# Введение в программирование

**Язык программирования** – формальная знаковая система, предназначенная для записи компьютерных программ. Язык программирования определяет набор лексических, синтаксических и семантических правил, задающих внешний вид программы и действия, которые выполнит исполнитель (компьютер) под ее управлением.

Современные цифровые компьютеры являются двоичными и данные хранят в двоичном (бинарном) коде (хотя возможны реализации и в других системах счисления). Эти данные как правило отражают информацию из реального мира (имена, банковские счета, измерения и др.), представляющую высокоуровневые концепции.

Особая система, по которой данные организуются в программе – это **система типов языка программирования**. Языки могут быть классифицированы как системы со статической типизацией и языки с динамической типизацией.

При **статической типизации** переменная, параметр программы, возвращаемое значение функции связывается с типом в момент объявления и тип не может быть изменён позже. Примеры статически типизированных языков – Ада, C++, Java, Паскаль. Статически-типизированные языки могут быть в дальнейшем подразделены на языки с обязательной декларацией, где каждая переменная и объявление функции имеет обязательное объявление типа, и языки с выводимыми типами.

При **динамической типизации** переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной. Таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов. Примеры языков с динамической типизацией – Python, Ruby, PHP, Perl, JavaScript, Lisp.

Далее рассмотрим достоинства и недостатки каждого из приёмов.

Язык программирования строится в соответствии с той или иной базовой моделью вычислений и парадигмой программирования.

**Парадигма программирования** – это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Несмотря на то, что большинство языков ориентировано на императивную модель вычислений, задаваемую фон-неймановской архитектурой ЭВМ, существуют и другие подходы. Можно упомянуть языки со стековой вычислительной моделью (Forth, Factor, Postscript и др.), а также функциональное (Lisp, Haskell, ML и др.) и логическое программирование (Prolog) и язык Рефал, основанный на модели вычислений, введённой советским математиком А. А. Марковым-младшим. В настоящее время также активно развиваются проблемно-ориентированные, декларативные и визуальные

Таблица 1: Сравнение статической и динамической типизации

| Статическая типизация | Динамическая типизация |
|---|---|
| + Хороша для написания сложного, но быстрого кода | + Упрощается написание несложных программ |
| + Многие ошибки исключаются уже на стадии компиляции. Хороша для написания сложного, но быстрого кода | - Не позволяет заметить при компиляции простые «ошибки по недосмотру» – требуется как минимум выполнить данный участок кода |
| - Тяжело работать с данными из внешних источников (например, с реляционными СУБД) | + Облегчается работа с СУБД, которые принципиально возвращают информацию в «динамически типизированном» виде |
| | + Иногда требуется работать с данными переменного типа (например, функция поиска подстроки возвращает позицию найденного символа или маркер «не найдено») |
| | - Снижение производительности из-за трат процессорного времени на динамическую проверку типа |
| | - Излишние расходы памяти на переменные, которые могут хранить «что угодно» |

языки программирования.

Далее более подробно рассмотрим, упомянутые выше и не только, парадигмы программирования.

Для **императивного программирования** характерно следующее:

1. в исходном коде программы записываются инструкции (команды);
2. инструкции должны выполняться последовательно;
3. при выполнении инструкции данные, полученные при выполнении предыдущих инструкций, могут читаться из памяти;
4. данные, полученные при выполнении инструкции, могут записываться в память.

Первыми императивными языками были **машинные инструкции (коды)** – команды, готовые к исполнению компьютером без каких-либо преобразований. В дальнейшем были созданы **ассемблеры**, и программы стали записывать на языках ассемблеров. **Ассемблер** – компьютерная программа, предназначенная для преобразования машинных инструкций, записанных в виде текста на языке, понятном человеку (языке ассемблера), в машинные инструкции в виде, понятном компьютеру (машинный код). Одной инструкции на языке ассемблера соответствовала одна инструкция на машинном языке. Разные компьютеры поддерживали разные наборы инструкций. Программы, записанные для одного компьютера, приходилось

заново переписывать для переноса на другой компьютер. Позднее были созданы языки программирования высокого уровня и **компиляторы** – программы, преобразующие текст с языка программирования на язык машины (машинный код). Одна инструкция языка высокого уровня соответствовала одной или нескольким инструкциям языка машины, и для разных машин эти инструкции были разными. Первым распространённым высокоуровневым языком программирования, получившим применения на практике, стал язык Fortran, разработанный Джоном Бэкусом в 1954 году.

**Декларативное программирование** — это парадигма программирования, в которой задаётся спецификация решения задачи, то есть описывается, что представляет собой проблема и ожидаемый результат. Противоположностью декларативного является императивное программирование, описывающее на том или ином уровне детализации, как решить задачу и представить результат.

Итак, что бы лучше понять разницу между императивной и декларативной парадигмой программирования, приведём несколько примеров.

Таблица 2: Задача приготовить яйчницу

| Императивный стиль | Декларативный стиль |
|---|---|
| 1. поставь сковородку на огонь;<br>2. возьми два яйца (куриных);<br>3. нанеси удар ножом по каждому;<br>4. вылей содержимое на сковородку;<br>5. выкинь скорлупу ... | 1. приготовь яичницу |

Заметим, что пример для императивного стиля в таблице 2 имеет примесь декларативного стиля т.к. исполнитель должен знать, что такое сковорода и яйца.

Для перехода от одной части кода к другой долгое время использовался оператор **goto** – оператор безусловного перехода. Оператор goto в языках высокого уровня является объектом критики, поскольку чрезмерное его применение приводит к созданию нечитаемого «спагетти-кода». Впервые эта точка зрения была отражена в статье Эдсгера Дейкстры «Доводы против оператора GOTO», который заметил, что качество программного кода обратно пропорционально количеству операторов goto в нём. Статья приобрела широкую известность как среди теоретиков, так и среди практиков программирования, в результате чего взгляды на использование оператора goto были существенно пересмотрены.

Данные сосмнения привели к созданию концепции **структурного программирования**. Данная парадигма программирования предложена в 1970-х годах Э. Дейкстрой и в её основе лежит цель представить программу в виде иерархической структуры блоков.

**Принципы структурного программирования**:

1. Следует отказаться от использования **оператора безусловного перехода goto**;

2. Любая программа строится из трёх базовых управляющих конструкций: последовательность, ветвление, цикл;

   > **Последовательность** – однократное выполнение операций в том порядке, в котором они записаны в тексте программы
   >
   > **Ветвление** – однократное выполнение одной из двух или более операций, в зависимости от выполнения заданного условия
   >
   > **Цикл** – многократное исполнение одной и той же операции до тех пор, пока выполняется заданное условие (условие продолжения цикла)

3. В программе базовые управляющие конструкции могут быть вложены друг в друга произвольным образом. Никаких других средств управления последовательностью выполнения операций не предусматривается;

4. Повторяющиеся фрагменты программы можно оформить в виде подпрограмм (процедур и функций). Таким же образом (в виде подпрограмм) можно оформить логически целостные фрагменты программы, даже если они не повторяются;

5. Каждую логически законченную группу инструкций следует оформить как блок;

   > **Блок** – это логически сгруппированная часть исходного кода, например, набор инструкций, записанных подряд в исходном коде программы. Понятие блок означает, что к блоку инструкций следует обращаться как к единой инструкции. Блоки служат для ограничения области видимости переменных и функций. Блоки

могут быть пустыми или вложенными один в другой. Границы блока строго определены.

6. Все перечисленные конструкции должны иметь один вход и один выход;

7. Разработка программы ведётся пошагово, методом «сверху вниз».

Далее поясним, что представляет **метод разработки «сверху вниз»**.

Сначала пишется текст основной программы, в котором, вместо каждого связного логического фрагмента текста, вставляется вызов подпрограммы, которая будет выполнять этот фрагмент. Вместо настоящих, работающих подпрограмм, в программу вставляются фиктивные части – заглушки, которые, говоря упрощенно, ничего не делают.

Если говорить точнее, заглушка удовлетворяет требованиям интерфейса заменяемого фрагмента (модуля), но не выполняет его функций или выполняет их частично. Затем заглушки заменяются или дорабатываются до настоящих полнофункциональных фрагментов (модулей) в соответствии с планом программирования. На каждой стадии процесса реализации уже созданная программа должна правильно работать по отношению к более низкому уровню. Полученная программа проверяется и отлаживается.

После того, как программист убедится, что подпрограммы вызываются в правильной последовательности (то есть общая структура программы верна), подпрограммы-заглушки последовательно заменяются на реально работающие, причём разработка каждой подпрограммы ведётся тем же методом, что и основной программы. Разработка заканчивается тогда, когда не останется ни одной заглушки.

Такая последовательность гарантирует, что на каждом этапе разработки программист одновременно имеет дело с обозримым и понятным ему множеством фрагментов, и может быть уверен, что общая структура всех более высоких уровней программы верна.

При сопровождении и внесении изменений в программу выясняется, в какие именно процедуры нужно внести изменения. Они вносятся, не затрагивая части программы, непосредственно не связанные с ними. Это позволяет гарантировать, что при внесении изменений и исправлении ошибок не выйдет из строя какая-то часть программы, находящаяся в данный момент вне зоны внимания программиста.

**Функциональное программирование** – парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании).

Теория, положенная в основу функционального подхода, родилась в 20-х – 30-х годах. В числе разработчиков математических основ функционального программирования можно назвать Моисея Шейнфинкеля и Хаскелла Карри, разработавших комбинаторную логику, и Алонзо Чёрча, создателя $\lambda$-исчисления. Лямбда-исчисление являются основой для функционального программирования, многие функциональные языки можно рассматривать как «надстройку» над ними.

Математические функции выражают связь между исходными данными и итоговым продуктом некоторого процесса. Процесс вычисления также имеет вход и выход, поэтому функция – вполне подходящее и адекватное

средство описания вычислений. Именно этот простой принцип положен в основу функциональной парадигмы и функционального стиля программирования. Функциональная программа представляет собой набор определений функций. Функции определяются через другие функции или рекурсивно через самих себя. При выполнении программы функции получают параметры, вычисляют и возвращают результат, при необходимости вычисляя значения других функций. На функциональном языке программист не должен описывать порядок вычислений. Нужно просто описать желаемый результат как систему функций.

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния (в отличие от императивного, где одной из базовых концепций является переменная, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма).

На практике отличие математической функции от понятия «функции» в императивном программировании заключается в том, что императивные функции могут опираться не только на аргументы, но и на состояние внешних по отношению к функции переменных, а также иметь побочные эффекты и менять состояние внешних переменных. Таким образом, в императивном программировании при вызове одной и той же функции с одинаковыми параметрами, но на разных этапах выполнения алгоритма, можно получить разные данные на выходе из-за влияния на функцию состояния переменных. А в функциональном языке при вызове функции с одними и теми же аргументами мы всегда получим одинаковый результат: выходные данные зависят только от входных.

**Основные принципы функционального программирования**:

1. Все функции – чистые

   Функция являются чистой, если они удовлетворяют двум условиям:

   (a) Функция, вызываемая от одних и тех же аргументов, всегда возвращает одинаковое значение.

   (b) Во время выполнения функции не возникают побочные эффекты.

2. Все функции – первого класса и высшего порядка

   Для того, чтобы функция была первоклассной, у неё должна быть возможность быть объявленной в виде переменной. Это позволяет управлять функцией как обычным типом данных и в то же время исполнять её. Функции высшего порядка же определяются как функции, принимающие другую функцию как аргумент или возвращающие функцию.

3. Переменные неизменяемы

   Нельзя изменить переменную после её инициализации.

4. Относительная прозрачность функций.

Если вы можете заменить вызов функции на возвращаемое значение, и состояние при этом не изменится, то функция относительно прозрачна.

5. Функциональное программирование основано на лямбда-исчислении

   (a) В лямбда-исчислении все функции могут быть анонимными, поскольку единственная значимая часть заголовка функции – это список аргументов

   (b) При вызове все функции проходят процесс каррирования. Он заключается в следующем: если вызывается функция с несколькими аргументами, то сперва она будет выполнена лишь с первым аргументом и вернёт новую функцию, содержащую на 1 аргумент меньше, которая будет немедленно вызвана. Этот процесс рекурсивен и продолжается до тех пор, пока не будут применены все аргументы, возвращая финальный результат. Поскольку функции являются чистыми, это работает.

В качестве примера давайте попробуем испечь императивный и функциональный пирог.

Таблица 3: Задача испечь пирог

| Императивный стиль | Функциональный стиль |
|---|---|
| 1. Разогрейте духовку до 175 градусов. Смажьте маслом и посыпьте мукой противень. В маленькой миске смешайте муку, пищевую соду и соль<br>2. В большой миске взбивайте масло, сахар-песок и коричневый сахар до тех пор, пока масса не станет легкой и воздушной. Вбейте яйца, одно за раз. Добавьте бананы и разотрите до однородной консистенции. Поочередно добавляйте в получившуюся кремовую массу основу для теста из п. 1 и кефир. Добавьте измельченные грецкие орехи. Выложите тесто в подготовленный противень<br>3. Запекайте в разогретой духовке 30 минут. Выньте противень из духовки, поставьте на полотенце, чтоб пирог остыл | 1. Пирог – это горячий пирог, остывший на полотенце, где горячий пирог – это подготовленный пирог, выпекавшийся в разогретой духовке 30 минут<br>2. Разогретая духовка – это духовка, разогретая до 175 градусов<br>3. Подготовленный пирог – это тесто, выложенное в подготовленный противень, где тесто—это кремовая масса, в которую добавили измельченные грецкие орехи. Где кремовая масса – это масло, сахар-песок и коричневый сахар, взбитые в большой миске до тех пор, пока они не стали легкими и воздушными, где ... |

**Логическое программирование** – парадигма программирования, основанная на автоматическом доказательстве теорем, а также раздел дискретной математики, изучающий принципы логического вывода информации на основе заданных фактов и правил вывода. Логическое программирование основано на теории и аппарате математической логики с использованием математических принципов резолюций. Самым известным языком логического программирования является Prolog.

Далее более подробно рассмотрим принцыпы объектно-ориентированного програмирования. Данный подход позволяет моделировать информационные объкекты и рещает на новом уровне основную задачу структурного программирования: структурирование информации с точки зрения управляемости, что существенно улучшает управляемость самим процессом моделирования, что в свою очередь особенно важно при реализации крупных проектов.

**Объектно-ориентированное программирование (ООП)** – методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

**Ключевые черты ООП**:

1. **Инкапсуляция** – это определение классов – пользовательских типов данных, объединяющих своё содержимое в единый тип и реализующих некоторые операции или методы над ним
2. **Наследование** – способ определения нового типа, когда новый тип наследует элементы (свойства и методы) существующего, модифицируя или расширяя их
3. **Полиморфизм** позволяет единообразно ссылаться на объекты различных классов (обычно внутри некоторой иерархии). Это делает классы ещё удобнее и облегчает расширение и поддержку программ, основанных на них

Появление в ООП отдельного понятия класса закономерно вытекает из желания иметь множество объектов со сходным поведением. Класс в ООП – это в чистом виде абстрактный тип данных, создаваемый программистом. С этой точки зрения объекты являются значениями данного абстрактного типа, а определение класса задаёт внутреннюю структуру значений и набор операций, которые над этими значениями могут быть выполнены. Желательность иерархии классов (а значит, наследования) вытекает из требований к повторному использованию кода – если несколько классов имеют сходное поведение, нет смысла дублировать их описание, лучше выделить общую часть в общий родительский класс, а в описании самих этих классов оставить только различающиеся элементы.

Необходимость совместного использования объектов разных классов, способных обрабатывать однотипные сообщения, требует поддержки полиморфизма – возможности записывать разные объекты в переменные одного и того же типа. В таких условиях объект, отправляя сообщение, может не знать в точности, к какому классу относится адресат, и одни и те же сообщения, отправленные переменным одного типа, содержащим объекты разных классов, вызовут различную реакцию.

Как уже говорилось выше, в современных объектно-ориентированных языках программирования каждый объект является значением, относящимся к определённому классу. Класс представляет собой объявленный программистом составной тип данных, имеющий в составе:

1. Поля данных

   Параметры объекта, задающие его состояние. Фактически поля представляют собой значения (переменные, константы), объявленные как принадлежащие классу.

2. Методы

   Процедуры и функции, связанные с классом. Они определяют действия, которые можно выполнять над объектом такого типа, и которые сам объект может выполнять.

Классы могут наследоваться друг от друга. Класс-потомок получает все поля и методы класса-родителя, но может дополнять их собственными либо переопределять уже имеющиеся. Большинство языков программирования поддерживает только единичное наследование (класс может иметь только один класс-родитель), лишь в некоторых допускается множественное наследование – порождение класса от двух или более классов-родителей. Множественное наследование создаёт целый ряд проблем, как логических, так и чисто реализационных, поэтому в полном объёме его поддержка не распространена. Вместо этого в 1990-е годы появилось и стало активно вводиться в объектно-ориентированные языки понятие интерфейса. **Интерфейс** – это класс без полей и без реализации, включающий только заголовки методов. Если некий класс наследует (или, как говорят, реализует) интерфейс, он должен реализовать все входящие в него методы. Использование интерфейсов предоставляет относительно дешёвую альтернативу множественному наследованию.

Взаимодействие объектов в абсолютном большинстве случаев обеспечивается вызовом ими методов друг друга.

Инкапсуляция обеспечивается следующими средствами:

1. Контроль доступа

   Поскольку методы класса могут быть как чисто внутренними, обеспечивающими логику функционирования объекта, так и внешними, с помощью которых взаимодействуют объекты, необходимо обеспечить скрытость первых при доступности извне вторых. Для этого в языки вводятся специальные синтаксические конструкции, явно задающие область видимости каждого члена класса. Традиционно это модификаторы public, protected и private, обозначающие, соответственно, открытые члены класса, члены класса, доступные внутри класса и из классов-потомков, и скрытые, доступные только внутри класса. Конкретная номенклатура модификаторов и их точный смысл различаются в разных языках.

2. Методы доступа

Поля класса в общем случае не должны быть доступны извне, поскольку такой доступ позволил бы произвольным образом менять внутреннее состояние объектов. Поэтому поля обычно объявляются скрытыми (либо язык в принципе не позволяет обращаться к полям класса извне), а для доступа к находящимся в полях данным используются специальные методы, называемые методами доступа. Такие методы либо возвращают значение того или иного поля, либо производят запись в это поле нового значения. При записи метод доступа может проконтролировать допустимость записываемого значения и, при необходимости, произвести другие манипуляции с данными объекта, чтобы они остались корректными (внутренне согласованными). Методы доступа называют ещё аксессорами (от англ. access – доступ), а по отдельности – геттерами (англ. get – чтение) и сеттерами (англ. set – запись).

3. Свойства объекта

   Псевдополя, доступные для чтения и/или записи. Свойства внешне выглядят как поля и используются аналогично доступным полям (с некоторыми исключениями), однако фактически при обращении к ним происходит вызов методов доступа. Таким образом, свойства можно рассматривать как «умные» поля данных, сопровождающие доступ к внутренним данным объекта какими-либо дополнительными действиями (например, когда изменение координаты объекта сопровождается его перерисовкой на новом месте). Свойства, по сути, не более чем синтаксический сахар, поскольку никаких новых возможностей они не добавляют, а лишь скрывают вызов методов доступа. Конкретная языковая реализация свойств может быть разной.

Полиморфизм реализуется путём введения в язык правил, согласно которым переменной типа «класс» может быть присвоен объект любого класса-потомка её класса.

Отметим, что парадигма программирования не определяется однозначно языком программирования; практически все современные языки программирования в той или иной мере допускают использование различных парадигм. Python поддерживает несколько парадигм программирования, в том числе императивное, структурное, функциональное и объектно-ориентированное. Основные архитектурные черты – динамическая типизация, автоматическое управление памятью, механизм обработки исключений, поддержка многопоточных вычислений и удобные высокоуровневые структуры данных. Код в Python организовывается в функции и классы, которые могут объединяться в модули (они в свою очередь могут быть объединены в пакеты).

В конце рассмотрим способы реализации языков. Языки программирования могут быть реализованы как **компилируемые** и **интерпретируемые**.

Программа на компилируемом языке при помощи специальной программы компилятора преобразуется (компилируется) в набор инструкций для данного типа процессора (машинный код) и далее записывается в исполнимый модуль, который может быть запущен на выполнение как отдельная программа. Другими словами, компилятор переводит исходный текст

программы с языка программирования высокого уровня в двоичные коды инструкций процессора.

Если программа написана на интерпретируемом языке, то интерпретатор непосредственно выполняет (интерпретирует) исходный текст без предварительного перевода. При этом программа остаётся на исходном языке и не может быть запущена без интерпретатора. Можно сказать, что процессор компьютера – это интерпретатор машинного кода.

Кратко говоря, компилятор переводит исходный текст программы на машинный язык сразу и целиком, создавая при этом отдельную машинно-исполняемую программу, а интерпретатор выполняет исходный текст прямо во время исполнения программы («интерпретируя» его своими средствами).

Разделение на компилируемые и интерпретируемые языки является условным. Так, для любого традиционно компилируемого языка, как, например, Pascal, можно написать интерпретатор. Кроме того, большинство современных «чистых» интерпретаторов не исполняют конструкции языка непосредственно, а компилируют их в некоторое высокоуровневое промежуточное представление (например, с разыменованием переменных и раскрытием макросов). Для любого интерпретируемого языка можно создать компилятор – например, язык Lisp, изначально интерпретируемый, может компилироваться без каких бы то ни было ограничений. Создаваемый во время исполнения программы код может так же динамически компилироваться во время исполнения.

Как правило, скомпилированные программы выполняются быстрее и не требуют для выполнения дополнительных программ, так как уже переведены на машинный язык. Вместе с тем, при каждом изменении текста программы требуется её перекомпиляция, что создаёт трудности при разработке. Кроме того, скомпилированная программа может выполняться только на том же типе компьютеров и, как правило, под той же операционной системой, на которую был рассчитан компилятор. Чтобы создать исполняемый файл для машины другого типа, требуется новая компиляция.

Интерпретируемые языки можно запускать сразу же после изменения, что облегчает разработку. Программа на интерпретируемом языке может быть зачастую запущена на разных типах машин и операционных систем без дополнительных усилий. Однако интерпретируемые программы выполняются заметно медленнее, чем компилируемые, кроме того, они не могут выполняться без дополнительной программы-интерпретатора.

Python является интерпретируемый, компилируемый в MSIL, компилируемый в байт-код Java.

## Список литературы


[1] Т. Пратт, М. Зелковиц. Языки программирования: разработка и реализация. 4-е изд. СПб.:Питер, 2003

[2] Дал У., Дейкстра Э., Хоор К. Структурное программирование. – М.: Мир, 1975

[3] Барендрегт Х. Ламбда-исчисление. Его синтаксис и семантика: Пер. с англ. — М.: Мир, 1985

[4] Филд А., Харрисон П. Функциональное программирование. М.: Мир, 1993

[5] Хендерсон П. Функциональное программирование. Применение и реализация. М.: Мир, 1983

[6] Джонс С., Лестер Д. Реализация функциональных языков. М.: Мир, 1991

[7] Анатолий Адаменко, Андрей Кучуков. Логическое программирование и Visual Prolog (с CD). – СПб.: «БХВ-Петербург», 2003

[8] Иан Грэхем. Объектно-ориентированные методы. Принципы и практика. – 3-е изд. – М.: «Вильямс», 2004