EXPERT PROGRAMMING

Shahrokh Amani, Zahra Pasandideh, Kiyan Rahmati, Mani Hosseini, Mahdi Tamini, Fatemeh Sadat Bolboli

JUNE 5, 2026
EMPEROR-DEV
Toronto, ON, Canada

فهرست مطالب

مقدمه

برنامه‌نویسی شیءگرا با زبان سی‌شارپ یکی از دروس پایه و اساسی در رشته مهندسی نرم‌افزار است. این زبان که توسط شرکت مایکروسافت توسعه یافته است، به دلیل سادگی، قدرت و انعطاف‌پذیری بالا، یکی از محبوب‌ترین زبان‌های برنامه‌نویسی در دنیا محسوب می‌شود. در این جزوه، سعی شده است تمامی مفاهیم پایه تا پیشرفته این زبان به صورت گام به گام و با مثال‌های متنوع آموزش داده شود.

هدف از این آموزش، فراگیری اصول صحیح برنامه‌نویسی، آشنایی با مفاهیم شیءگرایی و توانایی حل مسائل واقعی با استفاده از زبان سی‌شارپ است. این جزوه به گونه‌ای تنظیم شده که هم برای دانشجویان مبتدی و هم برای آن‌هایی که آشنایی اولیه با برنامه‌نویسی دارند، مفید واقع شود.

فصل اول

آشنایی با محیط برنامه‌نویسی ویژوال استودیو و مفاهیم پایه

1.1. معرفی محیط توسعه یکپارچه ویژوال استودیو

ویژوال استودیو به عنوان یک محیط توسعه یکپارچه (IDE) پیشرفته، تمامی ابزارهای مورد نیاز برای توسعه نرم‌افزار را در یک پلتفرم واحد گرد هم آورده است. این محیط که توسط شرکت مایکروسافت طراحی و عرضه شده است، امکان نوشتن، ویرایش، دیباگ و کامپایل کدها را به ساده‌ترین شکل ممکن فراهم می‌آورد. ویژوال استودیو از زبان‌های برنامه‌نویسی متعددی پشتیبانی می‌کند که سی‌شارپ یکی از اصلی‌ترین و محبوب‌ترین آن‌هاست. این محیط با ارائه ویژگی‌هایی مانند تکمیل خودکار کد، هایلایت کردن سینتکس، تشخیص خطاها در حین تایپ و ابزارهای قدرتمند دیباگ، تجربه برنامه‌نویسی را برای توسعه‌دهندگان لذت‌بخش و کارآمد می‌سازد. همچنین پشتیبانی از کتابخانه‌ها و فریم‌ورک‌های مختلف، ویژوال استودیو را به انتخابی ایده‌آل برای پروژه‌های کوچک تا سازمانی تبدیل کرده است.محیط ویژوال استودیو از بخش‌های متنوع و تخصصی تشکیل شده است که هر کدام وظیفه مشخصی را بر عهده دارند.

ویرایشگر کد قلب اصلی این محیط است که در آن کدها نوشته و ویرایش می‌شوند. این ویرایشگر امکانات پیشرفته‌ای مانند رنگ‌بندی سینتکس، فولدینگ کد و تکمیل هوشمند ارائه می‌دهد. اکسپلورر راه‌حل یا Solution Explorer ساختار درختی پروژه را نمایش می‌دهد و امکان مدیریت فایل‌ها، پوشه‌ها و منابع پروژه را فراهم می‌کند. پنجره خصوصیات یا Properties Window برای مشاهده و تغییر ویژگی‌های المان‌های مختلف پروژه استفاده می‌شود. لیست خطاها یا Error List تمامی هشدارها و خطاهای موجود در پروژه را به صورت زنده نمایش می‌دهد. پنجره خروجی یا Output Window محل نمایش نتایج اجرای برنامه، پیام‌های کامپایلر و دیگر اطلاعات سیستم است. همچنین جعبه ابزار یا Toolbox در پروژه‌های ویندوز فرم، شامل کنترل‌های گرافیکی متنوع برای طراحی رابط کاربری می‌باشد.

1.2. مراحل ایجاد پروژه جدید در ویژوال استودیو

ایجاد یک پروژه جدید در ویژوال استودیو بسیار ساده است. پس از اجرای برنامه، مراحل زیر را دنبال کنید:

1. از منوی File گزینه New و سپس Project را انتخاب کنید.

2. در پنجره باز شده، از بخش Installed، زبان Visual C# را انتخاب نمایید.

3. از لیست پروژه‌ها، گزینه Console App (.NET Framework) را انتخاب کنید. این گزینه برای ایجاد برنامه‌های کنسول مناسب است.

4. در بخش Name، نام پروژه خود را وارد کنید. بهتر است نامی معنادار و مرتبط با کار پروژه انتخاب شود.

5. در بخش Location، مسیر ذخیره‌سازی پروژه را مشخص نمایید.

6. در بخش Solution name، نام Solution را وارد کنید. Solution می‌تواند شامل چندین پروژه باشد.

7. بر روی دکمه OK کلیک کنید.

پس از طی این مراحل، ویژوال استودیو به طور خودکار یک پروژه کنسول با ساختار اولیه ایجاد می‌کند. این ساختار شامل فایل Program.cs می‌باشد که نقطه شروع برنامه است.

1.2. آشنایی با محیط و بخش‌های مختلف ویژوال استودیو

محیط ویژوال استودیو از بخش‌های مختلفی تشکیل شده است که هرکدام وظیفه خاصی دارند:

- ویرایشگر کد (Code Editor): بخش اصلی که کدهای برنامه در آن نوشته می‌شوند. این بخش دارای قابلیت‌های مختلفی مانند هایلایت سینتکس، تکمیل خودکار کد و نمایش خطاها است.
- Solution Explorer: این بخش ساختار پروژه را نشان می‌دهد. شامل فایل‌ها، پوشه‌ها و منابع پروژه می‌باشد.
- Properties Window: خصوصیات المان‌های مختلف پروژه در این بخش نمایش داده می‌شود و می‌توان آنها را تغییر داد.

- Error List: لیست خطاها و هشدارهای پروژه در این بخش نمایش داده می‌شود.
- Output Window: خروجی برنامه، پیام‌های کامپایل و دیگر اطلاعات در این بخش نمایش داده می‌شوند.
- Toolbox: در پروژه‌های ویندوز فرم، این بخش شامل کنترل‌های مختلف برای طراحی رابط کاربری است.

1.4. مفاهیم پایه: Namespace، کلاس و تابع Main

Name space یا فضای نام، یک مفهوم سازمان‌دهی در زبان سی‌شارپ است که به منظور جلوگیری از تداخل نام‌ها و گروه‌بندی منطقی کدها استفاده می‌شود. هر Name space مانند یک کانتینر است که می‌تواند شامل کلاس‌ها، اینترفیس‌ها، ساختارها و سایر المان‌های کد باشد. با استفاده از Name spaceها، می‌توان کدهای مرتبط را در یک گروه قرار داد و از بروز مشکلات ناشی از تشابه نام جلوگیری کرد. به عنوان مثال، تمامی کلاس‌های مربوط به کار با پایگاه داده می‌توانند در Name spaceی به نام DataAccess قرار گیرند. برای استفاده از المان‌های یک Name space در بخش دیگر کد، باید آن Name space را با دستور using وارد کرد

کلاس

کلاس بلوک اصلی و پایه‌ای در برنامه‌نویسی شیءگرا محسوب می‌شود. یک کلاس در واقع یک الگو یا قالب است که خصوصیات و رفتارهای یک موجودیت را تعریف می‌کند. هر کلاس می‌تواند شامل فیلدها (متغیرها)، خصوصیات (Properties) و متدها (توابع) باشد. به عنوان مثال، کلاسی به نام "Student" می‌تواند دارای خصوصیاتی مانند نام، شماره دانشجویی و معدل باشد و متدهایی مانند ثبت نام و محاسبه معدل را ارائه دهد. کلاس‌ها به برنامه‌نویس این امکان را می‌دهند که موجودیت‌های دنیای واقعی را در قالب کد مدل‌سازی کنند و از کپسوله‌سازی، وراثت و چندریختی بهره ببرند.

تابع main

تابع **Main** نقطه شروع اجرای هر برنامه سی‌شارپ است. هنگامی که یک برنامه اجرا می‌شود، اولین کدی که به صورت خودکار فراخوانی می‌شود، تابع **Main** می‌باشد. این تابع باید در یکی از کلاس‌های برنامه تعریف شده باشد و ساختار مشخصی دارد. معمولاً تابع **Main** به صورت **static void Main(string[] args)** تعریف می‌شود. کلمه کلیدی **static** نشان‌دهنده این است که برای فراخوانی این تابع نیازی به ایجاد نمونه از کلاس نیست. **void** مشخص می‌کند که تابع مقداری باز نمی‌گرداند و **string[] args** آرایه‌ای است که پارامترهای خط فرمان را دریافت می‌کند.

تمامی منطق اصلی برنامه در داخل این تابع یا با فراخوانی توابع دیگر از این تابع نوشته می‌شود.

کلمه **static** به این معنی است که برای استفاده از این تابع نیازی به ایجاد شیء از کلاس نیست. کلمه **void** نشان می‌دهد که این تابع مقداری باز نمی‌گرداند. **string[] args** نیز آرایه‌ای از رشته‌ها است که پارامترهای خط فرمان را دریافت می‌کند.

فصل دوم

ساختار برنامه‌های کنسول در سی‌شارپ

2.1. آشنایی با برنامه‌های کنسول

برنامه‌های کنسول، برنامه‌هایی هستند که رابط کاربری گرافیکی ندارند و تمامی ورودی و خروجی آنها از طریق یک پنجره متنی (Command Prompt) انجام می‌شود. این نوع برنامه‌ها برای کارهای ساده، آموزش مفاهیم برنامه‌نویسی و ابزارهای خط فرمان مناسب هستند. ساختار این برنامه‌ها ساده است و تمرکز اصلی بر روی منطق برنامه و الگوریتم‌ها می‌باشد، نه طراحی رابط کاربری.

مزیت برنامه‌های کنسول سادگی و سرعت اجرای آنها است. برای شروع یادگیری برنامه‌نویسی، معمولاً از برنامه‌های کنسول استفاده می‌شود زیرا پیچیدگی‌های رابط کاربری گرافیکی را ندارند و برنامه‌نویس می‌تواند بر روی مفاهیم اصلی برنامه‌نویسی تمرکز کند.

2.2. دستورات ورودی و خروجی در کنسول

برای ارتباط با کاربر در برنامه‌های کنسول، از دستورات ورودی و خروجی استفاده می‌شود. این دستورات در کلاس Console قرار دارند که بخشی از Namespace System می‌باشد. به همین دلیل در ابتدای هر برنامه سی‌شارپ، دستور using System; نوشته می‌شود تا بتوان به راحتی از کلاس Console استفاده کرد.

دستورات خروجی برای نمایش اطلاعات به کاربر و دستورات ورودی برای دریافت اطلاعات از کاربر استفاده می‌شوند. این دستورات ساده اما بسیار قدرتمند هستند و پایه و اساس هر برنامه کنسولی را تشکیل می‌دهند.

2.3. تفاوت بین Console.Write و Console.WriteLine

Console.Write و Console.WriteLine دو دستور اصلی برای نمایش خروجی در کنسول هستند که تفاوت اصلی آن‌ها در رفتار پس از نمایش متن است. دستور Console.Write متنی را در خروجی نمایش می‌دهد اما پس از اتمام نمایش، نشانگر مکان نما (Cursor) را به خط بعد منتقل نمی‌کند. این به معنای آن است که اگر دستور دیگری پس از آن برای نمایش خروجی داشته باشیم، خروجی آن در ادامه همان خط چاپ می‌شود. در مقابل، دستور Console.WriteLine پس از نمایش متن، به طور خودکار نشانگر مکان نما را به ابتدای خط بعد منتقل می‌کند. به عبارت دیگر، این دستور پس از چاپ متن، یک خط جدید ایجاد می‌کند. این تفاوت کوچک اما مهم، در سازماندهی خروجی برنامه بسیار تاثیرگذار است

به مثال زیر توجه کنید:

```
static void main (string[] args)
{
  Console.Write("Hello");
  Console.Write("World");
}
```

خروجی این کد چاپ عبارت Hello World در یک خط می باشد.

```
static void main (string[] args)
{
  Console.WriteLine("Hello");
  Console.WriteLine("World");
}
```

اما در این کد کلمه Hello و World هر کدام در یک خط نمایش داده می شوند.

2.4. دریافت ورودی از کاربر با Console.ReadLine()

برای دریافت ورودی از کاربر در برنامه‌های کنسول، از دستور Console.ReadLine() استفاده می‌شود. این دستور منتظر می‌ماند تا کاربر متنی را وارد کرده و کلید Enter را فشار دهد. پس از فشار دادن Enter، متن وارد شده به عنوان یک رشته (String) بازگردانده می‌شود.

مثال:

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("لطفا نام خود را وارد کنید: ");
        string name = Console.ReadLine();
        Console.WriteLine("سلام " + name + "!");
    }
}
```

در این مثال، ابتدا پیامی به کاربر نمایش داده می‌شود تا نام خود را وارد کند. سپس برنامه منتظر می‌ماند تا کاربر نام خود را وارد کرده و Enter را بزند. پس از آن، نام وارد شده در متغیر name ذخیره می‌شود و در نهایت پیام سلام با نام کاربر نمایش داده می‌شود.

نکته مهم: Console.ReadLine() همیشه یک رشته برمی‌گرداند، حتی اگر کاربر عدد وارد کند. بنابراین اگر نیاز به دریافت عدد داریم، باید رشته دریافتی را به عدد تبدیل کنیم که در فصل‌های بعدی به آن خواهیم پرداخت.

فصل سوم

انواع داده‌ها در سی‌شارپ (Data Types)

3.1. مفهوم انواع داده و اهمیت آن

در برنامه‌نویسی، نوع داده (Data Type) مشخص می‌کند که چه نوع مقداری در یک متغیر ذخیره می‌شود و چه عملیاتی را می‌توان روی آن انجام داد. انتخاب نوع داده مناسب برای هر متغیر بسیار مهم است زیرا:

1. استفاده بهینه از حافظه: هر نوع داده مقدار مشخصی از حافظه را اشغال می‌کند.
2. افزایش سرعت برنامه: استفاده از نوع داده مناسب باعث افزایش سرعت اجرای برنامه می‌شود.
3. کاهش خطاها: نوع داده‌های مناسب از بروز خطاهای منطقی جلوگیری می‌کنند.
4. خوانایی کد: استفاده از نوع داده مناسب باعث خوانایی بیشتر کد می‌شود

سی‌شارپ دو دسته اصلی از انواع داده دارد:

1. انواع داده مقداری (**Value Types**): مستقیماً در حافظه **Stack** ذخیره می‌شوند و شامل انواع عددی، کاراکتری و بولین هستند.
2. انواع داده مرجعی (**Reference Types**): در حافظه **Heap** ذخیره می‌شوند و آدرس آنها در **Stack** نگهداری می‌شود. شامل رشته‌ها، آرایه‌ها و کلاس‌ها هستند.

3.2. انواع داده عدد صحیح و مشخصات آنها

انواع داده عدد صحیح برای ذخیره اعداد صحیح (بدون اعشار) استفاده می‌شوند. هر کدام از این انواع، محدوده مشخصی از اعداد را پشتیبانی می‌کنند و مقدار حافظه متفاوتی اشغال می‌کنند:

1. byte: کوچکترین نوع عدد صحیح که 1 بایت (8 بیت) حافظه اشغال می‌کند. محدوده آن از 0 تا 255 است. فقط اعداد مثبت را پشتیبانی می‌کند.
2. sbyte: همانند byte اما علامت‌دار است. 1 بایت حافظه اشغال می‌کند و محدوده آن از -128 تا 127 است.

3. short: 2 بایت (16 بیت) حافظه اشغال می‌کند. محدوده آن از -32,768 تا 32,767 است.

4. ushort: نوع short بدون علامت. 2 بایت حافظه اشغال می‌کند و محدوده آن از 0 تا 65,535 است.

5. int: پرکاربردترین نوع عدد صحیح که 4 بایت (32 بیت) حافظه اشغال می‌کند. محدوده آن از -2,147,483,648 تا 2,147,483,647 است.

6. uint: نوع int بدون علامت. 4 بایت حافظه اشغال می‌کند و محدوده آن از 0 تا 4,294,967,295 است.

7. long: برای اعداد صحیح بزرگ استفاده می‌شود. 8 بایت (64 بیت) حافظه اشغال می‌کند. محدوده آن از -9,223,372,036,854,775,808 تا 9,223,372,036,854,775,807 است.

8. ulong: نوع long بدون علامت. 8 بایت حافظه اشغال می‌کند و محدوده آن از 0 تا 18,446,744,073,709,551,615 است.

انتخاب نوع مناسب بستگی به محدوده عددی مورد نیاز دارد. برای مثال، برای سن یک نفر از byte استفاده می‌کنیم (چون سن معمولاً کمتر از 255 است)، برای شماره دانشجویی از int استفاده می‌کنیم و برای کد ملی که 10 رقمی است از long استفاده می‌کنیم.

3.3. انواع داده اعشاری و کاربردهای هرکدام

انواع داده اعشاری برای ذخیره اعداد با بخش اعشاری استفاده می‌شوند:

1. نوع داده FLOAT 4 بایت حافظه اشغال می‌کند و دقت حدود 7 رقم اعشار دارد. پسوند f یا F باید به عدد اضافه شود. مثال:

```
float pi = 3.14f;
```

2. نوع داده double مقدار 8 بایت حافظه اشغال می کند و دقت حدود 15-16 رقم اعشار دارد. نوع پیش‌فرض برای اعداد اعشاری در سی‌شارپ است. مثال:

```
double salary = 1500.75
```

3. نوع داده decimal مقدار 16 بایت حافظه اشغال می‌کند و دقت حدود 28-29 رقم اعشار دارد. برای محاسبات مالی و پولی که دقت بالا نیاز است، استفاده می‌شود. پسوند m یا M باید به عدد اضافه شود. مثال:

```
decimal price = 99.99m;
```

انتخاب بین این سه نوع بستگی به دقت مورد نیاز و مقدار حافظه در دسترس دارد. برای محاسبات عمومی از double، برای محاسبات مالی از decimal و برای مواقعی که حافظه محدود است از float استفاده می‌شود.

3.4. انواع داده کاراکتری و رشته‌ای

1. **char**: برای ذخیره یک کاراکتر (نویسه) استفاده می‌شود. 2 بایت حافظه اشغال می‌کند. مقادیر **char** باید بین **single quotes** قرار گیرند. مثال:

```
char grade = 'A';
```

2. **string**: برای ذخیره دنباله‌ای از کاراکترها (رشته) استفاده می‌شود. جزو انواع داده مرجعی است. مقادیر **string** باید بین **double quotes** قرار گیرند. مثال:

```
string name = "Ali";
```

تفاوت اصلی بین char و string

char و string دو نوع داده متنی در سی‌شارپ هستند که تفاوت‌های اساسی با یکدیگر دارند. نوع char برای ذخیره یک کاراکتر (نویسه) استفاده می‌شود و مقدار آن باید بین single quotes قرار گیرد، مانند 'A' یا '5'. char یک نوع داده مقداری است و 2 بایت حافظه اشغال می‌کند. در مقابل، نوع string برای ذخیره دنباله‌ای از کاراکترها (رشته) به کار می‌رود و مقادیر آن باید بین double quotes قرار گیرند، مانند "Hello" یا "123". string یک نوع داده مرجعی است و می‌تواند از صفر تا تعداد زیادی کاراکتر را در خود جای دهد. همچنین عملیات و متدهای مختلفی برای کار با رشته‌ها در دسترس است.

3.5. نوع داده منطقی (بولین)

نوع داده bool برای ذخیره مقادیر منطقی true (درست) و false (غلط) استفاده می‌شود. این نوع داده که به نام بولین شناخته می‌شود، فقط این دو مقدار را می‌پذیرد و 1 بایت حافظه اشغال می‌کند. از نوع bool به طور گسترده در ساختارهای تصمیم‌گیری و حلقه‌ها استفاده می‌شود. به عنوان مثال، نتیجه مقایسه دو عدد (مانند 5 > 3) همیشه یک مقدار bool است. همچنین می‌توان از متغیرهای bool برای نشان دادن وضعیت‌های مختلف برنامه مانند isLoggedIn (وارد سیستم شده)، hasPermission (دارای مجوز) یا isCompleted (تکمیل شده) استفاده کرد. مقادیر bool نمی‌توانند به صورت مستقیم با اعداد تبدیل شوند. یعنی نمی‌توان نوشت bool x = 1; زیرا 1 یک عدد است نه مقدار بولین.

مثال‌ها:

```
bool isStudent = true;
bool hasPassed = false;
```

فصل چهارم

متغیرها و ثوابت در برنامه‌نویسی

4.1. تعریف متغیر و کاربرد آن

متغیر نامی نمادین برای یک محل در حافظه است که می‌تواند مقادیر مختلفی را در طول اجرای برنامه ذخیره کند. متغیرها برای نگهداری موقت داده‌ها استفاده می‌شوند تا بتوان در قسمت‌های مختلف برنامه به آنها دسترسی داشت و یا آنها را تغییر داد.

هر متغیر دارای سه ویژگی اصلی است:

1. نام (Name): شناسه متغیر که باید منحصر به فرد و معنادار باشد.
2. نوع (Type): مشخص می‌کند چه نوع داده‌ای در متغیر ذخیره می‌شود.
3. مقدار (Value): داده‌ای که در متغیر ذخیره شده است.

تعریف متغیر در سی‌شارپ به صورت زیر است:

[نوع داده] [نام متغیر];

یا

[نوع داده] [نام متغیر] = [مقدار];

مثال :

```
int age; // تعریف متغیر بدون مقدار اولیه
double salary = 1500.50; // تعریف متغیر با مقدار اولیه
string name = "Mohammad"; // تعریف متغیر رشته ای
```

4.2. قواعد نامگذاری متغیرها

نامگذاری مناسب متغیرها یکی از اصول مهم کدنویسی تمیز است. قواعد نامگذاری در سی‌شارپ:

1. حروف مجاز: نام می‌تواند شامل حروف انگلیسی (**a-z, A-Z**)، اعداد (0-9) و کاراکتر زیرخط (_) باشد.

2. شروع نام: نام باید با حرف یا زیرخط شروع شود، نه با عدد.

3. کلمات کلیدی: نمی‌توان از کلمات کلیدی سی‌شارپ (مانند **int, class, void**) به عنوان نام متغیر استفاده کرد.

4. حساسیت به حروف: سی‌شارپ به بزرگی و کوچکی حروف حساس است (**Case-sensitive**). یعنی **age** و **Age** دو متغیر متفاوت هستند.

5. طول نام: هیچ محدودیتی برای طول نام وجود ندارد اما بهتر است نام‌ها معقول و معنادار باشند.

مثال‌های نامگذاری صحیح:

```
// مثال‌های نام‌گذاری صحیح:
int age;
string firstName;
double _salary; // متغیر خصوصی یا داخلی کلاس
int studentCount;

// مثال‌های نام‌گذاری اشتباه:
// int 2age;          // خطا: شروع نام متغیر با عدد
// string first-name; // خطا: استفاده از خط تیره (-) در نام متغیر
// double class;      // خطا: استفاده از کلمه کلیدی (class) به عنوان نام متغیر
```

4.3. مقداردهی اولیه به متغیرها

مقداردهی اولیه به متغیرها می‌تواند در زمان تعریف یا بعد از آن انجام شود:

روش اول: تعریف و مقداردهی همزمان

```
int number = 10;
string name = "Ali";
```

روش دوم: تعریف و سپس مقداردهی

```
int age;
age = 25;
```

روش سوم: مقداردهی متغیرهای متعدد از یک نوع

```
int x = 5, y = 10, z = 15;
```

نکته مهم: متغیرهای محلی (متغیرهایی که در داخل متدها تعریف می‌شوند) باید قبل از استفاده مقداردهی شده باشند، در غیر این صورت خطای کامپایلر دریافت می‌کنیم.

4.4. ثوابت (Constants) و تفاوت آن با متغیرها

ثابت (Constant) نوعی از متغیر است که مقدار آن در طول اجرای برنامه ثابت می‌ماند و قابل تغییر نیست. ثوابت با کلمه کلیدی const تعریف می‌شوند.

تفاوت‌های اصلی ثوابت با متغیرها:

1. ثبات مقدار: مقدار ثوابت پس از تعریف قابل تغییر نیست.
2. مقداردهی اجباری: ثوابت باید در زمان تعریف مقداردهی شوند.
3. نامگذاری: معمولاً ثوابت با حروف بزرگ نوشته می‌شوند.
4. کارایی: استفاده از ثوابت می‌تواند باعث افزایش کارایی برنامه شود.

مثال تعریف ثابت:

```
const double PI = 3.14159;
const int MAX_SIZE = 100;
const string COMPANY_NAME = "MyCompany";
```

کاربردهای ثوابت:

- ذخیره مقادیر ثابت ریاضی (مانند PI)
- تعریف سایزهای ثابت (مانند اندازه آرایه)
- ذخیره مقادیر پیکربندی
- افزایش خوانایی کد

4.5. محدوده دسترسی متغیرها (Scope)

محدوده دسترسی (Scope) به بخشی از برنامه گفته می‌شود که در آن می‌توان به یک متغیر دسترسی داشت. در سی‌شارپ، محدوده دسترسی متغیرها به سه دسته تقسیم می‌شود:

1. محدوده بلوک (Block Scope): متغیرهایی که در داخل یک بلوک (بین آکولادهای {}) تعریف می‌شوند، فقط در همان بلوک و بلوک‌های داخلی آن قابل دسترسی هستند.

2. محدوده متد (Method Scope): متغیرهایی که در داخل یک متد تعریف می‌شوند، فقط در همان متد قابل دسترسی هستند.

3. محدوده کلاس (Class Scope): متغیرهایی که در سطح کلاس تعریف می‌شوند (فیلدهای کلاس)، در تمام متدهای همان کلاس قابل دسترسی هستند.

مثال:

```
static void Main(string[] args)
{
    int classVariable = 10;  //محدوده کلاس

    void MyMethod()
    {
        int methodVariable = 20;  //محدوده متد

        if (true)
        {
            int blockVariable = 30;  //محدوده بلوک
                                     //در اینجا به همه متغیر ها دسترسی داریم
        }
        //در دسترس نیست blockVariable در اینجا
        //اما classVariable و methodVariable در دسترس هستند

    }
}
```

فصل پنجم

عملگرها در برنامه‌نویسی سی‌شارپ

5.1. عملگرهای حسابی و ریاضی

عملگرهای حسابی برای انجام عملیات ریاضی پایه استفاده می‌شوند:

1. جمع (+): دو مقدار را با هم جمع می‌کند.

```
int sum = 5 + 3; // 8 :نتیجه
```

2. تفریق (-): مقدار دوم را از مقدار اول کم می‌کند.

```
int difference = 10 - 4; // 6 :نتیجه
```

3. ضرب (*): دو مقدار را در هم ضرب می‌کند.

```
int product = 6 * 7; // 42 :نتیجه
```

4. تقسیم (/): مقدار اول را بر مقدار دوم تقسیم می‌کند.

```
int quotient = 20 / 4; // 5 :نتیجه
```

5. باقیمانده (%): باقیمانده تقسیم مقدار اول بر مقدار دوم را برمی‌گرداند.

```
int remainder = 17 % 5; // نتیجه: 2 (چون 17 تقسیم بر 5 می‌شود 3 و باقیمانده 2)
```

نکته مهم در مورد عملگر تقسیم:

- اگر هر دو عملوند عدد صحیح باشند، نتیجه تقسیم نیز عدد صحیح خواهد بود (بخش اعشاری حذف می‌شود).
- اگر حداقل یکی از عملوندها اعشاری باشد، نتیجه تقسیم اعشاری خواهد بود.

مثال:

```
int result1 = 7 / 2;       // نتیجه: 3 (اعشار حذف شد)
double result2 = 7.0 / 2;  // نتیجه: 3.5
double result3 = 7 / 2.0;  // نتیجه: 3.5
```

5.2. عملگرهای رابطه‌ای و مقایسه‌ای

عملگرهای رابطه‌ای برای مقایسه دو مقدار استفاده می‌شوند و نتیجه آنها همیشه یک مقدار بولین (true یا false) است:

1. مساوی (==): بررسی می‌کند آیا دو مقدار با هم برابر هستند؟

عملگرهای رابطه‌ای برای مقایسه دو مقدار استفاده می‌شوند و نتیجه آنها همیشه یک مقدار بولین (true یا false) است:

```
bool isEqual = (5 == 5); // true
bool isNotEqual = (5 == 3); // false
```

2. نامساوی (!=): بررسی می‌کند

آیا دو مقدار با هم برابر نیستند؟

```
bool notEqual = (5 != 3); // true
bool equal = (5 != 5); // false
```

3. بزرگتر (>): بررسی می‌کند آیا مقدار اول از مقدار دوم بزرگتر است؟

```
bool isGreater = (10 > 5); // true
bool notGreater = (5 > 10); // false
```

4. کوچکتر (<): بررسی می‌کند آیا مقدار اول از مقدار دوم کوچکتر است؟

```
bool isLess = (3 < 7); // true
bool notLess = (10 < 5); // false
```

5. بزرگتر یا مساوی (>=): بررسی می‌کند آیا مقدار اول از مقدار دوم بزرگتر یا مساوی است؟

```
bool greaterOrEqual1 = (10 >= 10); // true
bool greaterOrEqual2 = (10 >= 5); // true
bool greaterOrEqual3 = (5 >= 10); // false
```

6. کوچکتر یا مساوی (<=): بررسی می‌کند آیا مقدار اول از مقدار دوم کوچکتر یا مساوی است؟

```
bool lessOrEqual1 = (5 <= 5); // true
bool lessOrEqual2 = (5 <= 10); // true
bool lessOrEqual3 = (10 <= 5); // false
```

5.3. عملگرهای منطقی

عملگرهای منطقی برای ترکیب چند شرط استفاده می‌شوند:

1. AND منطقی (&&): اگر هر دو شرط true باشند، نتیجه true است.

```
bool result = (5 > 3) && (10 < 15); // true && true = true
bool result2 = (5 > 3) && (10 > 15); // true && false = false
```

2. OR منطقی (||): اگر حداقل یکی از شرط ها true باشد، نتیجه true است.

```
bool result = (5 > 3)  (10 > 15); // true  false = true
bool result2 = (5 < 3)  (10 > 15); // false  false = false
```

3. NOT منطقی (!): نتیجه یک شرط را معکوس می کند.

```
bool result = !(5 > 3); // !true = false
bool result2 = !(5 < 3); // !false = true
```

جدول درستی (Truth Table) برای عملگرهای منطقی:

A	B	A&&B	A || B	!A
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

5.4. عملگرهای تخصیصی

عملگرهای تخصیصی برای اختصاص دادن مقادیر به متغیرها استفاده می‌شوند:

1. تخصیص ساده (=): مقدار سمت راست را به متغیر سمت چپ اختصاص می‌دهد.

```
int x = 10;
string name = "Ali";
```

2. تخصیص جمع (+=): مقدار سمت راست را با متغیر سمت چپ جمع کرده و نتیجه را در متغیر سمت چپ ذخیره می‌کند.

```
int x = 5;
x += 3; // معادل x = x + 3 → x = 8
```

3. تخصیص تفریق (-=): مقدار سمت راست را از متغیر سمت چپ کم کرده و نتیجه را در متغیر سمت چپ ذخیره می‌کند.

```
int x = 10;
x -= 4; // معادل x = x - 4 → x = 6
```

4. تخصیص ضرب (*=): متغیر سمت چپ را در مقدار سمت راست ضرب کرده و نتیجه را در متغیر سمت چپ ذخیره می‌کند.

```
int x = 5;
x *= 3; // معادل x = x * 3 → x = 15
```

5. تخصیص تقسیم (/=): متغیر سمت چپ را بر مقدار سمت راست تقسیم کرده و نتیجه را در متغیر سمت چپ ذخیره می‌کند.

```
int x = 20;
x /= 4; // معادل x = x / 4 → x = 5
```

6. تخصیص باقیمانده (٪=): باقیمانده تقسیم متغیر سمت چپ بر مقدار سمت راست را در متغیر سمت چپ ذخیره می‌کند.

```
int x = 17;
x %= 5; // معادل x = x % 5 → x = 2
```

5.5. عملگرهای افزایش و کاهش

این عملگرها برای افزایش یا کاهش مقدار یک متغیر به اندازه یک واحد استفاده می‌شوند:

1. افزایش پسوندی (x++): ابتدا مقدار فعلی متغیر استفاده می‌شود، سپس یک واحد افزایش می‌یابد.

```
int x = 5;
int y = x++; // y = 5, x = 6
```

2. افزایش پیشوندی (++x): ابتدا متغیر یک واحد افزایش می‌یابد، سپس مقدار جدید استفاده می‌شود.

```
int x = 5;
int y = ++x; // x = 6, y = 6
```

3. کاهش پسوندی (x--): ابتدا مقدار فعلی متغیر استفاده می‌شود، سپس یک واحد کاهش می‌یابد.

```
int x = 5;
int y = x--; // y = 5, x = 4
```

4. کاهش پیشوندی (--x): ابتدا متغیر یک واحد کاهش می‌یابد، سپس مقدار جدید استفاده می‌شود.

```
int x = 5;
int y = --x; // x = 4, y = 4
```

5.6. اولویت عملگرها

اولویت عملگرها مشخص می‌کند که در یک عبارت چندعاملی، کدام عملگرها زودتر محاسبه می‌شوند. اگر اولویت عملگرها یکسان باشد، ارزیابی از چپ به راست انجام می‌شود. می‌توان با استفاده از پرانتز، اولویت را تغییر داد.

اولویت عملگرها از بالا به پایین (باالاترین اولویت به پایین‌ترین):

1. پرانتز (): بالاترین اولویت
2. افزایش/کاهش پیشوندی (++x, --x)
3. ضرب، تقسیم، باقیمانده (*, /, %)
4. جمع و تفریق (+, -)
5. عملگرهای رابطه‌ای (<, >, <=, >=)
6. عملگرهای برابری (==, !=)
7. منطقی AND (&&)
8. منطقی OR (||)
9. *تخصیص (=, +=, -=, *=, /=, %=)

10. افزایش/کاهش پسوندی (x++, x--)

مثال:

```
int result = 2 + 3 * 4; // نتیجه: 14 (اول ضرب، سپس جمع)
int result2 = (2 + 3) * 4; // نتیجه: 20 (اول پرانتز، سپس ضرب)
bool check = 5 > 3 && 10 < 20; // نتیجه: true (اول مقایسه‌ها، سپس AND)
```

فصل ششم

ساختارهای تصمیم‌گیری و شرطی

6.1. دستور **if** و کاربرد آن

دستور **if** ساده‌ترین و پایه‌ترین ساختار تصمیم‌گیری در برنامه‌نویسی است. این دستور امکان اجرای کدهای خاص را بر اساس یک شرط فراهم می‌کند. اگر شرط داخل پرانتز بعد از **if** درست (**true**) باشد، کدهای داخل بلوک **if** اجرا می‌شوند. در غیر این صورت، از بلوک **if** عبور می‌کنیم.

ساختار کلی:

```
if (شرط)
{
    // کدهایی که اگر شرط درست باشد اجرا می‌شوند
}
```

مثال :

```
int age = 20;
if (age >= 18)
{
    Console.WriteLine("شما بزرگسال هستید.");
}
```

در این مثال، اگر سن کاربر 18 یا بیشتر باشد، پیام "شما بزرگسال هستید" نمایش داده می‌شود. در غیر این صورت، هیچ کاری انجام نمی‌شود.

6.2. دستور if-else

دستور **if-else** توسعه‌یافته دستور **if** است. در این ساختار، اگر شرط درست باشد، کدهای بلوک **if** اجرا می‌شوند و اگر شرط نادرست باشد، کدهای بلوک **else** اجرا می‌شوند.

ساختار کلی:

```
if (شرط)
{
    // کدهایی که اگر شرط درست باشد اجرا می‌شوند
}
else
{
    // کدهایی که اگر شرط نادرست باشد اجرا می‌شوند
}
```

مثال :

```
int number = 7;
if (number % 2 == 0)
{
    Console.WriteLine("عدد زوج است.");
}
else
{
    Console.WriteLine("عدد فرد است.");
}
```

در این مثال، اگر عدد بر 2 بخش‌پذیر باشد (باقیمانده تقسیم بر 2 برابر صفر باشد)، پیام "عدد زوج است" نمایش داده می‌شود و در غیر این صورت، پیام "عدد فرد است" نمایش داده می‌شود.

6.3. دستور if-else if-else

زمانی که نیاز به بررسی چندین شرط مختلف داریم، از ساختار if-else if-else استفاده می‌کنیم. این ساختار امکان بررسی چندین شرط متوالی را فراهم می‌کند.

ساختار کلی:

```
if (شرط1)
{
    // کدهایی که اگر شرط1 درست باشد اجرا می‌شوند
}
else if (شرط2)
{
    // کدهایی که اگر شرط2 درست باشد اجرا می‌شوند
}
else if (شرط3)
{
    // کدهایی که اگر شرط3 درست باشد اجرا می‌شوند
}
else
{
    // کدهایی که اگر هیچکدام از شروط درست نباشند اجرا می‌شوند
}
```

مثال :

```
int score = 85;
if (score >= 90)
{
    Console.WriteLine("نمره: A (عالی)");
}
else if (score >= 80)
{
    Console.WriteLine("نمره: B (خوب)");
}
else if (score >= 70)
{
    Console.WriteLine("نمره: C (متوسط)");
}
else if (score >= 60)
{
    Console.WriteLine("نمره: D (قبول)");
}
else
{
    Console.WriteLine("نمره: F (مردود)");
}
```

نکته مهم: در ساختار if-else if، شرط‌ها به ترتیب بررسی می‌شوند و به محض اینکه یک شرط درست باشد، بلوک مربوطه اجرا شده و بقیه شرط‌ها بررسی نمی‌شوند. در مثال بالا، اگر نمره 85 باشد، شرط اول (score >= 90) نادرست است، پس به شرط دوم می‌رود .

شرط دوم (score >= 80) درست است، بنابراین پیام "نمره: B (خوب)" نمایش داده می‌شود و بقیه شرط‌ها بررسی نمی‌شوند.

6.4. دستور switch-case

استفاده از دستور switch مزایای متعددی دارد. اولاً خوانایی کد را افزایش می‌دهد، مخصوصاً زمانی که با چندین مقدار ثابت سروکار داریم. دوماً در برخی موارد کارایی بهتری نسبت به زنجیره if-else if دارد. سوماً مدیریت و نگهداری کد آسان‌تر می‌شود، به ویژه هنگام اضافه کردن موارد جدید.

از نکات مهم در استفاده از switch می‌توان به لزوم استفاده از break در پایان هر case (به جز caseهای خالی)، اختیاری بودن بخش default، و امکان استفاده از انواع داده‌های مختلف (اعداد، کاراکترها، رشته‌ها) در cases اشاره کرد. همچنین در نسخه‌های جدید سی‌شارپ، قابلیت‌های پیشرفته‌تری به switch اضافه شده است.

ساختار کلی:

```
switch (عبارت)
{
    case مقدار1:
        // کدهای مربوط به مقدار1
        break;
    case مقدار2:
        // کدهای مربوط به مقدار2
        break;
    case مقدار3:
        // کدهای مربوط به مقدار3
        break;
    default:
        // کدهایی که اگر هیچکدام از موارد مطابقت نداشتند اجرا می‌شوند
        break;
}
```

مثال :

```
int dayNumber = 3;
switch (dayNumber)
{
    case 1:
        Console.WriteLine("شنبه");
        break;
    case 2:
        Console.WriteLine("یکشنبه");
        break;
    case 3:
        Console.WriteLine("دوشنبه");
        break;
    case 4:
        Console.WriteLine("سه‌شنبه");
        break;
    case 5:
        Console.WriteLine("چهارشنبه
        break; ");
    case 6:
        Console.WriteLine("پنجشنبه");
        break;
    case 7:
        Console.WriteLine("جمعه");
        break;
    default:
        Console.WriteLine("عدد وارد شده معتبر نیست");
        break;
}
```

مزایا و نکات مهم در رابطه با switch:

استفاده از دستور switch مزایای متعددی دارد. اولاً خوانایی کد را افزایش می‌دهد، مخصوصاً زمانی که با چندین مقدار ثابت سروکار داریم. دوماً در برخی موارد کارایی بهتری نسبت به زنجیره if-else if دارد. سوماً مدیریت و نگهداری کد آسان‌تر می‌شود، به ویژه هنگام اضافه کردن موارد جدید. از نکات مهم در استفاده از switch می‌توان به لزوم استفاده از break در پایان هر case (به جز caseهای خالی)، اختیاری بودن بخش default، و امکان استفاده از انواع داده‌های مختلف (اعداد، کاراکترها، رشته‌ها) در cases اشاره کرد. همچنین در نسخه‌های جدید سی‌شارپ، قابلیت‌های پیشرفته‌تری به switch اضافه شده است.

6.5. ساختارهای تصمیم‌گیری تو در تو

تصمیم‌گیری‌های تودرتو به ساختارهایی اشاره دارد که در آنها یک دستور شرطی داخل دستور شرطی دیگر قرار می‌گیرد. این روش زمانی استفاده می‌شود که نیاز به بررسی شرایط پیچیده و چندلایه داریم. به عنوان مثال، ممکن است ابتدا بررسی کنیم که یک فرد سن قانونی دارد یا نه، و سپس در صورت داشتن سن قانونی، بررسی کنیم که گواهینامه رانندگی دارد یا خیر. اگرچه تصمیم‌گیری‌های تودرتو امکان بررسی شرایط پیچیده را فراهم می‌کنند، اما باید مراقب خوانایی کد بود. ساختارهای با بیش از 2-3 سطح تودرتو، می‌توانند کد را پیچیده و سخت‌فهم کنند. در چنین مواردی، بهتر است از توابع مجزا استفاده شود.

مثال:

```
int age = 25;
bool hasLicense = true;

if (age >= 18)
{
    Console.WriteLine("شما سن قانونی رانندگی را دارید.");

    if (hasLicense)
    {
        Console.WriteLine("و گواهینامه رانندگی نیز دارید.");
        Console.WriteLine("می‌توانید رانندگی کنید.");
    }
    else
    {
        Console.WriteLine("اما گواهینامه رانندگی ندارید.");
        Console.WriteLine("نمی‌توانید رانندگی کنید.");
    }
}
else
{
    Console.WriteLine("شما سن قانونی رانندگی را ندارید.");
}
```

در این مثال، ابتدا بررسی می‌شود که آیا سن کاربر 18 یا بیشتر است. اگر باشد، سپس بررسی می‌شود که آیا گواهینامه دارد یا نه. این ساختار تو در تو به ما امکان می‌دهد شرایط پیچیده‌تری را بررسی کنیم.

نکته مهم در ساختارهای تو در تو: باید مراقب خوانایی کد باشیم. اگر بیش از 2-3 سطح تودرتو داشته باشیم، کد پیچیده و سخت‌خوان می‌شود. در چنین مواردی، بهتر است از توابع مجزا استفاده کنیم.

فصل هفتم

حلقه‌ها و تکرار در برنامه‌نویسی

7.1. مفهوم حلقه و کاربرد آن

حلقه ساختاری در برنامه‌نویسی است که امکان اجرای مکرر یک بلوک کد را فراهم می‌کند. هدف اصلی حلقه‌ها، اجتناب از تکرار کد و اتوماسیون عملیات تکراری است. هنگامی که نیاز داریم یک عمل را چندین بار انجام دهیم، به جای نوشتن مکرر همان کد، از حلقه استفاده می‌کنیم. کاربردهای اصلی حلقه‌ها شامل پیمایش آرایه‌ها و مجموعه‌ها، انجام محاسبات تکراری (مانند جمع اعداد)، خواندن داده‌ها تا رسیدن به شرایط خاص (مانند پایان فایل) و ایجاد الگوها و اشکال می‌شود. حلقه‌ها یکی از اساسی‌ترین مفاهیم برنامه‌نویسی هستند که کنترل جریان برنامه را ممکن می‌سازند.

کاربردهای اصلی حلقه‌ها:

1. پیمایش آرایه‌ها و مجموعه‌ها: دسترسی به تمام عناصر یک آرایه
2. انجام محاسبات تکراری: مانند محاسبه مجموع اعداد
3. خواندن داده‌ها تا رسیدن به شرایط خاص: مانند خواندن از فایل تا پایان فایل
4. ایجاد الگوها: مانند رسم اشکال با کاراکترها
5. شبیه‌سازی فرآیندها: مانند شبیه‌سازی یک بازی

7.2. حلقه **for** و اجزای تشکیل‌دهنده آن

حلقه **for** یکی از پرکاربردترین و منظم‌ترین انواع حلقه‌هاست که زمانی استفاده می‌شود که تعداد تکرارها از قبل مشخص باشد یا بتوان آن را محاسبه کرد. ساختار حلقه **for** شامل سه بخش اصلی است: آغازگر (تعریف و مقداردهی متغیر شمارنده)، شرط (شرط ادامه حلقه) و گام (افزایش یا کاهش متغیر شمارنده). این حلقه به دلیل ساختار منظم و جمع‌وجوری که دارد، برای مواردی مانند پیمایش آرایه‌ها با اندیس مشخص، تولید دنباله‌های عددی و اجرای عملیات به تعداد مشخص، بسیار مناسب است. حلقه **for** کنترل کاملی بر روی فرآیند تکرار ارائه می‌دهد.

ساختار کلی:

```
for (آغازگر ;شرط ;گام)
{
    // کدهایی که تکرار می‌شوند
}


اجزای حلقه for:

1. آغازگر (Initializer): معمولاً یک متغیر شمارنده تعریف و مقداردهی اولیه می‌شود.
2. شرط (Condition): اگر true باشد، حلقه ادامه می‌یابد. اگر false باشد، حلقه پایان می‌یابد.
3. گام (Increment/Decrement): پس از هر بار اجرای کدهای حلقه، اجرا می‌شود.

مثال:


for (int i = 0; i < 5; i++)
{
    Console.WriteLine("شماره تکرار: " + i);
}


خروجی:

شماره تکرار: 0
شماره تکرار: 1
شماره تکرار: 2
شماره تکرار: 3
شماره تکرار: 4
```

مراحل اجرای حلقه for:

1. ابتدا آغازگر اجرا می‌شود (int i = 0)

2. شرط بررسی می‌شود (i < 5). اگر true باشد، به مرحله 3 می‌رود. اگر false باشد، حلقه پایان می‌یابد.

3. کدهای داخل حلقه اجرا می‌شوند.

4. گام اجرا می‌شود (++i)

5. به مرحله 2 بازمی‌گردد.

7.3. حلقه while

حلقه while نوعی از حلقه است که زمانی استفاده می‌شود که تعداد تکرارها از قبل مشخص نیست و ادامه حلقه تنها به یک شرط وابسته است. ساختار while ساده است: ابتدا شرط بررسی می‌شود و اگر true باشد، کدهای داخل حلقه اجرا می‌شوند. پس از هر اجرا، مجدداً شرط بررسی می‌شود. این حلقه برای مواقعی مناسب است که نمی‌دانیم دقیقاً چند بار باید عملیاتی تکرار شود، مانند خواندن داده از فایل تا رسیدن به پایان فایل، یا انتظار برای ورودی خاص از کاربر. در حلقه while باید مراقب بود که شرط در نهایت false شود تا حلقه پایان یابد.

ساختار کلی:

```
while (شرط)
{
  //کد هایی که تکرار می شوند
}
// مثال :
int i =0 ;
while(i < 5)
{
  Console.WriteLine ("شماره : " + i );
  i++;
}
```

تفاوت اصلی for و while:

- در حلقه for، شمارنده، شرط و گام در یک خط تعریف می‌شوند.
- در حلقه while، شمارنده قبل از حلقه تعریف می‌شود، شرط در پرانتز while نوشته می‌شود و افزایش/کاهش شمارنده داخل حلقه انجام می‌شود.

7.4. حلقه do-while

حلقه **while** و حلقه **for** هر دو برای تکرار عملیات استفاده می‌شوند اما تفاوت‌های مهمی دارند. حلقه **for** زمانی استفاده می‌شود که تعداد تکرارها از قبل مشخص باشد؛ تمام اجزای کنترل حلقه (شمارنده، شرط، گام) در یک خط تعریف می‌شوند. حلقه **while** زمانی مناسب است که تعداد تکرارها نامشخص باشد؛ شمارنده قبل از حلقه تعریف می‌شود، شرط در پرانتز **while** قرار می‌گیرد و تغییرات شمارنده داخل بدنه حلقه انجام می‌شود. حلقه **for** ساختار منظم‌تر و فشرده‌تری دارد، در حالی که حلقه **while** انعطاف‌پذیری بیشتری در شرایط پیچیده ارائه می‌دهد.

ساختار کلی:

```
do
{
    // کدهایی که تکرار می‌شوند
} while (شرط);
مثال:

int number;
do
{
    Console.Write("عدد مثبت وارد کنید: ");
    number = Convert.ToInt32(Console.ReadLine());
} while (number <= 0);

Console.WriteLine("عدد مثبت وارد کردید: " + number);
```

در این مثال، حتی اگر کاربر اولین بار عدد منفی وارد کند، حلقه ادامه می‌یابد تا زمانی که عدد مثبت وارد شود. کدهای داخل حلقه حداقل یک بار اجرا می‌شوند.

7.6. کنترل حلقه‌ها با break و continue

گاهی اوقات نیاز داریم حلقه را زودتر از موعد پایان دهیم یا برخی تکرارها را رد کنیم. برای این کار از دو دستور break و continue استفاده می‌کنیم.

دستور break:

دستور **break** قدرتمندترین ابزار کنترل حلقه است که باعث خروج کامل و فوری از حلقه می‌شود. هنگامی که **break** اجرا می‌شود، حلقه بلافاصله و بدون توجه به شرط یا تعداد تکرارهای باقی‌مانده، پایان می‌یابد. از **break** معمولاً زمانی استفاده می‌شود که هدف از حلقه محقق شده باشد و نیازی به ادامه تکرارها نباشد، یا وقتی که شرایط خاصی اتفاق افتاده که ادامه حلقه منطقی نیست. به عنوان مثال، در جستجوی یک مقدار در آرایه، به محض یافتن مقدار مورد نظر، از **break** برای خروج از حلقه استفاده می‌کنیم.

مثال:

```
for (int i = 0; i < 10; i++)
{
    if (i == 5)
    {
        break; // وقتی i به 5 برسد، حلقه پایان می‌یابد
    }
    Console.WriteLine(i);
}
خروجی: 4 3 2 1 0
```

دستور continue:

دستور continue یک ابزار کنترل جریان در حلقه‌ها است که باعث می‌شود اجرای بدنه حلقه در تکرار جاری متوقف شده و بلافاصله به تکرار بعدی منتقل شویم. هنگامی که continue اجرا می‌شود، بقیه دستورات بدنه حلقه در آن تکرار خاص نادیده گرفته شده و کنترل به ابتدای حلقه بازمی‌گردد (بعد از اجرای بخش گام در حلقه for). از continue معمولاً برای رد کردن موارد خاص در یک مجموعه استفاده می‌شود. به عنوان مثال، در حلقه‌ای که اعداد 1 تا 10 را پردازش می‌کند، می‌توان از continue استفاده کرد تا اعداد زوج پردازش نشوند.

نکته: break و continue فقط روی داخلی‌ترین حلقه اثر می‌گذارند. اگر حلقه‌های تو در تو داشته باشیم، break فقط از حلقه داخلی خارج می‌شود.

فصل هشتم

آرایه‌ها و مجموعه‌ها

8.1. مفهوم آرایه و نیاز به آن

آرایه (**Array**) ساختمان داده‌ای است که مجموعه‌ای از عناصر هم‌نوع را در خود ذخیره می‌کند. تمام عناصر آرایه از یک نوع داده هستند و در حافظه به صورت پیوسته ذخیره می‌شوند.

نیاز به آرایه زمانی احساس می‌شود که بخواهیم چندین مقدار هم‌نوع را با هم مدیریت کنیم. به جای تعریف چندین متغیر جداگانه، از یک آرایه استفاده می‌کنیم.

مثال:

اگر بخواهیم نمرات 100 دانشجو را ذخیره کنیم:

بدون آرایه: باید 100 متغیر جداگانه تعریف کنیم .

با آرایه: فقط یک آرایه با 100 عنصر تعریف می‌کنیم .

8.2. آرایه‌های یک بعدی

آرایه یک‌بعدی ساده‌ترین نوع آرایه است که عناصر در یک ردیف قرار می‌گیرند.

تعریف آرایه:

```
// روش اول: تعریف و سپس مقداردهی
int[] numbers; // تعریف آرایه
numbers = new int[5]; // ایجاد آرایه با 5 عنصر
// روش دوم: تعریف و ایجاد در یک خط
int[] scores = new int[10];
// روش سوم: تعریف با مقداردهی اولیه
int[] ages = { 20, 22, 19, 25, 23 };
```

دسترسی به عناصر آرایه:

عناصر آرایه با استفاده از **index**(شاخص) قابل دسترسی هستند. **index** از 0 شروع می‌شود.

```
int[] numbers = { 10, 20, 30, 40, 50 };
int first = numbers[0]; // 10
int third = numbers[2]; // 30
int last = numbers[4]; // 50
```

تغییر مقدار عناصر:

```
numbers[1] = 25; // تغییر عنصر دوم از 20 به 25
numbers[3] = numbers[0] + numbers[2]; // عنصر چهارم = 10 + 30 = 40
```

ویژگی **Length**:

هر آرایه یک ویژگی به نام **Length** دارد که تعداد عناصر آرایه را برمی‌گرداند.

```
int[] numbers = { 1, 2, 3, 4, 5 };
int size = numbers.Length; // 5
```

8.3. آرایه‌های چند بعدی

آرایه‌های چندبعدی برای ذخیره داده‌های جدولی یا ماتریسی استفاده می‌شوند. رایج‌ترین نوع، آرایه دو بعدی است.

آرایه دو بعدی:

```
// تعریف آرایه 3x3
int[,] matrix = new int[3, 3];
// مقداردهی اولیه
int[,] grid = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
// دسترسی به عناصر
int element = grid[1, 2]; // 6 = ردیف دوم، ستون سوم
```

آرایه سه بعدی و بیشتر:

```
// آرایه سه بعدی 2x3x4
int[,,] cube = new int[2, 3, 4];
```

8.4. پیمایش آرایه‌ها با حلقه

پیمایش آرایه یک بعدی:

```
int[] numbers = { 10, 20, 30, 40, 50 };
// با حلقه for
for (int i = 0; i < numbers.Length; i++)
{
    Console.WriteLine("عنصر " + i + ": " + numbers[i]);
}

// با حلقه foreach
foreach (int num in numbers)
{
    Console.WriteLine(num);
}
```

پیمایش آرایه دو بعدی:

```
int[,] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

for (int row = 0; row < 3; row++)
{
    for (int col = 0; col < 3; col++)
    {
        Console.Write(matrix[row, col] + " ");
    }
    Console.WriteLine();
}
```

خروجی:

```
1 2 3
4 5 6
7 8 9
```

8.5. عملیات رایج روی آرایه‌ها

1. محاسبه مجموع عناصر:

```
int[] numbers = { 1, 2, 3, 4, 5 };
int sum = 0;
foreach (int num in numbers)
{
    sum += num;
}
Console.WriteLine("مجموع: " + sum); // 15
```

2. یافتن بزرگترین و کوچکترین عنصر:

```
int[] numbers = { 12, 45, 23, 67, 34 };
int max = numbers[0];
int min = numbers[0];

for (int i = 1; i < numbers.Length; i++)
{
    if (numbers[i] > max)
        max = numbers[i];
    if (numbers[i] < min)
        min = numbers[i];
}

Console.WriteLine("بزرگترین: " + max); // 67
Console.WriteLine("کوچکترین: " + min); // 12
```

2. جستجوی یک مقدار در آرایه:

```
int[] numbers = { 10, 20, 30, 40, 50 };
int target = 30;
bool found = false;

foreach (int num in numbers)
{
    if (num == target)
    {
        found = true;
        break;
    }
}

if (found)
    Console.WriteLine("مقدار پیدا شد");
else
    Console.WriteLine("مقدار پیدا نشد");
```

4. مرتب‌سازی آرایه (مرتب‌سازی حبابی):

```
int[] numbers = { 5, 2, 8, 1, 9, 3 };

for (int i = 0; i < numbers.Length - 1; i++)
{
    for (int j = 0; j < numbers.Length - i - 1; j++)
    {
        if (numbers[j] > numbers[j + 1])
        {
            // جابجایی
            int temp = numbers[j];
            numbers[j] = numbers[j + 1];
            numbers[j + 1] = temp;
        }
    }
}

// آرایه مرتب شده: 1, 2, 3, 5, 8, 9
```

5. معکوس کردن آرایه:

```
int[] numbers = { 1, 2, 3, 4, 5 };
int[] reversed = new int[numbers.Length];

for (int i = 0; i < numbers.Length; i++)
{
    reversed[i] = numbers[numbers.Length - 1 - i];
}

// reversed: 5, 4, 3, 2, 1
```

7.5. حلقه‌ی foreach و محدودیت‌های این حلقه

حلقه foreach نوع خاصی از حلقه است که برای پیمایش مجموعه‌ها و آرایه‌ها طراحی شده است. این حلقه به طور خودکار روی تمام عناصر یک مجموعه حرکت می‌کند بدون نیاز به تعریف شمارنده یا بررسی اندیس. ساختار آن ساده است: foreach (نوع متغیر in مجموعه) { ... }. محدودیت‌های اصلی foreach شامل این موارد است: اولاً فقط برای خواندن عناصر مناسب است و نمی‌توان مقادیر را تغییر داد. دوماً فقط در یک جهت (از ابتدا به انتها) حرکت می‌کند. سوماً به اندیس عناصر دسترسی ندارد. با این حال، برای پیمایش ساده و کامل مجموعه‌ها، بسیار مناسب و امن است.

ساختار کلی:

```
foreach (نوع متغیر in مجموعه)
{
    // کدهایی که برای هر عنصر اجرا می‌شوند
}
مثال:
string[] names = { "علی", "رضا", "مریم", "سارا" };
foreach (string name in names)
{
    Console.WriteLine("نام: " + name);
}
خروجی:
نام: علی
نام: رضا
نام: مریم
نام: سارا
مزایای foreach:
```

مزایا:

ساده‌تر و خواناتر است .

نیاز به تعریف شمارنده ندارد .

خطای index out of range ندارد .

برای پیمایش کامل مجموعه مناسب است .

محدودیت‌های foreach:

فقط برای خواندن عناصر مجموعه مناسب است (نمی‌توان مقادیر را تغییر داد) .

فقط در یک جهت حرکت می‌کند .

به index عناصر دسترسی نداریم .

لیست(List)

در سی‌شارپ مشابه آرایه است، با این تفاوت که تعداد عناصر آن قابل تغییر است. به این معنا که شما می‌توانید در طول زمان به لیست، عناصر جدید اضافه کنید یا عناصری را از آن حذف کنید. لیست‌ها نسبت به آرایه‌ها انعطاف‌پذیرتر هستند و برای مواردی که تعداد عناصر مشخص نیست، بسیار مناسب هستند.

تعریف List و مقدار دهی به آن در زیر عنوان شده است که می توانید از هر روشی که نیاز دارید استفاده کنید.

```
static void Main(string[] args)
{
    //Type 1
    List<string> listOfNames = new List<string>()
        {
        "John Doe",
        "Jane Doe",
        "Joe Doe"
        };

    //Type 2
    List<string> listOfStrings = new List<string>();
    listOfStrings.Add("a string");

}
```

روشن است با هر تعداد بخواهیم می توانیم از Add استفاده کنیم و عنصر جدید به لیست اضافه کنیم. همچنین می توانیم از انواع داده های مختلف برای تعریف لیستها استفاده کنیم.

حذف از لیست

برای حذف عنصر از لیست روشهای مختلفی وجود دارد که در زیر برخی را آورده ایم.

```
static void Main(string[] args)
{
    List<string> listOfNames = new List<string>()
        {
        "John Doe",
        "Jane Doe",
        "Joe Doe"
        };

    listOfNames.Remove("Joe Doe"); // حذف عنصر با مقدار آن
    listOfNames.RemoveAt(0); // حذف عنصر با ایندکس مورد نظر
    listOfNames.RemoveAt(listOfNames.Count-1); // حذف آخرین عنصر
}
```

به منظور چاپ عناصر داخل لیست می شود به صورت زیر عمل کرد.

```
static void Main(string[] args)
{
    List<string> listOfNames = new List<string>()
        {
        "John Doe",
        "Jane Doe",
        "Joe Doe"
        };

    foreach (string name in listOfNames)
        Console.WriteLine(name);
}
```

توابع پر کاربرد برای لیست در زیر عنوان شده است.

```
static void Main(string[] args)
{
    List<string> listOfNames = new List<string>()
        {
        "John Doe",
        "Jane Doe",
        "Joe Doe"
        };

    listOfNames.Sort(); // مرتب کردن لیست
    listOfNames.Reverse(); // برعکس کردن لیست
    bool b = listOfNames.Contains("Jane"); // اگر اسم مورد نظر در لیست باشد مقدار true
    listOfNames.ElementAt(2); // حذف عنصر 2 از لیست
    listOfNames.Clear(); // حذف تمام عناصر موجود در لیست
}
```

دیکشنری Dictionary

از Dictionary می‌توان برای ذخیره مجموعه‌ای از داده‌ها، شبیه به List معمولی استفاده کرد. تفاوت اصلی این است که یک Dictionary می تواند عناصر را به عنوان جفت Key/Value ذخیره کند.Key ها باید منحصر به فرد و نمی توانند null باشند .Value ها می توانند تکراری یا null باشند.

ایجادDictionary

```
static void Main(string[] args)
{
    Dictionary<int, string> myDict = new Dictionary<int, string>();
}
```

در مثال بالا کلیدهای ما از نوع داده 'int' و مقادیر ما از نوع داده 'string'هستند. وقتی دیکشنری خود را ایجاد می کنیم، باید ابتدا نوع داده را برای کلیدها و مقادیر خود مشخص کنیم. ما می توانیم از انواع داده های دیگر نیز برای هر دو استفاده کنیم.

نحوه افزودن عناصر به Dictionary

برای افزودن عناصر به Dictionaryباید از تابع Add استفاده کنیم.یک نمونه از مثال افزودن عنصر به Dictionaryدر سی شارپ در زیر آمده است:

```
static void Main(string[] args)
{
    Dictionary<int, string> myDict = new Dictionary<int, string>();
    myDict.Add(1, "One");
    myDict.Add(4, "Four");
    myDict.Add(17, "Seventeen");
    foreach (var element in myDict)
    {
        Console.WriteLine(element.Key + ": " + element.Value);
    }
}
```

در مثال بالا، می‌توانیم ببینیم که با استفاده از تابع Add چند عنصر را به Dictionaryخود اضافه کرده‌ایم. با چاپ Dictionary و بررسی خروجی می‌توانیم تأیید کنیم که آنها با موفقیت اضافه شده‌اند.

همانطور که در مثال بالا می بینیم، همچنین می توانیم با موفقیت Key و Value های همه عناصر را نیز بازیابی کنیم.

بررسی اندازه Dictionary

برای بدست آوردن تعداد عناصر موجود در Dictionary می‌توانیم از ویژگی «Count» مانند مثال زیر استفاده کنیم:

```
static void Main(string[] args)
{
    Dictionary<int, string> myDict = new Dictionary<int, string>();
    myDict.Add(1, "One");
    myDict.Add(4, "Four");
    myDict.Add(17, "Seventeen");
    Console.WriteLine(myDict.Count);
}
```

در مثال بالا، می بینیم که ما چند عنصر را به Dictionaryخود اضافه و سپس نتیجه ویژگی countدر Dictionaryخود را چاپ کرده ایم.

بررسی وجود یک Key یا کلید خاص در Dictionary

برای بررسی وجود یک Key خاص در Dictionaryمی توان از تابع ()ContainsKey مانند مثال زیر استفاده کنیم:

```
static void Main(string[] args)
{
    Dictionary<int, string> myDict = new Dictionary<int, string>();
    myDict.Add(1, "One");
    myDict.Add(4, "Four");
    myDict.Add(17, "Seventeen");
    Console.WriteLine(myDict.ContainsKey(1));
    Console.WriteLine(myDict.ContainsKey(2));
}
```

در مثال بالا، می بینیم که ما چند عنصر را به Dictionaryخود اضافه و سپس برای بررسی اینکه آیا Dictionaryما حاوی Key های 1 و 2 است یا خیر نتایج توابعContainsKe() را چاپ کرده ایم.

بررسی وجود یک Value یا مقدار خاص درDictionary

برای بررسی وجود یک Valueخاص در Dictionary می توان از تابع ContainsValue() مانند مثال زیر استفاده کنیم:

```
static void Main(string[] args)
{
    Dictionary<int, string> myDict = new Dictionary<int, string>();
    myDict.Add(1, "One");
    myDict.Add(4, "Four");
    myDict.Add(17, "Seventeen");
    Console.WriteLine(myDict.ContainsValue("One"));
    Console.WriteLine(myDict.ContainsValue("Two"));
}
```

در مثال بالا، می بینیم که ما چند عنصر را به Dictionaryخود اضافه و سپس برای بررسی اینکه آیا Dictionaryما حاوی Value های «one» و «two» است یا خیر نتایج توابع ContainsValue() را چاپ کرده ایم.

حذف یک عنصر از Dictionary

برای حذف یک عنصر از Dictionary می توانیم از تابع Remove() مانند مثال زیر استفاده کنیم:

```
static void Main(string[] args)
{
    Dictionary<int, string> myDict = new Dictionary<int, string>();
    myDict.Add(1, "One");
    myDict.Add(4, "Four");
    myDict.Add(17, "Seventeen");
    myDict.Remove(1);
    foreach (var element in myDict)
    {
        Console.WriteLine(element.Key + ": " + element.Value);
    }
}
```

در مثال بالا، می بینیم که ما چند عنصر را به Dictionaryخود اضافه و سپس از تابع Remove()برای حذف عنصر با کلید '1' استفاده کرده ایم. با چاپ Dictionaryو بررسی خروجی، می‌توانیم تأیید کنیم که عنصر مربوطه را با موفقیت حذف کرده‌ایم.

پاک کردن Dictionary

برای اینکه یک Dictionary را کاملاً پاک کنیم می توانیم از تابع Clear() مانند مثال زیر استفاده کنیم:

```
static void Main(string[] args)
{
    Dictionary<int, string> myDict = new Dictionary<int, string>();
    myDict.Add(1, "One");
    myDict.Add(4, "Four");
    myDict.Add(17, "Seventeen");
    myDict.Clear();
    foreach (var element in myDict)
    {
        Console.WriteLine(element.Key + ": " + element.Value);
    }
}
```

در مثال بالا، ما چند عنصر را به Dictionaryخود اضافه و سپس تابع Clear() در Dictionaryرا صدا زده ایم. با چاپ Dictionary و بررسی خروجی، می توانیم تأیید کنیم که همه عناصر حذف شده اند. خروجی ما باید خالی باشد زیرا همه چیز را حذف کرده ایم.

فصل نهم

توابع و متدها در سی‌شارپ

9.1. مفهوم تابع و مزایای استفاده از آن

تابع (Function) یا متد (Method) بلوکی از کد است که یک کار خاص را انجام می‌دهد و می‌توان آن را از قسمت‌های مختلف برنامه فراخوانی کرد. توابع برای سازماندهی کد، جلوگیری از تکرار و افزایش خوانایی استفاده می‌شوند.

مزایای استفاده از توابع:

1. تقسیم مسئله: برنامه بزرگ به بخش‌های کوچک‌تر تقسیم می‌شود.
2. استفاده مجدد: می‌توان یک تابع را چندین بار فراخوانی کرد.
3. کاهش تکرار: کدهای تکراری در یک تابع نوشته می‌شوند.
4. آسان‌تر کردن دیباگ: خطایابی راحت‌تر می‌شود.
5. افزایش خوانایی: کد منظم‌تر و قابل فهم‌تر می‌شود.
6. تسهیل کار تیمی: هر برنامه‌نویس روی بخش‌های مختلف کار می‌کند.

9.2. ساختار تعریف تابع

ساختار کلی تعریف تابع در سی‌شارپ:

```
[نوع دسترسی] [نوع بازگشتی] [نام تابع]([پارامترها])
{
    // بدنه تابع
    [دستور return;] // اگر نوع بازگشتی void نباشد
}
```

اجزای تابع:

1. نوع دسترسی (**Access Modifier**): مشخص می‌کند تابع از کجا قابل دسترسی است (**public,** **private** و ...)
2. نوع بازگشتی (**Return Type**): نوع داده‌ای که تابع برمی‌گرداند (**void** اگر چیزی برنگرداند)
3. نام تابع (**Function Name**): نام منحصر به فرد تابع
4. پارامترها (**Parameters**): مقادیری که به تابع داده می‌شوند (اختیاری)
5. بدنه تابع (**Body**): کدهای اصلی تابع
6. دستور **return**: مقدار بازگشتی تابع (اگر نوع بازگشتی **void** نباشد)

9.3.پارامترهای تابع

پارامترهای تابع متغیرهایی هستند که در تعریف تابع مشخص می‌شوند و مقادیر آن‌ها در زمان فراخوانی تابع ارسال می‌شوند. پارامترها امکان ورود داده به تابع را فراهم می‌کنند و باعث انعطاف‌پذیری توابع می‌شوند. یک تابع می‌تواند بدون پارامتر، با یک پارامتر یا با چندین پارامتر تعریف شود. پارامترها می‌توانند انواع داده مختلفی داشته باشند و حتی می‌توان پارامترهای اختیاری تعریف کرد که در صورت عدم ارسال مقدار، از مقدار پیش‌فرض استفاده کنند. استفاده صحیح از پارامترها، قابلیت استفاده مجدد توابع را افزایش می‌دهد.

تابع بدون پارامتر:

تابع بدون پارامتر تابعی است که هیچ ورودی از بیرون دریافت نمی‌کند. این نوع توابع معمولاً برای انجام کارهای ثابت و مشخص استفاده می‌شوند. به عنوان مثال، تابعی که پیام خوشامدگویی نمایش می‌دهد یا تابعی که تاریخ و زمان جاری را برمی‌گرداند. ساختار چنین توابعی ساده است و فقط نام تابع به همراه پرانتزهای خالی نوشته می‌شود.

اگرچه این توابع انعطاف‌پذیری کمتری دارند، اما برای عملیاتی که همیشه به یک شکل انجام می‌شوند، مناسب و ساده هستند.

```
void SayHello()
{
    Console.WriteLine("Hello!");
}
```

تابع با پارامتر:

تابع با پارامتر تابعی است که یک یا چند ورودی از بیرون دریافت می‌کند. این ورودی‌ها در داخل تابع به عنوان متغیرهای محلی استفاده می‌شوند. به عنوان مثال، تابعی که دو عدد دریافت کرده و مجموع آن‌ها را برمی‌گرداند. پارامترها باعث می‌شوند تابع بتواند بر روی داده‌های مختلف عمل کند و در نتیجه قابلیت استفاده مجدد بالایی داشته باشد. هر پارامتر باید نوع داده مشخصی داشته باشد و نام مناسبی که نشان‌دهنده نقش آن باشد.

```
void Greet(string name)
{
    Console.WriteLine("Hello, " + name + "!");
}

// فراخوانی
Greet("Ali"); // Hello, Ali!
Greet("Maryam"); // Hello, Maryam!
```

تابع با چند پارامتر:

تابع با چند پارامتر تابعی است که بیش از یک ورودی دریافت می‌کند. این پارامترها با کاما از هم جدا می‌شوند و هر کدام نوع داده مشخصی دارند. به عنوان مثال، تابعی که نام، سن و آدرس یک شخص را دریافت کرده و اطلاعات را نمایش می‌دهد. هنگام فراخوانی چنین تابعی، باید مقادیر را به همان ترتیب تعریف شده ارسال کرد. توابع با چند پارامتر انعطاف‌پذیری زیادی دارند اما باید مراقب بود که تعداد پارامترها بیش از حد زیاد نشود چون خوانایی و استفاده از تابع را سخت می‌کند.

```
void DisplayInfo(string name, int age)
{
    Console.WriteLine("Name: " + name + ", Age: " + age);
}

// فراخوانی
DisplayInfo("Ali", 25); // Name: Ali, Age: 25
```

پارامترهای اختیاری:

پارامترهای اختیاری ویژگی‌ای در سی‌شارپ است که به پارامترهای تابع مقدار پیش‌فرض اختصاص می‌دهد. اگر در فراخوانی تابع، مقداری برای این پارامترها ارسال نشود، از مقدار پیش‌فرض استفاده می‌شود. پارامترهای اختیاری باید بعد از پارامترهای اجباری تعریف شوند. این ویژگی انعطاف‌پذیری توابع را افزایش می‌دهد و نیاز به تعریف چندین نسخه از یک تابع را کاهش می‌دهد. به عنوان مثال، تابعی که پیامی را چاپ می‌کند می‌تواند پارامتر اختیاری برای تعداد دفعات چاپ داشته باشد.

```
void PrintMessage(string message, int times = 1)
{
    for (int i = 0; i < times; i++)
    {
        Console.WriteLine(message);
    }
}

// فراخوانی
PrintMessage("Hello"); // یک بار چاپ می‌کند
PrintMessage("Hi", 3); // سه بار چاپ می‌کند
```

9.4. مقدار بازگشتی توابع

مقدار بازگشتی نتیجه‌ای است که تابع پس از اجرا به محل فراخوانی برمی‌گرداند. نوع بازگشتی در تعریف تابع مشخص می‌شود (به جز void که نشان‌دهنده عدم بازگشت مقدار است). تابع می‌تواند هر نوع داده‌ای را بازگرداند: اعداد، رشته‌ها، اشیاء و حتی آرایه‌ها. مقدار بازگشتی با دستور return مشخص می‌شود. استفاده از مقادیر بازگشتی، توابع را قدرتمندتر می‌کند زیرا می‌توانند نتایج محاسبات را به بخش‌های دیگر برنامه منتقل کنند.

تابع بدون مقدار بازگشتی (void):

تابع بدون بازگشتی تابعی است که نوع بازگشتی آن void است. این توابع پس از اجرای عملیات، هیچ مقداری برنمی‌گردانند. معمولاً برای انجام کارهای جانبی مانند نمایش اطلاعات، ذخیره داده یا تغییر وضعیت استفاده می‌شوند. به عنوان مثال، تابعی که پیامی را در کنسول نمایش می‌دهد یا تابعی که مقدار یک متغیر سراسری را تغییر می‌دهد. این توابع با دستور فراخوانی ساده استفاده می‌شوند و نیازی به ذخیره نتیجه ندارند.

```
void PrintSum(int a, int b)
{
    int sum = a + b;
    Console.WriteLine("Sum: " + sum);
}
```

تابع با مقدار بازگشتی:

```
int Add(int a, int b)
{
    return a + b;
}

// فراخوانی
int result = Add(5, 3); // result = 8
```

تابع با نوع بازگشتی متفاوت:

توابع می‌توانند انواع داده مختلفی را بازگردانند بسته به نیاز برنامه. به عنوان مثال، تابعی که محاسبات ریاضی انجام می‌دهد ممکن است int یا double بازگرداند. تابعی که وضعیت را بررسی می‌کند ممکن است bool بازگرداند. تابعی که اطلاعات را تولید می‌کند ممکن است string یا یک شیء پیچیده بازگرداند.

```
double Divide(int a, int b)
{
    return (double)a / b; // تقسیم اعشاری
}

string GetGrade(int score)
{
    if (score >= 90) return "A";
    else if (score >= 80) return "B";
    else if (score >= 70) return "C";
    else return "F";
}

bool IsEven(int number)
{
    return number % 2 == 0;
}
```

9.5. توابع بازگشتی

تابع بازگشتی (Recursive Function) تابعی است که خودش را فراخوانی می‌کند. این نوع توابع برای حل مسائلی که می‌توانند به زیرمسائل کوچک‌تر تقسیم شوند، مناسب هستند.

مثال فاکتوریل

فاکتوریل یک عدد صحیح مثبت (**n!**) حاصل ضرب تمام اعداد صحیح مثبت از 1 تا آن عدد است. به عنوان مثال، **5!** = 5 × 4 × 3 × 2 × 1 = 120. فاکتوریل یک مثال کلاسیک برای توابع بازگشتی است زیرا **n!** را می‌توان به صورت **n! × (n-1)** تعریف کرد. شرط توقف این بازگشت، **1!** = 1 یا **0!** =1 است. محاسبه فاکتوریل به صورت بازگشتی، بیان ساده و مستقیمی از تعریف ریاضی آن است، اگرچه برای اعداد بزرگ ممکن است روش‌های بهینه‌تری وجود داشته باشد.

```
int Factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * Factorial(n - 1);
}

// فراخوانی
int result = Factorial(5); // 5! = 5 * 4 * 3 * 2 * 1 = 120
```

مثال: سری فیبوناچی

سری فیبوناچی دنباله‌ای از اعداد است که هر عدد حاصل جمع دو عدد قبلی است (با شروع از 0 و 1). به صورت: **0, 1, 2, 3, 5, 8, 13, ...** این سری نیز مثال کلاسیک دیگری برای بازگشت است زیرا هر عدد فیبوناچی (**F(n)**) را می‌توان به صورت **F(n-1) + F(n-2)** تعریف کرد. شرط‌های توقف معمولاً **F(0) = 0** و **F(1) = 1** هستند. اگرچه محاسبه بازگشتی فیبوناچی ساده و زیباست، اما برای مقادیر بزرگ ناکارآمد است زیرا محاسبات تکراری زیادی انجام می‌دهد.

```
int Fibonacci(int n)
{
    if (n <= 0) return 0;
    if (n == 1) return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}

// فراخوانی
for (int i = 0; i < 10; i++)
{
    Console.Write(Fibonacci(i) + " ");
}
// 34 21 13 8 5 3 2 1 1 0 :خروجی
```

نکات مهم در توابع بازگشتی:

1. شرط توقف (Base Case): باید شرطی وجود داشته باشد که بازگشت متوقف شود.
2. فراخوانی بازگشتی: تابع باید خودش را با مقدار کوچک‌تر فراخوانی کند.
3. پیشرفت به سمت شرط توقف: هر فراخوانی باید به شرط توقف نزدیک‌تر شود.

9.6. محدوده دسترسی توابع

محدودیت‌های دسترسی (Access Modifiers) مشخص می‌کنند که توابع از کجا قابل دسترسی هستند:

1. public: از هر جایی قابل دسترسی است.
2. private: فقط از داخل کلاس خودش قابل دسترسی است.
3. protected: از داخل کلاس خودش و کلاس‌های مشتق شده قابل دسترسی است.
4. internal: از داخل همان اسمبلی (Assembly) قابل دسترسی است.

مثال:

```
public class Calculator
{
    // تابع public - از هر جایی قابل دسترسی
    public int Add(int a, int b)
    {
        return a + b;
    }

    // تابع private - فقط از داخل همین کلاس
    private int Multiply(int a, int b)
    {
        return a * b;
    }

    // تابع internal - فقط از داخل همین اسمبلی
    internal double Divide(int a, int b)
    {
        return (double)a / b;
    }
```

9.7. ارسال با ارجاع به تابع

تا اینجای کار ارسال مقادیر به توابع بصورت مقداری بود. برای توضیح بهتر این موضوع تکه کد زیر را بررسی خواهیم کرد.

```
static void Main(string[] args)
{
    int a = 12, b = 13, c = 0;

    Swap(a,b);
}

1 reference
static void Swap( int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

در برنامه بالا مشاهده می شود که تابعی برای جابجا کردن مقادیر دو متغیر a و b نوشته شده است و با فراخوانی این تابع (Swap) توقع داریم تا مقادیر این دو متغییر جابجا شود. ولی با کپی شده مقادیر a و b در پارامتر های x و y عمل جابجائی در تابع به درستی انجام می گیرد. اما با بازگشت به تابع اصلی خواهیم دید که این جابجایی روی متغیر های a و b صورت نگرفته است. دلیل این موضوع این است که در این مثال یک کپی از مقدار موجود در a و b در x و y قرار گرفته است و با جابجایی مقدار متغیر ها در تابع، اتفاقی بر روی متغیر های اصلی نخواهد افتاد.

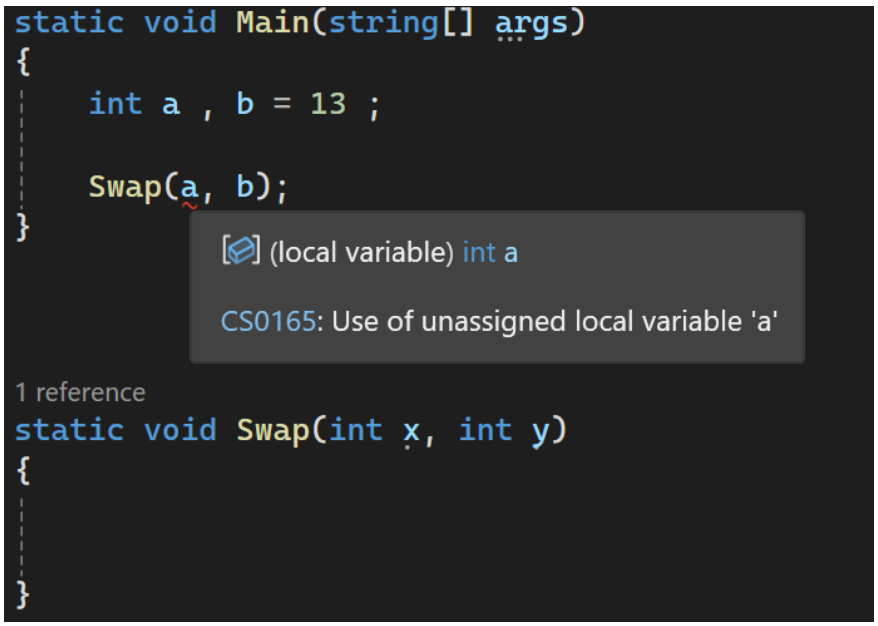
برای حل این مشکل می بایست نوع دیگری از فراخوانی را برای متغیر ها در نظر بگیریم. در کد زیر این نوع ارجاع را مورد استفاده قرار داده ایم.

```
static void Main(string[] args)
{
    int a = 12, b = 13, c = 0;

    Swap(ref a,ref b);
}

1 reference
static void Swap(ref int x,ref int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

با استفاده از کلمه ref در ارسال مقادیر و همچنین استفاده از همین کلمه در تعریف پارامتر های تابع یک فراخوانی با ارجاع صورت می پذیرد. با این کار هم a و هم x، همچنین b و y به متغیر مشترکی اشاره خواهند کرد و با جابجائی مقادیر در تابع این جابجایی در برنامه اصلی نیز دیده می شود.

کلمه کلیدی **out** و کاربرد آن

در زبان برنامه نویسی سی شارپ ما مجبور هستیم قبل از استفاده از یک متغیر آن را مقدار دهی کنیم.



در کد بالا مشاهده می شود که متغیر a تعریف شده است ولی مقدار دهی نشده است و دیده می شود هنگامی که قصد داریم ار آن به عنوان مقدار پارامتر تابع استفاده کنیم خطا اتفاق می افتد و همانطور که از خطای نمایش داده مشخص است، گفته می شود که متغیر دارد بدون مقدار دهی استفاده می شود.

ما قصد داریم مقداری برای این متغیر تعریف نکنیم و این کار را در جای دیگری مانند داخل تابع و یا هر جای دیگری انجام دهیم.

```
static void Main(string[] args)
{
    int a , b = 13 ;

    Swap(out a, b);
}

1 reference
static void Swap(out int x, int y)
{
    x =10;

}
```

برای رفع این مشکل میبینید با out به کامپایلر می گوئیم که مقدار دهی متغیر در جای دیگر انجام خواهد شد و قبل تعریف پارامتر x هم تاکید می کنیم این پارامتر در داخل این تابع تعریف خواهد شد. سپس مشاهده می شود که در تابع، تخصیص مقدار به متغیر صورت گرفته است و برنامه بدون هیچ مشکلی می تواند اجرا شود.

فصل دهم

ساختار

ساختار (Struct)

یک نوع داده‌ی کاربردی است که می‌تواند مقادیر متعدد و از انواع مختلف را در خود جای دهد. مثلا، یک struct می‌تواند شامل یک عدد صحیح، یک رشته و یک بولین باشد. این یک ویژگی خیلی قدرتمند است زیرا به شما اجازه می‌دهد تا داده‌ها را به شکل منطقی و مرتبط ذخیره کنید.

```
public struct Student
{
    public string name;
    public string family;
    public int Age;
}
```

در این مثال، Student یک struct است که شامل سه فیلد family, name و Age است.

نحوه استفاده از آن به صورت زیر می باشد.

```
static void Main(string[] args)
{
    Student s = new Student();
    s.name = "Ali";
    s.family = "Akbari";
    s.Age = 10;

}
```

تعریف آرایه ای از ساختار ها

در زیر آرایه ای از ساختار Student تعریف شده است و برای نمونه عنصر اول آن مقدار دهی شد. روشن است برای پر کردن کل آرایه می توان از حلقه استفاده کرد. در نهایت با یک حلقه foreach عناصر آرایه در خروجی چاپ می شود.

```
static void Main(string[] args)
{
    Student[] students = new Student[10];

    //مقدار دهی
    students[0].name = "Ali";
    students[0].family = "Akbari";
    students[0].Age = 10;

    //نمایش
    foreach (var item in students)
    {
        Console.WriteLine(item.name);
        Console.WriteLine(item.family);
        Console.WriteLine(item.Age);
    }
}
```

فصل یازدهم

مبانی برنامه‌نویسی شیءگرا

10.1. مفهوم شیءگرایی و مزایای آن

برنامه‌نویسی شیءگرا (Object-Oriented Programming - OOP) پارادایمی از برنامه‌نویسی است که در آن برنامه به صورت مجموعه‌ای از اشیاء در نظر گرفته می‌شود که با یکدیگر در تعامل هستند. هر شیء داده‌ها و رفتارهای مرتبط با خود را در یک واحد کپسوله می‌کند.

مفاهیم اصلی شیءگرایی:

1. کپسوله‌سازی (Encapsulation): پنهان‌سازی جزئیات داخلی
2. وراثت (Inheritance): ایجاد کلاس‌های جدید بر اساس کلاس‌های موجود
3. چندریختی (Polymorphism): توانایی ارسال پیام‌های یکسان به اشیاء مختلف
4. انتزاع یا تجرد (Abstraction): تمرکز روی بخش‌های مهم و حذف جزئیات غیرضروری

مزایای برنامه‌نویسی شیءگرا:

مزایای برنامه‌نویسی شیءگرا آن را به پارادایم غالب در توسعه نرم‌افزار تبدیل کرده است. مدولاریتی: برنامه به بخش‌های مستقل (کلاس‌ها) تقسیم می‌شود. استفاده مجدد: کدها از طریق وراثت و ترکیب مجدداً استفاده می‌شوند. قابلیت نگهداری: تغییرات به راحتی و فقط در یک مکان اعمال می‌شوند. مقیاس‌پذیری: برنامه‌های بزرگ به خوبی مدیریت می‌شوند. امنیت: کپسوله‌سازی از دسترسی غیرمجاز جلوگیری می‌کند. مدل‌سازی واقعی: اشیا دنیای واقعی به خوبی مدل می‌شوند.

10.2. کلاس‌ها و اشیاء

کلاس تعریف یا الگویی است که خصوصیات و رفتارهای یک شیء را مشخص می‌کند. کلاس مانند یک نقشه برای ساختن اشیاء است.

شیء نمونه‌ای از یک کلاس است که در حافظه ایجاد می‌شود و می‌تواند داده‌ها و رفتارهای تعریف شده در کلاس را داشته باشد.

مثال: کلاس "ماشین"

خصوصیات: رنگ، مدل، سرعت

رفتارها: حرکت کردن، توقف کردن، بوق زدن

```
// تعریف کلاس
public class Car
{
    // خصوصیات (Properties)
    public string Color { get; set; }
    public string Model { get; set; }

public int Speed { get; set; }

    // رفتارها (Methods)
    public void Accelerate(int increment)
    {
        Speed += increment;
        Console.WriteLine("سرعت به " + Speed + " افزایش یافت.");
    }

    public void Brake(int decrement)
    {
        Speed -= decrement;
        if (Speed < 0) Speed = 0;
        Console.WriteLine("سرعت به " + Speed + " کاهش یافت.");
    }
```

```
    public void Honk()
    {
        Console.WriteLine("بوق بوق!");
    }
}

// ایجاد شیء از کلاس
Car myCar = new Car();
myCar.Color = "قرمز";
myCar.Model = "پراید";
myCar.Speed = 0;

// استفاده از متدهای شیء
myCar.Accelerate(50); // سرعت به 50 افزایش یافت.
myCar.Honk(); // بوق بوق!
myCar.Brake(20); // سرعت به 30 کاهش یافت.
```

10.3. فیلدها و خصوصیات

فیلدها (Fields) متغیرهایی هستند که در سطح کلاس تعریف می‌شوند و داده‌های شیء را نگهداری می‌کنند.

خصوصیات (Properties) مکانیزمی برای دسترسی کنترل‌شده به فیلدها هستند.
خصوصیات می‌توانند دارای getter (برای خواندن) و setter (برای نوشتن) باشند.

```
public class Person
{
    // فیلدهای خصوصی
    private string name;
    private int age;

    // خصوصیات
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public int Age
    {
        get { return age; }
        set
        {
            if (value >= 0) // اعتبارسنجی
                age = value;
            else
                Console.WriteLine("سن نمی‌تواند منفی باشد.");
        }
    }

    // خصوصیات Auto-Implemented (از سی‌شارپ 3.0)
    public string Address { get; set; }
}
```

استفاده:

```
Person person = new Person();
person.Name = "علی"; // استفاده از setter
person.Age = 25; // استفاده از setter با اعتبارسنجی
person.Age = -5; // خطا: "سن نمی‌تواند منفی باشد."

string personName = person.Name; // استفاده از getter
```

10.4. متدها و رفتارها

متدها توابعی هستند که در داخل کلاس تعریف می‌شوند و رفتارهای شیء را تعریف می‌کنند.

```
public class BankAccount
{
    private double balance;

    public string AccountNumber { get; set; }
    public string Owner { get; set; }

// متدها
    public void Deposit(double amount)
    {
        if (amount > 0)
        {
            balance += amount;
            Console.WriteLine(amount + " تومان واریز شد. موجودی: " + balance);
        }
        else
        {
            Console.WriteLine("مبلغ واریزی باید مثبت باشد.");
        }
    }
```

```
public void Withdraw(double amount)
{
    if (amount > 0 && amount <= balance)
    {
        balance -= amount;
        Console.WriteLine(amount + " موجودی .شد برداشت برداشت: " + balance);
    }
    else if (amount > balance)
    {
        Console.WriteLine("موجودی کافی نیست.");
    }
    else
    {
        Console.WriteLine("مبلغ برداشت باید مثبت باشد.");
    }
}

public double GetBalance()
{
    return balance;
}
```

سازنده و مخرب ها- سازنده پیش‌فرض

سازنده متد خاصی است که هنگام ایجاد شیء از کلاس فراخوانی می‌شود. سازنده پیش‌فرض سازنده‌ای است که بدون پارامتر تعریف می‌شود. اگر هیچ سازنده‌ای تعریف نشود، کامپایلر به طور خودکار یک سازنده پیش‌فرض خالی ایجاد می‌کند. سازنده پیش‌فرض معمولاً مقادیر اولیه معقولی به فیلدهای کلاس اختصاص می‌دهد. به عنوان مثال، برای کلاس "دانشجو"، سازنده پیش‌فرض ممکن است نام را به "نامشخص" و شماره دانشجویی را به 0 مقداردهی کند.

```
public class Student
{
    public string Name { get; set; }
    public int StudentId { get; set; }
    public string Major { get; set; }

    // سازنده پیشفرض
    public Student()
    {
        Name = "نامشخص";
        StudentId = 0;
        Major = "نامشخص";
        Console.WriteLine("یک شیء Student ایجاد شد.");
    }

    // سازنده با پارامتر
```

سازنده با پارامتر سازنده‌ای است که پارامتر دریافت می‌کند و از آن‌ها برای مقداردهی اولیه فیلدهای کلاس استفاده می‌کند. این نوع سازنده امکان ایجاد اشیاء با حالت اولیه خاص را فراهم می‌آورد. به عنوان مثال، برای کلاس "دانشجو"، می‌توان سازنده‌ای تعریف کرد که نام و شماره دانشجویی را به عنوان پارامتر بگیرد و مستقیماً فیلدها را مقداردهی کند. سازنده با پارامتر انعطاف‌پذیری بیشتری در ایجاد اشیاء ارائه می‌دهد.

```
public Student(string name, int id, string major)
    {
        Name = name;
        StudentId = id;
        Major = major;
        Console.WriteLine("شیء Student برای " + name + " ایجاد شد.");
    }
    // سازنده کپی
```

سازنده کپی سازنده‌ای است که یک شیء از همان کلاس را به عنوان پارامتر گرفته و یک کپی از آن ایجاد می‌کند. این سازنده تمام فیلدهای شیء موجود را در شیء جدید کپی می‌کند. سازنده کپی برای ایجاد اشیاء جدید بر اساس اشیاء موجود مفید است. در سی‌شارپ، اگر سازنده کپی تعریف نشود، کامپایلر به طور پیش‌فرض یک سازنده کپی سطحی ایجاد می‌کند که ممکن است برای کلاس‌های پیچیده کافی نباشد.

```
public Student(Student other)
{
    Name = other.Name;
    StudentId = other.StudentId;
    Major = other.Major;
}

// مخرب
```

مخرب (Destructor) متد خاصی است که هنگام از بین رفتن شیء فراخوانی می‌شود. در سی‌شارپ، مخرب با نماد ~ و سپس نام کلاس تعریف می‌شود. مخرب‌ها برای پاکسازی منابع (مانند بستن فایل‌ها، آزاد کردن حافظه) استفاده می‌شوند. با این حال، در سی‌شارپ به دلیل

وجود زباله‌روب (Garbage Collector)، مخرب‌ها کمتر استفاده می‌شوند و معمولاً از واسط IDisposable برای مدیریت منابع استفاده می‌شود.

```
    ~Student()
    {
        Console.WriteLine("شیء Student برای " + Name + " از بین رفت.");
    }
```

استفاده :

```
// استفاده از سازنده پیشفرض
Student student1 = new Student();

// استفاده از سازنده با پارامتر
Student student2 = new Student("علی", 12345, "مهندسی نرم افزار");

// استفاده از سازنده کپی
Student student3 = new Student(student2);
```

فصل دوازدهم

مفاهیم پیشرفته شیءگرایی

11.1. کپسوله‌سازی

کپسوله‌سازی در سطح پیشرفته‌تر به معنی طراحی رابط‌های دقیق و محدود برای کلاس‌ها است. یک کلاس خوب باید حداقل لازم را در رابط عمومی خود ارائه دهد و جزئیات پیاده‌سازی را کاملاً پنهان کند. این شامل تعریف متدها و خصوصیات با سطوح دسترسی مناسب، استفاده از واسط‌ها برای تعریف قراردادها و جداسازی نگرانی‌ها است. کپسوله‌سازی پیشرفته، وابستگی بین بخش‌های مختلف برنامه را کاهش می‌دهد.

اهداف کپسوله‌سازی

اهداف اصلی کپسوله‌سازی شامل: پنهان‌سازی اطلاعات: کاربران نباید از جزئیات داخلی اطلاع داشته باشند. کاهش وابستگی: تغییرات داخلی نباید بر کاربران تأثیر بگذارد. افزایش امنیت: جلوگیری از دسترسی و تغییر نادرست داده‌ها. تسهیل تست‌پذیری: امکان تست واحدهای مجزا. افزایش قابلیت نگهداری: تغییرات آسان‌تر اعمال می‌شوند. انعطاف‌پذیری طراحی: امکان تغییر پیاده‌سازی بدون تأثیر بر رابط.

مثال:

```
public class BankAccount
{
    // فیلدهای خصوصی - از دسترس خارج هستند
    private string accountNumber;
    private double balance;
    private string owner;

    // خصوصیات برای دسترسی کنترل‌شده
    public string AccountNumber
    {
        get { return accountNumber; }
        private set { accountNumber = value; } // فقط درون کلاس قابل تنظیم
    }

    public double Balance
    {
        get { return balance; }
        private set { balance = value; } // فقط درون کلاس قابل تنظیم
    }

    public string Owner
    {
        get { return owner; }
        set { owner = value; }
    }
```

```
    // سازنده
    public BankAccount(string accNumber, string ownerName, double initialBalance)
    {
        AccountNumber = accNumber;
        Owner = ownerName;
        Balance = initialBalance;
    }

    // متدهای عمومی برای عملیات بانکی
    public void Deposit(double amount)
    {
        if (amount > 0)
        {
            Balance += amount;
            Console.WriteLine(amount + " واریز شد. موجودی جدید: " + Balance);
        }
    }

    public bool Withdraw(double amount)
    {
        if (amount > 0 && amount <= Balance)
        {
            Balance -= amount;
            Console.WriteLine(amount + " برداشت شد. موجودی جدید: " + Balance);
            return true;
        }
        Console.WriteLine("برداشت ناموفق.");
        return false;
    }
}
```

11.2. وراثت (Inheritance)

وراثت مزایای متعددی دارد که توسعه نرم‌افزار را کارآمدتر می‌سازد. استفاده مجدد از کد: می‌توان کدهای مشترک را در کلاس پایه نوشت و در کلاس‌های مشتق شده استفاده کرد. ایجاد سلسله مراتب منطقی: رابطه طبیعی "is-a" بین کلاس‌ها ایجاد می‌شود. قابلیت گسترش: می‌توان کلاس‌های موجود را گسترش داد بدون تغییر کد اصلی. کاهش پیچیدگی: کدهای تکراری حذف می‌شوند.

پایستگی: تغییرات در کلاس پایه به طور خودکار به کلاس‌های مشتق شده منتقل می‌شوند (با رعایت اصول طراحی).

```
// کلاس پایه
public class Vehicle
{
    public string Brand { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }

    public void Start()
    {
        Console.WriteLine("وسیله نقلیه روشن شد.");
    }

    public void Stop()
    {
        Console.WriteLine("وسیله نقلیه خاموش شد.");
    }

    public virtual void DisplayInfo()
    {
        Console.WriteLine("برند: " + Brand + "، مدل: " + Model + "، سال: " + Year);
    }
}

// کلاس مشتق شده
```

کلاس مشتق شده (Derived Class) کلاسی است که از یک کلاس پایه ارث‌بری می‌کند. این کلاس علاوه بر به ارث بردن همه اعضای عمومی و protected کلاس پایه، می‌تواند اعضای جدید اضافه کند یا رفتارهای ارث‌برده شده را تغییر دهد (با اورراید). کلاس مشتق شده رابطه "is-a" با کلاس پایه دارد. به عنوان مثال، کلاس "ماشین" می‌تواند از کلاس "وسیله نقلیه" مشتق شود. مشتق‌سازی امکان ایجاد سلسله مراتب تخصصی‌شده را فراهم می‌آورد.

```
public class Car : Vehicle
{
    public int NumberOfDoors { get; set; }

    // متد جدید
    public void Honk()
    {
        Console.WriteLine("بوق بوق!");
    }

    // Override متد والد
    public override void DisplayInfo()
    {
        base.DisplayInfo(); // فراخوانی متد والد
        Console.WriteLine("تعداد درها: " + NumberOfDoors);
    }
}

// کلاس مشتق شده دیگر
public class Motorcycle : Vehicle
{
    public bool HasSidecar { get; set; }

    // Override متد والد
    public override void DisplayInfo()
    {
        base.DisplayInfo();
        Console.WriteLine("دارای سایدکار: " + (HasSidecar ? "خیر" : "بله"));
    }
}
```

استفاده:

```
Car myCar = new Car();
myCar.Brand = "ایران خودرو";
myCar.Model = "پژو 206";
myCar.Year = 2020;
myCar.NumberOfDoors = 4;
myCar.Start();
myCar.Honk();
myCar.DisplayInfo();

Motorcycle myBike = new Motorcycle();
myBike.Brand = "هوندا";
myBike.Model = "CBR";
myBike.Year = 2021;
myBike.HasSidecar = false;
myBike.DisplayInfo();
```

11.3. چندریختی (Polymorphism)

چندریختی به توانایی اشیاء از کلاس‌های مختلف برای پاسخ دادن به یک پیام یکسان به روش‌های مختلف اشاره دارد. در سی‌شارپ، چندریختی از دو طریق پیاده‌سازی می‌شود:

1. چندریختی زمان اجرا (Runtime Polymorphism): با استفاده از متدهای مجازی و Override
2. چندریختی زمان کامپایل (Compile-time Polymorphism): با استفاده از Overloading

مثال چندریختی با وراثت:

```
public class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("رسم شکل کلی");
    }

    public virtual double GetArea()
    {
        return 0;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }

    public override void Draw()
    {
        Console.WriteLine("رسم دایره با شعاع " + Radius);
    }

    public override double GetArea()
    {
        return Math.PI * Radius * Radius;
    }
}
```

```
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public override void Draw()
    {
        Console.WriteLine("رسم مستطیل با ابعاد " + Width + "x" + Height);
    }

    public override double GetArea()
    {
        return Width * Height;
    }
}
```

استفاده از چندریختی:

```
// ایجاد آرایه‌ای از Shapeها
Shape[] shapes = new Shape[3];
shapes[0] = new Circle { Radius = 5 };
shapes[1] = new Rectangle { Width = 4, Height = 6 };
shapes[2] = new Circle { Radius = 3 };
```

```
// چندریختی در عمل
foreach (Shape shape in shapes)
{
    shape.Draw(); // هر شیء متد Draw خودش را فراخوانی می‌کند
    Console.WriteLine("مساحت: " + shape.GetArea());
    Console.WriteLine();
}
```

11.4. کلاس‌های انتزاعی (Abstract Classes)

کلاس انتزاعی کلاسی است که نمی‌توان از آن نمونه‌سازی کرد و فقط به عنوان کلاس پایه برای کلاس‌های دیگر استفاده می‌شود. کلاس انتزاعی می‌تواند شامل متدهای انتزاعی (بدون پیاده‌سازی) و متدهای معمولی (با پیاده‌سازی) باشد. کلاس‌های انتزاعی برای تعریف اساس مشترک یک خانواده از کلاس‌ها استفاده می‌شوند در حالی که برخی جزئیات را به کلاس‌های مشتق شده واگذار می‌کنند. این کلاس‌ها بین رابط‌های خالص و کلاس‌های کامل قرار می‌گیرند.

```
public abstract class Animal
{
    // خصوصیات
    public string Name { get; set; }
    public int Age { get; set; }

    // متد معمولی
    public void Sleep()
    {
        Console.WriteLine(Name + " در حال خوابیدن است.");
    }

    // متد انتزاعی (بدون پیاده‌سازی)
    public abstract void MakeSound();

    // متد انتزاعی دیگر
    public abstract void Move();
}
```

```
// کلاس مشتق شده
public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine(Name + " هاو هاو :می‌گوید!");
    }

    public override void Move()
    {
        Console.WriteLine(Name + " در حال دویدن است.");
    }
}
```

```
// کلاس مشتق شده دیگر
public class Bird : Animal
{
    public bool CanFly { get; set; }

    public override void MakeSound()
    {
        Console.WriteLine(Name + " جیک جیک :می‌گوید!");
    }

    public override void Move()
    {
        if (CanFly)
            Console.WriteLine(Name + " در حال پرواز است.");
        else
            Console.WriteLine(Name + " در حال راه رفتن است.");
    }
}
```

استفاده:

```
// نمی‌توان از Animal شیء ایجاد کرد
// Animal animal = new Animal(); // خطا

Dog myDog = new Dog();
myDog.Name = "رکس";
myDog.Age = 3;
myDog.MakeSound(); // هاو هاو!
myDog.Move(); // در حال دویدن است.
myDog.Sleep(); // در حال خوابیدن است.

Bird myBird = new Bird();
myBird.Name = "چیچی";
myBird.Age = 1;
myBird.CanFly = true;
myBird.MakeSound(); // جیک جیک!
myBird.Move(); // در حال پرواز است.
```

متد معمولی – متد انتزاعی

متد معمولی در کلاس انتزاعی، متدی است که پیاده‌سازی کامل دارد و می‌تواند توسط کلاس‌های مشتق شده استفاده یا اورراید شود. متد انتزاعی متدی است که فقط امضا دارد و فاقد پیاده‌سازی است. کلاس‌های مشتق شده از کلاس انتزاعی ملزم به پیاده‌سازی تمام متدهای انتزاعی هستند. متدهای انتزاعی قراردادهایی را تعریف می‌کنند که همه مشتق‌شده‌ها باید آن‌ها را رعایت کنند. این ترکیب انعطاف‌پذیری و اجبار را با هم ارائه می‌دهد.

11.5. واسط‌ها (Interfaces)

واسط (Interface) قراردادی است که فقط امضای متدها و خصوصیات را تعریف می‌کند (بدون پیاده‌سازی). یک کلاس می‌تواند چندین واسط را پیاده‌سازی کند. واسط‌ها برای تعریف قابلیت‌ها (توانایی‌ها) استفاده می‌شوند، نه ماهیت.

به عنوان مثال، واسط‌های IPrintable، IScannable. واسط‌ها انعطاف‌پذیری بیشتری نسبت به وراثت کلاس ارائه می‌دهند و اساس بسیاری از الگوهای طراحی و تزریق وابستگی هستند. در سی‌شارپ 8 به بعد، واسط‌ها می‌توانند پیاده‌سازی پیش‌فرض نیز داشته باشند.

```
// تعریف واسط
public interface IPrintable
{
    void Print();
}

public interface IScannable
{
    void Scan();
}

public interface IFaxable
{
    void SendFax();
    void ReceiveFax();
}
```

```
// کلاسی که چندین واسط را پیاده‌سازی می‌کند
public class MultiFunctionPrinter : IPrintable, IScannable, IFaxable
{
    public string Model { get; set; }

    public void Print()
    {
        Console.WriteLine("چاپ سند...");
    }

    public void Scan()
    {
        Console.WriteLine("اسکن سند...");
    }

    public void SendFax()
    {
        Console.WriteLine("ارسال فکس...");
    }

    public void ReceiveFax()
    {
        Console.WriteLine("دریافت فکس...");
    }
}
```

```
// واسط دیگر
public interface IShape
{
    double GetArea();
    double GetPerimeter();
}

public class Circle : IShape
{
    public double Radius { get; set; }

    public double GetArea()
    {
        return Math.PI * Radius * Radius;
    }

    public double GetPerimeter()
    {
        return 2 * Math.PI * Radius;
    }
}
```

استفاده:

```
MultiFunctionPrinter printer = new MultiFunctionPrinter();
printer.Model = "Xerox 123";

// استفاده از واسطها
IPrintable printable = printer;
printable.Print();

IScannable scannable = printer;
scannable.Scan();

IFaxable faxable = printer;
faxable.SendFax();

// آرایه‌ای از اشیاء IShape
IShape[] shapes = new IShape[2];
shapes[0] = new Circle { Radius = 5 };
shapes[1] = new Circle { Radius = 3 };

foreach (IShape shape in shapes)
{
    Console.WriteLine("مساحت: " + shape.GetArea());
    Console.WriteLine("محیط: " + shape.GetPerimeter());
}
```

کلاس‌های انتزاعی (Abstract Classes):

1. تعریف: کلاسی که نمی‌توان از آن نمونه‌سازی کرد و فقط به عنوان کلاس پایه استفاده می‌شود.
2. متدها: می‌تواند شامل متدهای انتزاعی (بدون پیاده‌سازی) و متدهای معمولی (با پیاده‌سازی) باشد.
3. فیلدها: می‌تواند فیلدها، خصوصیات و سازنده داشته باشد.
4. وراثت: یک کلاس فقط می‌تواند از یک کلاس انتزاعی ارث‌بری کند.
5. محدودیت دسترسی: می‌تواند اعضای protected داشته باشد.
6. کاربرد: زمانی استفاده می‌شود که کلاس‌های مشتق شده رابطه "is-a" دارند.

واسط‌ها (Interfaces):

1. تعریف: قراردادی که فقط امضای متدها و خصوصیات را تعریف می‌کند (تا سی‌شارپ 7.3).
2. متدها: همه متدها به طور پیش‌فرض abstract و public هستند.
3. فیلدها: نمی‌تواند فیلد داشته باشد (مگر ثوابت).
4. پیاده‌سازی: یک کلاس می‌تواند چندین واسط را پیاده‌سازی کند.
5. محدودیت دسترسی: همه اعضا public هستند.
6. کاربرد: زمانی استفاده می‌شود که کلاس‌ها رابطه "can-do" دارند.

فصل سیزدهم

اصول کدنویسی تمیز (Clean Code)

12.1. اصول نامگذاری

نامگذاری مناسب یکی از مهم‌ترین اصول کدنویسی تمیز است. نام‌های خوب باید:

1. معنادار باشند
2. قابل تلفظ باشند
3. قابل جستجو باشند
4. از کلمات کلیدی اجتناب کنند

قواعد نامگذاری در سی‌شارپ:

1. PascalCase: برای کلاس‌ها، متدها، خصوصیات، Namespaceها

```
public class StudentManager
{
    public void CalculateGrade() { }
    public string FirstName { get; set; }
}
```

2. camelCase: برای متغیرها، پارامترها

```
csharp
int studentCount;
string firstName;
double averageGrade;
```

```
3. UPPER_CASE: برای ثوابت

    const int MAX_SIZE = 100;
    const double PI_VALUE = 3.14159;
```

```
4. پیشوندهای رایج:
   · is, has, can برای بولین‌ها: isValid, hasPermission, canExecute
   · Get, Set, Calculate برای متدها: GetName(), SetValue(), CalculateTotal()
```

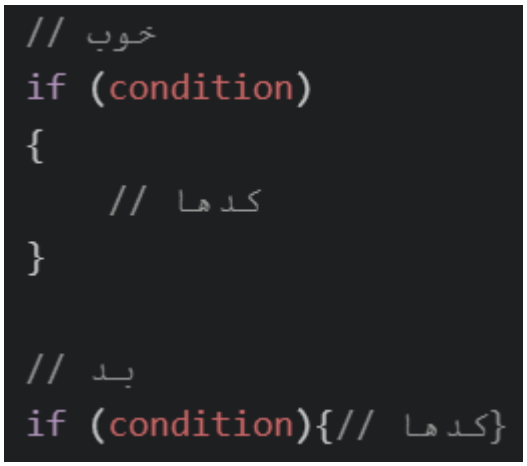
```
5. نام‌های معنادار:
    · بد: int d; (روز؟ فاصله؟ قطر؟)
    · خوب: int days;, int distance;, int diameter;
    · بد: var data = GetData();
    · خوب: var students = GetStudents();
```

12.2. ساختار کد و تو رفتگی

ساختار کد و تورفتگی مناسب، خوانایی کد را به شدت افزایش می‌دهد. تورفتگی (Indentation) معمولاً با 4 فضای خالی برای هر سطح انجام می‌شود. خطوط خالی برای جداکردن بخش‌های منطقی مختلف کد استفاده می‌شوند. طول خط بهتر است بیش از 80-120 کاراکتر نباشد. ترتیب منطقی در کلاس‌ها: ابتدا فیلدها، سپس خصوصیات، سپس سازنده، سپس متدها. ساختار منظم، درک کد را آسان‌تر کرده و خطاهای syntactical را کاهش می‌دهد.

قواعد ساختار کد:

قواعد ساختار کد شامل دستورالعمل‌هایی برای سازماندهی فیزیکی کد است. یک دستور در هر خط (مگر در موارد خاص مانند تعریف متغیرهای مرتبط). فضاگذاری مناسب اطراف عملگرها و بعد از کاما. آکولادها حتی برای بلوک‌های تک‌خطی (برای جلوگیری از خطاهای رایج). ترتیب usingها: ابتدا System، سپس دیگر Namespaceها. کد مرده (کدهای استفاده نشده) باید حذف شوند. رعایت این قواعد، یکپارچگی کد در تیم‌های بزرگ را حفظ می‌کند.



4. ترتیب منطقی:

- ابتدا using statements
- سپس namespace
- سپس کلاس‌ها
- در داخل کلاس: ابتدا فیلدها، سپس خصوصیات، سپس سازنده، سپس متدها

5. استفاده از خطوط خالی برای جداکردن بخش‌های منطقی:

```
public class Example
{
    // فیلدها
    private int count;

    // خصوصیات
    public string Name { get; set; }

    // سازنده
    public Example()
    {
        // ...
    }

    // متدها
    public void Method1()
    {
        // ...
    }

    // خط خالی برای جداکردن متدها
    public void Method2()
    {
        // ...
    }
}
```

12.3. کامنت‌گذاری مناسب

کامنت گذاری مناسب هنری است که بین کامنت زیاد (که می‌تواند منسوخ شود) و کامنت کم (که کد را مبهم می‌کند) تعادل برقرار می‌کند. کامنت‌ها باید چرایی کد را توضیح دهند، نه چگونگی آن. کد خوب باید تا حد امکان خود-مستند باشد. کامنت‌های خوب مانند راهنمای تور برای کد هستند که تصمیمات طراحی، فرضیات و هشدارها را توضیح می‌دهند.

انواع کامنت:

1. کامنت تک خطی:

```
// محاسبه میانگین نمرات
double average = total / count;
```

2. کامنت چند خطی:

```
/*
این متد نمرات دانشجو را محاسبه می‌کند
پارامترها:
- scores: آرایه‌ای از نمرات
بازگشتی: میانگین نمرات
*/
```

3. کامنت **XML Documentation** (برای مستندسازی):

```
/// <summary>
/// محاسبه میانگین نمرات دانشجو
/// </summary>
/// <param name="scores">آرایه‌ای از نمرات</param>
/// <returns>میانگین نمرات</returns>
public double CalculateAverage(double[] scores)
{
    // ...
}
```

چه زمانی کامنت بنویسیم؟

کامنت بنویسیم وقتی که: منطق پیچیده‌ای وجود دارد که با نگاه کردن به کد واضح نیست. تصمیمات طراحی مهمی گرفته شده که ممکن است برای دیگران عجیب به نظر برسد. کارایی غیربدیهی وجود دارد (مانند بهینه‌سازی‌هایی که خوانایی را کم کرده‌اند). هشدارهایی درباره استفاده از کد وجود دارد. کدی از کتابخانه خارجی استفاده شده که رفتارش مشخص نیست. الگوریتم‌های پیچیده پیاده‌سازی شده‌اند.

چه زمانی کامنت ننویسیم؟

کامنت ننویسیم وقتی که: کد ساده و واضح است و خودش گویاست. فقط کد را تکرار می‌کنیم (کامنت باید اطلاعات افزوده‌ای ارائه دهد). کامنت قدیمی و به‌روز نشده است (کامنت غلط از بی‌کامنت بدتر است). می‌توان کد را ساده‌تر کرد تا نیاز به کامنت از بین برود. نام‌های معنادار همه چیز را روشن کرده‌اند.

12.4. اصول طراحی SOLID

SOLID پنج اصل مهم برای طراحی نرم‌افزار شیءگرا است:

1. **Single Responsibility Principle (SRP)**: هر کلاس باید فقط یک دلیل برای تغییر داشته باشد.

```
// بد: یک کلاس که دو مسئولیت دارد
class Report
{
    public void GenerateReport() { /* تولید گزارش */ }
    public void PrintReport() { /* چاپ گزارش */ }
}


// خوب: دو کلاس با مسئولیت‌های جدا
class ReportGenerator
{
    public void Generate() { /* تولید گزارش */ }
}

class ReportPrinter
{
    public void Print() { /* چاپ گزارش */ }
}
```

2. **Open/Closed Principle (OCP)**: کلاس‌ها باید برای گسترش باز و برای تغییر بسته باشند.

```
// استفاده از واسط برای قابلیت گسترش
public interface IDiscount
{
    double Apply(double amount);
}

public class RegularDiscount : IDiscount { /* ... */ }
public class PremiumDiscount : IDiscount { /* ... */ }
```

3. Liskov Substitution Principle (LSP): اشیاء کلاس پایه باید بتوانند با اشیاء کلاس مشتق شده جایگزین شوند.

```
// کلاس مشتق شده نباید رفتار کلاس پایه را بشکند

public class Bird
{
    public virtual void Fly() { /* پرواز */ }
}

public class Penguin : Bird
{
    public override void Fly()
    {
        throw new NotSupportedException("پنگوئن‌ها نمی‌توانند پرواز کنند!");
    }
}
// این طراحی LSP را نقض می‌کند
```

4. Interface Segregation Principle (ISP): واسط‌های بزرگ را به واسط‌های کوچک‌تر و تخصصی‌تر تقسیم کنید.

```
// بد: واسط بزرگ
interface IWorker
{
    void Work();
    void Eat();
    void Sleep();
}

// خوب: واسط‌های جدا
interface IWorkable { void Work(); }
interface IEatable { void Eat(); }
interface ISleepable { void Sleep(); }
```

5. Dependency Inversion Principle (DIP): به انتزاع‌ها وابسته باشید، نه به جزئیات پیاده‌سازی.

```
// بد: وابستگی مستقیم
class EmailService
{
    public void SendEmail() { /* ... */ }
}

class Notification
{
    private EmailService emailService = new EmailService();
}

// خوب: وابستگی به واسط
interface IMessageService
{
    void Send();
}

class Notification
{
    private IMessageService messageService;

    public Notification(IMessageService service)
    {
        messageService = service;
    }
}
```

12.5. مدیریت خطاها

مدیریت صحیح خطاها بخش مهمی از کدنویسی تمیز است.

استفاده از try-catch:

```
try
{
    // کدی که ممکن است خطا دهد
    int result = Divide(10, 0);
}
catch (DivideByZeroException ex)
{
    // مدیریت خطای خاص
    Console.WriteLine("خطا: تقسیم بر صفر مجاز نیست.");
    LogError(ex);
}
catch (Exception ex)
{
    // مدیریت سایر خطاها
    Console.WriteLine("یک خطای ناشناخته رخ داد: " + ex.Message);
}
finally
{
    // کدی که همیشه اجرا می‌شود
    Console.WriteLine("پایان عملیات.");
}
```

ایجاد خطاهای سفارشی:

```
public class InvalidAgeException : Exception
{
    public InvalidAgeException() : base("سن نامعتبر است.") { }

    public InvalidAgeException(string message) : base(message) { }

    public InvalidAgeException(string message, Exception inner)
        : base(message, inner) { }
}

// استفاده
public void SetAge(int age)
{
    if (age < 0 || age > 150)
        throw new InvalidAgeException("سن باید بین 0 و 150 باشد.");

    this.age = age;
}
```

اصول مدیریت خطا:

1. خطاها را زود تشخیص دهید
2. خطاها را به سطح مناسب منتقل کنید
3. پیام خطای معنادار ارائه دهید
4. از خطاهای عمومی اجتناب کنید
5. لاگ مناسب از خطاها نگه دارید

مثال کاربردی:

```
public class FileProcessor
{
    public void ProcessFile(string filePath)
    {
        if (string.IsNullOrEmpty(filePath))
            throw new ArgumentException("مسیر فایل نمی‌تواند خالی باشد.");

        if (!File.Exists(filePath))
            throw new FileNotFoundException("فایل پیدا نشد: " + filePath);

try
        {
            string content = File.ReadAllText(filePath);

            // پردازش محتوا
        }
        catch (IOException ex)
        {
            throw new ApplicationException("خطا در خواندن فایل.", ex);
        }
        finally
        {
            // منابع را آزاد کن
        }
    }
}
```