
public class Main {  
    public static void main(String[] args) {  
        //Animal animal = new Animal(); this Wrong You can't declare it using the abstract class type
```

```

        Animal cat = new Cat();
        Animal dog = new Dog();
        cat.moving();
        dog.moving();
    }
}

```

المخرج

```

The Cat is Moving
The Dog is Moving

```

استخدام interface

الفرق الجوهرى بين **interface** و **abstract class** ان **interface** هي عبارة عن **abstract class** يحتوي فقط على **method** فارغة.

يتم تعريفه بكتابة كلمة **interface** ثم كتابة اسمه ثم كتابة **methods** فارغة
 كما في المثال التالي, تم تغيير **abstract class** السابق الى نوع **interface** و كتابة **method** فارغة بشكل
 طبيعي, لوراثة او استخدام **interface** نقوم بكتابة كلمة **implements** في الكلاس الفرعي (يمكن استخدام اكثر من
interface في الكلاس)

```

interface Animal {
    void moving();
}

class Dog implements Animal {
    @Override
    public void moving() {
        System.out.println("The Dog is Moving");
    }
}

class Cat implements Animal {
    @Override
    public void moving() {
        System.out.println("The Cat is Moving");
    }
}

```



```
}
```

يتم استخدام interface بنفس الطريقة الموجودة في Polymorphism ولكن يجب ان ننتبه انه لا يمكن تعريفه باستخدام نوع interface

```
public class Main {  
    public static void main(String[] args) {  
//Animal animal = new Animal(); Wrong You can't declare it using the interface type  
        Animal cat = new Cat();  
        Animal dog = new Dog();  
        cat.moving();  
        dog.moving();  
    }  
}
```

المخرج

```
The Cat is Moving  
The Dog is Moving
```