مفاهیم شی گرایی در جاوا

در جلسات قبل با نحوه ی کد نویسی در جاوا آشنا شدیم در این بخش می خواهیم به شی گرایی در جاوا بپردازیم.

شی گرایی چیست؟

یک شیوه برنامه نویسی است که ساختار اصلی اجزای آن، شی می باشد. استفاده از این شیوه برنامه نویسی باعث سازماندهی کدها می شود. مفاهیم شی گرایی در تمام زبان ها یکسان است ولی نحوه کاربرد آنها ممکن است متفاوت باشد. این مفاهیم شامل موارد زیر می باشد:

- **کلاس(Class)**
- **شی(Object)**
- **صفت(Attribute)**
- **رفتار(Behavior)**
- **ارث بری(Inheritance)**
- **چندریختی(Polymorphism)**
- **کپسوله سازی(Encapsulation)**
- **انتزاع یا تجرید(Abstraction)**

در ادامه به شرح تک تک مفاهیم شی گرایی در جاوا می پردازیم و از مثال خودرو (car) برای درک کامل مفاهیم استفاده می کنیم.

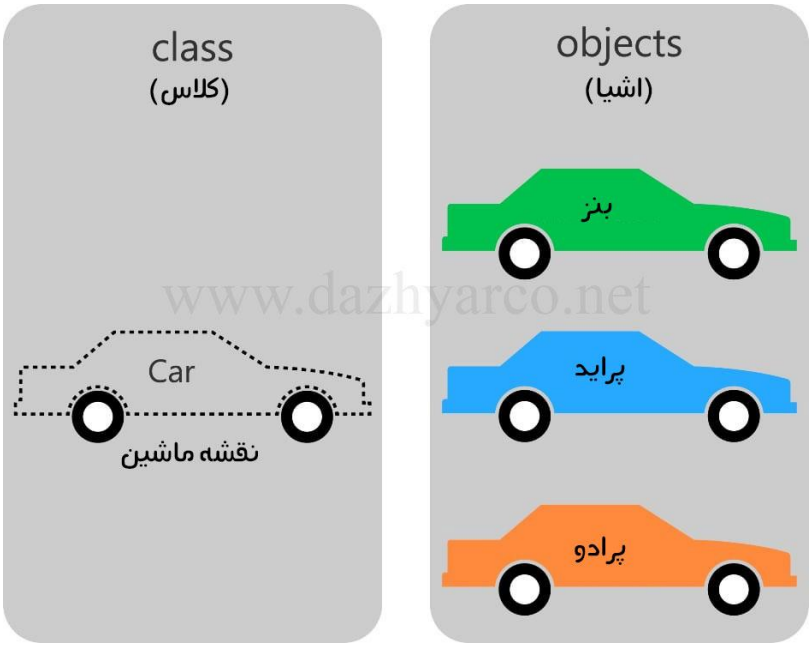
- **کلاس:(Class)**

کلاس یک الگویی است که ما تعریف می کنیم که بتوانیم بر اساس آن یک شی Object را ایجاد کنیم.

- **شی:(Object)**

اگر به عالم واقعیت نگاه کنید در عالمی زندگی می کنید پر از اشیا، مثل ماشین ،صندلی،چتر یا هر چیز دیگری که در دنیای واقعی می بینید یک شی است.

شی (Object)نمونه هایی است که از روی کلاس می سازیم.



- **صفت:(Attribute)**

یک خودرو رنگ دارد ،کمربند دارد ،تعدادی لاستیک دارد ،موتور دارد و الی آخر که به این ویژگی ها صفت می گویند.

- **رفتار:(Behavior)**

همین ماشین می تواند حرکت کند، سرعت بگیرد، ترمز کند، خاموش شود و الی آخر که به آن رفتار یا متد می گویند.

اجزای تشکیل دهنده یک کلاس

کلاس از دو جزء تشکیل شده است:

۱. **Variables یا متغیرها**

۲. **Methods یا رفتارها**

- **انواع متغیرها**

۱. **Class variables:** به زبان ساده متغیرهایی که فقط به درد همین کلاس می خورد و ربطی به خود شی پیدا نمی کند و در داخل متدها نیز قابل تعریف شدن نیست که مشخصه ی آن کلمه کلیدی static می باشد.

۲. **Local variables:** متغیرهایی که داخل متدها تعریف می شوند که به آنها متغیرهای محلی گفته می شود و در متدهای دیگر نمی توان از آنها استفاده کرد.

۳. **Instance variables:** متغیرهایی که متعلق به یک نمونه از کلاس هستند.

جایگاه این اجزا را داخل کلاس خودرو مشخص می کنیم:

```
class Car
{
    //class variable
    private static int status = 1;

    //instance variable
    private int currentspeed = 20;

    public  Car()
     {
      }
    private void  Driving()
      {
       //local variable
        Boolean isDriving = false;
       }
}
```

- **انواع متدها**

متد سازنده (constructor): متدی است که به محضی که از روی کلاس، شی ساخته شد اجرا می شود و باید هم نام با نام کلاس باشد.

```
class Car
{
    //class variable
    private static int status = 1;

    //instance variable
    private int currentspeed = 20;

   //متد سازنده
    public  Car()
     {
```

```
    }

  private void  Driving()
    {
     //local variable
      Boolean isDriving = false;
     }
}
```

نکته:یک کلاس می تواند چندین متد سازنده داشته باشد ولی به شرط (استفاده از مفهوم overloading)

مفهوم Overloading:

اگر یک تابع چندین بار با یک نام ولی با آرگومان های مختلف پیاده سازی شود Overloading گفته می شود.

```
public  Car()
{

}

 تابع با یک آرگومان//
public Car(string name)
{

}

تابع با دو آرگومان//
public Car(int id,string name)
{

}
```

مثال: ایجاد یک کلاس برای حل معادلات درجه ۱و۲ با استفاده از مفهوم overloading

```
public  class EquationSolver
   {
      public static void solveEq(double a, double b)
      {
        double x = - b / a;

         system.out.print("solving equation  a * x + b = 0  \n");

         system.out.format("where  a =%f and b =%f   \n" , a , b);

         system.out.print("solution: ");

         system.out.format("x = %f  \n" , x);
      }

      public static void solveEq(double a, double b, double c)
      {
        double delta = b*b - 4 * a * c ;

         system.out.print("solving equation  a * x ^2 + b *x + c = 0  \n");

         system.out.format("where  a =%f , b =%f , c =%f  \n" , a , b , c);

         system.out.print("solution: ");

         if(delta>0){ //Two Distanced Real Roots

               double x1 = (- b + Math.sqrt(delta)) / (2 * a);

               double x2 = (- b - Math.sqrt(delta)) / (2 * a);

               system.out.format("x1 =%f , x2 =%f  \n" , x1 , x2);

         } else if(delta == 0){ //A Real Double Root

                double x = - b / (2 * a);

                system.out.format("x =%f   \n" ,  x);

         }else{ //No Real Roots

               system.out.println("The Equation has not any Real Roots. \n");

         }
      }

   }
```

استفاده از کلاس تعریف شده در تابع main برنامه:

```
Main(){
        EquationSolver . solveEq(10 , 15);
        EquationSolver . solveEq(1 , -6 , 8);
}
```

نکته: چون متد های درون کلاس **EquationSolver** به صورت static تعریف شده اند برای دسترسی به آنها نیاز به تعریف نمونه یا شی نیست و فقط بانوشتن نام کلاس و دات می توان به آنها دسترسی داشت. و چون متدها با پارامترهای متفاوت تعریف شده اند می توان یک نام یکسان برای آنها نوشت.

ساختن یک شی (object)

مراحل ساخت شی از روی کلاس به ترتیب شامل موارد زیر می باشد:

۱. ابتدا نوشتن نام کلاس
۲. تعریف متغیر برای شی
۳. استفاده از کلمه ی new برای ساخت شی جدید
٤. فراخوانی متد سازنده بعد از کلمه یnew

```
Car car = new Car();
Car car = new Car("benz");
```

- **ارث بری(Inheritance)**

ارث بری یعنی چه؟ یکی از مسائل مهم در شی گرایی در جاوا بحث ارث بری است. اگر در عالم واقعیت هم نگاه کنیم مفهوم ارث بری را به وضوح می بینیم، مثال : در دنیای واقعی ممکن است یک سری خصوصیات و یا رفتارها را از پدرمان به ارث ببریم مثل شکل ظاهری، رفتار،اموال و.....

مثال خودروها:

ما یک کلاس خودرو داشتیم بنام کلاس Car ،کلاس های دیگری را ایجاد می کنیم بنام کلاس benz ، bmv، prid که از کلاس Car ارث بری کرده اند.

کلاس Benz که از کلاس Car ارث بری کرده و می گوید علاوه بر چیزهایی که کلاس پدرم دارد من GPSهم می خواهم اضافه تر داشته باشم.

نکته :وقتی بخواهیم که یک کلاس از کلاس دیگری ارث بری کند از کلمه ی extends استفاده می کنیم.

```
public  class Benz extends Car
  {
     private int gpsstatus;
     public Benz()
     {

     }
  }
```

Superclass: به کلاس Car که کلاس بزرگتر ی است می گویند Superclass .در واقع همان کلاس پدر است.

Subclass: کلاس Benz که از کلاس Car ارث بری می کند به آن Subclass می گویند. در واقع همان کلاس فرزند است.

- **Overriding**

پیاده سازی یک رفتار به ارث برده از superclass و تغییر آن بر اساس نیازهای خود را overriding گویند.

Overriding یعنی دوباره نوشتن؛ به زبان ساده درسته که یک چیزی را از پدرم به ارث بردم ولی مطابق میل خودم عوضش کردم البته اگر **سطح دسترسی** به ما اجازه دهد که این مبحث هم در مطالب بعدی بیان می کنیم.

نمونه آن ، کلاس بنز (Benz)که از کلاس Car ارث برده است و همچنین Override کرده است. یک تابعی قبلا داخل کلاس superclass بوده است بنام drive و brak ؛ که کلاس بنز آن ها

را Override کرده است و بعد گفته **super.drive یعنی من نمی خواهم این تابع را دستکاری کنم هر چیزی که برای پدرم هست به من هم برسد (البته می توانیم دستکاری کنیم و تغییر دهیم)**

```
public  class Benz extends car
  {
     private int gpsStatus;
     public Benz()
      {
       }
    @override
    public void drive()
      {
      super.drive();
       }
    @override
   public void brak()
    {
      super. brak ();
    }
  }
```

- **قوانین مهم override**

۱- لیست آرگومان های تابع دقیقا، همان تعدا آرگومان های تابع override شده باشد. یعنی اگر قرار است تابع را دستکاری کنید باید ببینید تابع چند تا ورودی داشته است همان را استفاده کنید و فقط می توانید دستوراتش را عوض کنید ولی نمی توانید دست به ورودی و خروجی آن بزنید.

۲- Return typeتابع دقیقا همان Return type تابع override شده باشد. یعنی اگر نوع برگشتی تابع از نوع int بود اینجا هم باید از نوع int در نظر بگیریم.

۳- تابعی که Final تعریف شده باشد،قابل override شدن نیست. یعنی اگر اول تابع کلمه ی Final بود کسی حق دستکاری داخل این تابع را ندارد.

٤- تابعی که Static تعریف شده باشد، قابل override شدن نیست ولی قابل دوباره تعریف کردن است.

٥- Constructorها (سازنده ها) قابل override شدن نیست.

- **کپسوله سازی(Encapsulation)**

جدا سازی Variable های (متغیرهای) یک کلاس از دید سایر کلاس ها (Data Hiding) را کپسوله سازی گویند.

موارد مورد نیاز برای دستیابی:Encapsulation

- استفاده از سطح دسترسی Private برای تعریف کردن متغیرهای کلاس
- نوشتن Getter و Setter برای دسترسی به Variable ها

به زبان ساده کپسوله سازی یعنی چیزی که از بیرون به آن دسترسی نداریم، برای مثال یک تلویزیون را در نظر می گیریم. آیا سازنده تلویزیون صفحه پشت تلویزیون را باز نگه می دارد و به مشتری می گوید که این مدارات و آی سی ها چنین کارهایی را انجام می دهد ؟ و برای اینکه مثلا صدا را کم یا زیاد کنی این دو سیم را به هم وصل کن؟ **نه** سازنده تلویزیون یک پکیجی را به صورت کنترل تلویزیون آماده می کند و می گوید که بوسیله این کنترل می توانید چه کارهایی را انجام دهید و مثلا برای کم کردن صدا این دکمه را فشار دهید.

با استفاده از تابع Getter متغیرها را از کاربر یا مشتری می گیرد و با استفاده از تابع Setter یک عملیاتی را انجام می دهد:

```
public  class EncapuslationExample
  {
     private int id;
     private string firstName;
     private string lastName;
     private string email;
     public int getId()
     {
        return id;
     }
     public void setId(int id)
     {
        this.id = id;
     }
     public string getFirstName()
     {
        return firstName;
     }
     public void setFirstName(string firstName)
     {
        this.firstName  = firstName;
     }
  }
```

- **مزایایEncapsulation**

۱) **قابلیت Read only یا Write only کردن متغیرهای کلاس.**

۲) **کنترل کامل داشتن کلاس روی اینکه چه دیتایی داخل متغیرهایش ذخیره شود.**

۳) **شخصی که از کلاس استفاده می کند، اصلا نمی داند چطور دیتا داخل کلاس ذخیره می شود. مثلا اگر نویسنده کلاس تصمیم بگیرد نوع متغیر را عوض کند،کسی که از کلاس او استفاده می کرده،نیازی به تغییر کد ندارد.**

مثال: نوشتن یک کلاس برای جمع و تفریق اعداد مختلط با استفاده از مفاهیم override و Encapsulation

عدد مختلط $z = x + i\,y$

x را عدد حقیقی (Real) و y را عدد موهومی (Imaginary) می گویند.

جمع و تفریق اعداد مختلط:

$$z_1 = x_1 + iy_1 \quad , \quad z_2 = x_2 + iy_2$$

$$z_1 \pm z_2 = (x_1 \pm x_2) + i(y_1 \pm y_2)$$

کلاس اعداد مختلط را با نام complex ایجاد می کنیم.

```
public  class complex{
    private double Real;
    private double Imag;
    public complex(){
            Real = 0;
            Imag = 0;
    }
    public complex(double x){
            Real = x;
            Imag = 0;
    }
```

```
public complex(double x, double y){
        Real = x;
        Imag = y;
}
public double getReal(){
        return Real;
}
public void setReal(double x){
        Real = x;
}
public double getImag(){
        return Imag;
}
public void setImag(double y){
        Imag = y;
}
@override
public string tostring(){ // متد تعریف شده توسط خود جاوا را دوباره تعریف کرده ایم.
        return string.format("%f + %f i", Real, Imag);
}
public complex add(complex z){ // جمع دو عدد مختلط
        return new complex(Real + z.getReal() , Imag + z.getImag());
}
public complex add(double x){ // جمع یک عدد مختلط با یک عدد حقیقی
        return new complex(Real + x , Imag);
}
public complex subtract(complex z){ // تفریق دو عدد مختلط
        return new complex(Real - z.getReal() , Imag - z.getImag());
```

```
    }
    public complex subtract(double x){ // تفریق یک عدد حقیقی از یک عدد مختلط
        return new complex(Real - x , Imag);
    }
  }
```

استفاده از کلاس تعریف شده در تابع main برنامه:

```
Main(){
    Complex z1 = new complex(1 , 2);
    Complex z2 = new complex(3 , 4);
    System.out.format("z1 =%s \n", z1.tostring());
    System.out.format("z2 =%s \n", z2.tostring());
    Complex z3 = z1.add(z2);
    Complex z4 = z1.subtract(z2);
    System.out.format("z3 = z1 + z2 =%s \n", z3);
    System.out.format("z4 = z1 – z2 =%s \n", z4);
}
```

- **انتزاع یا تجرید(Abstraction)**

پیاده سازی نکردن جزییات و فقط مشخص کردن عملکردهای کلاس را Abstraction می گویند.

- **چه زمانی از abstraction استفاده می کنیم؟ زمانی که می دانیم چه کارهایی باید انجام شود اما نمی دانیم چطوری باید انجام شود!**

برای مثال می دانیم که یک قسمت از برنامه ما مربوط به تاکسی تلفنی ها است اما نمی دانیم که چگونه باید این کلاس را پیاده سازی کنیم، پس در اینجا یک کلاس Abstractionتشکیل می دهیم.

```
public abstract class Abstractcar
  {
    private string name;
```

```
    abstract void driving();
    abstract void stop();

    private string getName()
    {
       return getName();
    }
    private void setName(string name)
    {
       this.name = name;
    }
  }
```

- **چطور از کلاس Abstract شی بسازیم؟قبلا روش ایجاد شی را برای شما توضیح دادیم ولی ایجاد شی برای کلاس Abstract فرق دارد. یک کلاس می تواند از کلاسی که به صورت Abstract تعریف شده است ارث بری کند و متدهای آن را به صورت override شده استفاده کند.**

```
  public  class Bmv extends Abstractcar
    {
      @override
      void driving()
       {

       }
      @override
      void stop()
       {

        }
     }
```

- **Interface**

Interface شبیه کلاسی است که فقط شامل Abstract Metod باشد.

```
public interface interfaceExample
{
        abstract void doThis();
         abstract void doThat();
}
```

- **چگونگی استفاده از Interface**

```
public  class  EncapuslationExample  implements interfaceExample
```

- **چندریختی Polymorphism**

قابلیتی که یک شی می تواند اشکال مختلفی به خود بگیرد . در مثال زیر یک شی به نام benz داریم که از کلاس Car و Vehicle و Object ارث بری می کند. در مثال زیر benz می تواند از Vehicle ارث بری کند چون benz از Car ارث بری کرده بود و کلاس Car هم از Vehicle .

```
public  class Vehicle extends Object
public  class Car extends Vehicle
public  class Benz extends Car

polymorphism  //

Benz benz =new Benz ();
Car car = benz;
Vehicle vehicle = benz;
Object object = benz;
```

توضیح مختصری درباره ساختار کدنویسی در جاوا:

ساختار کد نویسی در جاوا

- حساسیت به حروف کوچک و بزرگ (Case Sesentive) مثال: hello و Hello با هم تفاوت دارند.
- نام کلاس ها باید با حرف اول بزرگ شروع شوند. مثال: MainClass
- نام متدها باید با حرف اول کوچک شروع شوند . مثال:doThat()
- نام فایلی که ذخیره می کنیم حتما باید هم نام کلاس باشد.

قواعد نام گذاری Identifiers

- حتما باید با یکی از حروف الفبای انگلیسی (A to Z) یا علامت $ یا _ شروع شوند.
- بعد از اوین حرف، هر ترکیبی از کاراکترها را می توان استفاده کرد.
- بعد Keyword نمی تواند به عنوان Identifier استفاده شود.
- Identifier به حروف کوچک و بزرگ حساس هستند.

مثال:

نام گذاری صحیح: age,$salary,_value,_1

نام گذاری اشتباه: 123abc,-salary

مفهوم پکیج () در جاوا

در جاوا برای جلوگیری از تداخل در اسامی کلاس ها ،اینترفیس ها و جستجوی راحت بین آن ها و همچنین جلوگیری غیر مجاز، از مفهوم پکیج استفاده می شود .

Modifiers

- به کلماتی گفته می شود که می توانند وضعیت و نوع کلاس ها ، متدها و غیره را عوض کنند.
- Access modifiers: برای تعیین سطح دسترسی مورد استفاده قرار می گیرند.

public,private,protected,default

- Non- Access modifiers: برای کارهایی غیر تعیین سطح دسترسی مورد استفاده قرارمی گیرد.

final,abstract,static

- ## Access modifiers

private: فقط در کلاس خود قابل دیدن هستند.

public: در همه جا قابل دیدن هستند.

protected: فقط برای پکیج خود و تمامی Subclass ها قابل دیدن هستند.

default: برای پکیج فقط قابل دیدن هستند.

- ## Non Access modifiers

static: برای تعریف Class method و Class variable مورد استفاده قرار می گیرد.

abstract: برای تعریف کلاس یا متد Abstract به کار می رود .

final: زمانی استفاده می شود که دیگر نمی خواهیم به متغیری یا متدی اجازه تغییر دهیم.

یک مثال دیگر از تعریف کلاس در جاوا

```
public  class Student

{
     //Class variable
     public static int MAXIMUM_SCORS = 20;
    //Instance variable
```

```
        private string name;
        private string LastName;

        private string getFullName()
        {
            //Local variable
            string fullName = name + LastName;
            return fullName;
        }
}
```