Лекция № 1

Тема: **Введение в Java**

Java - кроссплатформенный , объектно-ориентированный, бесплатный язык программирования, разработанный компанией Sun Microsystems (в последующем приобретённой компанией Oracle).

Программы на Java транслируются в **байт-код**, выполняемый **виртуальной машиной Java** (Java VM, JVM) - программой, обрабатывающей байт-код и передающей инструкции оборудованию как **интерпретатор**.

Основное достоинство языка Java - именно в его кросс-платформенности. Байт-код не зависит от оборудования и легко переносим.

Главным недостатком Java является то, что, в отличие от C++ или Delphi, это все же не компилятор, а интерпретатор. Программа на Java работает в среднем в 2- 5 раз медленнее, чем программа на C++ и потребляет в среднем в 10 раз больше памяти.

Java является базовым языком программирования для операционной системы **Android**. Однако, прежде, чем приступить к изучению программирования на Java для Android, несколько лекций мы посвятим изучению основ Java без привязки к какой-либо операционной системе.

На рис . 1.1 приведена структурная схема жизненного цикла разработки и запуска программы на языке Java.

Рис. 1.1. Жизненный цикл разработки и запуска программы на языке Java

Исходный текст Java-программы должен быть файлом с расширением .java. Для компиляции программы в байт-код используется программа javac из JDK6, который мы научились устанавливать на компьютер в ходе лабораторной работы №1. Чтобы откомпилировать программу MyProg.java нужно в командной строке набрать javac MyProg.java. Если текст программы не содержит ошибок, компилятор создаст файл с таким же именем , но с расширением .class (байт-код программы). Теперь эту программу можно запустить с помощью Java VM (файл java.exe). Для этого нужно в командной строке набрать java MyProg (расширение файла не указывается).

Специальными компановщиками из файлов байткода и файлов ресурсов могут формироваться, например, файлы приложений для мобильных телефонов (расширение .jar) или установочные файлы приложений для операционной системы Android (с расширением .apk).

Файлы байткода исполняются виртуальной машиной Java (Java VM). Для каждой операционной системы или устройства разрабатывается своя Java VM, в то время как байт-код программы остается неизменным (см. рис. 1.2).

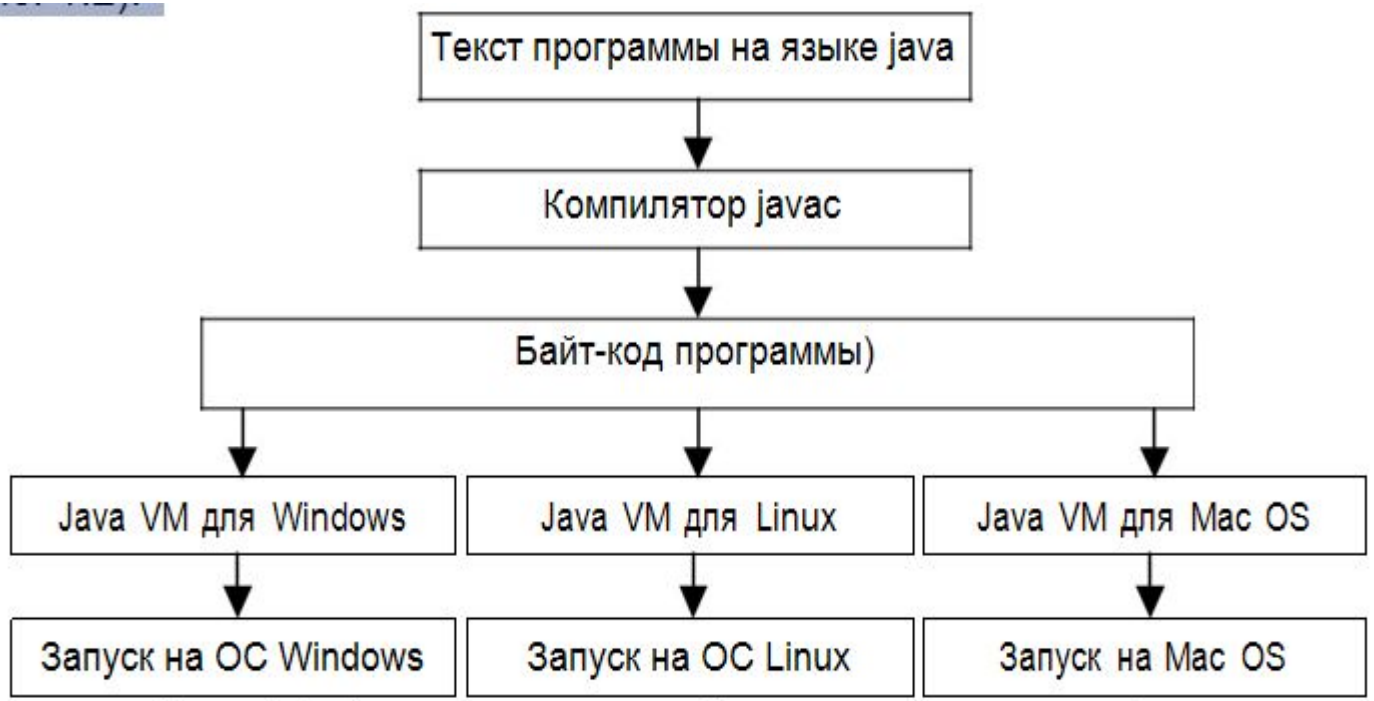

Рис. 1.2. Запуск приложений на различных платформах при помощи Java VM

Java и объектно-ориентированное программирование

- отличии от других языков программирования, например C++, PHP, где объектно-ориентированное программирование можно использовать, если есть желание, в языке программирования Java каждая программа – объектно-ориентированная.

Основная конструкция языка программирования Java, основной объект, с которым можно что-то делать – это **класс**. У каждого класса есть какие-то характеристики, называемые **полями** (другими словами – переменные) и умения что-то делать, называемые **методами** (другими словами - функции). На рис. 1.3 приведен пример текста простой программы, где есть метод и переменная.

Модификаторы public, private, protected

Как мы уже заметили, перед именами классов, методов и переменных у нас часто стоит служебное слово public. Так вот, это служебное слово сообщает компилятору Java, что помеченные им метод или поле можно без ограничений использовать в других классах (в других программах). Кроме служебного слова public, есть еще другие служебные слова, в частности private и protected. Вот, что означают эти слова:

public – методы и поля видно и **можно использовать где угодно**; **private** – методы и поля видно и можно использовать **только в этом классе**;

protected – методы и поля видно и можно использовать **только в этом классе или в классах, наследующих его** с помощью **extends.**

Для чего это все нужно? Модификаторы private и protected нужны, чтобы защитить поля от случайного изменения из других программ, которые используют класс.

Модификаторы public, private, protected

Как мы уже заметили, перед именами классов, методов и переменных у нас часто стоит служебное слово public. Так вот, это служебное слово сообщает компилятору Java, что помеченные им метод или поле можно без ограничений использовать в других классах (в других программах). Кроме служебного слова public, есть еще другие служебные слова, в частности private и protected. Вот, что означают эти слова:

public – методы и поля видно и **можно использовать где угодно**; **private** – методы и поля видно и можно использовать **только в этом классе**;

protected – методы и поля видно и можно использовать **только в этом классе или в классах, наследующих его** с помощью **extends.**

Для чего это все нужно? Модификаторы private и protected нужны, чтобы защитить поля от случайного изменения из других программ, которые используют класс.

Когда это нужно? Если речь идет о написании небольшой программы одним человеком, то все поля и методы спокойно можно помечать как public. Скорее всего этот один человек не запутается в своем коде программы.

Теперь представим, если речь идет о создании большого проекта, где десятки программистов совместно трудятся над созданием программного кода, причем каждый из них пишет свои классы, но приходится вызывать методы и из классов, написанных кем-то другим. Вэтом случае запросто может возникнуть ситуация , когда в тексте своего класса программист Вася дал какому-то полю название Value, и, одновременно, программист Петя дал названия Value одному из полей своего класса. В этом случае при присвоении полю Value какого-то значения вполне может возникнуть путаница, в результате чего весь сложный проект будет работать неправильно и потребуются значительные усилия для его отладки и выявления ошибки. Использование же в таких больших проектах модификаторов private или protected позволяет защитить поля и методы от случайного изменения из другого класса. Крайне рекомендуется защищать **все** поля классов модификатором private, а также защищать этим модификатором большинство методов, которые не предполагается в дальнейшем использовать из других классов. Причем даже начинающий программист на Java, который пишет небольшие программы просто для тренировки, с самого начала должен приучать себя к этому правилу. Тогда при поступлении на работу в серьезную фирму ему будет легко и привычно выполнять требования по написанию грамотного (с точки зрения надежности) кода.

Рассмотрим на примере использование модификатора private. На рис. 1.12 приведе текст класса factorial, а на рис. 1.13 - класса test, который его использует.

```
public class factorial {

  private int limit=10;
  public void demo() {
    int r=1;
    for (int i=2;i<=limit;i++) r=r*i;
    System.out.println("Факториал от значения
"+limit+" равен "+r);
    }

}
```

Рис. 1.12. Текст программы factorial.java

```
public class test{

 public static void main(String[] args) {
    factorial k=new factorial();
    k.demo();
  }

}
```

Рис. 1.13. Текст программы test.java

Метод demo() выводит на экран результат вычисления факториала от числа 10. Это число хранится в поле limit, защищенном модификатором private. Ни прочитать, ни изменить значения этого поля из класса test невозможно. При попытке вставить в текст метода main класса test, например, строку k.limit=6; компилятор **javac** выдаст ошибку.

Каким же образом, не нарушая принципа надежного программирования (все поля должны быть помечены private) тем не менее разрешить из других классов изменять значения полей? Очень просто - написать метод, который их изменяет. На рис. 1.14 приведена модификация класса factorial, позволяющая менять значение поля limit.

- этом случае компилятор не выдаст никакой ошибки, так как прямого обращения к полю limit в классе test нет, а изменение этого поля происходит внутри одного из методов класса factorial, что разрешается модификатором private, так как это поле декларировано именно в этом классе.
- результате модифицированная программа test.java выводит на экран значение факториала для числа 6.

Инкапсуляция. Доступ к полям только через методы

Только что рассмотренный нами пример на рис. 1.12 - 1.15 иллюстрирует термин **инкапсуляция**, часто упоминающийся при изложении принципов грамотного программирования на Java. Инкапсуляция как раз и подразумевает, что все поля нужно защищать модификатором private, а для чтения и изменения их значений создавать специальные методы.

Использование при программировании инкапсуляция позволяет повысить надежность больших программ на Java. В то же время инкапсуляция - это подход к программированию, а не обязательное требование . Как уже отмечалось, если программист пишет маленькую программку для себя, то для уменьшения длины текста этой программы инкапсуляции можно не придерживаться.

Интерфейсы

Интерфейс – это скелет (заготовка) класса с перечислением необходимых методов, но без их кода. Класс можно наследовать и использовать. Для интерфейса же нужно, чтобы кто-то создал класс, написав в нем тексты заявленных методов. И только после этого он (этот класс) будет пригоден к использованию.

На рис. 1.16 приведен пример интерфеса primer, на рис. 1.17- пример реализации этого интерфейса классом test, а на рис.1.18 - пример класса ura, в котором используется класс test. тексте класса, который реализует (имплементирует) интерфейс, должны быть написаны тексты всех методов, декларированных в этом интерфейсе. Класс может реализовывать сразу несколько интерфейсов. В этом случае они указываются через запятую после слова implements.

Интерфесы нужны в жизни опять таки при разработке больших программ большими коллективами программистов.

Пакеты

Иногда для больших проектов можно задать имя класса , например, Dog, а оно уже существует (другой программист создал класс с таким же именем). Чтобы таких конфликтов не было, в Java предлагается использовать пакеты.

Чтобы использовать пакет, нужно перед объявлением класса вставить объявление пакета, например, так:

```
package papa;
public class khai {
```

При этом текст программы этого класса нужно сохранить в папку с таким же именем, как у пакета.

Использовать класс из пакета можно двумя способами. Первый из них, ставить перед именем используемого класса имя пакета с точкой:

```
public class telek {
public static void main(String[] args) {
papa.khai k=new papa.khai(); k.reklama("Телекоммуникации",30);
}
}
```

Второй способ - вставить в начале программы ключевое слово **import** с названием пакета, вот так:

```
import papa.*;
public class telek2 {
public static void main(String[] args) {
khai k=new khai();
k.reklama("Телекоммуникации",30);
}
}
```

На практике используйте тот из этих способов, который кажется Вам более удобным в каждом конкретном случае.

Основными стандартными пакетами Java, с которыми нам придется иметь дело, являются:

java.lang - базовая функциональность языка и основные типы

java.util - коллекция классов структур данных

java.io - операции ввода-вывода

java.math - математические операции

java.nio - новый фреймворк для ввода-вывода

java.net - операции с сетями, сокетами, DNS-запросами

java.security - генерация ключей, шифрование и дешифрование

java.sql - Java Database Connectivity (JDBC) для доступа к базам данных

java.awt - иерархия основных пакетов для родных компонентов GUI

javax.swing - иерархия пакетов для платформенной независимости GUI

компонентов

Полиморфизм.

- полиморфизмом мы уже сталкивались, когда рассматривали перегрузку методов. В ООП полиморфизм реализуется за счёт переопределения методов в классах-потомках в процессе наследования.

Суть полиморфизма заключается в том, что *под одним и тем же именем скрываются разные варианты программного кода* . Какой из вариантов будет выполняться в момент обращения к имени – выбирается, исходя из контекста этого обращения. Для перегруженных методов контекстом был набор аргументов, а теперь контекстом является класс того объекта, для которого метод вызван.

Пример про домашних животных см. выше. Метод *voice* имеет одно название, но даёт разные результаты в зависимости от того, к какому объекту он применяется.

Основная цель полиморфизма: « если метод называется *print*, то он должен печатать что
угодно: для животного – кличку, для окружности – радиус».