

VDMJ High Precision Guide

The standard build of VDMJ performs arithmetic using Java longs (64-bits) for integer types and Java doubles for real types. This is sufficient for most model simulations, but there is an alternative build of VDMJ available that uses *BigIntegers* and *BigDecimals* to represent types. The alternative build is called the *high precision* build, and is available in the jar file called *vdmj-n.n.n-P.jar*.

The high precision build is exactly the same as the standard build for most purposes, and models that do not require high precision arithmetic will function in the same way, unless they depend on floating point results. Calculations are a few percent slower than the standard build for comparable precision, becoming progressively slower as the precision is increased.

The high precision build adds a new option to the command line, *-precision*, to specify the floating point precision required. By default this is 100, meaning floating point results are calculated to 100 significant figures. For example:

```
$ java -jar vdmj-3.0.1-P.jar -vdmsl -i stdlib/MATH.vdm -precision 50
Parsed 1 module in 0.292 secs. No syntax errors
Type checked 1 module in 0.015 secs. No type errors
Initialized 1 module in 0.238 secs.
Interpreter started
> precision
Decimal precision = 50
> print exp(1)
= 2.7182818284590452353602874713526624977572470937
Executed in 0.063 secs.

> precision 100
Decimal precision = 100
> print exp(1)
= 2.71828182845904523536028747135266249775724709369995957496696762772407663
0353547594571382178525166427
Executed in 0.084 secs.

> precision 1000
Decimal precision = 1000
> print exp(1)
= 2.71828182845904523536028747135266249775724709369995957496696762772407663
035354759457138217852516642742746639193200305992181741359662904357290033429
526059563073813232862794349076323382988075319525101901157383418793070215408
914993488416750924476146066808226480016847741185374234544243710753907774499
206955170276183860626133138458300075204493382656029760673711320070932870912
744374704723069697720931014169283681902551510865746377211125238978442505695
369677078544996996794686445490598793163688923009879312773617821542499922957
635148220826989519366803318252886939849646510582093923982948879332036250944
311730123819706841614039701983767932068328237646480429531180232878250981945
581530175671736133206981125099618188159304169035159888851934580727386673858
942287922849989208680582574927961048419844436346324496848756023362482704197
862320900216099023530436994184914631409343173814364054625315209618369088870
701676839642437814059271456354906130310720851038375051011574770417189861068
7396965521267154688957035035
Executed in 0.919 secs.
>
```

This example shows the high precision jar being started with the MATH library and an initial precision of 50. The *precision* command line option displays and/or changes the setting, so the three examples show the evaluation of the Euler constant, *e*, to 50, 100 and 1000 decimal places.

Integer calculations are not limited by the precision option, and are always produced to full accuracy, though ultimately the Java *BigInteger* type is limited in the number of digits it can hold. For example:

```
> precision 50
Decimal precision = 50
```

```
> print 123 ** 456
= 9925006877209885670083146205746963263729594081988690051981629888138286710
474939907792112866142614463805542423693627187249280035274164990211814381967
260156999810012079049675951763646544589562574160986620990050019840715324460
477896801696302805031026141761591446872991824068548787861764597693906346435
798616571173097639947850764922868634146696716791012665334213494274485146389
992748709248661097714611276356710167264595313219648143933987301708814041466
127119850033325571309614233515141463065168306551878408120367848770300280208
209123660351902625688062449968178138722757403548483127151568312374214909556
926046360965597770093884458061193124649516620869554031369814001163802732256
625268978083813635182879531427216211122223117090171561235570134755237153001
369385537983486566706001464330245910042978365396691378300229078428345562828
335547052993295605148447712933388115993021275868760279508857923043166169601
0232187390436601614145603241902386663442520160735566561
Executed in 0.01 secs.
>
```

Here we see that the decimal precision is set to 50, but the value of 123 raised to the power 456 has all of its digits returned, rather than the first 50 significant digits and the rest zeros (which would happen for a floating point calculation).

Although high precision values can be used anywhere in a VDM specification, the following uses will be limited to the least significant 64 bits:

- The field number in an expression like, *record.#3*
- The number of iterations in a map or function expression, like *x ** 3*.
- The index of a sequence.
- The arguments passed to *cycles*, *duration*, *periodic*, and *sporadic* statements.
- The *setPriority* argument for a CPU object in VDM-RT
- The number of repeats in a *traces* expression, like *op(){3}*.

Considering that a 64 bit value can be up to about 10^{18} there is no significant disadvantage in these limitations.

Note that if you use the VDMJUnit test framework, you need to link with the corresponding high precision version of the jar, called *vdmjunit-n.n.n-P.jar*. The difference is that the *runInt* and *runReal* methods in *VDMJUnitTest* return *BigInteger* and *BigDecimal* objects respectively, rather than returning *int* and *double*.