C# alapok

Jánosi-Rancz Katalin Tünde

Sapientia EMTE
tsuto@ms.sapientia.ro

Az OOP alappillérei

- Egységbezárás - encapsulation
- **Származtatás - inheritance**
- **Polimorfizmus - polymoriphism**

Származtatás vagy Öröklődés

- **származtatás:** egy OOP aspektus, amely lehetővé teszi a kód újrafelhasználását
- minden osztály őse az **Object**, így megkapja annak műveleteit (pl.: `Equals(…)`, `GetHashCode()`, `ToString()`)
- szintaxis:

```
class Car /* : Object */ { // ősosztály ...}
class MiniVan : Car {// leszármazott ...}
```

- AB-okból ismert `is_a` kapcsolat
- egy osztálynak csak **egy** alaposztálya (`base class`) lehet, egy osztálytól örökölhet csak
- de implementálhat **akármennyi** interfészt (később)

`Osztálydiagramok készítése VS-ban: Project -> Add New Item -> Class Diagram`

Lezárt - Sealed osztályok

- `sealed` kulcsszó: meggátolja a származtatást
- a Java: `final` osztálynak felel meg
- példa

```
sealed class MiniVan : Car {...}

//Error: sealed classes cannot be extended
class DeluxeMiniVan : MiniVan {...}
```

- teljesítményi okokból kevés .NET-ben a sealed osztály, pl. `string`
- NB: `struct` implicit `sealed`

Ősosztály konstruktorának meghívása a base kulcsszóval

- a base kulcsszót használjuk ha egy származtatott osztály szeretne hozzáférni a szülőosztály nyilvános vagy védett tagjaihoz
- meg lehet határozni, hogy a base class melyik konstruktora hívódjon meg explicit módon

```
class BaseType {
  public BaseType() {...}
  public BaseType(int iParam) {...}
}
class DerivedType : BaseType {
  public DerivedType(int iParam, double dParam) : base(iParam) {
  // a leszármazott konstruktor adatokat ad át a közvetlen
       szülokonstruktornak
  ...
  }
}
```

Polimorfizmus

Polimorfizmus

- **Polimorfizmus** = hasonló módon tudja kezelni a rokon objektumokat
- ugyanaz a dolog többféleképpen
  - lehetővé teszi, hogy az ősosztály meghatározzon egy olyan tagkészletet, amely felüldefiniálható az összes leszármazott osztályban
- `virtual` = felüldefiniálható
  - a virtualis tag az ősosztály olyan tagja, amelyet **módosíthat** a leszármazott osztály
  - az ősosztály biztosít alapértelmezett megvalósítást
- `abstract`
  - az abstract metódus az ősosztály olyan tagja, amelyet **át kell írnia** a leszármazott osztálynak
  - az ősosztály **nem** biztosít alapértelmezett megvalósítást, de gondoskodik a szignatúráról
- `override`
  - egy alosztály `override` kulcsszóval módosíthatja egy virtuális metódus megvalósításának részleteit

Felüldefiniálható metódusok - Method Overriding

- Java-val ellentétben, a polimorfikus viselkedést explicit specifikálni kell:
  - `virtual` és `override` kulcsszavakkal

```
class Base {
  public virtual void Method() {
    System.Console.WriteLine("base method");
  }
}
class Derived : Base {
  public override void Method() {
    System.Console.WriteLine("derived method");
  }
}
...
Base x = new Derived();
x.Method();  //derived method
```

- felüldefiniálni csak a `virtual` és `abstract` műveleteket, tulajdonságokat lehet
- NB. a ToString() override-olható

A `base` kulcsszó -2

- `base` kulcsszó használható, az alapértelmezett, eredeti viselkedés eléréséhez

```
public class House : Asset {
  ...
  public override decimal Liability  {
    get { return base.Liability + Mortgage; }
  }
}
```

Tagok eltakarása vagy árnyékolása - Member Shadowing

- ha nem adjuk meg az `override` kulcsszót, akkor a base metódus árnyékolja a származtatott osztály metódusát

```
class Base {
  public virtual void Method() {
    Console.WriteLine(”base method”);
  }
}
class Derived : Base {
  public void Method() {                  // az override hiányzik
    Console.WriteLine(”derived method”);  // Derived.Method árnyékolódik
  }
}
...
Base x = new Derived();                   // lefut! de Warning: Base.Method()
x.Method();                               // hívódik meg
```

- ha szándékosan el akarjuk takarni az eredeti implementációt, akkor a `new` kulcsszóval tehetjük meg:

```
class Derived : Base {
  public new void Method() {
    System.Console.WriteLine(”derived method”);
  }
}
```

Lezárt virtuális metódusok - Sealed Virtual Methods

- virtualis metódusok esetében is meggátolható a származtatás

```
class Base {
  public virtual void Method() {...}
}

class Derived : Base {
  public sealed override void Method() {...}
}

class FurtherDerived : Derived {
  public override void Method() {...} // ez nem jó
}
```

Absztrakt osztályok

Absztrakt osztályok

- célja az, hogy közös (részben implementált) felületet biztosítsunk a leszármazottainak
- absztrakt osztályt nem lehet példányosítani, nem lehet `sealed`
- absztrakt metódusok törzs nélküliek
- a leszármazottaknak definiálnia kell az örökölt absztrakt metódusokat
- ha egy osztálynak van legalább egy absztrakt metódusa, az osztályt is absztraktként kell jelölni
- absztrakt osztályban lehetnek leimplementált metódusok is

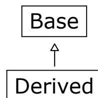
```
public abstract class Shape {
  public abstract void Draw();
}

public class Rectangle : Shape {
  int x, y, w, h;
  public Rectangle(int x, int y, int w, int h) {
    this.x = x; this.y = y;
    this.w = w; this.h = h;
  }
  public override void Draw() {
    Console.WriteLine("x:{0} y:{1} w{2} h:{3}", x, y, w, h);
  }
}
...
Shape s = new Rectangle(1, 2, 3, 4);
s.Draw();
```

Ősosztály - leszármazott osztály - típusmódosítási (kasztolási - casting) szabályok

Hasonló mint Java/C++:

- explicit típuskonverzió ((típus) a)
- **upcast** kasztoló operátor opcionális
- **downcast** kasztoló operátor kötelező
- explicit kasztolás **futásidőben** értékelődik ki
- explicit kasztolás kiválthat `InvalidCastingException`-t




```
object frank = new Manager(”...”);
Hexagon hex = (Hexagon) frank; //Whooops! InvalidCastingException
```

- `try/catch`-el elkaphatjuk az érvénytelen kasztolást
- típuskezelés `as` -el

```
Hexagon hex2 = frank as Hexagon; // as-el el tudjuk dönteni, h. kompatibilis-e
if (hex2 == null) // ha nem sikerül konvertálni -> null
  Console.WriteLine(”Sorry, frank is not a Hexagon...”);
```

- típusazonosítás `is` -el

```
if (employee is SalesPerson) ...
```

Interfészek

Interfészek használata

- **interface**: abstract tagok nevesített halmaza; csak a műveleteket határozza meg, a megvalósításukat nem
- egy szerződés, amit követni kell; meghatározzák egy osztály viselkedését; a megvalósító osztály dolga lesz majd implementálni a tagjait
- egy osztály implementálhat **tetszőlegesen sok** interfészt
- ha néhány metódus implementálatlan marad, az osztály abstract marad
- az interfész nevét konvenció szerint nagy `I` betűvel kezdjük
- amennyiben egy osztályból is származtatunk, akkor a felsorolásnál az ősosztály nevét kell előrevenni, utána jönnek az interfészek:

```
class Derived : IFace, Base { } // ez nem fordul le
```

- egy interfészt származtathatunk más interfészekből is

```
interface IDog : IAnimal {...}
```

- a .Net több 100 előre definiált interfészt nyújt (pl. `IEnumerator` a `foreach` –nek (is) biztosít felületet)

Interfészek használata -2

- az interfészben minden tag publikus, és absztrakt, ezért nem is írjuk ki a kulcsszavakat, felüldefiniáláskor sem
- az interfész nem példányosítható

```
IDog  d = new IDog(); // Fordítási hiba
```

- interfész tartalmazhat:
  - metódust, property-t, event, indexer-t
- interfész **nem** tartalmazhat:
  - mezőt, konstruktort, operátort, típust, megvalósítást
- szintaxis:

```
public interface IPointy {
  byte GetNumberOfPoints();
}
public class Pencil : IPointy {...}
public class Fork : Utensil, IPointy {...}
```

Interfészek vs Abstract osztályok

- az interfészek egy teljesen absztrakt típust definiálnak
- az absztrakt osztályok:
  - lehet leimplementált metódusa is
  - nem minden tagja kell publikus legyen
  - tartalmazhat konstruktort, property-t, fgv-t
  - egyetlen osztályból örökölhet

Egy adott osztály megvalósít–e egy interfészt

- explicit kasztolással

```
Circle c = new Circle(3,2,1);
IPointy itfPt = null;
try {
  itfPt = (IPointy) c;
} catch (InvalidCastException e) {...}
```

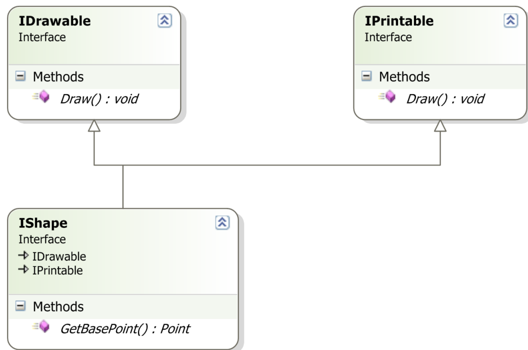
- az `as` kulcsszót használva
  - ha nem kezelhető a megadott interfészként, akkor `null` térül vissza

```
IPointy itfPt2 = c as IPointy;
```

- az `is` kulcsszót használva
  - ha nem kompatibilis, akkor a visszatérítési érték hamis

```
if (c is IPointy) ...
```

Többszörös öröklés interfésztípusokkal



- NB: interfészek örökölhetnek más interfészeket

Névütközés (Name Clashes) feloldása

- ütköző metódus nevek:

```
public interface IDrawToForm {
  void Draw();
}
```

```
public interface IDrawToMemory {
  void Draw();
}
```

- ha mindkét interfészt szeretnénk implementálni:

```
class Octagon : IDrawToForm, IDrawToMemory {
  public void Draw() {...}
}
```

- **explicit interfész megvalósítás**:

```
class Octagon : IDrawToForm, IDrawToMemory {
  public void IDrawToForm.Draw() {...}
  public void IDrawToMemory.Draw() {...}
}
...
((IDrawToMemory) myOctagon).Draw();
```

Felsorolható típusok készítése

- `IEnumerator` és `IEnumerable` a `System.Collections`-ból
- gyűjtemények használják, hogy végig iteráljanak rajta
- NB: **nincs** ADD() metódusa!!!
- enumerator: csak olvasható, csak előre menő mutató értékek sorozatán
- `IEnumerator`

```
public interface IEnumerator {
  bool MoveNext();
  object Current { get; }
  void Reset(); // elso elem elé
}
```

- `IEnumerable`

```
public interface IEnumerable {
  IEnumerator GetEnumerator(); //referenciát ad
      vissza a másik interfészre
}
```

Felsorolható típusok készítése -2

- a gyűjtemény olyan osztály, amely megvalósítja az IEnumerable interfészt (a GetEnumerator() metódussal)
- ha egy típus enumerable, `foreach` -el végig tudunk rajta haladni:

```
using System.Collections;
public class Garage : IEnumerable {
 private Car[] carArray;
 ...
 public IEnumerator GetEnumerator() {
  return carArray.GetEnumerator();
 }
}
```

```
foreach(Car c in garage)
  Console.WriteLine(c.ToString());
```

Összehasonlítható objektumok készítése `IComparable`

- a `System.IComparable` interfész lehetővé teszi egy kulcs alapján az objektumok sorba rendezését
- meghatározza más hasonló objektumokkal való kapcsolatát

```
public interface IComparable {
  int CompareTo(object o);
}
```

- a `System.Array` osztály `Sort( )` metódusa rendezni tudja az `int`, `short`, `string` stb. típusokat
  - mert ezek az adattípusok megvalósítják az `IComparable` interfészt
- mi történik ha egy `Car` típust akarok rendezni?
  - lehetőség 1, megírhatom én a `CompareTo()` metódust:

```
public class Car : IComparable {
  ...
  int IComparable.CompareTo(object obj) {
    //return a negative number if ”this < object”
    //return 0 if ”this == obj”
    //return a positive number otherwise
  }
}
...
Car[] myAutos = new Car[999];
myAutos[0] = new Car(”Mary”, 40, 1);
...
Array.Sort(myAutos);
```

Összehasonlítható objektumok készítése `IComparable` -2

- Lehetőség 2.
- modernizálhatjuk a `CompareTo()` metódust
  - tudva, hogy az `int` adattípusa megvalósítja az `IComparable` interfészt

```
int IComparable.CompareTo(object obj)
{
 Car temp= (Car)obj;
 return this.carID.CompareTo(temp.carID);
}

...
Car[] myAutos = new Car[999];
myAutos[0] = new Car(”Mary”, 40, 1);
...

Array.Sort(myAutos);
```

Többszörös rendezési sorrend megadása - `IComparer`

- mi történik ha a `Car` típust a nevek szerint is szeretnénk rendezni?
- az `IComparer` interfészt kell implementálni
- NB: **nem** IComparable

```
interface IComparer {
  int Compare(object o1, object o2);
}
```

- használat:

```
class NameComparer : IComparer
{
 int IComparer.Compare(object o1, object o2)
 {
    Car t1= Car(o1);
    Car t2= Car(o2);
    return String.Compare(t1.Name, t2.Name);
 }
}
...
Array.Sort(myAutos, new NameComparer());
```

`IComparer` mint egyedi tulajdonság

- az előbbi megoldás működik, de tovább cifrázhatom
- készíthetek egy saját statikus tulajdonságot, amely olyan objektumot ad vissza, amely megvalósítja az `IComparer` interfészt

```
public class Car : IComparer
{
    ...
    public static IComparer SortByName
    {
        get { return (IComparer) new NameComparer();}
    }
}
...
Array.Sort(myAutos, Car.SortByName);
```

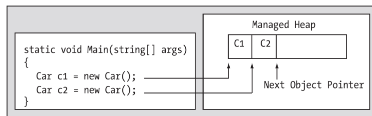
Az objektumok élettartama

Élettartam

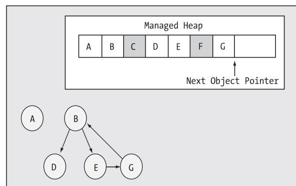
- a CLR két helyre tud adatokat pakolni:
  - verembe (**stack**)
  - halomba (**managed heap**)
- referenciatípusok minden esetben a **halomban** jönnek létre
- értéktípusok vagy a **stack**-ben vagy a **heap**-ben vannak attól függően, hogy hol deklaráltuk őket
  - metóduson belül, lokálisan deklarálva a **verembe** kerülnek
  - a referenciatípuson belül adattagként deklarálva pedig a **halomba**

Az objektumok élettartama

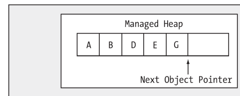
- objektumok (referenciatípusok) **managed heap**-ben jönnek létre, (**new**-vel, majd el is feledkezhetünk róla)



- automatikus szemétgyűjtő (**Garbage Collector - GC**)
  - felügyeli az elfoglalt memóriaterületet
  - memóriaszemetet eltávolítsa (pl. objektum nem érhető el a forráskód egyetlen részéről sem)
  - nem tudjuk megmondani, hogy mikor fut le



(a) before



(b) after

Garbage Collector

- a referencia null értékre állítása nem kényszeríti ki a szemétgyűjtés elindítását, heap-ben marad

```
Car myCar = new Car();
myCar = null;
```

kézzel is meghívható, de nem ajánlott, `System.GC` típus

```
Console.WriteLine(”Foglalt memória: {0}”, GC.
     GetTotalMemory(false));
for (int i = 0; i < 10; ++i)
{
  int[] x = new int[1000];
}
Console.WriteLine(”Foglalt memória: {0}”, GC.
     GetTotalMemory(false));
GC.Collect(); //  szemétgyujto kikényszerítése
Console.WriteLine(”Foglalt memória: {0}”, GC.
     GetTotalMemory(false));
```

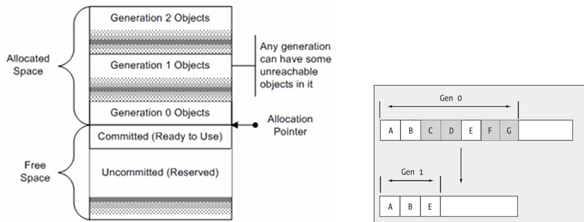
```
//Foglalt memória:
     21060

//Foglalt memória:
     70212

//Foglalt memória:
     27144
```

Objektumgenerációk

- heurisztika: a legfrissebben létrehozott objektumok lesznek leghamarabb felszabadíthatóak (pl. lokális valtozók)
- az objektumok összehasonlítódnak a generációjuk alapján
  - gen 0: újonnan foglalt objektumok, amelyet a rendszer még soha nem jelölt szemétgyűjtésre
  - gen 1: olyan objektumok, amelyek túlélték a gen0 szemétgyűjtést
  - gen 2: olyan objektumok, amelyek túléltek egynél több szemétgyűjtést



- a szemétgyűjtő mindig a 0. generációs objektumokat vizsgálja, majd 1. gen., 2. gen.
- minél hosszabb ideje létezik az objemtum a heapen, annál valószínűbb, hogy ott is marad

Eldobható (Disposable) objektumok készítése

- IDisposable interfész egy osztály által használt erőforrások felszabadítására képes, nem kell a GC –ra várni.

```
public interface IDisposable {
  void Dispose();
}
```

- Dispose() meghívható a nem felügyelt erőforrások kitakarítására

```
public class MyResourceWrapper : IDisposable {
  public void Dispose() {
  // Clean up unmanaged resources
  // Dispose other contained disposable objects
}
```

The `using` statement

- Az `IDisposable` interfészt megvalósító osztályok használhatóak ún. `using` blokkban, amely garantálja a `Dispose()` automatikus futtatását:

```
using(DbContext context = new MyDbContext()) {

    ... // Do work with context

} // itt automatikusan meghívódik a Dispose() és felszabadulnak az
     eroforrások
```

- a legtöbb I/O művelettel (fájl-, AB- és hálózatkezelés) kapcsolatos osztály megvalósítja az IDisposable –t, ezért ezeket ajánlott mindig using blokkban használni
- a using blokkon belüli referencia nem módosítható
- a `using` utasítás biztosítja, hogy a `Dispose` meghívódik akkor is ha egy hiba generálódik az objektum metódusainak meghívása közben

Kivételkezelés

C# kivételkezelés

- hasonló mint Java-ban
- 2 módon keletkezhet:
  - szándékosan mi magunk idézzük elő a `throw` utasítással
  - az alkalmazás hibás működése miatt
- .NET kivételkezelés entitásai:
  - egy osztálytípus, amely ábrázolja a kivétel részleteit
  - egy tag, amely dobja a kivételosztály egy példányát a hívónak
  - egy kódblokk meghívja a kivételre hajlamos tagot
  - egy kódblokk a hívó oldalán, amely feldolgozza (vagy elkapja) az előforduló kivételt
- kulcsszavak: `try`, `catch`, `throw`, `finally`
- Alap kivételosztály: `System.Exception`

C# kivételkezelés -2

- példa

```
try {
    // file megnyitás és olvasásis kód
} catch (System.IO.FileNotFoundException e) {
    // az ágak feldolgozása sorrendben történik, csak egy fut le
    // a file not found exception feldolgozása
} catch (System.IO.IOException e) {
    // más IO exception-ek feldolgozása
} catch {
    // paraméter  nélküli catch blokk
    // bármely további kivétel feldolgozása
} finally {
    // ez mindig végrehajtódik, ha van kivétel ha nincs
}
```

```
if(Price == -1)  {
  throw new Exception{} //szándékosan eloidézett kivétel, visszaküldi a
        hibaobjektumot a hívónak
}
```

- C# nem támogatja az ellenőrzött kivételkezelést (`throws` kulcsszó Java-ban: public void aMethod() throws IOException...)

Egyedi kivétel készítése

```csharp
public class ArticleHasMinusAmountException : Exception
{
    public Article MinusAmountArticle { get; set; }
}


public void TryCustomException()
{
    Article article = new Article { Amount = -1, Description = "Minus count article!" };

    try
    {
        int count = article.GetCount();
    }
    catch (ArticleHasMinusAmountException ex)
    {
        Console.WriteLine(ex.MinusAmountArticle.Description);
    }
    catch (Exception ex)
    {
        //Some code
    }
}
```

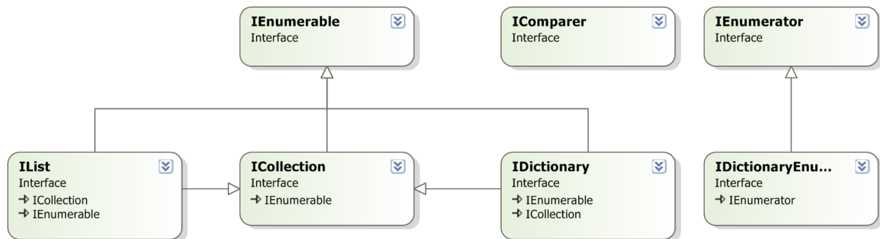
```csharp
public class Article
{
    public int Amount { get; set; }

    public string Description { get; set; }

    public int GetCount()
    {
        if (this.Amount < 0)
        {
            throw new ArticleHasMinusAmountException
                {
                    MinusAmountArticle = this
                };
        }
        else
        {
            return this.Amount;
        }
    }
}
```

Gyűjtemények és generikus típusok

A System.Collections névtér interfészei



- `ICollection`: gyűjteménytípusok által támogatott viselkedést definiál (elemek száma, System.Array típusba való másolás képességet, szálbiztosság)
- `IDictionary`: egy kulcs/értékpárt reprezentál (Add/Remove/Contains)
- `IDictionaryEnumerator`: `IDictionary`-t támogató típus tartalmának felsorolását biztosítja
- `IList`: elemek hozzáadását, eltávolítását és indexelését biztosítja az objektumok listáján

A System.Collections névtér osztálytípusai

Osztály	Használat	Megvalósított Interfészek
ArrayList	Objektumok dinamikusan méretezett tömbje	IList, ICollection, IEnumerable, IClonable
Hashtable	Numerikus kulcs által edfiniált gyűjtemények	IDictionary, ICollection, IEnumerable, IClonable
Queue	FIFO sor	ICollection, IClonable, IEnumerable
SortedList	Egy kulcs/értékpárt reprezentál, amelyek a kulcs szerint vannak rendezve és elérhetőek a kulcs és az indexen keresztül	IDictionary, ICollection, IEnumerable, IClonable
Stack	LIFO adattípus	ICollection, IClonable, IEnumerable

A bedobozolás, kidobozolás (Boxing, Unboxing) és a `System.Object` kapcsolata

- dobozolás: értéktípust $\longrightarrow$ referencia típussá alakít
- a CLR az új objektumot a `System.Object` típusban fogja eltárolni a heapen

```
short s = 25;
object objShort = s; // érték dobozolása objektum referenciába
```

- kidobozolás: referencia típust $\longrightarrow$ értéktípussá alakít
- a kidobozolást a megfelelő eredeti értéktípussá kell vissza alakítani (verem), ellenben `InvalidCastException`

```
short anotherShort = (short) objShort;
int i = (int)objShort; //InvalidCastException - a dobozban található
    érték nem int, hanem short!
```

- a boxing és az unboxing **költséges**
- használati példa:

```
ArrayList myInts = new ArrayList();
myInts.Add(10);                         //boxing: implicit
Console.WriteLine((int)myInts[0]);      //unboxing: explicit
```

Generikus gyűjtemények

Generikusok

- cél a kódújrafelhasználás volt
  - öröklődés és generikusok
- **generikus gyűjtemények**: lehetővé teszik a tárolt típus meghatározásának a késleltetését a létrehozás idejéig
- a generic-et csak egyszer kell definiálni és utána bármilyen típussal használható
- a generikus típusok **típus paramétert** deklarálnak a `<>` között
- a behelyetesíhető részt **típus argumentumnak** nevezzük
- példa: egy verem típus, amely bármely típusra alkalmazható

```
class Stack<T> {                        // T egy típus paraméter
  private T[] buffer;                   //
  ...                                   //
}                                       //
...                                     //
Stack<int> stack = new Stack<int>();   // int egy típus argumentum
stack.Push(42);
```

- `Stack<T>` az egy **open type**
- `Stack<int>` az egy **closed type**

Generikus gyűjtemények

- generikus gyűjtemények: `System.Collections.Generic` névtérben
- **nincs kidobozolási és bedobozolási költség**
- típusbiztos (egyetlen típusú objektumot tehetsz bele)

```
List<int> myInts = new List<int>(); // nincs bedobozolás
myInts.Add(10);
...
int i = myInts[0]; // nincs kidobozolás
```

- generikus interfészek: `ICollection<T>`, `IComparer<T>`, `IDictionary<TKey, TValue>`, `IEnumerable<T>`, `IEnumerator<T>`, `IList<T>`
- generikus konténerek: `Collection<T>`, `Dictionary<TKey, TValue>`, `List<T>`, `Queue<T>`, `SortedDictionary<TKey, TValue>`, `Stack<T>`, `LinkedList<T>`

Saját generikus gyűjtemény készítése

- egy generikus swap metódus, amely bármilyen adattípuson működik:

```
static void Swap<T>(ref T a, ref T b) {
  T temp = a;
  a = b;
  b = temp;
}
```

```
int x = 1, y = 2;
Swap<int>(ref x, ref y);
```

- a tagparaméterekből a fordító képes kikövetkeztetni a típusparamétert:

```
int x = 1, y = 2;
Swap(ref x, ref y); // érvényes
```

Generikus struktúrák és osztályok létrehozása

```
public struct Point<T> {
  private T xPos, yPos;
  public Point(T xVal, T yVal) { // generikus konstruktor
    xPos = xVal;
    yPos = yVal;
  }
  public T X {  // generikus tulajdonságok
    get { return xPos; }
    set { xPos = value; }
  }
  public T Y {
    get { return yPos; }
    set { yPos = value; }
  }
  public override string ToString() {
    return string.Format("[{0}, {1}]", xPos, yPos);
  }
  public void ResetPoint() {
    xPos = default(T);   //mezök visszaállítása a T
    yPos = default(T);   //alapértelmezett értékére
  }
}// a numerikus értékek default értéke 0, referenciatípus null, struktúrák
     mezoinek 0 vagy null
```

Típusparaméter korlátozása a `where` kulcsszó használatával

Szabályozhatjuk a típusparaméterek tulajdonságait

Generikus megszorítás	**Jelentése**
`where T : struct`	T értéktípus kell legyen
`where T : class`	T referenciatípus kell legyen
`where T : new()`	a `<T>`-nek rendelkeznie kell egy alapértelmezett konstruktorral.
`where T :` *`NameOfBaseClass`*	a `<T>`-nek a `NameOfBaseClass` által specifikált osztályból kell származnia.
`where T :` *`NameOfInterface`*	a <T>-nek a `NameOfInterface` által specifikált interfészt kell megvalósítania. Többszörös interfészeket megadhatunk vesszővel elválasztott listában.

Típusparaméter korlátozása, példák

```
public class MyGenericClass<T> where T : new()
{...}

// elemei támogatják az alapértelmezett konstruktort, és egyben osztályok,
     amelyek megvalósítják az IDrawable interfészt
public class MyGenericClass<T> where T : class, IDrawable, new()
{...}

public class MyGenericClass<T> : MyBase, ISomeInterface where T : struct
{...}

public class MyGenericClass<K, T> where K : new()
where T : IComparable<T>
{...}

//generikus metódus is használhatja a where-t
// értéktípusokat cserél, de osztályokat nem
static void Swap<T>(ref T a, ref T b) where T : struct
{...}
```

Generikus ősosztály létrehozása

- ha egy nem generikus osztály származtat egy generikus osztályból akkor kell specifikálja a típusparamétert:

```
public class MyList<T> {
  private List<T> listOfData = new List<T>();
}
public class MyStringList : MyList<string> {...}
```

- ha a generikus ősosztály definiál virtual vagy abstract metódusokat, akkor a származtatott típusnak felül kell definiálnia ezeket, a konkrét típusparaméterek használatával

```
public class MyList<T> {
  private List<T> listOfData = new List<T>();
  public virtual void PrintList(T data) { }
}
public class MyStringList : MyList<string> {
  public override void PrintList(string data) {...}
}
```

C# Generikus vc C++ templates

- a C# generikusai hasonlítanak a C++ sablonjaira
- C# generikusai kevésbé hatékonyak
- C# generikusainak sokkal biztonságosabb a használatuk
- két fontos különbség van a kettő közt:
  - a C++ **fordítási időben** készíti el a specializált metódusokat/osztályokat, míg a C# ezt a műveletet **futási időben** végzi el
  - a C++ fordításkor ki tudja szűrni azokat az eseteket, amelyek hibásak. A C# ezzel szemben kénytelen az ilyen problémákat megelőzni, a fordító csakis olyan műveletek elvégzését fogja engedélyezni, amelyek mindenképpen működni fognak.

Java Generics: Behind The Scenes

```
ArrayList<Person> foo = new ArrayList<Person>();
```

- the Java compiler enforces type safety just like the C# compiler
- the JVM doesn't create a specialized `ArrayListOfPerson` type, it uses the plain old `ArrayList`
- JVM does the casting for you, so that you can write

  ```
  Person p = foo.get(0);
  ```

  instead of

  ```
  Person p = (Person) foo.get(0);
  ```

- this is called **type erasure**
- pro: backward compatibility
- contra: speed hit, especially with primitive types (boxing/unboxing)

C++ Templates: Behind The Scenes

```
std::list<Person> * foo = new std::list<Person>();
```

- the C++ compiler creates a new specialized class (let's say `ListOfPersons`) which is compiled to raw binary code
- **any** code that can be written manually can be also generated with templates

```
template<class T>
T add(T a, T b) {
  return a + b;
}
```

- C++ doesn't care what the type argument `T` is, as long as `a+b` is semantically correct

C# Generics: Behind The Scenes

```
List<int> fooi = new List<int>();
List<Person> foop = new List<Person>();
```

- if the type argument `T` is a value type, the CLR creates a specialized type (e.g. `ListOfInts`) which is then JIT compiled into native code
- if `T` is a reference type, the CLR replaces the generic parameter to `Object` (e.g. `ListOfObjects`), which is then JIT compiled to native code, i.e. uses type erasure

C# Generics vs C++ Templates

C# generics do not provide the same amount of flexibility as C++ templates (C# compensates this with a strong reflection infrastructure)

- generic operations must be valid for **all** type arguments

```
T Add<T>(T arg1, T arg2) {          // C# code
  return arg1 + arg2;               // Operator '+' cannot be applied
}                                   // to operands of type 'T' and 'T'
```

- C# does not allow non-type template parameters, such as

```
template                            // C++ code
class BitSet<unsigned length> {     //
  ...                               //
};                                  //
```

- C# does not allow type parameters to have default values

```
class Container<T = char> {          // C# code: Syntax error
  ...                                //
}                                    //
...                                  //
Container c = new Container();       //
```

C# Generics vs C++ Templates (cont)

Further restrictions:

- in C# a generic type parameter cannot itself be a generic, although constructed types can be used as generics.

```
class Foo<T> {...}          // C# code
class Bar<T> {...}          //
class Bar<Foo<T>> {...}     // Syntax error
new Bar<Foo<int>>(   )      // OK
```

- C# does not support explicit / partial specialization

```
template<typename T>        // C++ code
struct Container {          // generic implementation
  ...                       // of a container
};
template<>
struct Container<int> {     // Explicit specialization
  ...                       // of the generic container, i.e.
};                          // a custom implementation for int
```