МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ЧЕРЕПОВЕЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт информационных технологий

Кафедра математики и информатики

УЧЕБНО-МЕТОДИЧЕСКИЕ МАТЕРИАЛЫ ПО ДИСЦИПЛИНЕ

«СИСТЕМЫ ПРОГРАММИРОВАНИЯ»

Направление подготовки (специальность):
01.03.02 Прикладная математика и информатика

Образовательная программа:
ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

Очная форма обучения

Составители:

Лавров В.В., старший
преподаватель кафедры МиИ

г. Череповец - 2022

Перечень основной и дополнительной учебной литературы, необходимой для освоения дисциплины (модуля)

Основная литература:

1. Ржевский, С. В. Математическое программирование : учебное пособие / С. В. Ржевский. — Санкт-Петербург : Лань, 2022. — 608 с. — ISBN 978-5-8114-3853-2. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/206993
2. Кривцов, А. Н. Алгоритмизация и программирование. Основы программирования на C/C++ : учебное пособие / А. Н. Кривцов, С. В. Хорошенко. — Санкт-Петербург : СПбГУТ им. М.А. Бонч-Бруевича, 2020. — 202 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/180057

Дополнительная литература:

1. Юрьева, А. А. Математическое программирование : учебное пособие / А. А. Юрьева. — 2-е изд., испр. и доп. — Санкт-Петербург : Лань, 2022. — 432 с. — ISBN 978-5-8114-1585-4. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/212210
2. Рацеев, С. М. Программирование на языке Си : учебное пособие для вузов / С. М. Рацеев. — Санкт-Петербург : Лань, 2022. — 332 с. — ISBN 978-5-8114-8585-7. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/193320
3. Гунько, А. В. Программирование : учебно-методическое пособие / А. В. Гунько. — Новосибирск : НГТУ, 2019. — 74 с. — ISBN 978-5-7782-3961-6. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/152231
4. Касторнов, А.Ф. Программирование на языке Паскаль : учебное пособие для вузов / Касторнов А.Ф., Касторнова В.А., Козлов О.А. - Нижний Новгород : НГПУ им. К.Минина, 2012. - 152 с.
5. Павловская, Т.А. C#: программирование на языке высокого уровня : учебное пособие для вузов. - СПб. : Питер, 2013. - 432с.
1. Немнюгин, С.А. TurboPascal: Программирование на языке высокого уровня : учебник для вузов / Немнюгин С.А. - 2-е изд. - СПб. : Питер, 2007. - 543 с.

Перечень ресурсов информационно-телекоммуникационной сети «Интернет», необходимых для освоения дисциплины (модуля), включая перечень информационных справочных систем (при необходимости)

1 Интерактивная доска.
2 http://www.ois.org.ua/spravka/mat/index.htm - электронная библиотека по математике.
3 http://eqworld.ipmnet.ru/ru/library.htm- учебно-образовательная физико-математическая библиотека.
4 http://www.exponenta.ru/- образовательный математический сайт.

Учебно-методические указания и рекомендации к изучению тем лекционных и практических занятий, самостоятельной работе студентов

Лекции

№ п/п	Тема лекции	Количество часов
1	Объектная модель: основные элементы объектной модели, отношения между объектами и классами.	2
2	Принципы объектно-ориентированного программирования.	2
3	Особенности платформы .NET. Отличия C# от C.	4
4	Поля класса. Методы класса.	4
5	Свойства и индексаторы. Функциональные типы в C#.	4
6	Наследование.	2
7	Виртуальные функции и абстрактные классы.	4
8	Создание иерархии исключений.	2
9	Обобщенные интерфейсы. Обобщенные методы.	4
10	Обобщенные делегаты.	2
	Итого	30

Лабораторные работы

№ п/п	Тема лабораторной работы	Количество часов
1-2	Объектно-ориентированное программирование.	6
3-6	Классы в C#.	12
7-9	Наследование и полиморфизм.	12
10-12	Обобщения.	10
	Итого	40

Лекция 1

1.1 Сложность разработки программного обеспечения

Мы окружены сложными системами:

- персональный компьютер;
- любое дерево, цветок, животное;
- любая материя – от атома до звезд и галактик;
- общественные институты – корпорации и сообщества.

Большинство сложных систем обладает иерархической структурой. Но не все ПО – сложное. Существует класс приложений которые проектиру- ются разрабатываются и используются одним и тем же человеком. Но они имеют ограниченную область применения. Вопросы сложности появляются при разработке корпоративного ПО, промышленного программирования.

Сложность ПО вызывается четырьмя основными причинами:

- сложностью реальной предметной области, из которой исходит заказ на разработку;
- трудностью управления проектированием;
- необходимостью обеспечить достаточную гибкость программы;
- сложность описания поведения больших дискретных систем.

1.2 Декомпозиция

Декомпозиция один из способов борьбы со сложностью.

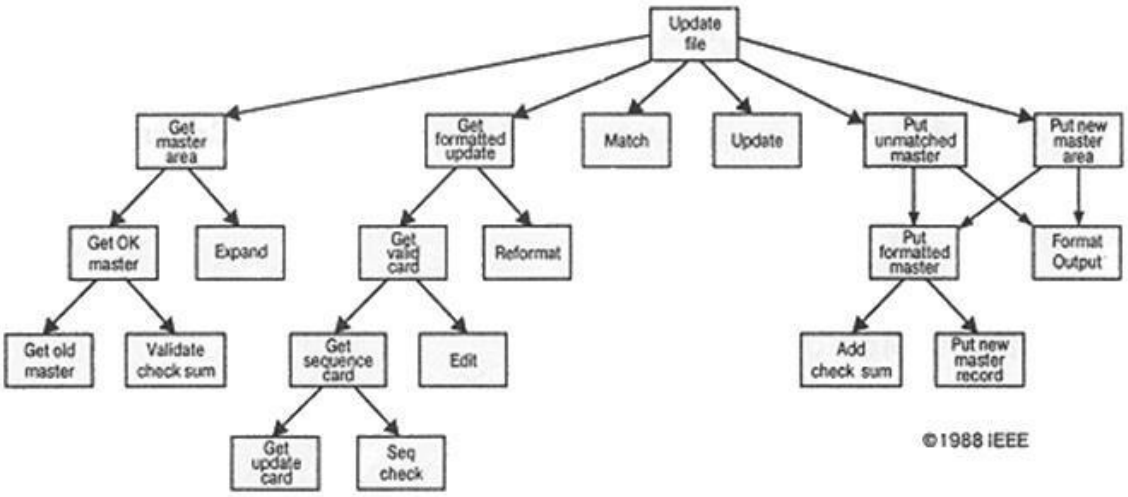


Рис. 1. Пример алгоритмической декомпозиции

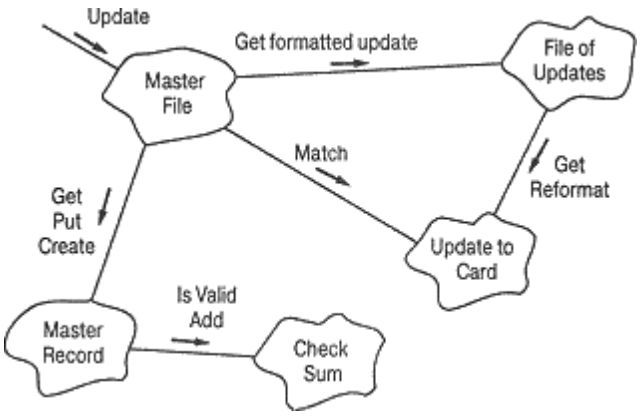


Рис. 2. Пример объектно-ориентированной декомпозиции

Необходимо разделять систему на независимые подсистемы, каждую из

которых разрабатывать отдельно. Выделяют следующие методы декомпо- зиции:

- алгоритмическая декомпозиция (см. рис. 1);
- объектно-ориентированная декомпозиция (см. рис. 2).

1.3 Краткая история языков программирования

Выделяют следующие этапы развития языков программирования высо- кого уровня:

- Языки первого поколения (1954□1958)
  - FORTRAN 1 Математические формулы
  - ALGOL-58 Математические формулы
- Языки второго поколения (1959□1961)
  - FORTRAN II Подпрограммы
  - ALGOL-60 Блочная структура, типы данных
  - COBOL Описание данных, работа с файлами
  - LISP Обработка списков, указатели, сборка мусора
- Языки третьего поколения (1962□1970)
  - PL/I FORTRAN+ALGOL+COBOL
  - Pascal Простой наследник ALGOL-60
  - Simula Классы, абстракция данных
- Разрыв преемственности (1970□1980)
  - C Эффективный высокоуровневый язык
  - FORTRAN 77 Блочная структура, типы данных
    □ Бум ООП (1980□1990)
  - Smalltalk 80 Чисто объектно-ориентированный язык
  - C++ C + Simula
  - Ada83 Строгая типизация; сильное влияние Pascal
- Появление инфраструктур (1990□…)
  - Java Блочная структура, типы данных
  - Python Объектно-ориентированный язык сценариев
  - Visual C# Конкурент языка Java для среды Microsoft .NET

1-е поколение (рис. 3) преимущественно использовалось для научных и технических вычислений, математический словарь. Языки освобождали от сложностей ассемблера, что позволяло отступать от технических деталей реализации компьютеров.

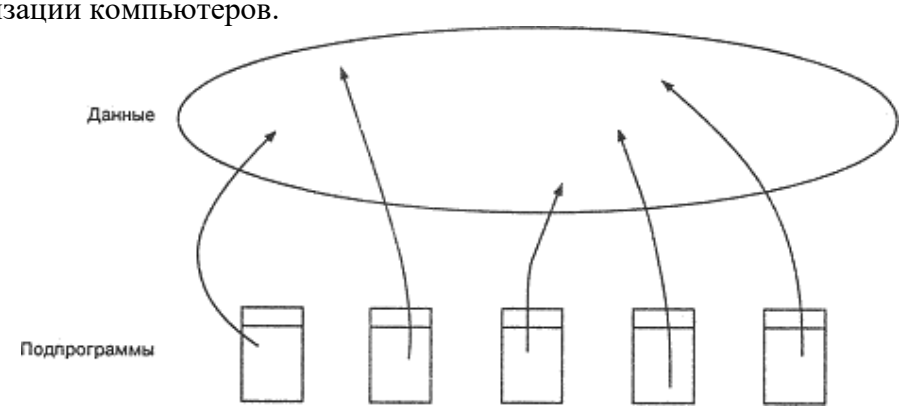


Рис. 3. Топология языков первого поколения

Программы, написанные на языках программирования первого поколе- ния, имеют относительно простую структуру, состоящую только из глобаль- ных

данных и подпрограмм.

Языки *2-го поколения* (рис. 4) сделали акцент на алгоритмических абстракциях, что приблизило разработчиков к предметной области. Появилась процедурная абстракция, которая позволила описать абстрактные про- граммные функции в виде подпрограмм.

На *3-е поколение* языков программирования (рис. 5) влияние оказало то, что стоимость аппаратного обеспечения резко упала, при этом, производи- тельность экспоненциально росла. Языки поддерживали абстракцию дан- ных и появилась возможность описывать свои собственные типы данных

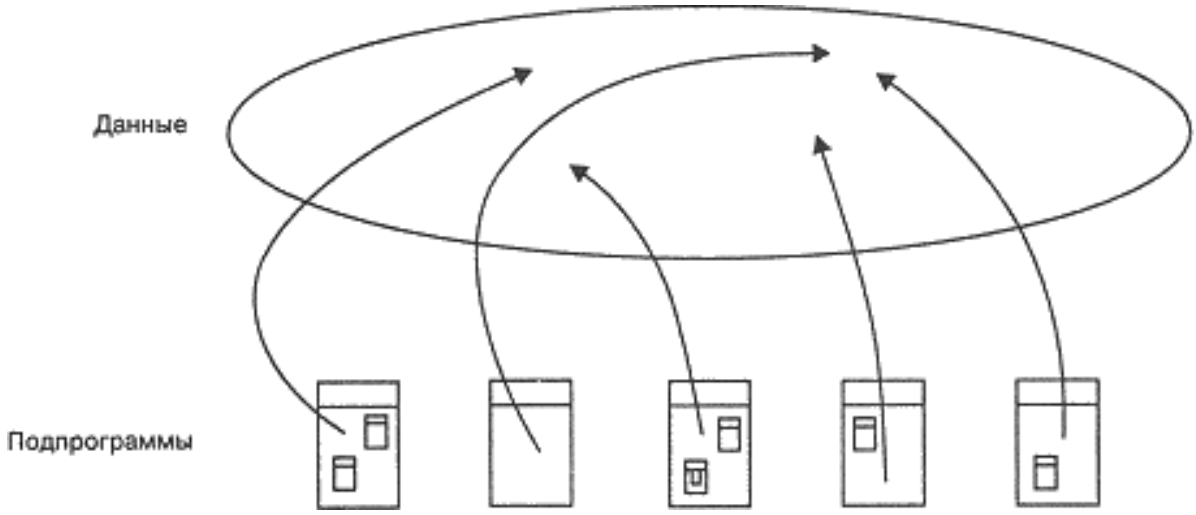


Рис. 4. Топология языков второго поколения

(структуры). В 1970-е годы было создано несколько тысяч языков программирования для решения конкретных задач, но практически все из них ис- чезли. Осталось только несколько известных сейчас языков, которые про- шли проверку временем.

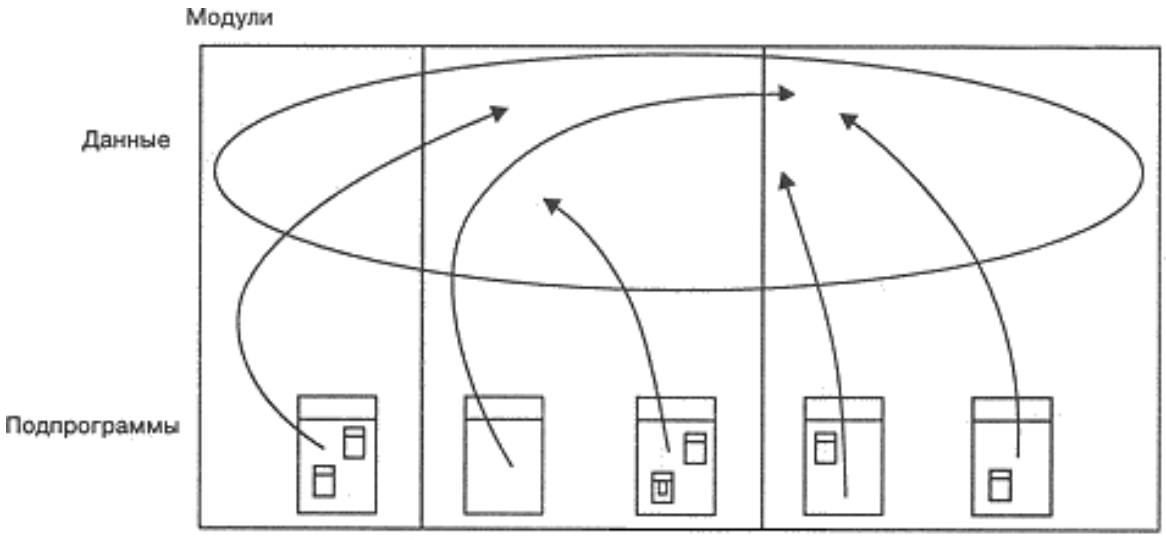


Рис. 5. Топология языков третьего поколения

В модули собирали подпрограммы, которые будут изменяться сов- местно, но их не рассматривали как новую технику абстракции.

В 1980-е произошел бум развития объектно-ориентированного программирования (рис. 6). Языки данного времени лучше всего поддерживают объ- ектно-ориентированную декомпозицию ПО. В 90-х появились инфраструк- туры (J2EE, .NET), предоставляющие огромные объемы интегрированных сервисов.

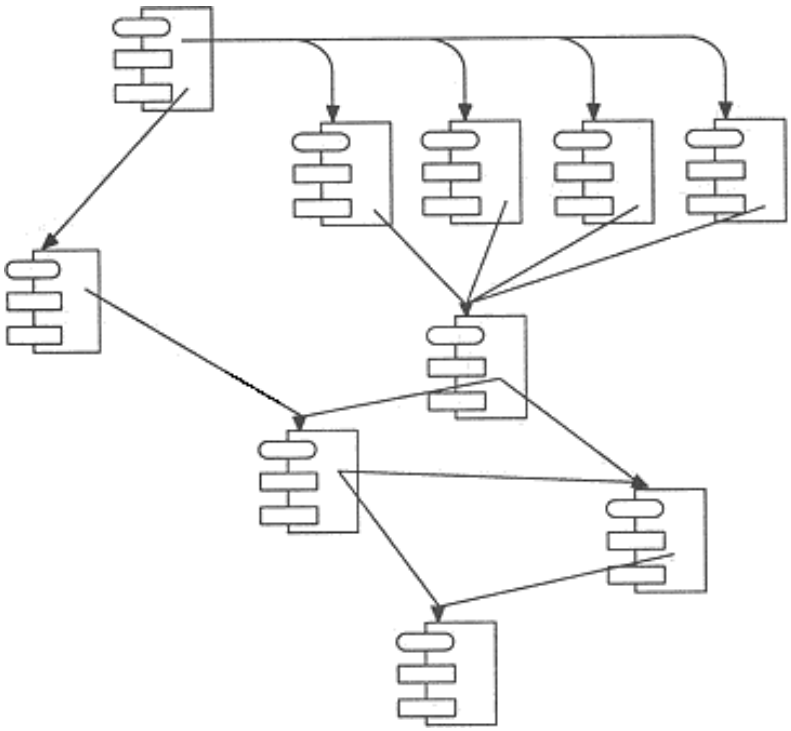


Рис. 6. Топология объектно-ориентированного программирования

Основным элементом конструкции служит модуль, составленный из логически связанных классов и объектов, а не подпрограмма.

1.4 Объектно-ориентированное программирование

Объектно-ориентированное программирование – это методология программирования, основанная на представлении программы в виде совокуп- ности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Объект – это нечто, имеющее четко определенные границы. Однако, этого недостаточно, чтобы отделить один объект от другого или дать оценку качества абстракции. Объект обладает состоянием, поведением и идентич- ностью; структура и поведение схожих объектов определяет общий для них класс; термины «экземпляр класса» и «объект» взаимозаменяемы.

Класс – это множество объектов, обладающих общей структурой, поведением и семантикой. Отдельный объект – это экземпляр класса. Класс представляет лишь абстракцию существенных свойств объекта.

Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значе- ниями каждого из этих свойств. Например: торговый автомат имеет свойство: способность принимать монеты; этому свойству соответствует дина- мическое

```
struct PersonnelRecord {
   char name[100];
   int socialSecurityNumber;
   char department[10];
   float salary;
};
```

значение – количество принятых монет. Пример описания состо- яния объекта:

Поведение объекта – это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений. Опера- цией называется определенное воздействие одного объекта на другой с це- лью вызвать соответствующую реакцию. Например, клиент может активи- зировать операции append() и pop() для того, чтобы управлять объектом- очередью:

```
class Queue {
public:
   Queue();
   Queue(const Queue&);
   virtual ~Queue();
   virtual Queue& operator=(const Queue&);
   virtual int operator==(const Queue&) const;
   int operator!=(const Queue&) const;
   virtual void clear();
   virtual void append(const void*);
   virtual void remove(int at);
   virtual int length() const;
   virtual int isEmpty() const;
   ...
};
```

Индивидуальность объекта – это такое свойство объекта, которое отли- чает его от всех других объектов. В большинстве языков программирования при создании объект именуется, поэтому многие путают адресуемость и индивидуальность. Невозможность отличить имя объекта от самого объекта является источником множества ошибок в ООП.

Лекция 2

2.1. Абстракция объекта

ООП определяет следующие основные понятие, относящиеся к объекту:

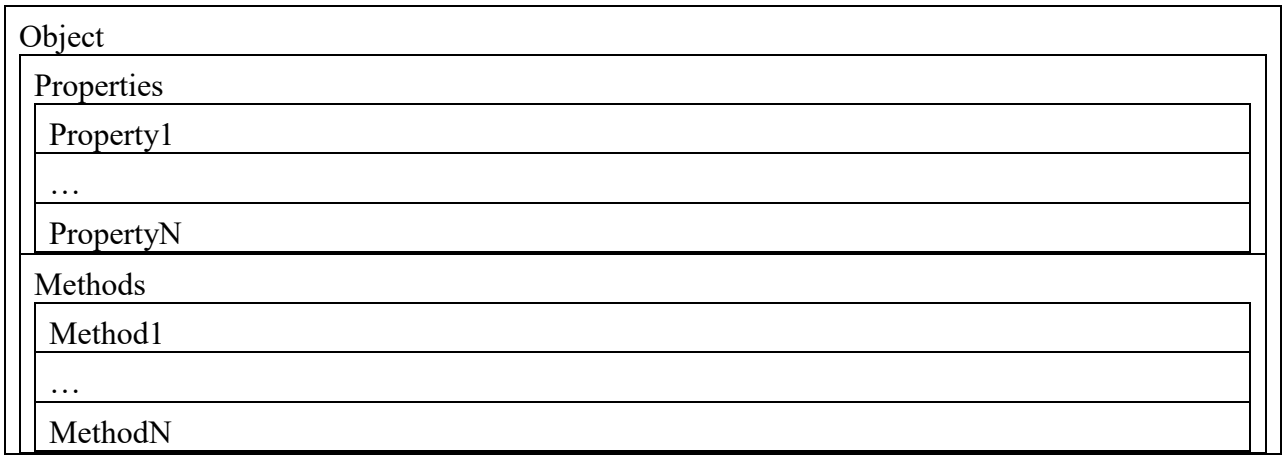
1) **Класс (Class)** – это обобщение понятия «тип данных», включающий в себя определение данных (т.е. описания структуры области оперативной памяти, которую будут использовать процедуры) и код процедур для обработки этих данных. Т.е. класс – это описание структуры данных и код процедур. Структура данных – это абстрактное описание, а код класса – это реальный код процедур. Класс – суть библиотека готовых процедур и типов данных. С точки зрения кодирования **Class** – это тип данных.
2) **Объект (Object) или экземпляр класса** – это размещенная в оперативной памяти компьютера и проинициализированная структура данных, необходимая для работы процедур класса. Обычно, экземпляр класса – это типизованный указатель на некую область памяти (адрес). При создании нового экземпляра класса ему выделяется область памяти под данные. Таким образом, экземпляры класса независимы друг от друга. Код процедур всех экземпляров одного класса – это один и тот же код. С точки зрения кодирования **Object** – это переменная (область оперативной памяти).
3) **Свойство (Property)** – это элемент данных объекта. Объект может иметь произвольное количество свойств. Свойства объекта служат для получения (чтение) результатов работы процедур объекта и для задания

(запись) данных, необходимых для работы процедур объекта. С точки зрения кодирования, свойство выглядит как обычная типизованная переменная, т.е. свойству можно присвоить значение: **Object.Property := <константа|переменная>** и прочитать значение **<переменная> := Object.Property.** Свойство объекта может быть элементарным типом данных или структурой данных, или объектом. Внешне выглядящее как обычная переменная, свойство объекта может, в реальности, быть сложной процедурой, компилятор преобразует в этом случае **Object.Property := <константа|переменная>** в вызов процедуры **Property(<константа|переменная>, <указатель на Object>)**, а вызов **<переменная> := Object.Property** в вызов функции **<переменная> := Property(<указатель на Object>)**. Таким образом достигается возможность совместить задание/чтение некоего свойства с более сложной инициализацией внутренних или возвращаемых данных и выполнением неких действий.

4) **Метод (Method)** – это процедура/функция, обрабатывающая данные объекта. С точки зрения кодирования, метод выглядит как обычная процедура и может иметь произвольное количество аргументов. Вызов метода аналогичен вызову процедуры: **Object.Method(<параметры>)**. Если заглянуть немного глубже в код, генерируемый компиляторами ООП-языка, то можно увидеть торчащие уши процедурного подхода. Вызов **Object.Method(<параметры>)**, фактически преобразуется компилятором в вызов **Method(<параметры>, <указатель на Object >).**

Если хорошенько задуматься над вышесказанным, то станет понятно, что деление на методы и свойства для объекта – суть условность и различаются они только «формальной записью обращения к ним». Т.е. свойства прикидываются переменными, а методы прикидываются процедурами.

В виде картинки ВНЕШНЕЕ представление объекта можно изобразить так:



В реальности, объект(ы) выгляд(ит/ят) так: экземпляр класса – это обычная переменная (область памяти), а методы-свойства – библиотека

процедур.

Данные	Код
Object1	Property1
…	…
ObjectN	PropertyN
	Method1
	…
	MethodN

2.2. Инкапсуляция

Инкапсуляция (англ. encapsulation) — это объектно-ориентированная концепция, позволяющая упаковывать данные и поведение в единый компонент с разделением его на видимую часть — интерфейс и невидимую часть — реализацию.

На практике, это развитие идей процедурного программирования об «ограничения области видимости».

Экземпляр объекта – суть область памяти (данные) и процедуры работы с этой областью памяти. И для данных, и для процедур существует проблема потенциальной возможности «совершить неправильные действия». Эти «неправильные действия» действия могут выражаться как в изменении данных, которые нельзя изменять, так и в неправильной последовательности действий или вызовах процедур, которые не следует вызывать.

Полностью решить эту проблему, по видимому, невозможно, но можно снизить вероятность ошибки, изолировав (спрятав, инкапсулировав) данные и процедуры, не предназначенные для «свободного изменения».

Реализация концепции инкапсуляции в ООП подходе реализуется разделением свойств (данных) и методов (процедур) на:

- «общедоступные» (public),
- «защищенные» (protected),
- «скрытые» (private).

Public – доступны как «видимые всем» свойства и методы. Именно public и образуют «интерфейс» объекта.

Protected – свойства и методы, необходимые для реализации и дальнейшего развития класса (см. далее наследование). Эти свойства и методы видны всюду в любых методах и свойствах при описании класса и реализации любых его методов, и во всех его наследниках. Т.е. когда вы пишете код метода класса – вы можете обращаться к другим protected-методам и свойствам. Но при использовании экземпляра класса – protected-методы и свойства недоступны.

Аналогично, private – свойства и методы, необходимые для реализации

класса, но доступные только в данной декларации класса и не доступные нигде более (в наследниках тоже).

Множество свойств и методов private может рассматриваться как некий аналог локальных переменных процедуры и вложенных субпроцедур.

Множество свойств protected может рассматриваться как некий аналог глобальных переменных модуля, хотя это и не совсем полная аналогия. Прямого аналога свойств и методов protected в процедурном подходе нет.

Множество свойств и методов Public может рассматриваться как некий аналог глобальных переменных и процедур.

Деление public-protected-private не абсолютное и не единственное. В различных реализациях ООП оно может быть сокращено до public-private. Существуют и расширения, например в C# реализовано четыре уровня инкапсуляции:

- public,
- protected,
- private,
- internal.

Хотя уровень internal не совсем относится к понятию класс.

Основная выгода от инкапсуляции – защищенные методы и свойства могут вызываться только разработчиками класса, т.е. «квалифицированным программистом», следовательно, можно снизить количество и уровень проверок входных данных для таких методов и свойств, предполагая, что они всегда вызываются с «допустимыми» занчениями и в «правильной» последовательности. Защищенные методы образуют некую «внутреннюю библиотеку» для класса, позволяя, с одной стороны реализовывать методы через процедурную парадигму, т.е. разбивая крупное действие на структурированную последовательность более мелких действий и, кроме того, позволяя повторно использовать код для типичных операций.

Другая выгода инкапсуляции – «снаружи», т.е. при использовании объекта, вся это громада сложного внутреннего устройства не видна. Это облегчает выбор правильного метода/свойства при работе с объектом.

Особо необходимо отметить существование для каждого класса двух специальных методов:

1) Конструктор;
2) Деструктор.

Эти методы должны быть вызваны явно или неявно (автоматически компилятором) при каждом создании/уничтожении экземпляра класса. Задача конструктора – подготовить данные объекта к использованию, т.е. инициализировать данные объекта правильными начальными значениями. Задача деструктора – удалить данные объекта, особенно ту часть, которая была размещена динамически. В новейших вариантах ООП-компиляторов (.NET) задача освобождения динамически выделенной памяти решается

автоматически, поэтому необходимости в деструкторе для простых объектов может и не быть. Но более сложные объекты, например, с подключением к внешним серверам или устройствам, могут выполнить в деструкторе процедуры отключения.

2.3. Наследование

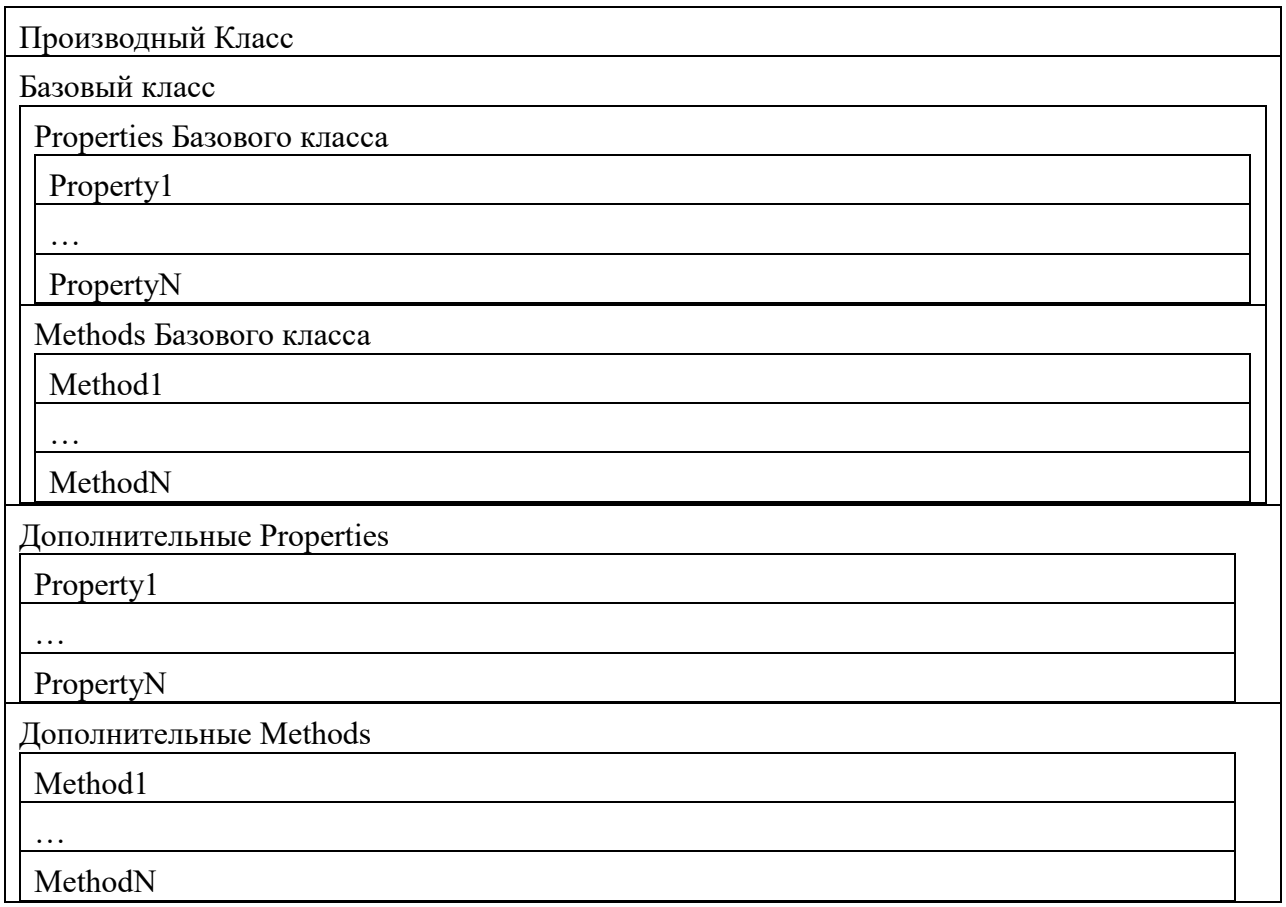
Наследование (англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой определение нового класса может наследовать данные и функциональность некоторого существующего класса и повторно использовать готовый программный код и структуры данных, дополняя его новыми свойствами и методами.

В ООП наследование реализуется как возможность объявить и описать класс на основе существующего класса. В этом случае исходный класс называют «базовым», а новый класс, построенный на основе базового, — «производным» классом.

Такой подход позволяет создавать новые классы и расширять функциональность, используя наследуемый код, т.е. свойства и методы базового класса.

В большинстве языков ООП, при наследовании, у производного класса может быть только один базовый класс. Хотя существуют и попытки реализации «множественного наследования», когда в качестве базовых могут использоваться несколько классов. Однако «множественное наследование» сталкивается с проблемами реализации.

Схематически наследование можно изобразить так



Реализация одиночного наследования в компиляторах не слишком сложна – к данным базового класса добавляются дополнительные данные «к концу» и… все. При вызове методов базового класса – они работают с той частью данных, что была унаследована от базового класса.

Производный класс волен объявлять свойства и методы, имена которых совпадают со свойствами и методами базового класса – «перекрывать» свойства и методы. Для всех пользователей обращение с такому «перекрытому» методу/свойству класса означает обращение к «последнему» варианту метода.

Однако и конечный пользователь и разработчик волен вызвать любой «предшествующий» вариант метода/свойства, для любого из предшествующих базовых классов. Это осуществляется полным указанием метода, включающим и имя класса.

Но это уже «полиморфизм».

2.4. Полиморфизм

Полиморфизм (от др.-греч. πολύμορφος — многообразный) — слово греческого происхождения, означающее «многообразие форм» и имеющее несколько аспектов смысла. В языках программирования и теории типов полиморфизмом называется способность функции обрабатывать данные разных типов. Другая «примитивная» интерпретация: полиморфизм — это способность обьекта использовать методы производного класса, которые не существуют на момент создания базового.

Существует несколько разновидностей полиморфизма в программировании. Две наиболее различных из них были описаны Кристофером Стрэчи в 1967 году: это «ad hoc» полиморфизм и параметрический полиморфизм.
Параметрический полиморфизм подразумевает исполнение одного и того же кода для всех допустимых типов аргументов. «Ad hoc» полиморфизм подразумевает исполнение потенциально разного кода для каждого типа или подтипа аргумента.
Определение полиморфизма как «один интерфейс — много реализаций», популяризованное Бьерном Страуструпом, относится к «ad hoc» полиморфизму, но в действительности, «ad hoc» полиморфизм не является истинным полиморфизмом.
Полиморфизм часто называется третьей концепцией объектно-ориентированного программирования после инкапсуляции и наследования.
По простому, полиморфизм – это возможность работать с экземплярами разных классов, вызывая «одноименные методы», т.е. один и тот же код может обрабатывать разные объекты с одноименными (и близкими по смыслу) методами.
Зачатки полиморфизма свойственны всем языкам программирования, например, встроенный оператор сложения: +. Мы пишем: 3 + 5 или 4.3 + 0.1 или 'c' + 'v' и т.п. Независимо от типов (int, double, char, ...) каждый раз мы ожидаем одного и того же результата - получения суммы операндов.
Реализация полиморфизма может быть статической или динамической. В первом случае, компилятор на этапе компиляции генерирует код вызова правильных процедур для каждого конкретного случая. Во втором случае – в ызов нужной процедуры происходит в момент исполнения кода по некоему «динамическому адресу», который содержится в данных.

2.4.1. Абстрактный класс

Это класс-шаблон. Он не реализует функциональность – он только описывает «общую желаемую функциональность» в виде набора свойств и методов. Предполагается, что эта функциональность будет реализована в наследниках абстрактного класса. Примером такого абстрактного класса может служить класс "Геометрическая фигура". Есть множество различных геометрических фигур, которые мы можем планировать создать на базе такого абстрактного класса и которые должны обладать единым набором операций над ними, например, «нарисовать», «повернуть», «передвинуть» и т.д. Аналогично они могут обладать единым набором некоторых свойств «координаты», «длина», «ширина». Абстрактный класс может продекларировать эти свойства и методы, но без реализации, как перечень необходимых.
Аналогично абстрактному классу вводится понятие «абстрактный метод или свойство». Это декларация заголовка метода или свойства без конкретного кода, реализующего данный метод или свойство.

Реализация абстрактных методов и свойств отличается от обычных методов. Прежде всего, отличается способ вызова абстрактного метода. Если обычный метод объекта вызывается «в момент компиляции», то абстрактный вызывается в момент исполнения. Абстрактные методы еще называют «виртуальными». В отличие от обычных «статических» методов, адрес для вызова «виртуального» метода хранится в данных объекта. В специальной ячейке или свойстве, которое можно условно назвать «адрес метода Имярек».
Компилятор при вызове «абстрактного» метода генерирует «вызов по адресу из ячейки памяти». Поэтому, при обработке конкретного экземпляра объекта, будет вызван именно метод, связанный с данным экземпляром.

2.4.2. Базовый класс

Разновидностью абстрактного класса может являться класс, реализующий некую общую функциональность, которую затем планируется дополнить в классах-наследниках.
Рассматривая предыдущий пример, можно реализовать (написать) код для некоторых методов, который будет применим ко всем производным классам. Такой класс называют базовым.

2.4.3. Производные классы

Производные классы – классы-наследники некоего базового класса. В этих классах абстрактные методы получают свою конкретную реализацию, применительно к уже конкретному классу.
Во время выполнения объекты производного класса могут рассматриваться как объекты базового класса в местах, где используются только тот же набор методов и свойств, что декларирован в базовом классе. Когда это происходит, экземпляр объекта может рассматриваться как экземпляр абстрактного класса, т.е. если в коде программы можно работать с «любым» экземпляром любого производного класса.
Это и есть полиморфизм, т.е. возможность работать с «разными по-сути» объектами «единообразно по форме».
Базовые (абстрактные) классы могут определять и реализовывать виртуальные методы, а производные классы могут переопределять их. Это означает, что они предоставляют свои собственные определения и реализацию.
Виртуальные методы позволяют работать с группами связанных объектов универсальным способом. Представим, например, приложение, позволяющее пользователю создавать различные виды фигур на поверхности для рисования. Во время компиляции вы еще не знаете, какие именно виды фигур создаст пользователь. При этом приложению необходимо отслеживать все различные типы создаваемых фигур и обновлять их в ответ на движения мыши. Для решения этой проблемы можно использовать полиморфизм, выполнив два основных действия:

1) Создать иерархию классов, в которой каждый отдельный класс фигур является производным из общего базового класса.
2) Применить виртуальный метод для вызова соответствующего метода на любой производный класс через единый вызов в метод базового класса.

Для начала создайте базовый класс с именем Shape и производные классы, например Rectangle, Circle и Triangle. Добавьте в класс Shape виртуальный метод с именем Draw и переопределите его в каждом производном классе для рисования конкретной фигуры, которую этот класс представляет. Создайте объект List<Shape> и добавьте в него круг, треугольник и прямоугольник. Для обновления поверхности рисования используйте цикл foreach, чтобы выполнить итерацию списка и вызвать метод Draw на каждом объекте Shape в списке. Несмотря на то, что каждый объект в списке имеет объявленный тип Shape, вызывать будет тип во время выполнения (переопределенная версия метода в каждом производном классе).

Виртуальные члены

Если производный класс наследуется из базового, он получает все методы, поля, свойства и события базового класса.Разработчик производного класса может выбрать следующее:

- переопределение виртуальных членов в базовом классе;
- наследование метода ближайшего базового класса без переопределения;
- определение новой, невиртуальной реализации тех членов, которые скрывают реализации базового класса.

Производный класс может переопределить член базового класса, только если последний будет объявлен виртуальным или абстрактным. Производный класс должен использовать ключевое слово переопределить, указывающее, что метод предназначен для участия в виртуальном вызове. Виртуальными могут быть только методы и свойства. Когда производный класс переопределяет виртуальный метод/свойство, он вызывается даже в том случае, если доступ к экземпляру этого класса осуществляется в качестве экземпляра базового класса.

Виртуальные методы и свойства позволяют производным классам расширять базовый класс без необходимости использовать реализацию метода базового класса.

Если вам нужно, чтобы производный член имел такое же имя, как и член в базовом классе, но вы не хотите, чтобы он участвовал в виртуальном вызове, используйте ключевое слово new новое. Ключевое слово new вставляется перед типом возвращаемого значения замещаемого члена класса. Доступ к скрытым членам базового класса можно по-прежнему осуществлять из клиентского кода приведением экземпляра производного класса к экземпляру базового класса.

Защита виртуальных членов от переопределения производными классами

Виртуальные члены остаются виртуальными на неограниченный срок независимо от количества классов, объявленных между виртуальным членом и классом, который объявил его изначально.Если класс A объявляет виртуальный член, класс B производится из класса A, а класс C — из класса B, то класс C наследует виртуальный член и получает возможность переопределить его независимо от того, объявляет ли класс B переопределение этого члена.

Производный класс может остановить виртуальное наследование, объявив переопределение как запечатанное. Для этого в объявление члена класса необходимо вставить ключевое слово sealed перед ключевым словом override.

Производный класс, который заменил или переопределил метод или свойство, может получить доступ к методу или свойству на базовом классе с помощью ключевого слова base.

Рекомендуется, чтобы виртуальные члены использовали base для вызова реализации базового класса этого члена в их собственной реализации. Разрешение поведения базового класса позволяет производному классу концентрироваться на реализации поведения, характерного для производного класса. Если реализация базового класса не вызывается, производный класс сопоставляет свое поведение с поведением базового класса по своему усмотрению.

Лекция 3

4.1. Общее описание C#

C# (произносится "Си-шарп") является языком программирования, который разработан для создания множества приложений, работающих в среде .NET Framework. Язык C# прост, типобезопасен и объектно-ориентирован. Благодаря множеству нововведений C# обеспечивает возможность быстрой разработки приложений, но при этом сохраняет выразительность и элегантность, присущую C-подобным языкам.

Visual C# — это реализация языка C# корпорацией Майкрософт. Поддержка Visual C# в Visual Studio обеспечивается с помощью полнофункционального редактора кода, компилятора, шаблонов проектов, конструкторов, мастеров кода, мощного и удобного отладчика и многих других средств. Библиотека классов .NET Framework предоставляет доступ ко многим службам операционной системы и к другим полезным, хорошо спроектированным классам, что существенно ускоряет цикл разработки.

4.2. Общая структура программы

Программа на языке C# может состоять из одного или нескольких файлов. Каждый файл может содержать ноль или более пространств имен. Пространство имен может включать такие элементы, как классы, структуры, интерфейсы, перечисления и делегаты, а также другие пространства имен. Ниже приведена скелетная структура программы C#, содержащая все указанные элементы.

```
// A skeleton of a C# program
using System;
namespace YourNamespace
{
        class YourClass
          {
          }

        struct YourStruct
          {
          }

        interface IYourInterface
          {
          }

    delegate int YourDelegate();

    enum YourEnum
          {
          }
```

```
namespace YourNestedNamespace
{
  struct YourStruct
        {
        }
}

class YourMainClass
{
  static int Main(string[] args)
        {
        //Your program starts here...
        }
}
```

Метод Main является точкой входа приложения C#. (Для библиотек и служб не требуется метод Main в качестве точки входа). При запуске приложения метод Main является первым вызываемым методом.

В программе C# возможна только одна точка входа. Если в наличие имеется больше одного класса, который имеет метод Main, то необходимо скомпилировать программу с параметром компилятора /main, чтобы указать, какой метод Main нужно использовать в качестве точки входа.

```
class TestClass
{
  static void Main(string[] args)
  {
    // Display the number of command line arguments:
    System.Console.WriteLine(args.Length);
  }
}
```

- Метод Main является точкой входа EXE-программы, в которой начинается и заканчивается управление программой.
- Метод Main объявляется внутри класса или структуры. Main должен быть статический и он не должен быть открытый. (В предыдущем примере он получает доступ по умолчанию типа закрытый.) Включающий класс или структура не обязательно должна быть статической.
- Main может иметь возвращаемый тип либо void, либо int.
- Метод Main может быть объявлен с параметром string[], который содержит аргументы командной строки, или без него. При использовании Visual Studio для создания приложений Windows Forms, можно добавить параметр вручную или использовать класс Environment для получения аргументов командной строки. Индексация считываемых параметров командной строки начинается с нуля. В отличие от C и C++, имя программы не рассматривается как первый аргумент командной строки.

4.3. Пространство имен (Namespace & Using)

Первым элементом, который необходимо понимать, чтобы использовать C# — это пространство имен.

В программировании на C# пространства имен используются с полной нагрузкой по двум направлениям.

Во-первых, платформа .NET Framework использует пространства имен для организации наименований множества классов. Это выполняется следующим образом.

```
System.Yes.No.Net.Console.WriteLine("Hello World!");
```

System — это пространство имен, а Console — класс в нем, WriteLine — метод класса. Таким образом определены все имена в C#. Пространства имен могут быть неограниченно вложены друг в друга: NameSpace1. NameSpace2. NameSpace3… NameSpaceN.Class.Method.

Во-вторых, объявление собственного пространства имен поможет в управлении областью действия имен классов и методов в крупных программных проектах. Т.е. вы можете называть классы и методы «одинаково» в разных пространствах имен.

Для объявления пространства имен воспользуйтесь ключевым словом *namespace*, как показано в следующем примере.

```
namespace SampleNamespace
{
   class SampleClass
   {
      public void WriteLine ()
      {
         System.Console.WriteLine(
           " WriteLine inside SampleNamespace");
      }
   }
}
```

Понятие «пространства имен» организует все имена, используемые в программе на C# в одно дерево. Пространство имен global является корневым пространством имен, например global::System всегда будет ссылаться на пространство имен платформы .NET Framework System. Все пространства имен как бы «вложены» в global, образуя дерево. Эта схема организации имен хорошо сочетается с «иерархией классов», где каждый класс имеет только одного предка – базовый класс.

Для однозначного обращения к объекту необходимо указать его полное имя. Для сокращения записи полных имен можно использовать ключевое слово *using*.

4.3.1. Доступ к пространствам имен

Использование ключевого слова using может отменить необходимость полного имени, как показано в следующем примере:

```
using System;
using System1;
Console.WriteLine("Hello");
Console.WriteLine("World!");
```

Большинство приложений на языке C# начинаются с раздела директив *using*. В этом разделе перечисляются пространства имен, которые будут часто использоваться приложением, и это спасает программиста от необходимости указывать полное имя каждый раз, когда используется содержащийся в них метод.

Например, включив строку:

```
using System;
Console.WriteLine("Hello, World!");
```

вместо кода:

```
System.Console.WriteLine("Hello, World!");
```

При наличии одноименных объектов в нескольких пространствах имен, указанных в using

```
using System1;
using System2;
```

код:

```
Console.WriteLine("Hello, World!");
```

будет использовать «последнее пространство имен using», т.е. System2.

4.3.2. Псевдонимы пространств имен

Директива using (Справочник по C#) также может использоваться для создания псевдонима пространства имен. Например, в случае использования написанного прежде пространства имен, содержащего вложенные пространства имен, можно объявить псевдоним для обеспечения быстрого способа обращения к одному из них, как показано в следующем примере:

```
using Co = Company.Proj.Nested;  // define an alias to represent a namespace
```

4.3.3. Использование пространств имен для управления областью действия

Ключевое слово *namespace* используется для объявления области действия. Возможность создавать области действия в рамках проекта помогает организовывать код и позволяет создавать глобально уникальные типы. В следующем примере в двух пространствах имен, одно из которых вложено в другое, определяется класс, названный SampleClass. Для того, чтобы различать, какой метод вызывается, используется Оператор . (справочник по C#).

```
namespace SampleNamespace
{
   class SampleClass
   {
      public void SampleMethod()
      {
         System.Console.WriteLine(
           "SampleMethod inside SampleNamespace");
      }
   }

   // Create a nested namespace, and define another class.
   namespace NestedNamespace
   {
      class SampleClass
      {
         public void SampleMethod()
         {
            System.Console.WriteLine(
              "SampleMethod inside NestedNamespace");
         }
      }
   }

   class Program
   {
      static void Main(string[] args)
      {
         // Displays "SampleMethod inside SampleNamespace."
         SampleClass outer = new SampleClass();
         outer.SampleMethod();

         // Displays "SampleMethod inside SampleNamespace."
         SampleNamespace.SampleClass outer2 = new SampleNamespace.SampleClass();
         outer2.SampleMethod();

         // Displays "SampleMethod inside NestedNamespace."
         NestedNamespace.SampleClass inner = new NestedNamespace.SampleClass();
         inner.SampleMethod();
      }
```

```
    }
}
```

4.3.4. Полные имена

Пространства имен и типы имеют уникальные названия, описываемые полными именами, показывающими логическую иерархию. Например, инструкция A.B подразумевает, что A — это имя пространства имен или типа, а B — это вложенный в него тип.

В следующем примере показаны вложенные классы и пространства имен. Полное имя указано в качестве примечания после каждой сущности.

```
namespace N1     // N1
{
   class C1      // N1.C1
   {
      class C2   // N1.C1.C2
      {
      }
   }
   namespace N2  // N1.N2
   {
      class C2   // N1.N2.C2
      {
      }
   }
}
```

В предыдущем фрагменте кода:

- пространство имен N1 является членом глобального пространства имен. Его полным именем является N1;
- пространство имен N2 является членом пространства имен N1. Его полным именем является N1.N2;
- класс C1 является членом пространства имен N1.Его полным именем является N1.C1;
- имя класса C2 используется в этом коде два раза.Однако полные имена являются уникальными.Первый экземпляр класса C2 объявлен внутри класса C1; следовательно, его полным именем является N1.C1.C2.Второй экземпляр класса C2 объявлен внутри пространства имен N2; следовательно, его полным именем является N1.N2.C2.

Используя предыдущий фрагмента кода, можно добавить новый член: класс C3, в пространство имен N1.N2, как показано ниже:

```
namespace N1.N2
{
   class C3   // N1.N2.C3
   {
   }
}
```

В общем, используйте ключевое слово :: для обращения к псевдониму пространства имен или ключевое слово global:: для обращения к глобальному пространству имен и ключевое слово . для уточнения типов или членов.

Ошибкой является использование ключевого слова :: с псевдонимом, ссылающимся на тип, а не на пространство имен. Например:

```
 using Alias = System.Console;
{
   static void Main()
   {
      // Error
      //Alias::WriteLine("Hi");

      // OK
      Alias.WriteLine("Hi");
   }
}
```

Обратите внимание, что ключевое слово global не является предопределенным псевдонимом. Следовательно, имя global.X не имеет какого-либо специального значения. Оно приобретает специальное значение только при использовании с ключевым словом ::.

В случае определения псевдонима global создается предупреждение компилятора CS0440, поскольку ключевое слово global:: всегда ссылается на глобальное пространство имен, а не на псевдоним. Например, следующая строка приведет к генерированию предупреждения:

```
using global = System.Collections;   // Warning
```

Использование ключевого слова :: с псевдонимами является правильной методикой, защищающей от неожиданного введения дополнительных типов. Рассмотрим, например, следующий фрагмент кода:

```
  using Alias = System;

  namespace Library
  {
     public class C : Alias.Exception { }
  }
```

Этот код работает, но если в последующем будет введен тип Alias, то

конструкция Alias. станет связана с этим типом. Использование конструкции Alias::Exception гарантирует, что имя Alias будет обрабатываться как псевдоним пространства имен и не будет ошибочно принято за тип.

4.3.5. Использование псевдонима глобального пространства имен

Возможность доступа к члену в глобальном пространстве имен полезна, когда этот член может быть скрыт другой сущностью с таким же именем. Например, в следующем коде Console разрешается в тип TestApp.Console вместо типа Console в пространстве имен System.

```
using System;
class TestApp
{
  // Define a new class called 'System' to cause problems.
  public class System { }

  // Define a constant called 'Console' to cause more problems.
  const int Console = 7;
  const int number = 66;

  static void Main()
  {
    // The following line causes an error. It accesses TestApp.Console,
    // which is a constant.
    //Console.WriteLine(number);
  }
}
```

При использовании System.Console все равно возникает ошибка, так как пространство имен System скрыто классом TestApp.System:

```
// The following line causes an error. It accesses TestApp.System,
// which does not have a Console.WriteLine method.
System.Console.WriteLine(number);
```

Однако данную ошибку можно устранить следующим образом, используя global::System.Console:

```
// OK
global::System.Console.WriteLine(number);
```

Когда левый идентификатор имеет значение global, поиск правого идентификатора начинается в глобальном пространстве имен. Например, следующее объявление ссылается на TestApp как на член глобального пространства.

```
class TestClass : global::TestApp
```

Очевидно, что необходимость в создании пространства имен System

отсутствует, а вероятность повстречать код, где это сделано, очень мала. Однако в более крупных проектах существует большая вероятность встретить ту или иную форму дублирования пространства имен. В такой ситуации квалификатор глобального пространства имен гарантирует возможность задания корневого пространства имен.

В данном примере пространство имен System используется для включения класса TestClass, поэтому для создания ссылки на класс System.Console, скрытый пространством имен System, необходимо использовать global::System.Console.Кроме того, псевдоним colAlias используется для создания ссылки на пространство имен System.Collections; таким образом при создании экземпляра System.Collections.Hashtable вместо пространства имен использовался этот псевдоним.

```
using colAlias = System.Collections;
namespace System
{
   class TestClass
   {
      static void Main()
      {
         // Searching the alias:
         colAlias::Hashtable test = new colAlias::Hashtable();

         // Add items to the table.
         test.Add("A", "1");
         test.Add("B", "2");
         test.Add("C", "3");

         foreach (string name in test.Keys)
         {
            // Searching the global namespace:
            global::System.Console.WriteLine(name + " " + test[name]);
         }
      }
   }
}
```

4.3.6. Физическое расположение пространства имен

Следует помнить, что пространства имен имеют физическую локализацию. Любое пространство имен для C# содержится в физическом файле динамической библиотеки (библиотека может содержать несколько пространств имен), но главное – не бывает пространства имен без файла-библиотеки.

Системные (встроенные) пространства имен и системные (встроенные) библиотеки доступны «по умолчанию». Но для любого «пользовательского» пространства имен — библиотека должна быть указана явно. Однако это уже не «пространство имен», а лишь «практическая реализация». Способы «указания ссылки на библиотеку» зависят от используемого компилятора.

Лекция 4

Класс — это логическая структура, позволяющая создавать свои собственные пользовательские типы путем группирования переменных других типов, методов и событий. Класс подобен чертежу. Он определяет данные и поведение типа. Если класс не объявлен статическим, то клиентский код может его использовать, создав объекты или иначе экземпляры, приписанные переменной. Переменная остается в памяти, пока все ссылки на нее не выйдут из области видимости. В это время среда CLR помечает ее пригодной для сборщика мусора. Если класс объявляется статическим, то в памяти остается только одна копия и клиентский код может получить к ней доступ только посредством самого класса, а не переменной экземпляра.

В отличие от структур классы поддерживают наследование, фундаментальную характеристику объектно-ориентированного программирования.

4.5.1. Объявление класса

Классы объявляются с помощью ключевого слова class, как показано в следующем примере.

```
public class Customer
{
   //Fields, properties, methods and events go here...
}
```

Ключевому слову class предшествует уровень доступа. Поскольку в данном случае используется public, любой может создавать объекты из этого класса. Имя класса указывается после ключевого слова class. Оставшаяся часть определения является телом класса, в котором задаются данные и поведение. Поля, свойства, методы и события в классе обозначаются термином члены класса.

4.5.2. Создание объектов

Класс и объект — это разные вещи, хотя в некоторых случаях они взаимозаменяемы. Класс определяет тип объекта, но не сам объект. Объект — это конкретная инициализированная область памяти (переменная), основанная на классе и называемая экземпляром класса.

Объекты можно создавать с помощью ключевого слова new, за которым

следует имя класса, на котором будет основан объект:

```
Customer object1 = new Customer();
```

При создании экземпляра класса ссылка на этот объект передается программисту. В предыдущем примере object1 является ссылкой на объект, основанный на Customer. Эта ссылка указывает на новый объект, но не содержит данные этого объекта. Фактически, можно создать ссылку на объект без создания самого объекта:

```
Customer object2;
```

Создание таких ссылок, которые не указывают на объект, не рекомендуется, так как попытка доступа к объекту по такой ссылке приведет к сбою во время выполнения. Однако такую ссылку можно сделать указывающей на объект, создав новый объект или назначив ее существующему объекту:

```
Customer object3 = new Customer();
Customer object4 = object3;
```

В данном коде создаются две ссылки на объекты, которые указывают на один объект. Поэтому любые изменения объекта, выполненные посредством object3, будут видны при последующем использовании object4. Поскольку на объекты, основанные на классах, указывают ссылки, классы называют ссылочными типами.

Интерфейсы

Интерфейс является ссылочным типом, в чем-то схожим с абстрактным базовым классом, который состоит только из абстрактных членов. Когда класс реализует интерфейс, он должен предоставить реализацию для всех членов интерфейса. В классе может быть реализовано несколько интерфейсов, хотя производным он может быть только от одного прямого базового класса.

Интерфейсы используются для определения определенных возможностей для классов, которые не обязательно имеют отношения тождественности. Например, интерфейс System.IEquatable<T> может быть реализован любым классом или структурой, включающей клиентский код для определения эквивалентности двух объектов типа. IEquatable<T> не подразумевает тот же вид отношений тождественности, который существует между базовым и производным классами (например, Mammal является Animal).

Доступ производного класса к членам базового класса

Из производного класса можно получить доступ к открытым, защищенным, внутренним и защищенным внутренним членам базового класса. Хотя производный класс и наследует закрытые члены базового класса, он не может получить доступ к этим членам. Однако все эти закрытые члены все же присутствуют в производном классе и могут выполнять ту же работу, что и в самом базовом классе. Например, предположим, что защищенный метод базового класса имеет доступ к закрытому полю. Это поле должно присутствовать в производном классе

для правильной работы унаследованного метода базового класса.

Предотвращение дальнейшего наследования

Класс может предотвратить наследование от него других классов или наследование от любых его членов, объявив себя или члены запечатанными.

Скрытие производного класса членов базового класса

Производный класс может скрывать члены базового класса путем объявления членов с тем же именем и сигнатурой. Модификатор new может использоваться, чтобы явно указать, что член не предназначен, чтобы быть переопределением базового члена. Использование new не является обязательным, но при отсутствии использования new будет сгенерировано предупреждение компилятора.

4.5.4. Полиморфизм

Полиморфизм часто называется третьей концепцией объектно-ориентированного программирования после инкапсуляции и наследования. Полиморфизм — слово греческого происхождения, означающее «многообразие форм» и имеющее несколько аспектов.

- Во время выполнения объекты производного класса могут рассматриваться как объекты базового класса в таких местах как параметры метода и коллекции или массивы. Когда это происходит, объявленный тип объекта перестает соответствовать своему типу во время выполнения.
- Базовые классы могут определять и реализовывать виртуальные методы, а производные классы могут переопределять их. Это означает, что они предоставляют свои собственные определения и реализацию. Во время выполнения, когда клиент вызывает метод, CLR выполняет поиск типа объекта и вызывает перезапись виртуального метода. Таким образом, в исходном коде можно вызвать метод на базовом классе и привести версию производного класса метода, который необходимо выполнить.

Виртуальные методы позволяют работать с группами связанных объектов универсальным способом. Представим, например, приложение, позволяющее пользователю создавать различные виды фигур на поверхности для рисования. Во время компиляции вы еще не знаете, какие именно виды фигур создаст пользователь. При этом приложению необходимо отслеживать все различные типы создаваемых фигур и обновлять их в ответ на движения мыши. Для решения этой проблемы можно использовать полиморфизм, выполнив два основных действия.

1) Создать иерархию классов, в которой каждый отдельный класс фигур является производным из общего базового класса.
2) Применить виртуальный метод для вызова соответствующего метода на любой производный класс через единый вызов в метод базового класса.

Для начала создайте базовый класс с именем Shape и производные классы,

например Rectangle, Circle и Triangle. Добавьте в класс Shape виртуальный метод с именем Draw и переопределите его в каждом производном классе для рисования конкретной фигуры, которую этот класс представляет. Создайте объект List<Shape> и добавьте в него круг, треугольник и прямоугольник. Для обновления поверхности рисования используйте цикл foreach, чтобы выполнить итерацию списка и вызвать метод Draw на каждом объекте Shape в списке. Несмотря на то, что каждый объект в списке имеет объявленный тип Shape, вызывать будет тип во время выполнения (переопределенная версия метода в каждом производном классе).

```
public class Shape
{
   // A few example members
   public int X { get; private set; }
   public int Y { get; private set; }
   public int Height { get; set; }
   public int Width { get; set; }

   // Virtual method
   public virtual void Draw()
   {
      Console.WriteLine("Performing base class drawing tasks");
   }
}
```

```
class Circle : Shape
{
   public override void Draw()
   {
      // Code to draw a circle...
      Console.WriteLine("Drawing a circle");
      base.Draw();
   }
}
class Rectangle : Shape
{
   public override void Draw()
   {
      // Code to draw a rectangle...
      Console.WriteLine("Drawing a rectangle");
      base.Draw();
   }
}
class Triangle : Shape
{
   public override void Draw()
   {
      // Code to draw a triangle...
      Console.WriteLine("Drawing a triangle");
      base.Draw();
   }
}

class Program
{
   static void Main(string[] args)
   {
      // Polymorphism at work #1: a Rectangle, Triangle and Circle
      // can all be used whereever a Shape is expected. No cast is
      // required because an implicit conversion exists from a derived
      // class to its base class.
      System.Collections.Generic.List<Shape> shapes = new
System.Collections.Generic.List<Shape>();
      shapes.Add(new Rectangle());
      shapes.Add(new Triangle());
```

```
        shapes.Add(new Circle());

        // Polymorphism at work #2: the virtual method Draw is
        // invoked on each of the derived classes, not the base class.
        foreach (Shape s in shapes)
        {
            s.Draw();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

}

/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
 */
```

В C# каждый тип является полиморфным, так как все типы, включая пользовательские, наследуют Object.

Лекция 5

4.5.5. Статические классы и члены статических классов

Статические классы имеют одно отличие: **нельзя создавать экземпляры статического класса**. Другими словами, нельзя использовать ключевое слово new для создания переменной типа класса. Поскольку нет переменной экземпляра, доступ к членам статического класса осуществляется с использованием самого имени класса (т.е. статический класс – суть статическая/глобальная переменная). Например, если имеется статический класс, называемый UtilityClass, имеющий открытый метод, называемый MethodA, вызов метода выполняется, как показано в следующем примере.

```
UtilityClass.MethodA();
```

Статический класс – является глобальной областью памяти, выделенной под экземпляр класса статически, и набором методов.

Статический класс может использоваться как обычный контейнер для наборов методов, работающих на входных параметрах, и **не должен возвращать или устанавливать каких-либо внутренних полей экземпляра.** Например, в библиотеке классов платформы .NET Framework статический класс System.Math содержит методы, выполняющие математические операции, без требования сохранять или извлекать данные, уникальные для конкретного экземпляра класса Math. Это значит, что члены класса применяются путем задания имени класса и имени метода, как показано в следующем примере.

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

Как и в случае с типами всех классов сведения о типе для статического класса загружаются средой CLR .NET Framework, когда загружается программа, которая ссылается на класс. Программа не может точно указать, когда загружается класс. Но гарантируется загрузка этого класса, инициализация его полей и вызов статического конструктора перед первым обращением к классу в программе. **Статический конструктор вызывается только один раз, и статический класс остается в памяти на время существования домена приложения, в котором находится программа.**

Возможно создание нестатического класса, который допускает создание только одного экземпляра самого себя, см. Singleton в C#.

Следующий список предоставляет основные характеристики статического класса:

- Содержит только статические члены.
- Создавать его экземпляры нельзя.
- Он запечатан.
- Не может содержать конструкторов экземпляров.

По сути, создание статического класса аналогично созданию класса, содержащего только статические члены и закрытый конструктор. Закрытый конструктор не допускает создания экземпляров класса. Преимущество применения статических классов заключается в том, что компилятор может проверить отсутствие случайно добавленных членов

экземпляров. Таким образом, компилятор гарантирует невозможность создания экземпляров таких классов.

Статические классы запечатаны, поэтому их нельзя наследовать. Они не могут быть унаследованы ни от каких классов, кроме Object. Статические классы не могут содержать конструктор экземпляров, но могут содержать статический конструктор. Нестатические классы также должны определять статический конструктор, если класс содержит статические члены, для которых нужна нетривиальная инициализация.

Ниже приведен пример статического класса, содержащего два метода, преобразующих температуру по Цельсию в температуру по Фаренгейту и наоборот.

```
public static class TemperatureConverter
{
   public static double CelsiusToFahrenheit(string temperatureCelsius)
   {
      // Convert argument to double for calculations.
      double celsius = Double.Parse(temperatureCelsius);

      // Convert Celsius to Fahrenheit.
      double fahrenheit = (celsius * 9 / 5) + 32;

      return fahrenheit;
   }

   public static double FahrenheitToCelsius(string temperatureFahrenheit)
   {
      // Convert argument to double for calculations.
      double fahrenheit = Double.Parse(temperatureFahrenheit);

      // Convert Fahrenheit to Celsius.
      double celsius = (fahrenheit - 32) * 5 / 9;

      return celsius;
   }
}
```

```
class TestTemperatureConverter
{
   static void Main()
   {
      Console.WriteLine("Please select the convertor direction");
      Console.WriteLine("1. From Celsius to Fahrenheit.");
      Console.WriteLine("2. From Fahrenheit to Celsius.");
      Console.Write(":");

      string selection = Console.ReadLine();
      double F, C = 0;

      switch (selection)
      {
         case "1":
            Console.Write("Please enter the Celsius temperature: ");
            F = TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine());
            Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
            break;

         case "2":
            Console.Write("Please enter the Fahrenheit temperature: ");
            C = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
            Console.WriteLine("Temperature in Celsius: {0:F2}", C);
            break;

         default:
            Console.WriteLine("Please select a convertor.");
            break;
      }
```

```
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Example Output:
   Please select the convertor direction
   1. From Celsius to Fahrenheit.
   2. From Fahrenheit to Celsius.
   :2
   Please enter the Fahrenheit temperature: 20
   Temperature in Celsius: -6.67
   Press any key to exit.
 */
```

Статические члены

Нестатический класс может содержать статические методы, поля, свойства или события. Статический член вызывается для класса даже в том случае, если не создан экземпляр класса. Доступ к статическому члену всегда выполняется по имени класса, а не по имени экземпляра. **Существует только одна копия статического члена, независимо от того, сколько создано экземпляров класса.**

Статические методы и свойства не могут обращаться к нестатическим полям и событиям в их содержащем типе, и они не могут обращаться к переменной экземпляра объекта, если он не передается явно в параметре метода.

Более типично объявлять нестатический класс с несколькими статическими членами, чем объявлять весь класс как статический. **Статические поля обычно используются для следующих двух целей: хранение счетчика числа созданных объектов, или хранение значения, которое должно совместно использоваться всеми экземплярами.**

Статические методы могут быть перегружены, но не переопределены, поскольку они относятся к классу, а не к экземпляру класса.

Хотя поле, не может быть объявлено как static const, поле const по своему поведению является статическим. Оно относится к типу, а не к экземплярам типа. Поэтому к полям const можно обращаться с использованием той же нотации ClassName.MemberName, что используется для статических полей. Экземпляр объекта не требуется.

C# не поддерживает статических локальных переменных (переменных, которые объявлены в области действия метода).

Для объявления статических методов класса используется ключевое слово static перед возвращаемым типом члена, как показано в следующем примере:

```
public class Automobile
{
   public static int NumberOfWheels = 4;
   public static int SizeOfGasTank
   {
      get
      {
         return 15;
      }
   }
   public static void Drive() { }
   public static event EventType RunOutOfGas;

   // Other non-static fields and properties...
}
```

Статические члены инициализируются перед первым доступом к статическому члену и перед вызовом статического конструктора, если он имеется. Для доступа к статическому члену класса следует использовать имя класса, а не имя переменной, указывая расположение члена, как показано в следующем примере:

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

Если класс содержит статические поля, должен быть предоставлен статический конструктор, который инициализирует эти поля при загрузке класса.

Вызов статического метода генерирует инструкцию вызова в промежуточном языке Microsoft (MSIL), в то время как вызов метода экземпляра генерирует инструкцию callvirt, которая также проверяет наличие ссылок на пустые объекты. Однако чаще всего разница в производительности двух видов вызовов несущественна.

4.5.6. Члены класса

Мы упоминали только методы и свойства, но класс C# может содержать и иные члены.

В классах и структурах есть члены, представляющие их данные и поведение. Члены класса включают все члены, объявленные в этом классе, а также все члены (кроме конструкторов и деструкторов), объявленные во всех классах в иерархии наследования данного класса. Закрытые члены в базовых классах наследуются, но недоступны из производных классов.

В следующей таблице перечислены виды членов, которые могут содержаться в классе или в структуре.

Элемент	Описание
Поля	Поля являются переменными, объявленными в области класса. Поле может иметь встроенный числовой тип или быть экземпляром другого класса. Например, в классе календаря может быть поле, содержащее текущую дату.
Константы	Константы — это поля или свойства, значения которых устанавливаются во время компиляции и не изменяются.
Свойства	Свойства — это методы класса. Доступ к ним осуществляется так же, как если бы они были полями этого класса. Свойство может защитить поле класса от изменений (независимо от объекта).
Методы	Методы определяют действия, которые может выполнить класс. Методы могут получать параметры, предоставляющие входные данные, и возвращать выходные данные посредством параметров. Также методы могут возвращать значения напрямую, без использования параметров.
События	События предоставляют другим объектам уведомления о различных случаях, таких как нажатие кнопки или успешное выполнение метода. События определяются и переключаются с помощью делегатов.
Операторы	Перегруженные операторы рассматриваются как члены класса. При перегрузке оператора его следует определить как открытый статический метод в классе. Предопределенные операторы (+, *, < и т. д.) не считаются членами. Для получения дополнительной информации см. Перегружаемые операторы .
Индексаторы	Индексаторы позволяют индексировать объекты аналогично массивам.
Конструкторы	Конструкторы — это методы классов, вызываемые при создании объекта заданного типа. Зачастую они используются для инициализации данных объекта.
Деструкторы	Деструкторы очень редко используются в C#. Они являются методами, вызываемыми средой выполнения, когда объект нужно удалить из памяти. Деструкторы обычно применяются для правильной обработке ресурсов, которые должны быть высвобождены.
Вложенные типы	Вложенными типами являются типы, объявленные в другом типе. Вложенные типы часто применяются для описания объектов, использующихся только типами, в которых эти объекты находятся.

4.5.7. Модификаторы доступа

Все типы и члены типов имеют уровень доступности, который определяет возможность их использования из другого кода в сборке разработчика или других сборках. Можно использовать следующие модификаторы доступа для указания доступности типа или члена при объявлении этого типа или члена.

public — Доступ к типу или члену возможен из любого другого кода в той же сборке или другой сборке, ссылающейся на него.

private — Доступ к типу или члену можно получить только из кода в том же классе или структуре.

protected — Доступ к типу или элементу можно получить только из кода в том же классе или структуре, либо в производном классе.

internal — Доступ к типу или члену возможен из любого кода в той же сборке, но не из другой сборки.

protected internal — Доступ к типу или элементу может осуществляться любым кодом в сборке, в которой он объявлен, или из наследованного класса другой сборки. Доступ из другой сборки должен осуществляться в пределах объявления класса, производного от класса, в котором объявлен защищенный внутренний элемент, и должен происходить через экземпляр типа производного класса.

В следующих примерах демонстрируется указание модификаторов доступа для типа или члена.

```
public class Bicycle
{
   public void Pedal() { }
}
```

Не все модификаторы доступа могут использоваться всеми типами или членами во всех контекстах, а в некоторых случаях доступность члена типа ограничивается доступностью его содержащего типа. Следующие подразделы содержат дополнительные сведения о доступности.

Доступность класса и структуры

Классы и структуры, объявленные непосредственно в пространстве имен (другими словами, не вложенные в другие классы или структуры) могут быть открытыми (public) или внутренними (internal). Если модификатор доступа не указан, по умолчанию используется внутренний тип (internal).

Члены структуры, включая вложенные классы и структуры, могут быть объявлены как открытые, внутренние или закрытые. Члены классов, включая вложенные классы и структуры, могут быть открытыми, защищенными внутренними, защищенными, внутренними или закрытыми. Уровень доступа для членов класса и членов структуры, включая вложенные классы и структуры, является закрытым по умолчанию. Закрытые вложенные типы недоступны за пределами содержащего типа.

Производные классы не могут обладать лучшей доступностью, чем их базовые типы. Другими словами, открытый класс B, являющийся производным от внутреннего класса A, не может использоваться. Его использование может быть приравнено к переводу класса A в открытый тип, поскольку все защищенные или внутренние члены A доступны из производного класса.

При помощи InternalsVisibleToAttribute можно сделать возможным доступ

других определенных сборок к внутренним типам.

Доступность члена класса и структуры

Члены класса (включая вложенные классы и структуры) можно объявить с любым из пяти типов доступа. Члены структуры нельзя объявлять защищенными, так как структуры не поддерживают наследование.

Как правило, доступность члена никогда не выше доступности содержащего его типа. **Однако открытый член внутреннего класса может быть доступен за пределами сборки, если член реализует методы интерфейса или переопределяет виртуальные методы, определенные в открытом базовом классе.**

Тип любого элемента, являющегося полем, свойством или событием, должен, по меньшей мере, быть таким же доступным, как и этот элемент. Аналогичным образом тип возвращаемого значения и типы параметров любого члена, который явл. методом, индексатором или делегатом, должны иметь по меньшей мере такой же уровень доступности, как сам элемент. Например, метод M, возвращающий класс C не может быть открытым, если C также не является открытым. Подобным образом, свойство типа A не может быть защищенным, если A объявлен закрытым.

Определенные пользователям операторы также должны быть объявлены как открытые.

Деструкторы не могут иметь модификаторов доступности.

Чтобы задать уровень доступа для класса или элемента структуры, воспользуйтесь соответствующим ключевым словом в объявлении элемента, как показано в следующем примере.

```
// public class:
public class Tricycle
{
   // protected method:
   protected void Pedal() { }

   // private field:
   private int wheels = 3;

   // protected internal property:
   protected internal int Wheels
   {
      get { return wheels; }
   }
}
```

Защищенный внутренний уровень доступности означает "защищенный OR внутренний", а не "защищенный AND внутренний". Другими словами, доступ к защищенному внутреннему члену может осуществляться из любого класса в одной сборке, в том числе

из производных классов. Чтобы ограничить доступность только производными классами в одной и той же сборке, сам класс необходимо объявить внутренним, а его члены – защищенными.

Другие типы

Интерфейсы, объявленные непосредственно в пространстве имен, можно объявить как открытые или внутренние, и подобно классам и структурам, интерфейсам по умолчанию присваивается внутренний доступ. Члены интерфейса всегда являются открытыми, поскольку целью интерфейса является предоставление возможности доступа к классу или структуре другим типам. Модификаторы доступа нельзя применить к членам интерфейса.

Члены перечисления всегда являются открытыми, и модификаторы доступа не применяются.

Делегаты ведут себя как классы и структуры. По умолчанию они имеют внутренний доступ, когда они объявлены непосредственно внутри пространства имен, и закрытый доступ, когда они являются вложенными.

4.5.8. Поля

Поле – это переменная любого типа, которая объявлена непосредственно в классе или структуре. Поле внешне выглядит как свойство, но при присваивании полю не выполняется никакого дополнительного кода.

Поля являются членами содержащих их типов.

Класс или структура могут иметь поля экземпляра или статические поля, либо поля обоих типов. Поля экземпляра определяются экземпляром типа. Если имеется класс T и поле экземпляра F, можно создать два объекта типа T и изменить значение поля F в каждом объекте, не изменяя значение в другом объекте. В противоположность этому, статическое поле относится к самому классу, и является общим для всех экземпляров этого класса. Изменения, выполненные из экземпляра A, будут немедленно видны экземплярам B и C, если они обращаются к полю.

Как правило, используются поля только для переменных, имеющих доступность private или protected. Данные, которые класс открывает для клиентского кода, должны предоставляться через методы, свойства и индексаторы. методами, свойствами и индексаторами Используя эти конструкции для косвенного доступа к внутренним полям, можно защититься от недопустимых входных значений. Закрытое поле, которое хранит данные, представленные открытым свойством, называется резервным хранилищем или резервным полем.

Поля обычно хранят данные, которые должны быть доступными нескольким методам класса и должны храниться дольше, чем время существования любого отдельного метода. Например, в классе, представляющем календарную дату, может быть три целочисленных поля: одно для месяца, одно для числа, одно для года. Переменные, не используемые вне области одного метода, должны быть объявлены как локальные переменные внутри тела самого метода.

Поля объявляются в блоке класса путем указания уровня доступа поля, за которым следует тип поля и имя поля. Например:

```
public class CalendarEntry
{
   // private field
   private DateTime date;

   // public field (Generally not recommended.)
   public string day;

   // Public property exposes date field safely.
   public DateTime Date
   {
      get
      {
         return date;
      }
      set
      {
         // Set some reasonable boundaries for likely birth dates.
         if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
         {
            date = value;
         }
         else
            throw new ArgumentOutOfRangeException();
      }

   }

   // Public method also exposes date field safely.
   // Example call: birthday.SetDate("1975, 6, 30");
   public void SetDate(string dateString)
   {
      DateTime dt = Convert.ToDateTime(dateString);
```

```
      // Set some reasonable boundaries for likely birth dates.
      if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
      {
         date = dt;
      }
      else
         throw new ArgumentOutOfRangeException();
   }

   public TimeSpan GetTimeSpan(string dateString)
   {
      DateTime dt = Convert.ToDateTime(dateString);

      if (dt != null && dt.Ticks < date.Ticks)
      {
         return date - dt;
      }
      else
         throw new ArgumentOutOfRangeException();

   }
}
```

Для доступа к члену объекта нужно добавить точку после имени объекта и указать имя поля: objectname.fieldname. Например:

```
CalendarEntry birthday = new CalendarEntry();
birthday.day = "Saturday";
```

Полю можно назначить первоначальное значение, используя оператор присвоения при объявлении поля. Например, чтобы автоматически присвоить полю day значение "Monday", можно объявить поле day как указано в следующем примере:

```
public class CalendarDateWithInitialization
{
   public string day = "Monday";
   //...
}
```

Поля инициализируются непосредственно перед вызовом конструктора для экземпляра объекта. Если конструктор присваивает полю значение, оно заменит значения, присвоенные при объявлении поля.

Инициализатор поля не может ссылаться на другие поля экземпляров.

Поля могут быть отмечены модификаторами public, private, protected, internal или protected internal. Эти модификаторы доступа определяют порядок доступа к полю для пользователей класса.

Также при необходимости поле может быть объявлено с модификатором static. При этом поле становится доступным для вызова в любое время, даже когда экземпляр класса отсутствует.

Также при необходимости поле может быть объявлено с модификатором readonly. Полю с этим модификатором (то есть полю, доступному только для чтения) значения могут быть присвоены только при инициализации или в конструкторе. Поле с модификаторами staticreadonly (статическое, доступное только для чтения) очень похоже на константу, за исключением того, что компилятор C# не имеет доступа к значению такого поля при компиляции: доступ возможен только во время выполнения.

4.5.9. Константы

Константы представляют собой неизменные значения, известные во время компиляции и неизменяемые на протяжении времени существования программы. Константы объявляются с модификатором const. Только встроенные типы C# (за исключением System.Object) могут быть объявлены как const. Список встроенных типов см. в разделе Таблица встроенных типов (Справочник по C#). Определяемые пользователем типы, включая классы, структуры и массивы, не могут быть const. Для создания класса, структуры или массива, которые инициализируются один раз во время выполнения (например, в конструкторе) и после этого не могут быть изменены, используется модификатор readonly.

Язык C# не поддерживает методы, свойства и события с ключевым словом const.

Тип перечисления позволяет определять именованные константы для целочисленных встроенных типов (например, int, uint, long и т. д.).

Константы должны инициализировать сразу после объявления. Например:

```
class Calendar1
{
   public const int months = 12;
}
```

В этом примере константа months всегда имеет значение 12, и ее значение не может быть изменено даже самим классом. Когда компилятор встречает идентификатор константы в исходном коде C# (например, months), он подставляет литеральное значение непосредственно в его создающий код IL. **Поскольку адрес переменной, связанный с константой во время выполнения, отсутствует, поля const не могут быть переданы по ссылке и отображены как значение l-value в выражении.**

При ссылке на значения констант, определенных в другом коде, например DLL, следует соблюдать осторожность. Если новое значение константы определяется в новой версии DLL, программа по-прежнему будет хранить старое литеральное значение вплоть до перекомпиляции в новую версию.

Несколько констант одного типа можно объявить одновременно, например:

```
class Calendar2
{
   const int months = 12, weeks = 52, days = 365;
}
```

Используемое для инициализации константы выражение может ссылаться на другую константу, если при этом не создается циклическая ссылка. Например:

```
class Calendar3
{
   const int months = 12;
   const int weeks = 52;
   const int days = 365;

   const double daysPerWeek = (double) days / (double) weeks;
   const double daysPerMonth = (double) days / (double) months;
}
```

Константы могут быть отмечены модификаторами public, private, protected, internal или protected internal. Эти модификаторы доступа определяют способ доступа к константе для пользователей класса.

Доступ к константам осуществляется так, как если бы они были статическими полями, поскольку значение константы одинаково для всех экземпляров типа. Для их объявления не нужно использовать ключевое слово static. **В выражениях, которые не входят в класс, в котором определена константа, для доступа к ней необходимо использовать имя класса, точку и имя этой константы.** Например:

```
int birthstones = Calendar.months;
```

Лекция 6

4.5.3. Наследование классов

Наследование выполняется с помощью образования производных, то есть класс объявляется с помощью базового класса, от которого он наследует данные и поведение. Базовый класс задается добавлением после имени производного класса двоеточия и имени базового класса:

```
public class Manager : Employee
{
   // Employee fields, properties, methods and events are inherited
   // New Manager fields, properties, methods and events go here...
}
```

Когда класс объявляет базовый класс, он наследует все члены базового класса, за исключением конструкторов. Конструкторы имеют то же имя, что и класс или структура, и они обычно инициализируют элементы данных нового объекта.

В отличие от C++, класс в C# может только напрямую наследовать от одного базового класса. Однако, поскольку базовый класс может сам наследовать от другого класса, класс может косвенно наследовать несколько базовых классов. Кроме того, класс может напрямую реализовать несколько интерфейсов (об этом позже…).

Класс может быть объявлен абстрактным. Абстрактный класс содержит абстрактные методы, которые имеют имена, но не имеют реализации. Нельзя создавать экземпляры абстрактных классов. Они могут использоваться только посредством производных классов, реализующих абстрактные методы.

И наоборот, запечатанный класс не позволяет другим классам быть от него производными.

Определения классов можно разделить между различными исходными файлами.

В следующем примере определяются открытый класс, содержащий одно поле, метод и специальный метод, называемый конструктором. После этого с помощью ключевого слова new создается экземпляр класса.

```
public class Person
{
   // Field
   public string name;
   // Constructor that takes no arguments.
   public Person()
   {
      name = "unknown";
   }
   // Constructor that takes one argument.
   public Person(string nm)
   {
      name = nm;
   }

   // Method
   public void SetName(string newName)
   {
      name = newName;
   }
}
```

```
class TestPerson
{
   static void Main()
   {
      // Call the constructor that has no parameters.
      Person person1 = new Person();
      Console.WriteLine(person1.name);

      person1.SetName("John Smith");
      Console.WriteLine(person1.name);

      // Call the constructor that has one parameter.
      Person person2 = new Person("Sarah Jones");
      Console.WriteLine(person2.name);

      // Keep the console window open in debug mode.
      Console.WriteLine("Press any key to exit.");
      Console.ReadKey();
   }
}
// Output:
// unknown
// John Smith
// Sarah Jones
```

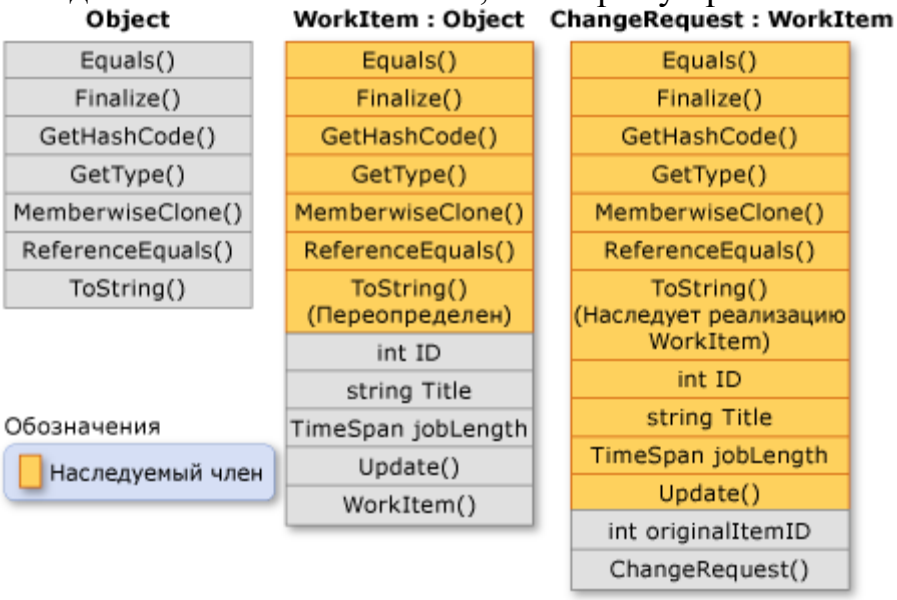
Наследование, вместе с инкапсуляцией и полиморфизмом, является одной из трех основных характеристик (или базовых понятий) объектно-ориентированного программирования. Наследование позволяет создавать новые классы, которые повторно используют, расширяют и изменяют поведение, определенное в других классах. Класс, члены которого наследуются, называется базовым классом, а класс, который наследует эти члены, называется производным классом. Производный класс может иметь только один непосредственный базовый класс. Однако наследование является транзитивным. Если ClassC является производным от ClassB, и ClassB является производным от ClassA, ClassC наследует члены, объявленные в ClassB и ClassA.

Концептуально, производный класс является специализацией базового класса. Например, при наличии базового класса Animal, возможно наличие одного производного класса, который называется Mammal, и еще одного производного класса, который называется Reptile. Mammal является Animal и Reptile является Animal, но каждый производный класс представляет разные специализации базового класса.

При определении класса для наследования от другого класса, производный

класс явно получает все члены базового класса, за исключением его конструкторов и деструкторов. Производный класс может таким образом повторно использовать код в базовом классе без необходимости в его повторной реализации. В производном классе можно добавить больше членов. Таким образом, производный класс расширяет функциональность базового класса.

Ниже иллюстрируется класс WorkItem, представляющий рабочий элемент в бизнес-процессе. Подобно всем классам, он является производным от System.Object и наследует все его методы. В WorkItem имеется пять собственных членов. Сюда входит конструктор, поскольку конструкторы не наследуются. Класс ChangeRequest наследуется от WorkItem и представляет конкретный вид рабочего элемента. ChangeRequest добавляет еще два члена к членам, унаследованным от WorkItem и Object. Он должен добавить собственный конструктор, и он также добавляет originalItemID. Свойство originalItemID позволяет связать экземпляр ChangeRequest с исходным объектом WorkItem, к которому применен запрос на изменение.



Лекция 7

Абстрактные и виртуальные методы

Когда базовый класс объявляет метод как виртуальный, производный класс может переопределить метод с помощью своей собственной реализации. Если базовый класс объявляет член как абстрактный, то этот метод должен быть переопределен в любом неабстрактном классе, который прямо наследует от этого класса. Если производный класс сам является абстрактным, то он наследует абстрактные члены, не реализуя их. Абстрактные и виртуальные члены являются основой для полиморфизма, который является второй основной характеристикой объектно-ориентированного программирования.

Абстрактные базовые классы

Можно объявить класс как абстрактный, если необходимо предотвратить прямое создание экземпляров с помощью ключевого слова new. При таком подходе класс можно использовать, только если новый класс является производным от него. Абстрактный класс может содержать один или несколько сигнатур методов, которые сами объявлены в качестве абстрактных. Эти сигнатуры задают параметры и возвращают значение, но не имеют реализации (тела метода). Абстрактному классу необязательно содержать абстрактные члены; однако, если класс все же содержит абстрактный член, то сам класс должен быть объявлен в качестве абстрактного. Производные классы, которые сами не являются абстрактными, должны предоставить реализацию для любых абстрактных методов из абстрактного базового класса.

Виртуальные члены

Если производный класс наследуется из базового, он получает все методы, поля, свойства и события базового класса. Разработчик производного класса может выбрать следующее:

- наследование метода ближайшего базового класса без переопределения;
- переопределение виртуальных членов в базовом классе;
- определение новой, невиртуальной реализации тех членов, которые скрывают реализации базового класса.

Производный класс может переопределить член базового класса, только если последний будет объявлен виртуальным или абстрактным. Производный член должен использовать ключевое слово переопределить, указывающее, что метод предназначен для участия в виртуальном вызове. Примером является следующий код:

```
public class BaseClass
{
   public virtual void DoWork() { }
   public virtual int WorkProperty
   {
      get { return 0; }
   }
}
public class DerivedClass : BaseClass
{
   public void DoWork() { }
   public override int WorkProperty
   {
      get { return 0; }
   }
}
DerivedClass B = new DerivedClass();
B.DoWork();
BaseClass A = (BaseClass)B;
A.DoWork();
```

Виртуальными могут быть только методы, свойства, события и индексаторы. Когда производный класс переопределяет виртуальный член, он вызывается даже в том случае, если доступ к экземпляру этого класса осуществляется в качестве экземпляра базового класса. Примером является следующий код:

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Also calls the new method.
```

Виртуальные методы и свойства позволяют производным классам расширять базовый класс без необходимости использовать реализацию метода базового класса. Для получения дополнительной информации см. Практическое руководство. Управление версиями с помощью ключевых слов "Override" и "New" . Еще одну возможность определения метода или набора методов, реализация которых оставлена производным классам, дает интерфейс (но об этом позднее).

Сокрытие членов базового класса новыми членами

Если вам нужно, чтобы производный член имел такое же имя, как и член в базовом классе, но вы не хотите, чтобы он участвовал в виртуальном вызове, используйте ключевое слово new новое. Ключевое слово new вставляется перед типом возвращаемого значения замещаемого члена

класса. Примером является следующий код:

```
public class BaseClass
{
   public void DoWork() { WorkField++; }
   public int WorkField;
   public int WorkProperty
   {
      get { return 0; }
   }
}

public class DerivedClass : BaseClass
{
   public new void DoWork() { WorkField++; }
   public new int WorkField;
   public new int WorkProperty
   {
      get { return 0; }
   }
}
```

Доступ к скрытым членам базового класса можно по-прежнему осуществлять из клиентского кода приведением экземпляра производного класса к экземпляру базового класса. Например:

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Calls the old method.
```

Защита виртуальных членов от переопределения производными классами

Виртуальные члены остаются виртуальными на неограниченный срок независимо от количества классов, объявленных между виртуальным членом и классом, который объявил его изначально. Если класс A объявляет виртуальный член, класс B производится из класса A, а класс C — из класса B, то класс C наследует виртуальный член и получает возможность переопределить его независимо от того, объявляет ли класс B переопределение этого члена. Примером является следующий код:

```
public class A
{
   public virtual void DoWork() { }
}
public class B : A
{
   public override void DoWork() { }
}
```

Производный класс может остановить виртуальное наследование, объявив переопределение как запечатанное. Для этого в объявление члена класса необходимо вставить ключевое слово sealed перед ключевым словом override. Примером является следующий код:

```
public class C : B
{
   public sealed override void DoWork() { }
}
```

В предыдущем примере метод DoWork более не является виртуальным ни для одного класса, произведенного из класса C. Он по-прежнему виртуален для экземпляров класса C, даже если они приводятся к типу B или типу A. Запечатанные методы можно заменить производными классами с помощью ключевого слова new, как показано в следующем примере.

```
public class D : C
{
   public new void DoWork() { }
}
```

В этом случае, если DoWork вызывается на D с помощью переменной типа D, вызывается новый DoWork. Если переменная типа C, B или A используется для доступа к экземпляру D, вызов DoWork будет выполняться по правилам виртуального наследования и направлять эти вызовы на реализацию DoWork на классе C.

Доступ к виртуальным членам базового класса из производных классов

Производный класс, который заменил или переопределил метод или свойство, может получить доступ к методу или свойству на базовом классе с помощью ключевого слова base. Примером является следующий код:

```
public class BaseClass
{
   public virtual void DoWork() {/*...*/ }
}
public class Derived : BaseClass
{
   public override void DoWork()
   {
      //Perform Derived's work here
      //...
      // Call DoWork on base class
      base.DoWork();
   }
}
```

Управление версиями с помощью ключевых слов "Override" и "New"

Функции языка C# позволяют обеспечить и поддерживать обратную совместимость за счет управления версиями между базовыми и производными классами в различных библиотеках. Это означает, например, что если в базовом классе будет создан новый член, имя которого совпадает с именем члена в производном классе, язык C# обработает такую ситуацию, и она не приведет к непредвиденным результатам. Это также означает, что в классе должно быть явно указано, будет ли метод переопределять наследуемый метод, или это новый метод, который будет скрывать наследуемый метод с тем же именем.

В C# производный класс может включать методы с теми же именами, что и у методов базового класса.

- Метод базового класса может быть определен как виртуальный.
- Если перед методом в производном классе не указано ключевое слово new или override, компилятор выдаст предупреждение, и обработка метода будет производиться как в случае наличия ключевого слова new.
- Если перед методом в производном классе указано ключевое слово new, то этот метод определен как независимый от метода в базовом классе.
- Если перед методом в производном классе указано ключевое слово override, то объекты производного класса будут вызывать этот метод вместо метода базового класса.
- Базовый метод можно вызвать из производного класса с помощью ключевого слова base.
- Ключевые слова override, virtual и new могут также применяться к свойствам, индексам и событиям.

По умолчанию методы в языке C# не являются виртуальными. Если метод

объявлен как виртуальный, то любой класс, наследующий этот метод, может реализовать собственную версию. Чтобы сделать метод виртуальным, в объявлении метода базового класса используется модификатор virtual. Производный класс может переопределить базовый виртуальный метод с помощью ключевого слова override или скрыть виртуальный метод в базовом классе с помощью ключевого слова new. Если ключевые слова override и new не указаны, компилятор выдаст предупреждение, и метод в производном классе будет скрывать метод в базовом классе.

Чтобы продемонстрировать это на практике, предположим, что компания А создала класс с именем GraphicsClass, используемый вашей программой. Это класс GraphicsClass:

```
class GraphicsClass
{
   public virtual void DrawLine() { }
   public virtual void DrawPoint() { }
}
```

Ваша компания использует этот класс, и вы применили его для создания собственного производного класса, добавив новый метод:

```
class YourDerivedGraphicsClass : GraphicsClass
{
   public void DrawRectangle() { }
}
```

Ваше приложение работало без проблем до тех пор, пока компания А не выпустила новую версию класса GraphicsClass со следующим кодом:

```
class GraphicsClass
{
   public virtual void DrawLine() { }
   public virtual void DrawPoint() { }
   public virtual void DrawRectangle() { }
}
```

Новая версия класса GraphicsClass теперь имеет метод с именем DrawRectangle. Сначала ничего особенного не произошло. Новая версия совместима со старой на уровне машинных кодов. Все развернутое программное обеспечение будет продолжать работать, даже если в компьютерных системах будет установлен новый класс. Все существующие вызовы метода DrawRectangle будут продолжать ссылаться на вашу версию в созданном вами производном классе.

Однако сразу после перекомпиляции приложения с помощью новой версии класса GraphicsClass компилятор выдаст предупреждение CS0108. В этом предупреждении вам будет предложено принять решение относительно работы метода DrawRectangle в вашем приложении.

Если ваш метод должен переопределить новый метод базового класса,

используйте ключевое слово override:

```
class YourDerivedGraphicsClass : GraphicsClass
{
   public override void DrawRectangle() { }
}
```

Ключевое слово override заставляет все объекты, являющиеся производными от YourDerivedGraphicsClass, использовать версию DrawRectangle производного класса. Объекты, являющиеся производными от класса YourDerivedGraphicsClass, могут продолжать использовать версию метода DrawRectangle базового класса, используя ключевое слово "base":

```
base.DrawRectangle();
```

Если ваш метод не должен переопределять новый метод базового класса, следуйте приведенным ниже рекомендациям. Чтобы избежать путаницы между двумя методами, вы можете переименовать свой метод. Это отнимает много времени и может привести к возникновению ошибок, поэтому не всегда удобно. Однако, если ваш проект невелик, можно использовать функции оптимизации кода Visual Studio для переименования метода.

Чтобы избежать появления предупреждения, можно также использовать ключевое слово new в определении производного класса:

```
class YourDerivedGraphicsClass : GraphicsClass
{
   public new void DrawRectangle() { }
}
```

Использование ключевого слова new сообщает компилятору, что ваше определение скрывает определение, содержащееся в базовом классе. Это поведение установлено по умолчанию.

Переопределение и выбор метода

При вызове метода класса компилятор C# выбирает наилучший из методов, совместимых с вызовом (например, если имеются два метода с одинаковыми именами), и параметров, которые совместимы с переданным параметром. Следующие методы будут совместимы:

```
class Derived : Base
{
   public override void DoWork(int param) { }
   public void DoWork(double param) { }
}
```

Если выполняется вызов DoWork из экземпляра Derived, компилятор C# сначала попытается обеспечить совместимость вызова с версиями DoWork, первоначально объявленными в Derived. Переназначенные методы не рассматриваются как объявленные в классе. Они являются новыми реализациями метода, объявленного в базовом классе. Только в

том случае, если компилятор C# не смог сопоставить вызов метода с исходным методом в классе Derived, он попытается сопоставить вызов с переназначенным методом с тем же именем и совместимыми параметрами. Например:

```
int val = 5;
Derived d = new Derived();
d.DoWork(val);  // Calls DoWork(double).
```

Поскольку переменную val можно неявно преобразовать в тип "double", компилятор C# вызовет DoWork(double) вместо DoWork(int).Чтобы избежать этого, существует два способа. Во-первых, избегайте объявления новых методов, имена которых совпадают с виртуальными методами. Во-вторых, можно сделать так, чтобы компилятор C# вызвал виртуальный метод, заставив его выполнить поиск в списке методов базового класса за счет приведения экземпляра Derived к Base. Поскольку метод является виртуальным, будет вызвана реализация DoWork(int) в классе Derived. Например:

```
((Base)d).DoWork(val);  // Calls DoWork(int) on Derived.
```

Переопределение метода ToString

Каждый класс или структура в C# неявно наследует класс Object. Поэтому каждый объект в языке C# получает метод ToString, который возвращает строковое представление данного объекта. Например, все переменные типа int имеют метод ToString, который позволяет им возвращать содержимое этой переменной в виде строки:

```
int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

При создании пользовательского класса или структуры необходимо переопределить метод ToString, чтобы передать информацию о типе клиентскому коду.

Дополнительные сведения об использовании строк форматирования и другие типы пользовательского форматирования с методом ToString см. в разделе Типы форматирования в .NET Framework.

При принятии решения относительно того, какая информация должна будет предоставляться посредством этого метода, подумайте, будет ли создаваемый класс или структура когда-либо использоваться ненадежным кодом. Постарайтесь не предоставлять информацию, которая может быть использована вредоносным кодом.

Переопределение метода ToString в классе или структуре

1) Объявите метод ToString со следующими модификаторами и типом возвращаемого значения:

```
public override string ToString(){}
```

2) Реализуйте этот метод таким образом, чтобы он возвращал строку.

В приведенном ниже примере возвращается не только имя класса, но и специфические данные для конкретного экземпляра класса.

```
class Person
{
   public string Name { get; set; }
   public int Age { get; set; }

   public override string ToString()
   {
      return "Person: " + Name + " " + Age;
   }
}
```

Метод ToString можно проверить с помощью показанного ниже кода.

```
Person person = new Person { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

Лекция 8

Исключения и обработка исключений

Функции обработки исключений на языке C# помогают обрабатывать любые непредвиденные или исключительные ситуации, происходящие при выполнении программы. При обработке исключений используются ключевые слова try, catch и finally для попыток применения действий, которые могут не достичь успеха, для обработки ошибок, если предполагается, что это может быть разумным, и для последующего освобождения ресурсов. Исключения могут генерироваться средой CLR, платформой .NET Framework или внешними библиотеками, либо кодом приложения. Исключения создаются при помощи ключевого слова throw.

Во многих случаях исключение может инициироваться не методом, вызванным непосредственно кодом, а другим методом, расположенным ниже в стеке вызовов. Когда это происходит, среда CLR выполняет откат стека в поисках метода с блоком catch для определенного типа исключения. При обнаружении первого такого блока catch этот блок выполняется. Если среда CLR не находит соответствующего блока catch где-либо в стеке вызовов, она завершает процесс и отображает пользователю сообщение.

В этом примере метод тестирует деление на ноль и выполняет перехват

соответствующей ошибки. Без обработки исключений эта программа была бы завершена с ошибкой DivideByZeroException was unhandled (не обработано исключение "деление на ноль").

```
class ExceptionTest
{
   static double SafeDivision(double x, double y)
   {
      if (y == 0)
         throw new System.DivideByZeroException();
      return x / y;
   }
   static void Main()
   {
      // Input for test purposes. Change the values to see
      // exception handling behavior.
      double a = 98, b = 0;
      double result = 0;

      try
      {
         result = SafeDivision(a, b);
         Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
      }
      catch (DivideByZeroException e)
      {
         Console.WriteLine("Attempted divide by zero.");
      }
   }
}
```

Исключения имеют следующие свойства.

- Исключения имеют типы, в конечном счете являющиеся производными от System.Exception.
- Следует использовать блок try для заключения в него инструкций, которые могут выдать исключения.
- При возникновении исключения в блоке try поток управления немедленно переходит к первому соответствующему обработчику исключений, присутствующему в стеке вызовов. В языке C# ключевое слово catch используется для определения обработчика исключений.
- Если обработчик для определенного исключения не существует, выполнение программы завершается с сообщением об ошибке.

- Не перехватывайте исключение, если его нельзя обработать, и оставьте приложение в известном состоянии. При перехвате System.Exception вновь инициируйте это исключение с использованием ключевого слова throw в конце блока catch.
- Если в блоке catch определяется переменная исключения, ее можно использовать для получения дополнительной информации о типе произошедшего исключения.
- Исключения могут явно генерироваться программой с помощью ключевого слова throw.
- Объекты исключения содержат подробные сведения об ошибке, такие как состояние стека вызовов и текстовое описание ошибки.
- Код в блоке finally выполняется, даже при возникновении исключения. Блок finally используется для освобождения ресурсов, например для закрытия потоков или файлов, открытых в блоке try.
- Управляемые исключения в платформе .NET Framework реализуются в начале механизма структурированной обработки исключений Win32.

Лекция 9

4.6.7. Интерфейсы

Интерфейс содержит определения для группы определенного функционала, который класс или структура могут реализовывать

С помощью интерфейсов можно, например, включить поведение из нескольких источников в классе. Эта возможность важна в C#, поскольку язык не поддерживает множественное наследование классов. Кроме того, необходимо использовать интерфейс, если требуется имитировать наследование для структур, поскольку они фактически не могут наследоваться из других структур или классов.

Определяется интерфейс с помощью ключевого слова интерфейс (interface), как показано в следующем примере.

```
interface IEquatable<T>
{
   bool Equals(T obj);
}
```

Любой класс или структура, реализующие интерфейс IEquatable<T>, должны содержать определение метода Equals, который соответствует сигнатуре, которую определяет интерфейс. В результате можно рассчитывать на то, что класс, реализующий интерфейс IEquatable<T>, будет содержать метод Equals, с которым экземпляр класса может определить, является ли он равным другому экземпляру класса.

Определение IEquatable<T> не предоставляет реализацию для метода Equals. Интерфейс определяет только сигнатуру. В этом смысле интерфейс в C# аналогичен абстрактному классу, в котором все методы являются абстрактными. Однако класс или структура может реализовывать несколько интерфейсов, но класс может наследовать только от одного класса, абстрактного или нет. Таким образом, с помощью интерфейсов можно включить в класс поведение из нескольких источников.

Интерфейсы могут содержать методы, свойства, события, индексаторы, а также любое сочетание этих четырех типов членов. Ссылки на примеры см. в разделе Связанные разделы. **Интерфейс не может содержать константы, поля, операторы, конструкторы экземпляров, деструкторы или типы.** Члены интерфейса автоматически открыты, и они не могут включать иные модификаторы доступа. Члены также не могут быть статический.

Для реализации члена интерфейса соответствующий член класса должен быть открытым, нестатическим, и иметь то же имя и сигнатуру, что и член интерфейса.

Когда класс или структура реализуют интерфейс, класс или структура должны обеспечивать реализацию всех членов которые определяет интерфейс. Сам по себе интерфейс не предоставляет никакой функциональности, которую бы могли наследовать класс или структура,

как это происходит в случае наследования от базового класса. Однако если базовый класс реализует интерфейс, то любой класс, производный от базового класса, наследует эту реализацию.

В следующем примере показана реализация интерфейса IEquatable<T>. Реализующий класс Car должен предоставлять реализацию метода Equals.

```
public class Car : IEquatable<Car>
{
   public string Make {get; set;}
   public string Model { get; set; }
   public string Year { get; set; }

   // Implementation of IEquatable<T> interface
   public bool Equals(Car car)
   {
      if (this.Make == car.Make &&
         this.Model == car.Model &&
         this.Year == car.Year)
      {
         return true;
      }
      else
         return false;
   }
}
```

Свойства и индексаторы класса могут определять дополнительные методы доступа для свойства или индексатора, определенного в интерфейсе. Например, интерфейс может объявлять свойство, имеющее метод доступа get. Класс, реализующий интерфейс, может объявлять это же свойство с методами доступа get и set. Однако если свойство или индексатор использует явную реализацию, методы доступа должны совпадать. Дополнительные сведения о явной реализации см. в разделах Явная реализация интерфейса (Руководство по программированию в C#) и Свойства интерфейса (Руководство по программированию на C#).

Интерфейсы могут реализовывать другие интерфейсы. Класс может включать интерфейс несколько раз через базовые классы, которые он наследует, или через интерфейсы, которые реализуют другие интерфейсы. Однако класс может предоставить реализацию интерфейса только однократно и только если класс объявляет интерфейс как часть определения класса (class ClassName : InterfaceName). Если интерфейс наследуется, поскольку наследуется базовый класс, реализующий этот интерфейс, то базовый класс предоставляет реализацию членов этого интерфейса. Однако производный класс может повторно реализовать члены интерфейса вместо использования унаследованной реализации.

Базовый класс также может реализовывать члены интерфейса с помощью виртуальных членов. В таком случае производный класс может изменять поведение интерфейса путем переопределения виртуальных членов. Дополнительные сведения о виртуальных членах см. в разделе Полиморфизм.

Интерфейс имеет следующие свойства:

- Интерфейс похож на абстрактный базовый класс. Любой класс (или структура), реализующий интерфейс, должен реализовывать все его члены.
- Невозможно создать экземпляр интерфейса напрямую. Его члены реализованы всеми классами или структурами, реализующими интерфейс.
- Интерфейсы могут содержать события, индексаторы, методы и свойства.
- Интерфейсы не содержат реализацию методов.
- Класс или структура может реализовывать несколько интерфейсов. Класс может наследовать базовому классу и также реализовывать один или несколько интерфейсов.

4.6.6. Явная реализация интерфейса

Если класс реализует два интерфейса, содержащих член с одинаковой сигнатурой, то при реализации этого члена в классе оба интерфейса будут использовать этот член для своей реализации. В следующем примере все вызовы Paint вызывают один метод.

```
class Test
{
   static void Main()
   {
      SampleClass sc = new SampleClass();
      IControl ctrl = (IControl)sc;
      ISurface srfc = (ISurface)sc;

      // The following lines all call the same method.
      sc.Paint();
      ctrl.Paint();
      srfc.Paint();
   }
}


interface IControl
{
   void Paint();
}
interface ISurface
{
   void Paint();
}
class SampleClass : IControl, ISurface
{
   // Both ISurface.Paint and IControl.Paint call this method.
   public void Paint()
   {
      Console.WriteLine("Paint method in SampleClass");
   }
}

// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

Однако, если члены двух интерфейсов не выполняют одинаковую функцию, это может привести к неверной реализации одного или обоих

интерфейсов. Возможна явная реализация члена интерфейса — путем создания члена класса, который вызывается только через интерфейс и имеет отношение только к этому интерфейсу. Это достигается путем включения в имя члена класса имени интерфейса с точкой. Например:

```
public class SampleClass : IControl, ISurface
{
   void IControl.Paint()
   {
      System.Console.WriteLine("IControl.Paint");
   }
   void ISurface.Paint()
   {
      System.Console.WriteLine("ISurface.Paint");
   }
}
```

Член класса IControl.Paint доступен только через интерфейс IControl, а член ISurface.Paint — только через интерфейс ISurface. Каждая реализация метода является независимой и недоступна в классе напрямую. Например:

```
// Call the Paint methods from Main.

SampleClass obj = new SampleClass();
//obj.Paint();  // Compiler error.

IControl c = (IControl)obj;
c.Paint();  // Calls IControl.Paint on SampleClass.

ISurface s = (ISurface)obj;
s.Paint(); // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

Явная реализация также используется для разрешения случаев, когда каждый из двух интерфейсов объявляет разные члены с одинаковым именем, например свойство и метод.

```
interface ILeft
{
   int P { get;}
}
interface IRight
{
   int P();
}
```

Для реализации обоих интерфейсов классу необходимо использовать явную реализацию либо для свойства P, либо для метода P, либо для обоих членов, чтобы избежать ошибки компилятора. Например:

```
class Middle : ILeft, IRight
{
   public int P() { return 0; }
   int ILeft.P { get { return 0; } }
}
```

Лекция 10

Делегаты

Делегат (delegate) – это тип, который представляет собой ссылки на методы с определенным списком параметров и возвращаемым типом. При создании экземпляра делегата этот экземпляр можно связать с любым методом с совместимой сигнатурой и возвращаемым типом. Метод можно вызвать (активировать) с помощью экземпляра делегата.

Делегаты используются для передачи методов в качестве аргументов к другим методам. Обработчики событий — это не что иное, как методы, вызываемые с помощью делегатов. Вы создаёте свой метод, а класс, такой как элемент управления Windows, может вызывать ваш метод при возникновении определенного события. В следующем примере показано объявление делегата:

```
public delegate int PerformCalculation(int x, int y);
```

Делегату может быть присвоен любой метод, соответствующий типу делегата, из любого доступного класса или структуры. Этот метод должен быть статическим методом или методом экземпляра. Это позволяет программно изменять вызовы метода, а также включать новый код в существующие классы.

В контексте перегрузки метода его сигнатура не содержит возвращаемое значение. Однако в контексте делегатов, сигнатура содержит возвращаемое значение. Другими словами, метод должен иметь тот же возвращаемый тип, что и делегат.

Благодаря возможности ссылаться на метод как на параметр, делегаты оптимально подходят для задания функций обратного вызова. Например,

ссылка на метод, сравнивающий два объекта, может быть передана в качестве аргумента алгоритму сортировки. Поскольку код сравнения находится в отдельной процедуре, алгоритм сортировки может быть написан в более обобщенном виде.

Делегаты имеют следующие свойства.

- Делегаты похожи на указатели функций в C++, но являются типобезопасными.
- Делегаты позволяют производить передачу методов подобно обычным параметрам.
- Делегаты можно использовать для определения методов обратного вызова.
- Делегаты можно связывать друг с другом; например, при появлении одного события можно вызывать несколько методов.
- Точное соответствие методов типу делегата не требуется.
- В C# версии 2.0 введена концепция анонимных методов, которые позволяют передавать блоки кода в виде параметров вместо использования отдельно определенного метода. В C# 3.0 для краткой записи встроенных блоков кода введены лямбда-выражения. В результате компиляции как анонимных методов, так и лямбда-выражений (в определенном контексте) получаются типы делегатов. В настоящее время эти возможности называются анонимными функциями.

Лабораторные работы

Занятие 1

Разработайте объектную модель системы «Абитуриент», предполагая, что основными ее классами являются следующие:

- «Абитуриент», свойствами которого являются ФИО, место жительства, номер школы, паспортные данные, выбранная специальность, результаты ЕГЭ и т.д;
- «Результат вступительных испытаний», определяющий баллы, которые получил конкретный абитуриент при сдаче экзамена по одному предмету (ЕГЭ или вступительные экзамены);
- «Направление подготовки»: код, название, план приема, проходной балл и т.д.

Занятие 2

Доработайте объектную модель системы «Абитуриент», добавив в нее следующие классы:

- «Институт»: код, название института, список направлений подготовки;
- «Приказ о зачислении», характеризующийся номером, датой и списком абитуриентов, зачисленных в студенты по различным направлениям подготовки;
- «Студент», основная информация о котором совпадает с атрибутами класса «Абитуриент», а также имеются новые свойства, характерные для других приложений.

Занятие 3

Пусть имеется описание класса комплексных чисел:

```
classcomplex{
doublere, im;
};
```

Напишите конструкторы, позволяющие создавать объект класса complex
а) по двум заданным параметрам;
б) по заданному значению re (при этом считать im=0);
в) без параметров.
Можно ли написать один конструктор, удовлетворяющий условиям а) –в) одновременно?

Занятие 4

Приведите описание класса «обыкновенная дробь», члены-данные класса–числитель и знаменатель. При создании объектов этого класса возможно задание значений числителя и знаменателя или только числителя (знаменатель в этом случае считается равным 1). При создании дроби должно проводиться сокращение, то есть дроби 1/2 и 6/12 должны стать одинаковыми. Нужно ли для создания копий объектов такого класса явно определять конструктор копирования?

Занятие 5

Разработайте класс для реализации методов решения квадратного уравнения $ax^2+bx+c=0$. Уравнение задается набором коэффициентов (от 1 до 3). Если при создании указывается иное количество коэффициентов, то квадратное уравнение определить нельзя, поэтому выдается предупреждение об ошибке. В классе должны быть предусмотрены средства для решения уравнений, в которых a=0, b=0 или c=0. Тогда уравнение может стать линейным ($0*x^2+5x+2=0$), обратиться в тождество (0 = 0) или стать неразрешимым (например, 6 = 0).

Занятие 6

Опишите реализацию абстрактного типа данных. Если для него требуются вспомогательные абстрактные типы данных, приведите их описание полностью:

- Почтовый адрес. Для объектов класса должны быть предусмотрены функции изменения индекса, города, улицы, номера дома, корпуса, квартиры, вывода адреса на экран.

Занятие 7

Опишите реализацию абстрактного типа данных. Если для него требуются вспомогательные абстрактные типы данных, приведите их описание полностью:

- Банковский счет. Необходимые данные: дата создания счета, сумма денег, которая на нем хранится, владелец счета (объект класса person, у которого есть фамилия и имя), информация о последних 10 операциях, проведенных со счетом. Операция, проводимая со счетом–объект, содержащий дату операции, вид операции(добавить/снять деньги) и сумму операции. У класса банковский счет должны быть методы: пополнить счет, снять деньги (если указана недопустимая сумма–должно печататься сообщение об ошибке, сумма на счете при этом не меняется), распечатать информацию о последних 10 операциях, распечатать доступную сумму денег на счете.

Занятие 8

Опишите реализацию абстрактного типа данных. Если для него требуются вспомогательные абстрактные типы данных, приведите их описание полностью:

- Зоопарк. Необходимые данные: количество зверей, массив, содержащий информацию о животных (имя, номер клетки, название любимой еды), часы работы зоопарка адрес зоопарка, фамилия сторожа. В классе нужно определить методы, позволяющие «накормить» зверя (изменить значение соответствующего поля), поменять сторожа зоопарка (записать другую фамилию сторожа в соответствующее поле).

Занятие 9

Пусть класс Door описан следующим образом:

```
classDoor{
intheight;
intwidth;
```

};
Приведите примеры операций, которые:
а) могут быть перегружены только с помощью функции–члена класса;
б) могут быть перегружены только с помощью функции, не являющейся членом класса;
с) могут быть перегружены как функцией–членом класса, так и функцией, не являющейся членом класса;
d) не могут быть перегружены.
Напишите соответствующие прототипы перегруженных операций для классаDoor.

Занятие 10

Опишите иерархию классов для хранения информации о сотрудниках, студентах и аспирантах университета. В качестве базового используйте класс Person.

Занятие 11

Определите иерархию классов для описания сессии в университете. Базовый класс–event, содержит информацию о дате события, фамилию действующего лица и виртуальную функцию print_res, печатающую информацию о событии; классы-наследники–test (зачет), exam (экзамен). В них должны быть определены дополнительные характеристики событий и метод print_res. В функции main определите сессию как массив указателей на события, проинициализируйте элементы массива и распечатайте информацию об экзаменах и зачетах.

Занятие 12

Разработайте обобщенный класс Set для хранения множества элементов некоторого типа. В методе Main(string [] args) продемонстрируйте применение обобщённого класса Set для хранения целых чисел, вещественных чисел и строк.

Средства контроля качества обучения

Вопросы к экзамену:

1. Основные элементы объектной модели.
2. Отношения между объектами и классами.
3. Абстрагирование.
4. Инкапсуляция.
5. Наследование.
6. Полиморфизм.
7. Особенности платформы .NET.
8. Классы с C#, поля класса.
9. Указание области видимости: public; private; protected; internal.
10. Статические поля.
11. Константы.
12. Методы класса: синтаксис; тип возвращаемого значения; список формальных параметров.
13. Модификаторы доступа к методам: public, private, protected, internal, static.
14. Методы с переменным числом параметров.
15. Определение выходных параметров метода.
16. Вызов метода.
17. Рекурсивные методы.
18. Конструкторы.
19. Деструкторы.
20. Перегрузка методов.
21. Перегрузка операторов.
22. Свойства класса.
23. Индексаторы класса.
24. Функциональные типы в C#, делегаты.
25. Функциональные типы в C#, события.
26. Наследование.
27. Виртуальные функции.
28. Абстрактные классы.
29. Создание иерархии исключений.
30. Обобщения, основные понятия.
31. Уточнения, используемые в обобщениях.
32. Обобщенные интерфейсы.
33. Обобщенные методы.
34. Обобщенные делегаты.