# الگوی Singleton در #C

## تعریف الگوی Singleton

الگوی Singleton یک الگوی طراحی **خلاقی** (Creational) است که تضمین می‌کند یک کلاس فقط **یک نمونه** (Instance) داشته باشد و یک نقطه دسترسی سراسری به آن نمونه فراهم می‌کند. این الگو زمانی مفید است که دقیقاً یک شی برای هماهنگ کردن اقدامات در سراسر سیستم نیاز باشد.

## مشکل اصلی که Singleton حل می‌کند

در بسیاری از scenarios، ما نیاز داریم که:

* فقط یک نمونه از یک کلاس در برنامه وجود داشته باشد
* آن نمونه به راحتی در دسترس همه بخش‌های برنامه باشد
* از ایجاد نمونه‌های متعدد به صورت تصادفی جلوگیری شود

مثال‌های واقعی:

* مدیریت تنظیمات برنامه
* اتصال به پایگاه داده
* سیستم لاگینگ
* کش (Cache) برنامه

## ساختار Singleton

public sealed class Singleton

{

// 1. ذخیره‌سازی تنها نمونه کلاس

private static Singleton \_instance;

// 2. سازنده خصوصی برای جلوگیری از ایجاد نمونه جدید

private Singleton() { }

// 3. متد عمومی برای دسترسی به نمونه

public static Singleton Instance {

get {

if (\_instance == null) { \_instance = new Singleton(); }

return \_instance;

}

}

// 4. سایر متدهای کسب و کار

public void DoSomething() { Console.WriteLine("Doing something..."); }

}

## جزئیات پیاده‌سازی

1. **کلاس sealed**: از ارث‌بری و ایجاد زیرکلاس‌ها جلوگیری می‌کند.
2. **سازنده خصوصی**: مانع از ایجاد نمونه با عملگر new می‌شود.
3. **متغیر استاتیک**: تنها نمونه کلاس را ذخیره می‌کند.
4. **ویژگی Instance**: دسترسی کنترل‌شده به نمونه منفرد را فراهم می‌کند.

## نسخه Thread-Safe

برای محیط‌های چندنخی، باید از race condition جلوگیری کرد:

public sealed class ThreadSafeSingleton

{

private static readonly object \_lock = new object();

private static ThreadSafeSingleton \_instance;

private ThreadSafeSingleton() { }

public static ThreadSafeSingleton Instance

{

get

{

if (\_instance == null)

{

lock (\_lock)

{

if (\_instance == null)

{

\_instance = new ThreadSafeSingleton();

}

}

}

return \_instance;

}

}

}

## مزایای Singleton

1. **کنترل دسترسی**: فقط یک نقطه دسترسی به نمونه وجود دارد.
2. **صرفه‌جویی در منابع**: از ایجاد نمونه‌های تکراری جلوگیری می‌کند.
3. **انعطاف‌پذیری**: می‌تواند به راحتی تغییر کند تا تعداد محدودی نمونه ایجاد کند.
4. **دسترسی سراسری**: از هر جای برنامه قابل دسترسی است.

## معایب Singleton

1. **تست‌پذیری**: می‌تواند تست واحد را دشوار کند.
2. **تخلف از اصول SOLID**: ممکن است اصل Single Responsibility را نقض کند.
3. **حالت سراسری**: می‌تواند منجر به مشکلات طراحی شود.
4. **مشکلات چندنخی**: نیاز به مدیریت صحیح در محیط‌های موازی دارد.

## مثال کاربردی در #C

public sealed class AppConfig

{

private static readonly Lazy<AppConfig> \_instance =

new Lazy<AppConfig>(() => new AppConfig());

public static AppConfig Instance => \_instance.Value;

public string DbConnectionString { get; private set; }

public int MaxConnections { get; private set; }

public bool DebugMode { get; private set; }

private AppConfig()

{

// بارگذاری تنظیمات از فایل کانفیگ

DbConnectionString = "Server=...";

MaxConnections = 10;

DebugMode = true;

}

public void ReloadConfig()

{

// منطق بارگذاری مجدد تنظیمات

}

}

// استفاده:

var config = AppConfig.Instance;

Console.WriteLine(config.DbConnectionString);

## بهترین روش‌های استفاده

1. از **Lazy<T>** برای پیاده‌سازی lazy initialization استفاده کنید.
2. برای سناریوهای تست‌پذیرتر، از **Dependency Injection** با Singleton استفاده کنید.
3. فقط زمانی از Singleton استفاده کنید که واقعاً به یک نمونه منفرد نیاز دارید.
4. از Singleton برای اشیا با حالت (stateful) که نیاز به هماهنگی دارند استفاده کنید.

## جایگزین‌های مدرن

در #C مدرن و با وجود سیستم‌های DI مانند:

* **Microsoft.Extensions.DependencyInjection**
* **Autofac**
* **Ninject**

می‌توان Singleton را به صورت زیر پیاده‌سازی کرد:

services.AddSingleton<IMyService, MyService> ();

این روش مزایای بیشتری از جمله تست‌پذیری بهتر و انعطاف‌پذیری بیشتر دارد.

-------------------------------------------------------------------------------------------------------