
بسم الله الرحمن الرحیم

مؤسسه آموزش عالی سنابلا گلبهار
Sanabad Golbahar Institute of Higher Education

طراحی سیستمهای شئ گرا

تهیه کننده: فرزاد زندی

zandi_farzad@yahoo.com

zandi8farzad@gmail.com

آنچه در این درس فرا می‌گیرید:

۱- آشنایی با مفاهیم برنامه‌نویسی شئ‌گرا
۲- آشنایی با مفاهیم کلاس, فیلد, ثابت, خاصیت, اعضای ایستا و ...
۳- آشنایی با مفهوم متد و انواع آن
۴- آشنایی با متد بازگشتی و ویژگی‌های آن
۵- آشنایی با سازنده, مخرب, فیلد فقط خواندنی و سربارگذاری آنها
۶- آشنایی با مفهوم وراثت
۷- آشنایی با متد مجازی, کلاس انتزاعی و رابط
۸- آشنایی با سربارگذاری عملگرها
۹- آشنایی با الگوهای طراحی

آنچه از شما انتظار می‌رود:

۱- شناخت و درک مفهوم شئ‌گرایی
۲- شناخت و درک مفاهیم فیلد, ثابت, خاصیت, اعضای ایستا و ...
۳- شناخت و درک مفهوم متد و انواع آن
۴- توانایی توضیح و کار با متد بازگشتی و ویژگی‌های آن
۵- توانایی پیاده‌سازی سازنده, مخرب, فیلد فقط خواندنی و سربارگذاری آنها
۶- درک مفهوم وراثت و پیاده‌سازی آن
۷- شناخت و درک مفهوم متد مجازی, کلاس انتزاعی و رابط
۸- توانایی پیاده‌سازی سربارگذاری عملگرها

نحوه ارزیابی نمره:

۱- حضور مستمر
۲- فعالیت کلاسی
۳- انجام تمرینات محول شده
۴- ارائه در کلاس
۵- تحویل پروژه

۱۵ نمره

۶- امتحان پایان ترم ۵ نمره

جلسه دوم:

آشنایی با مفاهیم برنامه‌نویسی شئ‌گرا

آنچه در این جلسه می‌خوانید:

۱- سبک‌های برنامه‌نویسی
۲- تفاوت برنامه‌نویسی ساخت‌یافته و شئ‌گرا
۳- کلاس
۴- فضای نام
۵- سطح دستیابی کلاس
۶- نمونه‌سازی از کلاس
۷- دستیابی به اعضای شئ
۸- نام مستعار برای کلاس

سبک‌های برنامه‌نویسی:

۱- برنامه‌نویسی ساخت‌یافته (Structured Programming):

در این سبک برنامه‌نویسی، برنامه بصورت مجموعه‌ای از فعالیتها در نظر گرفته می‌شودکه باید بر روی داده‌ها اجرا شوند. هر کار پیچیده‌ای به چند کار کوچکتر تقسیم می‌شود تا این کارها به راحتی قابل پیاده‌سازی باشند. سپس برای انجام هر کار یک مِتُد نوشته می‌شود. زبان C و Pascal دو نمونه از زبان‌های برنامه‌نویسی ساخت‌یافته هستَند.

مزیّت:

- قابلیت استفاده مجدد از متدها (کُد نویسی کمتر و پیاده‌سازی سریعتر)
- قابلیت نوشتن و تست همزمان متدها توسط اعضای گروه

سبک‌های برنامه‌نویسی:

۲- برنامه‌نویسی شئ‌گرا (Object Oriented Programming):

این نوع سبک، جدیدترین سبک برنامه‌نویسی است که در آن همه چیز به دید شئ نگریسته می‌شود.

برنامه‌نویسان با شناسایی مسأله و موجودیتهای آن، کلاسها را طراحی می‌کنند.

مثال: کلاس انسان:

- **خاصیتها:** رنگ چشم، رنگ پوست، جنسیت، سن، قد، وزن و ... .
- **متدها:** راه رفتن، حرف زدن، دویدن و ... .
- **رویدادها:** عصبانی شدن، شاد شدن و ... .

تفاوت برنامه‌نویسی ساخت‌یافته و شئ‌گرا:

با بررسی موارد زیر می‌توان تفاوت برنامه‌نویسی ساخت‌یافته و شئ‌گرا را تشخیص داد:

- **نهان‌سازی (encapsulation):**

بر اساس این اصل, اعضای یک کلاس در برابر دستیابی‌های غیرمجاز محفوظ نگه داشته می‌شوند.

- **وراثت (inheritance):**

بر اساس این اصل, یک کلاس می‌تواند رفتار یا صفاتی را از یک کلاس به ارث ببرد. (قابلیت استفاده مجدد یکی از مزایای اصلی وراثت است)

- **چندریختی (polymorphism):**

مفهوم چندریختی به قابلیت استفاده از شکلهای مختلف یک نوع داده, بدون پرداختن به جزئیات آن اشاره دارد که در زبانهای برنامه‌نویسی با تکنیک‌هایی نظیر سربارگذاری متدها, سربارگذاری عملگرها, متدهای مجازی و کلاسهای انتزاعی پیاده‌سازی می‌شود.

کلاس:

هر کلاس مجموعه‌ای از ثابت‌ها, خاصیت‌ها, متدها و غیره است که یکبار طراحی می‌شود اما می‌توان چندین نمونه یا شئ از آن ایجاد کرد.

شکل کلّی تعریف کلاس:

```
سطح دسترسی کلاس  class  نام کلاس
{
   اعضای کلاس
}
```

مثال:

```
Internal  class  test1
{
   public  int  x;     //عضو ۱
   public  int  y;    //عضو۲
}
```

فضای نام:

فضای نام ساختاری برای گروه‌بندی کلاس‌های مرتبط به هم است. هر کلاس باید در یک فضای نام تعریف شود.

شکل کلّی تعریف فضای نام:

```
namespace   نام فضای نام
{
    تعریف کلاس یا کلاسها
    تعریف فضای نام یا فضاهای نام
}
```

مثال:

```
namespace   NS1
{
    internal  class  test1
    {
        اعضای کلاس
    }
    …
}
```

نحوه استفاده از فضای نام:

```
using   NS1;
```

نحوه استفاده از کلاس تعریف شده در فضای نام:

```
using   NS1.test1;
```

سطح دستیابی کلاس:

سطح دستیابی کلاس مشخص می‌کند که کلاس تعریف شده چگونه باید در دسترس قرار گیرد.

دو سطح دستیابی قابل استفاده برای کلاس عبارت است از:

- **سطح دستیابی عمومی (public):**

در این سطح دستیابی, کلاس در تمام قسمتهای یک پروژه (چند برنامه مرتبط به هم) قابل استفاده خواهد بود. به عبارت دیگر محدودیت در دستیابی به کلاس وجود ندارد.

- **سطح دستیابی داخلی (internal):**

در این سطح دستیابی, کلاس فقط در برنامه فعلی قابل استفاده خواهد بود.

نکته:

اگر سطح دستیابی کلاس ذکر نشود, (داخلی) internal منظور خواهد شد.

نمونه‌سازی از کلاس:

می‌توان از یک کلاس چندین شئ ایجاد کرد. با ایجاد شئ از کلاس, موجودیتی با همان ساختار کلاس ایجاد می‌شود و می‌توان از اعضای تعریف شده در کلاس مربوطه استفاده کرد. (اعضای ایستا از این قاعده مستثنی هستند)

کلاس الگویی برای ساخت شئ و استفاده از آن است.

شکل کلی نمونه‌سازی شئ از کلاس:

```
نام کلاس    نام شئ  =  new   نام کلاس();
```

مثال:

```
using   NS1;
…
test1   obj1  =  new   test1();
```

دستیابی به اعضای شئ:

پس از ایجاد شئ از کلاس, باید بتوان به اعضای شئِ ایجاد شده دستیابی یافت.

شکل کلی استفاده از شئ:

نام عضو.نام شئ

مثال:

```
obj1.x = 10;
```

نام مستعار برای کلاس:

اگر دو یا چند فضای نام دارای کلاسی با نام مشترک باشند و در کلاس دیگری قصد استفاده از آن کلاسها را داشته باشید, باید برای هر کدام از آن کلاسها یک نام مستعار ایجاد کنید

شکل کلی تعریف نام مستعار:

```
using    نام مستعار برای کلاس  =  فضای نام.نام کلاس;
```

مثال:

```
using   T1 = NS1.test1;
```

مثال

```
using  system;
namespace  example
{
    internal  class  test1
    {
        public  int  x;
        public  int  y;
        public  int  area;
        public  void  f()
        {
            area = x*y;
        }
    }
    internal  class  program
    {
        void  main()
        {
            test1  obj1 = new  test1();
            obj1.y = 10;
            obj1.x = 20;
            obj1.f();
            console.writeline(obj1.area);
        }
    }
}
```

استفاده از فضای نام system

تعریف کلاس داخلی test1

تعریف کلاس داخلی program

تعریف فضای نام example

جلسه سوم:

آشنایی با
فیلد, ثابت, خاصیت و اعضای ایستا

آنچه در این جلسه می‌خوانید:

۱- اعضای کلاس

۲- سطح دستیابی به اعضای کلاس

۳- اعضای ایستا

اعضای کلاس:

تعریف کلاس بدون اعضای آن کامل نمی‌شود. یک کلاس می‌تواند دارای اعضای زیر باشد:

- فیلد (Field)
- ثابت (Constant)
- خاصیت (Property)
- متد (Method)
- سازنده (Constructor)
- مخرب (Destructor)
- اندیس‌ساز (Indexer)

سطح دستیابی به اعضای کلاس:

اعضای کلاس از هر نوعی که باشند, دارای سطح دستیابی‌اند. سطوح دستیابی اعضای کلاس عبارتند از:

- سطح دستیابی عمومی (public):

اعضای با سطح دستیابی عمومی, محدودیتی در دستیابی به آنها وجود ندارد و خارج از کلاس خود نیز قابل دستیابی هستند. (معمولاً متدها و خاصیت‌ها)

- سطح دستیابی خصوصی (private):

اعضای با سطح دستیابی خصوصی, فقط در داخل متدهای عضو کلاس خود قابل دستیابی هستند. (معمولاً فیلدها و ثابت‌ها)

نکته:

اگر سطح دستیابیِ عضوی از کلاس تعیین نشود, به طور پیش فرض خصوصی منظور خواهد شد.

فیلد (Field):

فیلدها همانند متغییرهای معمولی برای ذخیره داده‌ها در حافظه اصلی به کار می‌روند.

شکل کلی تعریف فیلد به صورت زیر است:

سطح دسترسی نوع داده فیلد نام فیلد = مقدار;

مثال:

```
private  int  a=10;
```

اعضای ایستا:

در حالت کلی یک عضو کلاس باید از طریق یک شئ از آن کلاس دستیابی شود. اما اگر اعضای کلاس به صورت ایستا (Static) تعریف شود, می‌توان مستقیماً و بدون ایجاد شئ از کلاس و فقط با استفاده از نام کلاس, از آن عضو استفاده کرد.

برای تعریف یک عضو ایستا, باید واژه کلیدی static را قبل یا بعد از سطح دسترسی عضو به کار برد.:

عضو غیر ایستا

```
namespace NS1
{
    internal class test1
    {
        public int age;
    }
}
test1 obj1 = new test1();
obj1.age = 16;
```

عضو ایستا

```
namespace NS1
{
    internal class test1
    {
        public static int age;
    }
}
test1.age = 16;
```

نکته:

کلاس را نیز می‌توان به صورت ایستا تعریف کرد.

۱- از کلاس ایستا نمی‌توان شئ ایجاد کرد.

۲- تمام اعضای کلاس ایستا باید ایستا تعریف شوند.

ثابت (Constant):

ثابت نوع خاصی از فیلد است که مقدار آن پس از مقداردهی قابل تغییر نیست.

شکل کلی تعریف ثابت به صورت زیر است:

سطح دسترسی const نوع داده ثابت نام ثابت = مقدار;

مثال:

```
private const string str="hello world...!";
```

خاصیت (Property):

برای اینکه بتوان خارج از کُدهای کلاس, فیلدهای با سطح دستیابی خصوصی را مقداردهی و یا مقادیر آنها را بازیابی کرد, باید از خاصیت‌ها استفاده کرد.

شکل کلی تعریف خاصیت به صورت زیر است:

```
سطح دستیابی   نوع داده خاصیت   نام خاصیت
{
        get  { return  نام فیلد;}
        set  { نام فیلد =  value;}
}
```

نکته:

- به طور کلی یک خاصیت, دستیابی به فیلد را مدیریت می‌کند.
- فیلد مرتبط با خاصیت, فیلد پشتیبانی نامیده می‌شود.
- نوع داده خاصیت, باید همنوع با نوع داده فیلد پشتیبانی باشد.
- اگر خاصیتی فقط شامل دستور set باشد, آنرا فقط نوشتنی می‌گویند.
- اگر خاصتی فقط شامل دستور get باشد, آنرا فقط خواندنی می‌گویند.
- اگر خاصیتی هم دستور set و هم دستور get داشته باشد, آنرا خواندنی نوشتنی می‌گویند.

مثال

```
namespace NS1
{
    internal class test1
    {
        private int age;
        public int Age
        {
            get { return age; }
            set
            {
                if ( value <=0 || value>=100 )
                    age = 10;
                else
                    age = value;
            }
        }
    }
}

void main()
{
    test1 obj1 = new test1();
    obj1.Age = console.readline();
    console.writeline(obj1.Age);
}
```

تعریف فیلد خصوصی age

تعریف خاصیت عمومی Age

تعریف کلاس داخلی test1

جلسه چهارم:

آشنایی با مِتُدها

آنچه در این جلسه می‌خوانید:

۱- مِتُد چیست؟
۲- نحوه فراخوانی مِتُد
۳- واژه کلیدی this

مِتُد چیست؟

یکی از اعضای مهم کلاس, متد است که بر حسب پارامترهای خود فعالیتی را انجام می‌دهد. متدها به دو دسته تابع و زیربرنامه تقسیم می‌شوند.

متد مجموعه‌ای از دستورات است که با فراخوانی و تعیین پارامترهای مورد نیاز اجرا می‌شود و در اغلب موارد یک وظیفه خاص را انجام می‌دهد. یک متد یا مقداری را به محل فراخوانی برمی‌گرداند (تابع) یا هیچ مقداری را به محل فراخوانی برنمی‌گرداند (زیربرنامه).

شکل کلی تعریف متد به صورت زیر است:

```
سطح دستیابی متد    نوع داده مقدار برگشتی متد    نام متد(پارامترها)
{
        دستورات بدنه متد;
}
```

نکته:

- متد می‌تواند مقادیری را از طریق پارامترها از محل فراخوانی دریافت کند.
- متد می‌تواند فاقد پارامتر باشد.
- نوع داده مقدار برگشتی متد, تعیین می‌کند که چه نوع داده‌ای باید به محل فراخوانی برگردد.

نحوه فراخوانی مِتُد:

برای استفاده از متد علاوه بر تعریف, باید آنرا اجرا کرد. اجرای متد را فراخوانی متد می‌نامند. فراخوانی متد با نام آن انجام می‌شود.

شکل کلی فراخوانی متد به صورت زیر است:

نام متد(آرگومان‌ها);

نکته:

- هنگام فراخوانی متدها, باید لیست مقادیر آرگومان‌ها متناسب با ترتیب و نوع داده پارامترهای تعریف شده باشد.
- برای برگرداندن مقدار برگشتی متد به محل فراخوانی متد, باید از دستور return در بدنه متد استفاده گردد.
- اگر به جای نوع داده مقدار برگشتی متد از واژه void استفاده شود, متد هیچ مقداری را به محل فراخوانی برنمی‌گرداند و دستور return الزامی نیست.

مثال

```
namespace NS1
{
    internal class test1
    {
        public int sum(int a,int b)
        {
            int n;
            n = a+b;
            return n;
        }
    }
}
```

بدنه متد

تعریف متد sum از نوع int با دو ورودی از نوع int

تعریف کلاس داخلی test1

```
using system;
using NS1;
test1 obj1 = new test1();
int x,y,s;
x = console.readline();
y = console.readline();
s = obj1.sum(x,y);
console.writeline("sum=",s);
```

فراخوانی متد sum با دو آرگومان

تمرین:

۱- برنامه‌ای بنویسید که دو عدد اعشاری را از ورودی دریافت کرده و بزرگترین آنها را نمایش دهد.

(با استفاده از کلاس و متد)

۲- برنامه‌ای بنویسید که دو عدد از ورودی دریافت کرده و حاصل جمع، تفریق، ضرب و تقسیم آنها را محاسبه و چاپ کند.

(با استفاده از کلاس و متد)

واژه کلیدی this:

واژه کلیدی this معمولاً در داخل بدنه متدهای یک کلاس استفاده می‌شود و به کلاس جاری اشاره می‌کند. یکی از کاربردهای این واژه کلیدی, اشاره به فیلدهای کلاس جاری است.

مثال:

```
namespace  NS1
{
    internal  class  test1
    {
        private  int  n=2;
        public  int  mul(int n)
        {
            int  r = n * this.n;
            return r;
        }
    }
}
```

پارامتر متد mul

فیلد n تعریف شده داخل کلاس

جلسه پنجم:

آشنایی با مِتُدها

(ادامه)

آنچه در این جلسه می‌خوانید:

۱- متدهای ایستا

۲- روش‌های ارسال آرگومان‌ها به متد

۳- ارسال با ارجاعِ انواع داده‌های مقداری

متدهای ایستا:

متدهای ایستا به اعضای غیر ایستای کلاس دسترسی ندارند.
متدهای غیر ایستا به تمام اعضای ایستا و غیر ایستای کلاس دسترسی دارند.
تعریف متد ایستا مانند تعریف عضو ایستا است.

مثال:

تعریف یک متد ایستای عمومی

```
public  class  test1
{
        public  static  int  sum(int x,int y)
        {
                return  x+y;
        }
}
```

نکته:
در متدهای ایستا, نمی‌توان از واژه کلیدی this استفاده کرد.

مثال

```
internal  class  test1
{
     static  int  sum(int a,int b)
     {
          return  a+b;
     }
     public  static  int  mul(int a,int b)
     {
          return  a*b;
     }
     public  int  m=10;
     public  static  int  n=20;
     public  static  int  div()
     {
          return  n/m;
     }
}
using  system;
int  x=5,y=8;
test1  obj1 = new  test1();
console.writeline("sum=",obj1.sum(x,y));
console.writeline("mul=",obj1.mul(x,y));
console.writeline("div=",obj1.div());
```

تعریف متد ایستای خصوصی sum

تعریف متد ایستای عمومی mul

تعریف عضو غیر ایستای عمومی

تعریف عضو ایستای عمومی

تعریف متد ایستای عمومی div

تعریف کلاس داخلی test1

```
console.writeline("sum=",test1.sum(x,y));
console.writeline("mul=",test1.mul(x,y));
console.writeline("div=",test1.div());
```

روش‌های ارسال آرگومان‌ها به متد:

متدها می‌توانند اطلاعاتی را از محل فراخوانی از طریق پارامتر دریافت کنند. مقادیر آرگومان‌ها می‌توانند شامل متغییرها (مانند متغییر C) و مقادیر ثابت (مثل مقدار ۶) باشند.

آرگومان‌ها اگر متغییر باشند, به دو روش به متدها ارسال می‌شوند:

۱- ارسال با مقدار:

در این روش, مقادیر آرگومان‌ها در پارامترها کپی می‌شوند و پس از آن, هر تغییری در مقادیر پارامترها به هیچ عنوان تأثیری در مقادیر ارگومان‌ها ندارد. در این روش ارتباط بین آرگومان و پارامتر یک‌طرفه است.

۲- ارسال با ارجاع:

در این روش, هر تغییری در مقادیر پارامترها بر روی مقادیر آرگومان‌ها نیز تأثیر خواهد گذاشت. در این روش ارتباط بین آرگومان و پارامتر دوطرفه است.

ارسال با ارجاعِ انواع داده‌های مقداری:

انواع داده‌های مقداری را نیز می‌توان از طریق ارسال با ارجاع به متدها ارسال کرد. برای این منظور باید از یکی از دو واژهٔ ref و out استفاده کرد.

اگر از واژه ref استفاده شود, حتماً باید به آرگومان, قبل از فراخوانی, مقدار اولیه تخصیص داده شود.

اگر از واژه out استفاده شود, نیازی به مقدار دادن به آرگومان قبل از فراخوانی نیست.

نکته:

در روش ارسال با ارجاع, هم در هنگام فراخوانی متد و هم در هنگام تعریف متد باید از این دو واژه استفاده کرد.

روش‌های ارسال آرگومان‌ها به متد (مثال):

```
internal class  test1
{
        public  static  void  inc1(int  a)
        { a++; }
        public  static  void  inc2(ref  int  a)
        { a++; }
}
int  x=5,y=5;

test1.inc1(x);      فراخوانی متد inc1 با متغییر x
console.writeline(x);                خروجی: ۵

test1.inc2(ref  y);   فراخوانی متد inc2 از طریق ارجاع با متغییر y
Console.writeline(y);                خروجی: ۶
```

روش‌های ارسال آرگومان‌ها به متد (مثال):

```
internal class  test2
{
        public  static  void  rand(out  int  a)
        {
                system.Random  r=new  system.Random();
                a = r.Next(100);
        }
}
int  x;
test2.rand(out  x);
console.writeline(x);
```

فراخوانی متد rand

جلسه ششم:

آشنایی با مِتُدها

(ادامه)

آنچه در این جلسه می‌خوانید:

۱- متد با تعداد پارامتر متغییر
۲- مقدار اولیه برای پارامترها
۳- آرگومان‌های نامدار
۴- سربارگذاری متد
۵- انواع متغییرها
۶- ارسال شئ به متد

متد با تعداد پارامتر متغییر:

برای تعریف متدهایی با تعداد پارامترهای متغییر باید از واژه کلیدی params و یک آرایه یک بُعدی استفاده کرد.

```
internal class test1
{
    public int sum(int len, params int[] array)
    {
        int s;
        for(int i=0; i<len; i++)
            s+=array[i];
        return s;
    }
}

test1 obj=new test1();

console.writeline(obj.sum(4,10,20,30,40));

console.writeline(obj.sum(7,1,3,5,7,9,11,13));
```

مقدار اولیه برای پارامترها:

در تعریف متدها می‌توان برای پارامترهای آن , مقدار اولیه تعیین کرد.

```
internal class program
{
    public static int sum(int a, int b, int c=0, int d=0)
    {
        return a+b+c+d;
    }
}
```

```
console.writeline(program.sum(6));
```
→ ?

```
console.writeline(program.sum(1,3));
```
→ 1+3+0+0 = 4

```
console.writeline(program.sum(1,3,5));
```
→ 1+3+5+0 = 9

```
console.writeline(program.sum(2,4,6,8));
```
→ 2+4+6+8 = 20

```
console.writeline(program.sum(2,4,6,8,9));
```
→ ?

آرگومان‌های نامدار:

در فراخوانی متدها, باید ترتیب نوشتن آرگومان‌ها دقیقاً متناظر با ترتیب پارامترهای تعریف شده باشد. به این حالت آرگومان‌های مکانی می‌گویند.

آرگومان‌های نامدار این محدودیت را کنار می‌گذارند. در این حالت می‌توان با ذکر نام پارامترها ترتیب آرگومان‌ها را به دلخواه تغییر داد.

مثال

```
internal  class  program1
{
      public  static  bool  isFactor(int  val, int  divisor)
      {
            if (val % divisor == 0)
                  return  true;
            else
                  return  false;
      }
}

int  x=8;
int  y=2;
if (program1.isFactor(x,y) == true)
      console.writeline(“2 is factor of 8");

if (program1.isFactor(val:x, divisor:y) == true)
      console.writeline(“2 is factor of 8");

if (program1.isFactor(divisor:2, val:8) == true)
      console.writeline(“2 is factor of 8");

if (program1.isFactor(8, divisor:2) == true)
      console.writeline(“2 is factor of 8");
```

سربارگذاری متد:

در هنگام تعریف متدها, می‌توان دو یا چند متد همنام در داخل یک کلاس تعریف کرد ولی باید امضای (Signature) این متدها با هم متفاوت باشد. این قابلیت را سربارگذاری متد (Method Overloading) می‌گویند.

امضای یک متد توسط لیست پارامترها (تعداد پارامترها, نوع داده پارامترها و ترتیب پارامترها) مشخص می‌شود.

نوع داده مقدار برگشتی متد و نام پارامترها تأثیری در امضای متد ندارند.

وقتی یکی از متدهای همنام فراخوانی می‌شود, با بررسی تعداد آرگومان‌ها, نوع داده آرگومان‌ها و ترتیب پارامترها, متد مناسب انتخاب می‌شود.

نکته:
سربارگذاری یکی از ,متد روش‌هایی است که توسط آن می‌توان چندریختی را پیاده‌سازی کرد.

مثال

```
internal class program2
{
    public static int SquareMethod(int n)
    {
        return n*n;
    }

    public static double SquareMethod(double n)
    {
        return n*n;
    }

    public static double SquareMethod(double n, double m)
    {
        return m*n;
    }
}

console.writeline(program2.SquareMethod(3));

console.writeline(program2.SquareMethod(3.5));

console.writeline(program2.SquareMethod(3.5,8.4));
```

انواع متغییرها:

- **متغییر محلی:**

متغییرها و پارامترهایی که در بدنه یک متد تعریف می‌شوند, متغییرهای محلی (Local Variable) نامیده می‌شوند و فقط در همان متد قابل استفاده هستند و

پس از خاتمه یافتن متد, از بین می‌روند. این متغییرها قبل از استفاده باید توسط برنامه‌نویس مقداردهی شوند. در غیر اینصورت با خطا مواجه می‌شود.

- **متغییر محلی ضمنی:**

متغییر محلی ضمنی (Implicity Local Variable) متغییری است که نوع داده آن پس از مقداردهی مشخص می‌شود.

شکل کلی تعریف متغییر محلی ضمنی به صورت زیر است:

```
var  نام متغییر = مقدار ;
```

```
var  a = 400;
var  ch = 'X';
var  s = "This is a text";
```

```
معادل  int  a = 400;
معادل  char  ch = 'X';
معادل  string  s = "This is a text";
```

ارسال شئ به متد و برگرداندن شئ توسط متد:

اشیاء را می‌توان به سادگی به عنوان پارامتر به متد ارسال کرد.

همچنین می‌توان یک شئ را به عنوان خروجی برگرداند.

مثال

```
internal class  program4
{
    public  int  a,b;
}

internal  class  program5
{
    public  void  show(program4  t1)
    {
        console.writeline(t1.a + t1.b);
    }
}

program4  obj1 = new program4();
obj1.a = 10;
obj1.b = 20;

program5  obj2 = new program5();
obj2.show(obj1);
```

تمرین:

۱- برنامه‌ای بنویسید که شعاع دایره‌ای را از ورودی دریافت کرده و محیط و مساحت آنرا محاسبه و در خروجی نمایش دهد.

۲- متدی با نام ABS را دوبار سربارگذاری کنید به نحوی که قدر مطلق پارامتر ورودی خود را برگرداند. (پارامتر ورودی می‌تواند از نوع int و float باشد)

۳- برنامه‌ای بنویسید که نام کارمند, تعداد ساعاتی که کار کرده و دستمزد ساعتی وی را از ورودی گرفته و حقوق وی را محاسبه و چاپ کند.

تمرین:

۴- برنامه‌ای بنویسید که یک حرف از ورودی گرفته, اگر جزو حروف کوچک بود آنرا به حروف بزرگ, و اگر جزو حروف بزرگ بود آنرا به حروف کوچک تبدیل کند.

۵- برنامه‌ای بنویسید که یک عدد از ورودی به عنوان پارامتر گرفته و تشخیص دهد عدد آرمسترانگ است یا خیر.

(عدد آرمسترانگ عددی است که حاصل جمع مکعب ارقامش برابر خود عدد است)

مثال $153=1^3+5^3+3^3$

تمرین:

۶- عملکرد زیربرنامه زیر را شرح دهید.

```
public static void swap(ref int x, ref int y)
{
        int temp = x;
        x = y;
        y = temp;
}
```

جلسه هفتم:

مِتُد بازگشتی

آنچه در این جلسه می‌خوانید:

۱- تعریف متد بازگشتی

۲- ویژگی‌های متد باگشتی

تعریف متد بازگشتی:

متد بازگشتی, نوع خاصی از متد است که توسط آن می‌توان به راحتی بسیاری از مسائل به ظاهر پیچیده را حل کرد.

متد بازگشتی (Recursive Method), متدی است که خودش را به طور مستقیم و یا غیرمستقیم فراخوانی می‌کند.

در روش مستقیم, متدی مثل M در داخل خودش,دستور فراخوانی به خودش دارد.

در روش غیرمستقیم, متدی مثل N متدی مثل L را فراخوانی می‌کند و متد L نیز در داخل خود متد N را فراخوانی می‌کند.

- **نکته:**

مسائلی را با متد بازگشتی حل می‌کنند که حالت n اُم آن به کمک حالت n-1 اُم قابل حل باشد.

تعریف متد بازگشتی:

هر متد بازگشتی نیاز به دو نقطه دارد:

۱- نقطه فراخوانی:

این نقطه, دستوری است که مجدداً متد را با مقداری کمتر و یا بیشتر فراخوانی می‌کند.

۲- نقطه بن‌بست:

این نقطه, دستوری است که با یک شرط مشخص, باعث اتمام فراخوانی‌ها می‌شود. اگر این نقطه وجود نداشته باشد, متد مرتباً خودش را فراخوانی می‌کند و پس از مدتی خطای زمان اجرای StackOverflow Exception رخ می‌دهد و اجرای برنامه متوقف می‌شود.

ویژگی‌های متد بازگشتی:

- متدهای بازگشتی را به راحتی می‌توان با دستورات کمتری نسبت به حالت غیربازگشتی پیاده‌سازی کرد.
- در متدهای بازگشتی به علت استفاده از حافظه پشته که مراحل فراخوانی را نگهداری می‌کند, اتلاف حافظه نسبت به حالت غیربازگشتی بیشتر است.
- به علت فراخوانی‌های مکرر در متدهای بازگشتی, سرعت اجرا نسبت به متدهای غیربازگشتی کمتر است.

رابطه بازگشتی دنباله فیبوناچی:

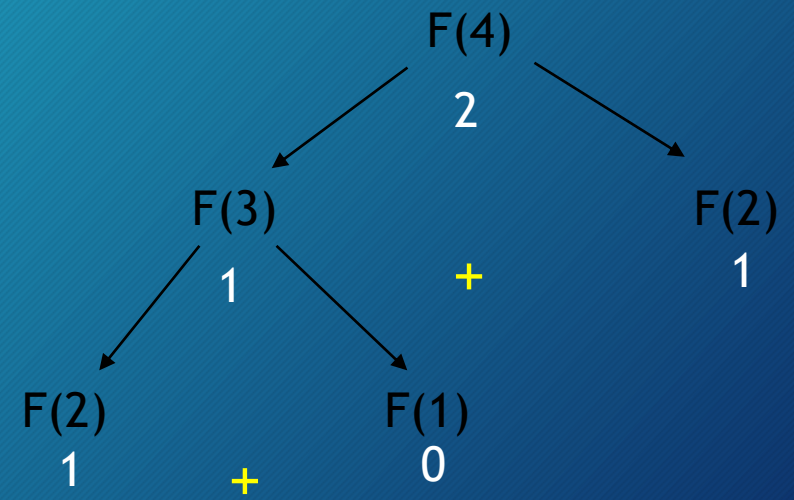
در دنباله فیبوناچی, هر جمله,جمع دو جمله قبل از خود است و $a_1=0$ و $a_2=1$ است.
چند جمله از این دنباله به صورت زیر است:

$$F(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ F(n-1) + F(n-2) & n > 2 \end{cases}$$

- **مثال:**

نحوه محاسبه جمله چهارم دنباله فیبوناچی از طریق رابطه بازگشتی فوق به صورت زیر است:

F(4) = F(3) + F(2) = (F(2) + F(1)) + F(2) = (1 + 0) + (1) = 2



```
internal class program1
{
    public static int fibo(int n)
    {
        if (n==1)
            return 0;
        else if (n==2)
            return 1;
        else
            return fibo(n-1) + fibo(n-2);
    }
}

int n;
console.readline(n);
console.writeline(program1.fibo(n));
```

مثال:
دنبالهٔ فیبوناچی

```
internal class program2
{
    public static int fibo(int n)
    {
        int a1=0, a2=1, a3=0;
        if (n==1)
            return a1;
        else if (n==2)
            return a2;
        else
        {
            for (int i=3; i<=n; i++)
            {
                a3 = a1 + a2;
                a1 = a2;
                a2 = a3;
            }
            return a3;
        }
    }
}

int n;
console.readline(n);
console.writeline(program2.fibo(n));
```

رابطه بازگشتی فاکتوریل عدد صحیح:

فاکتوریل عدد صحیح و مثبت به صورت زیر محاسبه می‌شود:

n! = n * (n-1) * (n-2) * … * 1

6! = 6 * 5 * 4 * 3 * 2 *1 = 720

رابطه بازگشتی محاسبه فاکتوریل عدد n به صورت زیر است:

$$F(n) = \begin{cases} n * F(n-1) & n > 1 \\ 1 & else \end{cases}$$

- **مثال:**

نحوه محاسبه 3! از طریق رابطه بازگشتی فوق به صورت زیر است:

F(3) = 3 * F(2) = 3 * (2 * F(1)) = 3 * (2 * 1) = 6

F(3)

↓

3 * F(2)

↓

2 * F(1)

↓

1

```
internal class program3
{
    public static int fact(int n)
    {
        if (n>1)
            return ( n * fact(n-1) );
        else
            return 1;
    }
}

int n;
console.readline(n);
console.writeline(program3.fact(n));
```

مثال:
فاكتوريل عدد n

```
internal class program4
{
    public static int fact(int n)
    {
        int f=1;
        for (int i=2; i<=n; i++)
            f =* i;
        return f;
        }
    }
}

int n;
console.readline(n);
console.writeline(program4.fact(n));
```

رابطه بازگشتی لگاریتم عدد صحیح و مثبت در پایه ۲:

رابطه بازگشتی محاسبه لگاریتم عدد n در پایه ۲ به صورت زیر است:

$$F(n) = \begin{cases} F\left(\frac{n}{2}\right) + 1 & n > 1 \\ 0 & else \end{cases}$$

- مثال:

$\log_2 n = k$ $\qquad$ $n = 2^k$

$F(8) = F(4) + 1 = (F(2) + 1) + 1 = ((F(1) + 1) + 1) + 1 = ((0 + 1) + 1) + 1 = 3$

مثال:
لگاریتم عدد n

```
internal class program5
{
    public static int log(int n)
    {
        if (n>1)
            return ( log(n/2) + 1 );
        else
            return 0;
    }
}

int n;
console.readline(n);
console.writeline(program5.log(n));
```

رابطه بازگشتی مسأله برج هانوی:

در مسأله برج هانوی سه میله (برج) و n دیسک با قطرهای متفاوت روی اولین میله (میله A) وجود دارد. دیسک‌ها به ترتیب نزولی قطرها روی اولین میله از پایین به بالا چیده شده اند.

هدف, انتقال تمام دیسک‌ها از میله A به میله C با رعایت محدودیت‌های زیر است:

۱- دقیقاً همان ترتیب دیسک‌ها در میله A در میله C پدید آید.

۲- در هر بار انتقال فقط یک دیسک می‌تواند جابجا شود.

۳- در هیچ مرحله‌ای نمی‌توان یک دیسک بزرگتر را روی دیسک کوچکتر قرار داد.

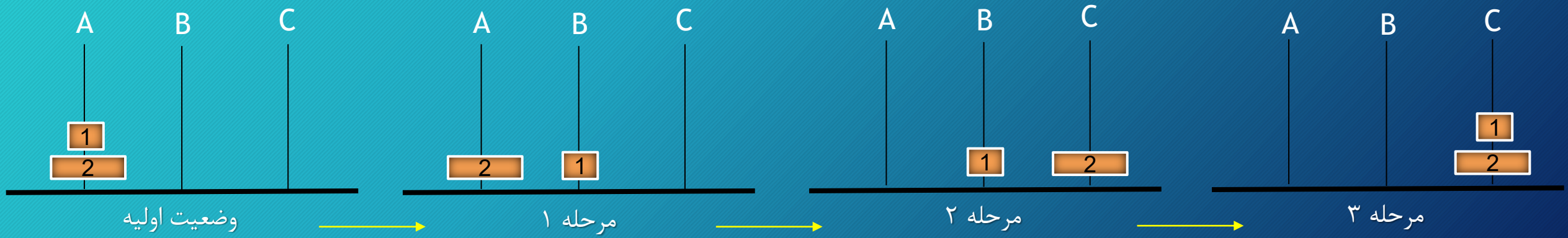
۴- می‌توان از میله B به عنوان میله کمکی استفاده کرد.

رابطه بازگشتی مسأله برج هانوی:

اگر n = 1 باشد, مسأله خیلی ساده است و تنها با یک جابجایی (بدون استفاده از میله B) حل می‌شود.

اگر n = 2 باشد, به سه جابجایی نیاز است.

در حالت کلی, برای حل مسأله برج هانوی با n دیسک, به 2^n-1 جابجایی نیاز است و برای n های بزرگ حل مسأله به کمک کامپیوتر ممکن است ساعت‌ها زمان ببرد.



مثال:
برج هانوی

```
internal  class  program6
{
    public  void  hanoi(int  n, char  source, char temp, char destination)
    {
        if (n==1)
            console.writeline(“Move disk ”, source,” to “, distination, “\n”);
        else
        {
            hanoi(n-1, source, destination, temp);
            console.writeline(“Move disk “, source, “ to “, destination, “\n”);
            hanoi(n-1, temp, source, destination);
        }
    }
}

int  n;
console.readline(n);
console.writeline(program6.hanoi(n, ‘A’, ‘B’, ‘C’));
```

جلسه هشتم:

اعضای کلاس

(سازنده , مخرب , فیلد فقط خواندنی , اندیس‌ساز)

آنچه در این جلسه می‌خوانید:

۱- سازنده
۲- سازندهٔ پارامتردار
۳- سازندهٔ ایستا
۴- سربارگذازی سازنده
۵- مخرّب
۶- فیلد فقط خواندنی
۷- اندیس‌ساز

سازنده:

سازنده یکی از اعضای کلاس است که در زمان ایجاد شئ, اعضای داده‌ای را مقداردهی می‌کند.

سازنده هم‌نام با نام کلاس است و نحوه تعریف آن شبیه تعریف متد است.

شکل کلی تعریف سازنده به صورت زیر است:

```
Public نام کلاس()
{
    دستورات بدنه سازنده;
}
```

- **نکته:**

اگر در هنگام تعریف فیلدها, آنها مقداردهی اولیه نشوند (در صورت عدم تعریف سازنده در کلاس), کامپایلر, فیلدهای از نوع bool را false فیلدهای از نوع عددی را صفر, فیلدهای از نوع کاراکتری را ‘\0’ و فیلدهای از نوع مرجع را تهی (null) در نظر می‌گیرد.

مثال

```
internal  class  test10
{
    public  int  x;
    public  test10()
    {
        x = 10;
    }
}

test10  obj1 = new  test10();
console.writeline(obj1.x);
```

سازندهٔ پارامتردار:

در موارد زیادی ممکن است نیاز باشد اعضای داده‌ای هر شئ, مقادیر خاص خودش را داشته باشد. در این حالت می‌توان از سازنده پارامتردار استفاده کرد.

پارامترها مشابه متد به سازنده اضافه می‌شوند و فقط نیاز است داخل پرانتز بعد از نام سازنده تعریف شوند.

مثال

```
internal  class  test11
{
     public  int  x;
     public  test11(int  a)
     {
          x = a;
     }
}

test11  obj1 = new  test11(300);
console.writeline(obj1.x);

test11  obj2 = new  test11(250);
console.writeline(obj2.x);

test11  obj3 = new  test11();
console.writeline(obj3.x);
```

مثال

```
internal  class  test12
{
    public  int  x = 45;
    public  test12()
    {
        x = 89;
    }
}

test12  obj1 = new  test12();
console.writeline(obj1.x);
```

سازندهٔ ایستا:

سازنده ایستا, اعضای داده‌ای ایستای کلاس را مقدار دهی می‌کند.

نکته:

- سازنده ایستا نمی‌تواند پارامتردار باشد.
- سازنده ایستا نمی‌تواند دارای سطح دستیابی باشد.
- تنها امکان داشتن یک سازنده ایستا در کلاس وجود دارد.
- یک کلاس می‌تواند هم سازنده ایستا و هم سازنده معمولی داشته باشد.
- امکان فراخوانی سازنده ایستا به طور صریح وجود ندارد. سازنده ایستا پس از بار شدن کلاس به صورت خودکار فراخوانی می‌شود.
- مانند متد ایستا, سازنده ایستا نیز نمی‌تواند به اعضای غیر ایستای کلاس دسترسی داشته باشد.

```
static نام کلاس()
{
    دستورات بدنه سازنده;
}
```

مثال

```
internal  class  test13
{
      public  static  int  a;
      public  int  b,c;
      static  test13()
      {
            a = 10;
      }
      public  test13()
      {
            b = 20;
      }
      static  test13(int  x)
      {
            c = 40;
      }
}

console.writeline(test13.a);

test13  obj1 = new  test13();
console.writeline(obj1.b);
```

سربارگذاری سازنده:

همانند متد, سازنده‌های کلاس نیز می‌توانند همنام باشند. به عبارت دیگر می‌توان چند سازنده با نام یکسان برای یک کلاس تعریف کرد. این سازنده‌ها باید در نوع, تعداد یا ترتیب پارامترها با هم متفاوت باشند.

مثال

```
internal class test14
{
    public int x;
    public double y;
    public test14()
    {
        x = 0;
        y = 0;
    }
    public test14(int a)
    {
        x = a;
        y = 0;
    }
    static test14(int a, double b)
    {
        x = a;
        y = b;
    }
}

test14 obj1 = new test14();
test14 obj2 = new test14(10);
test14 obj3 = new test14(10, 19.75);
console.writeline(obj1.x, obj1.y);
console.writeline(obj2.x, obj2.y);
console.writeline(obj3.x, obj3.y);
```

مخرّب:

برخلاف سازنده که در زمان ایجاد شئ فراخوانی می‌شود, مخرب در زمان نابودی شئ به صورت خودکار فراخوانی می‌شود.

```
~نام کلاس()
{
    دستورات بدنه مخرّب;
}
```

نکته:

- نام مخرب هم‌نام با کلاس است, ولی ابتدای آن علامت ~ (تیلدا) قرار می‌گیرد.
- از آنجایی که مخرب فاقد پارامتر است, نمی‌توان چند مخرب هم‌نام تعریف کرد. بنابراین هر کلاس فقط می‌تواند یک مخرب داشته باشد.
- اگر مخرب با واژه کلیدی static تعریف شود, برنامه با خطا مواجه می‌شود.
- در زمان خروج از برنامه, مخرب اشیایی که قبلاً توسط زباله‌روب حذف نشده‌اند, فراخوانی می‌شوند.

فیلد فقط خواندنی:

فیلد را می‌توان با استفاده از واژه کلیدی readonly به صورت فقط خواندنی تعریف کرد. فیلد فقط خواندنی فیلدی است که فقط در هنگام تعریف یا توسط سازنده مقدار اولیه می‌گیرد.

سطح دسترسی readonly نوع داده فیلد نام فیلد = مقدار;

- **نکته:**

اگر پس از مقداردهی فیلد فقط خواندنی (در زمان تعریف یا توسط سازنده), سعی در تغییر مقدار آن گردد, برنامه با خطا مواجه خواهد شد.

مثال

```
internal  class  test15
{
    public  const  float  pi = 3.14;
    public  readonly  int  radius = 0;
    public  test15(int  r)
    {
        radius = r;
    }
}

test15  obj1 = new  test15(16);
console.writeline(obj1.pi * obj1.radius * obj1.radius);
```

اندیس‌ساز:

با استفاده از اندیس‌ساز می‌توان به اعضای یک شئ همانند یک آرایه دستیابی پیدا کرد. در آرایه‌های معمولی, شماره اندیس باید یک عدد صحیح باشد ولی با استفاده از اندیس‌ساز, برنامه‌نویس می‌تواند اندیس‌هایی صحیح یا غیرصحیح تعریف کند.

از نظر تعریف, اندیس‌ساز شبیه خاصیت است و دارای دستیاب‌های get و set است ولی برخلاف خاصیت, نام آن همواره باید واژه کلیدی this باشد و نامی اختیاری ندارد.

```
سطح دسترسی  نوع مقدار برگشتی اندیس‌ساز  نوع داده اندیس[نام اندیس] this
{
        get {بدنه دستیاب;}
        set {بدنه دستیاب;}
}
```

اندیس‌ساز:

نوع داده مقدار برگشتی اندیس‌ساز, مشخص کننده نوع عنصر اندیس‌ساز است.
پارامتر اندیس‌ساز, اندیس عنصر مورد نظر برای دستیابی را مشخص می‌کند.

نکته:

- اگر اندیس‌ساز با واژه کلیدی static تعریف شود, برنامه خطا می‌دهد.
- همچون یک خاصیت, اندیس‌ساز نیز هیچ حافظه‌ای به خود اختصاص نمی‌دهد.
- دستیاب‌های اندیس‌ساز, همانند دستیاب‌های خاصیت تعریف می‌شوند.
- الزامی به تعریف اندیس‌ساز با مقدار صحیح عددی نیست و می‌توان اندیس‌سازی با پارامترهایی از نوع string یا double داشت.
- اندیس‌سازها قابلیت سربارگذاری دارند.
- اندیس‌سازها می‌توانند چندین پارامتری باشند مانند آرایه‌های دوبعدی که دو پارامتری هستند.

```
internal class test16
{
    int[] a;
    public test16(int n)
    {
        a = new int[n];
    }
    public int this[int idx]
    {
        set
        {
            a[idx] = value;
        }
        get
        {
            return a[idx];
        }
    }
}
```

مثال

```
test16 obj1 = new test16(5);

obj1[1] = 30;
obj1[2] = 33;
obj1[3] = 40;
obj1[4] = 44;
obj1[5] = 50;
obj1[1.9] = 55;

console.writeline(obj1[5]);
console.writeline(obj1[4]);
console.writeline(obj1[3]);
console.writeline(obj1[2]);
console.writeline(obj1[1]);
console.writeline(obj1[6]);
```

تمرین ۱:
خطای برنامه زیر را بیابید و آنرا تصحیح کنید.

```
internal  class  program
{
    public  int  x;
    public  program(int  a);
    {
        x = a;
    }
}

program  obj1 = new  program();
console.writeline(obj1.x);
```

تمرین ۲:
خطای برنامه زیر را بیابید و آنرا تصحیح کنید.

```
internal class program1
{
    public static readonly int x = 10;
    public program1();
    {
        x = 20;
    }
}

program1.x = 1000;
console.writeline(x);
```

تمرین ۳:
خطای برنامه زیر را بیابید و آنرا تصحیح کنید.

```
internal class program2
{
    public static readonly int x = 0;
    public static readonly int y = 0;

    public int sum();
    {
        return x*y;
    }
}

program2 obj1 = new program2();
obj1.x = 10;
Obj1.y = 20;
console.writeline(obj1.sum());
```

جلسه نهم:

وراثت

آنچه در این جلسه می‌خوانید:

۱- وراثت
۲- مزایای استفاده از وراثت
۳- نحوه تعریف کلاس مشتق
۴- سایر سطوح دستیابی اعضای کلاس
۵- کلاس مهر و موم شده
۶- مخفی کردن اعضای کلاس پایه در کلاس مشتق
۷- سازنده‌ها در وراثت
۸- مخرب‌ها در وراثت

وراثت:

وراثت یکی از اصول برنامه‌نویسی شئ‌گرا است که با استفاده از آن امکان استفاده مجدد از کد موجود فراهم می‌شود.

یک کلاس می‌تواند رفتار یا صفاتی را از یک کلاس دیگر به ارث ببرد.

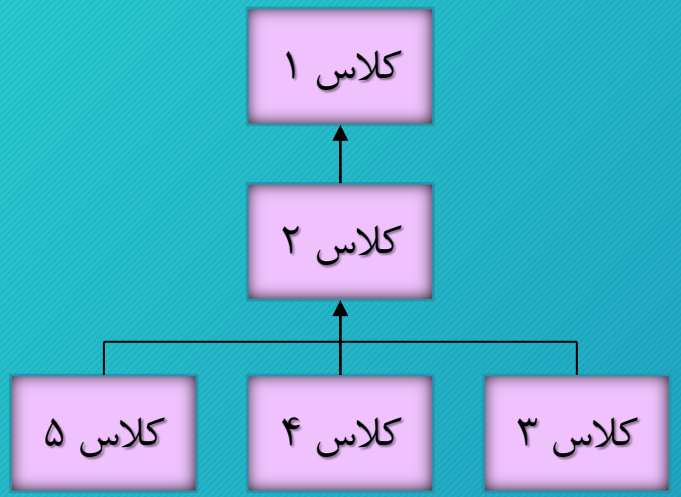
وراثت در برنامه‌نویسی شئ‌گرا, فرآیندی است که در آن می‌توان کلاس‌های جدیدی را از کلاس‌های موجود ایجاد کرد, به طوری که کلاس جدید رفتار و صفات کلاس موجود را به خودش اختصاص دهد. علاوه بر این, کلاس جدید می‌تواند صفات و رفتارهای خاص خودش را داشته باشد.

کلاس موجود را کلاس پایه یا کلاس مافوق و کلاسی که از کلاس پایه ایجاد می‌شود کلاس مشتق یا کلاس فرعی نامیده می‌شود.

وراثت:

نکته:

- هر کلاس مشتق فقط می‌تواند یک کلاس پایه داشته باشد.
- هر تعدادی از کلاس‌ها می‌توانند رفتارها و صفات یک کلاس خاص را به ارث برند.
- هر کلاس پایه می‌تواند مشتقِ یک کلاس پایه دیگر باشد که در این صورت سلسله مراتب ایجاد می‌شود.
- از یک کلاس پایه می‌توان چندین کلاس مشتق داشت.
- کلاس مشتق به اعضای خصوصی کلاس پایه‌اش دستیابی ندارد.
- در زبان C# برخلاف زبان C++ وراثت چندگانه (یک کلاس از چند کلاس به طور مستقیم مشتق شود) وجود ندارد.



مزایای استفاده از وراثت:

- صرفه‌جویی در زمان.
- کاهش خطا.
- سهولت درک برنامه.

- **نکته:**

یکی از مشکلات وراثت این است که کلاس مشتق می‌تواند فیلدها, متدها و خاصیت‌هایی را به ارث ببرد که ممکن است به آنها نیاز نداشته باشد.

نحوه تعریف کلاس مشتق:

نام کلاس پایه : نام کلاس مشتق class سطح دستیابی کلاس

{

اعضای کلاس مشتق

}

مثال

```
internal  class  test1
{
    private  int  a;
    float  b;
    public  int  sum(int  c, int d)
    {
        console.writeline( c+d );
    }
}
internal  class  test2 : test1
{
    int  p;
}

test2  obj = new  test2();
obj.sum(50,60);
```

مثال
وراثت سلسله مراتبی

```
internal  class  test1
{
    int  i=0, j=0;
}

internal  class  test2 : test1
{
    int  p;
}

internal  class  test3 : test2
{
    int  q=1;
}
```

سایر سطوح دستیابی اعضای کلاس:

دو مورد از سطوح دستیابی به اعضای کلاس public و private است که از مهمترین سطوح دستیابی می‌باشند.

اعضای کلاس می‌توانند سطوح دستیابی دیگری نیز داشته باشند:

- **سطح دستیابی حفاظت شده (protected):**

این سطح دستیابی در وراثت مورد استفاده قرار می‌گیرد. اعضای حفاظت شده کلاس پایه, فقط در داخل کلاس پایه, یا داخل هر کلاس که از آن مشتق شده است, قابل دستیابی می‌باشند.

- **سطح دستیابی داخلی (internal):**

یک عضو با سطح دستیابی داخلی, در پروژه خود مشابه سطح دستیابی عمومی و خارج از پروژه خود مشابه سطح دستیابی خصوصی عمل می‌کند.

- **سطح دستیابی حفاظت شده داخلی (protected internal):**

اعضای با این سطح دستیابی, هم در داخل کلاس‌های پروژه فعلی قابل استفاده خواهند بود و هم در داخل کلاس پایه و کلاس مشتق از کلاس پایه.

نکته:

پیشنهاد می‌شود در زمان انتخاب سطح دستیابی برای یک عضو, محدود کننده‌ترین سطح دستیابی انتخاب شود.

کلاس مهر و موم شده:

کلاس مهر و موم شده (Sealed class) کلاسی است که امکان ارث‌بری از آن غیر ممکن است. برای اینکه امکان ارث‌بری از یک کلاس غیرممکن شود, باید از واژه کلیدی sealed در ابتدای تعریف کلاس استفاده کرد.

```
sealed internal class test3
{
        ...
}
```

مخفی کردن اعضای کلاس پایه در کلاس مشتق:

در کلاس مشتق می‌توان عضوی همنام با عضو موجود در کلاس پایه تعریف کرد. در این حالت, عضو کلاس پایه در کلاس مشتق شده مخفی می‌شود. در چنین وضعیتی کامپایلر زبان C# یک پیام اخطار (نه خطا) می‌دهد.

برای جلوگیری از اخطار می‌توان از واژه کلیدی new در هنگام تعریف عضو استفاده کرد.

- **نکته:**

اگر نیاز به دسترسی به عضو مخفی باشد, می‌توان از واژه کلیدی base استفاده کرد.

مثال

```
internal  class  test1
{
    public  int  a=5;
}

internal  class  test2 : test1
{
    public  new  int  a=10;

    public  int  m()
    {
        return  base.a;
    }
}
test2  obj = new  test2();
console.writeline(obj.a);
console.writeline(obj.m());
```

سازنده‌ها در وراثت:

اگر هر کلاس پایه و کلاس مشتق دارای سازنده باشد, هنگام ساخت شئ از کلاس مشتق, سازنده کلاس پایه قبل از سازنده کلاس مشتق اجرا می‌شود.

از آنجا که مقداردهی اولیه در کلاس پایه ممکن است پیش‌نیاز مقداردهی اولیه در کلاس مشتق باشد, لذا سازنده آن باید زودتر اجرا شود.

مثال

```
internal  class  test1
{
    public  test1()
    {
        console.writeline(“Base class”);
    }
}

internal  class  test2 : test1
{
    public  test2()
    {
        console.writeline(“Derived class”);
    }
}

test2  obj = new  test2();
```

سازندهٔ پارامتردار در وراثت:

اگر فقط کلاس مشتق، سازنده پارامتردار داشته باشد، هنگامی که یک شئ از کلاس مشتق ساخته می‌شود فقط همان سازنده پارامتردار اجرا می‌شود.

اگر کلاس‌های پایه و مشتق، هر دو سازنده پارامتردار داشته باشند، در این حالت هنگام تعریف سازنده کلاس مشتق باید از واژه کلیدی base استفاده کرد تا بتوان به اعضای خصوصی کلاس پایه مقداردهی کرد.

به عبارت دیگر، برای این کار باید همه پارامترهایی که کلاس‌های پایه و مشتق نیاز دارند به سازنده کلاس مشتق ارسال شود و سپس به کمک واژه کلیدی base در معرفی سازنده کلاس مشتق، پارامترهای مناسب به کلاس پایه ارسال شود.

مثال

```
internal class test1
{
    private int a;
    public test1(int p)
    {
        console.writeline("Base class");
        a = p
    }
    public void showA()
    {
        console.writeline(a);
    }
}
```

```
internal class test2 : test1
{
    private int b;
    public test2(int p1, int p2) : base(p2)
    {
        console.writeline("Derived class");
        b = p1;
    }
    public void showB()
    {
        console.writeline(b);
    {

}

test2 obj = new test2(10,5);
obj.showA();
obj.showB();
```

مخرب‌ها در وراثت:

وقتی زباله‌روب، شئ کلاس مشتق را از حافظه حذف می‌کند، مخرب آن شئ را فراخوانی می‌کند. بدین ترتیب، زنجیره‌ای از فراخوانی‌های مخرب ایجاد می‌شود که در آن، مخرب‌های مستقیم و غیرمستقیم کلاس پایه به ترتیب معکوس اجرای سازنده اجرا می‌شوند.

جلسه دهم:

سربارگذاری عملگرها

آنچه در این جلسه می‌خوانید:

۱- مقدمه

۲- عملگرهای قابل سربارگذاری

۳- روش سربارگذاری عملگرها

۴- عملگرهای تبدیل

مقدمه:

سربارگذاری عملگرها امکانی است که با استفاده از آن می‌توان معنا عملکرد جدیدی برای عملگرها تعریف کرد.

سربارگذاری عملگرها یکی از مهمترین و جذاب‌ترین امکانات C# است زیرا گسترش داده‌ها را فراهم می‌کند.

گسترش داده‌ها ویژگی‌ای است که به برنامه‌نویس امکان می‌دهد تا به برنامه خود انواع داده‌های جدید اضافه کند.

به عنوان مثال می‌توان یک نوع داده جدید تعریف کرد و سپس عملگرها را با توجه به آن نوع داده جدید سربارگذاری نمود.

عملگرهای قابل سربارگذاری:

- عملگرهای یگانی (Unary Operators):

این عملگرها یک عملوند می‌گیرند و عبارتند از:

!, ~, ++, --

- عملگرهای دوتایی (Binary Operators):

این عملگرها دو عملوند می‌گیرند و عبارتند از:

+, -, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=

- نکته:

عملگرهای منطقی && و || را نیز می‌توان سربارگذاری کرد اما روش مستقیمی برای انجام این کار وجود ندارد.

روش سربارگذاری عملگرها:

شکل کلی سربارگذاری یک عملگر به صورت زیر است:

```
public  static  نوع داده برگشتی  operator  عملگر(پارامترها)
{
        عملیات مورد نظر برای تعریف مجدد عملگر;
}
```

چند نکته:

- متد عملگر باید عمومی و ایستا باشد.
- نوع داده برگشتی متد عملگر, اغلب باید از نوع کلاسی باشد که عملگر برای آن سربارگذاری شده است.
- توسط سربارگذاری عملگرها نمی‌توان اولویت اجرای عملگرها را تغییر داد.
- توسط سربارگذاری عملگرها نمی‌توان تعداد عملوندهای مورد نیاز را تغییر داد.
- منطقی آن است که در سربارگذاری عملگرها تعریف انجام شده با عملگر سازگار باشد.
- پارامترهای متد عملگر نمی‌توانند با واژه‌های کلیدی ref و out تعریف شوند.
- امکان سربارگذاری عملگرها برای اشیایی که ساخته نشده‌اند وجود ندارد.

مثال

```
internal class point
{
    int x,y;
    public point(int a, int b)
    {
        x = a;
        y = b;
    }
    public void print()
    {
        console.witeline(x,y);
    }
    public static point operator ++(point p)
    {
        point result = new point(p.x, p.y);
        result.x++;
        result.y++;
        return result;
    }
}

point p1 = new point(5,5);
p1++;
p1.print();
```

مثال

```
internal class point
{
    private int x,y;
    public point(int a, int b)
    {
        x = a;
        y = b;
    }
    public void print()
    {
        console.writeline(x,y);
    }
    public static point operator +(point p1, point p2)
    {
        point result = new point(p1.x+p2.x, p1.y+p2.y);
        return result;
    }
}
point p1 = new point(5,5);
point p2 = new point(10,15);
point result = p1+p2;
result.print();
```

مثال

```
internal class point
{
    private int x,y;
    public point(int a)
    {
        x = a;
    }
    public void print()
    {
        console.writeline(x);
    }
    public static point operator ++(point p)
    {
        point result = new point(p.x);
        result.x--;
        return result;
    }
}
point p1 = new point(5);
p1++;
result.print();
```

عملگرهای تبدیل:

توسط عملگرهای تبدیل, می‌توان یک شئ را به طور ضمنی (Implicit) و یا به طور صریح (Explicit) به نوع دیگری تبدیل کرد تا از این طریق بتوان آنرا بنا به نیاز مثلاً در یک عبارت محاسباتی استفاده کرد.

برای این منظور باید از متد عملگر خاصی به نام متد عملگر تبدیل استفاده کرد.

```
public static operator explicit/implicit نوع داده هدف(نام کلاس نام شئ)
{
        return مقدار;
}
```

مثال

```
internal class myclass
{
    int x,y;
    public myclass(int a, int b)
    {
        x = a;
        y = b;
    }
    public static implicit operator int(myclass p)
    {
        return p.x * p.y;
    }
}

myclass obj1 = new myclass(2,2);
myclass obj2 = new myclass(1,1);
int i = obj1 * 3;
console.writeline(i);
i = obj1 - 3;
console.writeline(i);
i = obj1 + obj2;
console.writeline(i);
i = obj1;
console.writeline(i);
```

مثال

```
internal  class  myclass
{
    int x,y;
    public  myclass(int  a, int  b)
    {
        x = a;
        y = b;
    }
    public  static  explicit  operator  int(myclass  p)
    {
        return  p.x * p.y;
    }
}

myclass  obj1 = new  myclass(2,2);
int  i = (int) obj1;
console.writeline(i);
```

پایان