## Абстракция (Abstraction)

Следващият основен принцип от обектно-ориентираното програмиране, който ще разгледаме, е "абстракция". **Абстракцията** означава да работим с нещо, което знаем как да използваме, но не знаем как работи вътрешно. Например имаме телевизор. Не е нужно да знаем как работи телевизорът отвътре, за да го ползваме. Нужно ни е само дистанционното и с малък брой бутони (интерфейс на дистанционното) можем да гледаме телевизия.

Същото се получава и с обектите в ООП. Ако имаме обект Лаптоп и той се нуждае от процесор, просто използваме обекта Процесор. Не знаем (или по-точно не се интересуваме) как той смята вътрешно. За да го използваме, е достатъчно да извикваме метода сметни() с подходящи параметри.

Абстракцията е нещо, което правим всеки ден. Това е действие, при което игнорираме всички детайли, които не ни интересуват от даден обект, и разглеждаме само детайлите, които имат значение за проблема, който решаваме. Например в хардуера съществува абстракция "устройство за съхранение на данни", което може да бъде твърд диск, USB memory stick, флопи диск или CD-ROM устройство. Всяко от тях работи вътрешно по различен начин, но от гледна точка на операционната система и на програмите в нея те се използват по еднакъв начин – на тях се записват файлове и директории. В Windows имаме Windows Explorer и той умее да работи по еднакъв начин с всички устройства, независимо дали са твърд диск или USB stick. Той работи с абстракцията "устройство за съхранение на данни" (storage device) и не се интересува как точно данните се четат и пишат. За това се грижат драйверите за съответните устройства. Те се явяват конкретни имплементации на интерфейса "устройство за съхране­ние на данни".

Абстракцията е една от най-важните концепции в програмирането и в ООП. Тя ни позволява да пишем **код, който работи с абстрактни струк­тури от данни** (например списък, речник, множество и други). Имайки абстрактния тип данни, ние можем да работим с него през неговия интер­фейс, без да се интересуваме от имплементацията му. Например можем да запазим във файл всички елементи на списък, без да се интересуваме дали той е реализиран с масив, чрез свързан списък или по друг начин. Този код остава непроменен, когато работим с различни конкретни типове данни. Дори можем да пишем нови типове данни (които се появяват на по-късен етап) и те да работят с нашата програма, без да я променяме.

Абстракцията ни позволява и нещо много важно – **да дефинираме интерфейс на нашите програми**, т.е. да дефинираме всички задачи, които тази програма може да извърши, както и съответните входни и изходни данни. Така можем да направим няколко по-малки програми, всяка от които да извършва някаква по-малка задача. Като прибавим това към факта, че можем да работим с абстрактни данни, ни дава голяма гъвка­вост при свързването на тези по-малки програми в една по-голяма и ни дава повече възможности за преизползване на код. Тези малки подпро­грами се наричат компоненти. Този начин на писане на програми намира широко приложение в практиката, защото ни позволява не само да преизползваме обекти, а дори цели подпрограми.

### Абстракция – пример за абстрактни данни

Ето един пример, в който дефинираме конкретен тип данни "африкански лъв", но след това го използваме по абстрактен начин – чрез абстрак­цията "лъв". Тази абстракция не се интересува от детайлите на всички видове лъвове.

|  |
| --- |
| AbstractionExample.cs |
| public class **AbstractionExample**  {  public static void Main()  {  Lion lion = new Lion(true, 150);  Felidae bigCat1 = lion;  AfricanLion africanLion = new AfricanLion(true, 80);  Felidae bigCat2 = africanLion;  }  } |

### Интерфейси

В езика C# **интерфейсът** е дефиниция на роля (на група абстрактни действия). Той дефинира какво поведение трябва да има един обект, без да указва как точно се реализира това поведение.

Един обект може да има много роли (да имплементира много интерфейси) и ползвателите му могат да го използват от различни гледни точки.

Например един обект Човек може да има ролите Военен (с поведение "стреляй по противника"), Съпруг (с поведение "обичай жена си"), Данъкоплатец (с поведение "плати си данъка"). Всеки човек обаче импле­ментира това поведение по различен начин: Иван си плаща данъците навреме, Георги – не навреме, Петър – въобще не ги плаща.

Някой може да попита защо най-базовият за всички обекти клас Object не е всъщност интерфейс. Причината е, че тогава всеки клас щеше да трябва да имплементира една малка, но много важна, група методи, а това би отнемало излишно време. Оказва се, че и не всеки клас има нужда от специфична реализация на Object.GetHashCode(), Object.Equals(…), Object.ToString(), тоест имплемен­тацията по подразбиране върши работа в повечето случаи. От класа Object не е нужно да се пренапише (повторно имплементира) никой метод, но ако се наложи, това може да се направи. Пренаписването на методи е обяснено в детайли в секцията за виртуални методи.

#### Интерфейси – ключови понятия

В интерфейса може да има само декларации на методи и константи.

**Сигнатура на метод (method signature)** е съвкупността от името на метода + описание на параметрите (тип и последователност). В един клас/интерфейс всички методи трябва да са с различни сигнатури и да не съвпадат със сигнатури на наследени методи.

**Декларация на метод (method declaration)** е съвкупността от връщания тип на метода + сигнатурата на метода. Връщаният тип е просто за яснота какво ще върне метода.

|  |  |
| --- | --- |
| idi_exc | Това, което идентифицира един метод, е неговата сигна­тура. Връщаният тип не е част нея. Причината е, че ако два метода се различават само по връщания тип (напри­мер два класа, които се наследяват един друг), то не може еднозначно да се идентифицира кой метод трябва да се извика. |

**Имплементация на клас/метод (class/method implementation)** е тялото със сорс код на класа/метода. Най често е заключено между скобите { и }. При методите се нарича още **тяло на метод**.

#### Интерфейси – пример

Интерфейсът в .NET се дефинира с ключовата думичка interface. В него може да има само декларации на методи, както и статични променливи (за константи например). Ето един пример за интерфейс:

|  |
| --- |
| Reproducible.cs |
| public interface Reproducible<T> where T:Felidae  {  T[] Reproduce(T mate);  } |

За шаблонни типове (Generics) сме говорили в главата "Дефиниране на класове". Интерфейсът, който сме написали, има един метод от тип Т (Т трябва да наследява Felidae) и връща масив от Т.

Ето как изглежда и класът Lion, който имплементира интерфейса Reproducible:

|  |
| --- |
| Lion.cs |
| public class Lion : Felidae, Reproducible<Lion>  {  // ...  Lion[] Reproducible<Lion>.Reproduce(Lion mate)  {  return new Lion[]{new Lion(true, 12), new Lion(false, 10)};  }  } |

Името на интерфейса се записва в декларацията на класа (първия ред) и се специфицира шаблонният клас.

Можем да укажем метод на кой интерфейс имплементираме, като му напишем името:

|  |
| --- |
| Lion[] Reproducible<Lion>.Reproduce(Lion mate) |

В интерфейса методите само се декларират, имплементацията е в класа, който имплементира интерфейса – Lion.

Класът, който имплементира даден интерфейс, трябва да имплементира всеки метод от него. Изключение – ако класът е абстрактен, тогава може да имплементира нула, няколко или всички методи. Всички останали методи се имплементират в някой от класовете наследници.

#### Абстракция и интерфейси

Най-добрият начин да се реализира абстракция е да се работи с интер­фейси. Един компонент работи с интерфейси, които друг имплементира. Така подмяната на втория компонент няма да се отрази на първия, стига новият компонент да имплементира старите интерфейси. Интерфейсът се нарича още **договор** (**contract**). Всеки компонент, имплементирайки един интерфейс, спазва определен договор (сигнатурата на методите). Така два компонента, стига да спазват правилата на договора, могат да общуват един с друг, без да знаят как работи другата страна.

Примери за важни интерфейси от Common Type System (CTS) са System.Collections.Generic.IList<T> и System.­Collections.­Generic. ICollection<T>. Всички стандартни колекции импле­мен­тират тези интерфейси и различните компоненти си прехвърлят различни имплементации (масиви или свързани списъци, хеш-таблици, червено-черни дървета и др.) винаги под общ интерфейс.

Колекциите са един отличен пример на обектно-ориентирана библиотека с класове и интерфейси, при която се използват много активно всички основни принципи на ООП: абстракция, наследяване, капсулация и поли­морфизъм.

#### Кога да използваме абстракция и интерфейси?

Отговорът на този въпрос е: винаги, когато искаме да постигнем абстрак­ция на данни или действия, чиято имплементация по-късно може да се подмени. Код, който комуникира с друг код чрез интерфейси е много по-издръжлив срещу промени, отколкото код, напи­сан срещу конкретни кла­сове. Работата през интерфейси е често сре­щана и силно препоръчвана практика – едно от основните правила за писане на качествен код.

#### Кога да пишем интерфейси?

Винаги е добра идея да се използват интерфейси, когато се предоставя функционалност на друг компонент. В интерфейса се слага само функцио­налността (като декларация), която другите трябва да виждат.

Вътрешно в една програма/компонент интерфейсите могат да се използват за дефиниране на роли. Така един обект може да се използва от много класове чрез различните му роли.

## Шаблонни типове и типизиране (generics)

В тази секция ще обясним концепцията за типизиране на класове. Преди да започнем, обаче, нека разгледаме един пример, който ще ни помогне за разберем по-лесно идеята.

### Приют за бездомни животни – пример

Нека имаме два класа. Нека класът Dog описва куче:

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

И нека класът Cat описва котка:

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

След това искаме да си създадем клас, който описва приют за бездомни животни – AnimalShelter. Този клас има определен брой свободни клетки, който определя броя на животни, които могат да намерят подслон в приюта. Особеното на класа, който искаме да създадем е, че той трябва да подслонява само животни от един и същ вид, в нашия случай или само кучета, или само котки, защото съвместното съжителство на различни видове животни не винаги е добра идея.

Ако се замислим как ще решим задачата със знанията, които имаме до момента, стигаме до извода, че за да гарантираме, че нашият клас ще съдържа елементи само от един тип, трябва да използваме масив от еднакви обекти. Тези обекти може да са кучета, котки или просто инстан­ции на универсалния тип object.

Например, ако искаме да направим приют за кучета, ето как би изглеждал нашият клас:

|  |
| --- |
| AnimalsShelter.cs |
| public class AnimalShelter  {  private const int DefaultPlacesCount = 20;  private Dog[] animalList;  private int usedPlaces;  public AnimalShelter() : this(DefaultPlacesCount)  {  }  public AnimalShelter(int placesCount)  {  this.animalList = new Dog[placesCount];  this.usedPlaces = 0;  }  public void Shelter(Dog newAnimal)  {  if (this.usedPlaces >= this.animalList.Length)  {  throw new InvalidOperationException("Shelter is full.");  }  this.animalList[this.usedPlaces] = newAnimal;  this.usedPlaces++;  }  public Dog Release(int index)  {  if (index < 0 || index >= this.usedPlaces)  {  throw new ArgumentOutOfRangeException(  "Invalid cell index: " + index);  }  Dog releasedAnimal = this.animalList[index];  for (int i = index; i < this.usedPlaces - 1; i++)  {  this.animalList[i] = this.animalList[i + 1];  }  this.animalList[this.usedPlaces - 1] = null;  this.usedPlaces--;  return releasedAnimal;  }  } |

Капацитетът на приюта (броят животни, които могат да се приютят в него) се задава при създаване на обекта. По подразбиране е стойността на константата DefaultPlacesCount. Полето usedPlaces използва­ме за следене на заетите до момента клетки (едновременно с това го из­ползваме за индекс в масива, да "сочим" към първото свободно място отляво на дясно в масива).

dog-shelter-two-dogs

Създали сме метод за добавяне на ново куче в приюта – Shelter(…) и съот­вет­но за освобождаване от приюта – Release(int).

Методът Shelter() добавя всяко ново животно в първата свободна клетка в дясната част на масива (ако има такава).

Методът Release(int) приема номера на клетката, от която ще бъде освободено куче (т.е. номера на индекса в масива, където е съхранена връзка към обекта от тип Dog).

dog-shelter-release-dog-1

След това премества животните, които се намират в клетки с по-голям номер от номера на клетката, от която ще извадим едно куче, с една позиция на наляво (стъпки 2 и 3 на схемата по-долу).

dog-shelter-release-dog-2

Освободената клетка на позиция usedPlaces-1 се маркира като свободна, като й се присвоява стойност null. Това осигурява освобождаването на референцията към нея и съответно позволява на системата за почистване на паметта (garbage collector) да освободи обекта, ако той не се ползва никъде другаде в програмата в същия момент. Това предпазва недиректна загуба на памет (memory leak).

Накрая присвоява на полето usedPlaces номера на последната свобод­на клетка (стъпки 4 и 5 от схемата отгоре).

dog-shelter-release-dog-3

Забелязва се, че "изваждането" на животно от дадена клетка би могло да е бавна операция, тъй като изисква прехвърляне на животните от следващите клетки с една позиция наляво. В главата "Линейни структури от данни" ще разгледаме и по-ефективни начини за предста­вяне на приюта за животни, но за момента нека се фокусираме върху темата за шаблонните типове.

До този момент успяхме да имплементираме функционалността на приюта - класът AnimalShelter. Когато работим с обекти от тип Dog, всичко се компилира и изпълнява безпроблемно:

|  |
| --- |
| public static void Main()  {  AnimalShelter dogsShelter = new AnimalShelter(10);  Dog dog1 = new Dog();  Dog dog2 = new Dog();  Dog dog3 = new Dog();  dogsShelter.Shelter(dog1);  dogsShelter.Shelter(dog2);  dogsShelter.Shelter(dog3);  dogsShelter.Release(1); // Releasing dog2  } |

Какво ще стане, обаче, ако се опитаме да използваме класа AnimalShelter за обекти от тип Cat:

|  |
| --- |
| public static void Main()  {  AnimalShelter dogsShelter = new AnimalShelter(10);    Cat cat1 = new Cat();  dogsShelter.Shelter(cat1);  } |

Както се очаква, компилаторът извежда грешка:

|  |
| --- |
| The best overloaded method match for 'AnimalShelter.Shelter( Dog)' has some invalid arguments. Argument 1: cannot convert from 'Cat' to 'Dog' |

Следователно, ако искаме да направим приют за котки, няма да успеем да преизползваме вече създадения от нас клас, въпреки, че операциите по добавяне и изваждане на животни от приюта ще бъдат идентични. Следователно, буквално ще трябва да копираме класа AnimalShelter и да променим само типа на обектите, с които се работи – Cat.

Да, но ако решим да правим приют и за други видове животни? Колко класа за приюти за конкретния тип животни ще трябва да създадем?

Виждаме, че това решение на задачата не е достатъчно изчерпателно и не изпълнява изцяло условията, които си бяхме поставили, а именно – да съществува **един единствен клас**, който описва нашия приют за каквито и да е животни (т.е. за всякакви обекти) и при работа с него той да **съдържа само един вид животни** (т.е. единствено обекти от един и същ тип).

Бихме могли да използваме вместо типа Dog универсалния тип object, който може да приема като стойности Dog, Cat и всякакви други типове данни, но това ще създаде някои неудобства, свързани с нуждата от обратно преобразуване от object към Dog, когато се прави приют за кучета, а той съдържа клетки от тип object вместо от тип Dog.

За да решим задачата ефективно се налага да използваме една функцио­налност на езика С#, която ни позволява да удовлетворим всички условия едновременно. Тя се нарича **шаблонни класове (generics)**.

### Какво представляват шаблонните класове?

Както знаем, когато за работата на един метод е нужна допълнителна информация, тази информация се подава на метода чрез параметри. По време на изпълнение на програмата, при извикване на метода, подаваме аргументи на метода, те се присвояват на параметри­те му и след това се използват в тялото на метода.

По подобие на методите, когато знаем, че функционалността (действията) капсулирана в един клас, може да бъде приложена не само към обекти от един, а от много (разнородни) типове, и тези типове не са известни по време на деклариране на класа, можем да използваме една функционал­ност на езика С# наречена **шаблонни типове (generics)**. Тя ни позволява да декларираме параметри на самия клас, чрез които обознача­ваме неиз­вестния тип, с който класът ще работи в последствие. След това, когато инстанцираме нашия типизиран клас, ние заместваме неизвестния тип с конкретен. Съответно новосъздаденият обект ще работи само с обекти от конкретния тип, който сме задали при инициализацията му. Конкретният тип може да бъде всеки един клас, който компилаторът разпознава, включително структура, изброен тип или друг шаблонен клас.

За да добием по-ясна представа за същността на шаблонните типове, нека се върнем към нашата задача от предходната секция. Както се досещаме, кла­сът, който описва приют на животни (AnimalShelter), може да опе­рира с различни типове животни. Следователно, ако искаме да създа­дем генерално решение на задачата, по време на декларация на класа AnimalShelter, ние не можем да знаем какъв тип животни ще бъдат приютявани в приюта. Това е достатъчна индикация, че можем да типизираме нашия клас, добавяйки към декларацията на класа, като параметър, неизвестния ни тип на животни.

В последствие, когато искаме да създадем приют за кучета например, на този параметър на класа ще подадем името на нашия тип – класа Dog. Съответно, ако създаваме приют за котки, ще подадем типа Cat и т.н.

|  |  |
| --- | --- |
| idi_exc | Типизирането на клас (създаването на шаблонен клас) представлява добавяне, към декларацията на един клас, на параме­тър (за­ме­стител) на неизвестен тип, с който класът ще работи по време на изпълне­ние на програмата. В послед­ствие, когато класът бива инстанциран, този параметър се замества с името на някой кон­кретен тип. |

В следващите секции ще се запознаем със синтаксиса на типизирането на класове и ще представим нашия пример преработен, така че да използва типизиране.

### Декларация на типизиран (шаблонен) клас

Формално, типизирането на класове се прави, като към декларацията на класа, след самото име на класа се добави <T>, където T е заместителят (параметърът) на типа, който ще се използва в последствие:

|  |
| --- |
| **[<modifiers>] class <class\_name><T>**  **{**  **}** |

Трябва да отбележим, че знаците '<' и '>', които ограждат заместителя T са задължителна част от синтаксиса на езика С# и трябва да участват в декларацията на типизирането на даден клас.

Декларацията на типизирания клас, описващ приюта за бездомни живот­ни, би изглеждала по следния начин:

|  |
| --- |
| class AnimalShelter<T>  {  // Class body here ...  } |

По този начин, можем да си представим, че правим шаблон на нашия клас AnimalShelter, който в последствие ще конкретизираме, заменяйки T с конкретен тип, например Dog.

Eдин клас може да има и повече от един замести­тел (да е параметризиран по повече от един тип), в зависимост от нуждите му:

|  |
| --- |
| **[<modifiers>] class <class\_name><T1 [, T2, [... [, Tn]>**  **{**  **}** |

Ако класът се нуждае от няколко различни неизвестни типа, тези типове трябва да се изброят, чрез запетайка между знаците '<' и '>' в деклараци­ята на класа, като всеки един от използваните заместители трябва да е различен идентификатор (например различна буква) – в дефиницията са указани като T1, T2, ..., Тn.

В случай, че искахме да създадем приют за животни от смесен тип, такъв че да приютява кучета и котки едновременно, можехме да декларираме нашия клас по следния начин:

|  |
| --- |
| class AnimalShelter<T, U>  {  // Class body here ...  } |

Ако това беше нашия случай, щяхме да използваме първия параметър T, за означаване на обектите от тип Dog, с които нашия клас щеше да оперира и U – за означаване на обектите от тип Cat.

### Конкретизиране на типизирани класове

Преди да представим повече подробности за типизацията, нека поглед­нем как се използват типизираните класове. Използването на типизирани класове става по следния начин:

|  |
| --- |
| **<class\_name><concrete\_type> <variable\_name> =**  **new <class\_name><concrete\_type>();** |

Отново, подобно на заместителя T в декларацията на нашия клас, знаците '<' и '>', които ограждат кон­кретния клас concrete\_type, са задължителни.

Ако искаме да създадем два приюта, един за кучета и един за котки, ще трябва да използваме следния код:

|  |
| --- |
| AnimalShelter<Dog> dogsShelter = new AnimalShelter<Dog>();  AnimalShelter<Cat> catsShelter = new AnimalShelter<Cat>(); |

По този начин сме сигурни, че приютът dogsShelter винаги ще съдържа обекти от тип Dog, а променливата catsShelter ще оперира винаги с обекти от тип Cat.

### Използване на неизвестните типове в декларация на полета

Веднъж използвани по време на декларацията на класа, параметрите, които са използвани за указване на неизвестните типове са видими в цялото тяло на класа, следователно могат да се използват за деклариране на полета както всеки друг тип:

|  |
| --- |
| **[<modifiers>] T <field\_name>;** |

Както можем да се досетим, в нашия пример с приюта за бездомни животни, можем да използваме тази възможност на езика С#, за да декларираме типа на полето animalsList, в което съхраняваме референ­ции към обектите на приютените животни, вместо с конкретния тип Dog, с параметъра Т:

|  |
| --- |
| private T[] animalList; |

За сега нека приемем, че когато създаваме обект от нашия клас, подавайки конкретен тип (например Dog), по време на изпълнение на програмата неизвестният тип Т ще бъде заменен с въпросния тип. Ако сме избрали да създадем приют за кучета, можем да смятаме, че нашето поле е декла­рирано по следния начин:

|  |
| --- |
| private Dog[] animalList; |

Съответно, когато искаме да инициализираме въпросното поле в кон­структора на нашия клас, ще трябва да го направим по същия начин, както обикновено – създаваме масив, само че използвайки заместителя на неизвестния тип – Т:

|  |
| --- |
| public AnimalShelter(int placesNumber)  {  animalList = new T[placesNumber]; // Initialization  usedPlaces = 0;  } |

### Използване на неизвестните типове в декларация на методи

Тъй като един неизвестен тип, използван в декларацията на един типизи­ран клас е видим от отварящата до затварящата скоба на тялото на класа, освен за декларация на полета, той може да бъде използван и в деклара­цията на методи, а именно:

* Като параметър в списъка от параметри на метода:

|  |
| --- |
| **<return\_type> MethodWithParamsOfT(T param)** |

* Като резултат от изпълнението на метода:

|  |
| --- |
| **Т MethodWithReturnTypeOfT(<params>)** |

Както вече се досещаме, използвайки нашия пример, можем да адаптира­ме методите Shelter и Release, съответно:

* Като метод с параметър от неизвестен тип Т:

|  |
| --- |
| public void Shelter(T newAnimal)  {  // Method's body goes here ...  } |

* И метод, който връща резултат от неизвестен тип Т:

|  |
| --- |
| public T Release(int i)  {  // Method's body goes here ...  } |

Както вече знаем, когато създадем обект от нашия клас приют и неиз­вестния тип го заменим с някой конкретен тип (например Cat), по време на изпълнение на програмата горните методи ще имат следния вид:

* Параметърът на метода Shelter ще бъде от тип Cat:

|  |
| --- |
| public void Shelter(Cat newAnimal)  {  // Method's body goes here ...  } |

* Методът Release ще връща резултат от тип Cat:

|  |
| --- |
| public Cat Release(int i)  {  // Method's body goes here ...  } |

### Типизирането (generics) зад кулисите

Преди да продължим, нека обясним какво става в паметта на компютъра, когато работим с типизирани класове.

under-the-hat-of-generics

Първо декларираме нашия типизиран клас MyClass<T> (generic class description в горната схема). След това компилаторът транслира нашия код на междинен език (MSIL), като транслираният код, съдържа инфор­мация, че класът е типизиран, т.е. работи с неопределени до момента типове. По време на изпълнение, когато някой се опитва да работи с нашия типизиран клас и да го използва с конкретен тип, се създава ново **описание на клас** (concrete type class description в схемата по-горе), което е идентично с това на типизирания клас, с тази разлика, че навсякъде където е използвано T, сега се заменя с конкретния тип. Например ако се опитаме да използваме MyClass<int>, навсякъде, където в нашия код e използван неизвестния параметър T, ще бъде заме­нен с int. Едва след това, можем да създадем обект от типизирания клас с конкретен тип int. Особеното тук е, че за да се създаде този обект, ще се използва описанието на класа, което бе създадено междувременно (concrete type class description). Инстанцирането на шаблонен клас по дадени конкретни типове на неговите параметри се нарича "**специали­зация на тип**" или "**разгъване на шаблонен клас**".

Използвайки нашия пример, ако създадем обект от тип AnimalShelter<T>, който работи само с обекти от тип Dog, ако се опитаме да добавим обект от тип Cat, това ще доведе до грешка при компилация почти идентична с грешките, които бяха изведени при опит за добавяне на обект от тип Cat, към обект от тип AnimalShelter, който създадохме в първата подсекция "Приют за бездомни животни – пример":

|  |
| --- |
| public static void Main()  {  AnimalShelter<Dog> dogsShelter = new AnimalShelter<Dog>(10);  Cat cat1 = new Cat();  dogsShelter.Shelter(cat1);  } |

Както се очакваше, получаваме следните съобщения:

|  |
| --- |
| The best overloaded method match for 'AnimalShelter< Dog>.Shelter(Dog)' has some invalid arguments  Argument 1: cannot convert from 'Cat' to 'Dog' |