JAVAFO PAIRING ENGINE ADVANCED USER MANUAL

(note: red is used to highlight changes from the previous version)

Introduction

JaVaFo is both a stand-alone program (provided that a java virtual machine exist to execute it) and an archive (.jar) that can be used by a program written in the java programming language.

This manual describes both possibilities.

JaVaFo as a stand-alone program

In the following chapters, it will be described how it is possible for an external program to integrate the JaVaFo Pairing Engine in order to use it to prepare pairings according to the FIDE (Dutch) Swiss System *(see section C.04.3 from FIDE Handbook)*.

How to input the data

Albeit JaVaFo supports many input formats (and other ones could be easily added), the most practical way to input data to JaVaFo is using the TRF(x), where **TRF** *(also called TRF16 to distinguish it from the TRF06 used in the past)*, is the official FIDE Tournament Report File defined in the FIDE handbook *(see)*, and the **(x)** stands for some extensions that have to be introduced in such format to make it useful for exchanging data between different programs. To be clearly understood by JaVaFo, it is recommended that such file is written using the UTF-8 encoding.

The extensions to the TRF16 are partly made by adding some new *codes (which are alphabetic in order to be completely different by the current numeric codes defined for the format*[1]*)*, partly by allowing writing in the TRF something that is not normally found in a TRF, partly by interpreting some data contained in it.

Interpretation extensions

A TRF is normally used only to generate data at the end of the tournament. The first extension is to allow the TRF to be generated also during the tournament. This partial TRF is fed to JaVaFo as its input.

It is important to note that the field called **Points** in the TRF16 definition *(position 81-84)* has to contain the correct number of points that each player got, because that number is tentatively used to infer the scoring point system used (i.e. the classic 1, ½, 0 or another one, like 3, 1, 0; or even weirder ones - for the complete list of values that JaVaFo tries to infer, see the ***scoring-point parameters*** in the section *Reducing randomness by way of a (RTG) configuration file - scoring-point parameters*).

However, with version 2.2, the above methodology of computing the scoring-point parameters has been deprecated. Such parameters, if different by the standard ones, have to be explicitly defined by using the option XXS described in the chapter dedicated to the *Extensions and other options* below *(see Scoring point system)*.

Unusual info extensions

The partial TRF contains information regarding the rounds that have already been played. However it says nothing regarding the current round (i.e. the one that should be paired). The most important thing is to tell JaVaFo which players should be paired or, which is the same, which players will not play that round.

If everybody plays in the current round, the partial TRF is enough. Otherwise, if somebody is missing, there are two not-contrasting ways to pass this piece of information to the pairing engine.

The first one consists in inserting into the proper columns for the current round the result code that is normally used for absent players, i.e. `"0000 - Z"`.

For the sake of clarity, it is also possible to use the codes that will be present in the final TRF16 *(i.e. `"0000 - H"`, for half-point-byes or `"0000 - F"` for the deprecated full-point-byes)*, but, in these cases, also the field **Points** *(position 81-84, see above)* must be updated.

Blank codes are obviously ignored in this instance, as each present player is identified by the absence of a result code.

Although this methodology has been not deprecated yet, in order to tell the pairing engine which players are not to be paired in a round, it is preferable to use the option XXZ described in the chapter dedicated to the *Extensions and other options* below *(see Absent players)*.

Extra codes extensions

As said in the introduction regarding input data, some alphabetic codes were added in order to transmit additional information to the pairing engine. Some of them are not essential (and will be shown in a following chapter), but one is: the pairing engine must know the total number of rounds in the tournament in order to know if the current round is the last one.

This is obtained by means of the following new alphabetic code:

`XXR` ***`number`***

where *number* is the number of rounds of the competition.

TRF(x) sample

1 The idea of alphabetic codes comes from the author of the TRF, Christian Krause, chairman of the Systems of Pairings and Programs (SPP) FIDE Commission

In the linked file, there is an example of the TRF(x) used to pair the fifth round of the **XX Open Internacional de Gros** which includes everything said above regarding the TRF extensions.

TRF(x) Sample

According to what was discussed before, it should be pretty evident that the players 22, 26 and 43 will not play the fifth round *(look at the field* **Points** *for the player 43, though)*.

How to invoke JaVaFo

Although, the title of this chapter is "*JaVaFo as a stand-alone program*", this terminology is highly simplicistic. In reality, JaVaFo is an executable java archive (jar), and here is described how to invoke a similar element.

First of all, a Java Virtual Machine (JVM) is needed. The beauty of this is that the operating system (O.S.) in use is not important. JaVaFo can run on any O.S. provided that a Java Virtual Machine exist in it. Be **java** the command to activate it.

Then the pairing engine itself, javafo.jar, is needed. The latest version of the jar can always be downloaded from the following web-site:

http://www.rrweb.org/javafo/current/javafo.jar

You can also find previous versions of javafo.jar replacing current with 1.0, 1.1, 1.2 and so on. Be aware, though, that the releases until 1.4 do not follow the specification of this manual, but are tied to ancient versions of the Dutch Rules.

The downloaded jar archive can be stored anywhere in the file system. Be JVF_DIR the pathname of the folder where javafo.jar has been downloaded.

The first test to see if both the JVM and the pairing engine work is to write in a command prompt:

java -ea -jar JVF_DIR\javafo.jar (or)

java -ea -jar JVF_DIR/javafo.jar

depending on the O.S. in use. The first line works for Windows. From here on, only the Windows commands are mentioned. Moreover, for the full command **java -ea -jar JVF_DIR/javafo.jar**, the string **javafo** is used, as if a file named ***javafo.bat*** exists somewhere in a PATH directory and contains the statement:

```
@java -ea -jar JVF_DIR/javafo.jar %*
```

If everything is properly set up, the above command should produce the following output (or something similar, more up-to-date):

```
JaVaFo (rrweb.org/javafo) - Rel. 2.2 (Build 3222)
```

After checking that the pairing engine is ready, a TRF16 file can be input to it *(a TRF06 will still work, but the output is not completely trustful)*. Even this file can be placed anywhere in the file system. Be TRF_DIR the pathname of the folder where the file **trn.trfx** is located *(trn.trfx is just a mnemonic name; dummy.foo is an equally valid name)*.

It should be also decided where the pairing engine output goes. Be **OUT_DIR\outfile.txt** the pathname of said output file (as above, this file can be called in any way). Be aware that no warning is issued if the file pre-exists before invoking javafo.jar.

From the same command prompt mentioned above, the following command line is needed to produce the pairings for the current round:

javafo TRF_DIR\trn.trfx -p OUT_DIR\outfile.txt

The meaning of the most useful options will be described in a following chapter. But the above command is the most generic way to invoke the javafo.jar pairing engine. If OUT_DIR\ is the same than TRF_DIR\, then OUT_DIR can be dropped, so that

javafo TRF_DIR\trn.trfx -p outfile.txt

generates outfile.txt in TRF_DIR. If TRF_DIR is the current directory (i.e. "."), TRF_DIR\ can be dropped.

How to read the output of JaVaFo

When invoked as shown in the previous command, the output is the file outfile.txt. The structure of this file is very simple:

[1] the first line reports the number of generated pairs (be P)

[2] from the second to the (P+1)-th line, each line contains a pair for the current round. Such pair is made using the pairing-id(s) of the players, i.e. the id(s) that are defined in the **001** line of the TRF(x). The first element of the pair gets white, the second one black. If the number of players to be paired in the round is odd, one of the pairs is formed by the id of the player that gets the pairing-allocated-bye, followed by 0.

The TRF sample shown above will produce the following output:

```
25
1 2
3 4
5 6
7 13
11 21
23 12
19 16
17 52
35 18
8 24
9 26
37 10
14 29
45 15
46 20
27 38
34 30
39 31
41 32
42 33
44 48
25 49
40 50
51 36
47 0
```

If something goes wrong the output file is not generated. Something is usually displayed on standard output or standard error, but it may be quite difficult to interpret.

The most common cause for an error is a malformed TRF(x). However, if the TRF(x) is totally correct, than an error must have occurred in the pairing engine. This may occasionally happen, but it is a very unlikely event (although not an impossible one, of course).

Extensions and other options

What was presented in the previous chapters covers the majority of the situations that can happen in a tournament. Sometimes, however, something may happen that requires some extra care.

Absent players

In *Unusual info extensions*, it has already been shown a way to register the players that are not going to be paired in a round. An alternative way is to use the extension code XXZ, the format of which is:

XXZ *list-of-pairing-id(s)*

where the pairing-id(s) following XXZ are intuitively the ones of the players that will miss the round to be paired.

There can be multiple XXZ records.

Scoring point system

In *Unusual info extensions*, it has already been shown a deprecated way to compute the scoring-point parameters. The securer way is to use the extension code XXS, the format of which is:

XXS *CODE*=*VALUE*

where *VALUE* is floatingpoint-number *(e.g. 1.5)*, and *CODE* is one of the following codes:

code	*default value*	description
WW	*1.0*	points for win with White
BW	*1.0*	points for win with Black
WD	*0.5*	points for draw with White
BD	*0.5*	points for draw with Black
WL	*0.0*	points for loss with White
BL	*0.0*	points for loss with Black
ZPB	*0.0*	points for zero-point-bye
HPB	*0.5*	points for half-point-bye
FPB	*1.0*	points for full-point-bye
PAB	*1.0*	points for pairing-allocated-bye
FW	*1.0*	points for forfeit win
FL	*0.0*	points for forfeit loss
W	*1.0*	encompasses all the codes WW, BW, FW, FPB
D	*0.5*	encompasses all the codes WD, BD, HPB
L	*0.0*	encompasses all the codes WL, BL

The sequence *CODE*=*VALUE* can be repeated multiple times in a XXS record, or there may be multiple XXS records. Codes that are not mentioned in the XXS records are assumed to have the standard value. Hence, when the standard scoring point system is used, there is no need to put XXS codes in the input TRF(x).

When a shortcut and a code encompassed by such shortcut are both used, the order (left-right, top-down) is decisive, because the latter element overrides the former.

Note that when XXS is used, javafo makes a strict check that there is an absolute correspondence between the field **Points** *(characters 81-84, in the 001 record of the TRF(x))* and the results. If the check fails, the program may crash.

Examples

1. *Although not needed, the standard scoring system may be described by:*

```
XXS WW=1 WD=.5 WL=0 BW=1 BD=0.5 BL=0
XXS FL=0 FW=1
XXS PAB=1 FPB=1 HPB=.5 ZPB=0
```

   or by:

```
XXS W=1 D=0.5 L=0 FL=0 ZPB=0 PAB=1
```

2. *The sequence:*

```
XXS PAB=3 D=1 W=3
```

   is enough to describe the 3/1/0 scoring point system (with the PAB equal to a win), and

```
XXS W=3
XXS D=1
```

is enough to describe the 3/1/0 scoring point system with the PAB equal to a draw (because the original PAB=1 is assumed).

3.
```
XXS PAB=.5
```

is enough, when a half-point PAB is requested and anything else is standard.

4. *The 3/2/1/0 score system, which is often talked about, can be represented by:*

```
XXS FL=0 W=3 D=2 PAB=2 L=1 ZPB=1
```

(note: this is __one__ possibility - FL=0 may be omitted, PAB may be equal to 3, and ZPB may be equal to 0, and there also other possibilities).

5. *As an example of overriding:*

```
XXS W=3 D=2 L=1
XXS PAB=3
XXS FPB=0 HPB=0 ZPB=0
```

is a 3/2/1/0 scoring system where all missed rounds net zero points, but writing:

```
XXS PAB=3
XXS FPB=0 HPB=0 ZPB=0
XXS W=3 D=2 L=1
```

will not reach the same goal because W=3 overrides FPB=0 (and D=2 overrides HPB=0).

Ranking id

JaVaFo identifies players with two numbers, the player-id and the positional-id.

The first one is obvious and it is the one that is associated with the **001** record in the TRF(x). The second one is implicitly given by the position of the players in the TRF(x). Albeit this is not mandatory, players are normally inserted into the TRF(x) in accordance with their pairing-id, so that the two sets of data (pairing-id(s) and positional-id(s)) are basically coincident.

They may differ, though, and sometimes for good reasons. For instance, please give a look at the linked file, which is a modified version of the previous sample: the players with a local rating (0 for FIDE) are placed at the end of the list.

Ranked TRF(x) Sample

The positional-id is 1 for Mirzoev, 6 for Lakunza (player-id: 7), 29 for Aizpurua (player-id: 32), 42 for Abalia (player-id: 42), 48 for Gorrochategui (player-id: 6) and so on.

JaVaFo computes the pairings using the pairing-id(s). Beware: JaVaFo uses the pairing-id(s), not the ratings, as specified by the FIDE (Dutch) rule A.2.b. This is a programming choice that cannot be modified.

However, the calling program is not forced to follow the same logics. If it desires that the rating be prevalent, it has two alternatives:

[a] before calling JaVaFo, redefine the pairing-id(s) in such a way that increasing pairing-id(s) are assigned to players with decreasing rating (which, by the way, is the most standard situation)

[b] insert players in the TRF(x) in order of rating and tell JaVaFo to use the positional-id(s) (also called ranking-id(s)) instead of the pairing-id(s)

The first choice is the recommended one. However, in order to let the calling program use the second alternative, JaVaFo provides the extension code:

`XXC rank`

The 'C' in XXC stands for configuration. The word **rank** tells JaVaFo to use the positional-id(s) in order to produce the pairings. The output file still contains the pairing-id(s).

If the "XXC rank" clause is added to the file shown in the Ranked TRF(x) Sample, JaVaFo will return the following output:

```
25
1 2
3 4
5 6
7 13
11 21
23 12
19 16
17 52
35 18
8 24
9 26
45 10
14 29
37 15
46 20
27 31
39 32
44 33
34 30
41 38
42 48
25 50
40 49
51 36
47 0
```

which is different by the previous one (differences highlighted in pink).

First round pairing

Although the rule for pairing the first round is very simple and therefore the calling program can generate it directly, it is recommended to use JaVaFo also to generate the first round.

The only information to pass to the pairing engine is the initial-colour, as defined in the section E of the FIDE (Dutch) Rules. There are three possibilities:

[a] white

[b] black

[c] let JaVaFo make the choice (i.e. random)

The latter choice is the default. Beware that it is a semi-random choice: to compute it, JaVaFo uses the hash of some data taken from the TRF(x); this means that repeating the process with the same TRF(x) will give the same result each time. Therefore, in order to use the JaVaFo random choice, the calling program needs doing nothing.

Otherwise, to force the choice [a], the TRF(x) must contain a line

`XXC white1`

To force the choice [b], the TRF(x) must contain a line

`XXC black1`

Please note that the XXC code is cumulative, so it is followed by all the configuration choices made by the calling program. For instance:

`XXC rank black1`

is a valid extension line and combines what was described in the previous and in the current chapters.

Accelerated rounds

The standard way to have accelerated rounds is to assign fictitious points to some players. How to assign such points depends on various methods and only one of them has been codified in the old FIDE Handbook *(see)*. Hence, this is not a matter of discussion here *(on the other hand, see Baku Acceleration Method, below)*.

However, JaVaFo can be informed of the fictitious points that are assigned to each player, using the extension code XXA.

The format for this code is

```
XXA NNNN pp.p pp.p ...
```

Where:

= **XXA** starts at column 1

= **NNNN** (player's id - same as in 001) starts at column 5

= **pp.p** (fictitious points) starts at column 10+5*(r-1), where **r** is the round in which the fictitious points must be added

It is mandatory to keep the full record of the fictitious points assigned round by round, because this record is used to determine the floaters history of each player (actually, if pairing for round X, it is enough to maintain the fictitious points history from the rounds from X-3 to the current one - but it seems simpler, also from a pairing-checking standpoint, to keep the full history).

Here is an example from a real tournament where seven accelerated rounds were used:

TRF(x) Acceleration Sample

Baku Acceleration Method

Upon request, JaVaFo can pair the current round by applying the Baku Acceleration Method *(see C.04.5.1 in the old FIDE Handbook)*. In order to do so, JaVaFo should be invoked using the option **-b** *(note: **old** FIDE Handbook, i.e. works only for tournaments longer than eight rounds)*.

Beware that JaVaFo applies the Baku methodology only to the current round, not to the already paired rounds. JaVaFo relies on the input TRF(x) to correctly convey the information (basically the fictitious points) related to the previous rounds.

Forbidden pairs

Sometimes some players are to be prevented from meeting each other. JaVaFo can be directed to fulfill this need by means of the extension code XXP.

The format of this code is:

```
XXP  list-of-pairing-id(s)
```

All the players mentioned in the list will not be paired against each other.

There is no limit on how many times a player can be part of a XXP list. So, for instance, if games between members of two groups of players cannot happen (for instance, assume that <13, 78, 102> and <68, 111> be these two groups), the following list of XXP extension codes should be generated:

```
XXP 13 68
XXP 13 111
XXP 78 68
XXP 78 111
XXP 102 68
XXP 102 111
```

Check-list

Upon request, JaVaFo can generate a check-list, i.e. a file that summarizes the situation after the pairing, with the following contents *(taken from the previous TRF(x) sample)*:

Check-List Sample

In the check-list, the majority of the fields are pretty intuitive. Some more explanation may be needed for the columns **Pref, -1R**, **-2R** *and all the columns* **G-**n.

Pref *reports the preference of the players. The following table explains the symbols and the associated preference:*

WWW	**BBB**	Double absolute preference *(see A.6.a)*; both conditions are met: twice-same-colour and colour-difference higher than |1|)
WW	**BB**	Absolute preference for colour-difference higher than |1| *(see A.6.a)*
W1	**B1**	Absolute preference for twice-same-colour and colour-difference equal to 1 *(see A.6.a)*
W	**B**	Absolute preference for twice-same-colour and colour-difference equal to 0 *(see A.6.a)*
(W)	**(B)**	Strong colour preference *(see A.6.b)*
(w)	**(b)**	Mild colour preference *(see A.6.c)*
A		No preference *(see A.6.d)*

-1R *and* **-2R** *are references to the floating history of the players. They show the kind of float (up or down) respectively in the last and in the penultimate round.*

The columns **G-***r*, …, **G-1** *represent the opponents of a player in the last* **g**-th *game he played (unplayed rounds are not considered). The presence of a* **[X]** *in the first of such colums means that the player cannot receive the pairing-allocated-bye.*

In order to produce the check-list, JaVaFo should be invoked using the option **-l,** optionally followed by the name of the file which to put the check-list in. If such name is missing, JaVaFo will produce the file **trn.list**, provided that -l follow the input file name.

For instance, this works:

```
javafo TRF_DIR\trn.trfx -p OUT_DIR\outfile.txt -l
```

and this works too:

```
javafo TRF_DIR\trn.trfx -p OUT_DIR\outfile.txt -l ANY_DIR\outfile.list
```

As usual, if ANY_DIR is omitted, outfile.list is produced in the TRF_DIR.

Release and build numbers

As mentioned above, the simple command **javafo** will print the release version and build on the standard output.

If an input filename is specified, this information is not output unless the **-r** option is used.

Pairings Checker

JaVaFo can also be used to check the correctness of a TRF produced by other software: the command line:

```
javafo TRF_DIR\trn.trfx -c
```

will produce on standard output something similar to what is shown below (related to the tournament described by the TRF(x) Acceleration Sample), with obvious meaning:

```
AcceleratedTRFXSample2: Round #1
```

```
AcceleratedTRFXSample2: Round #2
AcceleratedTRFXSample2: Round #3
AcceleratedTRFXSample2: Round #4
AcceleratedTRFXSample2: Round #5
AcceleratedTRFXSample2: Round #6
  Checker pairings        Tournament pairings
     55 -  73                 55 -  71
     61 -  71                 74 -  73
     74 -  75                 61 -  83
     80 -  83                 80 -  75

AcceleratedTRFXSample2: Round #7
AcceleratedTRFXSample2: Round #8
AcceleratedTRFXSample2: Round #9
  Checker pairings        Tournament pairings
     76 -  78                 76 -  75
     74 -  75                 61 -  78
     80 -  61                 80 -  74
```

In order to check a single round (and not the whole tournament), add the number of the round to the aforementioned command as in:

javafo TRF_DIR\trn.trfx -c 6

The output of this command for the previous sample is shown below:

```
AcceleratedTRFXSample2: Round #6
  Checker pairings        Tournament pairings
     55 -  73                 55 -  71
     61 -  71                 74 -  73
     74 -  75                 61 -  83
     80 -  83                 80 -  75
```

Random Tournament Generator (RTG)

In order to help an external pairing-checker, JaVaFo can generate random or quasi-random tournaments against which the external pairing-checker can be tested.

The command line (in its simplest form):

javafo -g -o trn.trf

will generate in the current folder (any pathname like ***TEST_DIR\trn.trf*** can be specified, though) a file **trn.trf**, which is a TRF16 of a random tournament, with a random number of players (usually between 15 and 415), a random number of rounds (usually between 5 and 17) and game results that depend on the rating difference between the involved players, applying a formula elaborated by Otto Milvang *(see here, Appendix A, pag. 8)* that, in the long run, will distribute points based on what actually happens in rated tournaments.

In the same command line, the option **-b** *(apply the Baku Acceleration Method)* can be added.

The most important utilization of this feature is to generate thousands of tournaments and then test them with the appropriate checker. Therefore it is advisable to use (on Windows) a statement like that:

@for /L %p IN (1000,1,5999) do @javafo -g -o test%p.trf

which will generate exactly 5000 random tournaments in the current folder.

The previously mentioned randomness in the generated files can be reduced in two **alternative** ways, using either a (RTG) configuration file or a model TRF.

Reducing randomness by way of a (RTG) configuration file

A (RTG) configuration file is a property-file[2] where the following parameters (properties) may be defined *(for the explanation of each single parameter, please look at the sample below)*:

ParameterName	Default (when the parameter is not defined)
PlayersNumber	*A random number between 15 and 415*
RoundsNumber	*A random number between 5 and 17*
ForfeitRate	*A random number between 10 and 100*
QuickgameRate	*A random number between 0 and 20*
ZPBRate	*A random number between 0 and 20*
HPBRate	*A random number between 5 and 45*
FPBRate	*A random number between 0 and 2*
HighestRating	*A random number between 2400 and 2800*
LowestRating	*A random number between 1400 and 2300*
Groups	*A random number between 0 and 10*
Separator	*A random number between 20 and 70*

The following twelve parameters are also called**scoring-point parameters**
*(they are the values that JaVaFo tries to infer using the field***Points***(position 81-84) of TRF16)*

WWPoints	*1.0*	points for win with White
BWPoints	*1.0*	points for win with Black
WDPoints	*0.5*	points for draw with White
BDPoints	*0.5*	points for draw with Black
WLPoints	*0.0*	points for loss with White
BLPoints	*0.0*	points for loss with Black
ZPBPoints	*0.0*	points for zero-point-bye
HPBPoints	*0.5*	points for half-point-bye
FPBPoints	*1.0*	points for full-point-bye
PABPoints	*1.0*	points for pairing-allocated-bye
FWPoints	*1.0*	points for forfeit win
FLPoints	*0.0*	points for forfeit loss

An example of a (RTG) configuration file (with comments) is shown here:

Random Tournament Generator Configuration Sample

In order to being used by the JaVaFo Random Tournament Generator, the (RTG) configuration file must be specified as a parameter to the **-g** option. Therefore, the full command line is:

```
javafo -g RTG_DIR\rtg.cfg -o TEST_DIR\trn.trf          (or)

javafo -g RTG_DIR\rtg.cfg -b -o TEST_DIR\trn.trf
```

if there is the desire to apply, when feasible, the Baku Acceleration Method.

Reducing randomness by way of a model tournament

A model tournament is a normal input TRF file with meaningful player ratings, which serves as a model in order to define all the parameters mentioned above.

For instance, from the input file seen in the TRF(x) Acceleration Sample, the following values are automatically retrieved:

2 A property-file is a file where empty lines have no meaning and lines introduced by the symbol # contain a comment. The meaningful lines have the format ***PropertyName=PropertyValue***, which assigns to the named property (or parameter) the value specified by *PropertyValue*.

ParameterName	Value
PlayersNumber	84
RoundsNumber	9
ForfeitRate	2
QuickgameRate	0
ZPBRate	30
HPBRate	0
FPBRate	0

Note that the ratings parameters *HighestRating, LowestRating, Groups* and *Separator* are not considered, as the ratings are exactly the same as the ones present in the input model file, while the scoring-points parameters are not shown as their value is equal to the corresponding default value.

As the model file is a standard input file, the command line to use it is:

javafo MODEL_DIR\model.trf -g -o TEST_DIR\trn.trf

It is also possible to add the option **-b** *(apply Baku Acceleration Method)*.

Repeating randomness *(by using a seed)*

The oxymoronic title means that it is possible to generate (or re-generate) the same *random* TRF replacing the configuration file (e.g. RTG_DIR\rtg.cfg) with a long integer number (from 0 to 9223372036854775807).

For instance, the command:

javafo -g 18980522 -o TEST_DIR\trn.trf

will always generate the following file:

Seed=18980522 Tournament Report File

as long as the build number of javafo.jar stays the same.

*Note 1. The seed of any tournament randomly generated by javafo RTG is retrievable from the field **012** (the first line of the output TRF).*

Note 2. If the name of the configuration file is a long integer number, it is taken as a seed.

Quick recap

Standard invocations:

```
javafo [-r]
javafo [-r] input-file -c [round-number]
javafo [-r] input-file [-b] -p [output-file] [-l [check-list-file]]
javafo [-r] [model-file] -g [-b] -o trf-file
javafo [-r] -g config-file [-b] -o trf-file
```

The square brackets [ ] represent something optional.

Some explanations:

javafo	indicates a file named **javafo.bat** with the following contents: `@java -ea -jar JVF_DIR/javafo.jar %*`
-r	show JaVaFo release and build numbers

input-file	in TRF(x) format
model-file	in TRF(x) format, model file for the random-tournament-generator
config-file	usually, it is a configuration file for the random-tournament-generator; however, if config-file is a long integer number (from 0 to

	9223372036854775807), such number is used as the seed for the random-generator used by the random-tournament-generator

-c ***[round-number]***	use JaVaFo as a checker **round-number**: number of the round to be checked; if missing, all the rounds are checked
-g	use JaVaFo as a random-tournament-generator; with this option at most one between model-file and config-file may be present
-b	apply, if feasible, the Baku Accceleration Method

-p ***[output-file]***	**output-file**: full or relative pathname where to write the pairing; if missing, the output file will be defaulted to standard output
-l ***[check-list-file]***	**check-list-file**: absolute or relative pathname where to write the check-list; if missing, the check-list will be defaulted to the input-file directory, with the same basename as the input-file and an extension of **.list**.
-o trf-file	**trf-file:** full or relative pathname where to write the (auto)generated TRF file

JaVaFo as Java archive

As an experimental tool, with versions 2.x, it is possible to include parts of JaVaFo in a Java project and have the possibility to directly interface JaVaFo as a pairing engine, a free pairing-checker or a random-tournament-generator.

The first operation is to open **javafo.jar** *(which is an archive that can be usually opened with the same tools that open .zip or .rar files)* and extract the file **main.jar**. The latter is the file to be included in a Java project. It exposes a static class, **JaVaFoApi**, that contains the definition of a method that can be called to invoke, depending on the parameters, the pairing engine, the pairing checker or the random generator.

The aforementioned method is:

```
        String JaVaFoApi.exec(int operation, Object... params);
```

that will execute the most common operations of JaVaFo (and a few others).

The first parameter (operation) may assume one of the following values:

operation	A mandatory four digit code (Java type **int**), identifying the particular JaVaFo operation:

Code	Mnemonics	Description
1000	PAIRING	*Standard pairing*
1001	PAIRING_WITH_BAKU	*Standard pairing, using, if applicable, the Baku Acceleration Method*
1100	PRE_PAIRING_CHECKLIST	*Check-list before doing the pairing* Note: this operation is undocumented using JaVaFo as a stand-alone program
1110	POST_PAIRING_CHECKLIST	*Check-list after the pairing has been done*
1111	POST_PAIRING_WITH_BAKU_CHECKLIST	*Check-list after the pairing has been done using, if applicable, the BakuAcceleration Method*
1200	CHECK_TOURNAMENT	*Check the correctness of a tournament*
1210	CHECK_ONE_ROUND	*Check the correctness of a single round of a tournament* Note: this operation is not possible using JaVaFo as a stand-alone program
1300	RANDOM_GENERATOR	*Generate a random tournament*

	1301	RANDOM_GENERATOR_WITH_BAKU	*Generate a random tournament using, if applicable, the Baku Acceleration Method*

Then the parameter list of **JaVaFoApi.exec** may contain the following parameters in any order (it is their Java type that identifies them):

input — For RTG operations is optional. Mandatory for all other operations.

Java type	Description
InputStream	*It is the stream thru which to read the TRF(x) of the tournament to be processed (a model for RTG operations).*
String	*Meaningful only if there is no InputStream parameter (otherwise it is ignored). It represents the TRF(x) of the tournament to be processed (a model for RTG operations).*
Properties	*Meaningful only for RTG operations if there are neither InputStream nor String parameters (otherwise it is ignored).* *It represents a collection of properties to be used during the generation of the random tournaments (see Reducing randomness by way of a (RTG) configuration file for the description of the usable properties).* *In addition to the previous list, also* **FirstSeed** *is a usable property, and represents the seed for the random-generator used by the RTG (the same seed will re-produce the same TRF as long as the same build of javafo.jar is used).*

output	Optional **OutputStream** parameter thru which the result of the requested operation is returned. If missing, the result of the operation will be returned in a String type thru the *exec* call (which is null, when there is an OutputStream parameter).
extra	Meaningful only for the CHECK_ONE_ROUND operation (otherwise it is ignored). It is an **Integer** (or an **int**) representing the round to be checked.

Examples

```
String out = JaVaFoApi.exec(1301, inputStreamTRF, 6);
```

Return into out *the TRF of a random-generated-tournament with the Baku Acceleration Method, using a model tournament read thru the* inputStreamTRF *input-stream. The actual parameter "6" is ignored.*

```
String out = JaVaFoApi.exec(1210, 6, new FileInputStream("C:\jvfck\trnXDW.trf"));
```

Return into out *the result of the verification of the 6th round of the tournament described by the TRF file* C:\jvfck\trnXDW.trf.

```
String out = JaVaFoApi.exec(1110, outputStream, trf);
```

Return thru the outputStream *output-stream the post-pairing check-list for the tournament of which the TRF is contained in the* trf *string. In* out *it is returned* null.

Note: it is possible to use the above operation to retrieve the pairing of the round, but it is not a very practical way to do it.

```
JaVaFoApi.exec(1000, "Hello World", inputStreamTRF, outputStream);
```

Return thru the outputStream *output-stream the pairings for the tournament of which the TRF has been read thru the* inputStreamTRF *input-stream. The string "Hello World" is ignored.*

```
Properties cfg = new Properties();
```

```
cfg.setProperty("PlayersNumber", 289);
cfg.setProperty("RoundsNumber", 9);
JaVaFoApi.exec(1300, cfg, new FileOutputStream("C:\jvfgen\trP289R9.trf");
```

Put into the file c:\jvfgen\trP289R9.trf *the TRF of a random-generated-tournament with 289 players and 9 rounds.*