

Работы с системой ввода-вывода и вычислительными потоками

Система ввода-вывода

Классы для организации ввода-вывода данных в приложение, располагаются в пространстве имен `java.io`. Единственным классом который позволяет работать с файловой системой напрямую является класс `File`. Экземпляр класса `File`, можно ассоциировать с файлом или директорией, после чего, использовать методы классы для получения информации о директории или файле, а также создавать новые объекты файловой системы.

Некоторые методы класса `file` (за полным списком методов, обратитесь к документации) :

Метод	Описание
<code>getName()</code>	Возвращает имя файла или директории
<code>getPath()</code>	Возвращает относительный путь к файлу или директории
<code>getAbsolutePath()</code>	Возвращает абсолютный путь к файлу или директории
<code>exists()</code>	Возвращает истину если объект файловой системы существует
<code>isDirectory()</code>	Возвращает истину если объектом файловой системы является директория
<code>isFile()</code>	Возвращает истину если объектом файловой системы является файл
<code>lastModified()</code>	Возвращает дату последней модификации
<code>length()</code>	Возвращает размер файла в байтах
<code>renameTo(File file)</code>	Переименовывает или перемещает файл
<code>createNewFile()</code>	Создает новый файл

Рассмотрим пример работы с файловой системой, создадим файл, после чего переместим его в другой каталог.

```
// Создаем объект File, ссылающийся на файл one.txt
// Обратите внимание, что на данном этапе создается
// связь между объектом файловой системы и экземпляром класса File,
// если объект файловой системы не существует, то он НЕ СОЗДАЕТСЯ
File file = new File("c:\\one.txt");

// Вызываем метод createNewFile, для
// создания файла
file.createNewFile();
// Создаем объект File, ссылающийся на файл one.txt
File file2 = new File("D:\\two.txt");
// Перемещаем файл c:\\one.txt в файл D:\\two.txt
file.renameTo(file2);
```

Потоки ввода\вывода

Для начала дадим определение потока. Потоком называется последовательность данных, которая перемещается от источника (поток для ввода) к получателю (поток для вывода).

Потоки разделяют на байтовые и строковые, различиями между ними в том что, для байтовых минимальные размеры информации которым они оперируют - один байт, а для строковых

один символ (т.е. два байта). Также потоки делятся, на потоки чтения и записи. Как для байтовых, так и для символьных потоков существуют пары базовых классов, и набор классов наследников которые реализовывают дополнительную функциональность.

Классы потоков

	Байтовые	Строковые
Базовые классы потоков	InputStream OutputStream	Reader Writer
Фильтрованные потоки	FilterInputStream FilterOutputStream	FilterReader FilterWriter
Буферизированные потоки	BufferedInputStream BufferedOutputStream	BufferedReader BufferedWriter
Канальные потоки	PipedInputStream PipedOutputStream	PipedReader PipedWriter
Потоки для работы с файлами	FileInputStream FileOutputStream	FileReader FileWriter

Как Вы заметили существует еще одно разделение потоков, это разделение по функциональности. Потоки делятся на:

- **Фильтрованные потоки**, позволяющие применять различные фильтры к информации которая читается или пишется в поток. Такие потоки могут объединяться в цепочки, реализующие сложный фильтр.
- **Буферизированные потоки** позволяют избегать необходимости доступа к файловой системе при выполнении каждой операции чтения или записи.
- **Канальные потоки** позволяют создавать пары потоков в которых один поток пишет а другой читает.
- **Файловые потоки** позволяют работать с файлами как в двоичном так и текстовом виде.

Итак, рассмотрим класс InputStream. Класс является абстрактным, а следовательно экземпляр этого класса мы создать неможем. Но объект класса наследника (FilterInputStream, BufferedInputStream и т.д.) мы можем привести к типу InputStream.

Метод	Описание
read()	Возвращает представление очередного доступного символа во входном потоке в виде целого.
read(byte b[])	Вычитывает b.length байтов из входного потока в массив b. Возвращает количество байтов, в действительности прочитанных из потока.
read(byte b[], int off, int len)	Вычитывает len байтов, расположив их в массиве b, начиная с элемента off. Возвращает количество реально прочитанных байтов.
skip(long n)	Пропускает во входном потоке n байт. Возвращает количество пропущенных байтов.
available()	Возвращает количество байт, доступных для чтения в настоящий момент.
mark(int readlimit)	Ставит метку в текущей позиции входного потока, которую можно будет использовать до тех пор, пока из потока не будет прочитано readlimit байтов.
reset()	возвращает указатель потока на установленную ранее метку.
markSupported()	Возвращает true, если данный поток поддерживает операции mark/reset.
close()	Закрывает источник ввода. Последующие попытки чтения из этого потока приводят к возбуждению IOException.

Класс OutputStream:

Метод	Описание
write(int b)	Записывает один байт в выходной поток. Обратите внимание — аргумент этого метода имеет тип int, что позволяет вызывать write, передавая ему выражение, при этом не нужно выполнять приведение его типа к byte.
write(byte b[])	Записывает в выходной поток весь указанный массив байтов.
write(byte b[], int off, int len)	Записывает в поток часть массива — len байтов, начиная с элемента b[off].
flush()	Очищает выходные буферы, завершая операцию вывода.
close()	Закрывает выходной поток. Последующие попытки записи в этот поток будут возбуждать IOException.

Пример работы с потоками, копирование данных из файла "c:\\one.txt" в файл "D:\\two.txt"

```
//Объекты связанные с файлами на диске
File file = new File("c:\\one.txt");
File file2 = new File("D:\\two.txt");

//Буфер для чтения
char buffer[] = new char[255];

try {
    //Создание потока для чтения
    FileReader rdr = new FileReader(file);
    //Создание потока для записи
    FileWriter wrt = new FileWriter(file2);

    while(rdr.read(buffer)!=-1){
        wrt.write(buffer);
    }
    rdr.close();
    wrt.close();
} catch (IOException e) {
    System.out.print("Ошибка чтения/записи");
}
```

После того как мы ознакомились с основными принципами работы с потоками, давайте перейдем к рассмотрению потоков с дополнительной функциональностью.

Фильтрованные потоки

Как говорилось ранее - это потоки позволяющие применять набор фильтров к обрабатываемой информации. Рассмотрим пример, в котором создадим фильтрованный поток позволяющий считывать данные с текстового документа, и преобразовывать их в заглавные буквы.

```
import java.io.FilterReader;
import java.io.Reader;
import java.io.IOException;
public class UpperCaseConvertor extends FilterReader {
    public UpperCaseConvertor(Reader reader) {
        super(reader);
    }
    public int read() throws IOException {
        int c = super.read();
        return (c == -1 ? c : Character.toUpperCase((char)c));
    }
}
```

```

    }
    public int read(char[] buf, int offset, int count) throws IOException {
        int nread = super.read(buf, offset, count);
        int last = offset + nread;
        for (int i = offset; i < last; i++) {
            buf[i] = Character.toUpperCase(buf[i]);
        }
        return nread;
    }
}

```

```

import java.io.StringReader;
import java.io.FilterReader;
import java.io.IOException;

public classUpperCaseConvertorTest {
    public static void main(String[] args) throws IOException {
        StringReader src = new StringReader(args[0]);
        FilterReader f = newUpperCaseConvertor(src);
        int c;
        while ((c = f.read()) != -1)
            System.out.print((char)c);
        System.out.println();
    }
}

```

Итак, мы создали класс наследник от FilterReader, и перегрузили метод read который считанные символы переводит в верхний регистр с помощью метода toUpperCase класса Character.

Буферизированные потоки

Для повышения быстродействия приложения, при считывании с внешних устройств данные зачастую буферизируют. Это позволяет сократить количество обращений к файловой системе, т.к. на уровне файловой системы чтение происходит не побайтово, а буферизированно. Для буферизации данных служат классы BufferedInputStream и BufferedOutputStream для работы с бинарными данными, и классы BufferedReader, BufferedWriter для работы с текстовыми данными. Классы содержат массивы, которые служат буфером для обрабатываемых данных. То есть когда байты из потокачитываются либо пропускаются (метод skip()), сначала заполняется буферный массив, причем, из надстраиваемого потока загружается сразу много байт, чтобы не требовалось обращаться к нему при каждой операции read или skip. Также класс BufferedInputStream добавляет поддержку методов mark() и reset(). Эти методы определены еще в классе InputStream, но там их реализация по умолчанию возбуждает исключение IOException. Метод mark() запоминает точку во входном потоке, а вызов метода reset() приводит к тому, что все байты, полученные после последнего вызова mark(), будут считываться повторно, прежде, чем новые байты начнут поступать из надстроенного входного потока. BufferedOutputStream предоставляет возможность производить многоократную запись небольших блоков данных без обращения к устройству вывода при записи каждого из них. Сначала данные записываются во внутренний буфер. Непосредственное обращение к устройству вывода и, соответственно, запись в него, произойдет, когда буфер заполнится. Инициировать передачу содержимого буфера на устройство вывода можно и явным образом, вызвав метод flush(). Так же буфер освобождается перед закрытием потока. При этом будет закрыт и надстраиваемый поток.

Проведем небольшой эксперимент. Произведем чтение из файла используя обычный поток и буферизованный.

```

try {
    String fileName = "d:\\file1";
    InputStream inStream = null;
    OutputStream outStream = null;

    //Записываем в файл некоторое количество байт
    long timeStart = System.currentTimeMillis();
    outStream = new FileOutputStream(fileName);
    outStream = new BufferedOutputStream(outStream);
    for(int i=1000000; --i>=0;) {
        outStream.write(i);
    }
    long time = System.currentTimeMillis() - timeStart;
    System.out.println("Время записи: " + time + " миллисекунд");
    outStream.close();

    //Определяем время считывания без буферизации
    timeStart = System.currentTimeMillis();
    inStream = new FileInputStream(fileName);
    while(inStream.read()!=-1){
    }
    time = System.currentTimeMillis() - timeStart;
    inStream.close();
    System.out.println("На чтение затрачено: " + (time) + " миллисекунд");

    //Определяем время считывания с буферизацией
    timeStart = System.currentTimeMillis();
    inStream = new FileInputStream(fileName);
    inStream = new BufferedInputStream(inStream);
    while(inStream.read()!=-1){
    }
    time = System.currentTimeMillis() - timeStart;
    inStream.close();
    System.out.println("На буферизированное чтение затрачено: " + (time) + " миллисекунд");
} catch (IOException e) {
    System.out.println("IOException: " + e.toString());
    e.printStackTrace();
}
}

```

Результат работы

```

Время записи: 252 миллисекунд
На чтение затрачено: 6716 миллисекунд
На буферизированное чтение затрачено 29 миллисекунд

```

Невооруженным глазом видно, что прирост производительности при использовании буфферизированных потоков на лицо.

Потоки размещаемые в оперативной памяти

В библиотеке классов Java есть три класса, специально предназначенных для создания потоков в оперативной памяти. Это классы ByteArrayOutputStream, ByteArrayInputStream

Класс ByteArrayOutputStream

Класс ByteArrayOutputStream является наследником класса OutputStream. В нем имеется

два конструктора:

```
public ByteArrayOutputStream();
public ByteArrayOutputStream(int size);
```

Первый из этих конструкторов создает выходной поток в оперативной памяти с начальным размером буфера, равным 32 байта. Второй позволяет указать необходимый размер буфера. Методы класса ByteArrayOutputStream определены методы:

Метод	Описание
void reset();	Сбрасывает счетчик байт, записанных в выходной поток. Если данные, записанные в поток вам больше не нужны, вы можете вызвать этот метод и использовать память, выделенную для потока, для записи других данных.
int size()	Определить количество байт данных, записанных в поток
byte[] toByteArray();	Позволяет скопировать данные, записанные в поток, в массив байт. Этот метод возвращает адрес созданного для этой цели массива.
void writeTo (OutputStream out);	Копирует содержимое данного потока в другой выходной поток, ссылка на который передается методу через параметр

Класс ByteArrayInputStream

С помощью класса ByteArrayInputStream вы можете создать входной поток на базе массива байт, расположенного в оперативной памяти. В этом классе определено два конструктора:

```
public ByteArrayInputStream(byte buf[]);
public ByteArrayInputStream(byte buf[], int offset, int length);
```

Первый конструктор получает через единственный параметр ссылку на массив, который будет использован для создания входного потока. Второй позволяет дополнительно указать смещение offset и размер области памяти length, которая будет использована для создания потока.

Метод	Описание
int available()	Возвращает количество доступных для чтения байт
void reset()	Сбрасывает счетчик байт

Так же, класс имеет реализацию методов read и skip, которые работают так же как и у всех потоков чтения.

Сериализация объектов

Преобразование объекта в последовательность байт называют сериализацией.

Поскольку сериализованный объект – это последовательность байт, которую можно легко сохранить в файл, передать по сети и т.д., то и объект затем можно восстановить на любой машине, вне зависимости от того, где проводилась сериализация.

Для того, чтобы сериализовать объект, класс, от которого он порожден, должен реализовывать интерфейс java.io.Serializable. В этом интерфейсе отсутствуют определения методов, а нужен он лишь для указания того, что объекты класса могут быть сериализованы. При попытке сериализовать объект, не имеющий такого интерфейса, будет возбуждена исключительная ситуация java.

io.NotSerializableException.

Классы ObjectInputStream и ObjectOutputStream, позволяют сериализовать \ десериализовать объекты. Для того, чтобы сериализировать объект, нужен выходной поток OutputStream, в который будет записываться сгенерированная последовательность байт. Этот поток передается в конструктор ObjectOutputStream. Затем вызовом метода writeObject() объект сериализуется и записывается в выходной поток. Например:

```
ByteArrayOutputStream os = new ByteArrayOutputStream();
Object objSave = new Integer(1);
ObjectOutputStream oos = new ObjectOutputStream(os);
oos.writeObject(objSave);
```

Результат сериализации Вы можете увидеть вызвав метод toByteArray():

```
byte[] bArray = os.toByteArray();
```

А чтобы восстановить объект, его нужно десериализовать из этого массива:

```
ByteArrayInputStream is = new ByteArrayInputStream(bArray);
ObjectInputStream ois = new ObjectInputStream(is);
Object objRead = ois.readObject();
```

Теперь можно убедиться, что восстановленный объект идентичен исходному:

```
System.out.println("readed object is: " + objRead.toString());
System.out.println("Object equality is: " + (objSave.equals(objRead)));
System.out.println("Reference equality is: " + (objSave==objRead));
```

Результатом выполнения приведенного выше кода будет:

```
readed object is: 1
Object equality is: true
Reference equality is: false
```

Обратите внимание что восстановленный объект не совпадает с исходным, но по значению они равны.

Сериализацию, активно используют для сохранения \ востонавления состояния объекта. Как правило состояния объекта описывается значениями его полей. Для того что бы объект мог быть успешно десериализован у него должен присутствовать конструктор по умолчанию. Т. к. десериализация - это процесс создания объекта (а неотъемлемая часть процесса создания объекта - есть вызов конструктора) и заполнения его полей значениями.

Иногда в объектах существуют поля которые не требуют сериализации, определив перечень таких полей можно увеличить производительность Вашего приложения. К примеру, Если сериализованный объект передается по сети, то исключение такого поля из сериализации позволяет уменьшить нагрузку на сеть. Итак рекомендуется не сериализовать:

- Поля которые хранят временные данные, и требуются при различных расчетах.
- Поля значения которых можно получить на основе значений других полей, к примеру в классе Заказ с полями "количество товара", "цена за единицу", "сумма заказа". Поле "сумма заказа" является

вычисляемым ("количество товара" X "цена за единицу"), сериализация \ десериализация этого поля займет больше времени чем вычисление.

- Поля хранящие конфиденциальную информацию (пароль, и т.д.). Для таких полей следует разработать отдельный механизм передачи, возможно основанный на сериализации. Для того чтобы исключиться поле из списка сериализуемых, следует установить спецификатор transient.

```
class Order implements java.io.Serializable {  
    private int amount;  
    private double price;  
    private transient double total;  
}
```

В приведенном выше классе, поле total исключается из списка сериализуемых.

Граф сериализации

До этого мы рассматривали объекты, которые имеют поля лишь примитивных типов. Если же сериализуемый объект ссылается на другие объекты, их также необходимо сохранить (записать в поток байт), а при десериализации – восстановить. Эти объекты, в свою очередь, также могут ссылаться на следующие объекты. При этом важно, что если несколько ссылок указывают на один и тот же объект, то этот объект должен быть сериализован лишь однажды, а при восстановлении все ссылки должны вновь указывать на него одного. Например, сериализуемый объект A ссылается на объекты B и C, каждый из которых, в свою очередь, ссылается на один и тот же объект D. После десериализации не должно возникнуть ситуации, когда B ссылается на D1, а C – на D2, где D1 и D2 – равные, но все же различные объекты. Для организации такого процесса стандартный механизм сериализации строит граф, включающий в себя все участвующие объекты и ссылки между ними. Если очередная ссылка указывает на некоторый объект, сначала проверяется – нет ли такого объекта в графе. Если есть – объект второй раз не сериализуется. Если нет – новый объект добавляется в граф. При построении графа может встретиться объект, порожденный от класса, не реализующего интерфейс Serializable. В этом случае сериализация прерывается, генерируется исключение java.io.NotSerializableException.

Потоки выполнения.

Итак, в этом разделе мы рассмотрим как устроено многопоточное программирование в JAVA.

Класс Thread

Поток выполнения в Java представляется экземпляром класса Thread. Для того, чтобы создать свой поток выполнения, необходимо пронаследовать от этого класса и переопределить метод run().

```
public class CalcThread extends Thread {  
    int start,end;  
  
    CalcThread(int start,int end){  
        this.start = start;  
        this.end = end;  
    }  
  
    public void run(){  
        int sum= 0;  
        for(int i= start; i<=end; i++)  
            sum+=i;  
        System.out.println("Сумма чисел на интервале "+start+" - "+end+" = "+ sum);  
    }  
}
```

```
}
```

Метод run() содержит программный код, который будет выполняться в новом потоке. Чтобы запустить его, необходимо создать экземпляр класса-наследника и вызвать унаследованный метод start(), который сообщает виртуальной машине, что требуется запустить новый поток исполнения и начать выполнять в нем метод run().

```
CalcThread t = new CalcThread(10,100);
t.start();
```

Интерфейс Runnable

Описанный подход имеет один недостаток. Поскольку в Java множественное наследование отсутствует, требование наследоваться от Thread может привести к конфликту. Если еще раз посмотреть на приведенный выше пример, станет понятно, что наследование производилось только с целью переопределения метода run(). Поэтому предлагается более простой способ создать свой поток исполнения. Достаточно реализовать интерфейс Runnable, в котором объявлен только один метод – уже знакомый void run(). Перепишем пример, приведенный выше, с помощью этого интерфейса:

```
public class CalcRunnable implements Runnable {
    int start,end;

    CalcRunnable(int start,int end){
        this.start = start;
        this.end = end;
    }

    public void run(){
        int sum= 0;
        for(int i= start; i<=end; i++)
            sum+=i;
        System.out.println("Сумма чисел на интервале "+start+" - "+end+" = "+ sum);
    }
}
```

Также незначительно меняется процедура запуска потока:

```
Runnable r = new CalcRunnable(10,100);
Thread t = new Thread(r);
t.start();
```

Если раньше объект, представляющий сам поток выполнения, и объект с методом run(), реализующим необходимую функциональность, были объединены в одном экземпляре класса MyThread, то теперь они разделены. Какой из двух подходов удобней, решается в каждом конкретном случае. Следует заметить, что Runnable не является полной заменой классу Thread, поскольку создание и запуск самого потока исполнения возможно только через метод Thread.start().

Работа с приоритетами

Для работы с приоритетами в Java, у класса Thread существуют два метода:

- **getPriority()**- возвращает приоритет потока

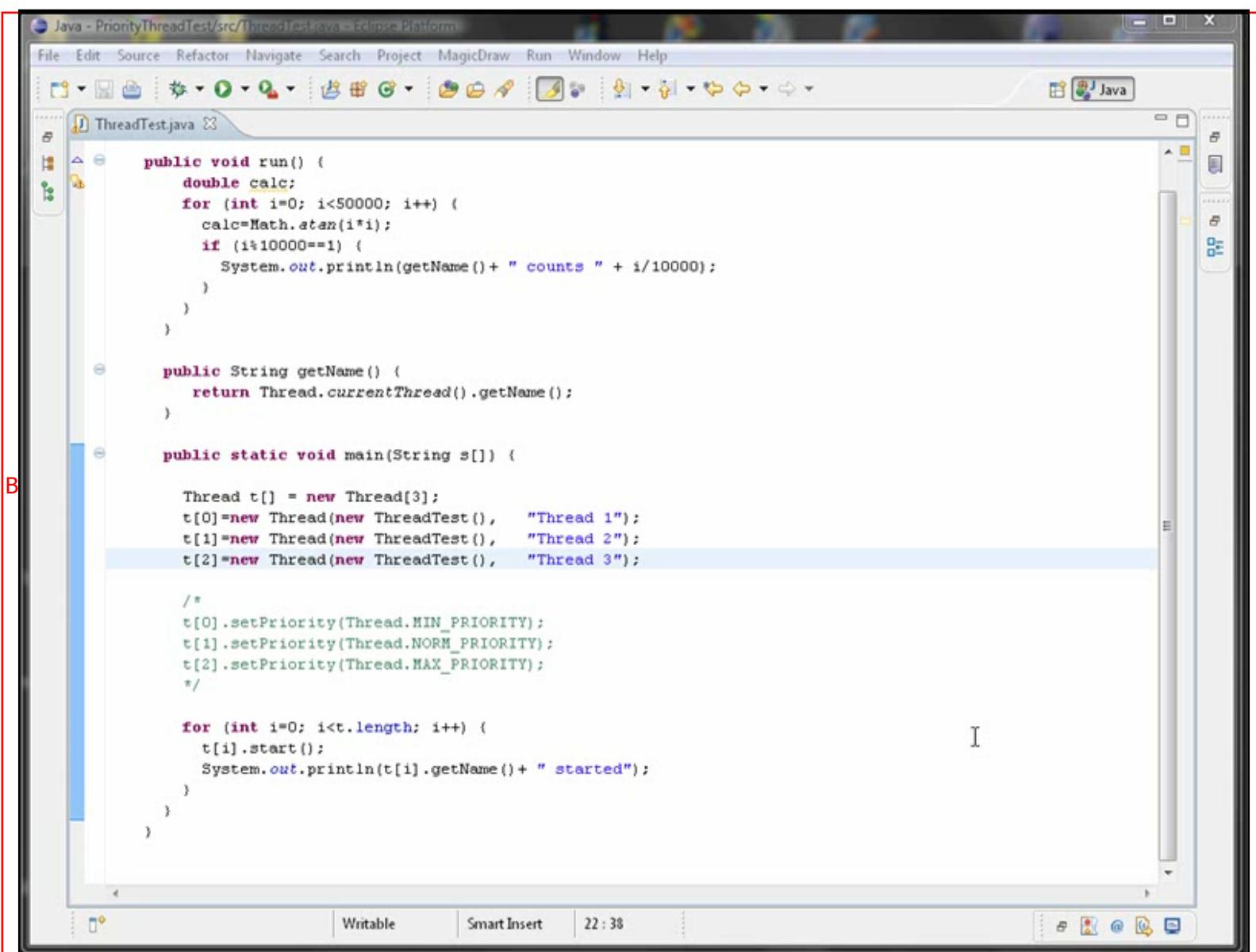
- **setPriority(int priority)**- устанавливает приоритет потока

Значения поределяющие приоритет потока хранятся в константах MIN_PRIORITY (минимальный приоритет), MAX_PRIORITY (максимальный приоритет), NORM_PRIORITY (нормальный приоритет).

Для начала просмотра видео, сделайте двойной клик на изображении ниже.

Дополнительная информация к видеодополнениям.

1. Каждому потоку который создается, автоматически назначается имя. Мы можем назначать имена потокам самостоятельно, используя соответствующий конструктор. Получить имя потока можно с помощью метода `getName()`
2. Статическим метод `currentThread()` класса `Thread`, возвращает текущий выполняемый поток.



```

Java - PriorityThreadTest/src/ThreadTest.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project MagicDraw Run Window Help
ThreadTest.java
public void run() {
    double calc;
    for (int i=0; i<50000; i++) {
        calc=Math.atan(i*i);
        if (i%10000==1)
            System.out.println(getName()+" counts "+ i/10000);
    }
}

public String getName() {
    return Thread.currentThread().getName();
}

public static void main(String s[]) {
    Thread t[] = new Thread[3];
    t[0]=new Thread(new ThreadTest(), "Thread 1");
    t[1]=new Thread(new ThreadTest(), "Thread 2");
    t[2]=new Thread(new ThreadTest(), "Thread 3");

    /*
    t[0].setPriority(Thread.MIN_PRIORITY);
    t[1].setPriority(Thread.NORM_PRIORITY);
    t[2].setPriority(Thread.MAX_PRIORITY);
    */

    for (int i=0; i<t.length; i++) {
        t[i].start();
        System.out.println(t[i].getName()+" started");
    }
}
}

```

Прерывание выполнения потоков

В текущей версии Java не предусмотрено никакого способа принудительного прекращения работы потока. Вместо этого применяется метод `interrupt`, который запрашивает о возможности

прекращения работы потока. Это значит, что метод run должен периодически проверять, не следует ли ему прекратить выполнение:

```
public void run() {
    while(нет запроса на прекращение работы и есть какая-то работа) {
        //выполнение какой-то работы
    }
}
```

С помощью метода interrupted() мы можем проверить был ли вызван метод interrupt.

```
public void run() {
    while(!interrupted()) {
        //выполнение какой-то работы
    }
}
```

Как известно, работа потока выполняется не постоянно, а иногда приостанавливается, чтобы дать возможность другим потокам решать свои задачи. Когда выполнение потока приостановлено, он не может проверить, следует ли завершить работу, и для этого применяется исключительная ситуация InterruptedException. Когда для блокированного объекта вызывается метод interrupt, работа блокирующего вызова (метода sleep или wait) завершается, и управление передается обработчику исключительной ситуации InterruptedException.

```
public void run() {
    try{
        while(!interrupted()){
            //выполнение какой-то работы
        }catch(InterruptedException ex){
            //обработка исключительной ситуации
        }
    }
}
```

Синхронизация потоков

В многопоточной модели программирования, очень часто встречается ситуация когда, два или более потоков начинают работать с одним ресурсом (массивом, файлом и т.д.). В следствии чего предугадать развязку "состязаний" глядя в исходный код становится практически невозможно. Для того что бы разограничить доступ к ресурсу используется механизм синхронизации потоков, который основан на принципах блокировки объекта. Объект блокируется таким образом, что доступ к нему в один момент времени может получить только один поток.

```
synchronized(some_object) {
    //Код приложения, который работает с объектом
}
```

Синхронизация потоков (видеодополнение)

Для начала просмотра видео, сделайте двойной клик на изображении ниже.

The screenshot shows the Eclipse IDE interface with the title bar "Java - PriorityThreadTest/src/ThreadTest.java - Eclipse Platform". The menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, MagicDraw, Run, Window, Help. The toolbar has various icons for file operations like Open, Save, Cut, Copy, Paste, Find, etc. The left sidebar shows the project structure with "ThreadTest.java" selected. The main editor area contains Java code for creating three threads and calculating atan values. The code uses the synchronized keyword on the main method. The status bar at the bottom shows "Writable", "Smart Insert", and the time "22:38".

```
public void run() {
    double calc;
    for (int i=0; i<50000; i++) {
        calc=Math.atan(i*i);
        if (i%10000==1)
            System.out.println(getName()+" counts "+ i/10000);
    }
}

public String getName() {
    return Thread.currentThread().getName();
}

public static void main(String s[]) {

    Thread t[] = new Thread[3];
    t[0]=new Thread(new ThreadTest(), "Thread 1");
    t[1]=new Thread(new ThreadTest(), "Thread 2");
    t[2]=new Thread(new ThreadTest(), "Thread 3");

    /*
    t[0].setPriority(Thread.MIN_PRIORITY);
    t[1].setPriority(Thread.NORM_PRIORITY);
    t[2].setPriority(Thread.MAX_PRIORITY);
    */

    for (int i=0; i<t.length; i++) {
        t[i].start();
        System.out.println(t[i].getName()+" started");
    }
}
}
```

В Java существует возможность создавать не только синхронизированные участки кода но и синхронизированные методы. Если метод объекта является синхронизированным то в один момент времени с ним может работать только один поток, ограничение доступа к методу регулируется на уровне объекта, а если метод является статическим то ограничение доступа происходит на уровне класса. К примеру реализуем код из видеодополнения через синхронизированный метод (для этого укажем спецификатор synchronized при объявлении метода)

```
public class ThreadSyncTest implements Runnable {
    private static ThreadSyncTest obj = new ThreadSyncTest();
    public synchronized void someFunctionality() {
        for(int i=0; i<3; i++) {
            System.out.println(Thread.currentThread().getName()+" "+i);
        }
    }

    public void run() {
        obj.someFunctionality();
    }

    public static void main(String s[]) {
        for(int i=0; i<3; i++) {
            new Thread(new ThreadSyncTest(), "Thread-"+i).start();
        }
    }
}
```

```
    }
}
```

Группы потоков

В некоторых программах содержится довольно много потоков. В этом случае имеет смысл разбить их на функциональные категории. Допустим, что в Web-браузере несколько потоков пытаются загрузить несколько изображений с Web-сервера. Если пользователь нажимает кнопку Stop (чтобы прервать загрузку текущей страницы), то было бы удобно прервать выполнение одновременно всех потоков загрузки изображений. В языке Java предусмотрен механизм создания так называемой группы потоков (threadgroup) для одновременной работы с несколькими потоками. Для создания группы применяется следующий конструктор:

```
ThreadGroup myGroup = new ThreadGroup( "DownloadsThread" );
```

Строковый аргумент конструктора ThreadGroup идентифицирует группу и поэтому должен быть уникальным. Потоки добавляются в группу за счет указания группы в конструкторе потока, как показано ниже.

```
Thread t = new Thread( myGroup, threadName );
```

Для проверки наличия выполняемых потоков в группе запущенных потоков применяется метод activeCount.

```
if(myGroup.activeCount() == 0){
// все потоки в группе остановлены
}
```

Для прерывания всех потоков в группе достаточно вызвать метод interrupt для объекта-группы.

```
myGroup.interrupt(); // прерывает все потоки в группе myGroup
```

Группы потоков могут содержать дочерние подгруппы. По умолчанию новая группа потоков становится дочерней по отношению к текущей, но в конструкторе можно явно указать имя родительской группы. Методы activeCount и interrupt относятся ко всем потокам группы и ее дочерним подгруппам. Замечательное свойство групп потоков заключается в том, что группа может послать уведомление, если поток был остановлен в результате возникновения исключительной ситуации. Для обработки этой исключительной ситуации необходимо создать дочерний класс на основе класса ThreadGroup и переопределить метод uncaughtException. После чего можно заменить выполняемые по умолчанию действия (т.е. вывод состояния стека в стандартный поток ошибок) какими-то более сложными действиями, например, записать ошибки в файл или вывести их в диалоговом окне.

Домашнее задание

К домашнему заданию из предыдущего урока реализуйте в классе который логирует ошибки, автосохранение ошибок в файл. К примеру, если размер буфера который содержит ошибки привышает 30-ть записей то, эту информацию необходимо дописать в файл и очистить буфер. Реализуйте это в виде трех отдельных потоков, каждый из которых записывает по 10-

ть сообщений.