Широкое распространение программного обеспечения

- Современный мир не может существовать без программ для компьютеров.
  - все государственные службы и инфраструктуры управляются системами, основанными на компьютерах.
  - большинство приборов и оборудования включают вычислительные компоненты и управляющее ПО.
  - отрасли развлечений (включая музыкальную индустрию), компьютерных игр, фильмов и телевидения активно используют ПО.
- В связи с этим жизненно важной для национальных и международных сообществ является технология разработки ***программного обеспечения***

Сложность современного ПО

- Программные системы стали
  - очень сложными,
  - распределенными,
  - трудными для понимания,
  - дорогими для изменения.
- Существует много разных типов ПО
  - информационные системы организаций;
  - распределенные системы;
  - встроенные системы.

Проблема разработки ПО

- Повышение требований
  - Программные системы становятся все более крупными, более сложными (должны иметь новые возможности, которые ранее были не осуществимыми).
  - Должны разрабатываться и устанавливаться (развертываться) быстрее.
  - Должны быть более надежными.
  - Должны иметь возможность быстро исправляться и развиваться.

Профессиональная разработка ПО

- Множество людей пишут программы –
  - ученые, инженеры, преподаватели составляют программы для обработки своих данных;
  - для любителей программирование является «хобби» - тем, чем они занимаются в свое свободное время (для своего интереса и развлечения).
- Но значительное большинство ПО разработано ***профессионалами*** (обычно командами профессионалов), которые создают программы по заказу других людей (потребителей) для получения денег.
- Профессиональное ПО предназначено для использования другими людьми (не разработчиками).
- Оно поддерживается в рабочем состоянии (сопровождается), меняется и развивается в течении всего времени своего существования.

ПО промышленного уровня

- Профессионалы разрабатывают ПО промышленного уровня для решения некоторых задач клиентов, которые используют их для целей своего бизнеса.
- Такое ПО требует поддерживать его в течении длительного времени.
- Для профессионального ПО требуется высокий уровень качества.
- Программная индустрия в значительной степени заинтересована в разработке *ПО промышленного уровня*.
- Далее в данном учебном курсе под термином ПО (software) понимается ПО промышленного уровня.

Стоимость, время и качество

- При разработке ПО промышленного уровня наиболее важными являются следующие три показателя:
  1. стоимость разработки;
  2. сроки разработки (время);
  3. качество программного обеспечения.
- ПО должно быть разработано
  - за разумные деньги,
  - в разумные сроки и
  - иметь хорошее качество.
- ПО промышленного уровня является очень дорогим в основном в связи с тем, что разработка такого ПО является очень трудоемким делом.
- Основными затратами на создание ПО является стоимость рабочей силы (зарплата специалистов).
- Чаще всего затраты на разработку ПО измеряются в терминах затраченных на это человеко-месяцев.

История развития методологии разработки ПО

1. Начальный этап развития программирования.
2. Этап хакеров.
3. Структурный подход к разработке ПО.
4. ОО подход к разработке ПО.

Начальный этап развития программирования

- В 50-е годы фактически не было систематизированного подхода к разработке ПО.
- Программирование для больших ЭВМ (мэйнфреймов), имеющих ограниченные ресурсы
  - несколько десятков килобайт оперативной памяти
  - в качестве устройства данных – устройство чтения с бумажной ленты.
  - в качестве устройств управления и ввода данных использовались телетайпы, которые требовали большие затраты энергии для каждого нажатия клавиши.
- Отсутствие отладчиков и дисплеев.

- **Ассемблерный язык рассматривался как решение кризиса разработки ПО.**
- В конце 50-х и начале 60-х годов начали появляться такие компиляторы высокоуровневых алгоритмических языков (FORTRAN, COBOL, BASIC).
- Алгоритмические языки абстрагировали 1-ы и 0-и в символьные имена, высокоуровневые операторы, блочные структуры и абстрагировали такие структуры, как массивы и записи.
- Это привело к появлению Этапа Хакеров, в которой основной была индивидуальная производительность программистов.

Этап хакеров

- Этап хакеров продолжалась с начала 60-х до середины 70-х годов.
- Очень способные люди могли производить огромное количество кода (до 100 KLOC/год на языке FORTRAN).
  - они должны быть очень способными, так как тратили огромное количество времени на отладку и
  - они должны быть очень упорными, чтобы получить результирующий код, который работает очень быстро.
- Они часто разрабатывали «красивые» решения проблем.
- *В 60-х годах определение (термин) “хакер” было положительным (хвалебным).*
- Хакер это программист, который мог создать большое количество программного кода, упорно работал и мог поддерживать работоспособность этого кода.
- Отсутствует организованный подход к созданию ПО.

- Однако к концу 70-х годов определение “хакер” стал отрицательным.
- Причина: хакеры переходили к разработке новых проектов и оставляли свой код для поддержки другим специалистами.
- Однако созданные хакерами программы требовали изменений:
  - выявлялась специальные ситуации, которые не обрабатывались программами;
  - мир менялся и программы тоже должны были совершенствоваться.
- Оказалось, что во многих случаях программы легче переписать, чем исправить.
- Появился термин «***поддерживаемый код»***, который можно было просто менять
- Не поддерживаемый (unmaintainable) код стал называться хакерским кодом.

Структурный подход к разработке ПО (СП)

Структурное подходы к разработке ПО

(Structured Development, SD)

- В конце 60-х годов появился более систематизированный подход к разработке ПО.
- К тому времени увеличились размеры программ и появилась идея, что они должны предварительно **проектироваться**.
- Начали появляться методологии, которые объединяли полученный опыт разработки ПО.
- **Проектирование ПО** стало видом деятельности отличной от **составления программ** (программирования).

Структурная разработка

- Структурный подход к разработке ПО бесспорно является наиболее важным достижением в технологии разработки ПО до 80-х годов.
- Он предоставил первый действительно систематизированный подход к разработке ПО.
- Данный подход совместно с языками программирования 3 поколения (3GLs) способствовал
  - значительному повышению производительности разработки ПО и
  - повышению надежности ПО (от 150 до 15 ошибок на KLOC к 1980 году).

Характеристики структурного подхода

1. Графическое представление разрабатываемого ПО.
2. Использование «**функциональной декомпозиции»** (функциональная изолированности):
   – Основная идея структурного подхода заключается в том, что п*рограмма состоит из большого количества алгоритмов разной сложности, которые совместно работают для решения имеющейся проблемы*.
3. Прикладной программный интерфейс (Application Programming Interfaces, API).
4. Программирование на основе контрактов – соглашений (programming by contract).

Функциональная декомпозиция (ФД)

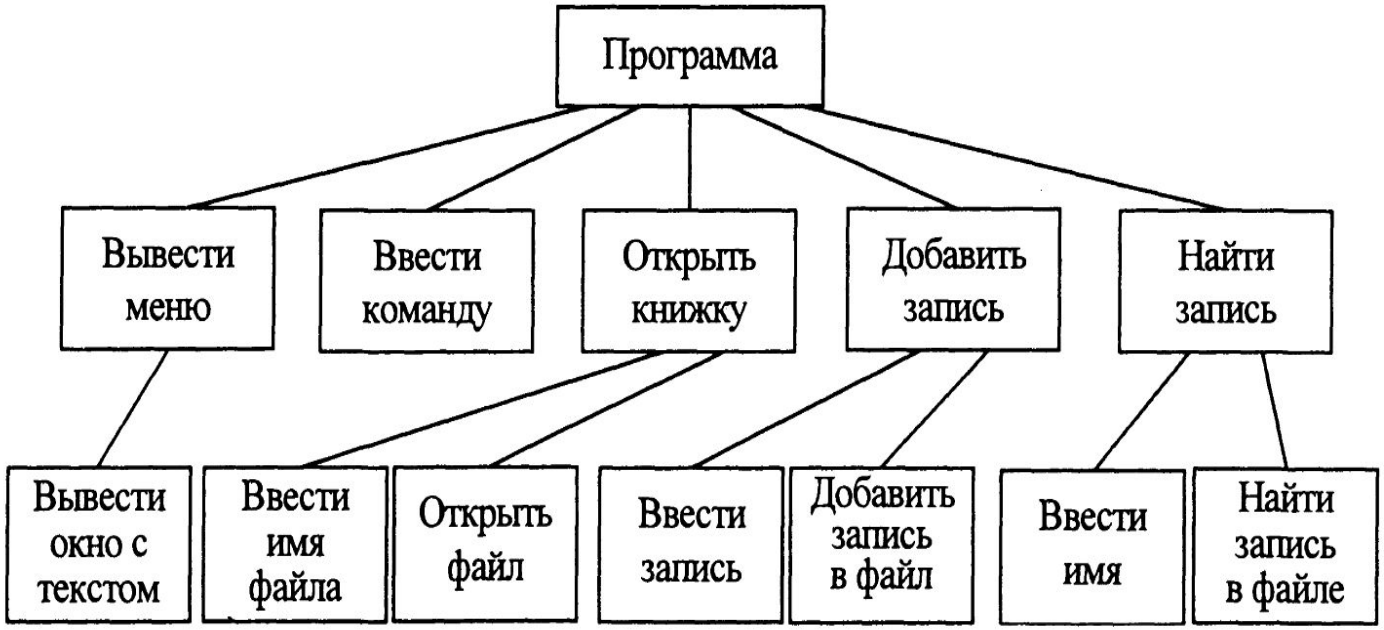
- Это был базовый метод проектирования, который использовался в каждом структурном подходе.
- *ФД рассматривает разработку программного решения исключительно, как разработку алгоритма.*
- Такой представление о программе намного более подходит к разработке программ для научных задач, чем, например, к разработке программ для информационных систем.

Функциональная декомпозиция (2)

- Базовым принципом ФД является принцип «разделяй и властвуй».
- В результате получается древовидная структура, в которой функции верхнего уровня, расположенные в верхней части, вызывают множество функций нижнего уровня, содержащих подразделенную функциональность.
- Листья (конечные вершины), расположенные в основе такого дерева, являются ***атомарными функциями*** (простейшими) (на уровне арифметических операторов), т.к. они настолько простые (базовые), что не могут подразделяться далее.

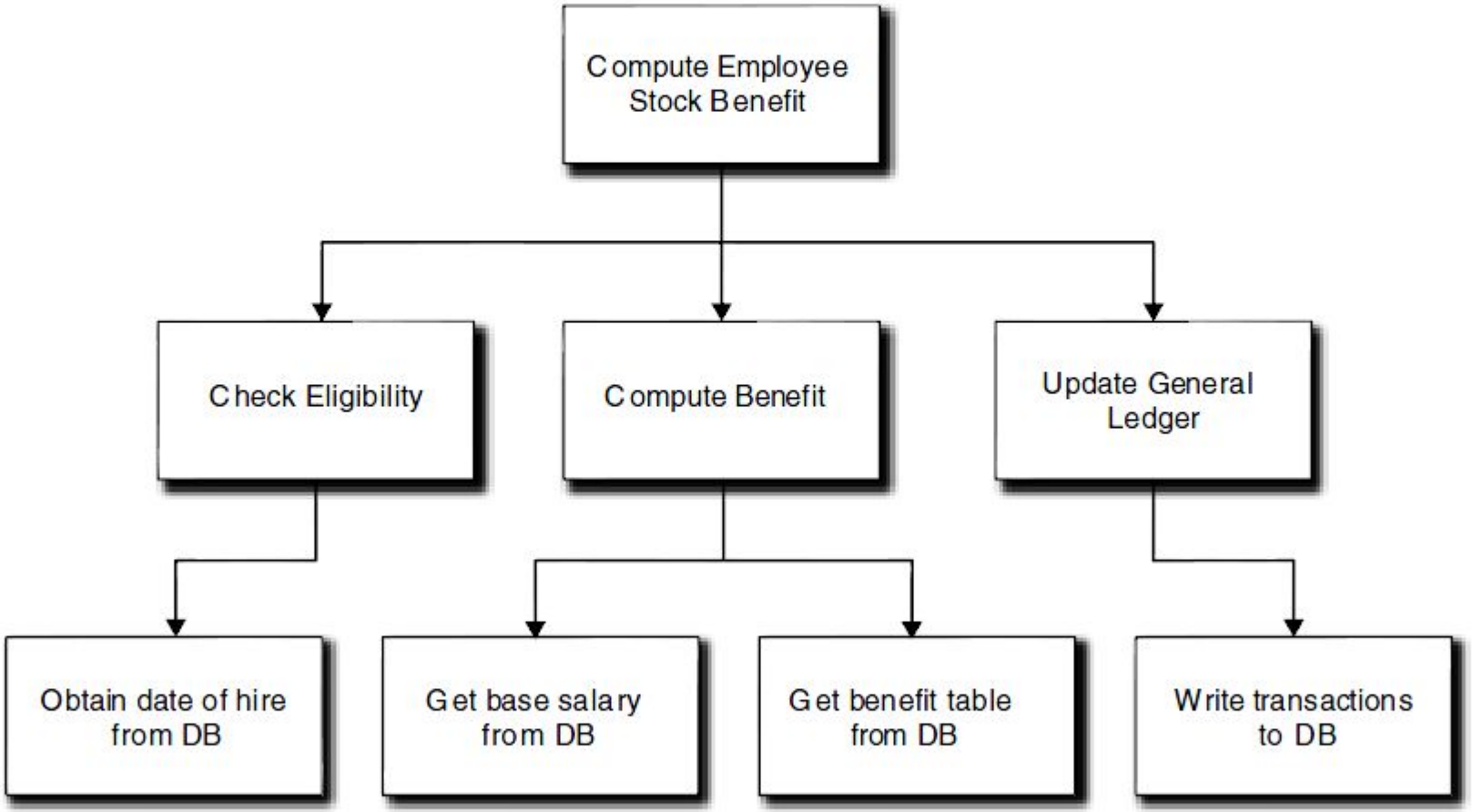
Функциональная декомпозиция

- Функциональная декомпозиция системы «Записная книжка»



Функциональная декомпозиция (3)

- Пример функциональной декомпозиции задач вычисления выплат сотрудникам от их доли акций компании на более управляемые части.



Достоинства функциональной декомпозиции

- Функциональная декомпозиция была мощной и привлекательной.
- Она была привлекательной, так как объединяла, такие понятия, как:
  - **функциональная изоляция** (например, ветви дерева и детали разделенных функций),
  - **программирование на основе контрактов** (programming by contract) – разделенные функции предоставляют сервисы для клиентов более высокого уровня,
  - **план выполнения нисходящей разработки**, после описания дерева декомпозиции функций,
  - **прикладные программные интерфейсы (API)** в форме сигнатур процедур;
  - **укрупненные средства навигации преимущественного в глубину** (depth-first navigation) для анализ потока управления;
  - **многократное использование** путем вызова одних и тех же базовых функций из разных мест программы (из разных контекстов).

- В общем, это было очень умный способ решения

Недостатки функциональной декомпозиции

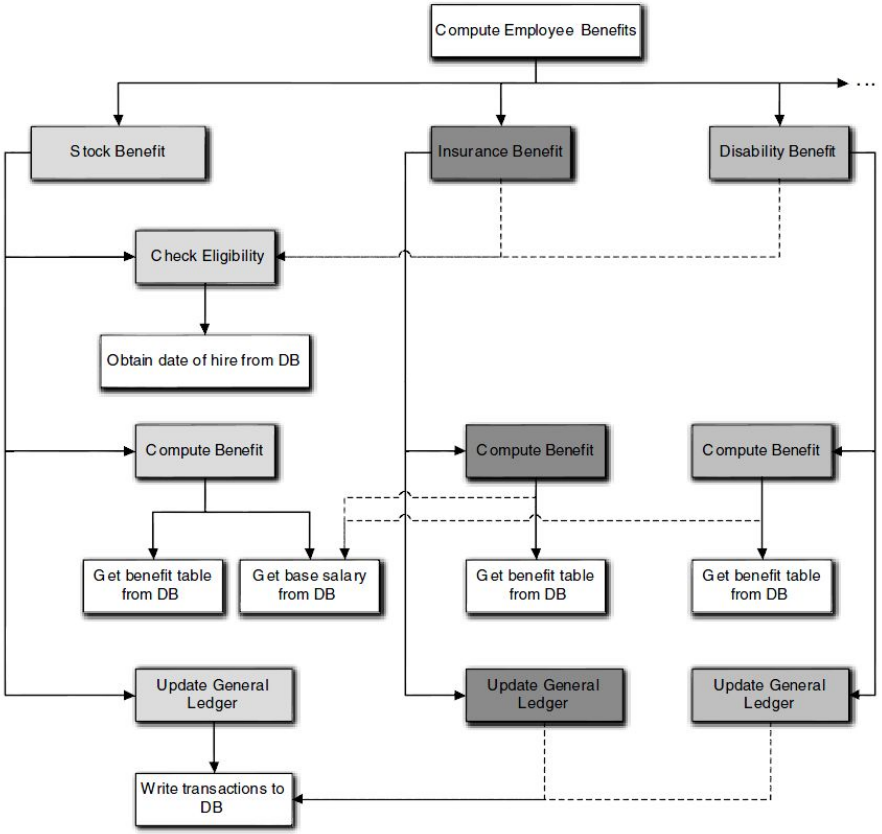
- Однако, в конце 70-х годов стало ясно, что SD привел к появлению нового набора проблем, которые никто не ожидал.
- Алгоритмы научных расчетов обычно меняются редко или меняются полностью с использованием более совершенных алгоритмов.
- ***В программировании бизнес-приложений правила, на основе которых работают программы постоянно меняются и программа постоянно развивается.***
- В связи с этим приложения должны модифицироваться в течение всей их жизни, иногда даже в течении начальной их разработки.

Трудность изменения иерархических структур

- Однако иерархические структуры функциональной декомпозиции оказались трудными для изменения.
- Изменение функций верхнего уровня вызывало изменения большого числа функций более низкого уровня.
- Другой проблемой была избыточность.
  - Набор простейших функций достаточно ограничен, в связи с этим в разных функциях требовалось использовать сходные простейшие функции.
  - Часто один и тот же набор простейших функций был в разных ветвях иерархии.
  - Повышалась трудоемкость их кодирования.
  - Одинаковые изменения требовалось делать в разных однотипных функциях, что повышало возможность ошибки.

Возможное улучшение функциональной декомпозиции

- Дерево функциональной декомпозиции стало решеткой.
- Закрашенные задачи являются базовыми элементами для расчета разных доплат (benefits).
- Пунктирные линии показывают перекрестные ссылки из одной ветви декомпозиции к другой, для устранения избыточности.



Проблемы структурного подхода

- Наибольшей трудностью для ФД было неявное знание контекста, которое появляется при выполнения обработки вглубь (depth-first processing).
- Высоко-уровневые функции являются просто набором инструкций (указаний) для низко-уровневых функций что-то сделать.
- Чтобы задать инструкцию что-то сделать, высокоуровневая функция должна
  - знать, кто будет это делать;
  - знать что он может это сделать;
  - знать что это следующая работа, которая должна быть выполнена в общем решении.
- Все это нарушало изоляцию функций
- Последнее требование указывает, что вызываемая функция должна понимать высокоуровневый контекст общего решения проблемы.

Объектно-ориентированны подход к разработке ПО (ООПх)

Объектно-ориентированные подходы к разработке ПО

- Данный период начался в начале 80-х годов и оказал влияние на почти все виды деятельности в области разработки ПО.
- ООП, более конкретно дисциплинированный подход к анализу и проектированию ПО, был только одним из множества инноваций.
- Прошло около десяти лет, пока ООА и ООПр развились до такого состояния, что стали универсальными.

Базовая философия ОО подхода

- ОО подход является достаточно сложным по сравнению с ранее предложенным структурным подходом к разработке ПО.
- ***Он не очень интуитивно понятный в смысле выполнения вычислений*** (по сравнению с декомпозицией функций), ***поэтому он требует специального образа мышления***.
- Данный подход также включает набор различных понятий, которые должны использоваться совместно.

Проблема сопровождения ПО

- В 70-х годах было проведено много статистических исследований того, как разные компании выполняют разработку ПО.
- В результате были полученные очень интересные результаты:
  - **большинство компаний тратили 70% своих усилий на сопровождение ранее созданного ПО;**
  - **работа по модификации существующего ПО часто требует в 5-10 раз больше усилий, чем первоначальная его разработка.**
- Стало понятно, что-то делалось неправильно в разработке ПО.

- Специалисты по ПО пришли к выводу, что не возможно исправить (улучшить) структурный подход.
- Стало понятно, что ООП является тем подходом, который выведет из проблем удобства сопровождения.
- ***Основной целью ООА/ООПр стало удобство сопровождения ПО.***

Удобство поддержки ПО

- Основной целью ОО подхода является удобство сопровождения ПО в связи постоянным изменением требований.
- Удобство сопровождения улучшится если структура ПО будет имитировать инфраструктуру проблемного пространства.

Результат использования ООПх

- ***Оказалось, что программы разработанные с использованием ООП более удобными для сопровождения (поддержки).***
- В начале 80-х годов в ОО методологии начали уделять основное внимание тому, как ООА/ООПр могут использоваться для решения недостатков СП.

- ***Показатель производительности первоначального создания ПО является менее важным, по сравнению с производительностью сопровождения***.
- Другой очень приоритетной целью разработки ПО является его ***надежность*** .
- Одной из основных причин того, что обычная поддержка ПО является трудоемкой, связано с тем, что изменения стали настолько сложными, что они вносили новые дефекты.

- ***Все в современном ООА/ООПр нацелено на разработку более понятных, легко поддерживаемых и надежных приложений.***
- Базовым допущением ОО подхода является -- ПО постоянно меняются в течение жизни программного продукта.
- Если это не выполняется. (программирование научных задач), то ОО подход может быть лучшим использовать

Основные виды деятельности в объектно-ориентированном подходе

- Основными видами работ в объектно-ориентированном подходе (ООП) являются:
  1. Объектно-ориентированный анализ (OOA),
  2. Объектно-ориентированное проектирование (ООПр, OOD),
  3. **Объектно-ориентированное программирование** (ООП, OOP)

Требований потребителей к ПО

- ***Функциональные*** - какие задачи ПО должно решать
- ***Не функциональные*** - как должно работать ПО
  - безопасность
  - производительность (время отклика)
  - соответствие стандартам
  - поддерживаемые протоколы взаимодействия

Объектно-ориентированный анализ

- *Результатом выполнения ООА является создание* ***решения для функциональных требований проблемы****, не зависящее от конкретной вычислительной среды.*
- ООА описывает представление потребителя о разрабатываемом решении (приложении), т.к. абстракция проблемного пространства (ПП) описывается в терминах структуры ПрО потребителя.
- Результат ООА описывается только в терминах потребителя, поэтому ***он может включать только функциональные требования***.
- Решение полученное в ООА не зависит от конкретной вычислительной среды, в которой данное приложение будет реализовано

Объектно-ориентированное проектирование

- *ООПр заключается в заключается детализации (доработке) решения полученного в результате выполнения ООА,* ***для определения решения для не функциональных требований на стратегическом уровне для конкретной вычислительной среды.***
- В результате ООПр создается высокоуровневое описание проекта решения, приспособленного к базовым характеристикам используемой вычислительной среды.

Объектно-ориентированное программирование (ООПм)

- ООПм это ***детальное уточнение решения, полученного на этапе ООПр***, которое предоставляет тактическое решение для всех требований с использованием уровня ОО языка программирования.
- ООПм предоставляет решение для всех требований (функциональных и не функциональных) на тактическом уровне (например, конкретного языка, сетевых протоколов, библиотеки классов, технологий и т.п.).

Последовательность получения решения

- Последовательность формирования решения:

Требования ->

Объектно-ориентированный анализ ->

Объектно-ориентированное проектирование ->

Объектно-ориентированное программирование ->

Компиляция (в макро-язык) ->

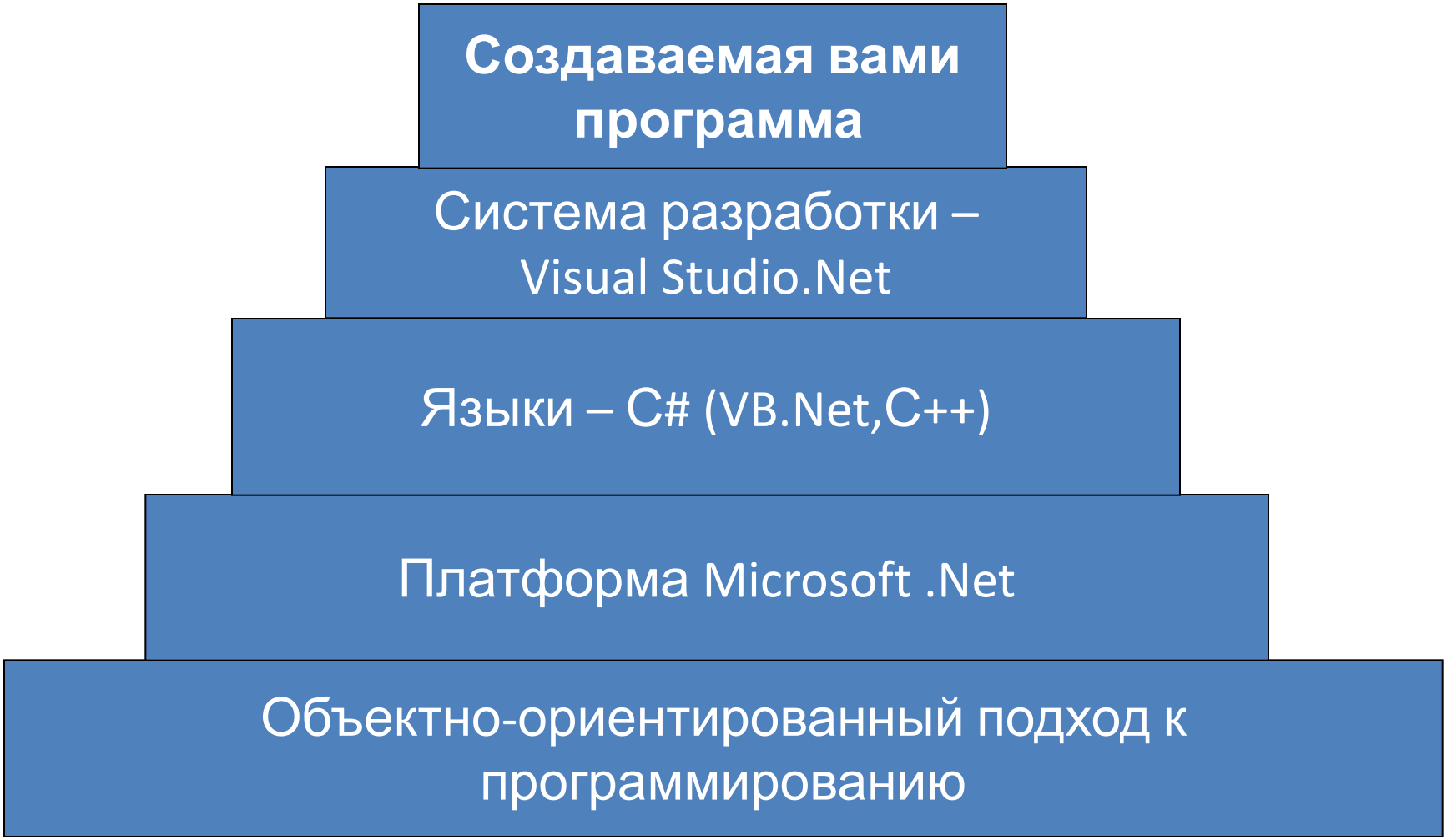
Исполняемый код

- При перемещении слева на право
  - уменьшается уровень абстракции и
  - увеличивается учет деталей компьютерной

Цель дисциплины ОО программирование

- Дать основы ОО подхода к разработке программ в среде OS Windows.
- Изучение новой технологии разработки ПО .Net, как наиболее совершенной в среде ОС Windows.
- Изучить программирование на новом алгоритмическом языке C#.
- Дать навыки работы с интегрированной средой разработки Visual Studio.Net.

Пирамида требуемых знаний



Виды программных систем

Локальные прикладные программы (приложение -application)

- Консольное приложение
- Приложение с графическим интерфейсом

Распределенные программные системы в локальной компьютерной сети (distributed application)

Web приложения (Web application, с использованием Интернет обозревателя)

Что такое ООП?

- Объектно-ориентированное программирование это подход к разработке программ в виде совокупности объектов, являющихся экземплярами определенного типа (класса), которые взаимодействуют между собой путем обмена сообщениями.
- ООП это специальные языки и технологии, которые позволяют быстро разрабатывать объектно-ориентированные программы

Основные средства достижения цели

- Современная методология разработки ПО - Object Oriented Programming, включает
  - Анализ
  - Проектирование
  - Программирование
- Современная технология разработки (.Net technology, component technology, языки программирования)
- Современные системы разработки (Visual Studio, CBuilder, Delphy)

Основные технологии разработки ПО

1. Язык Java (фирма Sun) и основанные на нем технологии (JavaBeabs, NetBeans, Eclipse, JDBC, JSP, …)

2. Платформа .Net (языки C#, Visual Basic, C++ и др., технологии ADO, ASP, LINQ, …)

История развития ООП

- Основные понятия ООП (объекты, классы, свойства, методы) появились в середине 1960-х годов в ходе разработки языка программирования Simula. Данный язык использовался для создания имитационных программ.
- Далее понятия ООП были развиты в 70-годы в связи с созданием языка Smalltalk.
- Хотя не все разработчики ПО сразу поняли и начали использовать ООП, объектно-ориентированные методологии продолжали развиваться.

История развития ООП

- В середине 80-х годов - всплеск интереса к ОО методологиям.
- Появились новые ОО языки программирования, которые стали популярными при программировании на больших компьютерах
  - Eiffel (1986)
  - C++ (1985)
- Популярность ООП продолжала расти в 90-е годы
  - в начале 90-х годов создан язык программирования Oak;
  - в 1996 году был создан язык программирования Java.
- В 2002 (вместе с новой платформы разработки Microsoft .NET Framework, были созданы
  - новый ОО язык программирования C#
  - доработан язык Visual, так чтобы он стал действительно ОО языком.

Введение в Объектно-Ориентированного Программирования

- ***Объект*** это программная конструкция, содержащая набор логически связанных данных и методов.

- Объекты являются автономными элементами, они предоставляют некоторую функциональность другим компонентам среду приложения, изолируя от них свое внутренне устройство.

- Объекты создаются на основе шаблонов, называемых ***классами***.
  - библиотека базовых классов .Net
  - можно создавать собственные классы

Объекты

- В отличие от процедурного программирования в объектно-ориентированном программировании (ООП) основными элементами программы являются не переменные и методы (процедуры), а объекты.
- ***Объекты*** – это программные конструкции, включающие набор логически связанных свойств (данных) и методов.
  - Объекты являются автономными сущностями, они предоставляют некоторую функциональность другим компонентам среды приложения, изолируя от них свое внутреннее устройство.
  - Объекты создаются на основе шаблонов, которые называются ***классами*** и являются экземплярами этих классов.

Объекты

- В среде разработки имеются наборы уже созданных классов, которые можно применять для создания объектов в разрабатываемых приложениях.
- В приложениях можно создавать собственные классы, требуемые для описания решаемой задачи.
- Например, класс Автомобилей:

```
class Автомобиль
{
   // описание данных
   // описание методов
}
```

Классы

- ***Классы*** *– это «шаблоны» (чертежи) объектов*.
- Классы определяют все элементы объекта:
  - свойства и его поведение (методы),
  - задают начальные значения для создаваемых объектов, если это необходимо.
- При создании экземпляра класса в памяти создается копия этого класса.

- *Экземпляр класса – это* ***объект****.*
- Экземпляр класса создается с помощью операции ***new.***
- Например:

```
// Объявление переменной типа Автомобиль
Автомобиль myAuto; // переменная это не объект!
// Создание экземпляра класса Автомобиль
// и сохранение ссылки на него в переменной myAuto
myAuto = new Автомобиль ();
```

Создание экземпляра класса

- При создании экземпляра класса, выделяется блок оперативной памяти, в который записывается копия данных.
- Адрес этого блока присваивается переменной (myAuto) которая хранит эту ссылку.
- Экземпляры класса не зависят друг от друга и являются отдельными программными конструкциями.
- Разрешается создавать произвольное число копий класса, которые могут существовать одновременно.

- Класс – это чертеж по которому делаются объекты:
  - если конкретный автомобиль – это объектом,
  - то чертежи автомобиля – это класс Автомобиль.
- По чертежу можно сделаете сколько угодно автомобилей.
- Если один из автомобилей будет работать не так, как все, это никак не повлияет на остальные.

Содержание объектов

- Объекты состоят из элементов, к которым относят ***поля***, ***свойства***, ***методы*** и ***события***, представляющие данные и функциональность объекта.
  - ***поля*** содержат данные объекта,
  - ***свойства*** – предоставляют управляемый способ доступа к данным объекта,
  - ***методы*** – определяют действия, которые объект способен выполнять, ***методы*** позволяют реализовать поведение объекта.
  - ***события*** уведомляют заинтересованных пользователей (другие класса, которые подпишутся на эти события), если в объекте происходит что-то важное.

- Например, объекты класса Автомобиль
  - ***Свойства*** описывающими состояние объекта, например, такие как Цвет, Модель, Расход_топлива и т.д.
  - ***Методы*** этих объектов описывают функциональность автомобиля, например:
    - Нажать_акселератор,
    - Переключить_передачу или
    - Повернуть_руль.
  - ***События*** представляют собой уведомления о важных происшествиях, например
    - количество бензина стало ниже заданной величины
    - событие Перегрев_двигателя.

Доступ к элементам объектов

- Для доступа к полям, свойствам, методам элементам объектов используется специальная операция точка “.”.
- Например:

  myAuto.Модель // определение Модели

  myAuto.Повернуть_руль() // поворот руля автомобиля
- Однако к элементам объекта имеется доступ не из всех частей программы (из методов данного класса или других классов, из других сборок).
- Возможность использования элементов класса задается (явно или неявно) с помощью указания режима доступа.
- Например, таким режимом может быть
  - private – означающий, что элемент, у которого он задан, может использоваться только в методах того класса, где он описан (закрытые элементы),
  - public – означающий, что данный элемент можно

Отношения между классами

- Классы разных объектов не являются изолированными друг от друга. Как и в реальном мире, они связаны между собой.
- Выделяются два основных типа взаимосвязи:
  - ***вложенность*** – это включение объектов одного класса в объекты другого класса.
  - ***наследование*** – это описание одного класса на основе другого класса.

Вложенность объектов

- Объекты одних классов могут включать объекты других классов в качестве своих полей
  - предоставлять к ним доступ, как и к другим своим элементам.
- Иерархия вложенности объектов друг в друга называется объектной моделью (object model).
  - В случае с автомобилем, объект класса Автомобиль, который сам по себе является объектом, состоит из ряда вложенных объектов,
    - таких как объект класса Двигатель, четырех объектов класса Колесо, объект класса Трансмиссия и т.д.
  - Компоновка вложенных объектов непосредственно определяет работу объекта класса Автомобиль.
    - Например, поведение объектов Автомобиль, у которых свойство Число_цилиндров вложенного объекта Двигатель равно соответственно 4 и 8, будет различным.

- У вложенных объектов могут быть собственные вложенные объекты.
  - Например, объект Двигатель (который является вложенным объектом объекта Автомобиль) может иметь несколько вложенных объектов Свеча_зажигания.
  - Тогда для получения сведений о марке свечей зажигания автомобиля нужно записать:

```
myAuto.Свеча_зажигания.Марка;
```

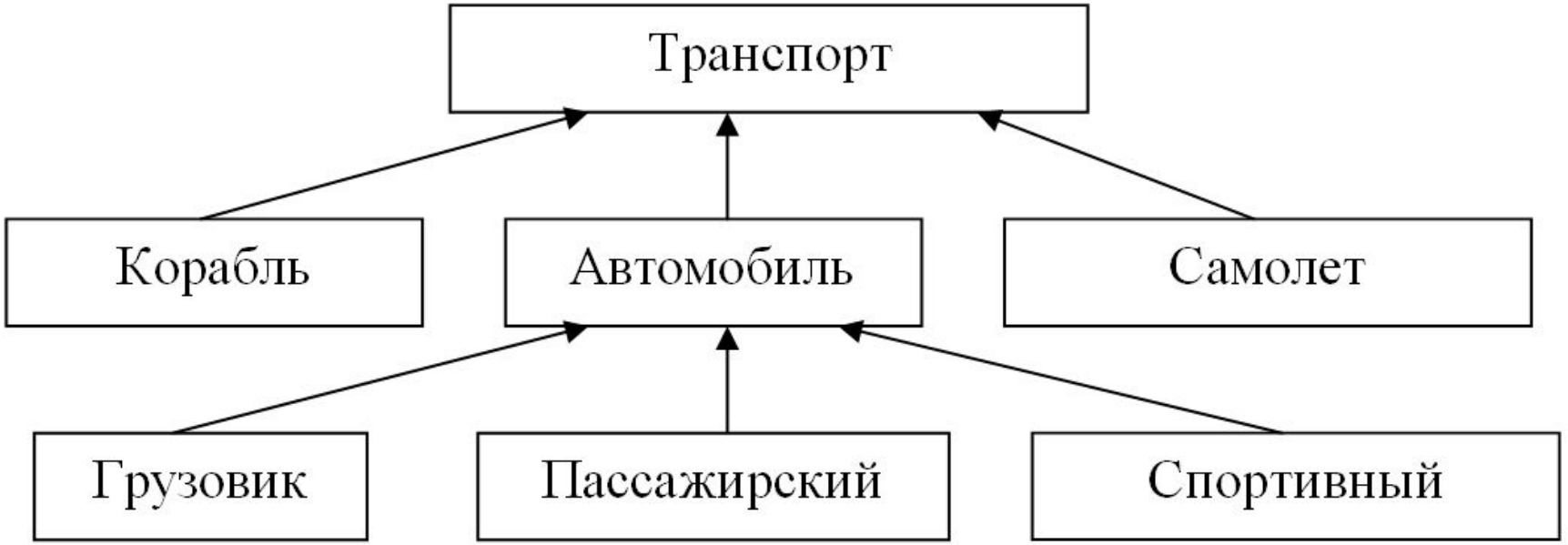
Наследование классов

- Один класс может быть описан на основе уже имеющегося описания другого класса.
  - В этом случае между классами задается отношение *наследования*.
- Наследование позволяет создавать новые классы на основе существующих, при этом новые классы обладают всей функциональностью старых и при необходимости могут модифицировать их.
- Класс, объявленный на основе некоторого (базового) класса, называется производным или классом-потомком.

Наследование классов

- У любого класса может быть только один прямой предок – его базовый класс (base class).
- У производного класса окажется тот же набор элементов, что и у базового, но, при необходимости, к производному классу разрешается добавлять дополнительные элементы.
- Можно также изменить реализацию членов, унаследованную от базового класса, переопределив их.

Схема наследования классов.



- Например, на основе описания класса Транспортное_средство (базовый класс) можно описать класс Автомобиль (производный класс)
- На основе класс Автомобиль можно описать классы другие производные классы:
  - Грузовик, Пассажирский_автомобиль и
  - Спортивный_автомобиль.

Пример описания производного класса Автомобиль

```
class Автомобиль : Транспорт
{
    // описание свойств
      public string model;
      public float Расход_топлива;
      private int Число_цилиндров;
    // описание методов
      public void Повернуть_руль(){...};
      private Регулировка_датчика(){...};
    // описание события
      public event Перегрев_двигателя();
}
```

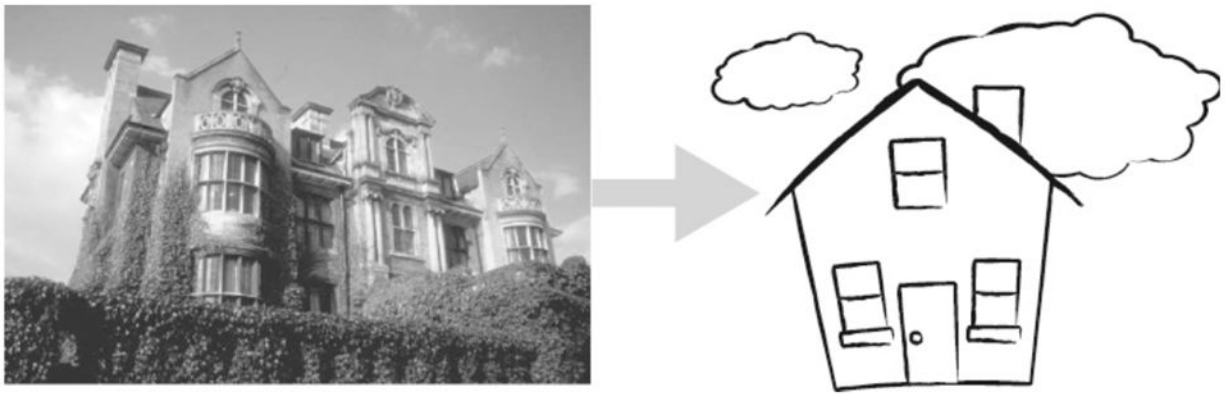
- В производном классе сохраняется функциональность (поведение), определенная в базовом классе.
- Кроме того, в производных классах могут описываться новые элементы и переопределяться методы (поведение), описанные в базовых классах.
- Для указания возможности доступа к наследуемым элементам (помимо public) также используется специальный режим доступа protected.

Фундаментальные принципы ООП

- Абстрагирование – способность описывать основные особенности и функциональность объектов реального мира.
- Инкапсуляция – управление правами доступа к элементам объекта;
- Агрегирование – возможность включать в один класс объекты из других классов.
- Наследование – возможность создавать новые классы на основе уже созданных классов.
- Полиморфизм – возможность однотипно работать с объектами разных классов.

Абстрагирование

- Объект – это программная конструкция, представляющая некоторую сущность окружающего мира.
- Например, в повседневной жизни сущностями, или объектами можно считать:
  - автомобили,
  - велосипеды,
  - настольные компьютеры,
  - банковский счет.
- Каждый объект обладает определенной функциональностью и свойствами.

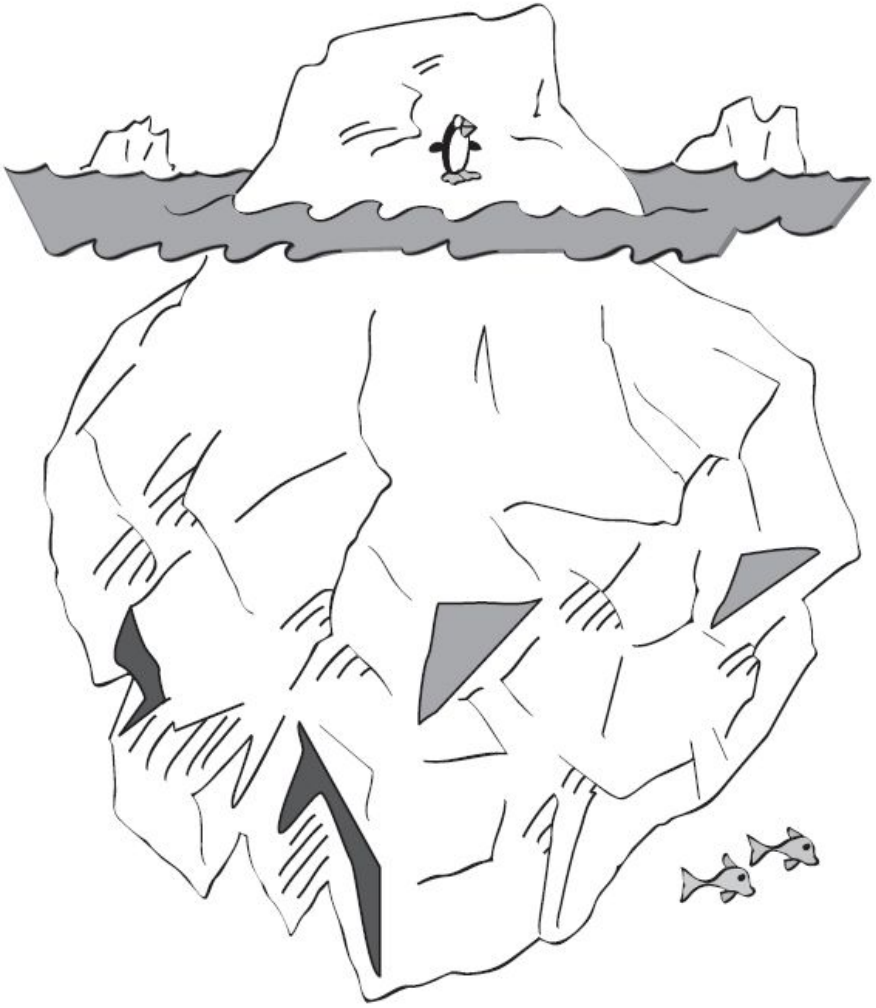


- Объект представляет собой завершенную функциональную единицу, содержащую все данные и предоставляющую всю функциональность, необходимую для решения задачи, для которой он предназначен.

- Описание объектов реального мира при помощи программных объектов называют ***абстрагированием*** (abstraction).

Инкапсуляция

- Смысл инкапсуляции состоит в отделении реализации объекта (его внутреннего содержания) от способа взаимодействия с ним.
- Другие объекты приложения взаимодействует с рассматриваемым объектом посредством имеющихся у него открытых (public) свойств и методов, которые составляют его ***интерфейс***.
- В общем виде под интерфейсом понимается открытый способ взаимодействия между разными системами.
- Интерфейс ни в коем случае не должен открывать доступ к внутренним данным объекта, поэтому поля с внутренними данными объекта обычно объявляют с модификатором private.



- Если интерфейс класса не будет меняться, то приложение сохраняет способность к взаимодействию с его объектами, даже если в новой версии класса его реализация значительно изменится.
- Объекты могут взаимодействовать друг с другом только через свои открытые методы и свойства, поэтому объект должен предоставлять доступ только к тем свойствам и методам, которые пользователям необходимы.

- Например у объектов класса Автомобиль, которые могут взаимодействовать с объектами класса Водитель через открытый интерфейс, открытыми объявлены только методы Ехать_вперед, Ехать_назад, Повернуть и Остановиться – их достаточно для взаимодействия объектов классов Водитель и Автомобиль.
- У объекта класса Автомобиль может быть вложенный объект класса Двигатель, но будет закрыт для объектов класса *Водитель*, которому будут открыты лишь методы, требуемые для управления автомобилем.
- В этом случае можно заменить вложенный объект класса Двигатель, и взаимодействующий с ним объект класса Водитель не заметит замены, если она не нарушит корректную работу интерфейса.

Полиморфизм

- ***Полиморфизм*** – многообразие форм.
- Благодаря полиморфизму, одни и те же открытые интерфейсы удается по-разному реализовать в разных классах.
- Полиморфизм позволяет вызывать методы и свойства объекта независимо от их реализации.

Пример полиморфизма

- Объект класса Водитель взаимодействует с объектом класса Автомобиль через открытый интерфейс.
- Если другой объект, например Грузовик или Гоночный_автомобиль, поддерживает такой открытый интерфейс, то объект класса Водитель сможет взаимодействовать и с ними (управлять ими), несмотря на различия в реализации их интерфейса.

Основных подходов к реализации полиморфизма

- через интерфейсы;
- через наследование.

Реализация полиморфизма с помощью интерфейсов

- Интерфейс (interface) – это соглашение, определяющее набор открытых методов, реализованных классом.
- Интерфейс определяет список методов класса, но ничего не говорит об их реализации.
- В объекте допустимо реализовать несколько интерфейсов, а один и тот же интерфейс можно реализовать в разных классах.
- Например, можно описать интерфейс возможности управления некоторыми объектами:

```
// имена интерфейсов обычно начинаться с буквы I
interface IDrivable {
  int Ехать(...);
  float Повернуть(...);
  bool Остановиться(...);
}
```

- Если класс реализует какой-то интерфейс, то в нем должны быть описаны все методы этого интерфейса. Например:

```
class Automobile : Транспорт, IDrivable
{
  int Ехать(...) {<реализация метода>};
  float Повернуть(...){<реализация метода>};
  bool Остановиться(...){<реализация метода>};
// описание других элементов ...
}
```

- Любые объекты, в которых реализован некоторый интерфейс, способны взаимодействовать друг с другом с его помощью.

- Интерфейс IDrivable также можно реализовать и в других классах, например, таких, как Грузовик, Автопогрузчик или Катер.
- В результате, эти объекты получат возможность взаимодействия с объектом класса Водитель.
- Объект класса Водитель находится в полном неведении относительно реализации интерфейса, с которым он взаимодействует, ему известен лишь сам

Реализация полиморфизма через наследование

- Производные классы сохраняют все характеристики своих базовых классов и способны взаимодействовать с другими объектами, под видом экземпляров базового класса!
- Т.е., переменным базового типа можно присваивать ссылки на объекты производных классов.
- Например:

```
Автомобиль myAuto; // переменная это не объект класса!
Спортивный_автомобиль sportAuto =
                    new Спортивный_автомобиль();
// можно присвоить, так как есть наследование
myAuto = sportAuto;
```

- В этом случае можно выполнять работу с объектом производного класса, как если бы он был объектом базового класса.

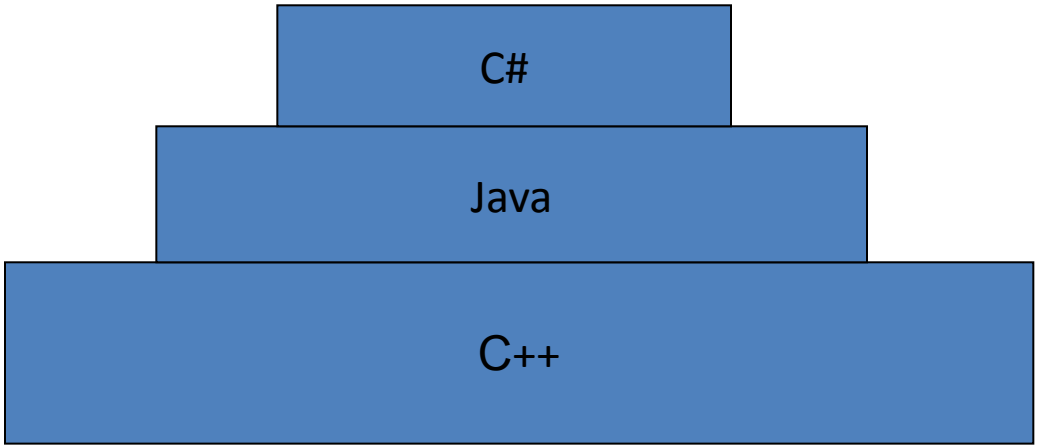
Новый язык программирования C#

Общие сведения по языку C#

- Появился в 2001 году.
- Основан на языках Java и Visual Basic
- Общий прародитель C++
- В первой версии языка:
  - 80 ключевых слов
  - 12 встроенных (базовых) типов данных
- Включает все необходимое для создания объектно-ориентированных, компонентных программ.
- Одобрен в качестве международного стандарта ECMA (**ECMA-334**) и ISO(**ISO/IEC 23270**)

Версии языка

- Версия C# 1.0 – 2001г. (для Framework 1.0)
- Версия C# 2.0 – 2005г. (для Framework 2.0)
- Версия C# 3.0 – 2007г. (для Framework 3.5)
- Версия C# 4.0 – 2010г. (для Framework 4.0)
- Версия C# 4.5 – 2012г. (для Framework 4.5)



Андерс Хейлсберг
(Anders Hejlsberg)

- Главный проектировщик и ведущий архитектор.
- Датский учёный в области информатики.
- В 1980 году он написал компилятор языка Паскаль, который продал фирме Borland (этот компилятор дожил до 7 версии (*Borland Pascal*).
- До 1996 года главный проектировщик фирмы Borland, создал новое поколение компиляторов Паскаля: получился язык Delphi.
- В 1996 году перешёл в Microsoft, где работал над языком J++ и библиотекой C++ - Windows Foundation Classes.
- Позже возглавил комиссию по созданию и проектированию языка C#.

Классы платформы Microsoft.Net

- ***Классы это основные пользовательские типы данных***.
- Экземпляры класса – Объекты.
- Классы описывают все элементы объекта (данные) и его поведение (методы), также устанавливают начальные значения для данных объекта, если это необходимо.
- При создании экземпляра класса в памяти создается копия данных этого класса. Созданный таким образом экземпляр класса называется объектом.

- Экземпляры классов создается с помощью оператора ***new***.
- Для получения данных объекта или вызова методов объекта, используется оператор “**.**” (точка).

```
Student s;
s = new Student();
s.Name = “Иванов А.”;
```

- При создании экземпляра класса, копия данных, описываемых этим классом, записывается в память и присваивается переменной ссылочного типа

- В этом случае описание класса Автомобиль может выглядеть следующим образом:

```
class Автомобиль
{
    // описание свойств
      public string Модель;
      public float Расход_топлива;
      private int Число_цилиндров;
    // описание методов
      public void Повернуть_руль(){...};
      private Регулировка_датчика(){...};
    // описание события
      event Перегрев_двигателя();
}
```

Описание классов программы

```
using XXX;   // чужие пространства имен
namespace MMM   // свое пространство имен
{
    class AAA // наш класс  MMM.AAA
    {
    …
    }
    class BBB //  другой наш класс MMM.BBB
    {
    …
    }
}
```

Пример класса

- Описание класса Автомобиль может выглядеть следующим образом:

```
class Автомобиль
{
  // описание свойств
  public string Модель;
  public float Расход_топлива;
  private int Число_цилиндров;
  // описание методов
  public void Повернуть_руль(){...};
  private Регулировка_датчика(){...};
  // описание события
  event Перегрев_двигателя();
}
```

- Для доступа к полям, свойствам, методам элементам объектов используется специальная операция точка (.). Например:

```
myAuto.Модель // определение Модели
myAuto.Повернуть_руль() // поворот руля автомобиля
```

Пример описания и использования класса

•Самый простой класс

```
class Car
{
}
```

•Класс с полями

```
class Car
{
   // состояние Car.
   public string petName;
   public int currSpeed;
}
```

•Класс с методами

```
class Car
{
  // состояние Car.
  public string petName;
  public int currSpeed;
  // функциональность Car.
  public void PrintState()
  {
    Console.WriteLine("{0} is going {1}
    MPH.",
       petName, currSpeed);
  }
  public void SpeedUp(int delta)
  {
    currSpeed += delta;
  }
}
```

```
static void Main(string[] args)
{
  Console.WriteLine
      ("***Используем Class Types***");
  // Создаем и настраиваем объект Car.
  Car myCar = new Car();
  myCar.petName = "Henry";
  myCar.currSpeed = 10;
  // Ускоряем автомобиль несколько раз
  // и выводим на печать новоесостояние.
  for (int i = 0; i <= 10; i++)
  {
    myCar.SpeedUp(5);
    myCar.PrintState();
  }
  Console.ReadLine();
}
```

• Неправильное использование класса

```
static void Main(string[] args)
{
  // Ошибка! Забыли использовать операцию 'new'!
  Car myCar;
  myCar.petName = "Fred";
}
```

Раздельное описание классов (partial classes)

- Класс может быть определен по частям в нескольких файлах исходного кода (единицах компиляции) – это удобно для пошаговой разработки программ и быстрого внесения изменений

```
// исходный файл 1
partial class C {
  public void M() { … }
}

// исходный файл 2
partial class C {
  public void N() { … }
}
```

- Множество элементов пошагово определенного класса является объединением всех множеств его элементов, определенных в отдельных исходных файлах

Инкапсуляция (***Encapsulation***)

- Максимальное закрытие доступа к состоянию объектов.
- Состояние объекта можно менять только используя свойства и методы (не используя переменные).
- Свойства и методы открытые внешним пользователям класса - интерфейс (interface).
- Это позволяет
  - Избежать неправильного использования объектов (защита от дурака)
  - Изменять и развивать класс не мешая его использовать

Инкапсуляция

- Отделение реализации объекта от его интерфейса. Приложение взаимодействует с объектом через его интерфейс, который состоит из открытых свойств и методов.
- Если интерфейс не меняется, то приложение может взаимодействовать с объектом, даже если его реализация будет сильно меняться.
- Объекты должны взаимодействовать друг с другом только через свои открытые методы и свойства, поэтому объекту требуются все необходимые данные и полный набор методов, необходимых для работы с этими данными.
- Интерфейс ни в коем случае не должен открывать доступ к внутренним данным объекта.
- Интерфейсы могут описываться отдельно от классов.
- Для того, чтобы указать, что класс поддерживает некоторый интерфейс, необходимо сделать класс производным от одного или нескольких интерфейсов.

Описатели режимов доступа

Access modifiers

Access Modifier	**Ограничения**
public	Нет ограничений. Элементы отмеченные public видны любому методу любого класса.
private	Элементы класса A отмеченные как private доступны только методам класса A.
protected	Элементы класса A отмеченные как protected доступны методам класса A и методам *производным* от класса A.
internal	Элементы класса A отмеченные как internal доступны методам любого класса в сборке, в которой описан класс A.
protected internal	Элементы класса A отмеченные как protected internal доступны методам класса A, методам классов производных от класса A, а также любому классу в сборке, где описан класс A. Это как protected ИЛИ internal. (Нет режима protected И internal.)

Режим доступа к классу

- Класс могут иметь режим доступа public или internal (по умолчанию).
- По умолчанию элементы класса имеют режим доступа private, а типы (классы, структуры и т.п.) по умолчанию имеют тип internal.

```
// An internal class with a private default constructor.
class Radio
{
   Radio(){}
}
```

- Для разрешения доступа к классу Radio из других сборок нужно добавить режим доступа public к описанию класса.
- Чтобы разрешить другим типам использовать элементы объекта, нужно их отметить, как открыто доступных.

```
// A public class with a public default constructor.
public class Radio
{
  public Radio(){}
}
```

Пример описания класса

```
class Person {
  public string name; // задается значение ""
  public int age;  // задается значение 0
  public double salary; // задается значение 0.0
  public Person(string n, int a, double salary)
  {
    name = n;
    age = a;
    this.salary = s;
  }
  public void PrintPerson(){
    Console.WriteLine("name= {0}, age = {1},
    salary ={2}", name, age, salary);
   }
}
```

Полиморфизм

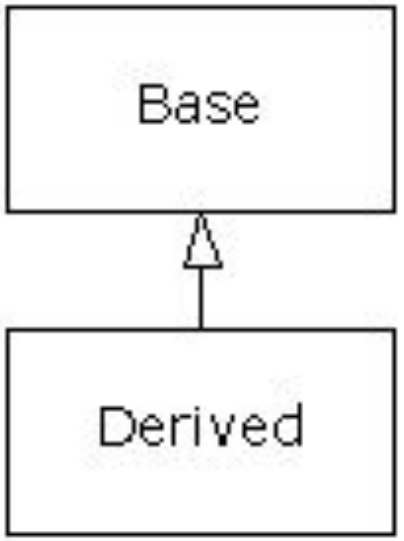
- Это способность по разному реализовать одни и те же ***открытые интерфейсы*** в разных классах.
- Полиморфизм позволяет вызывать методы и свойства объекта независимо от их реализации.
- Две способа реализации полиморфизма:
  - Через наследование
  - С помощью интерфейсов

Реализация полиморфизма через наследование

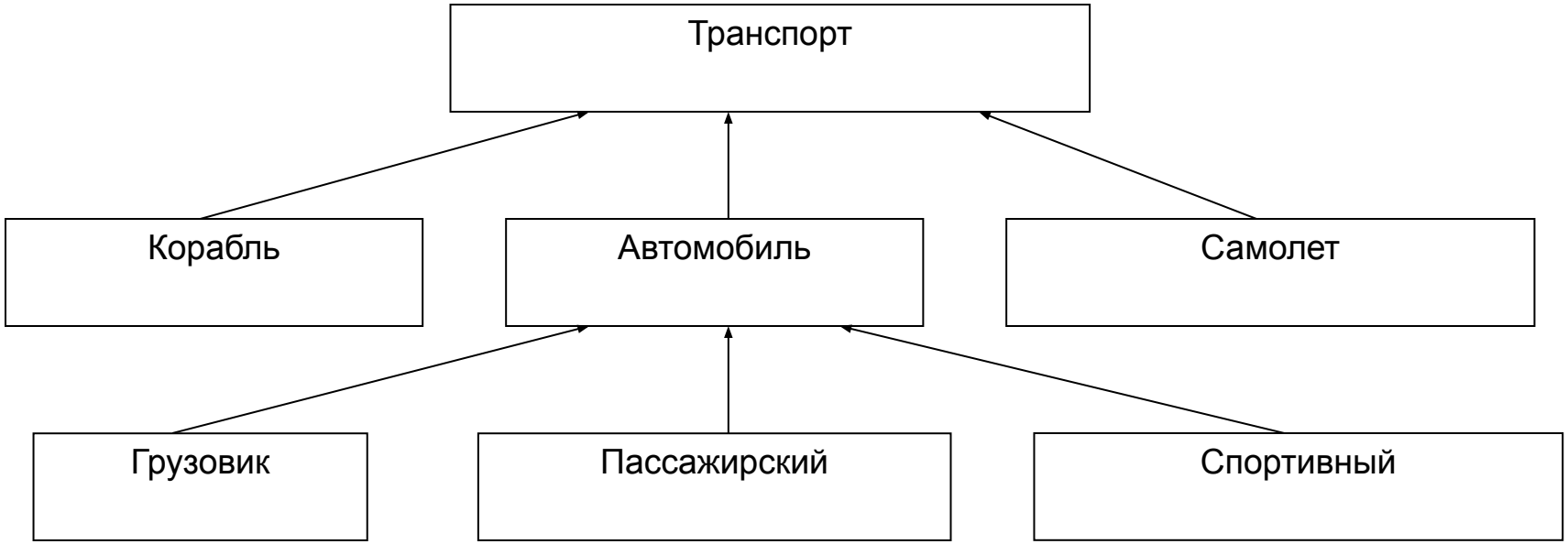
- Наследование позволяет создавать новые классы на основе существующих, при этом в новые классы включается protected и public функциональность старых классов. В новом классе эти элементы могут быть модифицированы и добавлены новые.
- Класс, объявленный на основе другого класса, называется его производным или классом-потомком.
- У любого класса может быть один прямой предок - базовый класс.
- Производные классы сохраняют все характеристики своего базового класса и способны взаимодействовать с другими объектами, как экземпляры базового класса.

Наследование (***Inheritance***)

- Любые знания о предметной области описываются в виде множества взимосвязанных понятий.
- Основная связь “is_a” (являться видом, вид-подвид) – наследование (другая важная связь – агрегирование (целое - часть))
- Есть базовый класс (обобщенный, может абстрактный), и есть производные классы (например, животные – собака, коза, лошадь)
- Производные классы наследуют (получают) от базового класса свойства и поведение, но могут его уточнять (дополнительные свойства, новое поведение, изменение поведения).



Наследование



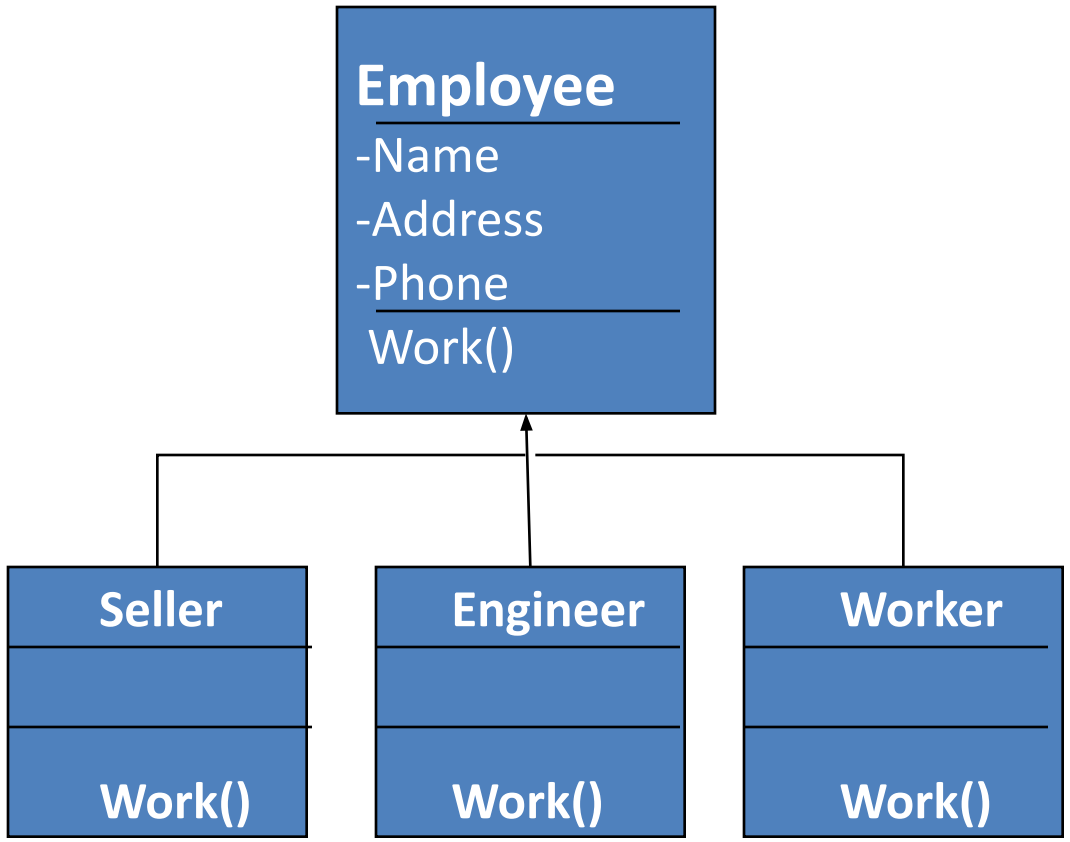
Наследование класса

```
Class Транспорт
{
...
}


class Автомобиль : Транспорт
{
  // описание свойств
  public string model;
  public float Расход_топлива;
  private int Число_цилиндров;
  // описание методов
  public void Повернуть_руль(){...};
  private Регулировка_датчика(){...};
  // описание события
  event Перегрев_двигателя();
}
```

Пример наследования сотрудников разных специальностей

- Базовый класс (Base Class)
  - Person
- Производные классы (Subclasses)
  - Employee
  - Customer
  - Student



Реализация полиморфизма через интерфейсы

- Интерфейс (interface) – соглашение, определяющее поведение объекта.
- Интерфейс определяет список свойств и методов класса.
- В объекте допустимо реализовать несколько интерфейсов, а один и тот же интерфейс можно реализовать в нескольких классах.
- Любые объекты, в которых реализован некоторый интерфейс могут взаимодействовать через него.

```
class Человек : IГрамотный, IВодитель

interface IГрамотный {Читат(); Писать();}
interface IВодитель {Завести(); Повернуть(); Затормозить
    ()}
```

Использование полиморфизма для программирования

```
//Массив сотрудников
Employee []  Company = new Employee [3];
// сохранение ссылок на объекты производных
  классов
Company[0] = new Seller();
Company[1] = new Manager();
Company[2] = new Worker();

//Выполнение работы сотрудниками
for (i=0; i<3; i++)
{
  Company[i].Work();
}
```

Составные элементы класса

1. Поля (field) – обычно скрытые данные класса (внутренне состояние)
2. Методы (methods) – операции над данными класса (поведение) (можно называть функциями)
3. Свойства (property) – доступ к данным класса с помощью функций
   - get – получить
   - set – задать
4. События (event) – оповещение пользователей класса о том, что произошло что-то важное.

Поля класса

Поля класса

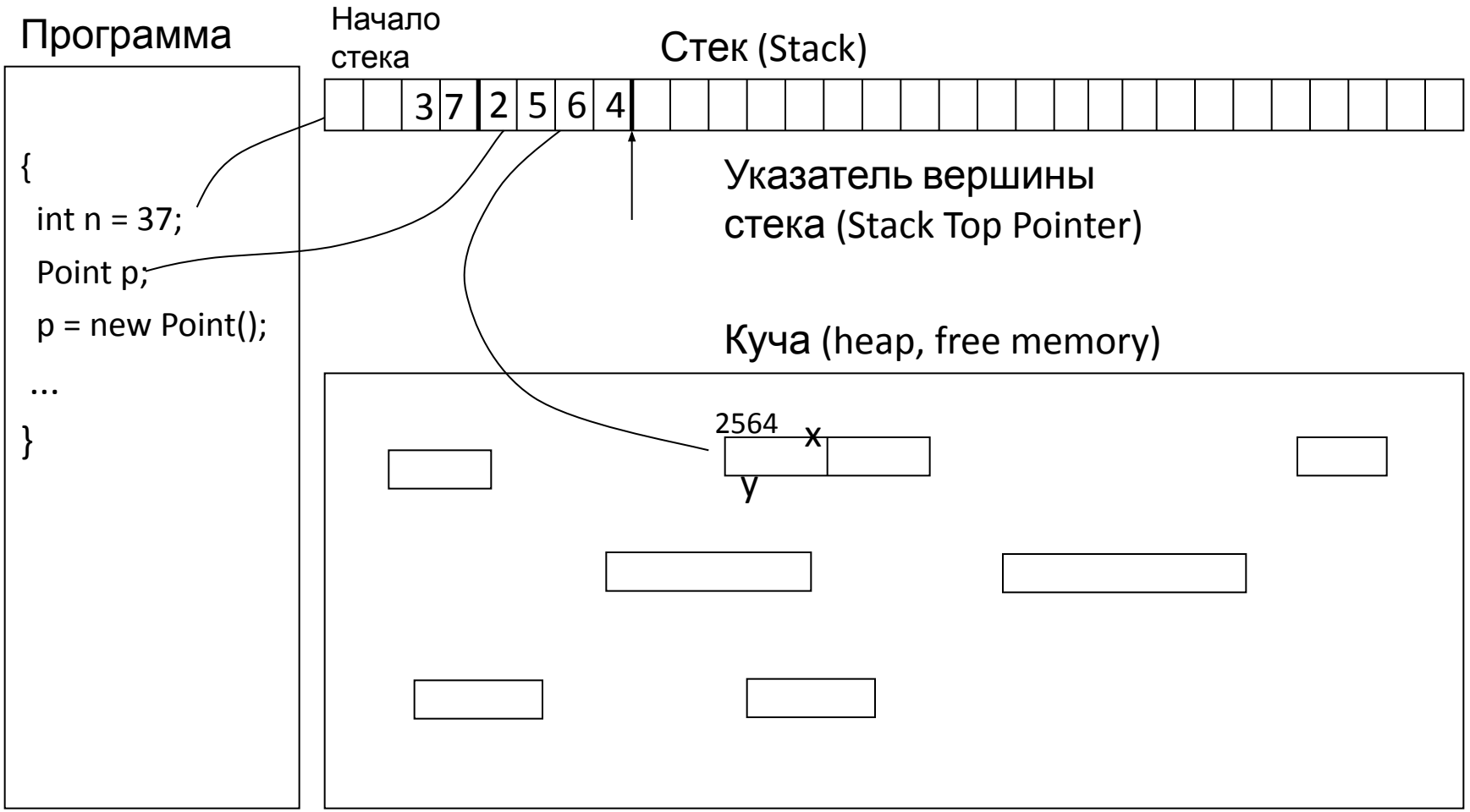
- Состояние объектов класса (а также структур, интерфейсов) задается с помощью переменных, которые называются ***полями*** (***fields***).
- При создании объекта – экземпляра класса, в динамической памяти выделяется участок памяти, содержащий набор *полей*, определяемых классом, и в них записываются значения, характеризующие начальное состояние данного экземпляра.
- Объявление полей выполняется следующим образом:
  [<режим_доступа>] [модификаторы] <тип> <имя>;

- Общим правилом является создание закрытых полей, имеющих режим доступа private. Данный режим задается полям по умолчанию.

- Любое воздействие на состояние объекта класса выполняется с использованием свойств или методов класса, которые контролируют последствия этих воздействий.

- Если полям класса не задается значение при объявлении, то они автоматически инициализируются значениями по умолчанию.
  - Для значащих переменных – это нулевое значение,
  - Для строк – это пустая строка,
  - Для ссылочных переменных – это стандартное значение null, как показано в комментариях описания класса Person.

Размещение полей в памяти программы



Поля класса (2)

- Поля класса создаются для каждого создаваемого объекта в выделенном ему участке памяти в "куче".
- Областью видимости полей являются все методы класса. При этом для использования поля требуется задавать только его имя.
- Например, метод вычисления возраста для объекта класса `Person` в днях может быть выполнено следующим образом:

```
public int CalcDays() { // вычисление возраста в днях
  int days = age * 365; // age – поле данного объекта
  return days;
}
```

- Если поле имеет режим public, то оно доступно там, где имеется ссылка на объект данного класса.
- Для обращения к этим полям из методов других классов (если поля открытые) нужно использовать ссылочную переменную, которая хранит ссылку на созданный объект.
- Например:

```
Person p; //объявление переменной типа Person
  p = new Person(); //создание объекта и сохр. ссылки
  p.Name = "Иванов П.И. "; //задание значения public поля
```

Поля класса (3)

- В качестве модификатора поля может использоваться ключевое слово static, обозначающее, что это статическое поле.
- Например, в классе Person может быть описано следующее статическое поле:

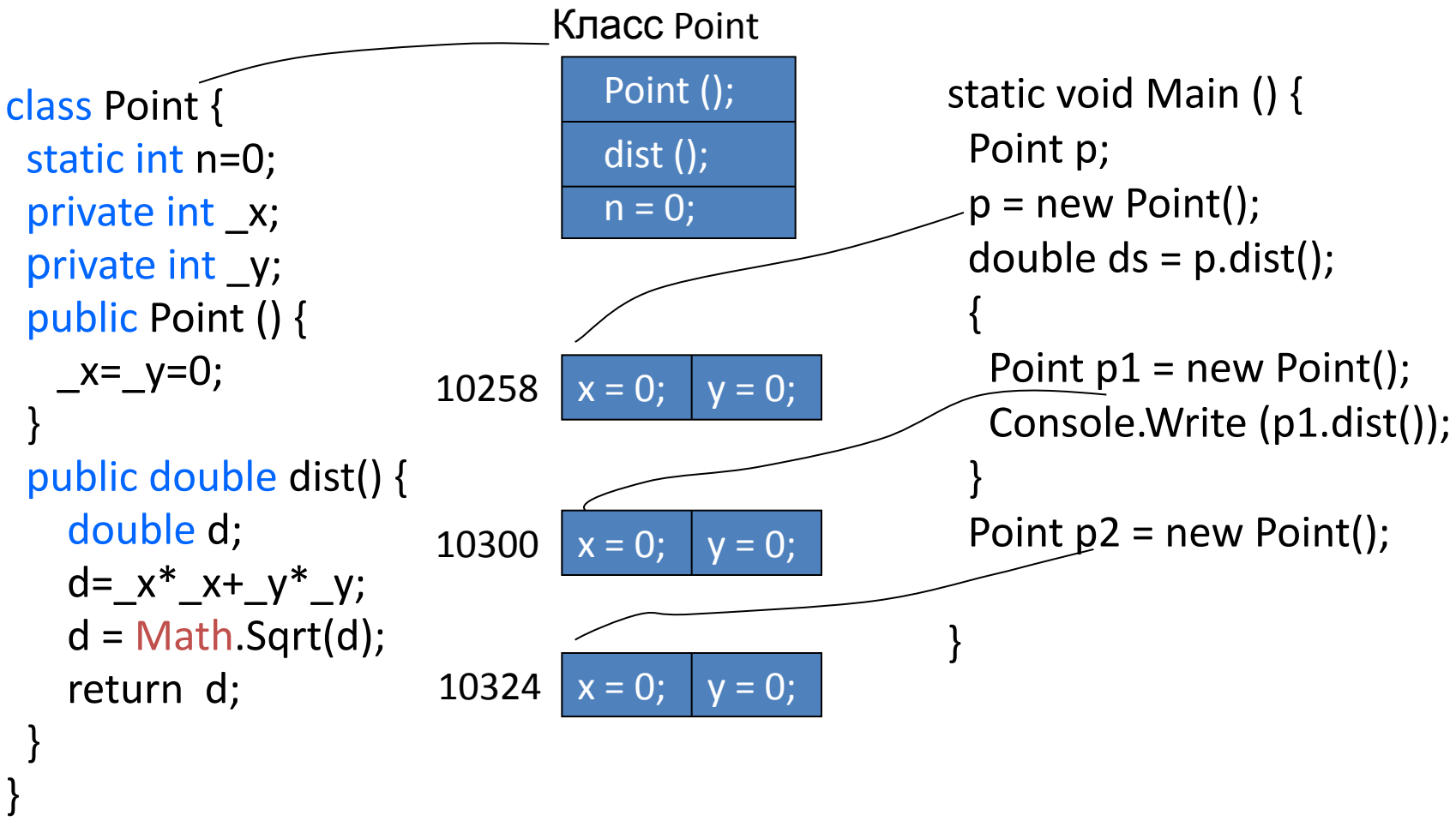
```
static int numPersons=0; // кол-во объектов класса
```

- Статическое поле класса создаются только одно для всего класса.
- Для обращения к нему нужно указать имя класса и через точку имя статического поля.
- Например:

```
Person.numPersons++;
```

- Время существования полей определяется объектом, которому они принадлежат. Объекты в "куче", с которыми не связана ни одна ссылочная переменная, становятся недоступными и удаляются сборщиком мусора.

Размещение описания методов класса и объектов



Методы классов C#

Методы программы

- Описываются только в классах
- Имеют доступ
  - к закрытым и открытым переменным класса (полям)
  - Локальным переменным

Описание и вызов метода

- Описание метода содержит

  <заголовок_метода>

  {

  тело_метода

  }

- Синтаксис заголовка метода
  - [модификаторы] {тип_результата} имя_метода ([список_формальных_параметров])

- Вызов метода
  - имя_метода([список_фактических_параметров])

Описание метода

- Заголовка метода

[режим доступа] <type> name (parameters) // method header
{
    statement;
    …
    statement;
    return X;
}

Method body

- Например:
  void A(int p) {...}
  int B(){...}
  public void C(){...}
- Описание параметра
  [ref|out|params]тип_параметра имя_параметра

Формальные параметры методов

- По умолчанию параметры передаются по значению.
- Значением переменной ссылочного типа является ее адрес.
- Можно задать передачу параметров по ссылке
  - ref – параметр должен иметь начальное значение
  - out – параметр может не иметь начального значения
- Синтаксис объявления формального параметра

  [ref | out] <тип> имя

Модификаторы параметров

Модификатор	Пояснение
(нет)	Если у параметра не задан модификатор, то предполагается что он передается по значению (т.е. вызываемый метод получает копию фактического параметра).
out	Данные передаются по ссылке. Данному параметру в вызываемом методе должно задаваться значение. Если вызываемый метод не задает значение данному параметру, то будет ошибка компиляции).
ref	Данные передаются по ссылке. Значение параметру задается вызывающим методом и может быть изменено в вызываемом методе.
params	Параметр с таким модификатором позволяет передавать переменное количество формальных параметров, как один логический параметр. В методе может быть только один формальный параметр с таким модификатором и он должен быть последним

Передача *произвольного числа* параметров

- Несмотря на фиксированное число формальных параметров, есть возможность при *вызове метода* передавать ему *произвольное число фактических* параметров.
- Для реализации этой возможности в списке формальных параметров необходимо задать ключевое слово params. Оно задается один раз и указывается только для последнего параметра списка, объявляемого как массив произвольного типа.
- При *вызове метода* этому формальному параметру соответствует *произвольное число фактических параметров*.
- Например:

```
void Sum(out long s, params int[] p)
{
    s = 0;
    for( int i = 0; i < p.Length; i++)
        p2 += (long)Math.Pow(p[i], 3);
    Console.WriteLine("Метод А-2");
}
```

Фактические параметры

- Фактические параметры должны соответствовать по количеству и типу формальным параметрам.
- Соответствие между типами параметров точно такое же, как и при присваивании:
  - Совпадение типов.
  - Тип формального параметра является производным от типа фактического параметра.
  - Задано неявное или явное преобразование между типами.
- Если метод перегружен, то компилятор будет искать метод с наиболее подходящей сигнатурой.

Выполнение вызова метода

- При *вызове метода* выполнение начинается с вычисления фактических параметров, которые являются выражениями.
- Упрощенно вызов метода можно рассматривать, как создание *блока*, соответствующий телу метода, в точке вызова. В этом *блоке* происходит замена имен формальных параметров фактическими параметрами.
- При передачи параметров *по значению* (нет модификаторов ref или out):
  - Создаются локальные переменные методов.
  - Тип локальных переменных определяется типом соответствующего формального параметра.
  - Вычисленные выражения фактических параметров присваиваются специально создаваемым локальным переменным метода.
  - ***Замена формальных параметров на фактические выполняется так же, как и оператор присваивания.***
- При передачи параметров *по ссылке* выполняется замена формальных параметров на реально существующий фактический параметр.
- Когда методу передается объект ссылочного типа, то все поля этого объекта могут меняться в методе любым образом, поэтому ref или out не часто появляются при описании параметров метода.

Перегрузка методов

- Перегруженные (overloaded) методы – это методы с одинаковым именем, но с разной ***сигнатурой***.
- ***Сигнатура*** это возвращаемый тип результата и типы передаваемых параметров.
- Например:

  int <имя метода> (int, float, double)
- Перегрузка методов выполняется для следующих целей:
  - чтобы метод принимал разное количество параметров.
  - чтобы метод принимал параметры разного типа, между которыми нет неявного преобразования.

Специальная переменная класса this

- В методах класса можно использовать переменную this.
- this это ссылка на тот объект для которого данный метод используется.
- this нигде не объявляется
- Чаще всего используется для обращения к полям класса, если имя параметров совпадает с именем поля.
- Например:

```
public Box (int Width, int Hight)
{
   this.Width = Width;
   this.Hight = Hight;
}
```

Формальные параметры методов

- Описание формального параметра
  [ref|out|params] тип_параметра имя_параметра
- **По умолчанию параметры передаются по значению.**
- Значением переменной ссылочного типа является ее адрес.
- Можно задать передачу параметров по ссылке
  - ref – параметр должен иметь начальное значение
  - out – параметр может не иметь начального значения
- Синтаксис объявления формального параметра
  [ref | out] <тип> имя

Пример

```
class Point {
    public void swap(ref int a, ref int b, out int c){
        //int c;
        c = a;
        a = b;
        b = c;
    }
    …
}
….
 int x = 5, y = 7, z;
Point p;
p = new Point();
p.swap(ref x, ref y, out z);
```

Пример передачи объектов по ссылке и значению

```
class Program
  {
     static void Main(string[] args)
     {
        MyClass mc;
        mc = new MyClass();
        MyMetod(ref mc);

        MyClass mc2;
        mc2 = new MyClass();

        swapMetod(ref mc, ref mc2);

        swapMetod2(mc, mc2);

        Console.WriteLine("Привет Мир!");
     }

    static void MyMetod(ref MyClass x)
     {
        x.a = 10;
     }
```

```
    static void swapMetod(ref MyClass x, ref MyClass y)
        {
           MyClass z;
           z = x;
           x = y;
           y = z;
        }

        static void swapMetod2(MyClass x, MyClass y)
        {
           MyClass z;
           z = x;
           x = y;
           y = z;
        }

  }
  class MyClass
  {
     public int a;
     public MyClass()
     {
        a = 0;
     }
  }
```

Перегрузка методов

- В C# не требуется уникальности имени метода в *классе*. Существование в *классе* методов с одним и тем же именем называется *перегрузкой*, а сами методы называются **перегруженными**.
- Уникальной характеристикой перегруженных методов является их ***сигнатура***.
- Перегруженные методы, имея одинаковое имя, должны отличаться
  - либо числом параметров,
  - либо их типами,
  - либо модификаторами (заметьте: с точки зрения сигнатуры, ключевые слова ref или out не отличаются).
- Уникальность сигнатуры позволяет вызвать требуемый перегруженный метод.

Пример перегрузки методов

Перегрузка метода A():

```
void A(out long p2, int p1){
   p2 =(long) Math.Pow(p1,3);
}
void A(out long p2, params int[] p){
  p2=0;
  for(int i=0; i <p.Length; i++)
     p2 += (long)Math.Pow(p[i],3);
   Console.WriteLine("Метод A-2");
}
void A(out double p2, double p1){
   p2 = Math.Pow(p1,3);
}
void A(out double p2, params double[] p){
   p2=0;
   for(int i=0; i <p.Length; i++)
      p2 += Math.Pow(p[i],3);
}
```

Вызовы перегруженного метода A():

```
public void TestLoadMethods(){
   long u=0; double v =0;
   A(out u, 7); A(out v, 7.5);
   Console.WriteLine ("u= {0}, v= {1}", u,v);
   A(out v,7);
   Console.WriteLine("v= {0}",v);
   A(out u, 7,11,13);
   A(out v, 7.5, Math.Sin(11.5)+Math.Cos(13.5),
15.5);
   Console.WriteLine ("u= {0}, v= {1}", u,v);
}
```

Спасибо за внимание !