# Clase generice şi avantajele utilizării lor

## Introducere

Genericurile au fost lansate în versiunea 2.0 a limbajului C# în 2005. Înainte de această versiune, programatorii trebuiau să scrie cod pentru fiecare tip de date pe care îl foloseau în cadrul unei clase. De exemplu, dacă aveți nevoie de o listă de numere întregi, o listă de șiruri de caractere și o listă de numere reale, atunci aveți nevoie de trei clase diferite. Cu ajutorul clasei generice, puteți crea o clasă care poate fi utilizată pentru a stoca orice tip de date. De exemplu, o clasă generică poate fi utilizată pentru a crea o listă, o coadă, o stivă sau un dictionar. Exista de asemenea opțiunea de a folosi obiectele de tip Object, însă la fiecare utilizare trebuie să fie verificată clasa obiectului, încălcând principiul Don't Repeat Yourself, iar în cazul variabilelor de tip valoare acestea trebuiesc convertite în obiect prin procesul de boxing, ceea ce total elimină toate avantajele de performanță a variabilelor din stack.

## Analiza claselor generice

### Structura claselor generice

Comparativ cu clasele simple, cele generice sunt de fapt doar șabloane ce se folosesc pentru generarea mai multor clase noi, numite clase construite, aplicând argumenții de tip. Argumenții dați se scriu între paranteze unghiulare ( “<” și “>” ) direct după denumirea tipului generic, și aceștia vor defini denumirea tipului necunoscut ce va fi utilizat mai târziu. De asemenea la definirea clasei pot fi setate limite pentru argument, precum necesitatea de a fi clasă, struct, sau chiar de a moșteni anumite clase sau interfețe. De exemplu:

public abstract class Settings<T> : ScriptableObject where T : Settings<T>  
{  
 //cod  
}

La crearea unui obiect dintr-un șablon este necesară definirea tipului pentru argumentul necunoscut. În urma specificării acestui fapt, la generarea obiectului specializat în fon se va crea o clasă separată ce realizează șablonul anume pentru tipul dat, adică Stack<int> și Stack<long> sunt două clase total diferite, precum și variabilele statice ale acestora.

public class GenericStuff<T>  
{  
 public static int SomeValue;  
}  
  
public static void Main()  
{  
 GenericStuff<int>.SomeValue = 1;  
 GenericStuff<long>.SomeValue = 2;  
 Console.WriteLine($"{GenericStuff<int>.SomeValue} {GenericStuff<long>.SomeValue}");  
 //1 2  
}

### Comparația claselor generice cu folosirea Object-urilor

Avantajele genericurilor vin din faptul că nu este nevoie de făcut cast repetat din Object pentru orice operație cu obiectul necunoscut, compilatorul pur și simplu va returna valoarea de tip dat mai sus, adică un obiect myStack de tip Stack<int> întotdeauna va returna un int după metoda myStack.Pop(). Acest fapt îmbunătățește semnificativ performanța, deoarece nu este nevoie de pierdut timp la cast în procesul de boxing și unboxing, IntelliSense va detecta imediat ce tip de date se folosește, astfel imediat completând codul și mărind viteza de scriere, și în final elimină cod inutil ce altfel ar trebui repetat la fiecare convertire a necunoscutei.

var genericStack = new Stack<int>();  
genericStack.Push(1);  
Console.WriteLine(genericStack.Pop() + 1);  
  
var nonGenericStack = new Stack();  
nonGenericStack.Push(1);  
Console.WriteLine((int)nonGenericStack.Pop() + 1);

De asemenea, compilatorul imediat va observa orice eroare de tip la interacțiunea cu necunoscuta, deoarece el își poate da seama de ce tip trebuie să fie necunoscuta în momentul necesar. Object-urile pur și simplu nu oferă așa posibilitate, deci toate erorile pot fi observate doar la runtime, garantând o mulțime de erori ascunse și greu de depanat.

var genericStack = new Stack<int>();  
// eroare identificată la compilare  
genericStack.Push(new object());  
Console.WriteLine(genericStack.Pop() + 1);  
  
var nonGenericStack = new Stack();  
nonGenericStack.Push(new object());  
//eroare ridicată la executare  
Console.WriteLine((int)nonGenericStack.Pop() + 1);

E important de menționat faptul că genericurile nu sunt soluția oricarei probleme în legătură cu tipuri necunoscute. Genericurile ne permit crearea unor colecții doar omogene, adică obiectele, fie ele chiar și de tip necunoscut, trebuie să fie de același tip, pe când Object-urile nu au așa limitare, ceea ce permite crearea colecțiilor eterogene. Exemple de asemenea colecții sunt în namespace-urile System.Collections și System.Collections.Generic

var genericStack = new Stack<int>();  
genericStack.Push(1);  
//eroare la compilare  
genericStack.Push(1.5);  
  
var nonGenericStack = new Stack();  
nonGenericStack.Push(1);  
nonGenericStack.Push(1.5);  
nonGenericStack.Push(DateTime.Now);  
nonGenericStack.Push(new object());  
//nicio eroare

## Exemple de utilizare a claselor generice

Genericurile au obținut o popularitate înaltă datorită compatibilității înalte a necunoscutelor cu compilatorul, astfel întâlnindu-se în majoritatea pachetelor pentru C#. Spre exemplu, pachetul MediatR, des utilizat pentru realizarea arhitecturii backend-urilor de tip Clean Architecture, folosește clase generice pentru a cunoaște ce tipuri de date va necesita un Handler și ce tip va returna acesta, oferind siguranța de tipuri garantată de genericuri:

public class DoGenericStuffQuery : IRequest<GenericStuff> {  
 public string SomeData { get; set; }  
}  
  
public class DoGenericStuffQueryHandler : IRequestHandler<DoGenericStuffQuery, GenericStuff> {  
 public async Task<GenericStuff> Handle(DoGenericStuffQuery query, CancellationToken cancellationToken) {  
 return await DoStuff(query.SomeData);  
 }  
}  
  
public class GenericStuffController {  
 private readonly IMediator \_mediator  
  
 public GenericStuffController(IMediator mediator) {  
 \_mediator = mediator;  
 }  
  
 [HttpPost]  
 public async Task<IActionResult> CallGenericStuff(string stuff) {  
 var query = new DoGenericStuffQuery { SomeData = stuff };  
 var result = await \_mediator.Send(query);  
 return Ok(result);  
 }  
}

În exemplul dat, DoGenericStuffQuery este clasa ce va avea funcția de date de intrare pentru handler, DoGenericStuffQueryHandler propriu-zis execută toată logica asupra query-ului, iar controllerul pornește handlerul prin mediator. Observați faptul că controllerul nu are acces direct la handler, mediatorul singur își dă seama ce handler să cheme în baza faptului că știe ce handlere vor accepta comanda dată, iar compilatorul va ști faptul că metoda \_mediator.Send returnează anume o valoare de tip GenericStuff, toate datorită faptului că tipurile date au fost notate în interfețele generice IRequest și IRequestHandler

De asemenea, genericurile se folosesc peste tot în cadrul programării asincrone. Comparativ cu cele sincrone, metodele async returnează un obiect de tip Task în cazul în care metoda nu returnează nimic, sau Task<T> în cazul în care metoda returnează o oarecare valoare de tip T. Fără genericuri, funcțiile date ar fi forțate să folosească castul din valoare specifică în Object și invers, împreună cu toate dezavantajele acestei metode.

public static async Task<int> GetOne() => 1;  
  
public static async Task Main()  
{  
 //metoda direct întoarce valoarea 1  
 Console.WriteLine(await GetOne());  
}

## Concluzie

## Bibliografie

* [Generic classes and methods](https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics)
* [C# specification / 8. Types / 8.4 Constructed types](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/types#84-constructed-types)
* [Tutorials Teacher / C# Generics](https://www.tutorialsteacher.com/csharp/csharp-generics)