**Лабораторная работа №4**

**Основы программирования на Java: классы, объекты, пакеты**

**Цель работы:** Освоить основные способы создания Java-программ либо с помощью обычного редактора, либо с помощью среды разработки. Используя примеры программ познакомиться с основными приемами в программировании на Java. Приобрести навыки в использовании системы помощи для поиска нужной информации по различным классам Java.

**Теоретические сведения**



Классы лежат в фундаменте объектно-ориентированных свойств языка Java.

***Статические и динамические элементы (модификатор static)***

В Java переменные и методы класса могут быть либо **элементами класса**, либо **элементами экземпляра (объекта)** класса, в котором они объявлены, что определяется присутствием или отсутствием модификатора **static.**

Если при определении элемента не используется ключевое слово **static**, то этот элемент по умолчанию является **динамическим** (dynamic). Динамические методы и переменные всегда являются **элементами объекта класса**, и доступ к ним осуществляется через переменную-объект.

**Статические** методы и переменные связаны с классом, а не с экземпляром класса, и поэтому имеют название **элементов класса**. Элементы класса можно использовать без создания объекта этого класса, доступ осуществляется через имя класса. Каждая *переменная класса* и каждый *метод класса* уникальны *в своем классе*; *методы* и *переменные экземпляра* уникальны в своем экземпляре класса. Различие между членами класса и экземпляра значительно, в особенности, если дело касается переменных

*Элементы класса,*будучи уникальными в своем классе, используются всеми объектами, созданными из этого класса, - то есть все объекты, созданные из данного класса, разделяют статические переменные и методы, определенные в этом классе (для всех объектов существует только один экземпляр статической переменной). *Элементы экземпляра класса*, с другой стороны, создаются каждый раз, когда создается объект.

*Элементы класса* могут, таким образом, считаться глобальными относительно класса, несмотря на то, что настоящие глобальные переменные в Java не поддерживаются. Когда какой-нибудь объект класса изменяет значение *переменной класса*, результат становится видим всем объектам. Благодаря этому переменные класса часто используют в качестве общих данных для всех объектов, созданных из этого класса.

Чтобы обратиться к *методам и переменным экземпляра (динамическим элементам)*, объект надо сначала реализовать из класса, после чего можно получить к ним доступ, используя синтаксис (запись с точкой):

***ИмяОбъекта.ИмяМетодаЭкземпляра(<Параметры>)***

***ИмяОбъекта.ИмяПеременнойЭкземпляра***

А для использования статических методов и переменных (элементов класса) объект можно не создавать, а пользоваться записью следующего вида:

***ИмяКласса.ИмяМетодаКласса(<Параметры>)***

***ИмяКласса.ИмяПеременнойКласса***

*Замечание.* Нужно отметить, что *статические методы* класса могут работать только со *статическими переменными* класса, для которых также не важно наличие реализованного объекта.

Рассмотрим пример, в котором сначала определяется класс, включающий статические и динамические методы, а затем его методы используются в функции main():

/\*------------- Пример 5. Файл TestElements.java -------------\*/

class StaticAndDynamic

{ int i=0; // переменная экземпляра

static int j=0; // переменная класса

// статические методы

static void setJ(int k)

{ System.out.println("Static Method"); j=k;

}

static int getJ()

{ System.out.println("Static Method"); return j;

}

// динамические методы

void setI(int k)

{ System.out.println("Dynamic Method"); i=k;

}

int getI()

{ System.out.println("Dynamic Method"); return i;

}

int summa()

{ // в динамических методах можно

// использовать статические переменные

System.out.println("Dynamic Method"); return i+j;

}

}

class TestElements

{ public static void main(String args[])

{ int ii,jj;

// для использования элементов класса объект необязателен

// использование статической переменной

StaticAndDynamic.j=6; jj=StaticAndDynamic.j;

System.out.println("Main, jj="+jj);

// вызов статических методов

StaticAndDynamic.setJ(4); jj=StaticAndDynamic.getJ();

System.out.println("Main, jj="+jj);

// перед использованием элементов экземпляра

// требуется реализовать объект (экземпляр)

StaticAndDynamic obj=new StaticAndDynamic();

// использование динамической переменной

obj.i=3; ii=obj.i;

System.out.println("Main, ii="+ii);

// вызов динамическим методов

obj.setI(8); ii=obj.getI();

System.out.println("Main, ii="+ii);

// вызов метода, в котором используются

// динамическая и статическая переменные

ii=obj.summa();

System.out.println("Main, summa="+ii);

}

} /\*------------------------------------------\*/

*Замечание.* Теперь понятно, почему при объявлении метода main всегда используется ключевое слово static. Дело в том, что при загрузке первичного класса в память он загружается в виде *типа*, а не в виде *объекта*. После этого оболочка времени выполнения просто вызывает *статический метод* main() этого первичного класса.

Модификаторы доступа

**Модификаторы доступа** используются для управления доступностью элементов класса из других частей программы (в других классах). Применение ключевых слов **public** (открытый), **protected** (защищенный), **private** (закрытый) к элементам помогает управлять способностью других объектов пользоваться ими.

Элемент, объявленный с ключевым словом **public**, доступен во всех классах, как в том пакете (о пакетах классов см. ниже), в котором он был объявлен, так и во всех классах в любом другом пакете. Из всех модификаторов данный накладывает наименьшие ограничения на доступность элемента - он доступен всем, является открытым для всех. Этот модификатор, в отличие от всех остальных, можно использовать и *при объявлении класса* (класс может иметь этот модификатор или не иметь). Тогда этот класс также доступен для всех других классов невзирая на то, частью какого пакета классов он является. В каждом файле должен содержаться только один открытый класс. При этом имя файла должно совпадать с именем такого класса.

Элемент, объявленный с модификатором **protected** в некоем классе A, доступен во всех классах , являющихся подклассами класса A.

Модификатор **private** сильнее всего ограничивает доступность элемента. Он его делает невидимым нигде за пределами данного класса. Даже подклассы данного класса не смогут обращаться к элементу, объявленному как private.

Если тип доступа к элементу не указан ( такой доступ является доступом по умолчанию), то он доступен из всех классов, входящих в данный пакет.

Модификатор доступа (private, protected ) используется для так называемого скрытия данных (data hiding). Объявление переменных или методов класса с закрывающим их модификатором делает невозможным их использование вне области доступности. Методы же их собственного класса имеют к ним доступ всегда. Поэтому, например, для доступа к закрытым переменным для установки из значений или получения значений, содержащихся в них, можно использовать специально созданные для этого незакрытые методы класса (если они конечно, существуют).

Приведем пример, в котором определяется класс, в котором есть открытая и закрытая переменные, и показываются методы доступа к этим переменным вне их класса.

/\*------------- Пример 6. Файл TestModificators.java -------------\*/

class A

{ public int k; // тип public - - доступ и в теле класса и по объекту класса

private int n; // тип private - доступ только в теле класса

A() { k=2; n=11; } // конструктор, инициализация переменных-элементов

int summa() { return k+n; } // метод класса, использующий обе переменные

// методы для доступа к защищенному элементу n класса A

// по объекту класса A вне тела класса A

public int getN() { return n; }

public void setN(int nn) { n=nn; }

}

class TestModificators

{ public static void main(String args[])

{ A obj=new A(); // создание объекта класса A

// получить значение переменных

int kk=obj.k; System.out.println("k="+kk);

int nn=obj.getN(); System.out.println("n="+nn);

// установить значения переменных и

// вызвать метод, использующий

//закрытую и открытую переменные

obj.k=10; obj.setN(15);

int s=obj.summa(); System.out.println("summa="+s);

}

}

Наследование классов

Модификаторы доступа делают классы более устойчивыми и надежными в работе, так как гарантируют, что снаружи класса можно будет получить доступ только к некоторым из методов и переменных. **Наследование** (inheritance), в свою очередь, упрощает практическое использование классов, так как позволяет *расширять* уже написанные и отлаженные классы, *добавляя* к ним новые свойства и возможности.

Например, вместо создания класса “с нуля”, можно значительно выиграть во времени, используя механизм наследования переменных и методов, определенных в другом классе, обладающем основными необходимыми свойствами. Таким образом создается то, что называется **подклассом** первоначального класса. Класс, который при этом наследуется (расширяется), называется **суперклассом.** При расширении какого-либо класса имеется возможность использования всех написанных и отлаженных методов этого класса - их не нужно создавать заново для подкласса. Это свойство, называемое **повторным использованием кода** (code reuse), является одним из главных преимуществ объектно-ориентированного програмирования.

Для того, чтобы наследовать свойства существующего класса, класс должен быть явно определен с ключевым словом **extends** (подкласс расширяет суперкласс). Например, объявления класса MyClass подклассом класса SuperClass, можно записать так:

class MyClass **extends** SuperClass

{ ..............

}

Если класс явно не объявляется подклассом какого-либо класса, компилятор предполагает по умолчанию его подклассом класса Object.

Специальные переменные

Каждый класс Java содержит три заранее определенные переменные, которыми можно пользоваться: **null, this, super**. Первые две переменные относятся к типу **Object**. Коротко говоря, **null** представляет собой несуществующий объект**, this** указывает на текущий экземпляр, переменная **super** разрешает доступ к открытым переменным и методам, определенным в суперклассе.

**Переменная null**

Прежде чем использовать какой-то класс, его нужно реализовать, т.е. создать объект (экземпляр) этого класса. До этого ссылка на объект имеет значение **null,** и говорят, что **объект равен нулю**. Если объект равен нулю, доступ к его элементам не разрешен, потому что не создан объект, с которым могли бы ассоциироваться эти элементы. Если попытаться обратиться к элементам до того, как создан объект, то вызовется исключение, которое остановит выполнение программы. Например, определим метод класса ClassB, использующий передаваемый ему объект класса ClassA:

piblic void SomeMethodСlassB(СlassA ObjClassA)

{

ObjClassA.SomeMethodClassA();

}

Следующий фрагмент метода класса ClassB, в котором вызывается метод SomeMethodСlassB, приведет к исключению NullPointerException, потому что не создан объект класса ClassA:

ClassA ObjectClassA;

SomeMethodClassB(ObjectClassA);

Чтобы уберечь программу от сбоев, необходимо перед использованием объектов проверять их на равенство null . Изменим метод SomeMethodClassB:

piblic void SomeMethodСlassB(СlassA ObjClassA)

{

if(ObjClassA==null) { System.out.println("A пусто!!!"); }

else { ObjClassA.SomeMethodClassA(); }

}

**Переменная this**

Иногда бывает необходимо передать другому методу **ссылку на текущий объект**. Это можно сделать, просто передав переменную **особую ссылочную переменную this**.

Кроме того, переменная this используется **для ссылок на переменную экземпляра**. Например, если на уровне класса объявлена переменная-элемент с именем *ИмяПеременной*, то ничто не мешает объявить локальную переменную с тем же именем *ИмяПеременной* внутри какого-либо из методов класса, которая скроет вышестоящую переменную и сделает ее недоступной (**скрытие переменной**). Внутри метода идентификатор *ИмяПеременной* будет относиться именно к локальной переменной, объявленной в этом методе. Однако существует возможность получить доступ к переменной-элементу, даже если она скрыта. Для этого нужно воспользоватьсяпеременной this. Переменная this всегда указывает на текущий класс, поэтому, чтобы в методе получить доступ к скрытой переменной, объявленной в текущем классе, нужно явным образом указать принадлежность переменной к классу, а не к методу:

***this..ИмяПеременной***

Еще одна возможность использования этой ссылочной переменной - **вызов в конструкторе класса другого конструктора этого же класса**:

***this(<ПараметрыДругогоКонструктораКласса>);***

**Переменная super**

Другая специальная переменная языка Java **super** ссылается **на суперкласс** объекта. При помощи нее можно получить **доступ к затененным переменным и замещенным методам родительского класса. Затенение переменной** класса в его подклассе возникает при объявлении в подклассе переменной таким же именем, что и имя переменной его суперкласса. Например, в классе объявлена переменная с именем *ИмяПеременной* и в его подклассе также объявлена переменная с тем же именем *ИмяПеременной.* Для осуществления в подклассе доступа к такой переменной суперкласса необходимо использовать переменную super:

***super.ИмяПеременной***

При **замещении метода** в классе создается метод, имеющий то же самое имя и список параметров, что и метод в суперклассе. Таким образом метод суперкласса замещается методом подкласса. Однако можно использовать переменную super выполнения замещенного метода суперкласса:

***super.ИмяМетодаСуперкласса*(*ПараметрыМетодаСуперкласса*);**

Используя переменную super, **в конструкторе подкласса можно вызвать конструктор родительского класса**. Нужно заметить, что *при создании объектов подкласса сначала автоматически вызывается конструктор суперкласса без параметров* (если он не определен, то вызывается задаваемый по умолчанию конструктор без параметров), *а затем только конструктор подкласса*. Но если в конструкторе подкласса есть явный вызов конструктора суперкласса, то автоматического вызова конструктора суперкласса без параметров не происходит. Сначала вызывается требуемый явным вызовом конструктор суперкласса, а затем только конструктор подкласса.

Если возникает необходимость вызвать конструктор суперкласса из конструктора подкласса, то это можно сделать, используя ключевое слово super. Для того, чтобы вызвать из конструктора подкласса конструктор суперкласса, нужно использовать следующую запись:

***super(<ПараметрыКонструктораСуперкласса>).***

В том конструкторе подкласса, в котором используется подобный вызов конструктора суперкласса, этот оператор должен быть первым исполняемым оператором.

*Замечание*. Не следует считать, что переменная super указывает на совершенно отдельный объект. Чтобы ее использовать, не нужно реализовывать суперкласс. На самом деле это просто способ обращения к переменным -элементам суперкласса и способ выполнения методов и конструкторов, определенных в суперкласса

**Пакеты и импортирование**

Классы являются основными строительными блоками любой Java-программы. В сравнении с классами **пакеты** выполняют чисто утилитарную функцию. Они просто содержат в себе наборы классов (а также исключения и интерфейсы). Кроме того, пакеты позволяют определять **защищенные** (protected) элементы классов, которые доступны всем классам, входящим в один и тот же пакет, но недоступны каким бы то ни было классам за пределами такого пакетов.

Использование пакетов

Основное назначение пакетов - создание библиотек кода, пакеты используются для организации и категоризации классов. Стандартные пакеты Java состоят из восьми пакетов API и пакета, содержащего классы и интерфейсы для отладки.

Импортирование пакетов

Существует ряд способов доступа к классам в пакетах, основным из которых является импортирование пакета в начале программ:

***import ИмяПакета .\****

или

***import ИмяПакета.ИмяКлассаПакета***

В этом случае импортирование просто должно предшествовать всем другим строка программы. Например:

import java.applet.\*

import java.awt.\*

import java.net.URL

При этом импортируются три пакета (если точнее, то открытые - public - классы пакетов) с использованием двух способов. В первых двух строках программы использовалась (\*), чтобы указать на возможность доступа ко всем открытым классам в пакете (тут нужно отметить, что классы подпакетов импортируются отдельно, с указанием отдельного оператора import). В последней строке явно установлено имя открытого класса для импортирования.

Надо заметить, что несмотря на то, что в первых двух строках указывается импортировать *все* классы пакета, на самом деле импортируются только классы, которые *используются* в программе; язык Java компонует классы, только если они действительно используются.

После того, как пакет импортирован, классы можно использовать просто по именам, без указания на принадлежность тому или иному пакету, например:

***ИмяКлассаПакета* Obj=new *ИмяКлассаПакета*(<*Параметры>*);**

При использовании класса из пакета необходимо или импортировать пакет, частью которого он является, или сослаться на него в программе, как это описано ниже. Однако **нет необходимости беспокоиться об импортировании пакета java.lang**, так как эти классы - базовые классы Java и автоматически импортируются независимо от того, заданы они или нет.

Явные ссылки на классы пакетов

Если пакет импортирован, то его классы могут использоваться столько раз, сколько требует программа. В случае, когда класс необходимо использовать только один раз, можно явно сослаться на класс, не импортируя пакет, частью которого он является. Для этого перед именем используемого класса просто указывается имя пакета, в котором находится этот класс, например:

***ИмяПакета.ИмяКласса* Obj=new *ИмяПакета.ИмяКласса*(*Параметры*);**

Способ явной ссылки на класс из пакета более удобен для чтения и понимания, откуда появился используемый класс, чем метод предложения импорта пакета, но менее удобен для кодирования, так как ссылка получается достаточно длинной.

***Конфликты именования***

Одним из достоинств явной ссылки на классы пакетов, помимо ясности самого текста программы, является то, что при этом имена классов не конфликтуют (что возможно при использовании предложения импорта).

Если программа импортирует два пакета и в каждом объявлены классы с одинаковыми именами, то непонятно, к примеру, по классу из какого пакета будет создаваться экземпляр класса. Это определить невозможно, поэтому компилятор выдаст ошибку и прервет компиляцию. Однако можно предотвратить этот потенциальный конфликт с помощью явного объявления имени пакета вместе с именем классов.

Уровни пакетов

Имена пакетов упорядочены по уровням, каждый уровень отделен от следующего. Например, пакет java.applet включает два уровня. Первый уровень - java, второй - applet. Некоторые стандартные пакеты API имеют и три уровня.

При создании пакетов разработчик должен сам принимать решение относительно имен и числа уровней пакетов.

Создание пакетов

Для создания пакета (т.е. добавления класса в пакет), просто используется следующее предложение в качестве первой строки программы (даже раньше предложения import) в исходном тексте программы:

***package ИмяПакета***

Содержимое пакета может хранится в одном или нескольких файлах. Каждый такой файл должен содержать только один общедоступный (public) класс. При компиляции этих файлов получающиеся в результате файлы с расширением .class будут помещены в каталоге, соответствующем имени пакета, все точки в котором будут заменены на символ /.

***Уникальные имена.***

Соглашение об именовании, используемое при создании пакетов, разработано таким образом, чтобы обеспечить уникальность имен пакетов при единообразии доступа к ним и не привести к конфликту имен. Все пакеты, которые создаются программистами и которые будут использоваться вне их организации, должны использовать имена Internet-доменов организации, записанные в обратном порядке, чтобы гарантировать уникальность имен.

Например, для того чтобы пакет хранился на сервере (домене) ermak.cs.nstu.ru и был широко доступен, при создании его надо снабдить именем домена, записанным в обратном порядке, например:

RU.nstu.cs.ermak.mypackage; // этот пакет уникальный

Как видно, первая часть уникального имени должна состоять из прописных букв, в оставшейся части используются строчные буквы.

Например, определим в пакете mypackage класс MyClass, тогда ссылка на него записывается следующим образом:

RU.nstu.cs.ermak.mypackage.MyClass;

В соответствии с соглашением об уникальности именования пакетов можно сразу сказать, что источником пакета mypackage является домен ermak.cs.nstu.ru. А так как только имена классов начинаются с прописной буквы, то очевидно, что MyClass - это класс, а не очередной уровень пакета mypackage.

**Задание**

Проверить и объяснить работу всех приложений, рассматриваемых в данной работе. Должны быть созданы следующие приложения **TestElements** (пример 5)**, TestModificators** (пример 6).

**Форма отчета**

1. Тема, цель

2. Скриншоты выполненных программ

3. Ответы на контрольные вопросы

**Контрольные вопросы**

1. Что такое совмещение методов?
2. Что такое элементы класса и элементы экземпляра класса, чем они отличаются друг от друга? Как нужно указывать, что переменная или метод является элементом класса, а не экземпляра?
3. Для чего используются модификаторы доступа? Какие существуют модификаторы доступа, как они ограничивают доступ к элементам?
4. Что позволяет делать процесс наследования?
5. Что такое суперкласс и подкласс?
6. Что такое повторное использование кода?
7. Какие заранее определенные переменные содержит каждый класс Java?
8. Что можно сделать при помощи переменной this?   
   Что можно сделать при помощи переменной super?
9. Что такое скрытие переменной, затемнение переменной и замещение метода?
10. Как импортировать классы из пакетов?
11. Как использовать явные ссылки на классы из пакетов?
12. Как добавить класс в пакет?