# 1　実験目的・課題

Socket 通信を行いクライアントと通信し、クライアントから入力された成績情報を記録するサーバプログラムを作成する。

作成するサーバーの要件

- サーバープログラムとクライアントプログラムをわけて作成
- 接続したクライアントは何度でもメッセージのやり取りが出来る
- 新しく成績情報と登録できる
- 登録された成績情報を記録できる
- 今までに登録された成績情報を閲覧できる
- 名前か番号で検索できる
- 入力にエラーがある場合は通知できる
- 名前と番号は重複登録はできない

# 2　実装方法

Java で複数のプログラム間でデータのやり取りを行うときは、Socket 通信というものを用いて通信を行う。

ここに、2 つのプログラムがあり、この 2 つで通信を行いたいとき、一方を「サーバー」もう一方を「クライアント」として話をする。サーバーとクライアントは最初、何のつながりも持っていない。ここで、サーバー側が ServerSocket クラスをインスタンス化し、その際に受け取ったポート番号の監視を始める。次に、クライアント側で Socket クラスにサーバー名とポート番号を渡しインスタンス化する際にサーバーに接続要求が送られる。サーバー側ではこの Socket による接続要求を accept メソッドによって受け取り、これで 2 つのプログラム間の接続が完了する。

## 2.1　全体の方針

今回の成績情報サーバーの作成では、サーバーとクライアントが交互に入出力を行うことにする。サーバーからクライアントにメッセージを送信、それを受け取ったクライアントがそのメッセージを標準出力に出力、それを見たユーザーがそれに対する応答をキーボードから入力する。その入力をクライアントよりサーバーに送信し、その入力を受けてサーバーで処理を行う。このまとまりを処理のひとまとまりとして扱い、これを繰り返すことで成績情報サーバーを作成する。この処理の例を図 1 に示す。

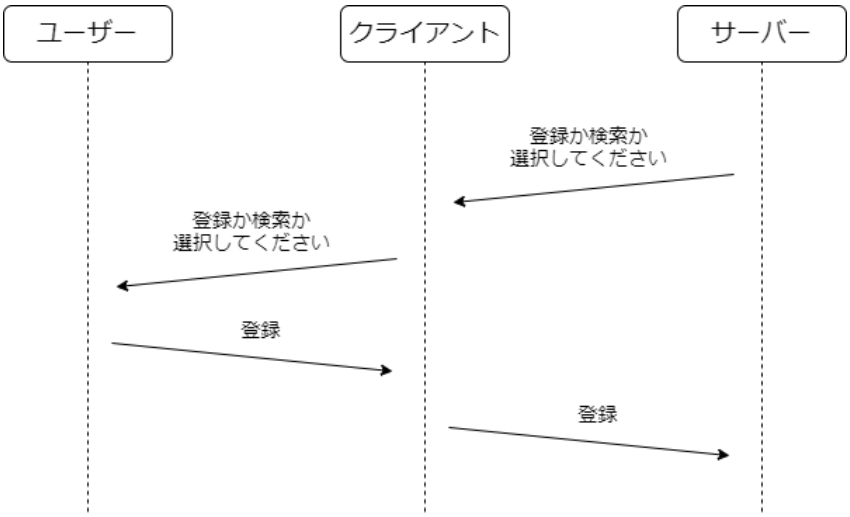


図 1　サーバーとクライアントの入出力例

クライアントプログラムでは、処理のひとまとまりを順に処理すればよいのでソケット入力、標準出力、標準入力、ソケット出力を順に 1 回だけ行うコードを while で囲い何度もループさせればよい。

サーバープログラムでは、サーバーの要件に従い、成績情報の登録、検索、閲覧ができるものを作成する。

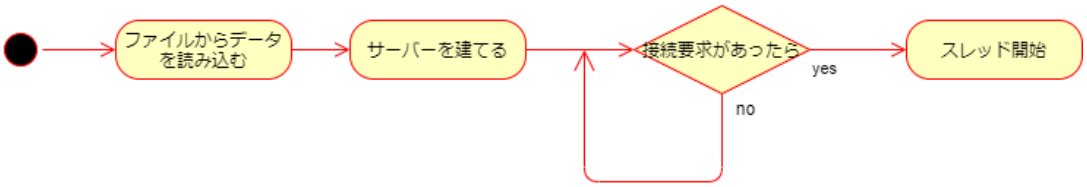


図 2　メイン関数のフローチャート

## 2.2　情報の記録

今回作成するプログラムは登録された情報を外部ファイルに書き込み保存し、次回サーバー起動時にそれを読み込み登録済み情報として保持することが出来るようにする。成績情報が書かれたファイルからの入力は「実験 4 ファイルからの入力と正規表現」で作成したプログラムのものを改変して使用する。出力ファイルへの書き込みはプログラム終了時に行い、Number,Name,Score をタブ区切りで出力する。

プログラム中で seiseki クラスのデータの保持には ArrayList ではなく TreeMap を使用する。TreeMap <K,V>は型 K のキーによって型 V の値にアクセスすることができるデータ構造である。型 K によって比較する赤黒木によって実装されており、要素数 N のとき、要素の検索、挿入、削除が log(n) 時間で行うことができるため、検索の面で ArrayList より優れていると判断し、これを採用した。

また、成績情報の入力において番号がバラバラに入力される可能性があるが、TreeMap でデータを保存しておけばプログラムの最後に出力するときにキーの昇順で成績情報を得られるので、データの閲覧時にわかりやすさが向上することが見込める。

## 2.3 登録

成績情報の登録には Number,Name,Score が必要である。今回はこれを一度に入力してもらうのではなく、それぞれについてクエリを投げてそれに回答してもらう形式にする。こうしなかった場合、例えば Number と Name の区切り文字をどうするかという事も決めなければならなくなるうえ、入力にミスが含まれる可能性が高くなると考えられる。こうして Number を表す文字列、Name を表す文字列、... を得られる。次にそれに区切り文字を付け一つの文字列になるよう連結する。このようにして得られた文字列が正規表現にマッチすれば入力を seiseki クラスに変換できる。

seiseki クラスに変換できたとしても、すぐに登録はできず、名前と番号が今までに登録されたものと重複していないか確認しなければならない。これは、名前、番号をそれぞれキーとした 2 つの TreeMap を用意しておいてそれぞれで containsKey メソッドで確認すればよい。

入力が正規表現にマッチし、今までと重複したものでもなければ新たに TreeMap に挿入する。

## 2.4 検索

名前で検索する場合は名前をキーとした TreeMap <String,seiseki>で検索を行い、キーとして名前が存在すればその名前で put すれば良い。

番号で検索する場合は番号をキーとした TreeMap <Integer,seiseki>で検索を行い、キーとして番号が存在すればその番号で put すれば良い。

## 2.5 閲覧

番号をキーとした TreeMap の values メソッドを使い、番号の昇順にならんだ seiseki クラスの集合を得る。それぞれに対して文字数が図 3 のようにフォーマットした文字列を送信する。

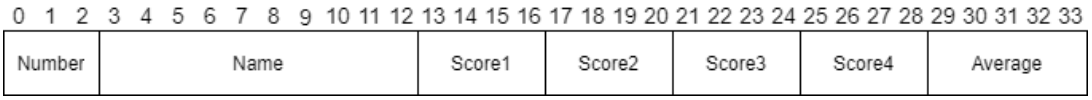


図 3　成績クラスの文字列フォーマット

この時だけ他の通信の時と違い、1 行だけでなくデータ数行分送信を行う。クライアント側では、通常 1 行受け取る処理しかしていないため、データ一覧を要求したときだけ if 文を使い複数行の入力に対応している。

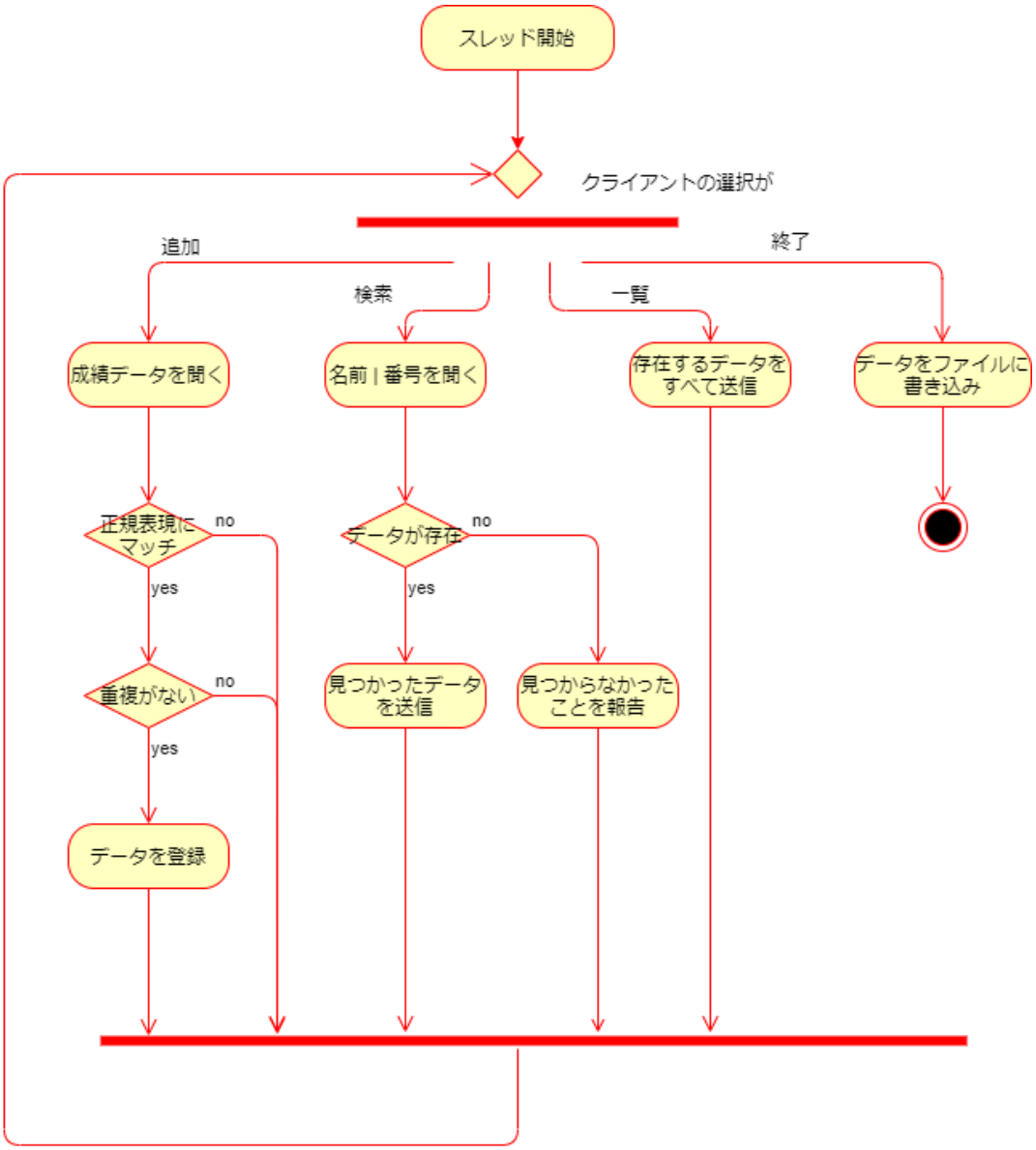


図 4　サーバーの処理

## 2.6　サーバーのマルチスレッド化

Java では Thread クラスの継承や Runnable インターフェースを実装することでプログラムを並列実行させることが出来る。ソースコード 1 を実行した結果を図 5,6 に示す。

ソースコード 1　マルチスレッドの例

```
1  public class Main extends Thread {
2    public static void main(String[] args){
3      Thread t1 = new Main();
4      Thread t2 = new Main();
```

```
5       t1.start();
6       t2.start();
7     }
8     public void run() {
9       for (int i = 0; i < 5; ++i)
10        System.out.println(getName() + " : " + i);
11    }
12  }
```

```
Thread-1 : 0
Thread-1 : 1
Thread-1 : 2
Thread-1 : 3
Thread-1 : 4
Thread-0 : 0
Thread-0 : 1
Thread-0 : 2
Thread-0 : 3
Thread-0 : 4
```

図 5　2 回目の実行結果

```
Thread-0 : 0
Thread-1 : 0
Thread-0 : 1
Thread-0 : 2
Thread-1 : 1
Thread-1 : 2
Thread-0 : 3
Thread-1 : 3
Thread-0 : 4
Thread-1 : 4
```

図 6　2 回目の実行結果

図 5 と 6 で実行結果が異なっていることがわかる。このようにして並列実行では同時にプログラムを実行することが出来る。ソースコード 1 の 5,6 行目の start メソッドが Thread の実行を表しており、オーバーライドされた run メソッドが実行される。run メソッドでは getName メソッドによりスレッド名を取得しそれとともに 0 から 4 までの数字を表示するだけのプログラムを実行させている。図 5,6 ともに Thread-0 だけ、Thread-1 だけ、で見てみると 0 から 1,2,3,4 の順に表示されていることから、run 内の処理は逐次実行されているとわかる。

これによって、サーバープログラムを並列実行させることで、クライアントが複数起動してもすべてに対応して情報をアップデートできるプログラムを作成する。

次に、変数の共有についてソースコード 2 に示すものを実行すると、”asdf1234”と表示される。本来これに期待される動作は”asdf”に”1234”を 10 回連接したものである。しかし、各スレッドで String 型の変数 s は共有されていないため、1 回だけ”1234”を連接したものが出力されてしまう。サーバープログラムではプログラムの動作中は成績情報を TreeMap によって管理しているため、複数のスレッドでそれぞれに追加された成績情報が共有されなければ重複登録が起こってしまう可能性が高いため、これは許容できない。

ソースコード 2　マルチスレッドの例 2

```
1   public class Main {
2     public static void main(String[] args) {
3       String s = "asdf";
4       hoge fuga = new hoge(s);
```

```
5       Thread t[] = new Thread[10];
6       for (int i = 0; i < 10; ++i) t[i] = new Thread(fuga);
7       for (int i = 0; i < 10; ++i) t[i].start();
8       for (int i = 0; i < 10; ++i)
9         try {
10          t[i].join();
11        } catch (InterruptedException e) {
12          e.printStackTrace();
13        }
14      System.out.println(fuga.getStr());
15    }
16  }
17  class hoge implements Runnable {
18    private String str;
19    public hoge(String s) {
20      this.str = s;
21    }
22    public void run() {
23      this.str += "1234";
24    }
25    public String getStr() {
26      return this.str;
27    }
28  }
```

これは、String がスレッドセーフなクラスではないためである。スレッドセーフとはあるプログラムを並列実行しても問題がないことである。スレッドセーフではないとは、あるスレッドで変更した共有データが他のスレッドによって上書きされてしまう可能性があることである。

ここで、Collections クラスの synchronizedSortedMap メソッドを使うことでスレッドセーフなソート・マップを使用することが出来る。今回はこれを使用してスレッドセーフなサーバークラスを作成した。

## 3 結果と考察

成績情報に表 1 の情報が出力ファイルに書き込まれているところから、”7 Mia 74 64 62 72”を追加するときのサーバーのログをソースコード 3 に示す。

| Number | Name | Score1 | Score2 | Score3 | Score4 |
|---|---|---|---|---|---|
| 1 | Michael | 75 | 86 | 89 | 31 |
| 2 | Emma | 74 | 63 | 70 | 48 |
| 3 | Hannah | 73 | 45 | 82 | 31 |
| 4 | Emily | 40 | 30 | 49 | 48 |
| 5 | Daniel | 46 | 59 | 47 | 70 |
| 6 | Oliver | 59 | 26 | 61 | 44 |

表 1　事前に書き込まれている成績情報

ソースコード 3　成績情報追加 1

```
1  Thread-0:send[select search,add,list : ]
2  Thread-0:recv[add]
3  Thread-0:send[Number : ]
4  Thread-0:recv[7]
5  Thread-0:send[ Name : ]
6  Thread-0:recv[Mia]
7  Thread-0:send[Score1 : ]
8  Thread-0:recv[74]
9  Thread-0:send[Score2 : ]
10 Thread-0:recv[64]
11 Thread-0:send[Score3 : ]
12 Thread-0:recv[62]
13 Thread-0:send[Score4 : ]
14 Thread-0:recv[72]
15 Thread-0:send[added data " 7 Mia 74 64 62 72 68.0" (enter to next)]
16 Thread-0:recv[]
```

ログ出力の 1 行は 3 つの部分からなり、最初に通信しているスレッド名、次に送信 (send) か、受信 (receive) か、最後に送受信された文字列を出力している。このときは 1 つのクライアントしか起動していないのでスレッド名は同じである。最初に定めた通信の手順に従い、送信と受信を交互に行っている。一番最初に送信した文字列によってデータの追加か検索か、一覧の出力を選択してもらい、それ以降は各クエリについての手順に従っている。成績の追加クエリでは、Number,Name,Score のそれぞれについて問いかけを行い回答を受け取り、seiseki クラスにパースしてデータを保存している。このときに登録された Mia の情報はプログラム内で保持されるのみで出力ファイルに出力はされていない。

次に、名前か番号が重複している入力、得点にエラーがある入力をしたときの出力をソースコード 4 に示す。

ソースコード 4　不正な成績追加入力

```
1  Thread-0:send[select search,add,list : ]
2  Thread-0:recv[add]
3  Thread-0:send[Number : ]
4  Thread-0:recv[8]
5  Thread-0:send[ Name : ]
6  Thread-0:recv[Mia]
```

```
Thread-0:send[Score1 : ]
Thread-0:recv[1]
Thread-0:send[Score2 : ]
Thread-0:recv[1]
Thread-0:send[Score3 : ]
Thread-0:recv[1]
Thread-0:send[Score4 : ]
Thread-0:recv[1]
Thread-0:send[duplication name (enter to next)]
Thread-0:recv[]
Thread-0:send[select search,add,list : ]
Thread-0:recv[add]
Thread-0:send[Number : ]
Thread-0:recv[2]
Thread-0:send[ Name : ]
Thread-0:recv[Luna]
Thread-0:send[Score1 : ]
Thread-0:recv[52]
Thread-0:send[Score2 : ]
Thread-0:recv[72]
Thread-0:send[Score3 : ]
Thread-0:recv[57]
Thread-0:send[Score4 : ]
Thread-0:recv[56]
Thread-0:send[duplication number (enter to next)]
Thread-0:recv[]
Thread-0:send[select search,add,list : ]
Thread-0:recv[add]
Thread-0:send[Number : ]
Thread-0:recv[8]
Thread-0:send[ Name : ]
Thread-0:recv[James]
Thread-0:send[Score1 : ]
Thread-0:recv[86]
Thread-0:send[Score2 : ]
Thread-0:recv[611]
Thread-0:send[Score3 : ]
Thread-0:recv[51]
Thread-0:send[Score4 : ]
Thread-0:recv[78]
Thread-0:send[invalid seiseki input (enter to next)]
Thread-0:recv[]
```

1 つ目は、”8 Mia 1 1 1 1”と入力している。これは Mia という名前が先に登録したものと同じため登録できない。次に、”2 Luna 52 72 57 56”と入力しているがこれは最初からファイルに入力してあった場号なので登録できない。最後に、”8 James 86 611 51 78”とい入力している、これは名前も番号も重複はしていないが

点数が 611 点の入力があるため正規表現にマッチせず、入力にエラーありと判断される。

次に、2 つ目のクライアントを起動して 2 つのクライアントで同じデータを追加しようとしたときのサーバーのログ出力をソースコード 5 に示す。

ソースコード 5　別のクライアントによる登録

```
1  Thread-0:send[select search,add,list : ]
2  new Thread
3  Thread-1:send[select search,add,list : ]
4  Thread-1:recv[add]
5  Thread-1:send[Number : ]
6  Thread-1:recv[10]
7  Thread-1:send[ Name : ]
8  Thread-1:recv[Luna]
9  Thread-1:send[Score1 : ]
10 Thread-1:recv[52]
11 Thread-1:send[Score2 : ]
12 Thread-1:recv[72]
13 Thread-1:send[Score3 : ]
14 Thread-1:recv[57]
15 Thread-1:send[Score4 : ]
16 Thread-1:recv[56]
17 Thread-1:send[added data " 10 Luna 52 72 57 56 59.3" (enter to next)]
18 Thread-1:recv[]
19 Thread-1:send[select search,add,list : ]
20 Thread-0:recv[add]
21 Thread-0:send[Number : ]
22 Thread-0:recv[10]
23 Thread-0:send[ Name : ]
24 Thread-0:recv[Luna]
25 Thread-0:send[Score1 : ]
26 Thread-0:recv[52]
27 Thread-0:send[Score2 : ]
28 Thread-0:recv[72]
29 Thread-0:send[Score3 : ]
30 Thread-0:recv[57]
31 Thread-0:send[Score4 : ]
32 Thread-0:recv[56]
33 Thread-0:send[duplication name (enter to next)]
34 Thread-0:recv[]
```

1 行目のセレクトクエリがスレッド 0 によって送られているがそれに接続しているクライアントが何らかの入力をするまでは何もできないため待機している。2 行目の”new thread”というのはメイン関数内で新たにポートに接続要求をしてきたクライアントがいたため新しく server.accept をし、別のソケットで新しいスレッドを開始したことを通知するものである。新しく開始したスレッド 1 によって”10 Luna 52 72 57 56”を追加し、その後、同じデータをスレッド 0 によって追加しようとしている。スレッド 0 によって追加しようと

したときは、データの重複のため、登録できていない。このことから、複数のスレッドで成績データを管理している TreeMap が同期されていることがわかる。

次に、検索クエリでの動作をソースコード 6 に示す。このときの成績データを表 2 に示す。

Number Name Score1 Score2 Score3 Score4 1 Michael 75 86 89 31 2 Emma 74 63 70 48 3 Hannah 73 45 82 31 4 Emily

表 2　検索をするときのデータ

ソースコード 6　検索クエリ

```
1  Thread-0:send[select search,add,list : ]
2  Thread-0:recv[search]
3  Thread-0:send[select Number,Name : ]
4  Thread-0:recv[name]
5  Thread-0:send[enter name : ]
6  Thread-0:recv[Mason]
7  Thread-0:send[not found (enter to next)]
8  Thread-0:recv[]
9  Thread-0:send[select search,add,list : ]
10 Thread-0:recv[search]
11 Thread-0:send[select Number,Name : ]
12 Thread-0:recv[name]
13 Thread-0:send[enter name : ]
14 Thread-0:recv[Luna]
15 Thread-0:send[ 10 Luna 52 72 57 56 59.3 (enter to next)]
16 Thread-0:recv[]
17 Thread-0:send[select search,add,list : ]
18 Thread-0:recv[search]
19 Thread-0:send[select Number,Name : ]
20 Thread-0:recv[number]
21 Thread-0:send[enter number : ]
22 Thread-0:recv[111]
23 Thread-0:send[not found (enter to next)]
24 Thread-0:recv[]
25 Thread-0:send[select search,add,list : ]
26 Thread-0:recv[search]
27 Thread-0:send[select Number,Name : ]
28 Thread-0:recv[number]
29 Thread-0:send[enter number : ]
30 Thread-0:recv[1]
31 Thread-0:send[ 1 Michael 75 86 89 31 70.3 (enter to next)]
32 Thread-0:recv[]
```

存在しない番号、名前で検索しようとしたときは”not found”となっていて、存在する番号、名前で検索したときは、Number,Name,Score,Acerage が表示されていることがわかる。

成績のデータを追加するときの重複判定、実際に追加すること、検索の時の存在判定はいずれもデータ数を N として $\Theta(logN)$ で実現されている。もし、TreeMap ではなく ArrayList でデータを管理していると、データの検索に $O(N)$ かかり、追加時には重複判定が必要なため $O(N) + O(1) = O(N)$ かかってしまうため、データ数が数万件になると数秒では処理できなくなってしまうと考えられる。

次に、通信を終了するときの処理をソースコード 7 に示す。

ソースコード 7　通信終了の処理

```
1  Thread-0:send[select search,add,list : ]
2  Thread-0:recv[end]
3  file output [ 1 Michael 75 86 89 31 70.3]
4  file output [ 2 Emma 74 63 70 48 63.8]
5  file output [ 3 Hannah 73 45 82 31 57.8]
6  file output [ 4 Emily 40 30 49 48 41.8]
7  file output [ 5 Daniel 46 59 47 70 55.5]
8  file output [ 6 Oliver 59 26 61 44 47.5]
9  file output [ 7 Mia 74 64 62 72 68.0]
10 file output [ 10 Luna 52 72 57 56 59.3]
```

通信を終了するときは、どんなタイミングでもクライアント側で”end”という文字列を打ち込むことでサーバー側に通信終了を通告し応答を待たずソケットを閉じる処理にしている。サーバー側ではクライアントから”end”という文字列が送られてきた場合そのスレッドの全ての処理を中断し、成績のデータを出力ファイルに書き込み、スレッドを終了する。

最後に、スレッド 1 で番号 8 の成績を追加して終了したときのログをソースコード 8 に示す。

ソースコード 8　番号がバラバラに追加されるときの処理

```
1  Thread-1:recv[list]
2  Thread-1:send[ 1 Michael 75 86 89 31 70.3]
3  Thread-1:send[ 2 Emma 74 63 70 48 63.8]
4  Thread-1:send[ 3 Hannah 73 45 82 31 57.8]
5  Thread-1:send[ 4 Emily 40 30 49 48 41.8]
6  Thread-1:send[ 5 Daniel 46 59 47 70 55.5]
7  Thread-1:send[ 6 Oliver 59 26 61 44 47.5]
8  Thread-1:send[ 7 Mia 74 64 62 72 68.0]
9  Thread-1:send[ 10 Luna 52 72 57 56 59.3]
10 Thread-1:send[select search,add,list : ]
11 Thread-1:recv[add]
12 Thread-1:send[Number : ]
13 Thread-1:recv[8]
14 Thread-1:send[ Name : ]
15 Thread-1:recv[James]
16 Thread-1:send[Score1 : ]
17 Thread-1:recv[86]
18 Thread-1:send[Score2 : ]
19 Thread-1:recv[61]
20 Thread-1:send[Score3 : ]
```

```
21  Thread-1:recv[51]
22  Thread-1:send[Score4 : ]
23  Thread-1:recv[78]
24  Thread-1:send[added data " 8 James 86 61 51 78 69.0" (enter to next)]
25  Thread-1:recv[]
26  Thread-1:send[select search,add,list : ]
27  Thread-1:recv[list]
28  Thread-1:send[ 1 Michael 75 86 89 31 70.3]
29  Thread-1:send[ 2 Emma 74 63 70 48 63.8]
30  Thread-1:send[ 3 Hannah 73 45 82 31 57.8]
31  Thread-1:send[ 4 Emily 40 30 49 48 41.8]
32  Thread-1:send[ 5 Daniel 46 59 47 70 55.5]
33  Thread-1:send[ 6 Oliver 59 26 61 44 47.5]
34  Thread-1:send[ 7 Mia 74 64 62 72 68.0]
35  Thread-1:send[ 8 James 86 61 51 78 69.0]
36  Thread-1:send[ 10 Luna 52 72 57 56 59.3]
37  Thread-1:send[select search,add,list : ]
38  Thread-1:recv[end]
39  file output [ 1 Michael 75 86 89 31 70.3]
40  file output [ 2 Emma 74 63 70 48 63.8]
41  file output [ 3 Hannah 73 45 82 31 57.8]
42  file output [ 4 Emily 40 30 49 48 41.8]
43  file output [ 5 Daniel 46 59 47 70 55.5]
44  file output [ 6 Oliver 59 26 61 44 47.5]
45  file output [ 7 Mia 74 64 62 72 68.0]
46  file output [ 8 James 86 61 51 78 69.0]
47  file output [ 10 Luna 52 72 57 56 59.3]
```

リストクエリによって表示されたデータは 1,2,3,4,5,6,7,10 の順になっている。ここに”8 James 86 61 51 78”を追加すると、例えば ArrayList によってデータを管理していれば 1,2,3,4,5,6,7,10,8 の順序で表示されるが、TreeMap で番号をキーとして管理していればソースコード 8 のように、7 と 10 の間に 8 が挿入される。

これによって例えば生徒がそれぞれ自分の成績を登録したとしてもそれを見るときには番号でソートされており誰が登録していないかなどが一目でわかるようになる。

## 参考文献

[1] Java プログラムにおける通信のしくみを理解する
https://crew-lab.sfc.keio.ac.jp/lectures/2000s_mmb/JavaLectures/Lecture8/Lec8-1.html