Конспект лекций по Java. Лекция 1

Java является объектно-ориентированным языком программирования, разработанным фирмой **Sun Microsystems** (сокращенно, **Sun**). **Основные достоинства языка**

- Наибольшая среди всех языков программирования степень переносимости программ.
- Мощные стандартные библиотеки.
- Встроенная поддержка работы в сетях (как локальных, так и Internet/Intranet).

Основные недостатки

- Низкое, в сравнении с другими языками, быстродействие, повышенные требования к объему оперативной памяти (ОП).
- Большой объем стандартных библиотек и технологий создает сложности в изучении языка.
- Постоянное развитие языка вызывает наличие как устаревших, так и новых средств, имеющих одно и то же функциональное назначение.

Основные особенности

- Java является полностью объектно-ориентированным языком. Например, C++ тоже является объектно-ориентированным, но в нем есть возможность писать программы не в объектноориентированном стиле, а в Java так нельзя.
- Реализован с использованием интерпретации P-кода (байт-кода). Т.е. программа сначала транслируется в машинонезависимый P-код, а потом интерпретируется некоторой программой-интерпретатором (виртуальная Java-машина, JVM).

Версии Java

Есть несколько версий Java (1.0, 1.1, 2.0). Последняя версия Java 2.0. Это версии языка. Существуют также версии стандартных средств разработки Java-программ от Sun. Sun выпускает новые версии этих стандартных средств раз или два в год. Ранее они назывались **JDK (Java Development Kit)**, в последних версиях название изменено на **SDK (Software Development Kit)**. Официальное название текущей версии **"Java (TM) SDK, Standart Edition, Version 1.3.0"**. Предыдущая версия имеет номер 1.2.2. Как и текущая версия она соответствует версии языка Java 2.0. SDK - это базовая среда разработки программ на Java. Она является невизуальной и имеет бесплатную лицензию на использование. Есть и визуальные среды разработки (**JBuilder, Semantec Cafe, VisualJ** и др.).

Новые версии языка и версии SDK являются расширением прежних. Это сделано для того, чтобы не возникала необходимость переписывать существующие программы. Так программа, написанная на Java 1.0 или Java 1.1, будет работать и

под SDK 1.3. Правда, при компиляции некоторых таких программ могут выдаваться предупреждающие сообщения типа "Deprecated ...". Это означает, что в программе использованы возможности (классы, методы), объявленные в новой версии устаревшими (deprecated).

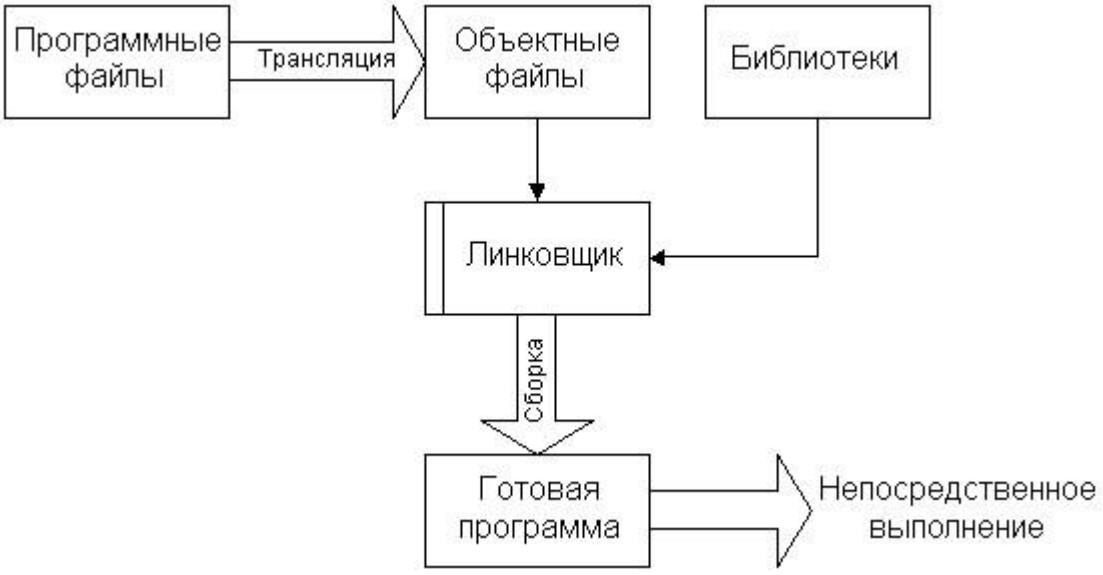
Аплеты

Аплеты являются одной из важных особенностей Java. Java позволяет строить как обычные приложения так и аплеты.

Аплет - это небольшая программа, выполняемая браузером (например, на Internet Explorer или Netscape Navigator). Аплет встраивается специальным образом в web-страничку. При подкачке такой странички браузером он выполняется виртуальной Javamашиной самого браузера. Аплеты расширяют возможности формирования web-страниц.

Жизненный цикл программы на Java

Под жизненным циклом мы будем понимать процесс, необходимый для создания работающего приложения. Для программ на Java он отличается от жизненного цикла программ на других языках программирования. Типичная картина жизненного цикла для большинства языков программирования выглядит примерно так.



Примерно такая схема действует в случае использования таких популярных языков как C++ или VB.

Для Java картина иная.

Из рисунка видно, что исходная Java-программа должна быть в файле с расширением **java**. Программа транслируется в байт-код компилятором **javac.exe**. Оттранслированная в байт-код программа имеет расширение **class**. Для запуска программы нужно вызвать интерпретатор **java.exe**, указав в параметрах вызова, какую программу ему следует выполнять. Кроме того, ему нужно указать, какие библиотеки нужно использовать при выполнении программы. Библиотеки размещены в файлах с расширением **jar** (в предыдущих версиях SDK использовались файлы ***.zip** и некоторые библиотеки все еще в таких файлах).

Структура пакета SDK

Пакеты Java SDK являются бесплатными. Текущая версия пакета может быть получена по адресу http://java.sun.com/j2se/.

Загруженная с этого адреса инсталляция пакета SDK будет, скорее всего, в виде одного большого exe-файла - файла инсталляции. Для установки SDK этот файл нужно запустить. При этом будет запрошен каталог для установки. Лучше выбрать тот каталог, который предлагается в инсталляции по умолчанию.

В выбранном Вами при инсталляции каталоге будет построена структура подкаталогов пакета SDK. Кратко рассмотрим эту структуру и выясним содержимое и назначение всех подкаталогов.

bin - каталог инструментария разработчика

demo - каталог с примерами **include** - для

взаимодействия с программами на C, C++ **include-**

old - аналогично, но предыдущая версия

jre - каталог инструментария пользователя (то, что поставляется конечному пользователю

при установке (deployment) готового приложения)

jre\bin - Java-машина(ы) (JVM)

jre\lib - библиотеки Java для конечных пользователей + ряд настроечных

файлов **lib** - библиотеки Java для разработчиков

Каталог bin\ содержит ряд программ, которые необходимы в процессе разработки Java-приложений. В частности, в нем расположен транслятор Java - **javac.exe**.

Каталог demo\ содержит ряд примеров, демонстрирующих те или иные возможности Java. Это, как всегда, полезно и интересно.

Каталоги include\ и include-old\ мы сейчас рассматривать не будем. Они используются при совместном использовании Java и C++. Каталог jre\ нужен для поставки конечному пользователю разработанного приложения. Он состоит из двух подкаталогов jre\bin\ и jre\lib\. Подкаталог jre\bin\, в частности, содержит виртуальную машину Java (JVM) - **java.exe**, которая нужна для запуска готовых приложений. Этот же файл (java.exe) можно найти и в подкаталоге bin\. Вы можете вызывать JVM как из bin\, так и из jre\bin\. Лично я предпочитаю jre\bin\.

Подкаталог jre\lib\ содержит библиотеки, необходимые для выполнения готовых приложений, а также ряд настроечных файлов. Основная системная библиотека находится в файле **rt.jar** (очевидно, от Run Time). Ее необходимо подключать при трансляции и выполнении любой Java-программы. Вторая по важности библиотека **i18n.jar**. Если Вам необходимо, чтобы программа поддерживала русскоязычную кодировку, то придется подключить и эту библиотеку.

Отступление. Продолжим инсталляцию SDK

Запустив программу инсталляции и указав каталог для инсталляции SDK мы сделали почти все, что нужно для установки рабочей версии инструментальных средств разработчика. Единственное действие, которое нужно еще произвести, это - настройка на русскоязычную кодировку.

Для настройки на ту или иную кодировку JVM использует файл **font.properties** из jre\lib\. Там же, в jre\lib\, находятся заготовки этих настроек для различных локализаций. Файл **font.properties.ru** содержит настройки для русскоязычной локализации.

Переименуйте **font.properties** в **font.properties.std**, а **font.properties.ru** в **font.properties**.

Это все, что требуется.

- Процедура переименования font.properties.ru в font.properties не нужна, если в Window установлена по умолчанию русская локализация.

Практическая работа

Рассмотрим демонстрационный пример - приложение типа "Hello World". Но сначала выполним некоторые подготовительные действия, которые облегчат нам жизнь в дальнейшем.

Подготовительные действия

После инсталляции пакета SDK можно транслировать программу на Java, вызывая компилятор `javac.exe` из командной строки. Аналогично, для выполнения можно тоже вызывать JVM - `java.exe`.

Но по ряду причин удобнее создать отдельные batфайлы для трансляции и выполнения Javaприложений. В частности, bat-файл позволит подключить при необходимости любой набор дополнительных библиотек, когда Вам это понадобится.

Ниже приведены примеры bat-файлов трансляции и выполнения. В них все установки сделаны на основе переменной `JDKHOME`, которая указывает каталог, где проинсталлирован SDK. Если у Вас другой каталог, измените значение этой переменной.

Файл для трансляции (**j.bat**)

```
REM Компилятор JAVA
set
JDKHOME=d:\jdk1.3
set CLASSPATH=.;%JDKHOME%\jre\lib\rt.jar
%JDKHOME%\bin\javac %1 %2 %3 %4 %5
```

Файл для выполнения (**jr.bat**)

```
REM Запуск программы на JAVA
set JDKHOME=d:\jdk1.3
set
CLASSPATH=.;%JDKHOME%\jre\lib\rt.jar;%JDKHOME%\jre\lib\i18n.ja
r
%JDKHOME%\jre\bin\java -cp %CLASSPATH% %1 %2 %3 %4 %5 %6
```

Первая программа

Для подготовки Java-программы подойдет любой текстовый редактор. Существенным является только наличие в нем поддержки длинных имен файлов. В простейшем случае Javaприложение состоит из одного java-файла. Наберем простейшую Java-программу (файл Hello.java):

```
public class Hello {
   public static void main(String[]
args) {
      System.out.println("Hello");
  }

}
```

Откомпилируем программу при помощи команды `j Hello.java`

Если Вы набрали текст правильно, то в результате компиляции на экран ничего не будет выведено. Если же нет, то на экран будут выданы сообщения об ошибках.

Альтернативный вариант компиляции программы (без bat-файла): `javac.exe Hello.java`

Теперь запустим ее на выполнение `jr Hello`

На экран будет выведено одно слово **Hello**.

Альтернативный вариант (без bat-файла):
`java.exe Hello`

Замечание

- Обратите внимание, что при трансляции программы задается имя файла (с расширением), а при выполнении имя файла без расширения. Очень часто начинающие работать с Java допускают здесь ошибки.

Что демонстрирует данный пример

Данный пример примитивен, но тем не менее на этом примере можно познакомиться с очень важными понятиями.
Рассмотрим, что он демонстрирует.

- Как мы рассмотрим подробнее позже, весь программный код в Java заключен внутри классов. Не может быть никакого программного текста (за исключением нескольких специальных директив) вне класса (или интерфейса).
- Каждый файл с именем **Name.java** должен содержать класс с именем **Name** (причем, учитывается регистр). Каждый `public`-класс с именем **Name** должен быть в своем файле **Name.java**.
- Внутри указанного файла могут быть и другие классы, но их имена должны отличаться от **Name** и они не должны быть `public`.
- Внутри класса может быть конструкция
  ```
  public static void main(String[] args)
  {   . . .
  }
  ```
  Это метод класса. Здесь `public, static, void` - это описатели, `main` - имя метода.

- Указанный метод main является специальным случаем. При запуске Java-программы мы указываем имя класса, и Java-машина ищет этот класс среди всех доступных ей файлов *.class, и в этом классе запускает на выполнение метод `main`.
- Описание метода `main` должно быть в точности таким, как приведено в примере (можно разве что изменить имя `args` на какое-то другое).
- В скобках после имени метода указываются параметры метода. Для `main`-метода параметры должны быть такими как указано. Это - массив строк. При вызове программы на Java можно задать параметры вызова. Java-машина обработает их и сформирует массив строк, который будет передан в `main`-метод в качестве параметра.

  Так, если вызвать программу

  командой `j Hello one two 3 4`

  то внутри программы args будет массивом из 4-х элементов

```
args[0] = "one"
args[1] = "two"
args[2] = "3"
args[3] = "4"
```

Знакомство с демонстрационными примерами SDK

В состав поставки JDK, как уже отмечалось, входят примеры. Они находятся в каталоге **demo**. Зайдем в каталог **demo\jfc** и посмотрим программы `SimpleExample` и `SwingSet2`.

Замечание

- Примеры подготовлены в jar-файлах и можно запускать их прямо оттуда. Для этого существует специальная опция JVM - "-jar".

Запустить пример `SimpleExample` можно при помощи следующей командной строки `java.exe -jar SimpleExample.jar`

Посмотрите эти примеры внимательно. Эти примеры показывают, что Java позволяет строить диалоговые программы. Причем, примеры демонстрируют такую особенность Java как **Look and Fill**. Т.е. программа на Java может быть настроена на различные типы интерфейсов, ряд из которых имеет вид, общепринятый в тех или иных системах.

Литература и ресурсы

В Internet можно найти много литературы по Java, начиная от фирменной документации от Sun до учебников и различных обучающих курсов по Java.

- Русскоязычная версия сайта Sun Microsystems. http://www.sun.ru/java.

- Одной из популярных книг по Java является "Thinking in Java", Bruce Eckel. Она может быть получена с http://www.bruceeckel.com/.
- Документация от Sun доступна по адресу http://java.sun.com/products/jdk/1.3/docs/index.html.
- Краткое описание доступной документации http://java.sun.com/products/jdk/1.3/devdocs-vs-specs.html.
- Обучающие курсы для разработчиков http://developer.java.sun.com/developer/onlineTraining/.
- Русскоязычный сайт по Java http://www.javable.com/.
- Популярный англоязычный сайт http://www.javaworld.com/.
- Книга Фролов А.В., Фролов Г.В. "Создание приложений Java" http://www.sun.ru/java/books/online/index.html или http://athena.vvsu.ru/docs/c-java/java_f/.
- Java FAQ (на русском) http://www.sun.ru/java/start/questions/faq/faq.html.

Замечательная книга по Swing'y: *Swing* by Matthew Robinson and Pavel Vorobiev.

- http://manning.spindoczine.com/sbe/.

Конспект лекций по Java. Лекция 2

Отступление

Тема данного занятия является вводной, но тем не менее очень важной. Здесь вводятся в рассмотрение ряд принципов объектно-ориентированного подхода, которые будут более подробно раскрыты на последующих занятиях.

Материал данного занятия может остаться не до конца понятым с первого раза. В этом случае можно порекомендовать продолжить знакомство со следующими темами и вернуться к нему позже.

JAVA - объектно-ориентированный язык программирования

Java является объектно-ориентированным языком программирования. Если сравнивать в этом смысле Java и C++, то между ними есть существенные различия.

C++ тоже является объектно-ориентированным языком. Но он является расширением C, который не является объектно-ориентированным. Из этого, в частности следует, что на C++ можно писать программы как в объектноориентированном стиле, так и в обычном. Т.е. на C++ можно программировать, не зная и не понимая объектно-ориентированного подхода. Можно даже смешивать оба подхода в едином продукте или пытаться программировать в объектноориентированном стиле, а тогда, когда это не получается, скатываться к традиционному программированию.

В Java так нельзя. В ней нет средств, позволяющих писать не объектно-ориентированные программы.

Из этого сразу следует один вывод. *Нельзя научиться программировать на Java, не овладев основами объектно-ориентированного подхода.*

5 принципов объектно-ориентированного подхода

Все является объектом

- Все данные программы хранятся в объектах. Каждый объект создается (есть средства для создания объектов), существует какое-то время, потом уничтожается.

Программа есть группа объектов, общающихся друг с другом

- Кроме того, что объект хранит какие-то данные, он умеет выполнять различные операции над своими данными и возвращать результаты этих операций. Теоретически эти операции выполняются как реакция на получение некоторого сообщения данным объектом. Практически это

происходит при вызове метода данного объекта. Что такое метод и как он относится к объекту, мы рассмотрим позднее.

- Каждый объект имеет свою память, состоящую из других объектов и/или элементарных данных.

  Объект хранит некоторые данные. Эти данные - это другие объекты, входящие в состав данного объекта и/или данные элементарных типов, такие как целое, вещественное, символ, и т.п.

Каждый объект имеет свой тип (класс)

- Т.е. в объектно-ориентированном подходе не рассматривается возможность создания произвольного объекта, состоящего из того, например, что мы укажем в момент его создания. Все объекты строго типизированы. Мы должны сначала описать (создать) тип (класс) объекта, указав в этом описании из каких элементов (полей) будут состоять объекты данного типа. После этого мы можем создавать объекты этого типа. Все они будут состоять из одних и тех же элементов (полей).

Все объекты одного и того же типа могут получать одни и те же сообщения

- Кроме описания структуры данных, входящих в объекты данного типа, описание типа содержит описание всех сообщений, которые могут получать объекты данного типа (всех методов данного класса). Более того, в описании типа мы должны задать не только перечень и сигнатуру сообщений данного типа, но и алгоритмы их обработки.

Реализация принципов объектно-ориентированного подхода в Java

Ссылки на объекты

В Java для манипулирования объектами в программном коде используются *ссылки на объекты (handles)*. Ссылка хранит в себе некоторый адрес объекта в оперативной памяти.

Может быть несколько ссылок на один объект. На какой-то объект может вообще не быть ссылок (тогда он для нас безвозвратно потерян). Ссылка может не ссылаться ни на какой объект - пустая (**null**) ссылка. Не может быть ссылки в никуда или ссылки на какуюто произвольную область памяти. Как транслятор Java, так и JVM внимательно следят за тем, чтобы нельзя было создать ссылку на

какую-то произвольную область памяти. Практически в Java это сделать невозможно.

Все ссылки имеют имя

Для манипулирования самими ссылками в программном коде необходимо как-то их обозначать. Это делается при помощи имени ссылки. Все ссылки, так или иначе, описываются, при этом каждой ссылке дается имя. Имена ссылок известны программе и встречаются в программном коде там, где нужно манипулировать объектами, на которые они ссылаются.

Все ссылки строго типизированы

При описании ссылки обязательно указывается ее тип. И эта ссылка может ссылаться только на объект данного типа. Есть определенные исключения из этого правила, связанные с наследованием классов. Мы рассмотрим это позднее.

Попытка присвоить ссылке адрес объекта не того класса пресекается как на этапе трансляции программы (выдаются ошибки трансляции), так и на этапе ее выполнения (возникает исключительная ситуация `ClassCastException`).

Приведем пример описания ссылки

```
MyType ref;
```

Здесь `MyType` - имя типа (как и ссылки, все типы имеют имя), `ref` - имя ссылки. После такого описания ссылке `ref` можно присвоить значение - адрес какого-то объекта типа `MyType`.

Создание объектов

Все объекты в Java создаются только явно, для чего используется операция **new**.

```
ref = new MyType();
```

Здесь создается объект типа `MyType` и его адрес заносится в `ref`. Почему здесь использованы скобки после `MyType`, мы рассмотрим позже. Еще один пример

```
MyType ref = new MyType();
```

Здесь описание ссылки совмещено с инициализацией.

Класс - способ описания типа

Для описания типов в Java используется механизм **классов**. За исключением базовых (иначе - элементарных) типов (`int, char, float` и др.), все остальные типы - это классы.

В простейшем случае описание класса выглядит так

```
class MyClass {

. . . // тело класса
```

```
}
```

Здесь **class** - ключевое слово, `MyClass` - имя класса. Внутри фигурных скобок находится тело класса.

Внутри тела класса описываются в произвольном порядке поля и методы класса.

Отступление Существуют общепринятые правила именования классов, их полей и методов. Следование этим правилам улучшает "читабельность" программы.

Имена классов принято начинать с большой буквы, а имена полей и методов - с маленькой. Если имя состоит из нескольких слов, то каждое новое слово начинают с большой буквы.

Общепринятых правил относительного размещения описаний полей и методов не существует. Но лучше, все же, размещать вместе все описания полей (например, в начале или в конце описания класса), а после или перед ними - описания методов.

В примере использован комментарий. В Java два типа комментариев. Все, что начинается с двух символов '/', является комментарием и этот комментарий продолжается до конца данной строки. Все, что начинается с символов "/*" является комментарием, который должен быть закрыт символами "*/".

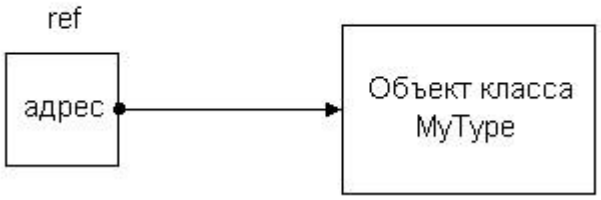
После того, как класс описан, мы можем создавать *объекты* (*objects*) класса (иначе - *экземпляры класса, class instances*).

Данные элементарных типов ссылками не являются

Ссылка хранит адрес объекта, а объект уже хранит какую-то содержательную информацию. В отличие от ссылок данные элементарных типов являются самосодержащими, они сами хранят содержательную информацию.

Так можно продемонстрировать ссылку на объект.

```
MyType ref = new MyType();
```



А так - данное элементарного типа.

```
int var = 3;
```

Базовые типы Java

Тип	Описатель	Размер	Комментарий
Логический	boolean	1 байт	-
Символьный	char	2 байта	Unicode

Байтовый	byte*	1 байт	(-128 - 127)
Короткий целый	short	2 байта	(-2^{15} - 2^{15}-1)
Целый	int	4 байта	(-2^{31} - 2^{31}-1)
Длинный целый	long	8 байт	(-2^{63} - 2^{63}-1)
Вещественный	float	4 байта	-
Вещественный двойной точности	double	8 байт	-
Пустой	void*	-	-

Замечания

- Тип `byte` является арифметическим целым типом.
- Все целые типы в Java знаковые, соответствующих беззнаковых эквивалентов, как в C++, в Java нет.
- Тип `void` фактически типом не является. Описатель `void` просто используется для указания того, что метод ничего не возвращает.

Поля класса и переменные программы

Как уже отмечалось, в классе можно описать *поля класса* (*fields or instance variable*). Поля класса определяют, из каких данных будут состоять объекты этого класса. Поля могут быть ссылками на другие объекты или элементарными данными.

В методах класса могут быть описаны *переменные*. Их не следует путать с полями класса. Как и поле класса, переменная может быть либо ссылкой, либо данным базового типа. Описание переменной выглядит точно так же, как и описание поля класса, за исключением того, что ряд описателей не применимы для переменных. Отличаются же (визуально) переменные от полей местом их описания. Поля класса описываются непосредственно в теле класса, на том же уровне вложенности, что и методы класса. Переменные описываются внутри методов. Пример:

```
class SomeClass {  // Это заголовок класса

    int i = 0;        // Это элементарное
данное, поле класса
    MyType ref;          // Это ссылка, тоже
поле класса

    int f() {                // Это
заголовок метода       int k = 0;
// Это элементарное данное, переменная
      MyType lref;       // Это ссылка,
переменная

      . . .              // Данный метод
чтото делает

    }                    // Это конец метода

    . . .               // В классе могут
быть и другие методы

 }                     // Это конец тела
класса
```

Область видимости и время жизни переменных

В различных языках программирования существуют различные типы или классы переменных - локальные, глобальные, статические и т.п. В Java только один тип переменных - локальные переменные. *Время жизни* переменной в Java определяется правилом:

- Переменная создается в точке ее описания и существует до момента окончания того блока, в котором находится данное описание.

В Java *блок* - это то, что начинается открывающей фигурной скобкой '{' и заканчивается закрывающей фигурной скобкой '}'.

Областью видимости переменной (*scope*) является фрагмент программы от точки ее описания до конца текущего блока.

- Область видимости - это статическое понятие, имеющее отношение к какому-то фрагменту текста программы. Время жизни, в отличие от области видимости, - это понятие динамики выполнения программы. Время жизни переменных в Java совпадает с их областью видимости с учетом отличия самих этих понятий.

Если в блоке, где описана данная переменная, вложены другие блоки, то переменная доступна в этих блоках (обычная практика языков программирования). Но, в отличие от многих других языков, в Java запрещено переопределять переменную во вложенных блоках (т.е. описывать другую переменную с тем же именем).

Рассмотрим примеры, демонстрирующие эти понятия.

Пример 1

```
. . .   // предшествующая часть программы

{       // начало блока

    . . .  // какие-то операторы внутри блока

    int x = 1;   // 1-я точка описания

    {            // еще один блок

        . . .    // какие-то операторы
внутри блока
        int y;   // 2-я точка описания

        . . .    // какие-то операторы внутри
блока

    }  // переменная y уничтожается и она
больше не видна

    . . .  // какие-то операторы внутри блока

}  // переменная x уничтожается и она больше не
видна

. . .   // последующая часть программы
```

Пример 2

```
{       // начало блока
     int x
= 1;
long y;

    {            // еще один блок
         int x;   // !!!
ошибка         int y;   //
!!! ошибка

        . . .    // какие-то операторы
внутри блока
    }

   . . .  // какие-то операторы внутри блока

}
```

Смысл примеров указан в комментариях в самих примерах. Первый пример просто показывает в каких пределах текста видна та или иная переменная (область видимости) и когда она создается и уничтожается (время жизни). Второй пример содержит ошибки. Он демонстрирует запрет переопределять переменные во вложенных блоках.

Область видимости и время жизни объектов

Иная картина наблюдается с объектами. Объекты доступны в программе только через ссылки на них. Поэтому область

видимости объекта определяется областью видимости ссылок на этот объект (на один объект может быть сколько угодно ссылок).

Время жизни объекта определяется следующим правилом.

- Объект существует, пока существует хотя бы одна ссылка на этот объект.

Это правило, однако, не утверждает, что объект будет уничтожен, как только пропадет последняя ссылка на него. Просто такой объект становится недоступным и может быть уничтожен.

- В Java нет явного уничтожения объектов. Объекты уничтожаются (говорят - утилизируются) *сборщиком мусора* (*garbage collector*), который работает в фоновом режиме параллельно с самой программой на Java.

Рассмотрим следующий фрагмент.

```
{
  SomeType localReference = new SomeType();
globalReference = localReference;
}
```

Здесь `SomeType` - это некоторый класс, `localReference` - локальная переменная-ссылка, `globalReference` - некоторая внешняя, по отношению к данному блоку, переменная или поле класса (из данного фрагмента нельзя сделать однозначный вывод, что это).

В этом фрагменте порождается объект класса `SomeType` и адрес этого объекта заносится в переменную `localReference`. После этого этот же адрес из `localReference` копируется в `globalReference`. По выходу из блока переменная `localReference` уничтожается, но переменная (или поле) `globalReference` продолжает существовать. Соответственно, продолжает существовать и порожденный объект.

Описание методов класса

В первом приближении *методы класса* (*class methods*) можно рассматривать как функции.

Описание метода выглядит следующим образом

```
<тип> <имя_метода> (<аргументы>) {
      <тело_метода>
}
```

Здесь <тип> - это один из базовых типов (см. таблицу выше) или пользовательский тип (т.е. некоторое имя класса). <аргументы> - это список, возможно пустой, параметров

метода. <тело_метода> - собственно программный код данного метода.
Каждый аргумент или параметр метода в данном описании - это пара "<тип> <имя_аргумента>". Аргументы отделяются друг от друга запятыми.

Описания методов расположены внутри класса, на том же уровне вложенности скобок, что и описание полей класса. Не может быть описания метода вне класса или внутри другого метода или блока.

Вызов методов

Вызов методов отличается от вызовов функций в не объектноориентированных языках программирования. При вызове обычного (не статического) метода класса обязательно должен быть указан объект этого класса и метод вызывается для этого объекта. Т.е. вызов метода - это вызов метода объекта.

Формальное исключение составляет вызов метода класса из другого (или того же) метода данного класса, в этом случае объект можно не указывать. Но фактически объект и в данном случае имеется, это - тот объект, для которого был вызван вызывающий метод.

Рассмотрим это на примерах. Опишем класс `SomeClass` и в нем методы `f` и `g`.

```
class SomeClass {
      int f(int
k) {
         . . .
     }
void g() {
          . . .
      }
}
```

Здесь описан метод `f` с одним параметром целого типа, возвращающий целое значение и метод `g` без параметров, не возвращающий никакого значения. Приведем примеры вызова этих методов из некоторого фрагмента программы.

```
a.f(x);
b.g();
v = b.f(3);
```

В приведенном фрагменте фигурируют переменные (или поля класса) `a, x, b` и `v`. Переменные `a` и `b` должны быть описаны как ссылки с типом `SomeClass`, переменные `x` и `v` должны быть целочисленными.

Данный фрагмент демонстрирует, что объект, для которого вызывается метод, должен быть указан при помощи ссылки, записанной перед собственно вызовом метода через точку.

При вызове метода класса из метода того же класса объект указывать не обязательно. Это, как указано выше, не нарушает

того правила, что при вызове метода всегда должен быть определен объект, для которого этот метод вызывается. Просто в данном случае этот объект уже определен при вызове "вызывающего" метода и для него же вызывается "вызываемый" метод.

Доступ к полям класса

Поля класса не существуют сами по себе (за исключением статических). Они расположены внутри объекта класса. Поэтому при доступе к полю должен быть определен объект.

Как и в случае вызова метода, при обращении к полю класса извне класса объект должен быть указан явно (при помощи ссылки на объект) перед именем поля через точку. Например, если в классе `SomeClass` есть поля `fld1` и `fld2`, а `obj` - ссылка на объект класса `SomeClass`, то

```
obj.fld1 = 2; x = obj.fld2;
```

являются примерами доступа к полям `fld1`, `fld2`. Изнутри класса, т.е. из нестатических методов класса, можно обращаться к полям класса напрямую, без указания объекта, поскольку такой объект определен при вызове данного метода.

Передача параметров

Для параметров функций, методов, процедур в программировании существует понятие тип передачи. Например, передача по значению или по ссылке. В Java существует всего один тип передачи - передача по значению. Это означает, что при вызове метода ему передается текущее значение параметра. Внутри метода можно произвольно изменять параметр, но это ни как не повлияет, скажем, на переменную, которая была указана в качестве параметра вызова.

Это нужно хорошо себе представлять, в особенности, когда передаются ссылки на объекты.

Рассмотрим пример. Пусть `ref` - ссылка на объект, передаваемая в качестве параметра при вызове метода `h(...)`.

```
h(ref);
```

Внутри метода `h` мы можем изменить параметр метода (т.е. присвоить ему ссылку на другой объект), но это никак не повлияет на саму ссылку `ref`, т.к. при вызове создается копия `ref` и изменяется именно она. С другой стороны мы можем внутри `h` менять данные того объекта, на который ссылается `ref`, и это реально отразится на этом объекте, т.к. создается копия только ссылки, но не самого объекта.

Конспект лекций по Java. Лекция 3

Реализация принципов объектно-ориентированного подхода в Java

Описатели ограничения доступа

Такие элементы языка Java как *класс*, *поле* и *метод* имеют *ограничения доступа*. Т.е. для них можно указать, где они будут доступны, а где нет.

Для описания ограничений доступа используются ключевые слова **public**, **private**, **protected**. Они являются опциональными описателями и дают нам три варианта ограничений доступа плюс четвертый вариант, если не указан не один из этих описателей.

- **public** - означает, что данный элемент доступен без каких-либо ограничений;
- **private** - доступ разрешен только из данного класса;
- **protected** - доступ разрешен из данного класса и из всех классов-потомков(это связано с наследованием классов, которое мы будем рассматривать позже), а также из всех классов данного пакета(что такое пакеты мы также будем рассматривать позже).
- *без описателя* - доступ разрешен из всех классов данного пакета.

Для классов применим только один описатель - public. Кроме того, классы могут не иметь никакого описателя ограничения доступа.

Как мы уже рассматривали на 1-м занятии, каждый java-файл должен содержать public-класс с именем этого файла и наоборот, каждый public-класс должен быть в java-файле с именем этого класса. Классы без описателя public могут быть в любом java-файле.

Для полей и методов применимы все 4 варианта ограничения доступа.

Мы, наконец, дошли до некоторой "критической массы" и можем написать несколько примеров программ так, что не все в этих программах будет непонятно. Отдельные элементы этих примеров, которые выходят за рамки рассмотренного материала, мы рассмотрим с содержательной стороны, т.е. разберемся, что они делают, не вдаваясь в подробности, что за понятия в них использованы.

Один из таких элементов - это вывод результатов. Он выглядит так

System.out.println(< то, что нужно напечатать>);

В скобках можно задать текст или переменную или выражение. Подробнее мы рассмотрим это на конкретных примерах.

Пример (файл XUser.java)

```
// Это простой демонстрационный пример
// Два класса X и XUser взаимодействуют друг с другом

class
X {
```

```
public
int x
= 0;
// В
классе
X одно
поле
   void f(int
y) {     x =
y;
  }

}
public class XUser {
   void
f() {
    int n = 6;    // Переменная n
    X x1 = new
X();     X x2 =
new X();     x1.x
= 5;     x1.f(n);

    x2.f(2);

    System.out.println("Поле x из x1: "+ x1.x);
    System.out.println("Поле x из x2: "+ x2.x);

}
  public static void main(String args[]) {
    XUser ref = new XUser();
ref.f();
  }
}
```

- Трансляция

```
j.bat XUser.java
```

- Запуск

```
jr.bat XUser
```

- Результат работы

```
Поле x из x1: 6
Поле x из x2: 2
```

Для того, чтобы разобраться в примере достаточно проследить всю цепочку его выполнения шаг за шагом, начиная от метода main. В заключение обратим внимание на строки

```
System.out.println("Поле x из x1: "+ x1.x);
System.out.println("Поле x из x2: "+
x2.x);
```

Параметром в данной конструкции является строка. Но для строк, как мы рассмотрим позднее, определена операция сложения строк друг с другом, с целыми и вещественными числами и даже с объектами.

Возврат значения в методе класса

Для возврата значения из метода класса используется оператор return вида

return < выражение>;

Например,

```
int f(int a, int b) {
  return a+b;   // возвращает сумму двух аргументов
данного метода
}
```

Как мы уже рассматривали, тип возвращаемого значения указывается в заголовке метода перед его именем. Тип выражения в операторе return должен соответствовать типу возвращаемого значения.

Ключевое слово this

Разберемся подробнее с вызовом метода. Мы рассматривали, что вызов метода всегда производится для некоторого объекта. При этом внутри метода можно обращаться к полям класса, не указывая объект.

Это реализуется путем передачи скрытого параметра - ссылки на тот объект, для которого вызван данный метод. Всегда, когда мы пишем обращение к полю или методу того же класса, что и сам данный метод, неявно подставляется этот скрытый параметр. В некоторых случаях требуется обратиться к этому параметру явно. Такая возможность предусмотрена. Для этого в языке определено ключевое слово **this**.

Например,

```
class
A {
  private int size = 0;

  public void setSize(int size) {
this.size = size;
  }
. .
.
}
```

Этот пример демонстрирует, как можно реализовать установку значения private-поля класса, т.е. поля недоступного извне класса. Для этого в классе реализован открытый (public) метод setSize(...). В этом методе имя параметра совпадает с именем поля класса

(size). Такое в Java разрешено, более того, это является общепринятым стилем. Но совпадение имен заставляет нас писать слово this с точкой перед именем поля (внутри метода имя size обозначает параметр, а не поле).

Мы рассмотрели базовые возможности языка, реализующие концепцию объектноориентированного программирования. Естественно, что это только "вершина айсберга".

Практическое задание

Рассмотрим программу Proba1.java.

```
public class Proba1 {

  public static void main(String args[]) {
    Oper op = new
Oper();     op.a = 6;
op.b = 7;
    System.out.println("Сумма=" + op.sum());
    System.out.println("Разность=" + op.dif());
  }

}
```

Это незаконченная программа. Разберем ее построчно, определим, что означает каждый оператор.

В программе используется класс Oper, который не реализован. Задача состоит в реализации этого класса. В частности, нужно определить

1. В каком файле разместить этот класс.
2. Какие поля должен содержать этот класс.
3. Какие методы должны быть в нем реализованы.

В заключение оттранслируем и запустим эту программу.

Статические поля и методы класса

Мы уже встречались в программах с описателем **static** в конструкции "public static void main(String args[])". Пора разобраться с тем, что он означает. В Java есть статические поля и статические методы. Для указания того, что поле или метод являются статическими, используется описатель **static** перед именем типа поля или метода. Например,

```
class SomeClass {
   static int t = 0;   // статическое поле
   . . .
   public static void f() {  // статический метод
       . . .
   }

}
```

Для поля описатель static означает, что такое поле создается в единственном экземпляре вне зависимости от количества объектов данного класса. Статическое поле существует даже в том случае, если не создано ни одного экземпляра класса. Статические поля класса размещаются VM Java отдельно

от объектов класса в некоторой области памяти в момент первого обращения к данному классу. Рассмотрим это на примере (файл Proba2.java)

```
// Демонстрация статических полей класса

public class Proba2 {
   int a = 10;         // обычное
поле   static int cnt = 0; //
статическое поле

  public void print() {
    System.out.println("cnt=" + cnt);
    System.out.println("a=" + a);
}
  public static void main(String
args[]) {     Proba2 obj1 = new
Proba2();     cnt++;          //
увеличим cnt на 1     obj1.print();
    Proba2 obj2 = new Proba2();
cnt++;          // увеличим cnt на 1
obj2.a = 0;     obj1.print();
obj2.print();
  }

}
```

В данном примере поле **cnt** является статическим. При печати обоих объектов в конце метода **main**(...) будет отпечатано одно и то же значение поля **cnt**. Это объясняется тем, что оно присутствует в одном экземпляре.

По аналогии со статическими полями, статические методы не привязаны к конкретному объекту класса. При вызове статического метода перед ним можно указать не ссылку, а имя класса. Например,

```
class SomeClass {
   static int t = 0;   // статическое поле
   . . .
   public static void f() {  // статический метод
        . . .
   }

}
.
.
.

SomeClass.f(); .
. .
```

Поскольку статический метод не связан с каким либо объектом, внутри такого метода нельзя обращаться к нестатическим полям класса без указания объекта перед именем поля. К статическим полям класса такой метод может обращаться свободно.

Это легко продемонстрировать на примере Proba2.java. Попробуем описать метод print(...) как статический. При трансляции измененной программы мы получим сообщение об ошибке в строке

```
System.out.println("a=" + a);
```

Действительно, здесь идет обращение к нестатическому полю **a** и, поскольку метод print(...) является статическим, то неизвестно к какому объекту относится это поле.

Если мы хотим сделать метод print(...) статическим (что, в общем-то, не соответствует общим принципам построения объектноориентированных программ), то мы должны определить в нем параметр типа Proba2 и использовать его для доступа к полю **a**.

Перепишем этот пример (файл Proba3.java)

```
// Демонстрация статических полей и методов класса

public class Proba3 {
   int a = 10;         // обычное
поле   static int cnt = 0; //
статическое поле
   public static void print(Proba3
obj) {     System.out.println("cnt="
+ cnt);
    System.out.println("a=" + obj.a);









}
  public static void main(String
args[]) {     Proba3 obj1 = new
Proba3();     cnt++;          //
увеличим cnt на 1
    Proba3.print(obj1);     Proba3
obj2 = new Proba3();     cnt++;
// увеличим cnt на 1     obj2.a = 0;
print(obj1);     print(obj2);
  }

}
```

Обратим внимание на то, что при первом вызове print(...) перед ним указано имя класса, а в двух других случаях вызова print(...) вызывается без каких-либо уточнителей. В действительности и при первом вызове мы могли не ставить префикс "Proba3.", поскольку мы вызываем статический метод класса из другого метода класса, а при этом указывать имя объекта или класса не обязательно.

Статические методы используются достаточно часто. В составе стандартной библиотеки Java есть целые классы, в которых все методы статические (Math, Arrays, Collections).

Знакомство с документацией

При работе с Java невозможно обойтись без документации. Документация поставляется в виде html-файлов. В каталоге c:\jdk1.3\docs находится головной индекс документации по Java. Откроем его. Там указано, какие разделы документации доступны локально (в колонке Location проставлено docs), а какие могут быть загружены (website).

Основная документация, необходимая в работе над программами - " Java 2 Platform API Specification " (c:\jdk1.3\docs\api\index.html). Это документация по стандартной библиотеке Java. Откроем ее. Выберем вариант FRAMES.

В левой колонке вверху список пакетов. Библиотеки в Java строятся из пакетов. Пакет - это просто способ группировки связанных друг с другом классов. Мы будем подробнее знакомиться с ними позже.

В левой колонке снизу - список классов. Изначально в списке пакетов выбран элемент "All packages" (Все пакеты). Это означает, что в списке классов виден перечень всех классов библиотеки. Если кликнуть на некотором конкретном пакете, то в списке классов останутся только классы данного пакета.

Основную часть экрана занимает собственно документация по конкретному классу. Если выбрать некоторый класс в списке классов, то в основной части появится документация по этому классу. Выберем в списке пакетов пакет java.lang и в нем класс Boolean. Просмотрим документацию по этому классу. Сначала идет небольшое описание класса, потом краткий список полей (Field Summary), потом перечень конструкторов (Constructor Summary), потом список методов (Method Summary), а потом уже детальное описание всех этих элементов.

Выберем в том же пакете класс Math и постараемся по документации выяснить его назначение и возможности.

- Поскольку эта документация нужна постоянно, лучше сделать для нее иконку на рабочем столе. Для этого лучше всего кликнуть правой клавишей мыши на ссылке FRAMES в верхней строке экрана, скопировать ссылку (Copy Shortcut), а потом перейти на рабочий стол и выполнить Paste Shortcut.

Конспект лекций по Java. Лекция 4

Операции (operators) в языке Java

Большинство операций Java просты и интуитивно понятны. Это такие операции, как +, -, *, /, <,> и др. Операции имеют свой порядок выполнения и приоритеты. Так в выражении `a + b * c`

сначала выполняется умножение, а потом сложение, поскольку приоритет у операции умножения выше, чем у операции сложения. В выражении `a + b - c`

сначала вычисляется a + b, а потом от результата вычитается c, поскольку порядок выполнения этих операций слева направо.

Но операции Java имеют и свои особенности. Не вдаваясь в детальное описание простейших операций, остановимся на особенностях.

Начнем с **присваивания**. В отличии от ряда других языков программирования в Java присваивание - это не оператор, а операция. Семантику этой операции можно описать так.

Операция присваивания обозначается символом '='. Она вычисляет значение своего правого операнда и присваивает его левому операнду, а также выдает в качестве результата присвоенное значение. Это значение может быть использовано другими операциями. Последовательность из нескольких операций присваивания выполняется справа налево. В простейшем случае все выглядит как обычно. `x = a + b;`

Здесь происходит именно то, что мы интуитивно подразумеваем, - вычисляется сумма a и b, результат заносится в x. Но вот два других примера.

`a = b = 1; a+b;`

В первом сначала 1 заносится в b, результатом операции является 1, потом этот результат заносится в a. Во втором примере вычисляется сумма a и b и результат теряется. Это бессмысленно, но синтаксически допустимо.

Операции сравнения

Это операции >, <, >=, <=, != и ==. Следует обратить внимание, что сравнение на равенство обозначается двумя знаками '='. Операндами этих операций могут быть арифметические данные, результат - типа boolean.

Операции инкремента, декремента

Это операции ++ и --. Так y++ (инкремент) является сокращенной записью y = y +1, аналогично и с операцией декремента (--).

Но с этими операциями есть одна тонкость. Они существуют в двух формах - префиксной (++y) и постфиксной (y++). Действие этих операций одно и то же - они увеличивают (операции декремента - уменьшают) свой операнд на 1, а вот результат у них разный. Префиксная форма в качестве результата выдает уже измененное на 1 значение операнда, а постфиксна -значение операнда до изменения.

```
a =
5; x
=
a++;
```

```
y =
++a;
```

В этом фрагменте x получит значение 5, а y - 7.

Операция целочисленного деления

Нужно учитывать, что деление одного целого на другое выдает целое, причем не округляет, а отбрасывает дробную часть.

Остаток от деления (значение по модулю)

В Java имеется операция %, которая обозначает остаток от деления.

Расширенные операции присваивания

Кроме обычной операции '=' в Java существуют операции +=, -=, *=, /= и др. Это сокращенные записи. Так a += b полностью эквивалентна a = a + b. Аналогично и с другими такими операциями.

Логические операции

```
!       - отрицание
&&      - логическое 'и'
||      - логическое
'или'
```

Операнды этих операций должны быть типа boolean, результат - boolean. Операции && и || имеют одну особенность - их правый операнд может и не вычислиться, если результат уже известен по левому операнду. Так, если левый операнд операции && - ложь (false), то правый операнд вычисляться не будет, т.к. результат все равно - ложь.

Это свойство нужно учитывать, особенно тогда, когда правый операнд содержит вызов некоторой функции.

Побитовые логические операции

Это операции

```
&    - побитовое 'и'
|    - побитовое 'или'
^    - побитовое 'исключающее
или' ~    - побитовое
отрицание
```

Они выполняются для каждой пары битов своих операндов.

Операции сдвига

```
<<     - сдвиг влево
>>     - сдвиг вправо
>>>    - беззнаковый сдвиг вправо
```

Эти операции сдвигают значение своего левого операнда на число бит, заданное правым операндом.

Условная операци

Это единственная тернарная операция, т.е. операция, имеющая три операнда. Соответственно, для нее используется не один знак операции, а два.

```
        <условие> ? <выражение1> : < выражение2>
```

Если <условие> истинно, то результатом будет < выражение1>, иначе < выражение2>.

Например, "a < b ? a : b" вычисляет минимум из a и b.

Операция приведения типов

Это очень важная операция. По умолчанию все преобразования, которые могут привести к проблемам, в Java запрещены. Так, нельзя long-значение присвоить intоперанду. В тех случаях, когда это все же необходимо, нужно поставить явное преобразование типа.

Например, пусть метод f(...) выдает long.

```
int x = (int)f(10);
```

Здесь (int) - это операция преобразования типа. Операция преобразования типа обозначается при помощи имени типа, взятого в скобки.

Эта операция применима не только к базовым типам, но и к классам. Мы разберем это подробнее, когда будем рассматривать наследование.

Литералы (константы)

Арифметические

Примеры арифметических констант

```
- 10
- 010 - это 8
- 0123 - это 83  (1*64 + 2*8 + 3)
- 0x10 - это 16
- 0x123 - это 291 (1*256 + 2*16 +3)
- 1e5 - это 100000
- 1.23e-3 - это 0.00123
```

Для указания типа константы применяются суффиксы: l (или L) - long, f (или F) - float, d (или D) - double. Например, 1L - единица, но типа long.

Логические литералы

Логические литералы - это true (истина) и false (ложь)

Строковые литералы

Записываются в двойных кавычках, например

"это строка"

Символьные литералы

Записываются в апострофах, например 'F', 'ш'.

В строковых и символьных литералах есть правила для записи спец. символов. Во-первых, есть набор предопределенных спец. символов. Это

```
- '\n' - конец строки (перевод строки)
- '\r' - возврат каретки
- '\t' - табуляция
```

и ряд других.
Во-вторых, можно явно записать код символа (нужно только знать его). Запись кода обычно выполняется в восьмеричной системе: '\001' - символ с кодом 1 и т.п.

Операторы (statements)

Оператор - выражение

Синтаксис
<выражение>;
Такой оператор состоит из одного выражения, в конце стоит ';'. Его действие состоит в вычислении выражения, значение, вычисленное данным выражением, теряется. Т.е. обычно такое выражение содержит операцию присваивания, хотя это по синтаксису не обязательно.

Примеры

```
a = 0; x = (a >
b ? a : b);
cnt++;
```

Условный оператор (if)

Синтаксис

```
if ( <условие> )
       <оператор1>
[else
       <оператор2>]
```

Здесь <условие> - это логическое выражение, т.е. выражение, возвращающее true или false. Как видно из синтаксиса, часть else вляется необязательной. После if и после else стоит по одному оператору. Если нужно поместить туда несколько операторов, то нужно поставить **блок**. **Блок** начинается с '{' и заканчивается '}'. В Java принято блок ставить всегда, даже если после if или else стоит один оператор.

Примеры

```
if ( a > b )
{     x = a;
} else {
x = b; }
if ( flag ) {
flag = false;
init(); }
```

В последнем примере flag - логическая переменная или поле, init() -метод, вызываемый, если флаг равен true (говорят, "если flag установлен").

Оператор return (уже рассматривали)

Оператор цикла по предусловию (while)

Синтаксис

```
while ( <условие> )
<оператор>
```

Как и в случае оператора if, в Java принято <оператор> заключать в фигурные скобки.

Пример

```
int i = 0; while (
more ) {     x /=
2;     more = (++i
< 10);
}
```

В этом примере more должна быть логической переменной или полем, x - некоторой арифметической переменной или полем. Цикл выполняется 10 раз.

Оператор цикла по постусловию (do while)

Синтаксис

```
do
        <оператор>
while ( <условие> );
```

Оператор цикла по постусловию отличается от оператора цикла по предусловию только тем, что в нем виток цикла выполняется всегда как минимум один раз, в то время как в операторе по предусловию может не быть ни одного витка цикла (если условие сразу ложно).

Пример

```
int i =
0; do {
x \= 2;
    more = (++i < 10);
} while (more
);
```

Содержательно, данный пример эквивалентен предыдущему.

Оператор цикла "со счетчиком" (for)

Синтаксис

```
for (<инициализация>; <условие>; <инкремент> )
     <оператор>
```

Заголовок такого цикла содержит три выражения (в простейшем случае). Из наименования оператора можно понять, что он служит для организации цикла со счетчиком. Поэтому выражение <инициализация> выполняется один раз перед первым витком цикла. После каждого витка цикла выполняется выражение <инкремент>, а потом выражение <условие>. Последнее выражение должно быть логическим и служит для задания условия продолжения цикла. Т.е. пока оно истинно, витки цикла будут продолжаться.

Для удобства составления операторов цикла со счетчиком в данной конструкции введено множество расширений.

- <инициализация> может быть не выражением, а описанием с инициализацией типа "int i = 0".
- <инициализация> может быть списком выражений через запятую, например, "i = 0, r = 1" .
- <инкремент> тоже может быть списком выражений, например, "i++, r*=2"
- Все составляющие (<инициализация>, <условие> и <инкремент>) являются необязательными.

Для выражения <условие> это означает, что условие считается всегда истинным (т.е. выход из цикла должен быть организован какими-то средствами внутри самого цикла). Т.е. допустим и такой цикл (бесконечный цикл):

```
for (;;) {
   . . .
}
```

Пример

```
for (int i = 0; i < 10;
i++)     x /=2;
```

Это самая "экономичная" реализация цикла из предыдущих примеров.

Операторы break и continue

В Java нет операторов goto. Как известно, goto приводит к появлению неструктурированных программ. Операторы break и continue являются структурированными аналогами goto.

Они могут применяться в циклах, а break еще и в операторе выбора (switch). Выполнение оператора break приводит к немедленному завершению цикла. Оператор continue вызывает окончание текущего витка цикла и начало нового. Проверка условия в этом случае все же выполняется, так что continue может вызвать и окончание цикла.

Пример

```
for ( int i = 0; ;i++ ) {
if ( oddOnly && i%2 == 0 )
continue;     y = (x +
1)/x;     if ( y - x <
0.001 )           break;
x = y; }
```

Здесь oddOnly - логическая переменная. Если она установлена, то все витки цикла с четными номерами пропускаются с использованием оператора continue;

Условие окончания цикла в данном примере проверяется в середине цикла и, если оно выполнено, то цикл прекращается при помощи оператора break.

При вложенных циклах операторы break и continue могут относиться не только к тому циклу, в котором они расположены, но и к охватывающему циклу. Для этого охватывающий оператор цикла должен быть помечен меткой, которая должна быть указана в операторе break или continue. Например,

```
lbl: while ( ... ) {
   . . .
   for ( ... ) {
      . . .
if ( ... )
break lbl;
   }
. .
. }
```

Здесь оператор break вызовет прекращение как цикла for, так и while.

Оператор выбора (switch)

Служит для организации выбора по некоторому значению одной из нескольких ветвей выполнения. Синтаксис

```
switch ( <выражение> ) {
case <константа1>:
<операторы1>        case
<константа2>:
<операторы2>
       . . .
       [default:
             <операторы_D>]
}
```

Выражение должно выдавать целочисленное или символьное значение, константы должны быть того же типа, что и значение этого выражения.
Элементы "case <константа>:" являются метками перехода, если значение выражения совпадает с константой, то будет осуществлен переход на эту метку.
Если значение выражения не совпадает ни с одной из констант, то все зависит от наличия фрагмента default. Если он есть, то переход происходит на метку default, если его нет, то весь оператор switch пропускается.

- В операторе switch фрагменты case не образуют какие-либо блоки. Если после последнего оператора данного case-фрагмента стоит следующий case, то выполнение будет продолжено, начиная с первого оператора этого case-фрагмента.
- В силу этого в операторе switch обычно применяется оператор break. Он ставится в конце каждого case-фрагмента.

Пример (файл SymbolTest.java)

Рассмотрим демонстрационную программу, в которой использованы операторы for и switch.

Эта программа генерирует случайным образом 100 символов латинского алфавита и классифицирует их как "гласные", "согласные" и "иногда гласные".
В последнюю категорию отнесены символы 'y' и 'w'.

```
public class SymbolTest {
     public static void main(String[]
args) {          for ( int i = 0; i <
100; i++ ) {
             char c = (char)(Math.random()*26 + 'a');
             System.out.print(c + ":
");              switch ( c ) {
case 'a': case 'e': case 'i':
case 'o': case 'u':

System.out.println("гласная");
break;              case 'y': case 'w':
                System.out.println("иногда
гласная");                  break;
default:
                System.out.println("согласная");
             }
         }

     }

}
```

В данном примере есть несколько новых для нас элементов.

- Используется метод random() класса Math. Посмотрим документацию по классу Math и разберемся, что он делает.
- В операторе

```
char c = (char)(Math.random()*26 + 'a');
```

  производится сложение арифметического значения с символом. При таком сложении в Java символ преобразуется в число, которое равно коду этого символа.

- В операторе

```
System.out.print(c + ": ");
```

  используется не println(...), а print(...). Метод print(...) отличается от println(...) только тем, что не переводит печать на новую строку, так что следующий оператор print(...) или println(...) продолжит печать в той же строке.

Следует также обратить внимание на фрагменты case. Формально здесь 7 таких фрагментов, но 5 из них не содержат никаких операторов. Так что, можно считать, что здесь 2 case-фрагмента, но каждый из них имеет несколько меток case.

Конспект лекций по Java. Лекция 5

Массивы в Java

Мы не рассмотрели еще массивы. В Java есть как одномерные, так и многомерные массивы. Но реализация массивов в Java имеет свои особенности. Во-первых массив в Java это объект. Этот объект состоит из размера массива (поле length) и собственно массива.

Рассмотрим сначала простейшие одномерные массивы базовых типов. int intAry[]; или

int[] intAry;

Это описание переменной (или поля) intAry - ссылки на массив. Соответственно, в этом описании размер массива не указывается и сам массив не создается. Как и любой другой объект массив должен быть создан операцией **new**.

```
intAry = new int[10];
```

Создает массив из 10 целых и адрес этого массива заносит в intAry. Как обычно в Java допустимо объединять описание и инициализацию. int[] intAry = new int[10];

Для массивов допустима инициализация списком значений. int intAry[] = {1, 2, 3, 4};

Здесь описан массив из 4-х элементов и сразу определены их начальные значения.

Элементы массивов в Java нумеруются с 0. При обращении к элементу массивы его индексы задаются в квадратных скобках. Например.

```
x = a[3];  // x присваивается значение 3-го (4-го по порядку) элемента
массива a
intAry[i] = 0;  // i - му элементу массива intAry
присваивается значение 0
```

Java жестко контролирует выход за пределы массива. При попытке обратиться к несуществующему элементу массива возникает IndexOutOfBoundException.

Для двумерных массивов ставится не одна пара скобок, а две, для трехмерных - три и т.д. Например.

```
s = someAry[i][0];
tAry[i][j][k] = 10;
```

Как уже указывалось, массив является объектом, который, в частности, хранит поле length - размер массива. Это позволяет задавать обработку массивов произвольного размера. Рассмотрим пример метода, вычисляющего сумму элементов массива.

```
public double sum(double ary[]) {
double s = 0;
       for ( int i = 0; i < ary.length; i++ )
{              s += ary[i];
       }
return s;
}
```

Массивы объектов

Одномерный массив объектов - это массив ссылок на объекты. Соответственно, нужно создать как массив, так и сами объекты. Наиболее частая ошибка у начинающих при работе с массивами классов примерно следующая. Создается сам массив, например, `A[] a1 = new A[10];`

а потом сразу идет попытка работы с элементами этого массива. Но здесь построен только массив ссылок, а сами объекты еще не созданы.

Пусть у нас есть некоторый класс SomeClass и нужно построить массив из 4-х объектов этого класса.

Вариант 1. (явное создание)

```
SomeClass objAry[] = new
SomeClass[4]; for (int j = 0; j <
4; j++ )      objAry[j] = new
SomeClass();
```

Вариант 2. (использование списка инициализации)

```
SomeClass objAry[] = new SomeClass[] {
                                                new
                    SomeClass(),
                  new SomeClass(),
                  new SomeClass(),
                 new SomeClass(),
                                     };
```

Многомерные массивы

Они строятся по принципу "массив массивов". Разберемся с этим. Массив вляется объектом. Двумерный массив - это массив ссылок на объекты-массивы. Трехмерный массив - это массив ссылок на массивы, которые, в свою очередь, являются массивами ссылок на массивы.

Это выглядит несколько сложно. Но, к счастью, есть сокращенные варианты создания массива, которые позволяют сразу разместить все необходимые массивы ссылок. Кроме того, если удается использовать списки инициализации (а это возможно, когда размер массива заранее известен), то все гораздо проще. Пусть, например, требуется создать массив целых 3*3.

Вариант 1. (явное создание)

```
int ary[][] = new
int[3][3];
```

Вариант 2. (использование списка инициализации)

```
int ary[][] = new int[][] {
                            {1, 1, 1},
                            {2, 2, 2},
                            {1, 2, 3},
                           };
```

Внимание: в варианте 1 массив создан, но его элементы имеют неопределенное значение. Если попытаться их использовать, возникнет Exception.

Соответственно, все несколько усложняется при использовании массивов объектов. Создадим аналогичный массив объектов класса SomeClass.

Вариант 1. (явное создание)

```
SomeClass ary[][] = new SomeClass[3][3];
for ( int k = 0; k < 3; k++ )
    for ( int j = 0; j < 3; j++ )
ary[k][j] = new SomeClass();
```

Вариант 2. (использование списка инициализации)

```
SomeClass ary[][] = new SomeClass[][] {
        { new SomeClass(), new SomeClass(), new
SomeClass(), },
        { new SomeClass(), new SomeClass(), new
SomeClass(), },
        { new SomeClass(), new SomeClass(), new
SomeClass(), },
         };
```

Глядя на примеры со списком инициализации, можно задаться вопросом, что будет, если мы в одних строках зададим одно количество элементов, а в других - другое. Например.

```
int ary[][] = new int[][] {
                            {1, 1, 1, 1},
                            {2, 2, 2},
                            {1, 2, 3, 4, 5},
                            };
```

В Java такое допустимо и именно потому, что многомерный массив является массивом ссылок на массивы. Т.е. каждый массив следующего уровня является самостоятельным массивом и может иметь свой размер. Причем, создание таких "непрямоугольных" массивов возможно не только с использованием списка инициализации, но и явно.

```
int ary[][] = new
int[3][]; ary[0] = new
int[5]; ary[1] = new
int[2]; ary[2] = new
int[6];
```

Увлекаться такими "непрямоугольными" массивами не стоит. На практике очень редко встречаются задачи, в которых подобные возможности могут потребоваться. Но знать о них нужно. Хотя бы для того, чтобы понять смысл ошибки, возникшей в результате непреднамеренного создания подобного массива.

Присваивание и копирование

В приведенных выше примерах имя массива - это переменная-ссылка (или полессылка), содержащая адрес объекта массива. Поэтому присваивание таких переменных друг другу - это не копирование массивов, а копирование ссылок на объекты. Например.

```
int ary[][] = new int[][] {
                          {1, 1, 1, 1},
                          {2, 2, 2},
                          {1, 2, 3, 4, 5},
                          };

int copyAry[][] = ary;
```

В данном примере все абсолютно корректно, но copyAry - это не ссылка на копию массива, а еще одна ссылка на тот же массив. Для создания копии придется написать соответствующий метод.

В состав стандартной библиотеки Java входят разнообразные средства работы с массивами. В пакете java.util имеется класс Arrays, который обеспечивает множество полезных операций над массивами (см. документацию).

Резюмируем основные правила

- **1.** Массивы являются объектами специфической формы. В частности, любой массив имеет поле length, которое определяет его размер.
- **2.** Массивы индексируются от 0.
- 3. Java жестко контролирует выход за границы массива (прерывание IndexOutOfBoundException).
- 4. Массив элементарного типа, например int, - это действительно массив значений (т.е. массив целых чисел). Массив объектов - это массив ссылок на объекты. Т.е. недостаточно создать сам массив, нужно еще создать объекты, входящие в него.
- **5.** Существуют два способа создания массива - операцией new и явной инициализацией.
- **6.** Для многомерных массивов существует возможность задания разного размера массивов второго, третьего и т.д. измерений, но это "экзотика".

Конструкторы классов

Мы уже познакомились с понятием класса в Java. Мы определили, что класс является как бы шаблоном для создания объектов класса (экземпляров класса) и что класс состоит из полей и методов класса. Но мы еще не рассмотрели очень важную составляющую классов - **конструкторы класса**.

Конструктор класса - это специальный метод класса. Он вызывается при создании объекта класса. Конструкторы (в своем описании) отличаются от других методов класса тем, что их имя совпадает с именем класса. Кроме того, при описании любого метода класса, кроме конструктора, мы обязаны указать

тип возвращаемого значения, а если метод не возвращает никакого значение, то мы должны вместо типа явно указать void. При описании конструктора тип возвращаемого значения вообще не указывается.

Разберем более подробно, что происходит при создании объекта класса.

- Ищется класс объекта среди уже используемых в программе классов. Если его нет, то он ищется во всех доступных программе каталогах и библиотеках. После обнаружения класса в каталоге или библиотеке выполняется создание и инициализация статических полей класса. Т.е. для каждого класса статические поля инициализируются только один раз.
- Выделяется память под объект.
- Выполняется инициализация полей класса.

Отрабатывает конструктор класса.

Это не полная схема, а упрощенная.

Итак, что такое **конструктор**.

- 1. Это специальный метод класса.
- 2. Его имя совпадает с именем класса.
- 3. Конструктор не возвращает никакого значения.
- 4. Конструктор, как и любой другой метод, может иметь параметры.
- 5. Конструктор без параметров называется конструктором по умолчанию (default constructor).
- 6. В классе может быть несколько конструкторов. В этом случае они должны иметь разные наборы параметров.
- 7. Если в классе нет ни одного конструктора, то генерируется пустой конструктор по умолчанию. Если в классе есть хотя бы один конструктор, то конструктор по умолчанию не генерируется. До сих пор в примерах мы использовали примерно следующий синтаксис создания объектов.

```
SomeClass obj = new SomeClass();
```

В данном случае создается объект и при его создании используется конструктор без параметров (конструктор по умолчанию). Но возможен и другой вариант.

```
SomeClass obj = new SomeClass(1, 'a');
```

Здесь при создании объекта вызывается конструктор с двумя параметрами. Т.е. в классе SomeClass должен быть описан конструктор, имеющий один арифметический параметр (например, int) и один символьный параметр.

Для того чтобы обе вышеприведенные строки были корректными, описание класса SomeClass должно выглядеть примерно так.

```
class SomeClass {
   . . .     public SomeClass() {     // это конструктор
по умолчанию
          . . .
    }

    SomeClass( int a, char c) {   // это конструктор с двумя
параметрами
          . . .

}
. .
.
}
```

Вызов одного конструктора из другого

В Java есть одна удобная возможность, позволяющая сократить суммарный объем кода конструкторов. Обычно все конструкторы класса имеют общую часть кода, поскольку они должны выполнять одинаковые действия, которые отличаются только некоторыми деталями. Можно вынести эту общую часть кода в отдельный метод и вызывать его из всех конструкторов. Но есть и другая возможность. В Java можно вызвать один конструктор из другого. Для этого используется ключевое слово **this**. Рассмотрим это на примере.

```
public class Point {
private double x, y;

     public Point(double x, double
y) {            this.x = x;
this.y = y;

}
     public Point() {
           this(0, 0);     // вызов конструктора с двумя
параметрами
     }
. . .
}
```

В классе Point (точка) имеется два конструктора. Один, более общий, с двумя параметрами, предназначен для конструирования точки плоскости с заданными координатами. Другой, без параметров, строит точку с координатами (0, 0). В нем для экономии кода просто вызывается первый из конструкторов с параметрами (0,
0).

Есть одно ограничение на вызов одного конструктора из другого. Такой вызов должен быть первым оператором в вызывающем конструкторе.

Работа со строками (класс String)

Первый стандартный класс Java, который мы рассмотрим - это класс String. Этот класс определен в стандартной библиотеке Java. Он используется для работы со строками.

Откроем документацию по классу String. Во-первых, обратим внимание на конструкторы класса. В документации конструкторы класса описываются сразу после описания полей класса. Конструкторы класса String предоставляют широкие возможности конструирования строк. public String()
Создает пустую строку
public String(char[] value)
Создает строку из массива символов.
public String(byte[] bytes)
Создает строку из массива байт, преобразуя байты в символы в соответствии с кодировкой по умолчанию.

Есть и другие конструкторы класса String.

В силу важности строк в Java для класса String существуют расширенные возможности языка. По общим правилам создания объектов мы должны были бы при построении строки писать так String str = new String("какая-то строка");

Такая запись допустима, но существует ее упрощенный вариант: String str = "какая-то строка";

Для строк определена операция сложения, которая означает конкатенацию (сцепление) строк. Определена операция сложения строки с числом. При этом сначала число преобразуется в строку, а потом выполняетс конкатенация полученных строк.

Также определена операция сложения строки с любым объектом. Она выполняется так. Сначала для этого объекта вызывается метод toString(), потом выполняется конкатенация полученных строк. Метод toString() есть у всех объектов Java (рассмотрим подробнее при изучении наследования).

Примеры сложения строк с числами нам уже встречались - в операторах типа System.out.println("результат=" + x);

Вернемся к документации по классу String. Следует обратить внимание на следующие методы этого класса. public char charAt(int index)
Выбирает из строки символ с индексом index (символы индексируются от нуля).
public int compareTo(String anotherString) Сравнивает строку с другой строкой public int indexOf(int ch) Ищет символ в строке public int indexOf(String str)
Ищет указанную параметром строку в данной public int length() Возвращает длину строки
public String substring(int beginIndex, int endIndex)
Выделяет подстроку из строки
public String trim()
Удаляет из строки начальные и концевые пробелы
Набор методов valueOf(...) позволяет переводить значения различных типов в строки.

Конспект лекций по Java. Лекция 6

Практическая работа

Для повторения и закрепления материала предыдущих занятий рассмотрим пример класса, в котором выполняется работа с массивами, а также реализован статический и обычный метод.

Это класс DoubleVector (файл DoubleVector.java). Он предназначен для работы с векторами. В приведенном примере реализовано только скалярное умножение векторов, но при необходимости этот класс можно расширить другими операциями над векторами.

```
public class DoubleVector {

    private double[] vector = null;

    public DoubleVector(double[] vector) {
this.vector = vector;
    }

    /**
     * Скалярное произведение векторов
     */
    public double mult(DoubleVector anotherVector) {
double s = 0;
        for ( int i = 0; i < vector.length; i++ )
{             s += vector[i] *
anotherVector.vector[i];
        }
return s;

}
    public static double mult(DoubleVector a, DoubleVector b) {
return a.mult(b);

}
    public static void main(String[]
args) {         double[] a = {1, 2, 3,
4};         double[] b = {1, 1, 1, 1};
double[] c = {2, 2, 2, 2};
        DoubleVector v1 = new DoubleVector(a);
        DoubleVector v2 = new DoubleVector(b);
        DoubleVector v3 = new DoubleVector(c);
        System.out.println("v1*v2=" + v1.mult(v2));
        System.out.println("v1*v2=" + DoubleVector.mult(v1,
v2));
        System.out.println("v1*v3=" +
v1.mult(v3));     }

}
```

В классе DoubleVector есть поле - массив vector, конструктор для построения вектора из массива, два метода mult(...), один из которых статический, а также метод main(...) для проверки работоспособности класса.

Следует обратить внимание на методы mult(...). Первый из них предназначен для умножения данного вектора на другой вектор. Он используется в методе main(...) в строках

```
System.out.println("v1*v2=" +
v1.mult(v2));  и
System.out.println("v1*v3=" + v1.mult(v3));
```

Второй для умножения двух векторов. Он используется в строке

```
System.out.println("v1*v2=" + DoubleVector.mult(v1, v2));
```

Статический метод mult(...) реализован на базе обычного метода, путем возврата "a.mult(b)". Альтернативой является реализация алгоритма умножения в статическом методе и вызов его из обычного метода. Это выглядело бы так.

```
    public double mult(DoubleVector anotherVector) {
return mult(this, anotherVector);

}
    public static double mult(DoubleVector a, DoubleVector b) {
double s = 0;
        for ( int i = 0; i < a.vector.length; i++ )
{             s += a.vector[i] * b.vector[i];
        }
return s;
    }
```

- Данный класс не лишен недостатков. Он предполагает, что при умножении оба вектора имеют одинаковую длину. Для устранения подобных недостатков в Java следует применять аппарат исключительных ситуаций (exceptions), но мы его еще не рассматривали.

Знакомство с библиотеками и пакетами.

Библиотека Java - это сборник классов. Если программе нужен какой-то класс, то нужно подключить библиотеку, в которой этот класс находится. Для этого либо устанавливается переменная окружения CLASSPATH, либо задается параметр вызова компилятора и JVM. Для транслятора (javac.exe) - это параметр -classpath, для JVM (java.exe) - это параметр -cp.

Например,

```
rem трансляция
d:\jdk1.3\bin\javac.exe -classpath .;d:\jdk1.3\jre\lib\rt.jar My.java
rem выполнение
d:\jdk1.3\bin\java.exe -cp .;d:\jdk1.3\jre\lib\rt.jar
My
```

Обычно физически библиотека - это jar-файл (rt.jar, например). Но свою личную библиотеку можно сделать и просто в каком-либо каталоге. Кроме того, библиотека может быть zip-файлом.

Но библиотека является довольно крупным хранилищем классов. На практике требуется какой-то дополнительный механизм разбиения всего множества классов, хранящихся в библиотеке на отдельные части. В Java таким механизмом являются пакеты.

Пока познакомимся с использованием пакетов из стандартной библиотеки Java. Лучше всего обратиться к документации по API Java. Полный список пакетов стандартной библиотеки Java:

```
java.applet java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font
java.awt.geom
java.awt.im
java.awt.im.spi
java.awt.image
java.awt.image.rendera
ble java.awt.print
java.beans
java.beans.beancontext
java.io java.lang
java.lang.ref
java.lang.reflect
java.math
```

```
java.net java.rmi
java.rmi.activation
java.rmi.dgc
java.rmi.registry
java.rmi.server
java.security
java.security.acl
java.security.cert
java.security.interfaces
java.security.spec
java.sql java.text
java.util java.util.jar
java.util.zip
javax.accessibility
javax.naming
javax.naming.directory
javax.naming.event
javax.naming.ldap
javax.naming.spi
javax.rmi javax.rmi.CORBA
javax.sound.midi
javax.sound.midi.spi
javax.sound.sampled
javax.sound.sampled.spi
javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.plaf.multi
javax.swing.table
javax.swing.text
javax.swing.text.html
```

```
javax.swing.text.html.pars
er javax.swing.text.rtf
javax.swing.tree
javax.swing.undo
javax.transaction
org.omg.CORBA
org.omg.CORBA_2_3
```

```
org.omg.CORBA_2_3.portable
org.omg.CORBA.DynAnyPackag
e org.omg.CORBA.ORBPackage
org.omg.CORBA.portable
org.omg.CORBA.TypeCodePack
age org.omg.CosNaming
org.omg.CosNaming.NamingContextPack
age org.omg.SendingContext
org.omg.stub.java.rmi
```

Нужно разобраться с именами пакетов. Как видно, имя составное, разделенное точками. Это связано с общепринятым в Java принципом построения имен пакетов. Этот принцип состоит в том, что в имени пакета присутствует Internetадрес разработчика пакета в обратном порядке. На примере:

Пусть Ваш адрес petr@provider.da . Тогда все имена пакетов Ваших приложений должны начинаться с "da.provider.petr.". Так, для пакета, содержащего вспомогательные сервисные классы подойдет имя "da.provider.petr.util".

Кроме того, с именем пакета связана структура каталогов, в которых должны размещаться классы. Это будет рассмотрено подробнее позже, когда будем рассматривать создание собственных пакетов.

Некоторые важные пакеты Java

- java.lang - "самый базовый" из всех базовых пакетов. Без него Java не работает.
- java.io - пакет поддержки ввода/вывода.
- java.util - классы поддержки коллекций объектов, работа с календарем, и др. полезные классы.
- java.awt - пакет AWT: поддержка визуального программирования (базовый пакет).
- javax.swing - пакет SWING: новый пакет визуального программирования. Появился в Java2, базируется на AWT и функционально замещает его (не полностью).
- java.applet - пакет для поддержки создания аплетов.

Использование пакетов в программах

Для того чтобы класс из библиотеки мог быть использован в программе, его нужно подключить. Для этого в начале java-файла нужно поместить оператор "import". Например, пусть требуется использовать класс ArrayList в разрабатываемом нами классе Ex1 (ArrayList находится в пакете java.util).

Тогда файл Ex1.java может начинаться примерно так:

```
// Ex1.java
 import
java.util.ArrayList;
  . . .
```

далее в программе мы уже можем использовать ArrayList. Например,

```
  ArrayList objList = new ArrayList();
  . . .
```

Если в одном программном модуле требуется несколько классов из одного пакета, то можно подключить весь пакет, например, import java.util.*;

- Пакет java.lang можно не подключать, он подключается автоматически.

Создание своих собственных пакетов

Для создания собственного пакета нужно

1. Определиться с именем пакета.

- Пусть, например, Internet-адрес равен my@prov.ua и мы хотим создать пакет вспомогательных программ (подходящее имя - util). Тогда полное имя пакета должно быть ua.prov.my.util.

- 2. Создать необходимую структуру каталогов. Нужно выбрать или создать каталог, где мы будем хранить все свои пакеты. Пусть, например, это каталог c:\javaproj. В нем нужно создать каталог ua, в каталоге ua создать каталог prov, в нем - my, и наконец в prov - util. Все java-файлы создаваемого пакета должны помещаться в каталог util. Т.е. структура имени пакета определяет структуру каталогов.

- **3.**Указать путь к пакету в bat-файлах трансляции и выполнения. Для удобства работы со своими пакетами желательно прописать путь к пакетам в bat-файлах трансляции и выполнения java-программ. Приведем примеры batфайлов j.bat и jr.bat, в которых указан путь к выбранному нами для хранения всех пакетов каталогу c:\javaproj. Файл для трансляции (j.bat)

```
REM Компилятор JAVA
set
JDKHOME=d:\jdk1.3
set CLASSPATH=.;%JDKHOME%\jre\lib\rt.jar;c:\javaproj
d:\jdk1.3\bin\javac %1 %2 %3 %4 %5
```

Файл для выполнения (jr.bat)

```
REM Запуск программы на JAVA
set JDKH=d:\jdk1.3
set
CLASSPATH=.;%JDKH%\jre\lib\rt.jar;%JDKHOME%\jre\lib\i18n.jar;c:\javap
roj
%JDKH%\jre\bin\java -cp %CLASSPATH% %1 %2 %3 %4 %5 %6
```

- **4.** В java-файлах пакета указать оператор package. Первым оператором в каждом java-файле пакета должен быть оператор package, в котором указано имя данного пакета. В нашем случае это должен быть оператор вида

package ua.prov.my.util;

Реализуем простой пример. Создадим в каталоге util такой файл S.java.

```
package ua.prov.my.util;

public class S {
     public static void o(String
str) {
          System.out.println(str);
    }

}
```

Оттранслируем его командой
j S.java

Теперь вернемся к примеру в начале занятия и модифицируем его.

1. В начало файла вставим
import ua.prov.my.util.S;

2. Строки

```
System.out.println("v1*v2=" + v1.mult(v2));
System.out.println("v1*v2=" + DoubleVector.mult(v1,
v2));
System.out.println("v1*v3=" + v1.mult(v3));
```

заменим на

```
S.o("v1*v2=" + v1.mult(v2));
S.o("v1*v2=" + DoubleVector.mult(v1, v2));
S.o("v1*v3=" + v1.mult(v3));
```

3. Оттранслируем и запустим измененную программу. Если мы все сделали правильно, то она будет работать как и прежде.

Можно сказать, что мы создали библиотеку (это каталог c:\javaproj). В дальнейшем мы можем расширять ее новыми пакетами и классами.

Практическая работа.

То, что мы изучили, достаточно для первого знакомства с визуальным программированием на Java. Некоторые элементы программ будут пока не ясны и мы будем использовать их *as is*. Но, в основном, приводимые далее программы могут быть поняты на основе изученного материала и могут быть использованы как прототипы для построения своих программ в стиле "сборка из известных компонент".

При создании визуальных приложений мы будем, в основном, использовать пакет javax.swing, а также java.awt и java.awt.event.

Рассмотрим первый пример диалоговой программы (файл Dialog1.java)

```
// Dialog1.java
// Первый пример визульного приложения на Java.
 import java.awt.*;
import
java.awt.event.*;
import
javax.swing.*;

public class Dialog1 {

  public static void main(String[] args) {
// фрагмент as is (1)
try  {
```

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName(
));
    }
    catch(Exception e) {
    }
//

    JFrame frm = new JFrame("Первое визуальное приложение");
frm.setSize(300, 200);
    Container c = frm.getContentPane();
c.add(new JLabel("Hello, привет"));

// фрагмент as is (2)
    WindowListener wndCloser = new WindowAdapter() {
public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
};
    frm.addWindowListener(wndCloser);
/
/
    frm.setVisible(true);
```

```
    }
}
```

Краткие пояснения к примеру

1. Здесь использованы следующие классы стандартной библиотеки Java

   - UIManager - рассмотрим позднее;
   - JFrame - позволяет сформировать основное окно приложения. Все остальные визуальные компоненты помещаются внутрь этого окна.
   - Container - класс для визуальных классов-контейнеров, т.е. визуальных компонент, которые могут внутри себя содержать другие визуальные компоненты.
   - JLabel - класс для создания меток.
   - WindowListener и WindowAdapter - as is.

2. "фрагмент as is (1)" обеспечивает Windows Look&Fill. Закомментируем его и посмотрим, что получится.
3. "фрагмент as is (2)" обеспечивает завершение всего приложения в случае, если закрылось главное окно приложения. Тоже полезно закомментировать и посмотреть, что получиться.
4. Строки

```
JFrame frm = new JFrame("Первое визуальное приложение");
frm.setSize(300, 200);
```

имеют очевидный смысл. (см.документацию по JFrame в пакете javax.swing).

5. Строки

```
Container c = frm.getContentPane();
c.add(new JLabel("Hello, привет"));
```

требуют пояснения. Главное окно (JFrame) устроено сложным образом. Оно состоит из ряда элементов. Тот элемент, в который следует добавлять другие визуальные компоненты, может быть получен при помощи метода getContentPane(). Мы запоминаем ссылку на него в переменной 'c' и потом используем его во второй строке фрагмента, которая просто создает и добавляет (метод add) метку на окно.

- При добавлении визуальных компонент в Java не указывается ни их позиция, ни размер. Это связано с тем, что все визуальные приложения Java масштабируемы и для обеспечения масштабируемости используются другие принципы размещения визуальных элементов, а именно, используются различные Layout'ы.

6. Строка

```
frm.setVisible(true);
```

выводит окно на экран и активизирует диалог с пользователем.

Конспект лекций по Java. Лекция 7

Наследование классов

Наследование классов (*inheritance*) один из существенных атрибутов ООП (объектноориентированного программирования). Оно позволяет строить новые классы на базе существующих, добавляя в них новые возможности или переопределяя существующие.

Что такое наследование.

Пусть есть класс A, он имеет поля a1, a2, ..., an и методы f1(), f2(), ... , fm(). Тогда мы можем на его основе построить класс B. Класс B наследует все поля и методы класса A (за исключением конструкторов). Кроме того, в B можно:

- добавить новые поля;
- добавить новые методы;
- переопределить какие-либо методы класса A.

Это в основном, без деталей.

Синтаксис:

```
class B extends A {    .
. .    // тело класса B
}
```

Внутри записываются "дополнения и изменения", вносимые классом B.

Класс A называют базовым классом или суперклассом (superclass), иногда - родительским классом, предком. B - порожденным, дочерним, подклассом (subclass), классом-потомком.

В свою очередь, класс A может быть порожден на базе другого класса, тогда этот класс является предком как для A, так и для B.

От одного класса может быть порождено произвольное количество новых классов. В результате получается иерархия классов, порожденных один от другого. Пример (наследование полей)

```
class Base {
int a, b, c;
     . . .
}
class Derived extends B {
long d, e;
     . .
. }
```

Объекты класса Base имеют три поля (a, b и c), объекты класса Derived - пять полей (a, b, c, d и e). Пример (наследование методов)

```
 class Base {
 int f() {
 . . .      }
      void g(int p) {
           . . .
      }




}
 class Derived extends B {
       long
 f1() {
           . . .
      }

 }
```

Класс Base имеет два метода (f() и g(...)), класс Derived - три метода (f(), g(...) и f1()).

Пример (переопределение методов)

```
 class Base {
 int f() {
 . . .      }
 void g(int p) {
           . . .
      }
```

```
}
class Derived extends B {
      int
f() {
. . .
     }

}
```

Класс Base имеет два метода (f() и g(...)) и класс Derived тоже имеет два метода (f() и g(...)). Но для объектов класса Derived будет вызываться не метод f() из Base, а метод f() из Derived. На английском переопределение методов называется overriding.

При переопределении (overriding) метод в порожденном классе должен иметь в точности то же описание, что и метод в базовом классе. В частности, попытка описать "long f()..." или даже "private int f()..." привела бы к ошибке при трансляции программы.

Класс Object

В Java фактически все классы строятся на базе наследования, т.к., даже если не написать "extends имя_класса", будет подразумеваться extends Object. Т.е. класс Object является базовым классом для всех классов Java.

Класс Object имеет ряд методов, при наследовании методы базового класса наследуются его потомками. Следовательно, все классы Java имеют как минимум те методы, которые есть в классе Object.

Взглянем на документацию по классу Object. В классе Object определены методы, большая часть из которых имеют непонятное для нас (на текущий момент) назначение. Но некоторые из них мы можем уже сейчас рассмотреть. public String toString()
Предназначен для формирования некоторого текстового представления объекта. protected Object clone() throws CloneNotSupportedException
Предназначен для создания копии объекта. public boolean equals(Object obj) Позволяет сравнивать объекты. public final Class getClass()
Выдает класс данного объекта в виде объекта класса Class.
protected void finalize() throws Throwable
Этот метод вызывается при удалении объекта из памяти сборщиком мусора. Все указанные методы, кроме `getClass` и, частично, `toString`, не предназначены для непосредственного использования (в том смысле, что их нужно переопределить в порожденных классах).

Метод `getClass` возвращает специальный объект класса `Class`, содержащий много полезной информации о классе объекта. Метод `toString` по умолчанию выдает полное имя класса объекта и его адрес, что можно использовать при отладке программы.

Если почитать документацию по остальным методам, то мы увидим, что она, в основном, определяет, что должен делать тот или иной метод и лишь в конце описывается, как этот метод реализован в классе Object.

Разберем, например, метод `equals`. Его назначение - сравнивать объекты на равенство. Наличие его в классе Object (базовом для всех остальных классов) позволяет нам применять этот метод для любых объектов, что очень удобно. Но в классе `Object` он реализован "самым жестким" образом - как сравнение адресов объектов. Т.е. при сравнении двух объектов мы получим равенство только в том случае, если на самом деле это один и тот же объект. Естественно, что чаще всего нам потребуется какая-то другая реализация данного метода.

Наследование классов имеет много связанных с ним нюансов и особенностей. И в дальнейшем рассмотрении мы будем постепенно разбираться с ними.

Инициализация полей при наследовании классов

Уточним определение инициализации объектов в случае наследования классов.

Все действия по инициализации выполняются этап за этапом в порядке наследования классов, т.е. сначала п.1 для класса `Object`, потом п.1 для следующего класса и т.д., потом п.2 в том же порядке и т.д.

- При первом обращении к классу выделяется память под статические поля класса и выполняется их инициализация.
- Выполняется распределение памяти под создаваемый объект.
- Выполняются все инициализаторы нестатических полей класса.
- Выполняется вызов конструктора класса.

Рассмотрим абстрактный пример.

```
class A {
   ...
   A() {...}
   ...
}  class B
extends A {
   ...
   B() { ... }
   ...
}
```

где-то

```
B b = new B();
```

При этом сначала выполнится конструктор A(), потом конструктор B().

В данном примере класс A имеет конструктор по умолчанию. Но, что делать, если базовый класс, например, не имеет конструктора по умолчанию, или же нужно, чтобы при создании объекта отработал не конструктор по умолчанию, а какойлибо другой конструктор.

В Java эта ситуация предусмотрена. Используя ключевое слово super, можно в конструкторе порожденного класса вызвать нужный конструктор базового класса.

Пример

```
class X {
X(int a) { ...
}
   ...
}  class Y
extends X {
Y() {
super(0);
         ...

}
...
}
```

Ключевое слово super может использоваться и для явного вызова методов базового класса. Это необходимо, если некоторый метод базового класса был переопределен в порожденном классе.

Пример

```
class Base {
int x = 1;
long y;

    Base(long y) {
this.y = y;
    }

Base() {
          this(0);   // вызов конструктора Base(long y)
    }      public
long f() {
return x*y;
    }
```

```
}
class Derived extends Base {

    String name = "";

    Derived(String name, long par) {
          super(par);   // вызов конструктора Base(long y)
this.name = name;

}
    public long g(int r) {
         return r+super.f();   // вызов метода f() класса
Base
    }      public
long f() {
x++;
return 2*y;
    }
}
```

Рассмотрим подробно, из каких полей и методов состоит класс Derived. Из класса Base в него вошли поля x и y и метод f(). Плюс в

нем определено поле name и методы g(...) и f(), причем метод f() переопределяет одноименный метод класса Base.

В нашем примере, в методе g(...), требуется вызвать не метод f() класса Derived, а метод f() класса Base. При этом применяется ключевое слово super. Без него вызвался бы метод f() класса Derived.

Контрольная задача

Рассмотрим пример использования класса Derived. Требуется определить, что произойдет при выполнении такого фрагмента.

```
Derived d = new Derived("test",
10); long c = d.g(5); long p =
d.f();
```

В Java нет множественного наследования

В отличие от C++ в Java нет множественного наследования. Т.е. у класса может быть только один базовый класс. Соответственно отношение наследования формирует строгую иерархию классов - иерархию наследования (дерево классов). Это хорошо видно на примере документации.

Откроем документацию по стандартной библиотеке Java. Вверху любой страницы имеется банер с гиперссылками. Кликнем на ссылке "Tree". Мы увидим дерево классов стандартной библиотеки
Java.

Практическая работа

Вернемся к примеру Dialog1. Его можно реализовать иначе, если использовать аппарат наследования.

```
// Dialog2.java
// 2-й пример визульного приложения на Java.
 import java.awt.*;
import
java.awt.event.*;
import
javax.swing.*;

public class Dialog2 extends JFrame {

  Dialog2() {
     super("Первое визуальное приложение");

try  {
```

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelC
las sName());      }
     catch(Exception e) {

}
    setSize(300, 200);
    Container c = getContentPane();
    c.add(new JLabel("Hello, привет"));
    WindowListener wndCloser = new WindowAdapter() {
public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
};
    addWindowListener(wndCloser);

setVisible(true);

}
  public static void main(String[] args) {
 new Dialog2();
   }
 }
```

Этот пример более соответствует стилю Java, чем Dialog1.

Рассмотрим более сложный пример.

```
// Dialog3.java
// Визульное приложения с текстовой областью.
 import java.awt.*;
import
java.awt.event.*;
import
javax.swing.*;

public class Dialog3 extends JFrame {

  JTextArea txt;

  Dialog3() {
     super("Визульное приложения с текстовой областью");

try  {
```

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelC
las sName());      }
     catch(Exception e) {

}
    setSize(400, 200);
    Container c = getContentPane();
    c.add(new JLabel("Hello, привет"), BorderLayout.NORTH);
// 0
```

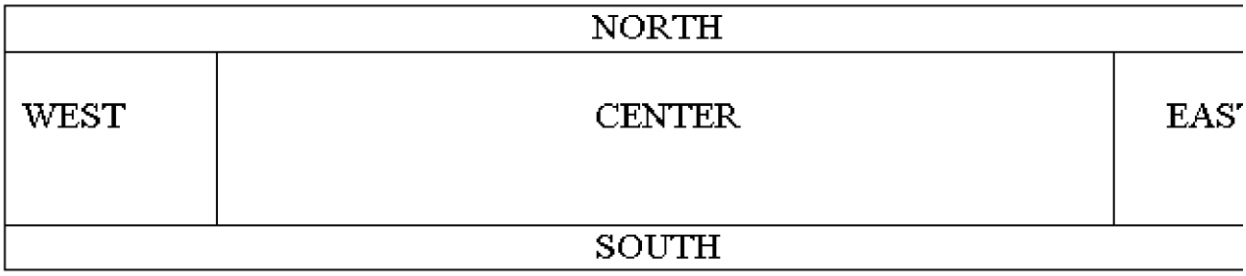
```
    txt = new JTextArea(5, 30);
// 1
    JScrollPane pane = new JScrollPane(txt);
// 2
    c.add(pane, BorderLayout.CENTER);
// 3

    WindowListener wndCloser = new WindowAdapter() {
public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
};
    addWindowListener(wndCloser);

    setVisible(true);

}
  public void test() {
txt.append("Первая строка\n");
txt.append("Вторая строка\n");

}
  public static void main(String[] args) {
Dialog3 d = new Dialog3();
    d.test();
  }
}
```

Этот пример требует некоторых пояснений по принципам организации диалога в Java. В отличие от других языков, в частности VB, в Java все визуальные компоненты масштабируемы.
Поэтому при их добавлении на окно приложения нельзя указать их координаты и размеры. Вместо этого используется понятие Layout'a ("разместитель", компоновщик). В этом примере задействован BorderLayout (он является Layout'ом по умолчанию для JFrame). Потом мы подробнее познакомимся с этим понятием. В данном случае просто разберем, что обеспечивает BorderLayout и как с ним работать.

Компоновщик BorderLayout разбивает область окна (панели) на следующие части.

Он позволяет добавлять компоненты в любую из этих частей. При добавлении (метод add(...)) нужно указать в какую часть панели мы добавляем компоненту (см.строки 0 и 3).

Класс JTextArea позволяет создать многострочную область ввода/вывода данных. Для того, чтобы эта область была скроллируемой, дополнительно используется JScrollPane, внутрь которого помещается объект txt класса JTextArea. !!!
На панель окна мы заносим не txt, а объект pane класса JScrollPane.



Конспект лекций по Java. Лекция 8

Модификаторы доступа при наследовании

На 3-ем занятии мы рассматривали описатели ограничения доступа (или, более кратко, модификаторы доступа). Несколько моментов в этом рассмотрении остались за рамками наших возможностей. Теперь, после знакомства с наследованием классов, мы можем доопределить их смысл. Поля и методы классов могут быть описаны как

- **public** - означает, что данный элемент доступен без каких-либо ограничений;
- **private** - доступ разрешен только из данного класса;
- **protected** - доступ разрешен из данного класса и из всех классов-потомков
- *без описателя* - доступ разрешен из всех классов данного пакета

Т.е. `public`-поля и методы доступны без ограничений во всех классах; `private` доступны только в данном классе и недоступны в порожденных классах; `protected` отличаются от `private` тем, что доступны в порожденных классах, а поля и методы без описателя доступны только в классах данного пакета вне зависимости от наследования.

Наследование классов - инструмент построения абстракций

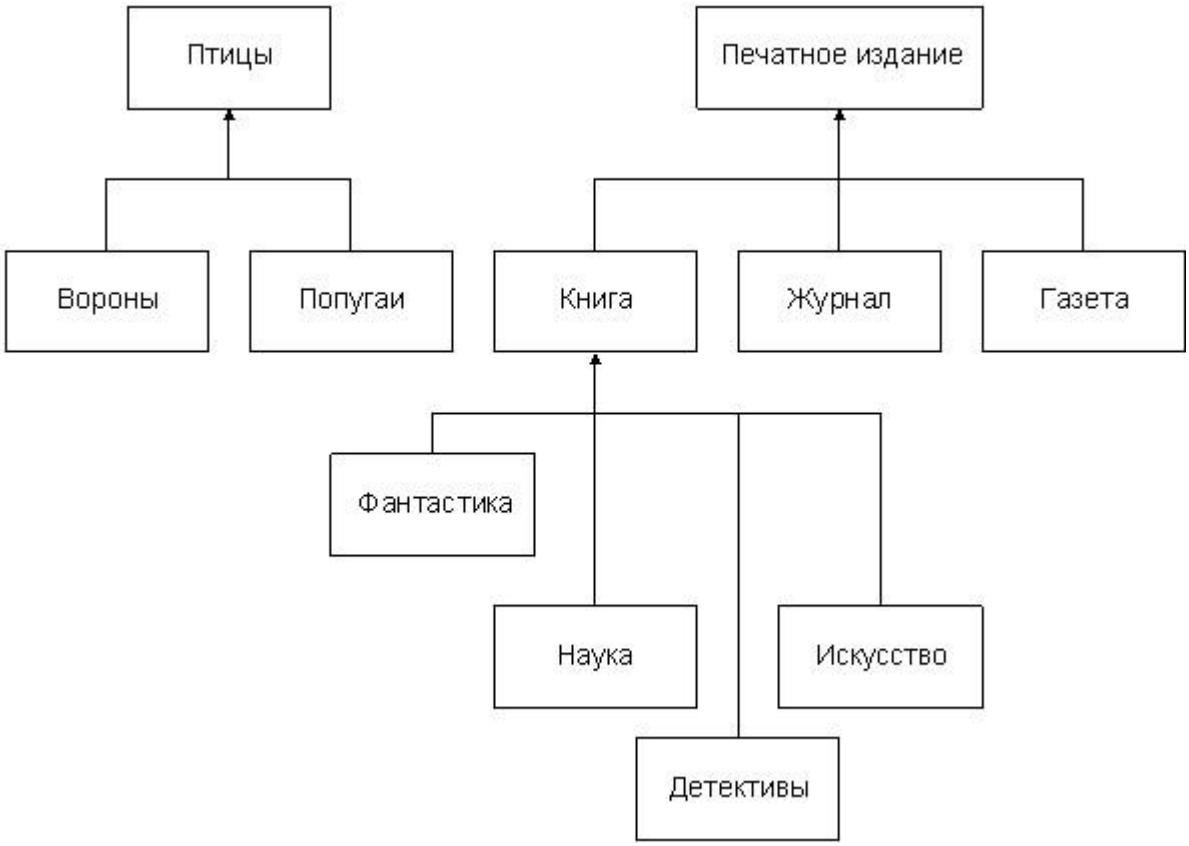
(Небольшое теоретическое отступление)

Вместе с другими категориями ООП (полиморфизм, инкапсуляция) наследование служит инструментом построения абстракций. Причем, приемы, применяемые в ООП, не совсем обычны для тех, кто ранее программировал без использования принципов ООП.

Рассмотрим, например, как выглядят возможности наследования с первого взгляда. Наследование дает возможность построить простой класс, потом на его базе более сложный (добавив поля и методы), потом еще более сложный и т.д. Так это выглядит на первый взгляд. Но это, можно сказать, примитивный взгляд на возможности механизма наследования.

В ООП наследование обычно используется иначе и для других целей. Наследование классов - это мощный аппарат построения абстракций. При создании иерархии наследования классов в вершине иерархии расположены наиболее абстрактные сущности, которые все более конкретизируются и специализируются при движении вниз по дереву классов.

Для более предметного обсуждения рассмотрим несколько типичных примеров построения дерева наследования в стиле ООП. Примеры



Отступление.

В визуальном представлении дерева наследования его изображают сверху вниз - от базовых классов к производным. На основе такого представления формируется и терминология (например, upcasting и downcasting, см. ниже).

Тип "Птицы" включает в себя свойства, характерные для всех птиц. А порожденные от него типы "Вороны" и "Попугаи" добавляют (каждый) какие-то свойства, характерные только для данных видов птиц.

Тип "Печатное издание" хранит наиболее общую информацию, характерную для всех печатных изданий. Например, каждое печатное издание имеет наименование. В типах, порожденных от него, добавляются свойства, характерные только для книг (например - список авторов), журналов (год, номер) или газет (дата).

Приведенные примеры демонстрируют не усложнение, а детализацию и конкретизацию классов при наследовании. Именно это является основным общепринятым принципом при построении дерева наследования классов: наверху - наиболее общие, абстрактные понятия, и при движении вниз по дереву иерархии они детализируются.

Можно поставить такую абстрактную задачу.

Пусть требуется формировать множества, которые могут включать в себя как элементарные объекты, так и множества того же типа. Набор типов элементарных объектов фиксирован.

Правильное решение состоит в том, что в вершину иерархии мы помещаем класс для самого этого множества, а классы элементарных объектов мы строим как наследники класса множества. Детализация заключается в том, что эти классы - это множества из одного элемента.

В дальнейшем изучении мы постараемся помнить о том, что наследование - это механизм построения на основе более общих, более абстрактных классов новых классов - более специфических, более конкретных.

Продолжим знакомство с возможностями аппарата наследования классов.

Преобразования типов (классов) при наследовании

Аппарат наследования классов предусматривает возможности преобразования типов между суперклассом и подклассом. Преобразование типов в каком-то смысле является формальным. Сам объект при таком преобразовании не изменяется, *преобразование относится только к типу ссылки на объект*.

Рассмотрим это на примере.

Пример

```
class A
{
int x;
   . . .
}
class B extends A {
int y;
   . . .
}

B b = new B();
A a = b;   // здесь происходит формальное преобразование типа: B =>
A
```

Различаются два вида преобразований типов - *upcasting* и *downcasting*. Повышающее преобразование (upcasting) - это преобразование от типа порожденного класса (от подкласса) к базовому (суперклассу). Такое преобразование допустимо всегда. На него нет никаких ограничений и для его проведения не требуется применять никаких дополнительных синтаксических конструкций (см. предыдущий пример). Это связано с тем, что объект подкласса всегда в себе содержит как свою часть объект суперкласса. Понижающее преобразование (downcasting) - это преобразование от суперкласса к подклассу. Такое преобразование имеет ряд ограничений. Во-первых, оно может задаваться только явно при помощи операции преобразования типов, например, `B b1 = (B)a;`

Во-вторых, объект, подвергаемый преобразованию, реально должен быть того класса, к которому он преобразуется. Если это не так, то возникает исключение ClassCastException в процессе выполнения программы.

Все это выглядит странно, для тех, кто не знаком с ООП. Действительно, получается, что единственная допустимая ситуация, когда такое

преобразование возможно, это когда мы построили объект класса B, где B вляется подклассом A, затем зачем-то преобразовали его к классу A, а потом, опять же непонятно для чего, преобразовали его обратно к классу B.

На самом деле это имеет смысл. Для класса A может быть создан программный код, выполняющий что-то полезное. Он имеет свои методы, и они предполагают работу с объектами класса A. Потом те, кто желают воспользоваться этим программным кодом, строят подкласс B класса A. Но при работе с программным кодом, разработанным для класса A, приходится объекты класса B преобразовывать к классу A (upcasting), поскольку программный код для класса A ничего не знает о классе B (и ему подобных).

Получив какие-то результаты от программного кода класса A, нужно опять вернуться к работе с классом B (downcasting). Это один из типичных сценариев, требующих преобразования типов как в одну, так и в другую сторону.

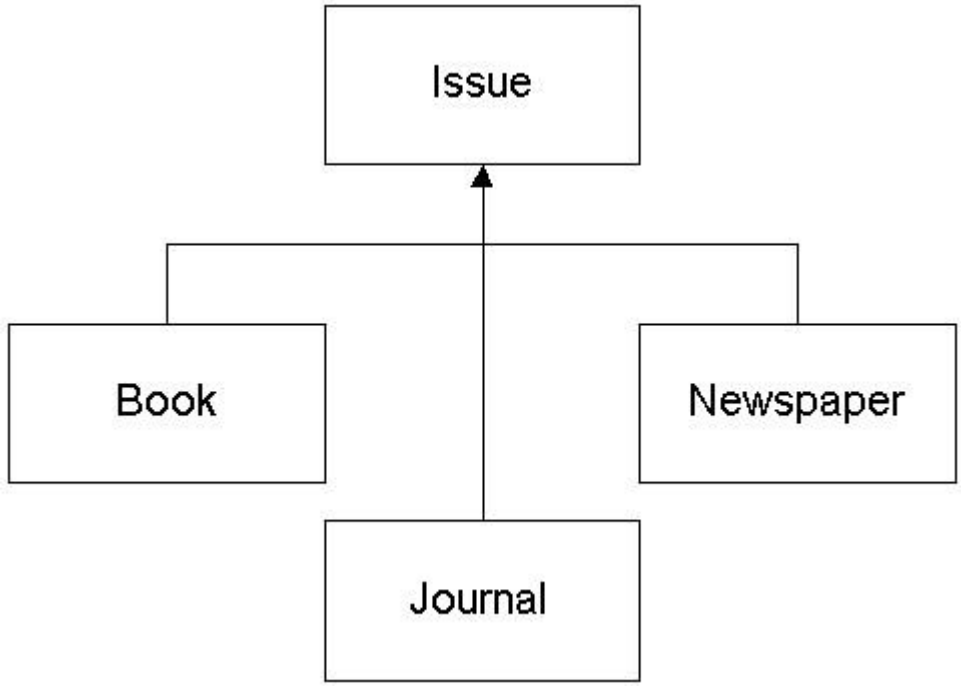
В Java для проверки типа объекта есть операция **instanceof**. Она часто применяется при понижающем (downcasting) преобразовании. Эта операция проверяет, имеет ли ее левый операнд класс, заданный правым операндом.

```
if ( a instanceof
B )     b1 =
(B)a;
```

Рассмотрим несколько более содержательный пример, в котором применяются оба вида преобразований, а также операция instanceof.

Пример

Рассмотрим иерархию классов (Issue - печатное издание, Book - книга, Newspaper - газета, Journal - журнал).



Пусть классы Issue и Book реализованы след. образом:

```
public class Issue {
    String name;
    public Issue(String name) {
        this.name = name;
    }
    public void printName(PrintStream out) {
         out.println("Наименование:");
         out.println(name);
    }
    . . .
}

public class Book extends Issue {
    String authors;
    public Book(String name, String authors) {
        super(name);
        this.authors = authors;
    }

    public void printAuthors(PrintStream out) {
         out.println("Авторы:");
         out.println(authors);
    }
    . . .
}
```

где-то в программе присутствует такой фрагмент

```
Issue[] catalog = new Issue[] {
     new Journal("Play Boy"),
     new Newspaper("Спид Инфо"),
     new Book("Война и мир", "Л.Толстой"), };
. . .


for(int i = 0; i < catalog.length; i++) {
if ( catalog[i] instanceof Book )
          ((Book) catalog[i]).printAuthors(System.out);
catalog[i].printName(System.out);
```

```
}
```

Рассмотрим его более детально. Здесь порождается каталог (массив печатных изданий), причем каждое из печатных изданий каталога может быть как книгой, так и газетой или журналом. При построении массива выполняется приведение к базовому типу (upcasting). Далее в цикле мы печатаем информацию из каталога. Причем, для книг кроме наименования печатается еще и список авторов. Для этого с использованием операции instanceof проверяется тип печатного издания, а при самой печати списка авторов элемент каталога преобразуется к типу Book. Если этого не сделать, транслятор выдаст ошибку, т.к. метод printAuthors(...) есть только в классе Book, но не в классе Issue.

Полиморфизм

В ООП применяется понятие *полиморфизм*.

Полиморфизм в ООП означает возможность применения одноименных методов с одинаковыми или различными наборами параметров в одном классе или в группе классов, связанных отношением наследования. Понятие полиморфизма, в свою очередь, опирается на два других понятия: совместное использование (*overloading*) и переопределение (*overriding*).
Рассмотрим их подробнее.

Термин *overloading* можно перевести как перегрузку, доопределение, совместное использование. Мы будем использовать перевод *совместное использование*. Под совместным использованием понимают использование одноименных методов с различным набором параметров. При вызове метода в зависимости от набора параметров выбирается требуемый метод. При этом одноименные методы могут быть как в составе одного класса, так и в разных классах, связанных отношением наследования. Это *статический* полиморфизм методов классов. Примеры совместного использования мы уже встречали ранее. Приведем еще несколько примеров.

```
class X {
int f() {
      . . .
   }     void f(int
k) {
      . . .
}
... }
```

В классе X есть два метода f(...), но с разными типами возвращаемого значения и разными наборами параметров. Тип возвращаемого значения не является определяющим фактором при совместном использовании - при вызове метода транслятору нужно определить, какой из одноименных методов вызывать, а тип возвращаемого значения, в общем случае, не позволяет сделать это однозначно. Поэтому нельзя описать в рамках одного класса два метода с одинаковым набором параметров и разными типами возвращаемых значений.

```
class Base {
     int f(int k) {
      . . .
    }
. . .
}  class Derived extends Base
{     int f(String s, int k) {
      . . .
    }     .
. .
}
```

В данном примере представлено совместное использование при наследовании. Класс Derived имеет два метода f(...). Один он наследует от класса Base, другой описан в самом классе Derived.

Понятие *overloading* нужно отличать от понятия *overriding* (задавливание, подавление, переопределение). При переопределении (overriding) методов речь идет только о паре классов - базовом и порожденном. В порожденном классе определяется метод полностью идентичный как по имени, так и по набору параметров тому, что есть в базовом.

Пример

```
class A {    int
x;    int f(int a)
{       return
a+x;
   }    .
. .
}
class B extends A {
int y;     int f(int
s) {        return
s*x;
    }
. . .
}

B b = new B();
A a = b;          // здесь происходит формальное преобразование типа: B
=> A int c = a.f(10); // ??? какой из f(...) будет вызван
???
```

Здесь самым интересным моментом является последняя строка. В ней "a" формально имеет тип A, но фактически ссылается на объект класса B. Возникает вопрос, какой из двух совершенно одинаково описанных методов f() будет вызван. Ответ на этот вопрос - B.f().

В Java (как и в других объектно-ориентрованных языках) выполняется вызов метода данного объекта с учетом того, что объект может быть не того же класса, что и ссылка, указывающая на него. Т.е. выполняется вызов метода того класса, к которому реально относится объект. Это - *динамический* полиморфизм методов. Он называется *поздним связыванием* (dynamic binding, late binding, run-time binding). В C++ соответствующий механизм называется механизмом виртуальных функций.

Рассмотрим содержательный пример использования возможностей, которые дает переопределение методов и позднее связывание. Реализуем классы Issue и Book иначе.

```
public class Issue {
String name;
    public Issue(String name) {
this.name = name;
    }
    public void print(PrintStream out) {
out.println("Наименование:");          out.println(name);
    }
. . .
}  public class Book extends Issue
{
    String authors;
    public Book(String name, String authors) {
super(name);
        this.authors = authors;
    }
    public void print(PrintStream out) {
out.println("Авторы:");
out.println(authors);
          super.print(out);       // явный вызов метода базового класса
}     . . .
}
```

и переделаем фрагмент, обеспечивающий печать нашего каталога. `Issue[] catalog = new Issue[] {  new Journal("Play Boy"),  new Newspaper("Спид Инфо"),  new Book("Война и мир", "Л.Толстой"), };`

```
. . . for(int i = 0; i < catalog.length;
i++) {
catalog[i].print(System.out);
}
```

В классах Issue и Book вместо двух методов printName(...) и printAuthors(...) теперь один метод print(..). В классе Book метод print(...) переопределяет одноименный метод класса Issue.

- При написании метода print(...) в Book для сокращения кода использован прием явного вызова метода базового класса с использованием ключевого слова **super**. Эту возможность мы рассматривали ранее.

Теперь при печати каталога мы можем не делать специальную проверку для Book. Нужный метод print(...) класса Book будет вызван автоматически благодаря механизму позднего связывания.

Ключевое слово final (Отступление)

В Java есть ключевое слово **final**, используемое как описатель полей, переменных, параметров и методов.

В применении к полям, переменным и параметрам оно означает, что их значение не может быть изменено. Поле или переменная с описателем final должны получить значение при описании, параметр просто не может быть изменен внутри тела метода.

```
final double pi = 3.14;
```

Описатель final в сочетании с описателем static позволяют создать константы, т.е.

поля, неизменные во всей программе. Так pi логичнее было бы описать так.

```
static final double pi = 3.14;
```

Если нужно запретить переопределение (overriding) метода во всех порожденных классах, то этот метод можно описать как final.

Кроме того, ключевое слово final может применяться к классам. Это означает, что данный класс не может быть унаследован другим классом.

Абстрактные классы

Класс может представлять собой как бы заготовку, в которой часть методов реализована, а часть - нет. В этом случае в описании класса перед словом class должен стоять описатель **abstact** и при описании нереализованных методов тоже должен использоваться этот описатель. Пример

```
public abstract class D {
. . .
   int g1(int s) {
       . . .
    }
    public abstract g2(String str);

    . . .
}
```

В классе D метод g1 - это обычный метод, g2 - абстрактный, он содержит только заголовок, но не содержит реализации.

Как видно из примера, тело абстрактного метода отсутствует, сразу после заголовка метода стоит точка с запятой.

Абстрактный класс не может использоваться непосредственно для порождения объектов. Для этого необходимо, используя этот класс как базовый, породить другой класс, в котором нужно определить все абстрактные методы. Тогда можно будет создавать объекты.

С другой стороны не запрещено описывать переменные абстрактного класса. Просто им нужно присваивать ссылки на объекты неабстрактных классов.

- В этой ситуации всегда применяется upcasting.

Возвращаясь к примеру с печатными изданиями, можно отметить, что класс Issue можно было бы реализовать как абстрактный, определив но не реализовав в нем метод print(...). Тогда во всех порожденных классах (Book, Journal, Newspaper) пришлось бы реализовать метод print(...). Фрагмент, печатающий каталог, при этом остался бы прежним.

Интерфейсы

Понятие интерфейса чем-то похоже на абстрактный класс. Интерфейс - это полностью абстрактный класс, не содержащий никаких полей, кроме констант (static final - поля).

- Терминология. Класс *наследует* другой класс, но, класс *удовлетворяет* интерфейсу, класс *реализует, выполняет* интерфейс.

Существует, однако, серьезное отличие интерфейсов от классов вообще и от абстрактных классов, в частности. Интерфейсы допускают множественное наследование. Т.е. один класс может удовлетворять нескольким интерфейсам сразу.

- Это связано с тем, что интерфейсы не порождают проблем с множественным наследованием, поскольку они не содержат полей.

Синтаксис:

```
public interface XXX {
    . . .
    int f(String s);
}
```

Это описание интерфейса XXX. Внутри скобок могут находиться только описания методов (без реализации) и описания констант (static final - полей). В данном случае интерфейс XXX содержит, в частности метод f(...).

```
public class R implements Serializable, XXX {
     . . .
}
```

Класс R реализует интерфейсы Serializable и XXX.

Внутри класса, реализующего некоторый интерфейс, должны быть реализованы все методы, описанные в этом интерфейсе. Поскольку XXX имеет метод f(...), то в классе R он должен быть реализован:

```
public class R implements Serializable, XXX {
     . . .
   public int f(String s) {
      ...
 }
 ...
 }
```

Обратите внимание, что в интерфейсе f(...) описан без описателя public, а в классе R с описателем public. Дело в том, что все методы интерфейса по умолчанию считаются public, так что этот описатель там можно опустить. А в классе R мы обязаны его использовать явно.

Еще одним общим моментом интерфейсов и абстрактных классов является то, что хотя и нельзя создавать объекты интерфейсов, но можно описывать переменные типа интерфейсов.

- Интерфейсы широко используются при написании различных стандартов. Стандарт определяет, в частности, набор каких-то интерфейсов. И, кроме того, он содержит вербальное описание семантики этих интерфейсов. После этого, прикладные разработчики могут писать программы, используя интерфейсы стандарта. А фирмы-разработчики могут разрабатывать конкретные реализации этих стандартов. При внедрении (deployment) прикладного программного обеспечения можно взять продукт любой фирмы-разработчика, реализующий данный стандарт (на практике, конечно, все несколько сложнее).

В последующем изучении мы редко будем строить абстрактные классы и интерфейсы, но очень часто будем использовать таковые из стандартной библиотеки Java.