## Наследяване (Inheritance)

**Наследяването** е основен принцип от обектно-ориентираното програми­ране. То позволява на един клас да "наследява" (поведение и характе­ристики) от друг, по-общ клас. Например лъвът е от семейство котки. Всички котки имат четири лапи, хищници са, преследват жертвите си. Тази функцио­налност може да се напише веднъж в клас Котка и всички хищници да я преизползват – тигър, пума, рис и т.н.

### Как се дефинира наследяване в .NET?

Наследяването в .NET става със специална структура при декларацията на класа. В .NET и други модерни езици за програмиране един клас може да наследи само един друг клас **(single inheritance)**, за разлика от C++, където се поддържа множествено наследяване **(multiple inheritance)**. Ограничението е поро­дено от това, че при наследяване на два класа с еднакъв метод е трудно да се реши кой от тях да се използва (при C++ този проблем е решен много сложно). В .NET могат да се наследяват множество интерфейси, за които ще говорим по-късно.

Класът, който наследяваме, се нарича **клас-родител** или още **базов клас (base class, super class)**.

#### Наследяване на класове – пример

Да разгледаме един пример за наследяване на класове в .NET. Ето как изглежда базовият (родителски) клас:

|  |
| --- |
| Felidae.cs |
| /// <summary>/// Felidae is latin for "cat"/// </summary>public class Felidae{ private bool male; // This constructor calls another .ctor public Felidae() : this(true) {} // This is the .ctor that is inherited public Felidae(bool male) { this.male = male; } public bool Male { get { return male; } set { this.male = value; } }} |

Ето как изглежда и класът-наследник Lion:

|  |
| --- |
| Lion.cs |
| public class Lion : Felidae{ private int weight; // Shall be explained in the next paragraph public Lion(bool male, int weight) : base(male) { this.weight = weight; } public int Weight { get { return weight; } set { this.weight = value; } }} |

### Ключовата дума base

В горния пример в конструктора на класа Lion използваме ключовата дума base. Тя указва да бъде използван базовият клас и позволява достъп до негови методи, конструктори и член-променливи. С base() можем да извикваме конструктор на базовия клас. С base.method(…) можем да извикваме метод на базовия клас, да му подаваме параметри и да използ­ваме резултата от него. С base.field можем да вземем стойността на член-променлива на базовия клас или да й присвоим друга стойност.

В .NET наследените от базовия клас методи, които са декларирани като виртуални (virtual) могат да се **пренаписват** (**override**). Това означава да им се подмени имплементацията, като ори­ги­налният сорс код от базовия клас се игнорира, а на негово място се написва друг код. Повече за пренаписването на методи ще обясним в секцията "Виртуални методи".

Можем да извикваме непренаписан метод от базовия клас и без base. Употребата на ключовата дума е необходима само ако имаме пренаписан метод или променлива със същото име в наследения клас.

|  |  |
| --- | --- |
| idi_exc | base може да се използва изрично, за яснота. base. method(…) извиква метод, който задължително е от базовия клас. Такъв код се чете по-лесно, защото знаем къде да търсим въпросния метод.Имайте предвид, че ситуацията с this не е такава. this може да означава както метод от конкретния клас, така и метод от който и да е базов клас. |

Можете да погледнете примера в секцията нива на достъп при наследя­ване. В него ясно се вижда до кои членове (методи, конструктори и член-променливи) на базовия клас имаме достъп.

### Конструкторите при наследяване

При наследяване на един клас, нашите конструктори задължително трябва да извикат конструктор на базовия клас, за да може и той да инициализира член-променливите си. Ако не го направим изрично, в началото на всеки наш конструктор компилаторът поставя извикване на базовия конструктор без параметри: ":base()". Ето и пример:

|  |
| --- |
| public class ExtendingClass : BaseClass{ public ExtendingClass()} |

Всъщност изглежда така (намерете разликите ☺):

|  |
| --- |
| public class ExtendingClass : BaseClass{ public ExtendingClass() : base()} |

Ако базовият клас няма конструктор по подразбиране (без параметри) или този конструктор е скрит, нашите конструктори трябва да извикат изрично някои от другите конструктори на базовия клас. Липсата на изрично извикване предизвиква грешка при компилация.

|  |  |
| --- | --- |
| idi_exc | Ако един клас има само невидими конструктори (private), то това означава, че той не може да бъде наследяван.Ако един клас има само невидими конструктори (private), то това означава още много неща – например, че никой не може да създава негови инстанции освен самият той. Всъщност точно по този начин се имплементира един от най-известните шаблони описан накрая на тази глава – нарича се Singleton. |

#### Конструкторите и base – пример

Разгледайте класа Lion от последния пример, той няма конструктор по подразбиране. Да разгледаме следния клас-наследник на Lion:

|  |
| --- |
| AfricanLion.cs |
| public class AfricanLion : Lion{ // ... // If we comment the next line with ":base(male, weight)" // the class will not compile. Try it. public AfricanLion(bool male, int weight) : base(male, weight) {} public override string ToString() { return string.Format( "(AfricanLion, male: {0}, weight: {1})", this.Male, this.Weight); } // ...} |

Ако коментираме или изтрием реда ":base(male, weight);", класът AfricanLion няма да се компилира. Опитайте.

|  |  |
| --- | --- |
| idi_exc | Извикването на конструктор на базов клас става извън тялото на конструктора. Идеята е полетата на базовия клас да бъдат инициализирани преди да започ­нем да инициа­лизираме полета в класа-наследник, защото може те да разчитат на някое поле от базовия клас. |

### Модификатори на достъп на членове на класа при наследяване

Да си припомним - в главата "Дефиниране на класове" разгледахме основните модификатори на достъпа. За членовете на един клас (методи, свойства, член-променливи) бяха разгледани public, private, internal. Всъщност има още два модификатора - protected и internal protected. Ето какво означават те:

* protected дефинира членове на класа, които са невидими за ползвателите на класа (тези, които го инстанцират и използват), но са видими за класовете наследници
* protected internal дефинира членове на класа, които са едновременно internal, тоест видими за ползвателите в цялото асембли, но едновременно с това са и protected - невидими за ползвателите на класа (извън асемблито), но са видими за класовете наследници (дори и тези извън асемблито).

Когато се наследява един базов клас:

* Всички негови public и protected, protected internal членове (методи, свойства и т.н.) са видими за класа наследник.
* Всички негови private методи, свойства и член-променливи не са видими за класа наследник.
* Всички негови internal членове са видими за класа наследник само ако базовият клас и наследникът са в едно и също асембли.

Ето един пример, с който ще демонстрираме нивата на видимост при наследяване:

|  |
| --- |
| Felidae.cs |
| /// <summary>/// Latin word for "cat"/// </summary>public class Felidae{ private bool male; public Felidae() : this(true) {} public Felidae(bool male) { this.male = male; } public bool Male { get { return male; } set { this.male = value; } }} |

Ето как изглежда и класът Lion:

|  |
| --- |
| Lion.cs |
| public class Lion : Felidae{ private int weight; public Lion(bool male, int weight) : base(male) { // Compiler error – base.male is not visible in Lion base.male = male;  this.weight = weight; } // ...} |

Ако се опитаме да компилираме този пример, ще получим грешка, тъй като private променливата male от класа Felidae не е достъпна от класа Lion:



### Класът Object

Появата на обектно-ориентираното програмиране де факто става попу­лярно с езика C++. В него често се налага да се пишат класове, които трябва да работят с обекти от всякакъв тип. В C++ този проблем се решава по начин, който не се смята за много обектно-ориентиран стил (чрез използване на указатели от тип void).

Архитектите на .NET поемат в друга посока. Те създават клас, който всички други класове пряко или косвено да наследяват и до който всеки обект може да бъде преобразуван. В този клас е удобно да бъдат сложени важни методи и тяхната имплементация по подразбиране. Този клас се нарича Object.

В .NET всеки клас, който не наследява друг клас изрично, наследява системния клас System.Object по подразбиране. За това се грижи компилаторът. Всеки клас, който наследява друг клас, наследява инди­ректно Object от него. Така всеки клас явно или неявно наследява Object и има в себе си всички негови методи и полета.

Благодарение на това свойство всеки обект може да бъде преобразуван до Object. Типичен пример за ползата от неявното наследяване на Object е при колекциите, които разгледахме в главите за структури от данни. Списъчните структури (например **System.­Collections.­­ArrayList**) могат да рабо­тят с всякакви обекти, защото ги разглеждат като инстанции на класа Object.

|  |  |
| --- | --- |
| idi_exc | Специално за колекциите и работата с различни типове обекти има т.нар. Generics (обяснени подробно в главата "[Дефиниране на класове](#_Глава_14._Дефиниране_1)"). Тя позволява създаването на типизирани кла­сове – например колекция, която работи само с обекти от тип Lion. |

#### .NET, стандартните библиотеки и Object

В .NET има много предварително написани класове (вече разгледахме доста от тях в главите за колекции, текстови файлове и символни низове). Тези класове са част от .NET платформата – навсякъде, където има .NET, ги има и тях. Тези класове се наричат **обща система от типове – Common Type System (CTS)**.

.NET е една от първите платформи, която идва с такъв богат набор от предва­рително написани класове. Голяма част от тях работят с Object, за да могат да бъдат използвани на възможно най-много места.

В .NET има и доста библиотеки, които могат да се добавят допълнително и съвсем логично се наричат просто клас-библиотеки или още външни библиотеки.

#### Object, upcasting, downcasting – пример

Нека разгледаме класа Object с един пример:

|  |
| --- |
| ObjectExample.cs |
| public class ObjectExample{ public static void main() { AfricanLion africanLion = new AfricanLion(true, 80); // Implicit casting object obj = africanLion; }} |

В този пример преобразувахме един AfricanLion в Object. Тази операция се нарича **upcasting** и е позволена, защото AfricanLion е непряк наслед­ник на класа Object.

|  |  |
| --- | --- |
| idi_exc | Тук е моментът да споменем, че ключовите думи string и object са само компилаторни трикове и всъщност при компилация се заменят съответно със System.String и System.Object. |

Нека продължим примера:

|  |
| --- |
| ObjectExample.cs |
| // ...AfricanLion africanLion = new AfricanLion(true, 80);// Implicit castingobject obj = africanLion;try{ // Explicit casting AfricanLion castedLion = (AfricanLion) obj; }catch (InvalidCastException ice){ Console.WriteLine("obj cannot be downcasted to AfricanLion");} |

В този пример преобразувахме един Object в AfricanLion. Тази операция се нарича **downcasting** и е позволена само ако изрично укажем към кой тип искаме да преминем, защото Object е родител на AfricanLion и не е ясно дали променливата obj е от тип AfricanLion. Ако не е, се хвърля InvalidCastException.

#### Методът Object.ТoString()

Един от най-използваните методи, идващи от класа Object, е ToString(). Той връща текстово представяне на обекта. Всеки обект има такъв метод и следователно има текстово представяне. Този метод се използва, когато отпечатваме обект чрез Console.WriteLine(…).

#### Object.ToString() – пример

Ето един пример, в който извикваме метода ToString():

|  |
| --- |
| ToStringExample.cs |
| public class **ToStringExample**{ public static void Main() { Console.WriteLine(new object()); Console.WriteLine(new Felidae(true)); Console.WriteLine(new Lion(true, 80)); }} |

Резултатът е:

|  |
| --- |
| System.ObjectChapter\_20\_OOP.FelidaeChapter\_20\_OOP.LionPress any key to continue . . . |

Тъй като Lion не пренаписва (override) метода ToString(), в конкретния случай се извиква имплементацията от базовия клас. Felidae също не пренаписва този метод, следователно се извиква импле­ментацията, наследена от класа System.Object. В резултата, който виждаме по-горе, се съдържа именното пространство (namespace) на обекта и името на класа.

#### Пренаписване на ТoString() – пример

Нека сега ви покажем колко полезно може да е пренаписването на метода ToString(), наследено от System.Object:

|  |
| --- |
| AfricanLion.cs |
| public class AfricanLion : Lion{ // ... public override string ToString() { return string.Format( "(AfricanLion, male: {0}, weight: {1})", this.Male, this.Weight); } // ...} |

В горния код използваме String.Format(…) метода, за да форматираме резултата по подходящ начин. Ето как можем след това да извикваме пренаписания метод ToString():

|  |
| --- |
| OverrideExample.cs |
| public class **OverrideExample**{ public static void Main() { Console.WriteLine(new object()); Console.WriteLine(new Felidae(true)); Console.WriteLine(new Lion(true, 80)); Console.WriteLine(new AfricanLion(true, 80)); }} |

Резултатът е:

|  |
| --- |
| System.ObjectChapter\_20\_OOP.FelidaeChapter\_20\_OOP.Lion(AfricanLion, male: True, weight: 80)Press any key to continue . . . |

Забележете, че извикването на ToString() става скрито. Когато на метода WriteLine() подадем някакъв обект, този обект първо се преобразува до символен низ чрез метода му ToString() и след това се отпечатва в изходния поток. Така при печатане на конзолата няма нужда изрично да преобразу­ваме обектите до символен низ.

### Транзитивност при наследяването

В математиката транзитивност означава прехвърляне на взаимоотноше­ния. Нека вземем операцията "по-голямо". Ако А>В и В>С, то можем да заключим, че А>С. Това означава, че релацията "по-голямо" (>) е транзи­тивна, защото може еднозначно да бъде определено дали А е по-голямо от С или обратното.

Ако клас Lion наследява клас Felidae, а клас AfricanLion наследява клас Lion, това индиректно означава, че AfricanLion наследява Felidae. Следователно AfricanLion също има свойство Male, което е дефинирано във Felidae. Това полезно свойство позволява определена функционал­ност да бъде описана в най-подходящия за нея клас.

#### Транзитивност – пример

Ето един пример, който демонстрира транзитивността при наследяване:

|  |
| --- |
| TransitivityExample.cs |
| public class **TransitivityExample**{ public static void Main() { AfricanLion africanLion = new AfricanLion(true, 15); // Property defined in Felidae bool male = africanLion.Male; africanLion.Male = true; }} |

Заради транзитивността на наследяването можем да сме сигурни, че всички класове имат ToString() и другите методи на Object без значение кой клас наследяват.

#### Йерархия на наследяване

Ако тръгнем да описваме всички големи котки, рано или късно се стига до сравнително голяма група класове, които се наследяват един друг. Всички тези класове, заедно с базовите такива, образуват йерархия от класове на големите котки. Такива йерархии могат да се опишат най-лесно чрез клас-диаграми. Нека разгледаме какво е това "клас-диаграма".

### Клас-диаграми

**Клас-диаграмата** е един от няколко вида диаграми дефинирани в UML. **UML (Unified Modeling Language)** е нотация за визуализация на раз­лични процеси и обекти, свързани с разработката на софтуер. За UML се говори по-подробно в секцията за нотацията UML. Сега, нека ви разкажем малко за клас-диаграмите, защото те се използват, за да описват визуално йерар­хиите от класове, наследяването и вътрешността на самите класове.

В клас-диаграмите има възприети правила класовете да се рисуват като правоъгълници с име, атрибути (член-променливи) и операции (методи), а връзките между тях се обозначават с различни видове стрелки.

Накратко ще обясним два термина от UML, за по-ясно разбиране на при­мерите. Единият е **генерализация (generalization)**. Генерализация е обобщаващо понятие за наследяване на клас или имплементация на интерфейс (за интерфейси ще обясним след малко). Другият термин се нарича **асоциация (association)**. Например "Лъвът има лапи", където Лапа е друг клас.

|  |  |
| --- | --- |
| idi_exc | Генерализация и асоциация са двата най-основни начина за преизползване на код. |

#### Един клас от клас диаграма – пример

Ето как изглежда една примерна клас-диаграма на един клас:



Класът е представен като правоъгълник, разделен на 3 части, разполо­жени една под друга. В най-горната част е дефинирано името на класа. В следващата част след него са **атрибутите** (термин от UML) на класа (в .NET се наричат член-променливи и свойства). Най-отдолу са **операциите** (в UML) или методите (в .NET). Плюсът/минусът в началото указват дали атрибутът/операцията са видими (+ означава public) или невидими (- означава private). Protected членовете се означават със символа #.

#### Клас-диаграма – генерализация – пример

Ето пример за клас диаграма, показваща генерализация:



В този пример стрелките означават генерализация или наследяване.

#### Асоциации

Асоциациите представляват връзки между класовете. Те моделират взаи­моотношения. Могат да дефинират множественост (1 към 1, 1 към много, много към 1, 1 към 2, ..., и много към много).

Асоциация **много към много (many-to-many)** се означава по следния начин:



Асоциация **много към много (many-to-many) по атрибут** се означава по следния начин:



В този случай има свързващи атрибути, които показват в кои променливи се държи връзката между класовете.

Асоциация **едно към много (one-to-many)** се означава така:



Асоциация **едно към едно (one-to-one)** се означава така:



#### От диаграми към класове

От клас-диаграмите най-често се създават класове. Диаграмите улесняват и ускоряват дизайна на класовете на един софтуерен проект.

От горната диаграма можем директно да създадем класове.

Ето класа Capital:

|  |
| --- |
| Capital.cs |
| public class Capital { } |

 Ето и класа Country:

|  |
| --- |
| Country.cs |
| public class Country{ /// <summary> /// Country's capital. /// </summary> private Capital capital; // ... public Capital Capital { get { return capital; } set { this.capital = value; } } // ...} |

#### Агрегация

Агрегацията е специален вид асоциация. Тя моделира връзката "цяло / част". **Агрегат** наричаме родителския клас. **Компоненти** наричаме агре­ги­­раните класове. В единия край на агрегацията има празен ромб:



#### Композиция

Запълнен ромб означава композиция. Композицията е агрегация, при която компонентите не могат да съществуват без агрегата (родителя):

