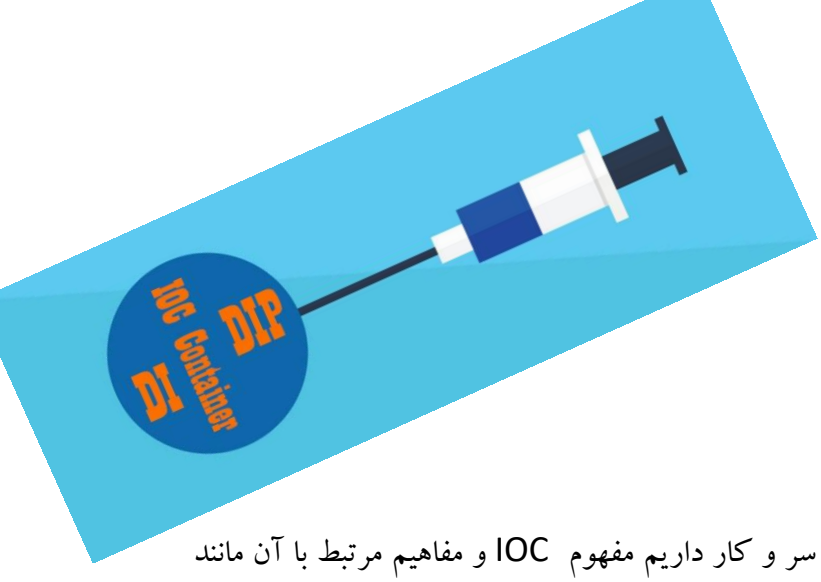

مفهوم IOC و مفاهیم مرتبط با آن.

مهدی کیانی

مقدمه:

یکی از واژه هایی که امروزه در برنامه نویسی زیاد با آن سر و کار داریم مفهوم IOC و مفاهیم مرتبط با آن مانند DIP و DI و IOC Container می باشد. این مفاهیم در خصوص از بین بردن وابستگی بین اشیا صحبت می کنند. وابستگی اشیا در برنامه نویسی دارای مضرات بسیاری است که بحث در خصوص آن در حوصله این مقاله نمی باشد. در این مقاله قصد دارم تا معرفی و پیاده سازی مختصری از این مفاهیم را ارائه دهم.

IOC چیست؟

واژه IOC سر آغاز کلمه های Inversion Of Control می باشد. معنای لغوی این واژه تا حدودی معنای کاربردی آن را نیز مشخص می کند. " معکوس کردن کنترل"

اما مفهوم معکوس کردن کنترل چیست؟ یقینا شما از این مفهوم در زندگی خود استفاده کرده اید.حتی اگر مفهوم آن را تا امروز نمی دانستید. در واقع هرگاه مسئولیت کاری که تا الان بر عهده شما بوده را به شخص دیگری واگذار کنید از مفهوم IOC استفاده کرده اید.

معکوس کردن کنترل به همین معناست. در واقع کنترل شرایط یا کاری را به دیگری واگذار کردن.

اما این مفهوم در برنامه نویسی چگونه است. برای روشن شدن مطلب یک مثال ارائه خوام کرد.

به دستورات زیر دقت کنید:

```
public class Word
    {
        public string Document
        {
            get; set;
        }
        public void Print()
```

```
        {
            Printer p1 = new Printer();
            p1.Print(Document);
        }
    }


public class Printer
    {
        public void Print(string data)
        {
            //TODO: Print data
        }
    }
```

همانطور که مشاهده می کنید کلاس Word برای چاپ کردن محتوای خود نیاز به کلاس Printer دارد.

در واقع مسئول ایجاد کردن شیی از کلاس Printer خود کلاس Word می باشد. به نظر شما این شرایط چه اشکالی دارد؟ این مقاله در خصوص حل مشکلاتی است که این روش کد نویسی دارد.

حال میخواهیم مفهوم IOC را در خصوص این مثال پیاده سازی کنیم. در واقع می خواهیم مسئولیت ایجاد نمونه ای از کلاس Printer را از کلاس Word گرفته و برون سپاری کنیم.

برای این منظور کلاسی به نام PrinterFactory به صورت زیر تعریف می کنیم:

```
public class PrinterFactory
    {
        public static Printer GetPrinter()
        {
            return new Printer();
        }
    }
```

حال کلاس Wordرا به صورت زیر تغییر می دهیم:

```
public class Word
    {
        public string Document
        {
            get; set;
        }
        public void Print()
        {
            Printer p1 = PrinterFactory.GetPrinter();
            p1.Print(Document);
        }
    }
```

همانطور که مشاهده می کنید کلاس Word دیگر مسئول ایجاد نمونه ای از کلاس Printer نمی باشد. بلکه این وظیفه را کلاس PrinterFactory به عهده گرفته است.

DIP چیست؟

پس از درک مفهوم IOC وقت آن رسیده تا مفهوم DIP را نیز بدانیم. کلمه DIP سرآغاز کلمه های **Dependency Inversion Principle** می باشد. واژه Principle در این عبارت به این معناست که DIP یک مفهوم سطح بالاست نه یک الگوی پیاده سازی.

بر اساس مفهوم DIP اشیای سطح بالا نمی بایستی به اشیای سطح پایین وابسته باشند.

اما اشیای سطح بالا و پایین چه هستند؟ یک شی سطح بالا شیئی است که عملکرد آن(یا بخشی از عملکرد آن) به شی یا اشیای دیگری وابسته است. این شی را شی سطح بالا و شی یا اشیایی که شی سطح بالا به آن(ها) وابسته است را شی یا شیای سطح پایین می گویند.

در مثال فوق کلاس Word شی سطح بالا و کلاس Printer شی سطح پایین می باشد. چرا که عملکرد کلاس Word به کلاس Printer وابسته است.

مفهوم DIP به عدم وجود چنین وابستگی هایی اشاره دارد. راهکار DIP این است که هم شی سطح بالا و هم شی سطح پایین هر دو به اشیای انتزاعی وابستگی داشته باشند یا به عبارت دیگر، اشیای درون برنامه نویسی بر اساس مدل های انتزاعی نوشته و پیاده سازی شوند.

اگر به مثال فوق دقت کنید، با اینکه مفهوم IOC در آن دیده می شود ، اما هنوز مفهوم DIP در آن پیاده سازی نشده است. چرا که کلاس Word همچنان به کلاس Printer وابسته است.

برای پیاده سازی مفاهیم انتزاعی در هر زبان برنامه نویسی ابزار های مختلفی وجود دارد و استفاده از هر ابزار بستگی به شرایط برنامه و عملکرد آن دارد.

استفاده از اینترفیس ها در زبان سی شارپ برای این منظور توصیه می گردد.

برای پیاده سازی مفهوم DIP در مثال فوق ابتدا یک اینترفیس به صورت زیر تعریف می کنیم :

```
public interface IPrinter
    {
        void Print(string data);
    }
```

سپس تعارف کلاس های مثال فوق را به صورت زیر تغییر می دهیم :

کلاس Printer :

```
public class Printer : IPrinter
    {
```

```
        public void Print(string data)
        {
            //TODO: Print data
        }
    }
```

کلاس PrinterFactory

```
public class PrinterFactory
    {
        public static IPrinter GetPrinter()
        {
            return new Printer();
        }
    }
```

کلاس Word:

```
  public class Word
    {
        public string Document
        {
            get; set;
        }
        public void Print()
        {
            IPrinter p1 = PrinterFactory.GetPrinter();
            p1.Print(Document);
        }
    }
```

در دستورات فوق ، یک اینترفیس به نام IPrinter تعریف شده است. همانطور که اشاره شد اینترفیس ها در زبان سی شارپ یکی از ابزار های بسیار مفید برای پیاده سازی مفاهیم انتزاعی هستند. اینترفیس IPrinter دارای یک متد به نام Print با یک آرگومان از جنس string می باشد.

از ابتدای ظهور زبان سی شارپ تا نسخه فعلی(۷,۲) اینتفریس ها فقط می توانند دارای تعریف باشند و نه پیاده سازی.پیاده سازی اینترفیس ها درون کلاس ها صورت می پذیرد. ماکروسافت در حال کار بر روی اینترفیس هایی است که دارای پیاده سازی پیش فرض نیز باشند. ممکن است این نوع اینترفیس ها در سی شارپ ۸ معرفی شوند.

کلاس Printer یک پیاده سازی از اینترفیس IPrinter می باشد. حال به کلاس PrinterFactiry دقت کنید. خروجی متد Getprinter از حالت **Printer** به **IPrinter** تغییر کرده است. اما بدنه متد تغییر نکرده است.

توجه داشته باشید که بر اساس مفاهیم شی گرایی می توانیم هر زمان که نیازی به یک اینترفیس داشتیم، یک نمونه از کلاسی که اینترفیس مذکور را پیاده سازی می کند استفاده کنیم.

به عنوان مثال دستور زیر یک دستور رایج و معتبر است : (به طرفین تساوی در دستور زیر دقت کنید.)

```
IPrinter p1 = new Printer();
```

حال اگر به دستورات کلاس Word دقت کنید، مشاهده خواهید کرد که دیگر اثری از کلاس Printer در این کلاس وجود نداشته و به جای آن از اینترفیس IPrinter استفاده شده است :

```
IPrinter p1 = PrinterFactory.GetPrinter();
```

حال کد های مثال ما شرایط مفهوم DIP را مهیا می کنند. چرا که کلاس Word به کلاس Printer وابسته نبوده و هردو کلاس Word و Printer بر اساس مفاهیم انتزاعی(پیاده سازی اینترفیس IPrinter) تعریف شده اند.

اما مثال ما هنوز دارای یک ایراد است. آن هم وابستگی کلاس Word به کلاس PrinterFactory است.

این ایراد توسط DI و پیاده سازی آن حل خواهد شد که در ادامه خواهیم دید.

واژه DI چیست ؟

واژه DI سر آغاز کلمه Dependency Injection می باشد و بر خلاف DIP و IOC که فقط به مفاهیم می پردازند، DI یک الگوی طراحی است. کلمه Injection در این عبارت به این مفهوم است :

اشای مورد نیاز هر شی توسط سرویس هایی در آن تزریق می شوند.به آن تزریق وابستگی می گویند.

برای پیاده سازی DI نیاز به یک شی واسط دیگر داریم که به آن سرویس می گوییم. وظیفه شی سرویس تهیه اشیای مورد نیاز یک شی ثالث بوده و آن ها را درون شی مذکور تزریق می کند.

تزریق وابستگی ها به روش های مختلفی پیاده سازی می شوند که شایع ترین آن ها تزریق توسط کانستراکوتر کلاس می باشد.

کلاس Word را به صورت زیر تغییر می دهیم :

```
public class Word
    {
        private readonly IPrinter printer;

        public Word(IPrinter printer)
        {
            this.printer = printer;
        }
        public string Document
        {
            get; set;
        }
        public void Print()
        {
            printer.Print(Document);
        }
    }
```

همانطور که مشاهده می کنید یک کانستراکتور برای کلاس Word تعریف شده است. این کانستراکتور دارای یک پارامتر از جنس IPrinter می باشد. بنار این در متد Print دیگر نیازی به ایجاد نمونه ای از IPrinter توسط کلاس

PrinterFactory نمی باشد و از متغیر printer که در کلاس تعریف شده و در کانسترکتور مقدار دهی شده است، استفاده می گردد.

با این کار وابستگی کلاس Word به کلاس PrinterFactory از بین می رود. در واقع با این تغییر کلاس Word و کلاس Printer هیچگونه وابستگی به یکدیگر نداشته و کاملا مستقل از هم می باشند.

حال اگر بخواهیم از کلاس Word استفاده کنیم چه باید کرد؟ در هنگام نمونه گیری از این کلاس می بایستی شی IPrinter را به آن ارسال کنیم.

مثلا به صورت زیر :

```
Word w1 = new Word(new Printer());
```

همانطور که مشاهده می کنید ، یک نمونه از کلاس Printer ایجاد و به عنوان آرگومان ورودی کلاسWord ارسال کرده ایم.می توانید نمونه گیری از کلاس Printer را به کلاس PrinterFactory واگذار کنید.

اما در مفهوم DI این عملیات به یک کلاس واسط به نام سرویس واگذار می گردد. برای این منظور کلاس WordService را به صورت زیر تعریف می کنیم :

```
public class WordService
    {
        private readonly Word word;
        public WordService()
        {
            this.word = new Word(new Printer());
        }
        public void Print()
        {
            word.Print();
        }
        public void SetDocument(string document)
```

```
        {
            word.Document = document;
        }
    }
```

همانطور که می بینید کلاس WordService وظیفه مدیریت کلاس Word را بر عهده گرفته است.

حال می توانیم به روش زیر از کلاس WordService استفاده کنیم :

```
static void Main(string[] args)
        {
            var ws = new WordService();
            ws.SetDocument(Console.ReadLine());
            ws.Print();
        }
```

کد های فوق اگرچه تا حدودی قابل قبول هستند اما هنوز قابل بهینه تر شدن می باشند. در واقع می توانیم کلاس Word را نیز به صورت انتزاعی پیاده سازی کنیم. برای این منظور ابتدا یک اینترفیس به صورت زیر تعریف می کنیم:

```
public interface IWord
    {
        string Document
        {
            get; set;
        }
        void Print();
    }
```

حال کلاس Word را به صورت زیر تغییر می دهیم :

```
public class Word : IWord
    {
        private readonly IPrinter printer;
```

```
        public Word(IPrinter printer)
        {
            this.printer = printer;
        }
        public string Document
        {
            get; set;
        }
        public void Print()
        {
            printer.Print(Document);
        }
    }
```

سپس کلاس WordService را به صورت زیر اصلاح می کنیم :

```
public class WordService
    {
        public static IWord GetWord()
        {
            return new Word(new Printer());
        }
    }
```

در نهایت برای استفاده از کلاس Word به صورت زیر می توانیم عمل کنیم :

```
static void Main(string[] args)
        {
            IWord word = WordService.GetWord();
            word.Document = Console.ReadLine();
            word.Print();
        }
```

همانطور که مشاهده می کنید، هیچ اثری از استفاده مستقیم کلاس Word و Printer در استفاده نهایی نمی باشد.این روش شبیه به روشی است که در IOC Container ها استفاده می شود که در ادامه با آن ها آشنا خواهیم شد.

IOC Container ها فریم ورک هایی هستند که تمامی عملیاتی که ما در این مقاله به صورت دستی انجام دادیم را به صورت خودکار انجام می دهند.فریم ورک های زیادی برای این منظور به وجود آمده اند مانند AutoFac ، Unity و ...

خارج از نقاط قوت و ضعف هر یک از فریم ورک ها، همه آن ها دارای ۳ امکان مشترک می باشند.

۱- Register کردن اشیا و پیاده سازی های آن ها

۲- Resolve کردن آن ها هنگام نیاز

۳- مدیریت حیات اشیا

به عنوان یک مثال ساده، فریم ورک Autofac به صورت زیر کار می کند :

```
  var builder = new ContainerBuilder();

builder.RegisterType<Word>().As<IWord>();
builder.RegisterType<Printer>().As<IPrinter>();

var container = builder.Build();

using (var scope = container.BeginLifetimeScope())
      {
        var word = scope.Resolve<IWord>();
        word.Document = Console.ReadLine();
        word.Print();
      }
```

همانطور که مشاهده می کنید سه عملیات Register ، Resolve و مدیریت حیات اشیا در دستورات فوق آمده است.با استفاده از کانتینر ها فقط کافیست که به نوشتن اینترفیس ها و پیاده سازی آن ها فکر کنید و دیگر نیازی به نوشت کلاس های سرویس جهت مدیریت اشیا و حذف وابستگی ها نمی باشد.

پایان.

راه های ارتباط با من

Homepage: http://mkiani.ir
Email: mkiani3000@gmail.com
Telegram channel: @codeway