Quantity System Framework Documentation

Release 1.2

Ahmed Sadek Mohamed Tawfik

April 30, 2015

Contents

Contents:

CHAPTER 1

Introduction

The **Quantity System** is a platform that contains two foundational libraries. *Quantity System Framework* and the *Quantity System Runtime*

Quantity System Framework is a nearly complete solution for adding the physical units functionality into any CLR compatible language (Quantity System itself is written with C#)

Quantity System Runtime is a dynamic language intended for scientists and engineers to aid them in making a clear calculations by aiding of units, and based on the underlieng quantity dimension.

one of the most distinguishable factors of quantity system is its unique handling of quantities by implying the use of dimensional analysis for all the quantities it contains.

historically a lot of attempts has been made to add the units of measurments into the programming languages either dynamically in runtime or in compile time.

http://jscience.org/ for example has added the support of units of measurments.

F# also has introduced the concept of units of measurements to be used in a compile time approach.

another noticable project called Frink in http://futureboy.us/frinkdocs/ employs also the units of measurements.

These projects however, despite of their understanding of the quantity dimensions of the basic quantities, didn't introduce a solution that is a dimensionally unique .. but instead introduced a solution that depends on the units itself to distringush between different quantities.

This approach may help in keeping the consistency of summing different units .. but it doesn't help in predicting the quantiy that comes out from the rest of mathematical operations i.e. multiplication and division of quantities .. and in turn prevent the quantity type check in calculations.

The approach of treating quantities with their units instead of their dimensions was unavoidable due to the look alike dimensions of quantities such that Work and Torque. The dimension similarties between these quantities and other quantities led any attempt to make a truly quantity validation an impossible process.

In this study and this project, we are keeping a high profile of the engineering requirements for keeping the quantity types consistency during calculations. This study introduced a new solution for differentiating between the similar quantities yet different in their physical meaning to keep the consisteny of quantities during calculations.

CHAPTER 2

Quantity System Framework

The framework of quantity system contains the physical quantities beside their representing units in a strongly typed way that defines the quantity name and its underlieng dimension and its corresponding units.

2.1 QuantityDimension

This class is the representation of the

2.2 AnyQuantity<T>

This class is the base class of all base quantities defined in the framework. The base quantities are those quantities serves as the most abstracted physical quantities such as Length, and Mass.

This class contains all the mathematical operations algorithms required to carry out the calculations between different inherited quantities.

<*T*> The T type is the storage type that is internally stored in the quantity. This architecture design was selected after a thoughtful considerations based on the idea that the quantity of something should not be limited to one numerical type aka double precession for example.

This architecture allows the extension of the framework to use any numerical storage required such as Big Integers in addition any user specific storage.

In this direction .. the one is able to sum a mass of integer to the mass of double *Mass*<*double*>+*Mass*<*int*> as long as there is an op_add operator between the double and integer types (which ofcourse it exists)

The framework however is optimized for the <*T*> of value types such as all numerical based types.

2.3 Base Quantities

1. Mass
2. Length
3. Time
4. ElectricCurrent
5. Temperature

6. AmountOfSubstance
7. LuminousIntensity
8. Currency

The first seven quantities are the nature physical quantities, however, the last quantity is the currency which holds all the currency units in the world.

It worth mentioning also that the framework should implement a special quantity for the data storage as of bit, byte ,.. etc.

2.4 DerivedQuantity<T>

This class is the direct parent of all derived quantities defined in the framework. The class contains the required algorithms to discover the returned quantity from calculations plus it acts as a generic quantity for the quantities that doesn't have a strongly typed class.

Following code illustrate how Force Quantity is being declared using this base class:

```
public class Force<T> : DerivedQuantity<T>
{
    public Force()
        : base(1, new Mass<T>(), new Acceleration<T>())
    {
    }

    public Force(float exponent)
        : base(exponent, new Mass<T>(exponent), new Acceleration<T>(exponent))
    {
    }

    public static implicit operator Force<T>(T value)
    {
        Force<T> Q = new Force<T>();

        Q.Value = value;

        return Q;
    }

}
```

CHAPTER 3

Quantity System Runtime

The runtime of the quantity system contains the new language introduced in this project.

This language aims to help the developer and the engineer in using a natural mathematical tone when using the computer to carry out different calculations.

The runtime introduce the parser that takes the user input and convert into an executable expressions using the Linq experssions.

Linq Expressions is then compiled and carried to the .NET runtime to be executed and getting back the required results.

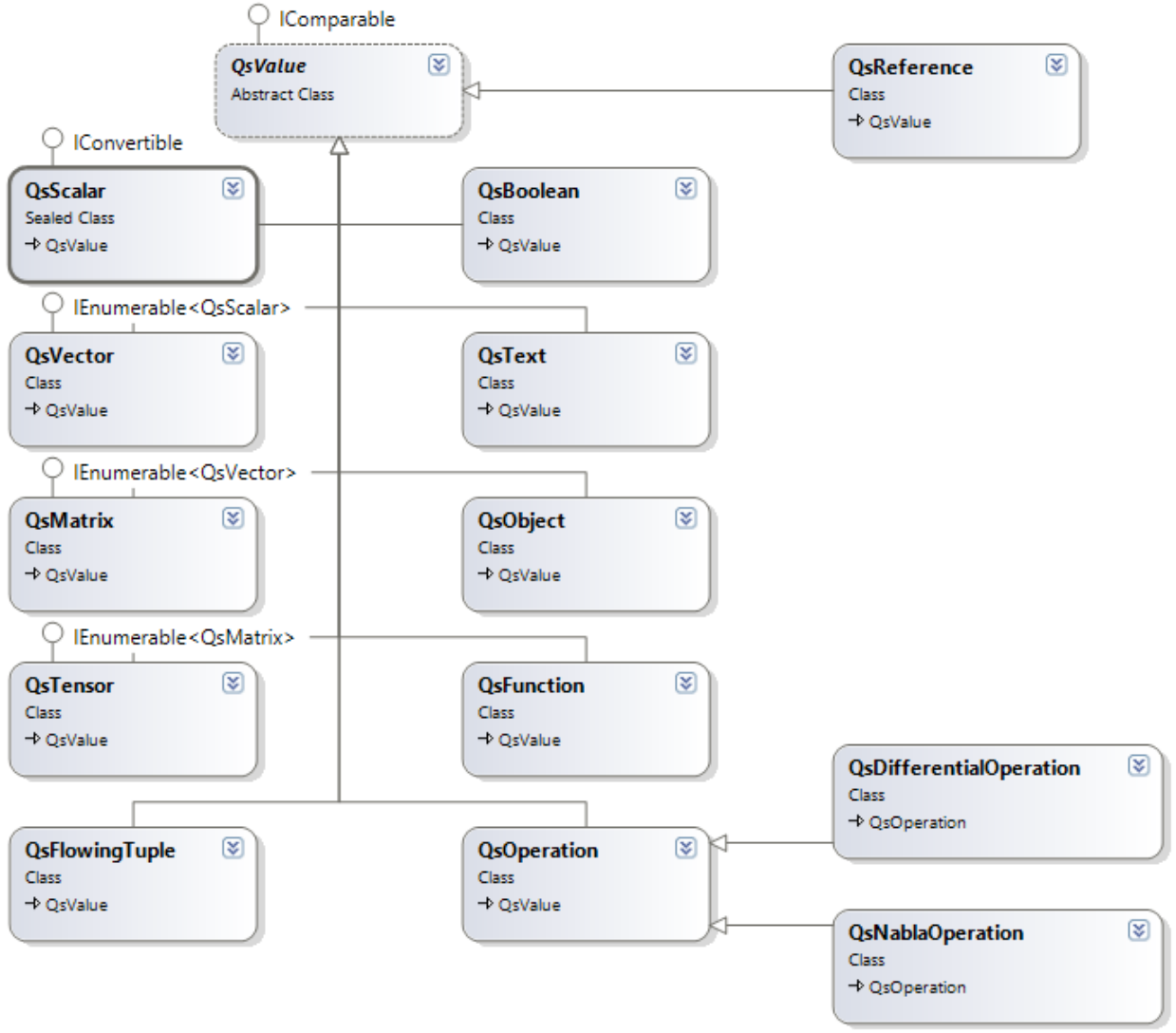
The runtime is in continous development to add the missing puzzle parts of mathematics

CHAPTER 4

Runtime Types

All the runtime types are inherited from the *QsValue* reference type. These types has gone through a careful architecture to describe the sematics of the language elements.

The following class diagram illustrate how these types are related to the *QsValue*. After this diagram a breif discussion for each type is illustrated for the reader to follow what the runtime is capable of.

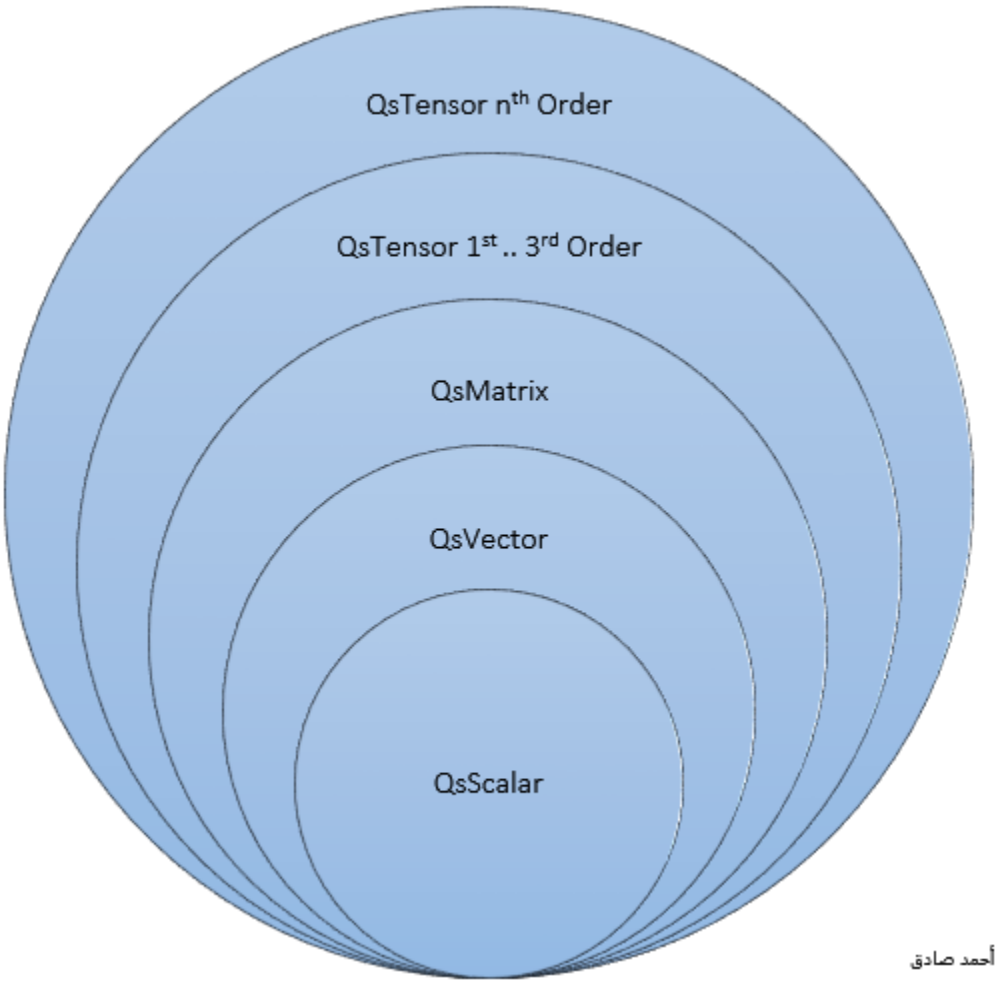


4.1 Mathematical Types

Mathematical Types are those types that shape the algebraic operations in the runtime.

These mathematical types includes Scalars, Vectors, Matrices, and Tensors.

The relation between these types can be show in the next figure.



The explanation of these types comes in subsequent chapters.

1. Making a Scalar Value obeys the format *variable* = *value*<*unit*>:

```
Qs> a = 5<kg>
    Mass: 5 kg
Qs> b = 6<slug>
    Mass: 6 slug
Qs> a+b
    Mass: 92.563421551991 kg
```

2. Making a vector value obeys the format *variable* = *{scalar scalar ... scalars}*:

```
Qs> v = {3<m> 4<m> 5<m>}
    QsVector: 3<m> 4<m> 5<m>
Qs> _||v||_                                                    #This is the vector magn
    Length: 7.07106781186548 <m>
```

3. Matrix values looks like the matrix declaration in matlab but units aware:

```
Qs> m = [10<in> 20<mm> 30<ft>; 5<fur> 30<m> 400<cm>; 60<in> 5<mil> 6<Mm>]
    QsMatrix:
      10<in>        20<mm>        30<ft>
      5<fur>         30<m>       400<cm>
      60<in>        5<mil>         6<Mm>
Qs> _|m|_                                                          #This is the determinant
    Volume: -38.6016990720004 <in.km^2>
Qs> 0<m^3> + _|m|_                                  #To convert the value into cubic meters
    Volume: -980483.156428811 <m^3>
```

4. Tensors declaration in the runtime are valid up to any higher dimensions Tensor declaration depends on a recursive mechanism in writing the values which implies that Tensor of Rank N can contain many tensors of Rank N-1:

```
# Vectors
a = {4 3 2}
b = {3 2 7}
c = {5 3 1}

# Tensors from zero rank to 4th rank
T0 = <| 3 |>
T1 = <| a b |>
T2 = <| a ;b ; c|>
T3 = <| a;b;c | b;c;a | c;a;b |>
T4 = <| T3 | T3*2 |>
```

4.2 QsBoolean

This is a True, and False values .. However this type was introduced recently in the runtime and most of comparison statements still use the System.Boolean type.

it worth to mention also that operations on boolean values are also implemented:

```
Qs> true + false
    True
Qs> true * false
    False
```

4.3 QsText

Any text that is written between “” is considered a QsText type. Shifting is available in QsText shifting with right and left operators that looks like C operators:

```
Qs> name = "Ahmed Sadek"
    Ahmed Sadek
Qs> name >> 4
    adekAhmed S
Qs> name << 5
    SadekAhmed
```

Applying indexer to the QsText result in getting the text separated by lines.

4.4 QsFunction

Defining a function is a corner stone in the runtime, to define the function you simply write *f(x) = expression*:

```
Qs> f(x) = x^2
Qs> f(7)
    49
```

4.5 QsObject

Special type for dealing with CLR objects.

4.6 QsReference

The reference type is a value that reference another value. It was created during the discussions of **rvalues** in C++ ISO C++ 11 specifications. Example:

```
Qs> P=35
    DimensionlessQuantity: 35 <1>
Qs> &PR = P
    P: DimensionlessQuantity: 35 <1>
Qs> PR = "hello there"
    P: hello there
```

The declaration of reference varaiable requires to preceded *&* similar to C++ references before the variable name. changing the reference value variable only changes the referenced value.

4.7 QsOperation

This is a base class for currenlty two implemented operations Differential and Nabla Operations.

The implemented operations are stored in the QsScalar class with an operation type that carried out during calculations to obtain specific techniques.

4.8 QsFlowingTuple

This is the tuple implementation in the runtime. The tuple is declared with between two brackets with more than one value and can contain any of the types mentioned in here beside itself.

Following is the declaration example of the tuple:

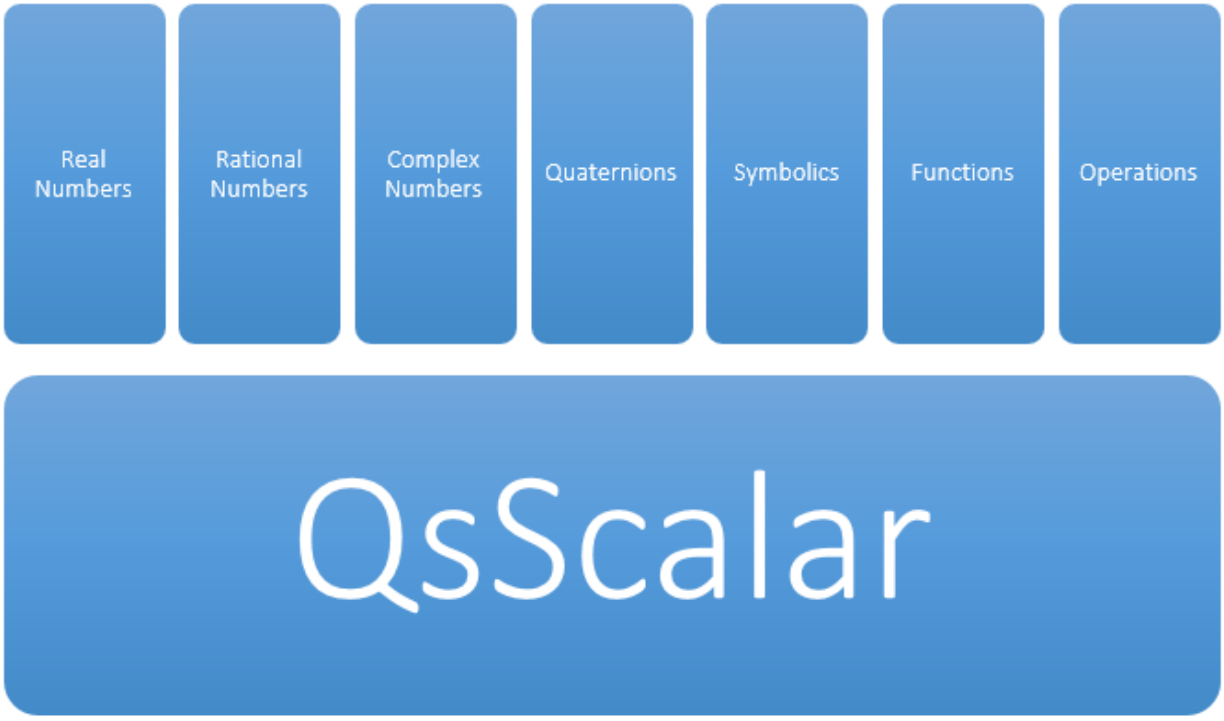
```
Qs> T = (40,80<kg>, (3,4,5), "hello there", @|$x)
    FlowingTuple (40<1>, 80<kg>, QsTuple[3 Elements], "hello there", @|x)
Qs>
```

CHAPTER 5

Scalars Types

Scalar quantitiy can be instantiated by simply typing *variable = value<unit>* if you omit the unit part then the quantity is dimensionless quantity.

The scalar quantity has 6 sub-types of values that is supported in quantity system as a first citizen.



5.1 Real Numbers

Declaring a real number is the most forward just type *variable = value*

5.2 Rational Numbers

Raional numbers contain a numerator and denominator part .. this can be acheived by typing *Q{numerator denominator}* In a music misura of 4/4 Example:

```
ronde   = Q{4 4}
blanche = Q{2 4}
noire   = Q{1 4}
croche  = noire / 2  # gives you DimensionlessQuantity: (1/8) <1>
dcroche = croche /2  # gives (1/16)
```

5.3 Complex Numbers

Complex numbers contains a real part and an imaginary part .. this can be acheived by typing *C{real imaginary}*. Example:

```
Qs> C{3 2} / C{4 -5}
        DimensionlessQuantity: (0.0487804878048781, 0.560975609756098i) <1>
Qs>
```

5.4 Quaternions

Quaternions are also supported and its prefix is *H*. The quaternion has a real number and three additional numbers i,j, and k.

Example:

```
Qs> H{4 3 2 1}
        DimensionlessQuantity: (4, 3i, 2j, 1k ) <1>
Qs>
```

Additional feature of quaternion that used in 3D programmign is to get the rotation matrix representation of the quaternion value this can be acheived by calling the RotationMatrix property of the quaternion:

```
Qs> rm = H{4 3 2 1}->RotationMatrix
        QsMatrix:
                -9<1>          4<1>          22<1>
                20<1>        -19<1>        -20<1>
           -10<1>         28<1>         -25<1>

Qs>
```

5.5 Symbolics

In quantity system .. symbolic values can be declared and treated like any other value. Symbolic values can be declared with *$* prefix. There are two ways to declare the symbolic value. First one is to write *$* preceded with any letters and this will be considered a simple symbolic value Example:

```
Qs> a = $a
Qs> b = $b
Qs> c = $vax
```

Second method is to enclose a symbolic value expression between brackets as shown before in the case of rational, and quaternion subtypes.

Exmaple:

```
Qs> a = ${a+b}
Qs> a^6
    DimensionlessQuantity: a^6+6*b*a^5+15*b^2*a^4+20*b^3*a^3+15*b^4*a^2+6*b^5*a+b^6 <1>
```

5.6 Functions

An important feature in quantity system is the ability to treat the functions as if they are variables. Declaring Scalar Functions can be made by two ways.

1. Referring to an existing function by preceding it with @:

   ```
   Qs> f(x,y) = x^2+y^4-5
   Qs> my_f = @f(x,y)
           DimensionlessQuantity: f(x,y) = x^2+y^2 <1>
   Qs>
   ```

2. Declaring the function anonymously inline:

   ```
   Qs> my_f = @{(x,y)=x^2+y^4-5}
   ```

Functions acting as a scalar value can be also be called like real functions by invoking indexing typing *my_f[3,5]* will get the required result.

5.7 Operations

Another unique scalar sub-type is being the operation type. This type defines and operation that can take action into calculations currently the operation introduced is the differentiation operation.

differentiation operation starts with @| followed by the variables that we wish to differentiate with respect to.

for example to declare a differentiation operation for variable *x* we write *@|$x* and if we want to differentiate for *x* then for *y* we write *@|$x|$y*

Exmaple:

```
Qs> my_o = @|$x
        @|x
Qs> f(x) = x^3
Qs> j = my_o * @f(x)   # this is equivalent to  @|$x * {(x)=x^3}
        DimensionlessQuantity: _(x) = 3*x^2 <1>
```

Another operator is the del operator */*

CHAPTER 6

Functions

6.1 Declaration

Functions in the runtime declared as it is written in mathematical contexts. Functions is not an imperative functions and only used to store mathematical expressions or a map that transform the value from the function domain to its result:

```
Qs> f(x) = x^2
```

Functions can have more than one parameters:

```
Qs> f(x,y,z) = x^2+y^2+z^2
```

Overloaded functions are implemented with parameters names:

```
Qs> f(u,v) = u+v^3
Qs> f(i,j) = i^3-j/exp(j)
Qs> f(i,j,k) = i/j/k
```

6.2 Function Calling

Functions are called in the context normally as the rest of other languages, however, if the function is overloaded by parameter names then the user should specify the desired function to be called by injecting the parameter name.

The runtime will choose the best function suits the named parameters specified:

```
Qs> a = f(6,5)

Qs> f(6,5)                                              # call f(u,v)
    DimensionlessQuantity: 131 <1>

Qs> f(j=5,i=6)                                  # call f(i,j)
    DimensionlessQuantity: 215.966310265005 <1>

Qs> f(v=5,u=6)                                  # call f(u,v) .. notice the same result.
    DimensionlessQuantity: 131 <1>

Qs> f(5,4,2)                                    # call f(x,y,z)
    DimensionlessQuantity: 45 <1>
```

```
Qs> f(5,4,k=2)                                  # call f(i,j,k)  because the inculsion of k in the pa
    DimensionlessQuantity: 0.625 <1>
```

6.3 Function Operations

Functions in the runtime has an interesting behaviours :). Functions can be added, multiplied, subtracted, and divided (thanks to the strong symbolic algebra project implemented by me also :D)

Whenever an operation done to the functions the runtime will grab the parameters and adding them into the new parameters

consider the following example continuing from the last one:

```
Qs> @f(u,v) + @f(i,j)  #summing two functions
DimensionlessQuantity: _(u,v,i,j) = u+v^3+i^3-j/exp(j) <1>

Qs> @k = @f(u,v) + @f(i,j)  #summing two functions with storing the new function into new name
Qs> @k
DimensionlessQuantity: _(u,v,i,j) = u+v^3+i^3-j/exp(j) <1>

Qs> k(3,4,5,2)    #calling the new function
DimensionlessQuantity: 191.729329433527 <1>
```

6.4 Function Derivation

Another strong ability of the runtime that helps in blending the symbolic calculations with numerical calculations more is the ability to derive the function using a syntax that is nearly looks like the normal mathematical expressions of the partial differntiation. By suffixing veritcal bar | and a symbolic variable that starts with *$*:

```
Qs> @K|$x                                                                  # No x parameter exists
    DimensionlessQuantity: _(u,v,i,j) = 0 <1>

Qs> @K|$u                                                                  # differentiate with respect
    DimensionlessQuantity: _(u,v,i,j) = 1 <1>

Qs> @K|$v
    DimensionlessQuantity: _(u,v,i,j) = 3*v^2 <1>

Qs> @K|$i
    DimensionlessQuantity: _(u,v,i,j) = 3*i^2 <1>

Qs> @K|$j
    DimensionlessQuantity: _(u,v,i,j) = -1/exp(j)+j/exp(j) <1>
```

The differentiation can take more than one variable in the same expression:

```
Qs> g(x,y,z) = x^3*y^2*z^4+log(x^2*z^4)
Qs> @g|$x
    DimensionlessQuantity: _(x,y,z) = 3*y^2*z^4*x^2+2/x <1>

Qs> @g|$y
    DimensionlessQuantity: _(x,y,z) = 2*x^3*z^4*y <1>

Qs> @g|$z
```

```
    DimensionlessQuantity: _(x,y,z) = 4*x^3*y^2*z^3+4/z <1>

Qs> @g|$z|$x
    DimensionlessQuantity: _(x,y,z) = 12*y^2*z^3*x^2 <1>

Qs> @g|$z|$x|$y
    DimensionlessQuantity: _(x,y,z) = 24*z^3*x^2*y <1>

Qs> @g|$x|$y|$y
    DimensionlessQuantity: _(x,y,z) = 6*z^4*x^2 <1>

Qs> @g|$x|$y|$z^2
    DimensionlessQuantity: _(x,y,z) = 72*x^2*y*z^2 <1>
```

6.5 Examples

```
Cr(n,k) = n!/(k!*(n-k)!)                    #Combinications
Pr(n,r) = n!/(n-r)!                                 #Permutations
```

Note: The exlamination mark *!* used to get the factorial

CHAPTER 7

Sequences

This unique feature is only included in the quantity system runtime which permits the user to form a mathematical sequences.

7.1 Declaration

The sequence in mathematics is used to generate values, the simplest example for a sequence:

```
Qs> S[n] ..> 2*n
Qs> S[0]
    DimensionlessQuantity: 0 <1>
Qs> S[1]
    DimensionlessQuantity: 2 <1>
Qs> S[10]
    DimensionlessQuantity: 20 <1>
```

Another valid declaration is to define the sequence without any counters at all:

```
Qs> P[] ..> 10;20;30
Qs> P[1]
    DimensionlessQuantity: 20 <1>
Qs> P[30]
    DimensionlessQuantity: 30 <1>
```

The last sequence expression is repeated when calling the sequence with an index that exceeds the declared ones. In previous example the 30 value were called at the index 30 because of that behaviour. The runtime will always gets the latest value occured on the latest indexed expression in the definition.

7.1.1 Fibonaccy Example

Following code is how to generate fibonaccy series using a recursive technique:

```
Qs> fib[n] ..> 0; 1; fib[n-1] + fib[n-2]                    #fibonaccy sequence
Qs> fib[0]
    DimensionlessQuantity: 0 <1>
Qs> fib[1]
        DimensionlessQuantity: 1 <1>
Qs> fib[5]
        DimensionlessQuantity: 5 <1>
Qs> fib[6]
        DimensionlessQuantity: 8 <1>
```

```
Qs> fib[1..10]
        QsVector: 1<1> 1<1> 2<1> 3<1> 5<1> 8<1> 13<1> 21<1> 34<1> 55<1>
Qs> fib[0..10]
        QsVector: 0<1> 1<1> 1<1> 2<1> 3<1> 5<1> 8<1> 13<1> 21<1> 34<1> 55<1>
```

although the series were made with a recursive technique, however, the runtime is actually caching the elements results in the case of a non changing element value case.

7.2 Declaration with Parameters

While sequence can have one sequence number that serves as a counter, the sequence can also have a parameters:

```
Qs> P[n](x) ..> x^n
Qs> P[0..10]
        QsVector: 1<1> x<1> x^2<1> x^3<1> x^4<1> x^5<1> x^6<1> x^7<1> x^8<1> x^9<1> x^(10)<1>

Qs> P[0..10](5)
        QsVector: 1<1> 5<1> 25<1> 125<1> 625<1> 3125<1> 15625<1> 78125<1> 390625<1> 1953125<1> 976562

Qs> P[0++10]
        DimensionlessQuantity: _(x) = (x^0) + (x^1) + (x^2) + (x^3) + (x^4) + (x^5) + (x^6) + (x^7) +

Qs> P[0++10](2)
        DimensionlessQuantity: 2047 <1>
```

when calling the sequence without specifying a parameter the sequence returns a function expression that can be stored into a function variable.

7.3 Sequence Calling

In addition to calling the sequence with indices sequence has another calling techniques that resemble mathematical series.

1. [n ++ m]: summation of series from *n* to *m*
2. [n ** m]: multiplication of series from *n* to *m*
3. [n !! m]: Average value between elements from *n* to *m*
4. [n !% m]: Standard Deviation between elements from *n* to *m*
5. [n .. m]: Returns Vector if the elements results were scalars, and matrix if the elements results were vectors.

consider the sin series to get the sin value of any number. For example the sequence declaration of numerical sin is:

```
Qs> math:my_sin[n](x) ..> ((-1)^n*x^(2*n+1))/(2*n+1)!           #Sin sequence
```

This sequence has one counter and one parameter. One can encapsulate the sequence calls into a function:

```
Qs> math:my_sin(x) = math:my_sin[0++50](x)                                    #Sin
Qs> math:my_sin(%PI/2)
    DimensionlessQuantity: 1 <1>
```

Note: %PI is a built in constant value.

The series proved that summation from 0 to 50 is good enough to get a reasonable result.

CHAPTER 8

Tuples

Quantity System Runtime supports tuples but in a very revolutional way.

Tuple is a list of values more specifically quantity system values that can be grouped together in the same variable.

Tuple also can contain another tuples.

8.1 Declaration

8.1.1 Normal Declaration

```
Qs> T = (20<kg>, 40<m>, "TextValue", (5,3))
    FlowingTuple (20<kg>, 40<m>, "TextValue", QsTuple[2 Elements])
Qs> T[0]
    Mass: 20 kg
Qs> T[2]
    TextValue
```

8.1.2 Declaration with Numerical Identifiers

Tuples can be declared with a numerical identifiers

```
Qs> H = (10:"First Value", 20:"Second Value", 22:50<kg>)
    FlowingTuple ("First Value", "Second Value", 50<kg>)
Qs> H:20
    Second Value
Qs> H:22
    Mass: 50 kg
```

8.1.3 Declaration with Textual Identifiers

Tuples can be declared with named elements, and this technique it resemble the key/value structral concept

```
Qs> N = (Name!"Ahmed Sadek", Age!35, City!"Cairo", Country!"Egypt", Father!(Name!"Sadek", Occupation
    FlowingTuple (Name!"Ahmed Sadek", Age!35<1>, City!"Cairo", Country!"Egypt", Father!QsTuple[2 Eler
Qs> N!City
    Cairo
Qs> N!Father
    FlowingTuple (Name!"Sadek", Occupation!"Retired")
```

```
Qs> N!Father!Name
    Sadek
Qs> N!Father!Occupation
    Retired
Qs> N!Age
    DimensionlessQuantity: 35 <1>
```

8.1.4 Declaration with Naming and Identifier

To complete the tuple declaration picture, the full syntax of tuple declaration looks like:

```
Qs> GearBox = (0:Neutral!"", 1:FirstShift!"Maximum Power", 2:SecondShift!"Getting Serious", 3:ThirdSh
    FlowingTuple (Neutral!"", FirstShift!"Maximum Power", SecondShift!"Getting Serious", ThirdShift!"
Qs> GearBox:3
    Are you nuts?!!
Qs> GearBox!SecondShift
    Getting Serious
```

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*