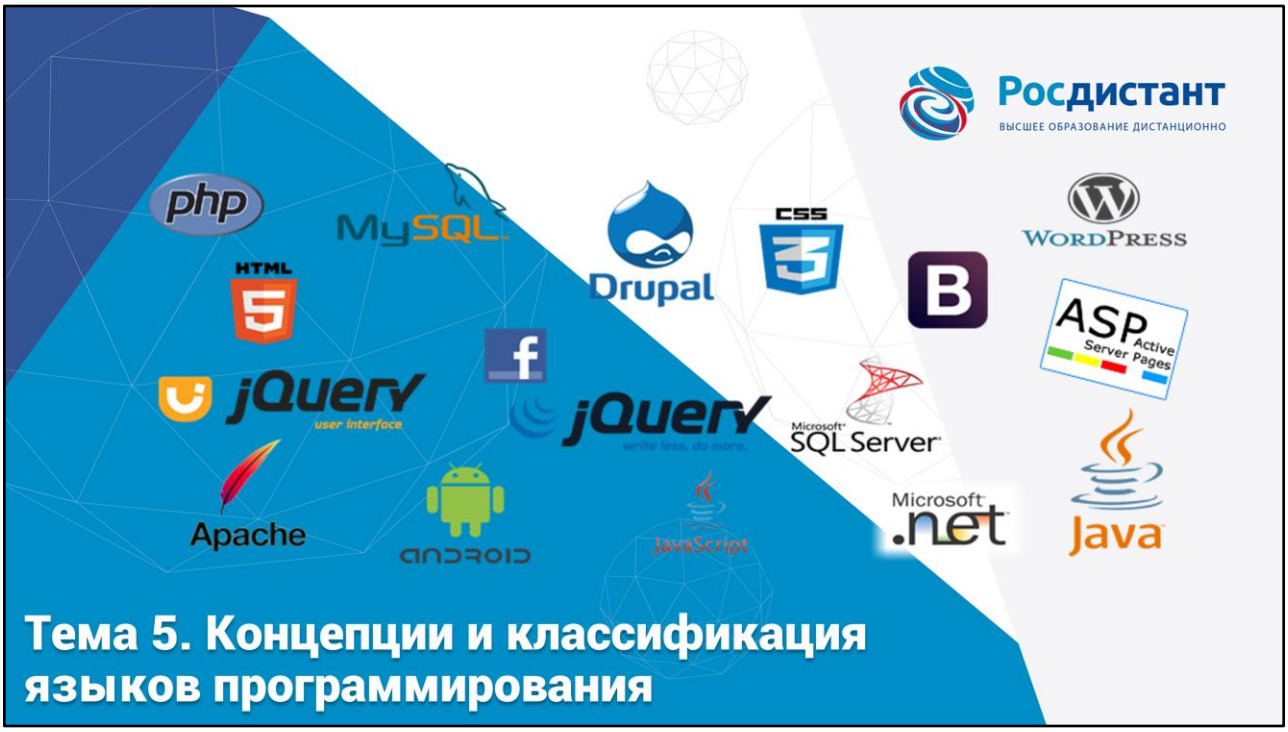
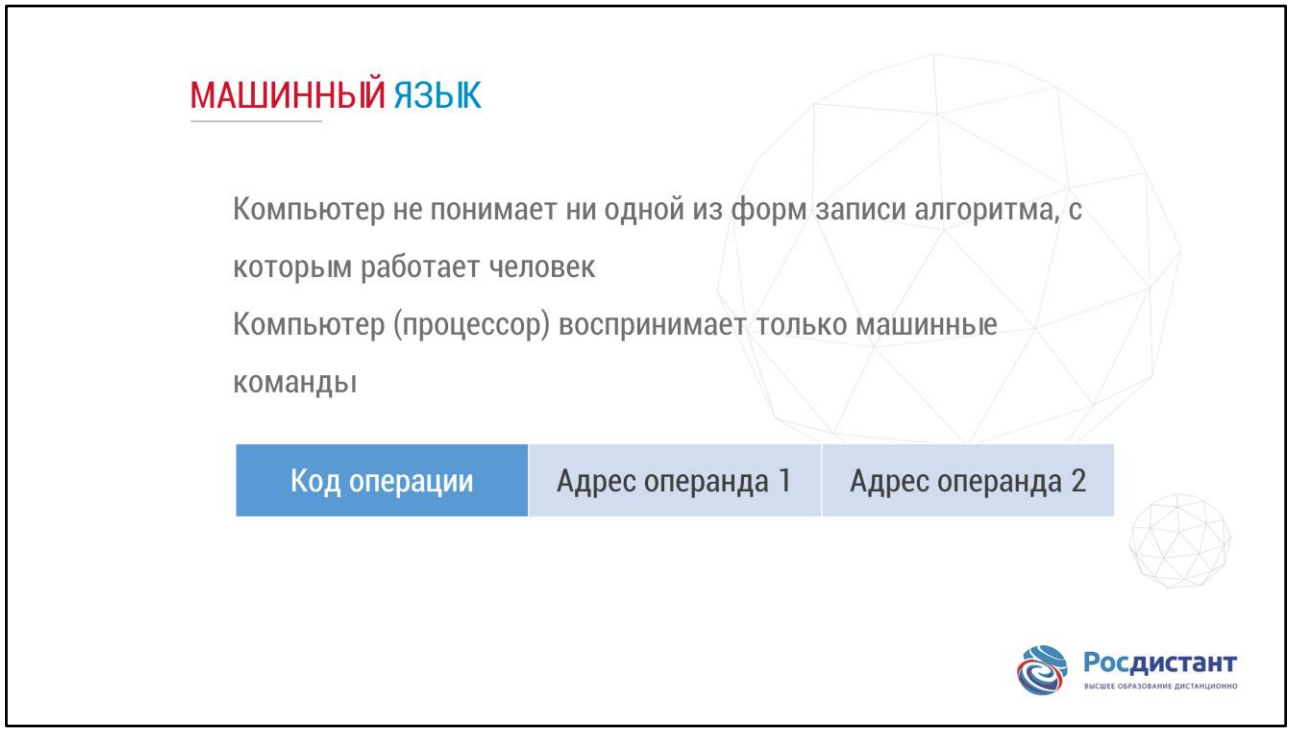
Росдистант
ВЫСШЕЕ ОБРАЗОВАНИЕ ДИСТАНЦИОННО
МЕТОДЫ РЕШЕНИЯ ПРОБЛЕМ В
ИНФОРМАТИКЕ
5 ЧАСТЬ



Тема 5. Концепции и классификация языков программирования

В данной теме мы рассмотрим общее определение языка программирования, его состав и классификации.

Материал данной темы позволит лучше ориентироваться в огромном количестве языков программирования и выбирать подходящий для решения конкретной задачи.

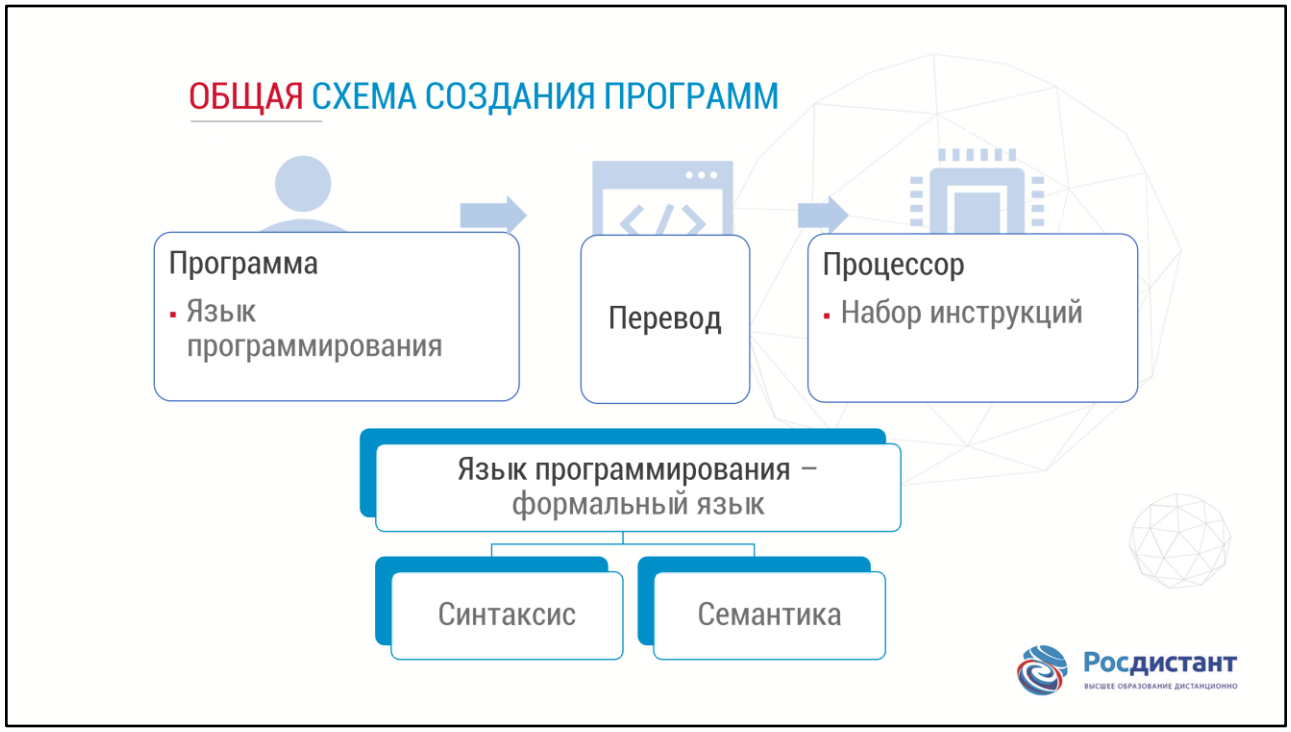


Машинный язык

Как уже обсуждалось выше, в результате решения поставленной задачи формируется алгоритм ее решения, который в последующем и должен выполнить компьютер. Но возникает проблема: как реально указать компьютеру, что он должен сделать? Ведь на самом деле компьютер напрямую не может понять ни словесную, ни графическую форму записи алгоритмов, так как компьютер не воспринимает ни человеческую речь, ни образы.

Компьютер, а точнее процессор, способен выполнять только определенный набор команд, которые называются машинными. Обычно команда процессора состоит из номера команды и указаний областей памяти, таких как регистр, кэш-память, оперативное запоминающее устройство. С ними необходимо осуществить указанную команду.

Команды процессора могут организовываться по разным схемам. На слайде приведена так называемая двухадресная схема.



Общая схема создания программ для компьютера

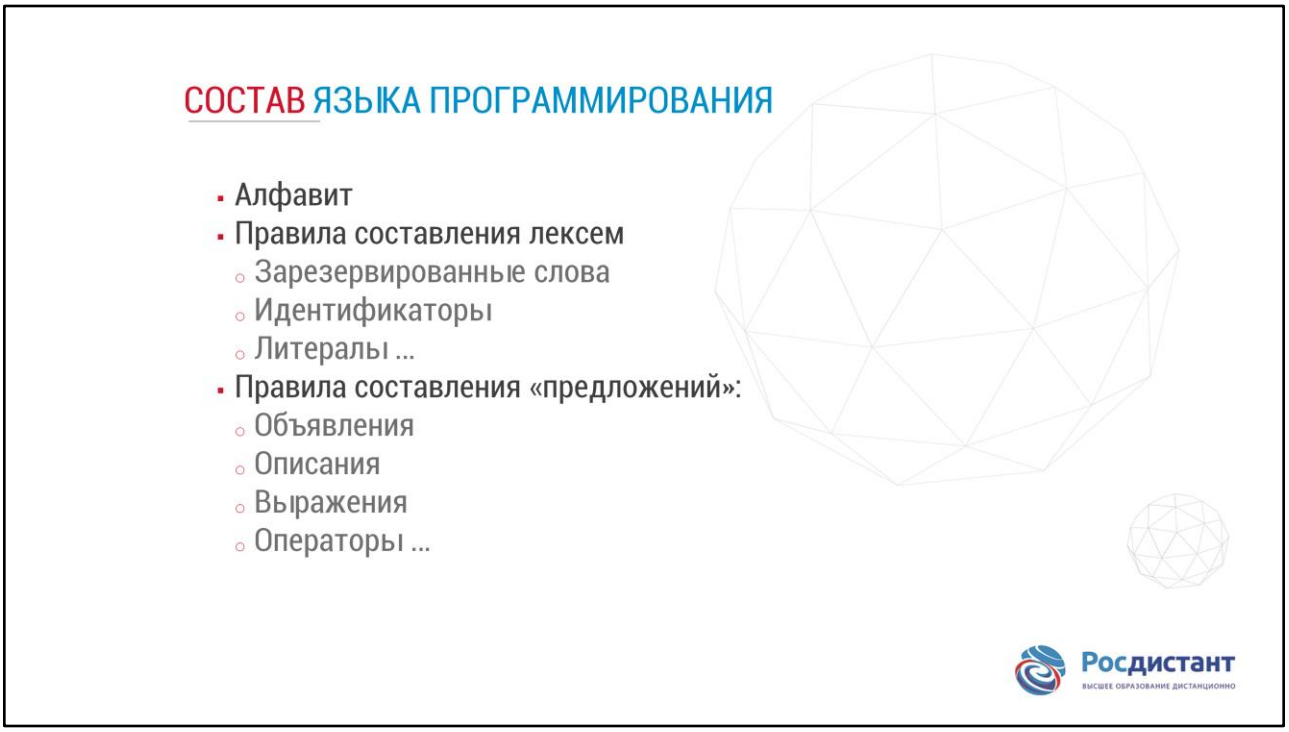
Для того чтобы облегчить процесс разработки программ для компьютера, было предложено писать программы на специальном языке программирования, а затем преобразовывать такую программу в набор машинных инструкций.

Под языками программирования понимается класс формальных языков. Любой человеческий язык нельзя считать формальным, так как в человеческих языках можно отступать от определённых правил. При этом вы сможете передать собеседнику свою мысль, и ваш собеседник ее поймет.

Формальный язык – это строгий язык. В формальном языке определен набор правил, выполнение которых не может нарушаться. Выполнение этих правил позволяет создавать синтаксически корректные конструкции, а программам, которые переводят запись на языке программирования в машинный код, проверять синтаксис конструкций, записанных программистом. Если программа-переводчик обнаружит, что конструкция синтаксически записана неправильно, то выдается ошибка и перевод программы останавливается. Таким образом, можно говорить о синтаксисе языка программирования.

Синтаксически корректные конструкции имеют свою семантику, то есть смысл. Например, синтаксически корректно записанный оператор for означает необходимость выполнения тела цикла определенное количество раз.

Следовательно, в языке программирования определены синтаксис и семантика.



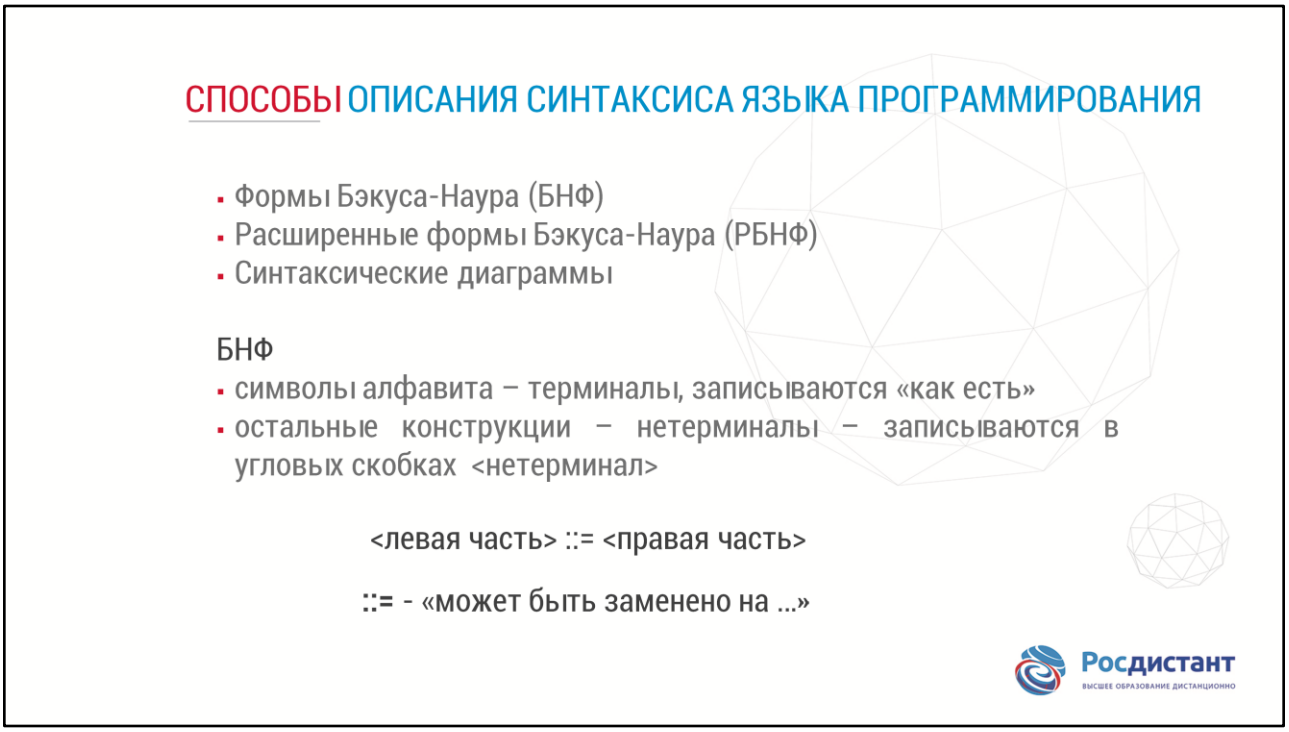
Состав языка программирования

Состав языка программирования определяется правилами построения синтаксически корректных конструкций.

Во-первых, в языке программирования жестко определен алфавит языка – набор используемых символов. Никакие другие символы нельзя использовать при написании выражений на данном языке программирования.

Далее определяются правила построения лексем. Условно под лексемами можно подразумевать корректные слова языка программирования. Например, в состав лексем входят так называемые зарезервированные слова, идентификаторы, литералы и т. д. Следует отметить, что если иные конструкции языка программирования можно декомпозировать, то разложить лексему на более мелкие конструкции нельзя – теряется смысл. Так, можно разложить блок операторов на последовательность операторов, а последние – на наборы зарезервированных слов и идентификаторов, но нельзя разложить лексему – зарезервированное слово «select» на более мелкие части.

Далее строятся правила составления синтаксически корректных конструкций языка программирования, определяющие, как правильно записывать «предложения» языка программирования. Например, определяются правила записи операторов, блоков операторов, процедур, функций, объявлений и т. п.



Способы описания синтаксиса языка программирования

Рассмотрим формальные способы описания синтаксиса языков программирования.

Наибольшее распространение получили две формы:

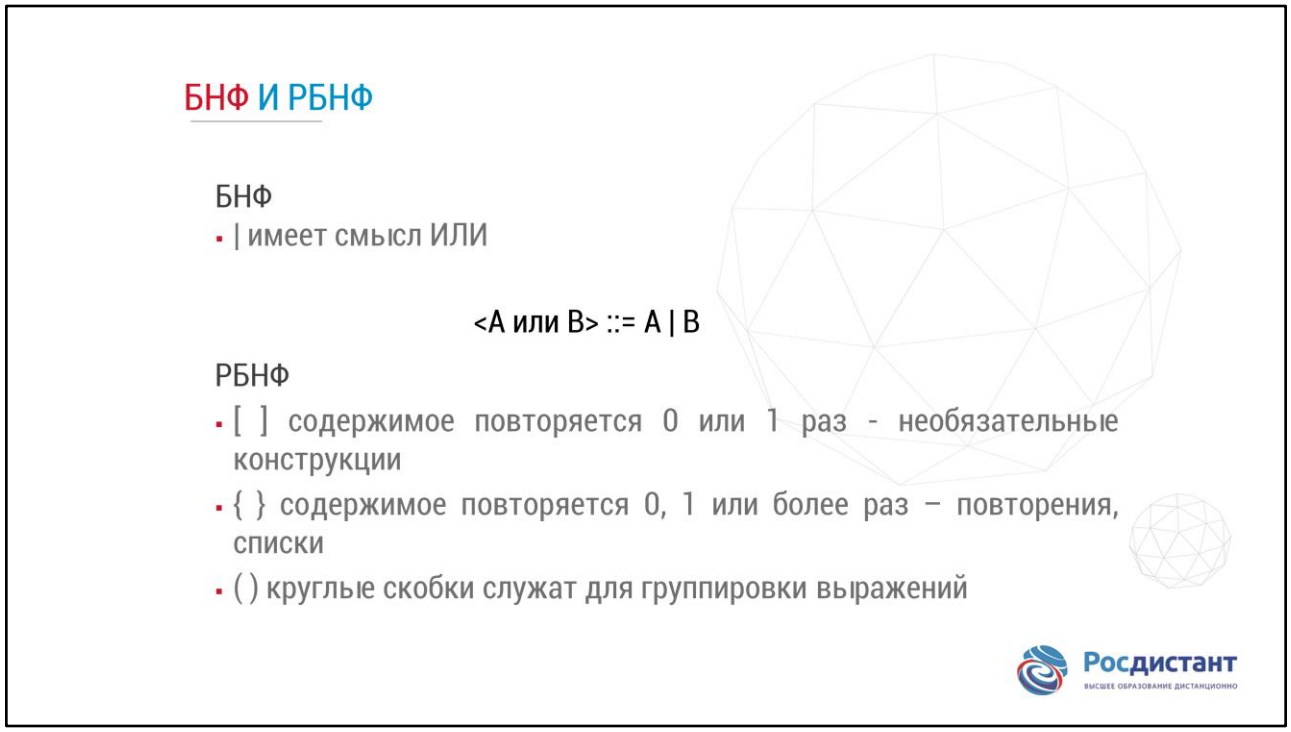
- формы Бэкуса – Наура БНФ и их последующая модификация – расширенные формы Бэкуса – Наура РБНФ;
- синтаксические диаграммы.
- Рассмотрим формы Бэкуса – Наура.

БНФ – это текстовая форма представления грамматик. БНФ были разработаны известными учеными в области компьютерных наук Джоном Бэкусом и Петером Науром.

Неопределяемые конструкции называются терминальными символами или терминалами и в БНФ записываются в неизменном виде. Не углубляясь в теорию грамматик, отметим, что в качестве терминалов обычно используют символы алфавита и зарезервированные слова, хотя последние могут быть просто описаны с использованием БНФ. Все остальные конструкции, получаемые путем применения БНФ, называются нетерминальными или нетерминалами. Нетерминалы записываются в угловых скобках. БНФ имеет левую и правую части, разделенные конструкцией «::=», что может пониматься как «может быть заменено на …». То есть форма <левая часть> ::= <правая часть> читается как «левая часть может быть заменена на правую часть».

Таким образом, в левой части записывают определяемое понятие в виде

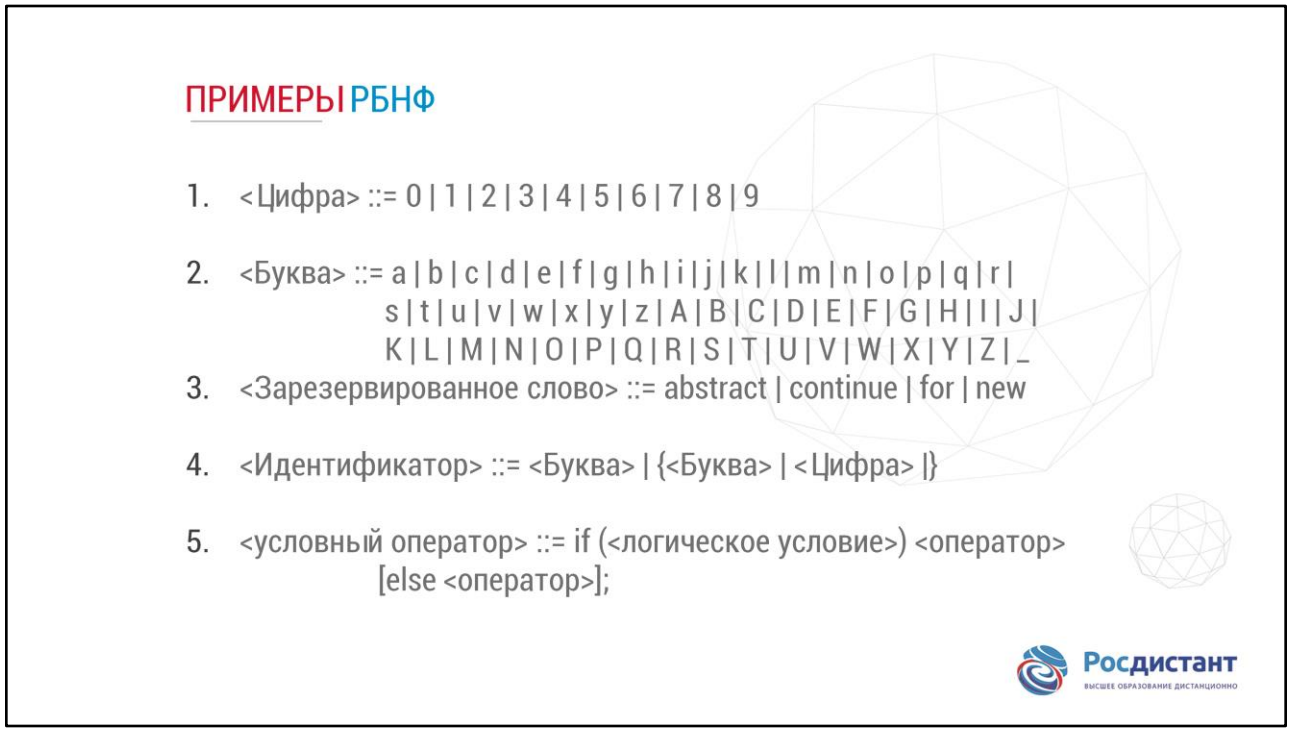
нетерминала, а справа – определяющую конструкцию.



В правой части БНФ могут быть использованы следующие символы:
| – символ вертикальной черты, имеет значение «или».
Например, форма <A или B> ::= A | B.
Формы Бэкуса – Наура достаточно громоздки, поэтому были придуманы их расширения РБНФ, которые дополняются следующими символами:

- квадратные скобки [ ]. Содержимое в квадратных скобках может повторяться 0 или 1 раз. Таким образом, такое содержимое может быть, а может отсутствовать. Таким образом определяются необязательные конструкции;
- фигурные скобки { }. Содержимое в фигурных скобках может повторяться 0, 1 или более раз. Таким образом, можно определять различные повторения, например, списки;
- круглые скобки ( ). Круглые скобки служат для группировки выражений.

Это основные дополнительные элементы РБНФ. В литературе можно найти использование других элементов, например «,».
Рассмотрим несколько примеров РБНФ.



Примеры РБНФ

Первая форма определяет понятие «цифра».

Вторая форма определяет понятие «буква». Обратите внимание, что символ подчеркивания считается буквой.

Третья форма вводит набор зарезервированных слов. Это условный пример. В реальности количество зарезервированных слов может достигать нескольких десятков.

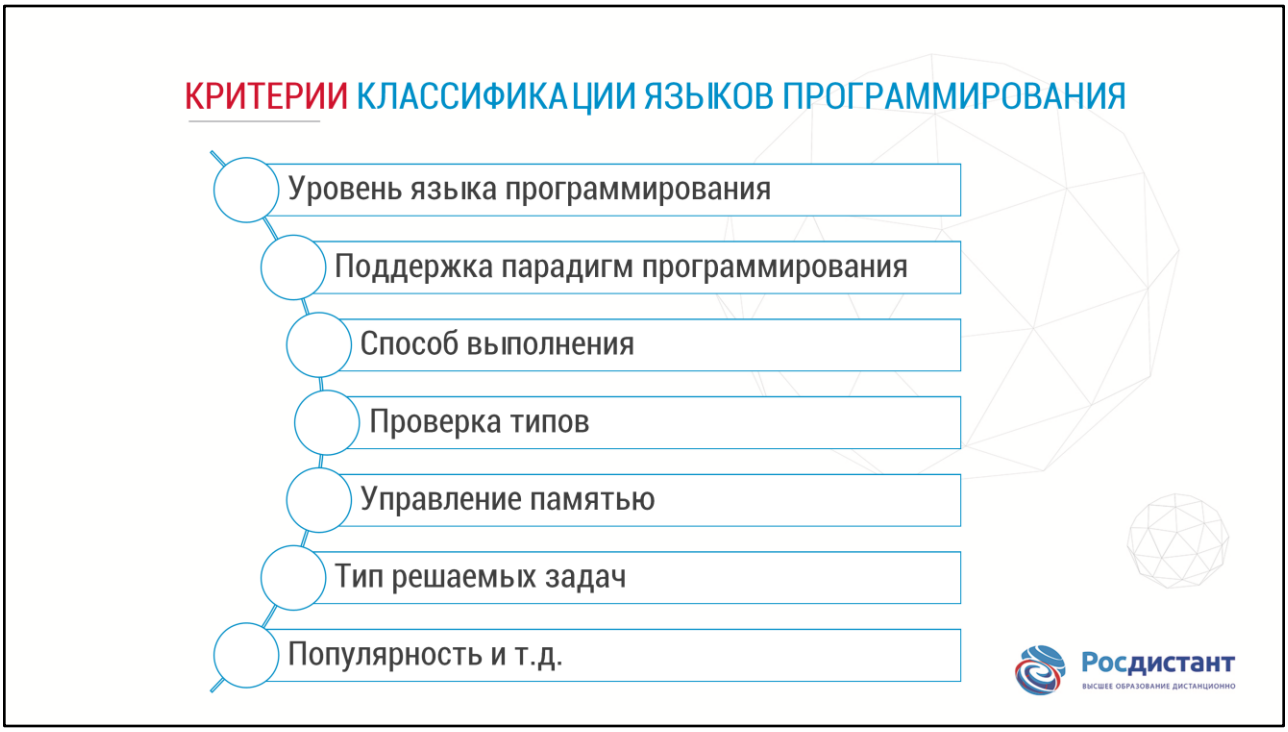
Четвертная форма вводит понятие «идентификатор». Из примера видно, что идентификатор – это последовательность букв и цифр, причем первым символом должна быть буква.

Пятый пример – синтаксис условного оператора if.

Последний пример показывает, что вслед за ключевым словом if в круглых скобках записывается логическое условие. Далее за скобками записывается оператор, который выполняется, если логическое условие истинно. В квадратных скобках записана необязательная часть, реализующая альтернативную ветку – зарезервированное слово else и оператор, который будет выполнен, если логическое условие ложно.

Данный пример показывает несущественный недостаток РБНФ: в современных языках программирования символы РБНФ – вертикальная черта, квадратные, угловые и фигурные скобки – являются символами и самого языка программирования. Поэтому для различения символов РБНФ и языка программирования могут применяться различные способы отличения. Так, в

приведенном примере терминальные символы выделены жирным шрифтом.

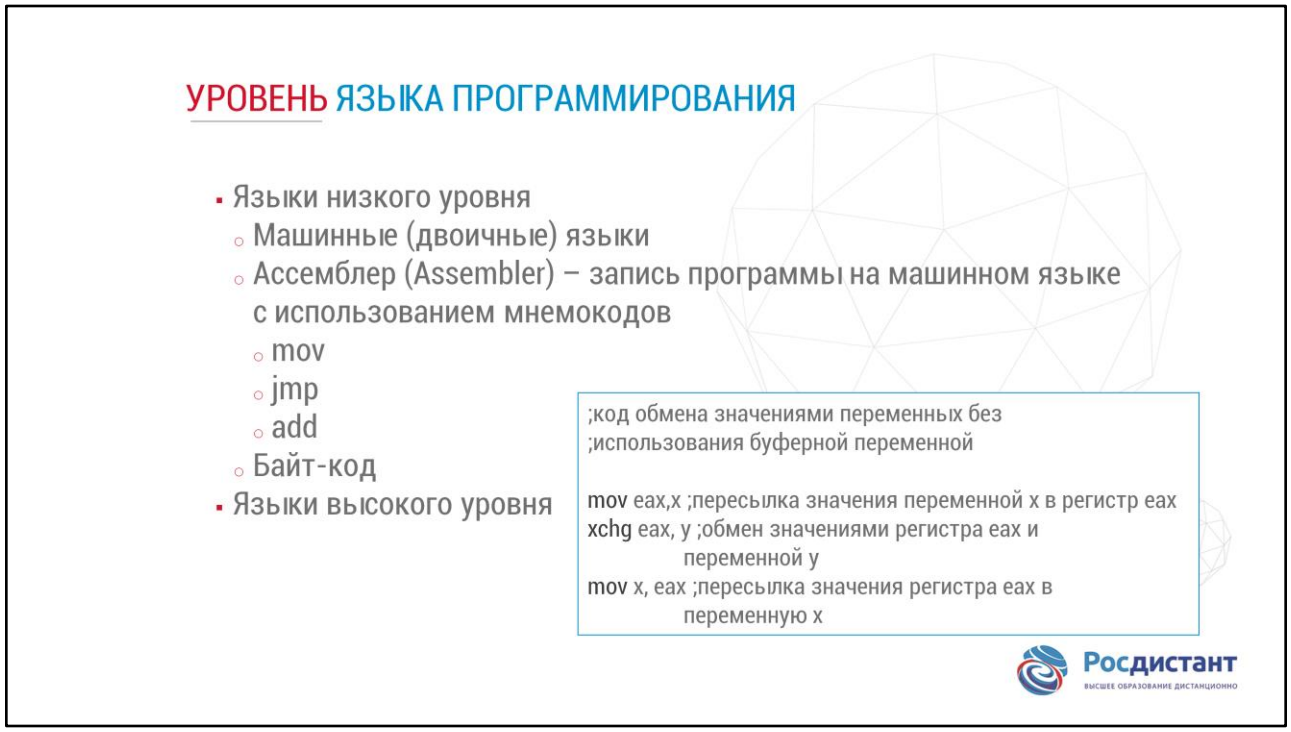


Критерии классификации языков программирования

В настоящее время, по разным источникам, общее число языков программирования превышает восемь тысяч. Эти языки программирования разрабатывались для широчайшего круга задач. В этом курсе мы уделяем внимание только языкам программирования, которые наиболее востребованы на рынке разработки прикладного программного обеспечения.

Классифицировать языки программирования можно по разным критериям:

- уровень языка программирования;
- поддержка парадигм программирования;
- способ выполнения;
- проверка типов;
- управление памятью;
- тип решаемых задач;
- популярность и многое другое.



Уровень языка программирования

Языки программирования делятся на два класса: языки низкого уровня и языки высокого уровня. Однако в последнее время можно наблюдать и иные виды классификации.

К языкам низкого уровня относят машинный язык, который был рассмотрен ранее. Напомним, что машинный язык строго ориентирован на конкретную архитектуру процессора, а программы на машинном языке записываются в так называемом машинном коде.

Сегодня программы в машинном коде практически не пишутся, за исключением случаев системного программирования отдельных мелких задач. Некоторые из современных языков промышленного, как мы называем, программирования позволяют вставлять в свой код инструкции на двоичном языке.

Первыми языками, которые позволили программисту поднять уровень конструкций программирования, был язык ассемблера. Язык ассемблера, по сути, есть способ записи программы на машинном языке с использованием мнемокода. Мнемокод – это символьная запись команды на машинном языке: код команды и ее аргументы.

Приведем некоторые примеры:

- *mov* [мув] – пересылка данных из областей памяти;
- *jmp* [джамп] – команда перехода на заданную метку;
- *add* [эд] – команда сложения и т. д.

На слайде также приведен фрагмент кода на языке ассемблера, меняющий

значение двух переменных без использования буферной переменной.

Уровень языка программирования

Чем сильнее отличается семантика языка от машинного, тем выше уровень языка.

Несмотря на то, что программы, написанные на низкоуровневых языках программирования, позволяют писать высокоскоростные программы, учитывающие особенности вычислительной архитектуры, они не обладают свойством переносимости и масштабирования. Такие программы нельзя перенести на другую вычислительную архитектуру.

Для работы с различными типами данных, включая типы данных, определяемые пользователями, а также с различными алгоритмическими структурами эти понятия поднимались на более высокий уровень абстракции. Новые языковые конструкции позволяли программисту сосредоточиться именно на разработке алгоритма, не вдаваясь в особенности той или иной вычислительной архитектуры.

Примерами высокоуровневых языков программирования являются Algol, Fortran, C, C++, C#, Java, Python, R и многие другие.

Развитие и классификация высокоуровневых языков программирования тесно связаны с ориентацией на одну или несколько парадигм программирования.

Парадигмы программирования

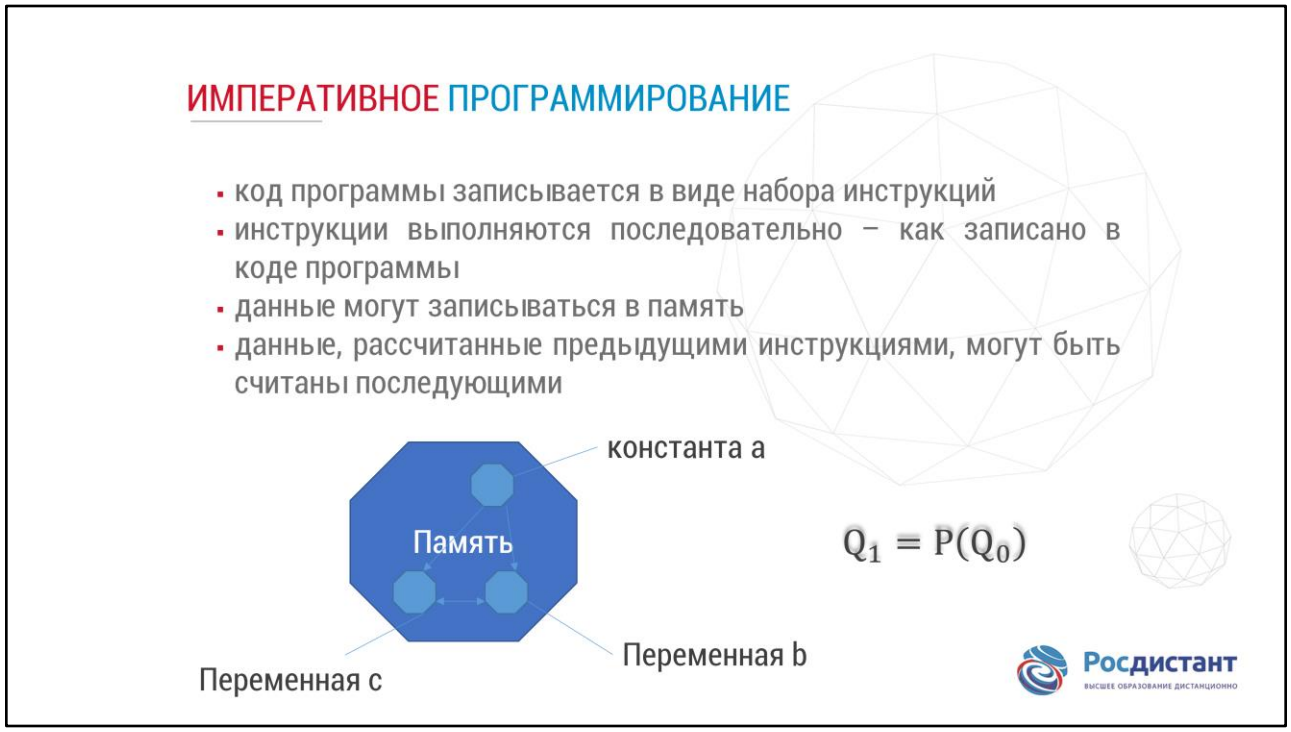
Развитие высокоуровневых языков тесно связано с понятием парадигмы, или модели программирования, – совокупности подходов, принципов, методов и понятий, определяющих способ разработки и написания программ.

Таким образом, парадигма программирования определяет, какими абстракциями может оперировать программист, а также какие отношения между абстракциями допустимы в том или ином языке программирования.

В настоящее время не существует стандартного набора парадигм программирования, однако можно привести наиболее устоявшийся:

- императивное программирование;
- структурное программирование;
- объектно-ориентированное программирование;
- функциональное программирование;
- декларативное программирование;
- логическое программирование.

Рассмотрим эти и другие понятия более подробно.



Императивное программирование

Императивное программирование подразумевает выполнение следующих требований:

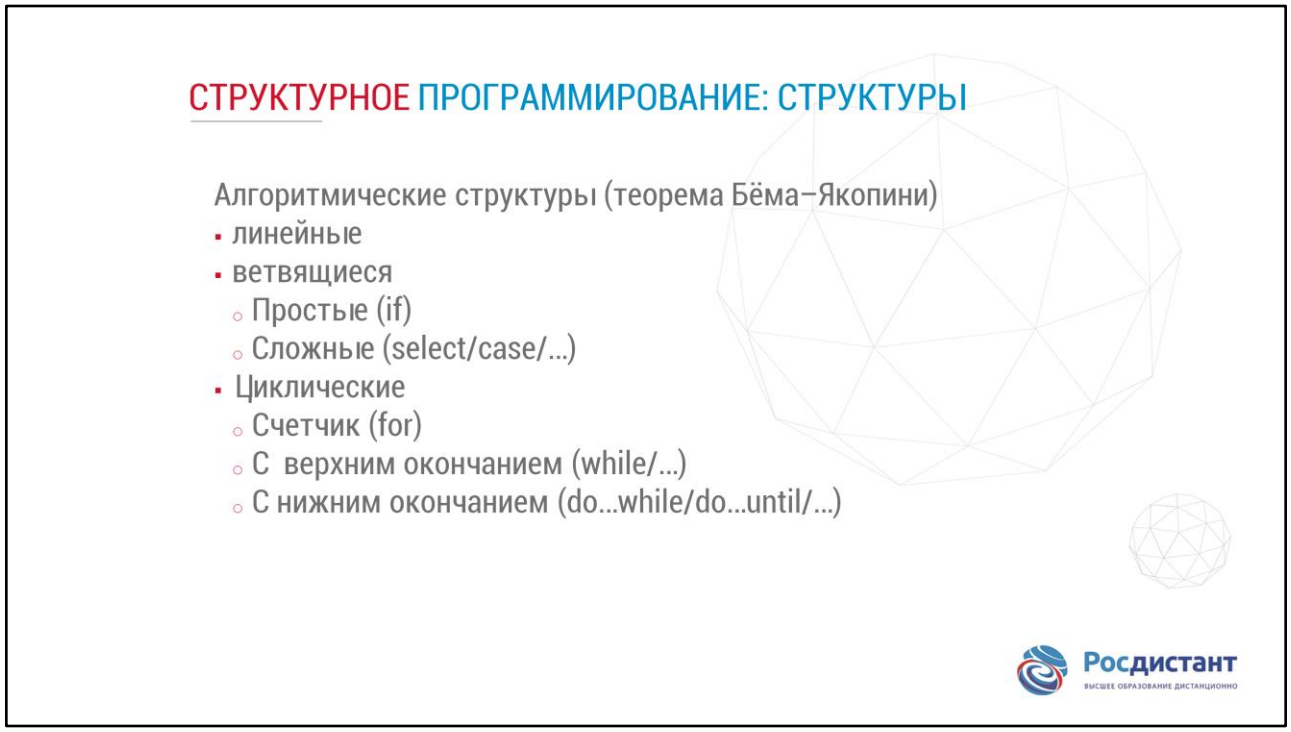
- код программы записывается в виде набора инструкций;
- инструкции выполняются последовательно, как записано в коде программы;
- данные могут записываться в память;
- данные, рассчитанные предыдущими инструкциями, могут быть считаны последующими.

Таким образом, в императивной программе инструкции записываются в той последовательности, которая обоснована разрабатываемым алгоритмом. Важнейшее значение в императивной программе имеет концепция памяти: каждая выделенная область памяти может иметь свое именование и хранить значение определенного типа. Так появились типизированные константы и переменные. Если в память, указанную как константная, во время программы нельзя записывать значения, то в переменные это делать разрешено. Значение константы записывается в памяти во время инициализации программы.

И наконец, после записи данных в память – область переменных – можно считывать значения. В то же время, если в переменную еще не записано значение, а программист хочет его использовать, то при проверке могут быть выброшены предупреждение или ошибка «Попытка чтения из неинициализированной памяти».

Условно работу императивной программы можно считать переводом памяти из

начального состояния Q_0 в конечное Q_1 посредством отображения, задаваемого программой P.



Структурное программирование

Структурная парадигма предполагает использование трех алгоритмических структур:

- линейных;
- ветвящихся;
- циклических.

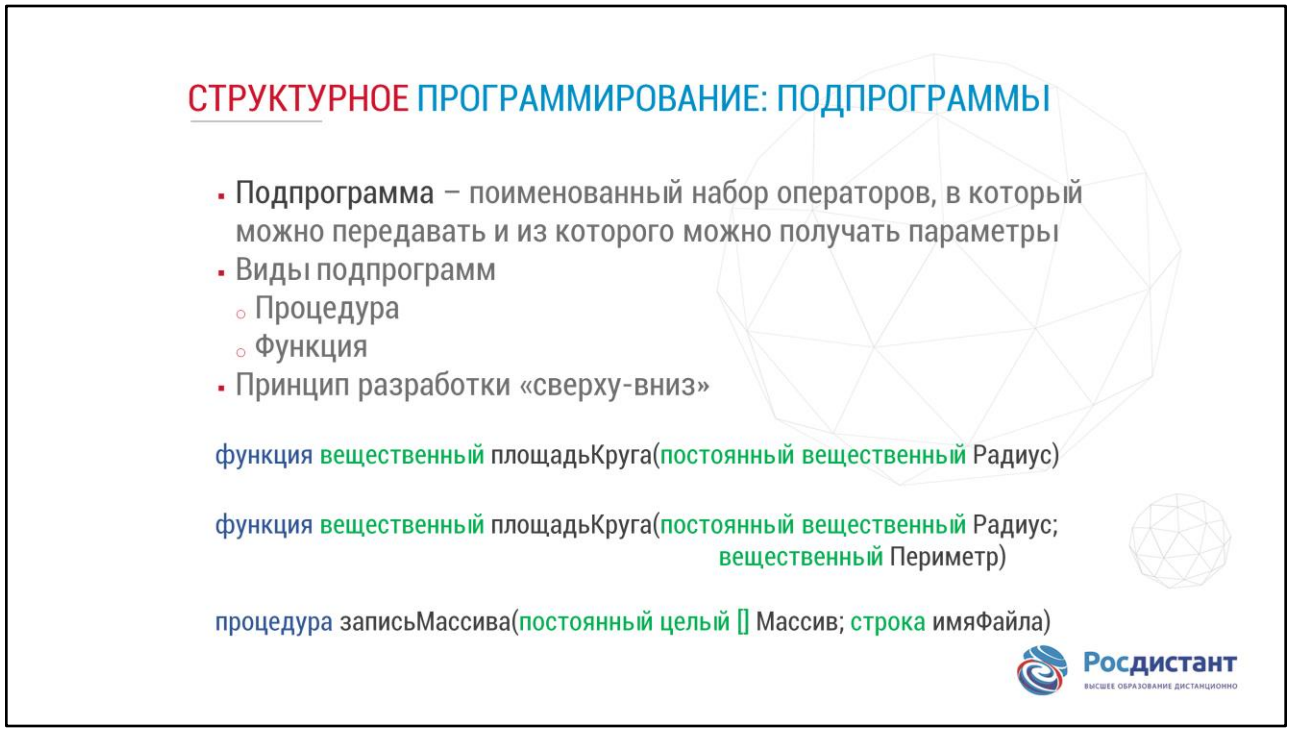
Данное утверждение обосновывается теоремой Бёма – Якопини, утверждающей, что любой алгоритм может быть реализован в виде комбинаций линейных, ветвящихся и циклических структур.

Линейные структуры задаются записанной последовательностью, как того требует императивное программирование.

Ветвящиеся структуры бывают простыми и множественными. Простые ветвящиеся структуры реализуются на языках программирования с помощью оператора if, сложные – с помощью операторов select, case и т. п.

Циклические структуры реализуются операторами цикла:

- оператор-счетчик for;
- операторы цикла с верхним условием-окончанием, например, while;
- операторы цикла с нижним условием-окончанием, например, do.



Подпрограммы

Кроме того, структурное программирование предполагает использование подпрограмм – поименованных наборов операторов, в которых можно передавать и из которых можно получать параметры. Различают два вида подпрограмм:

- процедуры – подпрограммы, не возвращающие значения. Возврат значений возможен через параметры вызова;
- функции – подпрограммы, возвращающие значение. Возврат значений возможен также через параметры вызова.
- Очевидно, что наиболее общим понятием является понятие «функция». Например, на языке С можно описывать только функции.

Рассмотрим сказанное более подробно.

Предположим, нам нужно часто считать площадь круга по известному радиусу. Для этого можно объявить и реализовать функцию, получающую единственный вещественный аргумент – радиус и возвращающую собственно площадь круга, как вещественный результат.

В примере слово «постоянный» означает, что значение параметра внутри функции изменять нельзя. Действительно, зачем менять радиус объекта, если надо просто посчитать его площадь? Это классический пример правильного описания параметров функции: запрещать менять значения параметров, если это не указано явно.

Если, к примеру, надо вычислять еще и периметр, то к нашей функции можно

добавить еще один выходной параметр, в который будет записываться периметр круга.
Обратите внимание, что здесь параметр «периметр» объявлен не как константный, то есть его можно и нужно изменить внутри функции.
Если блок операторов неоднократно должен выполнять только последовательность действий без возврата результата, то можно использовать процедуру. Например, если необходимо многократно записывать массив чисел в файл, то можно создать подпрограмму, которая в качестве своих аргументов получает два параметра: массив и имя файла.
Квадратные скобки показывают, что передается массив целых чисел.
Наконец, необходимо отметить, что структурное программирование подразумевает разработку программы по принципу «сверху вниз».

Появление объектно-ориентированного программирования

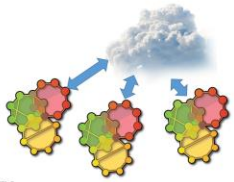
В 70-е года XX века сложность и объем разрабатываемого программного обеспечения выросли многократно. Этот рост объема и сложности был обусловлен широким внедрением различных информационных систем, к которым предъявлялись все более жесткие требования по объему, скорости и видам обрабатываемой информации. Даже используя языки структурного и императивного программирования и лучшие практики управления ИТ-проектами, командам разработчиков все чаще и чаще не удавалось выполнить требования контракта. Все больше проектов заканчивались либо со значительным превышением бюджета или сроков выполнения, либо с превышением того и другого одновременно. В то же время существенная доля проектов вообще снималась с выполнения по причине невозможности реализации в разумные сроки и за разумные бюджеты. Так начался известный кризис программирования начала 70-х годов.

Основная причина кризиса программирования заключалась в том, что в императивных программах данные и алгоритмы их обработки разделены. И чем больше алгоритмов, обрабатывающих данные, тем сложнее отследить их целостность и актуальность. В больших проектах количество модулей, изменяющих одни и те же данные, становилось огромным, и система полностью выходила из-под контроля.

Одним из выходов из кризиса стало появление объектно-ориентированного программирования.

ОСНОВНЫЕ ПРИНЦИПЫ ООП: ИНКАПСУЛЯЦИЯ

Императивное и структурное: Данные и алгоритмы отдельно



Инкапсуляция:

- объединение данных и методов по их обработке в объекте
- разделение доступа к данным и методам
- свойства C#, Pascal
- суррогатные свойства C++, Java
- поля + методы = члены класса

ООП, принцип инкапсуляции: Данные и алгоритмы объедены и «охраняются» объектом

C#

```
class Person
{
  private string _name; // the name field
  public string Name => _name;    // the Name property
}
```

Java

```
public class Person {
    private String name; //the name field
    //getter getName as property
    public String getName() {
        return this.name;
    }
}
```

Росдистант
ВЫСШЕЕ ОБРАЗОВАНИЕ ДИСТАНЦИОННО

Основные принципы ООП: инкапсуляция

В объектно-ориентированном программировании есть несколько основополагающих принципов.

В объектно-ориентированных программах основным понятием является понятие «объект». В отличие от подходов, лежащих в основе императивного и структурного программирования, в объектно-ориентированном программировании данные и методы, обрабатывающие эти данные, заключены в объекте. Говорят, что объект инкапсулирует и данные, и методы. Гениальность этой идеи заключается в том, что теперь к данным можно обратиться только через объект, который определяет механизмы чтения-записи в свои данные. Таким образом, первым принципом объектно-ориентированного программирования стала инкапсуляция, которую необходимо рассматривать как совокупность двух принципов:

- объединение данных и методов по их обработке в объекте;
- разделение доступа к данным и методам.

В теории объектно-ориентированного программирования механизмы чтения-записи принято называть свойствами. Ряд языков программирования, таких как Паскаль, C#, поддерживают свойства напрямую. В других языках программирования, таких как C++, Джава, механизм свойств напрямую не используется. Программисты должны создавать так называемые суррогатные свойства, используя геттеры и сеттеры.

В ООП поля и методы класса часто объединяют под термином «члены класса».

Основные принципы ООП: наследование

Объекты в ООП выполняют роль переменных в императивном программировании. Как известно, объекты должны описываться некоторым типом данных, такие объектные типы называются классами. Очень часто говорят, что объект – это экземпляр класса.

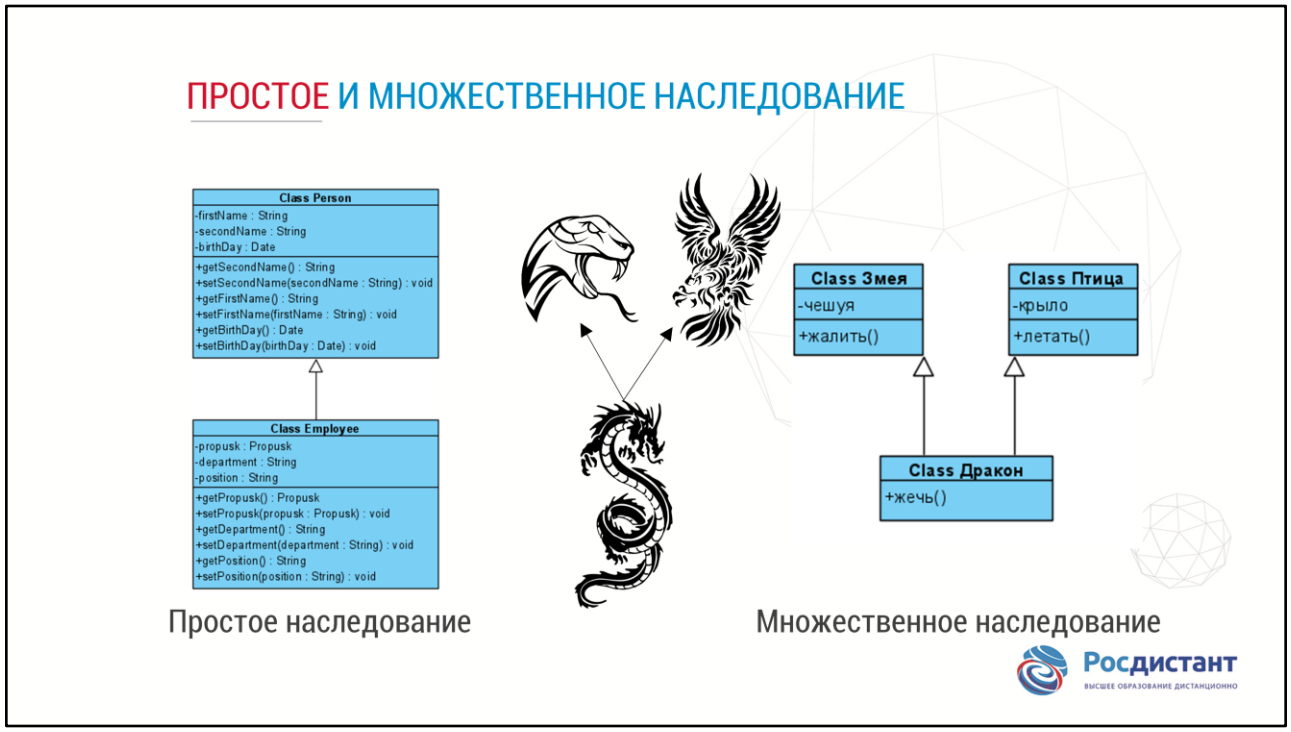
Вторым важнейшим принципом ООП является способность классов образовывать иерархии наследования, когда дочерние, или производные, классы наследуют поля и методы родительских, или базовых, классов. Говорят, что дочерние классы расширяют родительские классы, добавляя новые поля или методы. На слайде класс A – родительский – основатель иерархии наследования, класс B – дочерний от класса A, а класс C – дочерний от классов B и A.

Принцип наследования позволяет не копировать код родительских классов и легко создавать новые классы, «умеющие» делать все то, что есть в родительских. Появление и реализация в языках программирования наследования позволили резко увеличить важнейший показатель качества программного обеспечения – коэффициент повторного использования кода. Это стало одним из факторов, позволяющих и в наше время создавать достаточно надежные программы, реализующие большой набор сложнейшего функционала.

Рассмотрим пример.

Пусть имеется класс Человек, в котором определены основные данные о

человеке – имя, фамилия, дата рождения и т. д. Определим дочерний класс Сотрудник, в который добавим поля Пропуск, Подразделение, Должность. Таким образом, теперь мы можем обращаться с экземплярами класса Сотрудник и как с экземплярами класса Человек, и как с экземплярами класса Сотрудник, узнавая, к примеру, его должность. Для экземпляра класса Человек запрашиваются имя, дата рождения, возраст. Доступ к полям классов осуществляется через суррогатные свойства – геттеры и сеттеры.



Простое и множественное наследование

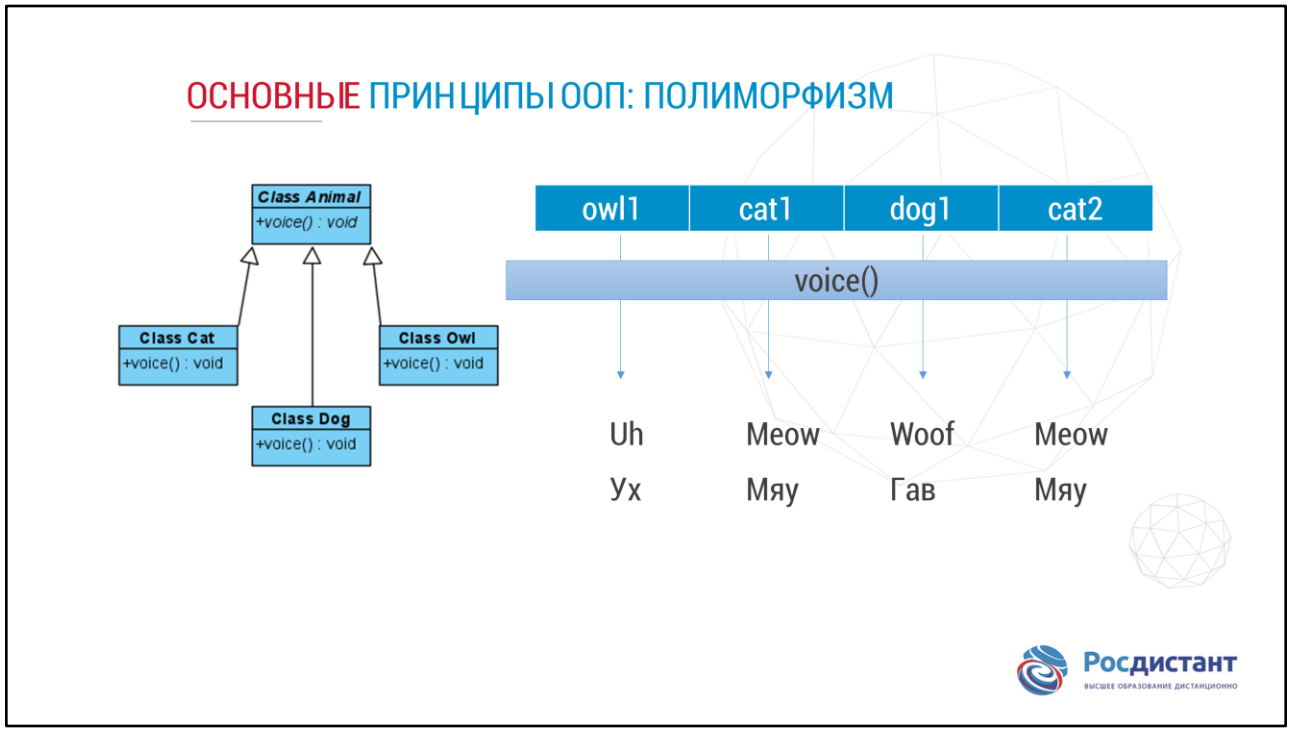
Наследование может быть простым и множественным.

При простом наследовании класс может иметь только одного родителя, а при множественном – двух и более.

Рассмотренный выше пример о классе Сотрудник – это пример простого наследования. Действительно, класс Сотрудник имеет только одного предка – Человек и, следовательно, может использовать поля и методы только одной иерархии наследования.

А как быть, если класс должен использовать члены классов из двух и более иерархий наследования?

Если мы, программируя, к примеру, компьютерную игру, реализуем класс Дракон, то можем реализовать наследование от двух классов: Змея и Птица. Таким образом, наш дракон будет иметь чешую, крылья, уметь летать и быть способным к ядовитому укусу. Остается только ввести новый метод пускать из пасти пламя и новый тип существ готов!

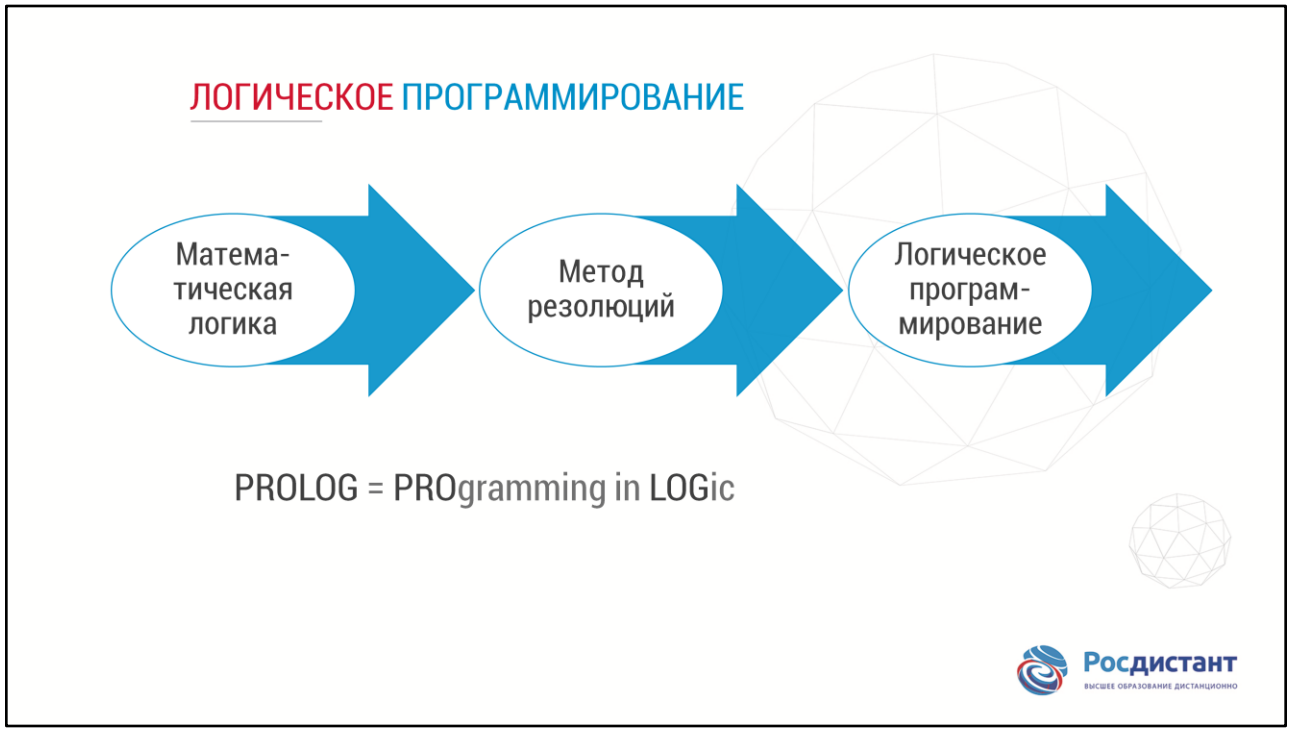


Основные принципы ООП: полиморфизм

Третьим важнейшим принципом объектно-ориентированного программирования является полиморфизм. Понятие полиморфизма в теории языков программирования очень обширно, однако в этом курсе мы рассмотрим его применение только относительно рассматриваемого объектно-ориентированного программирования.

Под полиморфизмом в ООП понимают свойство объектов в иерархии наследования объявлять и реализовывать полиморфные методы – методы, имеющие одинаковое название и параметры – сигнатуру, но реализующие себя по-разному.

Рассмотрим пример. Пусть имеется класс Животное, у которого объявлен метод «звук», т. е. животное может издавать звуки. Понятно, что каждое животное это делает по-разному. Например, кошка мяукает, собака лает, филин ухает и так далее. Во всех классах этот метод называется одинаково – «голос», но реализован по-разному. Например, объекты класса Кошка выводят строку «мяу», а класса Собака – «гав». Предположим, что мы создали массив или коллекцию из экземпляров всех перечисленных классов. Проходя по массиву, мы у каждого элемента вызываем метод «голос». Если эти методы полиморфные, то каждый элемент массива животных будет издавать свой голос.



Логическое программирование

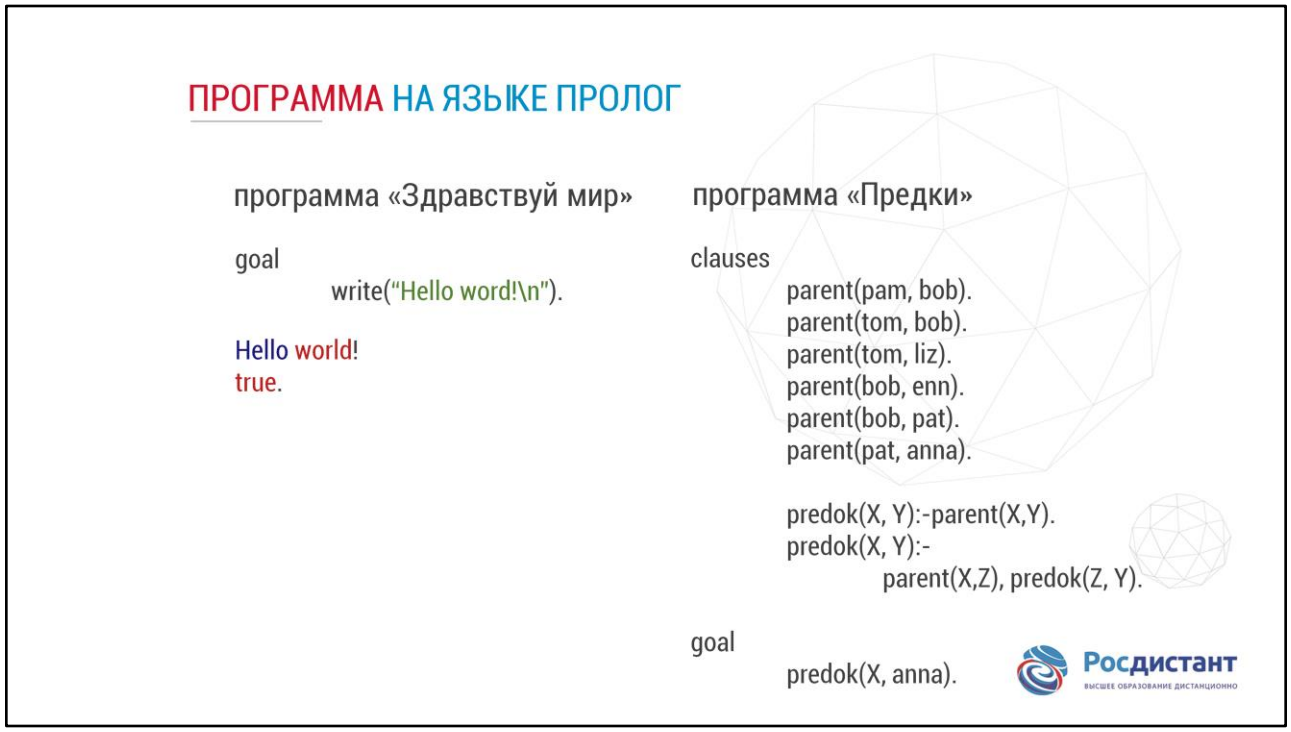
Логическое программирование – следующая парадигма программирования. Основой логического программирования является метод резолюций – раздел математической логики, изучающий методы доказательства правильности. Существует ряд языков логического программирования, но, пожалуй, самым известным считается язык Пролог и его потомки. Название языка PROLOG состоит из сокращения английской фразы «программирование в терминах логики». Пролог был создан французским ученым Аланом Кольмероэ в 1973 году.

Долгое время языки логического программирования использовались при разработке экспертных интеллектуальных систем. Однако в настоящее время в силу развития теории искусственного интеллекта массовое применение находят другие разделы, такие как нейросети, машинное и глубокое обучение.

Язык Пролог относится к декларативным языкам, то есть программа на языке Пролог описывает имеющиеся факты и отношения между ними в форме правил. Далее записывается утверждение, и программа на логическом языке программирования, основываясь на имеющихся правилах и фактах, пытается доказать правильность целевого высказывания. По сути, программа отвечает на вопрос, истина или ложна в указанных фактах и правилах цель, выводя в консоль «true» или «false». Однако можно «попросить» программу выводить на экран или базу данных пояснения, приведшие к результату.

Необходимо отметить, что никаких алгоритмов, привычных для массовых

языков, поддерживающих структурную парадигму, в языках логического программирования попросту не существует. Тут нет никаких циклов и ветвлений!



Программа на языке Пролог

Минимально программа на языке Пролог состоит из двух частей. Первая часть описывает факты и правила – предложения clauses. Вторая часть – раздел goal – цель – содержит единственную цель, которую надо попробовать доказать. Например, программа «Здравствуй мир» на Прологе не содержит ни одного факта или правила, а имеет единственную цель – доказать, что предикат вывода на консоль будет истинным. А так как предикат вывода всегда истинен, то и цель будет истинной. Пролог-программа, пытаясь доказать истинность цели, выведет на экран «Hello world» и, выполнив единственный истинный предикат, докажет истинность цели и выведет итоговое «true».

Вторая программа описывает известные отношения – кто кому является родителем. Например, первая запись говорит, что у Боба родитель Пам. После фактов описано правило Предок, говорящее, что предком является родитель или, рекурсивно, предок родителя. Цель – найти всех предков Анны. Пролог-программа, просматривая все факты и поставляя их в правило, определит, что предками Анны являются Пат, Боб, Том и Пам. При этом цель будет истинной.

Приведенные примеры показывают лишь простейшие программы на Прологе. Однако на Прологе и его модификации Visual Prolog можно писать очень эффективные программы, где используются рекурсия, большое количество переборов, особые виды структур данных, например, B+ деревья и т. п. Современные языки логического программирования позволяют писать узкоспециализированные алгоритмы и компилировать их в динамические

библиотеки, которые можно использовать при разработке программного обеспечения в качестве языков массового применения.

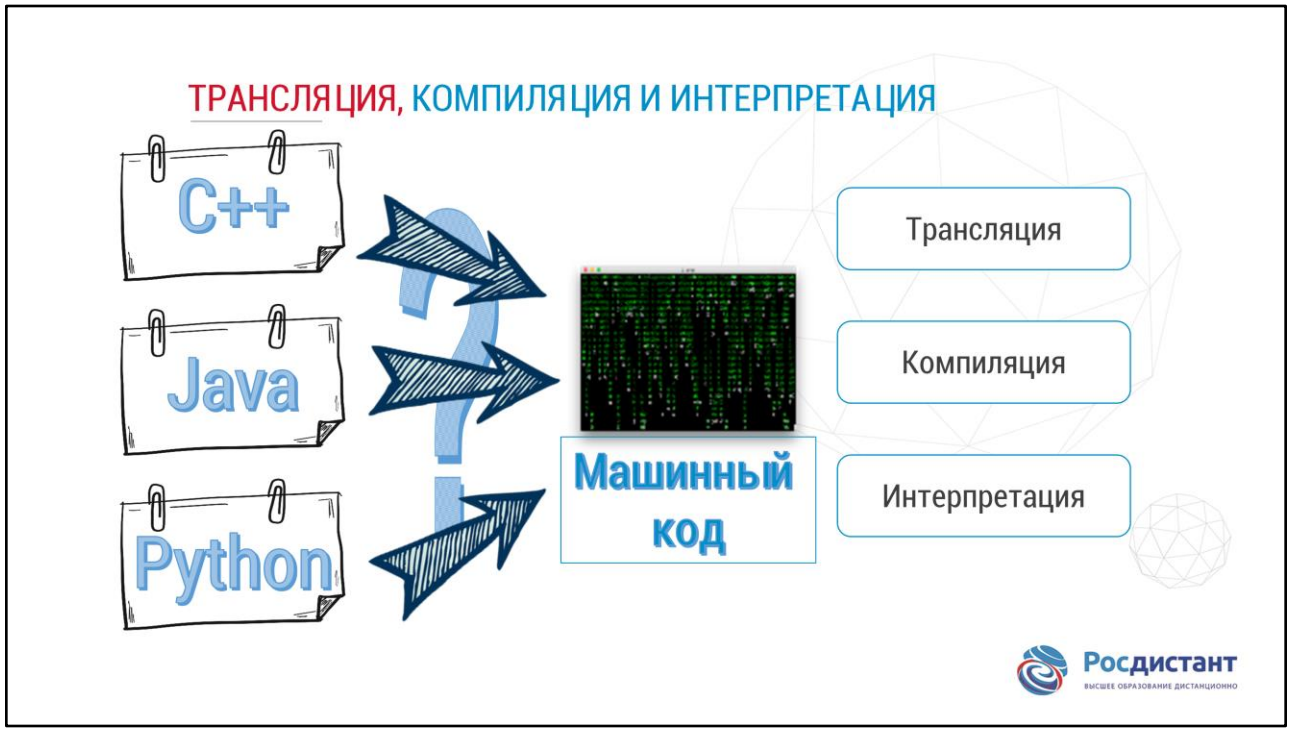
Функциональное программирование

Еще одной парадигмой программирования является функциональное, или аппликативное, программирование. В отличие от императивных и объектно-ориентированных языков, в функциональном программировании нет операторов, переменных, есть только функции и выражения над функциями. Математическую основу функционального программирования заложили исследования американского математика Алонзо Чёрча. Это привело к появлению лямбда-исчисления.

Первым функциональным языком программирования стал язык LISP, созданный в 1959 году. Впоследствии на основе этого языка появились другие языки программирования, такие как ML, Haskell, F#.

Первые языки программирования были разработаны для решения задач в основном числовой обработки. В них изначально практически отсутствовали механизмы обработки текстовой информации и списков. Впоследствии рост сложности автоматизируемых задач поставил новые требования к языкам программирования, касающиеся обработки именно текстовой информации. Поэтому совершенствование массовых языков программирования проходило также с учетом этого тренда.

Так, в современных версиях языков программирования, таких как Джава, Си Шарп, Пайтон, введены лямбда-выражения, представляющие собой программные объекты, написанные в функциональной парадигме.



Трансляция, компиляция и интерпретация

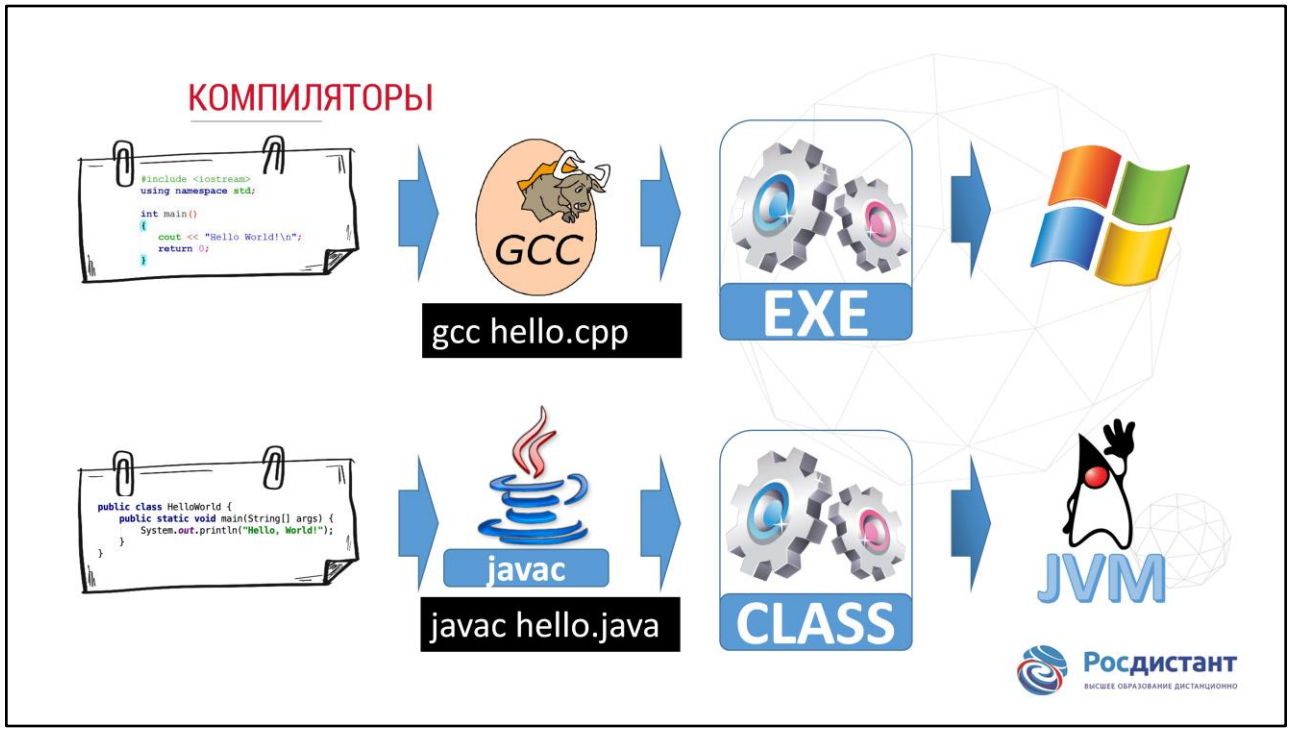
Как мы уже говорили ранее, современные языки программирования являются высокоуровневыми, а программы на этих языках пишутся в форме, приближенной к человеческому языку. Как правило, код программы хранится в файле в текстовом формате, и можно обычной программой просмотра открыть файл с кодом на языке C++, Python и т. д. Однако возникает вопрос, как перевести текст в машинные коды.

Для перевода текста программы в машинный код существует два способа: компиляция и интерпретация.

Прежде того, как мы рассмотрим понятие компиляции, введем понятие трансляции. Трансляция – это перевод программы с одного языка программирования на другой. Таким образом, мы можем транслировать программу с одного высокоуровневого языка на другой высокоуровневый или низкоуровневый. Трансляцию выполняют специальные программы, называемые трансляторами. Так, существуют трансляторы с языков высокого уровня на так называемый объектный язык, который является промежуточным между высокоуровневым и машинными языками.

Компиляция – это процесс перевода всего кода программы в машинный код. Таким образом, компиляция – это частный вид трансляции программы на языке высокого уровня в программу на машинном языке. Как правило, в процессе компиляции мы получаем исполняемый файл, который может быть запущен средствами операционной среды. Например, компилируя код для

операционной системы Windows, мы получаем файл с расширением «exe». Также с помощью компилятора мы можем получить файл динамической библиотеки и т. п.



Компиляторы

Компиляция выполняется с использованием специальной программы, которая называется компилятором.

Таким образом, для того чтобы скомпилировать код и получить исполняемую в операционной среде программу, надо запустить компилятор и передать ему в качестве аргумента файл для компиляции. Например, для языка C++ разработано много компиляторов, одним из которых является компилятор gcc, поэтому скомпилировать программу с именем hello.cpp можно следующей командой:

gcc hello.cpp.

Для одного языка программирования может существовать большое количество компиляторов, причем для каждой операционной системы должен быть свой компилятор. Например, если мы пишем код на C++ под Linux, нужен один компилятор, если под Windows, то другой.

Однако есть ряд языков программирования, программы на которых не выполняются непосредственно в среде операционной системы. Например, Java и C# предполагают наличие так называемых виртуальных машин, а программы, написанные на этих языках, компилируются не на машинный код, а на байт-код, который потом и запускается на виртуальной машине. Поэтому для приведенных языков программирования не стоит искать отдельные компиляторы для Windows или Linux, в этом случае надо устанавливать соответствующие виртуальные машины. Сам же процесс компиляции, к

примеру, на Java, запускается аналогично ранее рассмотренному, путем вызова компилятора javac.
В результате компиляции получим файл hello.class, который и будет содержать байт-код, выполняемый виртуальной машиной Java.

Интерпретаторы

Интерпретатор – это тоже программа, но, в отличие от компилятора, она переводит входную программу построчно и выполняет операторы, записанные в строке.

Таким образом, если компилятор берет весь код целиком и переводит его в исполняемый код, который далее может быть запущен, то интерпретатор берет строчку кода и, интерпретируя его, тут же выполняет.

В режиме интерпретации очень легко начать выполнение команды и сразу же вносить изменения в код, что принципиально нельзя сделать в компилируемых программах.

Однако, как правило, интерпретируемые программы работают значительно медленнее компилируемых.

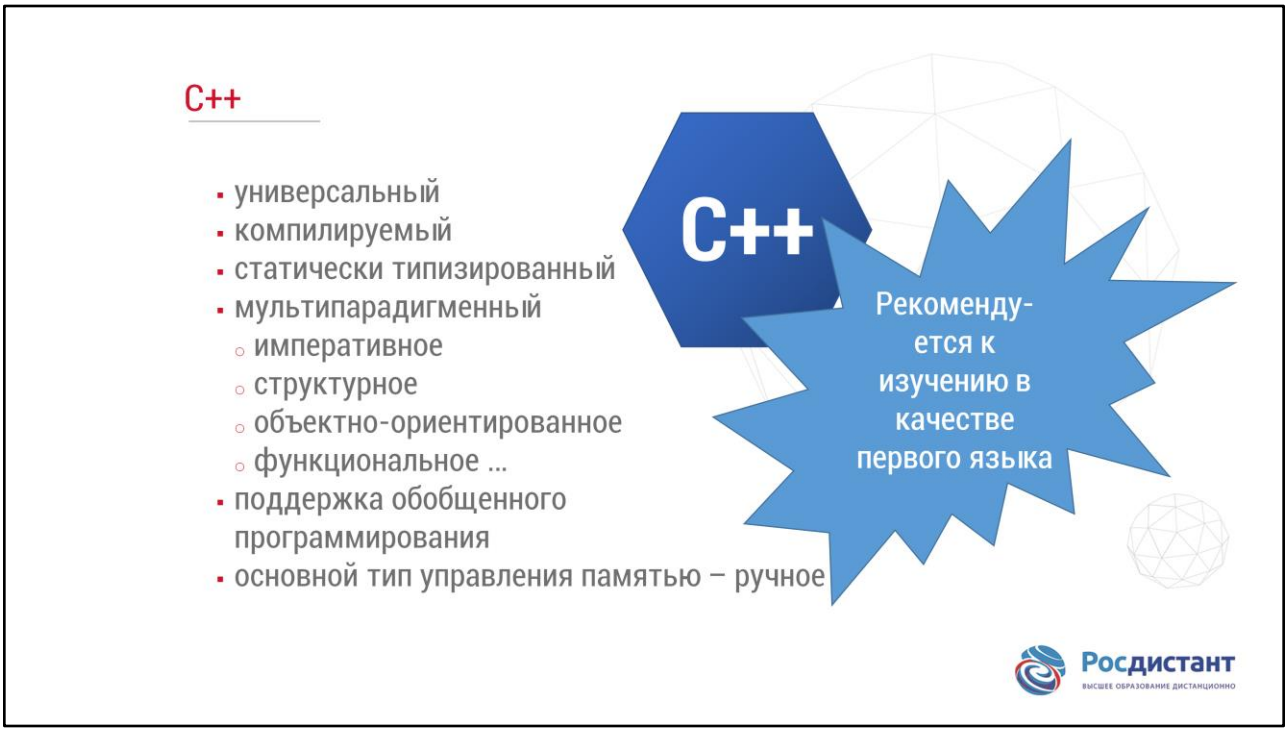
Примерами интерпретируемых языков являются Java, Python, Visual Basic for Application – VBA, JavaScript и многие другие.

Для устранения невысокой скорости работы в режиме интерпретации практически все перечисленные языки программирования имеют тот или иной вид компиляции.

Например, как мы говорили ранее, Java-код компилируется в байт-код, который во время выполнения переводится в машинные инструкции посредством JIT – Just in time компилятора.

В то же время режим интерпретации получил очень большое применение в обработке больших данных. Такие языки, как Python и R, использовались в

большой степени из-за режима интерпретации. Поэтому, начиная с версии 9, в Java появился «настоящий» интерпретатор jshell.
Классификацию языков программирования можно продолжать и по другим критериям: способу управления памятью, наличию типов, способов типизации, применимости к разработке многопоточных и параллельных приложений. Часть из этих классификаций будет рассмотрена позже в этом курсе.



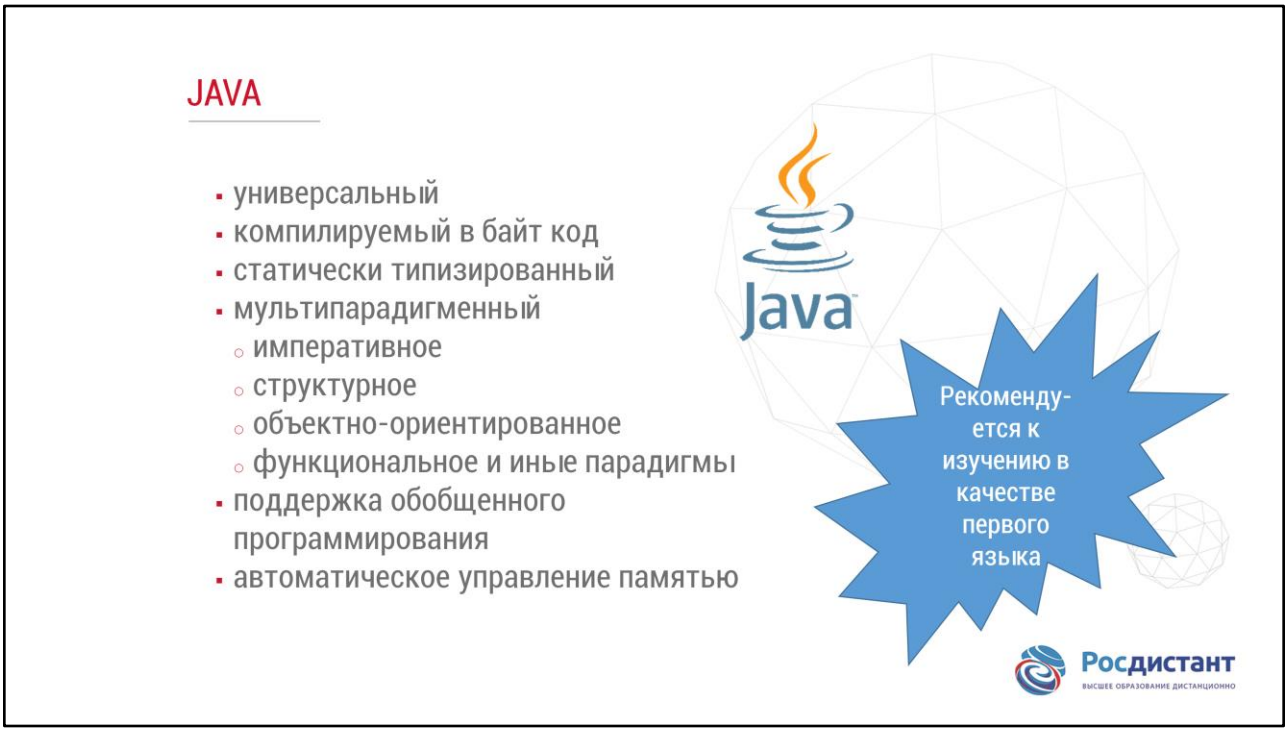
Си++

Дадим самый общий обзор языков программирования массового использования, или универсальных языков программирования. Мы не будем рассматривать эти языки досконально, поскольку такой анализ требует предварительных знаний в области теории программирования, компиляции, трансляции, архитектур вычислительных систем и многого другого.

Рассмотрим язык Си++.

Си++ – это универсальный, компилируемый, статически типизированный, мультипарадигменный язык программирования. Си++ поддерживает императивное, структурное, объектно-ориентированное, функциональное программирование и иные парадигмы программирования.

С++ – универсальный язык программирования, используемый для разработки как системного, так и прикладного программного обеспечения. Однако слабая типизация, наличие большого количества правил по умолчанию, использование указателей и, в основном, ручное управление памятью не способствуют написанию высоконадежных больших прикладных приложений. С++ сочетает низкоуровневые и высокоуровневые механизмы программирования. Появление в последних версиях новых механизмов, направленных на нивелирование рассмотренных недостатков, обеспечивает С++ многолетнее присутствие в верхней части рейтингов языков программирования.



Java

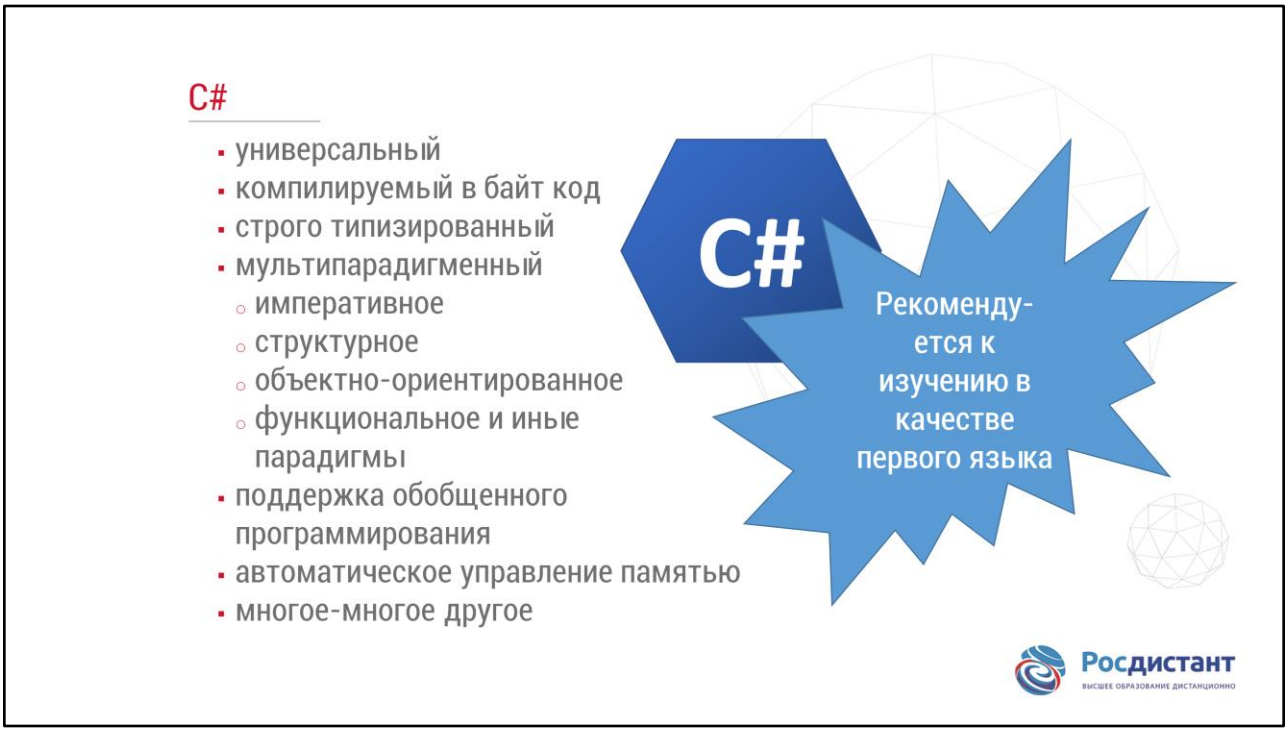
Язык программирования Java был разработан в 1995 году сотрудником корпорации Sun Microsystems Джеймсом Гослингом. При разработке языка программирования были заложены революционные для того времени принципы, обеспечившие присутствие Java в топ-рейтингах языков программирования на протяжении многих лет.

Java компилируется в байт-код, исполняемый виртуальной машиной Java. Джава – это универсальный, компилируемый в байт код, интерпретируемый в виртуальной машине, строго типизированный, мультипарадигменный язык. Джава поддерживает императивное, структурное, объектно-ориентированное, функциональное программирование и иные парадигмы программирования. Язык Java изначально рассматривался как язык для разработки прикладного программного обеспечения, в том числе распределенного, масштабируемого, переносимого, многопоточного. Реализация этих принципов позволило Java захватить и по сей день удерживать большой сегмент рынка программного обеспечения, особенно серверного.

Однако с появлением специальных процессоров и переносимых компьютеров, Джава стал активно использоваться для программирования так называемых внедряемых устройств: специализированных компьютерных карт, принтеров, банкоматов и т. д.

Как правило, приложения, написанные на Java, требуют бóльших ресурсов, чем приложения, написанные на C++, но обеспечивают очень высокие показатели

надежности. Поэтому, как правило, на Java пишутся крупные серверные приложения.



C#

Язык программирования C# был разработан на рубеже 2000 годов в корпорации Microsoft для платформы Microsoft.NET Framework.

C# имеет Си-подобный синтаксис, при разработке включил в себя преимущества многих языков программирования, таких как Java, Delhi, Модула и других.

Как и Java, C# компилируется в байт-код для исполняемой среды CLR.

Си Шарп – это универсальный, компилируемый в байт код, строго типизированный, мультипарадигменный язык программирования. Си Шарп поддерживает императивное, структурное, объектно-ориентированное, функциональное программирование.

Си Шарп предназначен для разработки сложного, масштабируемого, в том числе серверного прикладного программного обеспечения. В настоящее время это самый распространённый язык разработки приложений для платформы Microsoft .NET Framework.

Надежность кода обеспечивается современными механизмами, такими как достаточно строгая статическая типизация, автоматическая сборка мусора и т. п.

В отличие от C++ и Java, включает в себя большое количество современных возможностей, что делает его одним из языков, настоятельно рекомендуемых к изучению.

До недавнего времени существовала проблема наличия хороших реализаций CLR для UNIX подобных операционных систем, которая к настоящему времени устранена.

Python

Язык Пайтон, разработанный в 1991 году, стал необычайно популярен уже через много лет после его создания.

По сравнению с C++, C#, Java является динамически типизированным языком, то есть тип переменной определяется на этапе выполнения программы.

Синтаксис своеобразен и минималистичен.

Python – это универсальный, интерпретируемый, динамически типизированный, мультипарадигменный язык программирования. Пайтон поддерживает императивное, структурное, объектно-ориентированное, функциональное программирование и иные парадигмы программирования.

Простота программирования и огромный набор библиотек обеспечили высочайшую популярность языка Пайтон. Наряду с языком R используется в приложениях анализа данных, в том числе больших данных.

Идеален для прототипирования приложений: многие команды используют его в стартапах для быстрой разработки прототипов.

Отсутствие типов и статической типизации, по некоторым мнениям, не способствуют разработке больших приложений. Однако на Пайтоне достаточно эффективно реализуются бэкенды небольшого и среднего размера.

Сравнение языков программирования

Приведем таблицу, сравнивающую наиболее популярные языки программирования.

В таблице представлен далеко не исчерпывающий набор характеристик, которыми должен обладать универсальный язык программирования.

Читабельность – субъективная характеристика, показывающая, насколько прост код для понимания и разбора написанных на языке программирования конструкций.

Быстродействие. Обобщающая характеристика, говорящая о том, что язык C++, являясь компилируемым языком и языком, обладающим низкоуровневыми механизмами, в среднем обеспечивает разработку более быстрых приложений. Хотя на практике проводились тесты, показывающие, что программы, написанные на Java и исполняемые на некоторых версиях виртуальных машин, выполнялись быстрее, чем скомпилированные C++ программы.

Безопасность – характеристика, показывающая способность разрабатывать безопасный с точки зрения устойчивости код. Как говорилось ранее, наличие в C++ указателей и правил по умолчанию не позволяет обеспечить при равных условиях требуемый уровень безопасности.

Строгая типизация, особенно статическая, способствует написанию высоконадежного кода, так как позволяет на ранней стадии, на этапе компиляции, обнаружить большой набор ошибок. Иными словами, при использовании таких языков, как C#, число ошибок, к примеру, на 1000 строк

кода при разработке прикладного программного обеспечения существенно меньше, чем на C++.

Несмотря на то, что все рассматриваемые языки программирования поддерживают объектно-ориентированное программирование, наличие тех или иных механизмов наделяет языки программирования некоторыми особенностями. Например, возможностями поддержки множественного наследования, использования интерфейсов и многими другими.

Параллелизм – важнейшая особенность современных языков программирования, обеспечивающая легкость и надежность разработки параллельных и многопоточных приложений. Исходя из того, что в современном мире даже в смартфонах процессоры уже давно являются многоядерными, параллелизм является важнейшей характеристикой языка программирования.

И наконец, автоматическое управление памятью. Ручное управление памятью – основной механизм работы с объектами на C++ – делает высоковероятной утечку памяти. Утечка памяти – это ситуация, когда память из-под программных объектов не освобождается или освобождается не полностью. Когда объекты создаются часто, а память от них не освобождается, это приводит к исчерпанию памяти и краху приложения. Как правило, эти ошибки трудно обнаружить.

Таким образом, мы видим, что ни один из языков программирования не может считаться универсальным. Поэтому программисту-профессионалу необходимо знать несколько языков программирования и использовать их с учетом области программирования.