**Лабораторная работа № 1**

**«Основы языка Java»**

**Цель работы**: изучить основы разработки *Java-*программ. Изучить основные типы *Java,* создание и работу с массивами.

**Задание на лабораторную работу**

В ходе выполнения данной лабораторной работы необходимо ознакомиться с основными типами *Java.* Создать консольное приложение, которое рассчитывает значение функции по рекуррентной формуле, согласно варианту задания.

**Краткие теоретические сведения**

Хотя все переменные в Java трактуются как объекты, идентификатор, который используется для манипулирования объектом, на самом деле является «ссылкой» на объект. Таким образом, если просто объявить ссылку, это не значит, что она уже связана с объектом. Например, ссылка на объект класса String:

String s;

Здесь создается только ссылка, а не объект. Если послать сообщение для s в этом месте, то возникнет ошибка (времени выполнения), потому что s ни к чему не присоединено. Поэтому необходимо всегда инициализировать ссылку при ее создании:

String s = "asdf";

Здесь использована особенность Java: строки могут быть инициализированы текстом в кавычках. Обычно необходимо использовать более общий тип инициализации для объектов.

***Создание объектов***

В общем случае объекты создаются с помощью ключевого слова **new**.

String s = new String("asdf");

Здесь кроме создания нового объекта класса String указывается, каким

значением инициализировать строку.

***Примитивные типы***

Есть группа типов, требующих особого обращения. Речь идет о так называемых «примитивных» типах, которые достаточно часто используются в программировании. Причина специального использования в том, что создание объектов с помощью **new** — особенно маленьких, простых переменных — не очень рационально, поскольку **new** помещает объекты в кучу. Для этих типов Java возвращается к подходу, принятому в C и C++. Так что вместо создания переменной с использованием **new**, «автоматические» переменные создаются не по ссылке. Переменная хранит значение, и оно помещается в стек, так как это более эффективно.

Java определяет размер каждого примитивного типа. Размеры не меняются при переходе от одной архитектуры машины к другой, как это сделано во многих языках. Этот размер инвариантен – это причина того, что программирование на Java свободно переносимо между разными платформами.

Таблица 1 - Примитивные типы в Java

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Примитивный тип | Размер, в битах | Минимум | Максимум | Класс  оболочки |
| boolean | — | — | — | Boolean |
| char | 16 | Unicode 0 | Unicode 216- 1 | Character |
| byte | 8 | -128 | +127 | Byte |
| short | 16 | -215 | +215 - 1 | Short |
| int | 32 | -231 | +231 - 1 | Integer |
| long | 64 | -263 | +263 - 1 | Long |
| float | 32 | IEEE754 | IEEE754 | Float |
| double | 64 | IEEE754 | IEEE754 | Double |
| Void | — | — | — | Void |

Размер **boolean** типов точно не определен; только указано, что они способны принимать литерные значения **true** или **false**.

Примитивные типы данных также имеют классы «оболочки» для них. Это означает, что если необходимо создать не примитивный объект в куче для представления примитивного типа, необходимо использовать ассоциированную

оболочку. Например:

char c = 'x';

Character C = new Character(c);

Или также можно использовать:

Character C = new Character('x');

Примитивный тип всегда можно привести к типу класса-оболочки и наоборот.

Character C = new Character('x');

char c = C.charValue();

Integer I = new Integer(3);

int i = I.intValue();

***Числа высокой точности***

Java включает два класса для работы с высокоточной арифметикой: **BigInteger** и **BigDecimal**. Хотя они приблизительно попадают в ту же категорию, что и классы «оболочки», ни один из них не имеет примитивного

аналога.

Оба класса имеют методы, обеспечивающие аналогичные операции, которые выполняются для примитивных типов. Так что с классами **BigInteger** или **BigDecimal** можно делать все, что можно делать с int или float, только необходимо использовать вызов методов вместо операторов. Следует заметить, что такие операции выполняются медленнее. Скорость меняется на точность.

**BigInteger** поддерживают целые числа произвольной точности. Это означает, что можно точно представить значение целого числа любого размера без потерь любой информации во время операций. **BigDecimal** для чисел с фиксированной точкой произвольной точности.

***Уничтожение объектов***

Большинство процедурных языков имеют концепцию границ. Они определяют и видимость, и время жизни имен, определенных в таких границах. В C, C++ и Java границы определяются расстановкой фигурных скобок {}. Так, например:

{

int x = 12;

/\* доступно только x \*/

{

int q = 96;

/\* доступны и x, и q \*/

}

/\* Доступно только x \*/

/\* q “за границами” \*/

}

Переменная, определенная внутри границ доступна только до конца этой

границы.

Выравнивание делает Java код легким для чтения. Так как Java – это язык свободной формы, дополнительные пробелы, табуляции и возврат каретки не

влияют на результат программы.

Обратите внимание, что нельзя сделать следующее, хотя это разрешено в

С и C ++:

{

int x = 12;

{

int x = 96; /\* недопустимо \*/

}

}

Компилятор объявит, что переменная **x** уже определена. C и C++ способны «прятать» переменные в больших границах, что не позволяется в Java, поскольку разработчики подумали, что это будет запутывать программы. Java объекты имеют не то же самое время жизни, что примитивы. Когда создается Java объект, используя **new**, он продолжает существовать после конца границы.

{

String s = new String("a string");

} /\* конец блока \*/

Ссылка **s** исчезает по окончании границы. Однако объект String, на который указывал **s**, продолжает занимать память. Вне этого участка кода нет способа получить доступ к объекту, поскольку ссылка на него существует только внутри границ.

Так как объекты создаются с помощью **new**, они остаются столько, сколько это необходимо программисту, что создавало в C++ проблемы при программировании, и что просто исчезло в Java. В C++ необходимо проверять, что объект уничтожен, когда с ним завершили работу.

Защита от переполнения памяти и остановки программы, что случается в

C++, в Java решается за счет использования *сборщика мусора*, который смотрит на все объекты, которые были созданы с помощью **new**, и решает, на какие из них больше нигде нет ссылок. Затем он освобождает память этого объекта, так что память может использоваться для новых объектов. Это означает, что нет необходимости самостоятельно заботиться об утилизации памяти. Когда объект больше не нужен, он сам исчезнет. Это подавляет определенных класс проблем программирования: так называемую “утечку памяти”, при которой программисты забывают освободить память.

***Видимость имен***

Проблема каждого языка программирования состоит в управлении именами. Если использовать имена в одном модуле программы, а другой программист использует эти же имена в другом модуле, возникает проблема как отличить одно имя от другого и предохранить два имени от «конфликта»? В C это обычная проблема, поэтому программы часто содержат неуправляемое море имен. Классы C++ (на которых основываются классы Java) содержат функции внутри классов, так что они не могут конфликтовать с именами функций, расположенных в других классах.

В Java для производства недвусмысленных имен для библиотеки используется спецификатор, мало чем отличающийся от доменных имен Internet. Фактически, создатели Java хотят использовать доменные имена Internet в обратном порядке, так как это гарантирует их уникальность. Например:

ua.cn.stu.oop.labs

Этот механизм означает, что все файлы автоматически живут в своем собственном пространстве имен, и каждый класс в файле должен иметь уникальный идентификатор.

***Использование других компонентов***

Когда требуется использовать предопределенные классы в программе, компилятор должен знать, где они расположены. Конечно, класс может существовать в том же самом файле исходного кода, из которого он вызывается. В этом случае просто используется класс — даже если класс определен дальше по файлу. Java устранит проблему «ранней ссылки», так что нет необходимости думать об этом.

Решение проблемы использования класса, определенного в другом файле, выполняется путем точного сообщения компилятору Java классов, которые не

обходимо использовать, с помощью ключевого слова **import**. **import** говорит

компилятору о введении пакета, который является библиотекой классов, например:

import java.util.ArrayList;

Пакет util содержит несколько классов, и можно использовать некоторые из них, не объявляя их точно. Это легче всего выполнить, используя ‘\*’. Это более общий способ для импорта набора классов, в отличие от индивидуального импорта каждого класса.

import java.util.\*;

***Ключевое слово static***

Обычно, когда создается класс, описывается, как выглядит объект класса и как он будет себя вести. На самом деле ничего не получается, пока не создается объект класса с помощью **new**, и в этом месте создается хранилище данных, и становятся доступными методы.

Но есть две ситуации, в которых этот подход не достаточен. Один из них, если требуется иметь только одну часть хранилища для определенных данных, не зависимо от того, сколько объектов создано, или даже если не было создано объектов этого класса. Второй, если нужен метод, который не ассоциируется с объектом определенного класса. То есть, нужен метод, который можно вызвать, даже если объект не создан. Можно достигнуть этих эффектов с помощью ключевого слова **static**. Когда говорится о чем-то **static**, это означает, что данные или метод не привязаны к определенному экземпляру объекта класса. Даже если никогда не создавать объект этого класса, можно вызвать статический метод или получить доступ к части статических данных. Так как статическим методам не нужно создавать объект до их использования, они не могут получить прямой доступ к не статическим членам или методам простым вызовом этих методов без указания имени объекта (так как не статические члены и методы должны быть привязаны к определенному объекту).

Некоторые объектно-ориентированные языки используют термины данные класса и методы класса в том смысле, что данные и методы существуют только для класса, как целое, а не для любого определенного объекта класса. Иногда литература по Java тоже использует эти термины.

Чтобы сделать член-данное или член-метод статическим, необходимо просто поместить ключевое слово перед определением. Например, следующий код производит статический член-данное и инициализирует его:

class StaticTest

{

static int i = 47;

}

Теперь, даже если создать два объекта StaticTest, будет только одна часть хранилища для StaticTest.i. Оба объекта будут разделять одну и ту же i. Например:

StaticTest st1 = new StaticTest();

StaticTest st2 = new StaticTest();

В этом месте и **st1.i**, и **st2.i** имеют одно и то же значение 47, так как они ссылаются на одну и ту же область памяти.

Есть два способа сослаться на статическую переменную. Как показано выше, можно назвать ее через объект, например, сказав **st2.i**. Также можете прямо сослаться через имя класса, что нельзя сделать с не статическими членами. Это предпочтительный способ сослаться на статическую переменную, так как это подчеркивает, что переменная имеет статическую природу.

StaticTest.i++;

Оператор **++** инкрементирует переменную. В этом месте и **st1.i**, и **st2.i** будут иметь значение 48.

Сходная логика применима и к статическим методам. Можно сослаться на статический метод либо через объект, как можно сделать с любым методом, или с помощью специального дополнительного синтаксиса ClassName.method( ). Статический метод определяется сходным образом:

class StaticFun

{

static void incr() { StaticTest.i++; }

}

Метод **incr()** класса StaticFun инкрементирует статическую переменную i. Можно вызвать **incr()** обычным способом, через объект:

StaticFun sf = new StaticFun();

sf.incr();

Или можно вызвать его прямо через класс:

StaticFun.incr();

Когда **static** применяется к членам-данным, это изменяет путь создания данных (один экземпляр для всего класса против не статического: один для каждого объекта). Важность использования static для методов в том, чтобы позволить вызывать этот метод без создания объекта.

Как и любой метод, статический метод может создавать или использовать именованные объекты того же типа.

***Массивы***

Фактически, все языки программирования поддерживают массивы. Использование массивов в C и C++ рискованно, поскольку эти массивы всего лишь блоки памяти. Если программа обращается к массиву вне пределов этого блока, или использует память до инициализации (общая ошибка программирования), получится непредсказуемый результат.

Когда создается массив объектов, на самом деле создается массив ссылок, а каждая из этих ссылок автоматически инициализируется специальным значением, имеющим собственное ключевое слово: **null**. Когда Java видит **null**, он распознает, что опрашиваемая ссылка не указывает на объект. Необходимо присвоить объект каждой ссылке, прежде чем использовать ее. Таким образом, типичные ошибки при работе с массивами предотвращены в Java.

Также можно создать массив примитивов. Компилятор гарантирует инициализацию, поскольку он заполняет нулями память для этого массива.

Массив – это просто последовательность либо объектов, либо примитивных типов, которые все имеют один тип и упакованы вместе под одним идентификатором. Массивы определяются и используются с квадратными скобками оператора индексирования **[]**. Для определения массива просто указывается имя типа, за которым следуют пустые квадратные скобки:

int[] a1;

Также можно поместить квадратные скобки после идентификатора, что

имеет то же самое значение:

int a1[];

Компилятор не позволяет объявить величину массива. Это происходит из-за свойств «ссылок». Получается только ссылка на массив, и здесь не резервируется место для массива. Для создания хранилища для массива необходимо написать выражение инициализации. Для массивов инициализация может быть выполнена в любом месте кода, но также можно использовать особый вид выражения инициализации, которая должна происходить в точке создания. Эта особая инициализация обеспечивает набор значений, заключенных в фигурные скобки. О резервировании хранилища (эквивалентно использованию **new**) в этом случае заботится компилятор. Например:

int[] a1 = { 1, 2, 3, 4, 5 };

В Java можно определять ссылку на массив без массива:

int[] a2;

И далее присвоить один массив в Java другому. На самом деле здесь выполняется копирование ссылок.

a2 = a1;

Все массивы имеют внутренний член (не зависимо от того, есть ли массив объектов, или массив примитивных типов), который можно опросить — но не изменить — и он скажет, сколько элементов есть в массиве. Этот член – **length**.

Массив может быть определен и инициализирован в одной инструкции:

int[] a = new int[20];

Если создается массив не примитивных объектов, необходимо всегда использовать **new**. Это происходит из-за использования ссылок, так как создается массив ссылок.

// Создание массива не примитивных объектов.

import java.util.\*;

public class ArrayClassObj

{

public static void main(String[] args)

{

Integer[] a = new Integer[20];

for(int i = 0; i < a.length; i++)

{

a[i] = new Integer(500);

}

}

}

Также возможно инициализировать массив, используя список, окруженный фигурными скобками. Существует две формы:

// Инициализация массива.

public class ArrayInit

{

public static void main(String[] args)

{

Integer[] a = {

new Integer(1),

new Integer(2),

new Integer(3),

};

Integer[] b = new Integer[] {

new Integer(1),

new Integer(2),

new Integer(3),

};

}

}

Java позволяет легко создавать многомерные массивы.

Первый пример показывает многомерный массив примитивных типов. Каждый вектор в массиве ограничивается с помощью фигурных скобок:

int[][] a1 = {

{ 1, 2, 3, },

{ 4, 5, 6, },

};

В следующем примере создается трехмерный массив, резервируемый с

помощью new. Здесь весь массив резервируется сразу:

int[][][] a2 = new int[2][2][4];

***Обработка ошибок с помощью исключений***

Исключительное состояние – это проблема, которая мешает последовательному исполнению метода или ограниченного участка. Важно различать исключительные состояния и обычные проблемы, в которых имеется достаточно информации в текущем контексте, чтобы как-то справиться с трудностью. В исключительном состоянии невозможно продолжать обработку, потому что нет необходимой информации, чтобы разобраться с проблемой в текущем контексте. Все, что можно сделать – это выйти из текущего контекста и отослать эту проблему к высшему контексту. Это то, что случается, когда выбрасывается исключение.

Когда выбрасывается исключение, случается несколько вещей. Во-первых, создается объект исключения тем же способом, что и любой Java объект: в куче, с помощью **new**. Затем текущий путь выполнения останавливается, и ссылка на объект исключения выталкивается из текущего контекста. В этот момент вступает механизм обработки исключений и начинает искать подходящее место для продолжения выполнения программы. Это подходящее место – обработчик исключения, чья работа - извлечь проблему, чтобы программа могла попробовать другой способ, либо просто продолжиться.

if(t == null)

throw new NullPointerException();

Есть два конструктора для всех стандартных исключений: первый - конструктор по умолчанию, и второй принимает строковый аргумент, куда можно поместить подходящую информацию в исключение:

if(t == null)

throw new NullPointerException("t = null");

Если, находясь внутри метода, возникло исключение, такой метод перейдет в процесс бросания. Если необходимо остаться в методе, необходимо установить специальный блок внутри такого метода для поимки исключения. Блок проверки – это обычный блок, которому предшествует ключевое слово **try**:

try

{

// Код, который может сгенерировать исключение

}

Выбрасывание исключения должно где-то заканчиваться. Это «место» –

*обработчик исключения*, и есть один обработчик для каждого типа исключения, которые необходимо поймать. Обработчики исключений следуют сразу за блоком проверки и объявляются ключевым словом **catch**:

try

{

// Код, который может сгенерировать исключение

}

catch(Type1 id1)

{

// Обработка исключения Type1

}

catch(Type2 id2)

{

// Обработка исключения Type2

}

catch(Type3 id3)

{

// Обработка исключения Type3

}

Для создания собственного класса исключения необходимо унаследовать его от исключения существующего типа, предпочтительно от того, которое наиболее близко подходит для вашего нового исключения.

Можно создать обработчик, ловящий любой тип исключения. Это можно сделать, перехватив исключение базового типа Exception (есть другие типы базовых исключений, но Exception – это базовый тип, которому принадлежит фактически вся программная активность):

catch(Exception e)

{

System.err.println("Caught an exception");

}

Часто есть такие места кода, которые необходимо выполнить независимо от того, было ли выброшено исключение в блоке **try**, или нет. Это обычно относится к некоторым операциям, отличным от утилизации памяти (так как об этом заботится сборщик мусора). Для достижения этого эффекта необходимо использовать предложение **finally** в конце списка всех обработчиков исключений. **finally** необходимо, когда нужно что-то установить, *отличное* от блока памяти, в его оригинальное состояние. Это может быть очистка определенного вида, такое как открытие файла или сетевого соединения.

Также для вывода отладочной информации можно использовать стандартный поток вывода System.out, который позволяет выводить на консоль данные различного типа.

try

{

// Критическая область: Опасная активность,

// при которой могут быть выброшены A, B или C

System.out.println(“Критическая область выполнилась

успешно”);

}

catch(A a1)

{

// Обработчик ситуации A

}

catch(B b1)

{

// Обработчик ситуации B

}

catch(C c1)

{

// Обработчик ситуации C

}

finally

{

// Действия, совершаемые всякий раз

}

***Разработка консольного приложения Java в среде NetBeans***

Задача - создать с помощью среды NetBeans приложение Java. Для этого необходимо запустить интегрированную среду разработки (IDE) NetBeans и выбрать в главном меню **Файл/Создать проект…** В открывшемся диалоге выбрать **Java/ Приложение Java/ Далее>**.

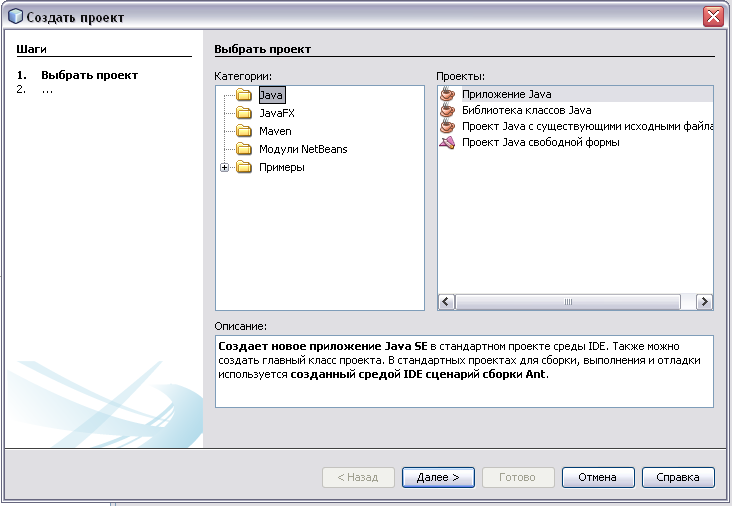


Рисунок 1 -  Создание нового проекта. Шаг 1

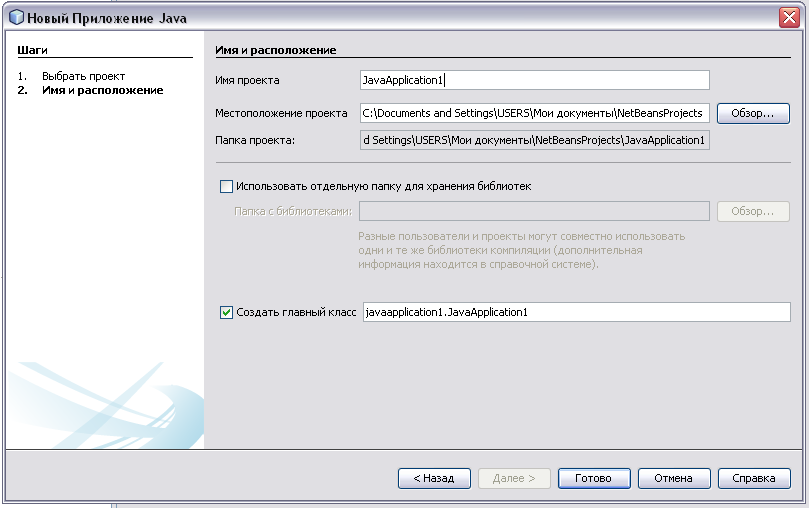
После чего можно нажимать кнопку **Готово** – значения по умолчанию для начала менять не стоит. Это можно будет делать потом после изучения среды разработки.

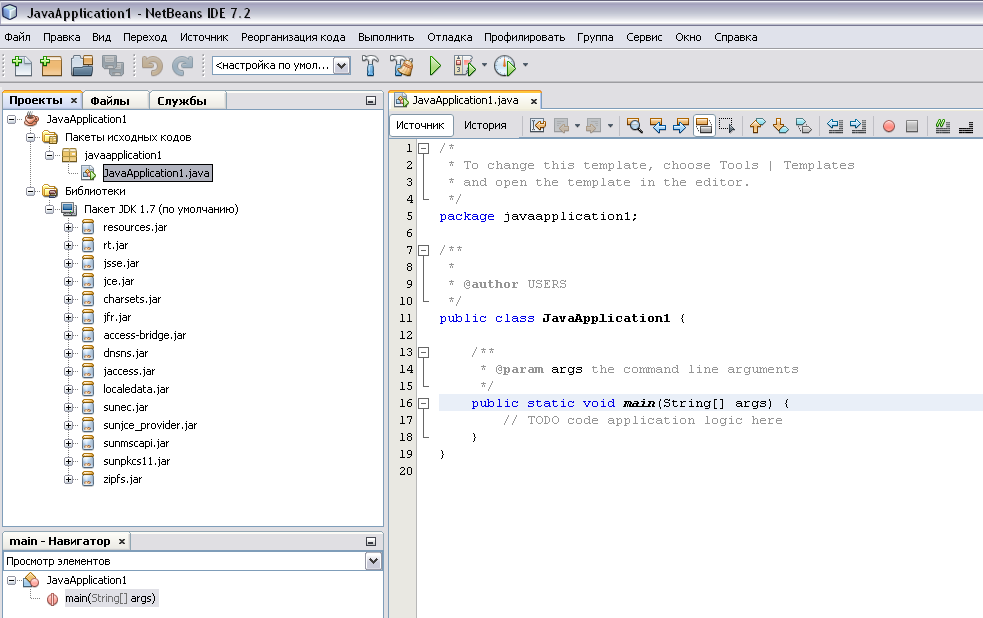
Рисунок 2 -  Создание нового проекта. Шаг 2

На следующем рисунке показано, как выглядит редактирование исходного кода приложения в среде NetBeans.

В левом верхнем окне **«Проекты»** показывается дерево проектов. В нашем случае это дерево для проекта JavaApplication1. Это окно может быть использовано для одновременного показа произвольного числа проектов. По умолчанию все деревья свернуты, и нужные узлы следует разворачивать щелчком по узлу с «плюсиком» или двойным щелчком по соответствующему имени.

В правом окне **«Источник»** показывается исходный код проекта.

В левом нижнем окне  **«Навигатор»** показывается список имен членов класса приложения – имена переменных и подпрограмм. Двойной щелчок по имени приводит к тому, что в окне редактора исходного кода происходит переход на то место, где задана соответствующая переменная или подпрограмма.

Рисунок 3 -  Редактирование исходного кода приложения

Среда NetBeans создает заготовку методов – в них имеется пустое тело. Для осуществления методом какой-либо деятельности следует дописать свой собственный код, например, код, генерирующий заданную последовательность чисел Фибоначчи. Для этого вместо комментария

// TODO code application logic here

(«описать тут логику работы приложения»)

следует написать код программы

int i = 1;

int j = 1;

int k = 0;

int n = 0;

System.out.println("Серия чисел Фибоначчи");

// создаём объект класса Scanner

Scanner sc = new Scanner(System.in);

System.out.print("Введите длину серии: ");

if(sc.hasNextInt())

{

// возвращает истину, если можно считать целое число

n = sc.nextInt(); // считывается целое число

}

else

{

System.out.println("Вы ввели не целое число");

}

while(i <= n)

{

System.out.println(+j);

j=j+k;

k=j-k;

i++;

}

Для ввода данных используется класс Scanner из пакета java.util. Его необходимо импортировать, вставив перед описанием класса

import java.util.Scanner; // импортируем класс

Для работы с потоком ввода необходимо создать объект класса Scanner, при создании указав, с каким потоком ввода он будет связан. Стандартный поток ввода (клавиатура) в Java представлен объектом — System.in.  А стандартный поток вывода (дисплей) — объектом System.out. Есть ещё стандартный поток для вывода ошибок — System.err.

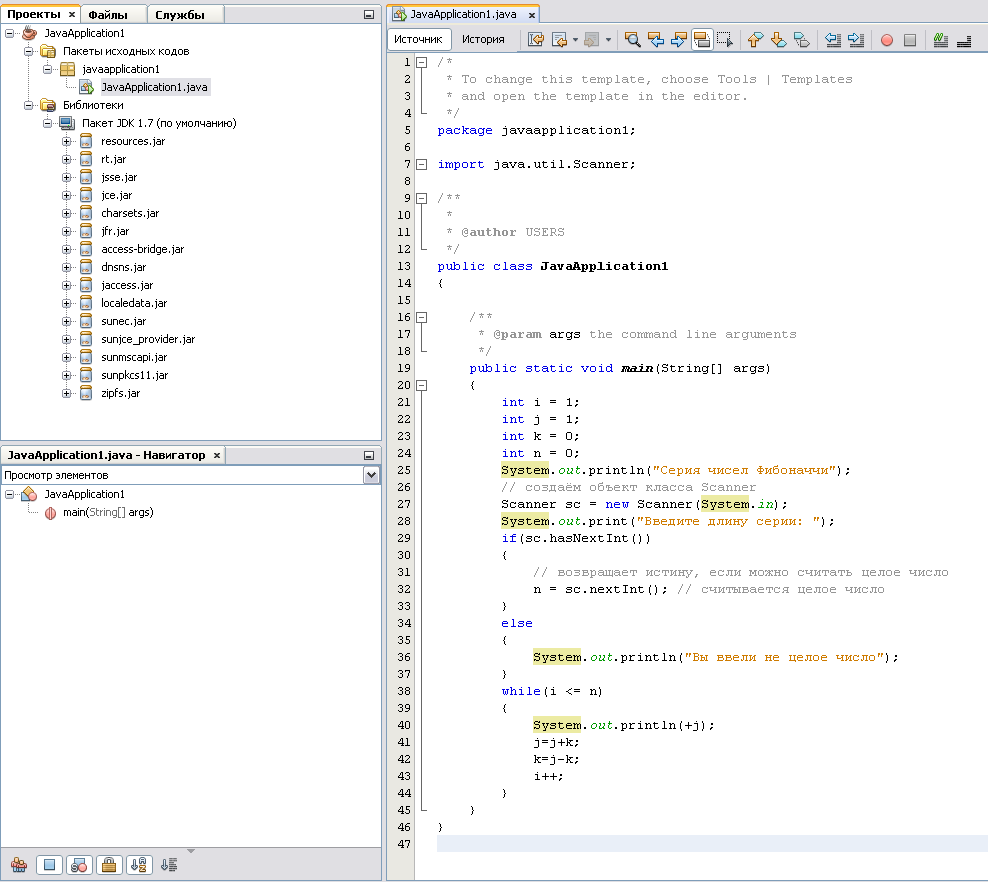


Рисунок 4 - Исходный код приложения

Консольный ввод-вывод ранее широко применялся в операционных системах, ориентированных на работу в режиме командной строки. При этом основным средством взаимодействия пользователей с программами служила текстовая консоль («пульт управления»). В ней устройством ввода служила клавиатура, а устройством вывода – окно операционной системы, обеспечивающее вывод текста в режиме пишущей машинки (системным шрифтом с буквами, имеющими одинаковую ширину). Очень много примеров программ в учебных курсах по Java ориентированы на работу в таком режиме. В настоящее время в связи с тем, что подавляющее большинство пользователей работают с программами в графическом режиме, работу в консольном режиме нельзя рассматривать как основную форму ввода-вывода. Тем более, что NetBeans позволяет без особых усилий создавать графический пользовательский интерфейс (GUI – Graphics User Interface) приложения. А консольный режим следует применять только как промежуточный, удобный в отладочном режиме как средство вывода вспомогательной информации.

После ввода программы можно перейти к компиляции файлов проекта и запуску консольного приложения.

В современных средах разработки используется два режима компиляции  - compile («скомпилировать») и build («построить»). В режиме «compile» происходит компиляция только тех файлов проекта, которые были изменены в процессе редактирования после последней компиляции. А в режиме «build» перекомпилируются заново все файлы.

Для компиляции проекта следует выбрать в меню среды разработки **Выполнить/ Собрать проект** (или, что то же, клавиша **<F11>**, или на панели инструментов иконка с молотком). При этом будут заново скомпилированы из исходных кодов все классы проекта.

Пункт **Выполнить/ Очистить и собрать проект** (или, что то же, комбинация клавиш **<Shift> <F11>**, или на панели инструментов иконка с молотком и веником) удаляет все выходные файлы проекта, после чего по новой компилируются все классы проекта.

Пункт **Выполнить/ Создать документацию Java** запускает создание документации по проекту. При этом из исходных кодов классов проекта выбирается информация, заключенная в документационные комментарии /\*\* … \*/, и на ее основе создается гипертекстовый HTML-документ.

Пункт **Выполнить/Компилировать файл** (или, что то же, клавиша **<F9>**) компилирует выбранный файл проекта.

Для того чтобы запустить скомпилированное приложение из среды разработки, следует выбрать в меню среды разработки **Выполнить/Запустить проект** (или, что то же, клавиша **<F6>**, или на панели инструментов иконка с зеленым треугольником). При запуске приложение всегда автоматически компилируется (но не «строится»), так что после внесения изменений для запуска обычно достаточно нажать **<F6>**. После запуска проекта в выходной консоли, которая находится в нижней части окна проекта, появится служебная информация о ходе компиляции и запуска.

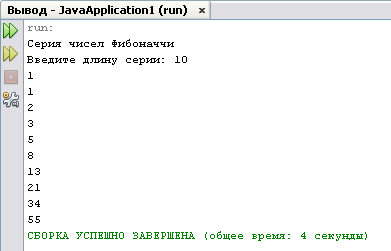


Рисунок 5 – Результат работы консольного приложения

**Порядок выполнения работы**

1. Ознакомиться с заданием на лабораторную работу
2. Написать программу, которая осуществляет расчет значения функции, используя рекуррентную формулу, указанную в вариантах к лабораторной работе в таблице 2. Для расчета значений ряда предусмотреть возможность ввода значения x и точности вычисления

Таблица 2 – Варианты заданий

|  |  |  |  |
| --- | --- | --- | --- |
| Вариант | Функция | Ряд | Реккурентная формула |
| 1 | ex |  |  |
| 2 | sin(x) |  |  |
| 3 | cos(x) |  |  |
| 4 | arctg(x) |  |  |
| 5 | ln(1+x) |  |  |
| 6 | tg(x) |  |  |
| 7 |  |  | Связь между слагаемыми Cn+1 = Cn \* T, |
| 8 |  |  |  |

Для расчета значений ряда следует создать класс со статическими методами для расчета значений i-того члена ряда и для расчета суммы ряда.

1. Составить отчет о выполнении лабораторной работы

**Содержание отчета**

1. Тема лабораторной работы
2. Цель лабораторной работы
3. Задание на лабораторную работу
4. Разработка математического и алгоритмического обеспечения
5. Разработка программного обеспечения
6. Выводы

**Контрольные вопросы и задания**

1. Описание классов
2. Примитивные типы
3. Видимость имен
4. Ключевое слово static
5. Уничтожение объектов в Java
6. Массивы в Java
7. Обработка исключений в Java
8. Написать программу, реализующую сортировку массива методом пузырька
9. Написать программу, которая вычисляет факториал числа BigInteger

любой длины

1. Написать программу, которая возводит в степень число
2. Написать программу для поиска элемента в отсортированном массиве
3. Написать программу возведения в квадрат каждого элемента массива