Data streams

Класе `DataInputStream` и `DataOutputStream` обезбеђују методе за читање и писање података примитивних Јава типова и стрингова у бинарном формату. Бинарни формати који се том приликом користе примарно су намењени размени података између два Јава програма кроз мрежну конекцију, фајл или нешто треће. Оно што излазни ток пише, улазни може да чита.
Међутим, исте те формате користи и већина Интернет протокола који размењују бинарне бројеве.
Нпр. time протокол користи 32-битне big-endian целе бројеве, управо попут Јавиног типа `int`. Слично је и са 32-битним IEEE 754 бројевима у покретном зарезу, који одговарају Јавином типу `float`. Ово отуда што су и Јаву и већину мрежних протокола дизајнирали Unix програмери, па и једно и друго теже коришћењу формата који су заједнички за већину Unix система.
Међутим, то не важи за све мрежне протоколе, па је неопходно проверити детаље протокола који се користи. Нпр. Network Time Protocol (NTP) представља времена 64-битним бројевима у фиксном зарезу, где је целобројни део у прва, а фракција у друга 32 бита. То не одговара ниједном примитивном типу података ни у једном уобичајеном програмском језику, иако је прилично праволинијски радити са њим - бар што се тиче NTP-а.

Класа `DataOutputStream` поседује 11 метода за писање одређених Јава типова података:

```
public final void writeBoolean(boolean b) throws IOException
public final void writeByte(int b) throws IOException
public final void writeShort(int s) throws IOException
public final void writeChar(int c) throws IOException
public final void writeInt(int i) throws IOException
public final void writeLong(long l) throws IOException
public final void writeFloat(float f) throws IOException
public final void writeDouble(double d) throws IOException
public final void writeChars(String s) throws IOException
public final void writeBytes(String s) throws IOException
public final void writeUTF(String s) throws IOException
```

Сви подаци се записују у big-endian формату.
Цели бројеви се записују у потпуном комплементу: `byte` у 1 бајту, `short` у 2, `int` у 4, а `long` у 8 бајтова.
Реални бројеви се записују у IEEE 754 формату: `float` у 4, а `double` у 8 бајтова.
`boolean` се записује као 1 бајт са вредношћу 0 за `false`, односно 1 за `true`.
`char` се записује као два неозначена бајта.

Последња три метода:
`writeChars()` просто итерира кроз свој аргумент типа `String` и записује сваки карактер као 2-бајтни big-endian Unicode карактер (прецизније као UTF-16 code point)

`writeBytes()` итерира кроз аргумент типа `String`, али пише само бајт најмање тежине сваког карактера. Према томе, долази до губитка информације за све стрингове који садрже карактере изван карактерског скупа Latin-1. Овај метод може бити користан за мрежне протоколе који користе ASCII кодирање, али већи део времена га треба избегавати.

Ни `writeChars()` ни `writeBytes()` не кодирају дужину стринга у излазни ток. Због тога није могуће направити разлику између низа карактера и карактера који сачињавају стринг.
Метод `writeUTF()` укључује дужину стринга. Он кодира стринг варијантом UTF-8 кодирања која није компатибилна са већином софтвера који није писан у Јави, па га треба **користити само за размену података са другим Јава програмима који користе `DataInputStream` метод `readUTF()` за читање стрингова. За размену UTF-8 кодираног текста са свим осталим софтвером, потребно је користити `InputStreamReader` уз одговарајуће кодирање.**

Осим горњих метода за писање бинарних бројева и стрингова, класа `DataOutputStream` поседује и уобичајене методе `write()`, `flush()` и `close()` које поседује свака `OutputStream` класа.

`DataInputStream` је класа комплементарна класи `DataOutputStream`. Сваки формат који `DataOutputStream` пише `DataInputStream` може да прочита. Додатно, `DataInputStream` поседује уобичајене методе `read()`, `available()`, `skip()` и `close()`, као и методе за читање целих низова бајтова и линија текста.

Постоји 9 метода за читање бинарних података који се упарују са 11 метода класе `DataOutputStream` (не постоје тачни комплементи за `writeBytes()` и `writeChars()` - они се обрађују читањем бајтова и карактера један по један):

```
public final boolean readBoolean( ) throws IOException
public final byte readByte( ) throws IOException
public final char readChar( ) throws IOException
public final short readShort( ) throws IOException
public final int readInt( ) throws IOException
public final long readLong( ) throws IOException
public final float readFloat( ) throws IOException
public final double readDouble( ) throws IOException
public final String readUTF( ) throws IOException
```

Додатно, постоје два метода за читање неозначених бајтова и неозначених `short`-ова, који враћају еквивалентан `int`. Јава не поседује ниједан од ова да типа података, али може се наићи на њих приликом читања бинарних података које је писао C-програм:

```
public final int readUnsignedByte( ) throws IOException
public final int readUnsignedShort( ) throws IOException
```

Постоје и два уобичајена мултибајт `read()` метода који читају податке у низ или подниз и враћају број прочитаних бајтова. Такође, постоје и два `readFully()` метода који узастопно читају податке из придруженог улазног тока у низ све док не прочитају задати број бајтова. Уколико не може бити прочитана довољна количина података, избацује се изузетак типа `IOException`. Ови методи су посебно корисни када унапред знамо колико бајтова треба да прочитамо. То може бити случај када смо прочитали поље Content-length из HTTP заглавља.

```
public final int read(byte[] input) throws IOException
public final int read(byte[] input, int offset, int length) throws IOException
public final void readFully(byte[] input) throws IOException
public final void readFully(byte[] input, int offset, int length) throws IOException
```

Најзад, постоји популарни метод `readLine()` који чита линију текста и враћа стринг:

```
public final String readLine( ) throws IOException
```

Међутим, овај метод не треба користити ни под којим околностима.
Линије текста треба читати користећи метод `readLine()` класе `BufferedReader`.

Compressing Streams

Пакет `java.util.zip` садржи филтер токове који компресују токове у zip и gzip формате. Овај пакет омогућује да Јава апликације једноставно размене компресоване податке преко мреже. HTTP 1.1 укључује подршку за трансфер компресованих фајлова, у коме сервер компресује, а browser декомпресује фајлове, правећи компромис између јевтине процесорске снаге и скупог мрежног протока.

Постоји 6 класа за компресију и декомпресију; улазни токови декомпресују податке, а излазни их компресују:

```
public class DeflaterOutputStream extends FilterOutputStream
public class InflaterInputStream extends FilterInputStream
public class GZIPOutputStream extends FilterOutputStream
public class GZIPInputStream extends FilterInputStream
public class ZipOutputStream extends FilterOutputStream
public class ZipInputStream extends FilterInputStream
```

Све ове класе користе исти алгоритам за компресију. Разликују се само по различитим константама и мета-информацијама које додају компресованим подацима. Додатно, zip ток може садржати више од једног компресованог фајла.

Компресија и декомпресија података овим класама је скоро тривијална. Просто се оланча филтер на придружени ток и чита и пише уобичајено.
Нпр. претпоставимо да желимо да читамо компресовани фајл allnames.gz. Просто отворимо `FileInputStream` за фајл и оланчамо `GZIPInputStream` на њега:

```
FileInputStream fin = new FileInputStream("allnames.gz");
GZIPInputStream gzin = new GZIPInputStream(fin);
```

Надаље, можемо читати некомпресоване податке из `gzin` користећи уобичајене методе `read()`, `skip()` и `available()`. Нпр. следећи фрагмент кода чита и декомпресује фајл allnames.gz у текући директоријум:

```
FileInputStream fin = new FileInputStream("allnames.gz");
GZIPInputStream gzin = new GZIPInputStream(fin);
FileOutputStream fout = new FileOutputStream("allnames");
int b = 0;
while ((b = gzin.read( )) != -1) fout.write(b);
gzin.close( );
fout.flush( );
fout.close( );
```

Заправо, за ово није неопходно ни знати да је `gzin` типа `GZIPInputStream`. Тип `InputStream` радио би подједнако добро. Нпр.

```
InputStream in = new GZIPInputStream(new FileInputStream("allnames.gz"));
```

`ZipInputStream` и `ZipOutputStream` су мало компликованије јер .zip фајл је заправо архива која може садржати већи број уноса које је неопходно прочитати сваки понаособ. Сваки фајл у .zip архиви представљен је објектом типа `ZipEntry`, чији метод `getName()` враћа оригинално име фајла. Нпр. следећи фрагмент кода декомпресује архиву shareware.zip у текући директоријум:

```
FileInputStream fin = new FileInputStream("shareware.zip");
ZipInputStream zin = new ZipInputStream(fin);
ZipEntry ze = null;
int b = 0;
while ((ze = zin.getNextEntry( )) != null) {
      FileOutputStream fout = new FileOutputStream(ze.getName( ));
      while ((b = zin.read( )) != -1) fout.write(b);
      zin.closeEntry( );
      fout.flush( );
      fout.close( );
}
zin.close( );
```

Digest Streams

Пакет `java.util.security` садржи два филтер тока која могу да рачунају резиме (енг. digest) поруке за ток. То су `DigestInputStream` и `DigestOutputStream`. Резиме поруке, представљен у Јави објектом класе `java.util.security.MessageDigest`, је јаки хеш-код за ток, тј. велики цео број (обично дужине 20 бајтова у бинарном формату) који се може једноставно израчунати за ток произвољне дужине, а тако да ниједна информација о току није доступна из резимеа. Резимеи порука могу се користити за дигиталне потписе и за откривање података који су измењени на путу кроз мрежу.

У пракси, употреба резимеа порука за дигиталне потписе је важнија. За откривање грешака у подацима могу се користити доста једноставнији и мање рачунски захтевни алгоритми. Међутим, резиме филтер токови се толико једноставно користе да се исплати платити рачунску цену зарад раста продуктивности програмера.

Да би се израчунао резиме излазног тока, најпре се конструише објекат типа `MessageDigest` који користи одређени алгоритам, попут Secure Hash Algorithm (SHA). Затим се `MessageDigest` објекат и ток за који се жели резиме прослеђују конструктору класе `DigestOutputStream`. Овим се резиме ток оланчава на придружени излазни ток. Потом се подаци пишу у ток нормално, уради се flush, затвори ток и позове метод `getMessageDigest()` како би се дохватио `MessageDigest` објекат. Коначно, позове се метод `digest()` за `MessageDigest` објекат како би се израчунао резиме. Нпр.

```
MessageDigest sha = MessageDigest.getInstance("SHA");
DigestOutputStream dout = new DigestOutputStream(out, sha);
byte[] buffer = new byte[128];
while (true) {
      int bytesRead = in.read(buffer);
      if (bytesRead < 0) break;
      dout.write(buffer, 0, bytesRead);
}
dout.flush( );
dout.close( );
byte[] result = dout.getMessageDigest().digest( );
```

Рачунање резимеа за улазни ток је подједнако просто. Ипак, неопходно је бити бар маргинално упознат са методима класе `MessageDigest`.

Наравно, неопходно је придружити одређени резиме поруке одређеном току. У неким околностима, резиме се може слати преко истог канала који се користи за слање података који се резимирају. Пошиљалац рачуна резиме док шаље податке, док прималац рачуна резиме док их прима. Када пошиљалац заврши, шаље сигнал који прималац препознаје као индикатор краја тока и затим шаље резиме. Прималац прима резиме, проверава да ли је примљени резиме једнак ономе који је сам израчунао и затвара конекцију. Уколико се резимеи не подударе, прималац може тражити да му пошиљалац поново пошаље поруку.

Алтернативно, резиме и фајлови које резимира могу бити смештени у исту .zip архиву.
Постоје и многе друге могућности. Због тога је потребно дизајнирати пригодан формални протокол. Међутим, иако протокол може бити компликован, рачунање резимеа је праволинијско, захваљујући филтер класама `DigestInputStream` и `DigestOutputStream`.

Encrypting Streams

Класе `CipherInputStream` и `CipherOutputStream` из пакета `javax.crypto` омогућују сервисе криптовања и декриптовања. Објекат класе `Cipher` енкапсулира алгоритам који се користи за криптовање и декриптовање. Променом `Cipher` објекта мења се коришћени алгоритам. У већини случајева неопходан је тзв. кључ. Симетрични или алгоритми са тајним кључем користе исти кључ и за криптовање и за декриптовање. Асиметрични или алгоритми са јавним кључем користе различите кључеве. Кључ за криптовање може се дистрибуирати, док је кључ за декрипцију тајни. У Јави, кључеви се представљају инстанцама класа које имплементирају интерфејс `java.security.Key`. `Cipher` објекат се поставља конструктором.

```
public CipherInputStream(InputStream in, Cipher c)
public CipherOutputStream(OutputStream out, Cipher c)
```

Из законских разлога полисе су издвојене у стандардну Јава екстензију Java Cryptography Extension (JCE).

Како би `Cipher` објекат био исправно иницијализован, користи се статички метод `Cipher.getInstance()`. Овај `Cipher` објекат мора бити иницијализован, како за криптовање, тако и за декриптовање, методом `init()` пре прослеђивања неком од претходних конструктора. Нпр. следећи фрагмент кода припрема `CipherInputStream` за декрипцију коришћењем шифре "two and not a fnord" и алгоритма Data Encryption Standard (DES):

```
byte[] desKeyData = "two and not a fnord".getBytes( );
DESKeySpec desKeySpec = new DESKeySpec(desKeyData);
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
SecretKey desKey = keyFactory.generateSecret(desKeySpec);
Cipher des = Cipher.getInstance("DES");
des.init(Cipher.DECRYPT_MODE, desKey);
CipherInputStream cin = new CipherInputStream(fin, des);
```

Овај фрагмент користи класе из пакета `java.security`, `java.security.spec`, `javax.crypto` и `javax.crypto.spec`. Различите имплементације JCE подржавају различите групе алгоритама за криптовање. Уобичајени алгоритми су DES, RSA и Blowfish. Конструкција кључа зависи од алгоритма, па је потребно консултовати документацију одговарајуће JCE имплементације.

`CipherInputStream` предефинише већину уобичајених `InputStream` метода попут `read()` и `available()`. `CipherOutputStream` предефинише већину уобичајених `OutputStream` метода попут `write()` и `flush()`. Сви ови методи се позивају као и за било који други ток. Међутим, док се подаци читају или пишу, `Cipher` објекат тока декриптује или криптује податке. (Под претпоставком да Ваш програм ради са декриптованим подацима - што је обично случај - улазни ток декриптује податке, а излазни из криптује.) Нпр. следећи фрагмент кода криптује фајл secrets.txt користећи шифру "Mary had a little spider":

```
String infile = "secrets.txt";
String outfile = "secrets.des";
String password = "Mary had a little spider";

try {
      FileInputStream fin = new FileInputStream(infile);
      FileOutputStream fout = new FileOutputStream(outfile);

      // register the provider that implements the algorithm
      Provider sunJce = new com.sun.crypto.provider.SunJCE( );
      Security.addProvider(sunJce);

      // create a key
      char[] pbeKeyData = password.toCharArray( );
      PBEKeySpec pbeKeySpec = new PBEKeySpec(pbeKeyData);
      SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBEWithMD5AndDES");
      SecretKey pbeKey = keyFactory.generateSecret(pbeKeySpec);
```

```
      // use Data Encryption Standard
      Cipher pbe = Cipher.getInstance("PBEWithMD5AndDES");
      pbe.init(Cipher.ENCRYPT_MODE, pbeKey);
      CipherOutputStream cout = new CipherOutputStream(fout, pbe);

      byte[] input = new byte[64];
      while (true) {
            int bytesRead = fin.read(input);
            if (bytesRead == -1) break;
            cout.write(input, 0, bytesRead);
      }

      cout.flush( );
      cout.close( );
      fin.close( );
}
catch (Exception ex) {
      System.err.println(ex);
}
```