**ПРОГРАММА**

Вся информация хранится в файлах. По сути **файл** — **это некоторая область на жестком диске, которая является логическим целым.** Именно логическим — ваш файл может быть разбросан по нескольким секциям жесткого диска (или иного носителя, например флешки или компакт-диска). Специальная программа — операционная система — управляет файлами на диске. А человек (и другие программы) видят файл как одно целое. И могут его копировать, переносить и даже удалять.

Так вот программа — **это просто файл с набором команд**, которые могут быть выполнены процессором. Запускает программу обычно операционная система — она считывает программы из файла, загружает в память и запускает.

Каждая программа при своем создании проходит простой цикл — **написание текста, компиляция, запуск и проверка работоспособности**. Большинство программ — это обычный текст, который можно набирать в том же приложении «Блокнот» в Windows (или еще каком-либо простом текстовом редакторе). В большие программы конечно же подключаются картинки, видео, звук и прочие ресурсы, но тем не менее основным элементом программ является обычный текст, который описывает порядок выполнения вашей программы. Шаг за шагом.

Компиляция — это перевод текста программы в набор кодов, которые понятны процессору. Вы наверно не будете удивлены тем, что программы, которые исполняются на CPU (central processing unit, CPU, дословно — центральное обрабатывающее устройство, часто просто процессор) явно не текстовые. Они представляю из себя набор байтов (думаю, что о байтах вы что-то слышали) состоящие из нулей и единиц (и о двоичном коде возможно тоже что-то слышали). Каждая комбинация является командой или данными, которые обрабатывает CPU. Так вот много лет назад программы для компьютеров писались именно так — в память забивали набор байтов, состоящих из нулей и единиц и стартовали программу. Внешний вид такого чуда мысли программиста был мягко говоря слабо читаем. Хотя люди с опытом могли творить просто чудеса. Тем не менее такое кодирование имело еще одну проблему — надо было все делать самому. Где-то в памяти хранить данные и где-то хранить команды. Их надо было разделять, и ими надо было управлять И все это делалось вручную — т.е. программист был с одной стороны почти богом (мог делать все, что хочет), с другой стороны он был вынужден делать титаническую работу по разбиению программы на мелкие-мелкие детали. Как частенько говорят — «из-за деревьев леса не видно». Написать программу обработки текста — это была проблема. В итоге инженеры пришли к нормальной идее, что можно создать так называемые алгоритмические языки программирования. Идея достаточно простая — некоторые конструкции при написании алгоритмов являются стандартными и их на самом деле не так уж и много — мы это еще увидим. Конструкции стандартные — значит и перевод их с понятного представления в машинное тоже будет стандартным. Т.е. можно написать программу, которая будет «переводить» текст на алгоритмическом языке в машинный. И уже полученный результат можно исполнять на CPU. Правда тут возникла еще одна особенность — процесс компиляции может выдавать ошибки, если программа содержит некорректные с точки зрения компилятора конструкции. Но алгоритмические языки резко увеличили скорость программирования. Компилятор сам распределяет память для данных, команд — это огромное подспорье.

С этого и началась эра алгоритмических языков — люди писали программы в более-менее понятной форме, в виде текста. Компилятор их преобразовывал в машинный код, который можно было исполнять.

Стоит отметить. что также существуют программы-интерпретаторы. В отличии от компилируемых языков, которые на выходе получают сразу готовый машинный код, интерпретаторы получают на вход текст программы и пытаются ее выполнить «на лету». Т.е. каждая строка или команда сразу преобразуется в машинный код и выполняется. Это удобно — не надо компилировать — но с другой стороны это замедляет работу программы. Для каждой команды надо выполнить преобразование в машинный код, а потом выполнить этот машинный код. Долго, но зато удобно.

**Java Virtual Machine**

Java — это тоже компилируемый язык программирования. И как у других подобных языков, у него тоже есть стадии написания текста, компиляции и запуска. Но в отличии от языков типа C, Pascal и прочих, программа на Java не сразу компилируется в машинный код. После компиляции создается так называемый байт-код. И этот байт-код выполняется специальной программой — Java Virtual Machine, JVM (виртуальная машина Java). Зачем это надо ? Идея очень проста и логична. Каждый тип процессора имеет свой набор команд, что означает, что один и тот же текст на алгоритмическом языке будет преобразовываться в разные машинные коды в зависимости от того, под какой процессор вы компилируете вашу программу. Да и сам компилятор тоже должен существовать под конкретный процессор. И даже если вы сами написали нужный компилятор — надо иметь под рукой исходный текст. А его вам могут и не дать. Он денег стоит.

Из всего этого следует, что компилировать программу под каждый процессор, и даже под каждую операционную систему отдельно требует больших усилий. В компании Sun Microsystems пошли по следующему пути: решили, что программы на Java будут выполняться под управлением специальной программы, **по сути на виртуальном процессоре**, который имеет совершенно определенный набор команд. И каждая программа на Java может быть выполнена по сути на любом процессоре, на любой операционной системе под которую написана JVM. Таким образом один раз скомпилированная программа на Java будет по идее выполняться где угодно. Девиз Sun был именно таким: Write once run anywhere — написано однажды, запускается везде.

В итоге программа на Java тоже проходит стадию написания текста, компиляции и запуска. Но запускается она под управлением JVM, которая интерпретирует байт-код. вы вполне законно можете подумать, что раз «интерпретирует» — значит работает медленнее. Да, в некоторых случаях это так. Но и тут разработчики Sun смогли найти решение. Был изобретен JIT-компилятор (JIT — Just In Time). Этот механизм позволяет преобразовать байт-код в машинный код при первом проходе программы и в дальнейшем выполнять просто готовый машинный код. Т.е. появилась еще одна стадия — компиляция байт-кода в машинный код. Это позволяет существенно ускорить работу программы. Скорость работы программы на Java вполне сопоставима со скоростью работы программы на том же C или Pascal. Так что страшные истории про жуткие тормоза программ на Java — это только истории и больше ничего. Хотя относительно графических программ можно сказать, что Java уступает в быстродействии.

**Программа на Java — шаги по запуску программы**

Кроме того, что программу надо написать, программу надо еще суметь запустить. Так что программисты постоянно делают один и тот же цикл из четырех шагов:

1)-Редактировать код программы

2)-Собрать/скомпилировать программу из текста в код, понятный JVM — байт-код. Опять же — надо исправлять ошибки компилятора. Тогда возвращаемся к первому шагу.

3)-Если все скомпилировалось — запустить программу под управлением JVM

Посмотреть корректность работы программы. Если даже программа запустилась — не факт, что вы совершенно верно реализовали алгоритм. Нередко надо опять идти к первому шагу.

Если на каком-то шаге произошла ошибка, то повторяем все с самого начала. Если все хорошо, то цикл можно завершать.

**Установка Java Development Kit (JDK)**

JDK — это набор программ для разработки и запуска программ на Java. Для нас самыми главными будут две программы:

**javac** — компилятор файлов с программами, написанными на java. У таких файлов обычно расширение .java. Например First.java. После успешной компиляции появляется файл с расширением .class. Т.е. если мы скомпилируем First.java, то должен появиться файл First.class. По большому секрету файлов .class может быть несколько, но мы этот вопрос разберем, .когда придет время.

**java** — по сути это программа, которая и является JVM. Именно она исполняет **байт-код. который содержится в файле .class**

**Данные**

Давайте более подробно остановимся на том, откуда же можно брать данные. Ответ на этот вопрос дает нам список тех технологий (пакетов, подпрограмм), которые используются в современных языках программирования (в том числе и в Java). Итак:

1)- Файлы. Достаточно очевидное хранилище данных. Наверняка многие из вас создавали документы в программе «Блокнот», MS Word, Excel и многих других. Каждый документ сохраняется в файле. Это и есть наши внешние данные. т.е. они находятся вне программы. Совсем необязательно, что ваши файлы будут содержать столь сложные структуры, как документ в формате Word, но тем не менее нахождение данных в файле — важный момент.

2)- Внешние программы — это не только программы, которые запущены на Вашем компьютере — это программы, которые запущены на других компьютерах (хотя может быть и на том же самом — но это ДРУГИЕ программы). В эру сети Интернет взаимодействие программ становится очень важным элементом и возможности взаимодействия программ необходимо изучать. Разумеется в Java и в других современных языках есть целый ряд технологий, позволяющих осуществлять такое взаимодействие.

3)- Базы данных. Если быть точным, то работа с базами данных может быть включена как в раздел «Файлы», так и в раздел «Внешние программы». Базы данных могут быть организованы в виде простых файлов или в виде специальной программы, которая обрабатывает внешние запросы от других программ (такая организация сегодня является наиболее предпочтительной даже в случае, если база данных используется локально).

1. Цикл for

for(<инициализация>; <условие>; <последействие>) операция;

2. Цикл с предусловием while

while(<условие>) операция; Пока условие является истинным — цикл выполняется

3. Цикл с постусловием do .. while

do операция; while(<условие>); Т.е. делать операцию до тех пор, пока условие не станет ложным

Парадигмы ООП

**Инкапсуляция**

Не пытайтесь найти перевод этого термина с английского — вы получите encapsulation. Зато если вы попробуйте присмотреться, вы увидите главное — «капсула». **Каждый объект не должен выставлять наружу все свои параметры для изменения просто так**. Например, если у нашего робота есть координаты X и Y, то не вызывает сомнений факт, что их нельзя менять прямо. Робот должен поменять свои координаты в результате передвижения. Нельзя обратиться к переменной X внутри объекта Robot и сделать самое простое присваивание. Это будет как минимум нелогично. Лучше такого вообще не позволять. Т.е. мы таким образом должны создавать описание класса, чтобы нельзя было просто так получать доступ к его внутренним переменным. Это будет похоже на то, как если бы мы при производстве телевизора давали людям доступ ко всей схеме и каждый мог переключать проводки внутри него напрямую. Обычный человек таким телевизором вряд ли бы пользовался. Конечно нашлось бы несколько энтузиастов, которые обрадовались такому положению дел. Но это скорее всего экзотика. Более правильно в случае с роботом было сделать так, чтобы можно было получить значения координат. А менять координаты можно было бы только в результате движения робота. Проехал он 10 метров — поменялись его координаты X и Y в соответствии с расстоянием и курсом. Причем автоматически.

Т.е. **закрытие внутренних переменных — хорошая идея**. Для правильного поведения объекта. Нам самим будет проще — при условии, что класс более-менее разумно спроектирован и реализован. Достигается в том числе благодаря модификаторам доступа: **private, protected, public.**

**Наследование**

**Переопределение (override)**

Переопределение используется тогда, когда вы переписываете (переделываете, переопределяете) УЖЕ существующий метод. Ниже пример где переопределён метод toString. Данный класс наследуется,конечно-же, от Object у которого есть toString, который и переопределяем.

public class Robot{

private String name;

public Robot(String name) {

this.name = name;

}

@Override

public String toString() {

return "name=" + name;

}

}

@Override. Называется эта конструкция «аннотация». Служит для включения дополнительной информации, которую можно прочитать и использовать. Пока просто примем к сведению, что методы, которые вы ПЕРЕОПРЕДЕЛЯЕТЕ, лучше предварять этой аннотацией. В этом случае компилятор получает возможность проверить, что вы переопределили метод, а не написали новый. Таким образом можно избежать некоторых ошибок из-за невнимательности.

**Перегрузка (overload)**

Перегрузка метода заключается в следующем — вы создаете метод с таким же именем, но с другим набором параметров. Например, в классе может быть несколько методов с названием summa, но с разным набором парметров.

public double summa(double x1, double x2) {

return x1 + x2;

}

public double summa(double x1, double x2, double x3) {

return x1 + x2 + x3;

}

public double summa(double x1, double x2, double x3, double x4) {

return x1 + x2 + x3 + x4;

}

перегрузить можно любой метод, в том числе и конструктор

—----—----------------------------------------------------------------**интерфейс**—---------------------------------------------------------------

Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. или интерфейс - это описание голой функциональности без каких либо привязок к особенностям класса. Если выражаться немного образно, то классы получили возможность иметь профессии — отправитель почты, управляющий транзакциями, распределитель страниц, контроллер и т.д.

Один класс может применить множество интерфейсов.Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации. Методы без реализации похожи на абстрактные методы абстрактных классов.Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ public.

Если класс применяет/реализует интерфейс, то он должен реализовать все методы интерфейса.Нельзя напрямую создавать объекты интерфейсов

**Полиморфизмом**

называется возможность работать с несколькими типами так, как будто это один и тот же тип и в то же время поведение каждого типа будет уникальным в зависимости от его реализации

—-----------------------------------------------------------**слово** **FINAL**—----------------------------------------------------

Применяя определение final к методу вы запрещаете переопределять этот метод в классах-потомках

Если же вы хотите совсем запретить расширения важного для вас класса, то при его определении вы можете использовать final.

public final class TestClass{

public String getFinalName() {

return "Final name for TestClass";

}

В Java достаточно много таких классов — Integer, String, Double.

—--------------------------------------------------------------**Интерпретация** —-----------------------------------------------------------------

пооператорный (покомандный, построчный) анализ, обработка и тут же выполнение исходной программы или запроса (в отличие от компиляции, при которой программа транслируется без её выполнения).

Язык Java обладает как компилятором (javac), так и интерпретатором, в роли которого выступает виртуальная машина, которая построчно преобразует байт-код в машинный код и тут же его исполняет. Таким образом, когда мы запускаем скомпилированную программу, виртуальная машина начинает её интерпретацию, **то есть построчное преобразование байт-кода в машинный код, а также его исполнение.**

К сожалению, чистая интерпретация байт-кода является довольно долгим процессом и делает язык java медленным в сравнении с его конкурентами. Дабы избежать этого, был введен механизм, позволяющий ускорить интерпретацию байт-кода виртуальной машиной. Этот механизм называется Just-in-time компиляцией (JITC),- преобразует наиболее часто выполняемые участки программы в машинный код и хранит их в памяти. что в какой то мере ускоряет выполнение но требует больше памяти (где то ж надо хранить часто выполняемые участки кода).

Отношения между классами (объектами)

Теория ООП выделяет три основных отношения между классами:

Ассоциация

Агрегация и композиция

Обобщение/Расширение (наследование)

**Ассоциация** означает, что объекты двух классов могут ссылаться один на другой, иметь некоторую связь между друг другом. Например Менеджер может выписать Счет. Соответственно возникает ассоциация между Менеджером и Счетом. Еще пример — Преподаватель и Студент — т.е. какой-то Студент учится у какого-то Преподавателя. Ассоциация и есть описание связи между двумя объектами. Студент учится у Преподавателя. Идея достаточно простая — два объекта могут быть связаны между собой и это надо как-то описать.

**Агрегация** — отношение когда один объект является частью другого. Например Студент входит в Группу любителей физики.

**Композиция** — еще более «жесткое отношение, когда объект не только является частью другого объекта, но и вообще не может принадлежать еще кому-то. Например Машина и Двигатель. Хотя двигатель может быть и без машины, но он вряд ли сможет быть в двух или трех машинах одновременно. В отличии от студента, который может входить и в другие группы тоже. Такие описания всегда несколько условны, но тем не менее.