Table of Contents

1
The Julia Type System

In this chapter and the next two, we will discuss the features we will be looking the features which makes Julia appealing to the data scientist and scientific programmer.

Julia was conceived to meet the frustrations of the principal developers with existing programming languages, it is well designed and beautifully written. Moreover much of the code is written in Julia so is available to be inspected and changed. Although we do not advocate modifying much of the base code (also known as the standard library) it is there to look atand learn from.

Much of this book is aimed at the analyst, with some programming skills and the jobbing programmer, so we will postpone the guts of the Julia system until the last chapter when we consider package development and contributing to the Julia community.

This chapter will cover:

More about functions

We have met functions in previous chapters defined as in **`function() ... end`** block and that there is a convenient one-line syntax for the simplest of cases

```
# sq(x) = x*x is exactly equivalent to:
function sq(x)
    y = x*x
    return y
end
```

The variable y is not needed (of course). It is local to the**`sq()`** function and has no existence outside the function call andhe last statement could be written as return x*x or even just as x*x , since functions in Julia return their last value.

First class objects

Functions are first class objects in Julia. This means allows them to be assigned to other identifiers, passed as arguments to other functions, returned as the value from other functions, stored as collections and applied ('mapped') to a set of values at run-time.
The argument list consists of a set of dummy variables and the data structure using the () notation is called a tuple. By default the arguments are on type {Any} but explicit argument types can be specified which aids the compiler in assigning memory and optimising the generated code.

So sq(x) above would work with any data structures where the '*'operator is defined where as a definition of the form **`sq(x::Integer) = x*x`** would only work for integers. Suprisingly, *perhaps*, **`sq()`** does work for strings since the '*' operator is used for string concatenation rather than '+'.

```
sq(x) = x*x
sq("Hello") ; # => HelloHello
```

It is possible to overload the **+** operator for strings but since it is part of **Base** it is necessary to import it first

```
julia> "Hello"+" World"
ERROR: MethodError: no method matching +(::String, ::String)
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...) at operators.jl:504

import Base:+
+(s1::String,s2::String) = s1*s2; # or else string(s1,s2)

julia> "Hello"+" World"
"Hello World"
```

To apply a function to a list of values we can use the **`map()`** construct.

We are going to modify **`sq()`** slight so that it can broadcast over a more general type of data structures:

```
 julia> sq(x) = x.*x
 julia> map(sq, Any[1, 2.0, [1,2,3],7//5,"Hi"])
4-element Array{Any,1}:
 1
 4.0
 [1, 4, 9]
49//25
 "HiHi"
```

This definition of **sq()** will work with scalars too and we can use the **split()** function to turn strings into character arrays.

Notice the difference in the following constructs:

```
julia> map(sq,split("HI"));
julia> map(sq,split("H I"))
2-element Array{String,1}:
 "HH"
 "II"

julia> a=split("H E L L O")
5-element Array{SubString{String},1}:
 "H"
 "E"
 "L"
 "L"
 "O"

julia> b=split("W O R L D")
5-element Array{SubString{String},1}:
 "W"
 "O"
 "R"
 "L"
 "D"

julia> import Base.+
julia> +(s1,s2) = string(s1,s2)
+ (generic function with 176 methods)

julia> a.+b
5-element Array{String,1}:
 "HW"
 "EO"
 "LR"
 "LL"
 "OD"
```

We can list the methods of a function by using **methods()** which takes as its argument a function name. In Julia there is no difference in built-in and user defined functions (other than the requirement to import from Base) so our overloaded method for 'adding' strings is tacked on the the end of the list.

```
julia> methods(+)
# 176 methods for generic function "+":
+(x::Bool, z::Complex{Bool}) in Base at complex.jl:232
```

```
+(x::Bool, y::Bool) in Base at bool.jl:89
+(x::Bool) in Base at bool.jl:86
. . . .
. . . .
. . . .
+(s1, s2) in Main at REPL[12]:1
+(a, b, c, xs...) in Base at operators.jl:424
```

The **`string()`** function is quite useful because it can be used to convert convert and concatenate an **`Any`** datatype, although we need to be careful as a arithmetic expression will be evaluated before the string is created.

```
julia> +(s::String,a::Any) = string(s,a)
+ (generic function with 177 methods)
julia> +(a::Any, s::String) = string(a,s)
+ (generic function with 178 methods)
julia> "Hello " + 17//11 + " World"
"Hello 17//11 World"
julia> "Hello " + 17/11 + " World"
"Hello 1.5454545454545454 World"
```

Let's finish this section with an example other than squaring data structures by defining a function which computes the Hailstone sequence of numbers.
These can be generated from a starting positive integer, n by the following rules:

- If n is 1 then the sequence ends.
- If n is even then the next n of the sequence = n/2
- If n is odd then the next n of the sequence = (3 * n) + 1

There is a conjecture according to Collatz which states is that an hailstone

sequence for *any* starting number always *terminates*.

Here is the code which evaluate this and some sample output:

```
function hailstone(n::Integer)
  @assert n > 0
  k = 1
  a = [n]
  while n > 1
    n = (n % 2 == 0) ? n >> 1 : 3n + 1
    push!(a,n)
    k += 1
  end
 return (k,a)
```

```
end

julia> hailstone(17)
(13,[17,52,26,13,40,20,10,5,16,8,4,2,1])
julia> (m,s) = hailstone(1000)
(112,[1000,500,250,125,376,188,94,47,142 . . . 40,20,10,5,16,8,4,2,1])
julia> (m,s) = hailstone(1000000)
(153,[1000000,500000,250000,125000,62500,31250 . . . 10,5,16,8,4,2,1])
```

There is no obvious pattern to the number of iterations in order to converge but all integer values seem to eventually to do so. Note that we restrict the parameter type to be an integer using the modifier ***::Integer*** and check that it is positive with the ***@assert*** *macro.*

```
for i = 1000:1000:6000
   (mx,sx) = hailstone(i)
   println("hailstone($i) => $mx iterations")
end

hailstone(1000) => 112 iterations
hailstone(2000) => 113 iterations
hailstone(3000) => 49 iterations
hailstone(4000) => 114 iterations
hailstone(5000) => 29 iterations
hailstone(6000) => 50 iterations
```

The function starts by creating an array with the single entry 'n' and sets the counter (k) to 1. The while – end block will loop until the value of n reaches 1 and each new value is pushed onto the array. Since this effective modifies the array, by increasing its length, the convention of using a '!' is used.

1. The statement **`(n%2 = 0) ? n>>1 : 3n + 1`** encapsulates the algorithm's logic.
2. **`(condition)?statement-1:statement-2`** is a shorthand for an **`if else end`**, initially seen in C but borrowed by many languages including Julia.
3. **`n >> 1`** is a bit shift left so effectively halves n, when n is even.

The sequence continues until an odd prime occurs, when it is tripled and one added which results in a new even number and the process continues. While it is easy to see why the conjecture is true, the jury is still out on whether it has been proved or net.
It is worth noting that Julia orders its logical statements from left to right, so the operator **`||`** is equivalent to **`orelse`** and the operator **`&&`** to **`andthen`** .

This leads to another couple of constructs, termed short circuit evaluation, becoming popular with Julia developers:

```
(condition) || (statement)   # => if condition then true else perform the
statement
(condition) && (statement) # => if condition then perform the statement
else return false
```

Notice that because the constructs return a value this will be true for **||** if the condition is met and false for **&&** if it is not.
Finally the function returns two values, the number of iterations and the generated array and this must be assigned to a tuple. These constructs can be used to provide simple guards in a function via multiple return paths.

The following function checks whether an integer is prime:

```
function isp(n::T) where T<:Integer
  1 < n || return false
  n != 2 || return true
  isodd(n) || return false
  for i in 3:isqrt(n)
    n%i != 0 || return false
  end
  return true
end

julia> isp(107); # => true
julia> isp(119); # => false
```

The function again checks that the argument need to be an integer with a more general syntax by using a parametric types, using the syntax which we will discuss later in this chapter.

Closures and Currying

Since functions are first-class objects, this means that function references can be passed around in the same fashion as scalars, arrays and structures; this permits us to define closures in Julia.

A closure is an a way of storing a function while retaining its environment. The environment is a mapping associating each free variable of the function, viz. variables that are used locally, but defined in an enclosing scope with it's value or a reference to which the name was bound when the closure was created.

A closure, unlike a normal function, provides it with the ability those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

As an example consider the following code snippet:

```
julia> function counter()
   n = 0
   () -> n += 1, () -> n = 0
 end
counter (generic function with 1 method)
```

This is a very simple function which increase the variable *n;* it returns TWO references, the first to do the incrementing and the second to reset the counter

So it is called (instanced) as follows

```
julia> (addOne, reset) = counter()
(getfield(Main, Symbol("##3#5"))(Core.Box(0)),
 getfield(Main, Symbol("##4#6"))(Core.Box(0)))
```

So we can call it a few times, reset the counter and redo it, starting for zero.

```
julia> addOne(); addOne(); addOne()   #=> 3
julia> reset() #=> 0
julia> addOne(); addOne() #=> 2
```

Currying

Another consequence of functions returning references is that it is possible to instantiate some of the parameters and create new (simpler) functions which can be evaluated by specifying the remainder of parameters. This is a procedure well known to protagonists of functional programming.

The following is a simple example of currying in Julia

```
julia> function add(x)
  return function f(y)
    return x + y
  end
end
add (generic function with 1 method)
```

This is quite a simple *curried* function and a somewhat simplier definition is :

add(x) = y -> x + y

however as written above it makes the definition above a little clear.

We can demonstrate the use of this as:

```
# a3() creates a function to increment a values by 3.
julia> a3() = add(3)
#8 (generic function with 1 method)

# add() can be called in the following fashion
julia> add(3)(4)
7

# ... but also, more generally, as
julia> a3() = add(3);
julia> u = 4;
julia> a3()(u)
7
```

Passing arguments

Most function calls in Julia can involve a set of one or more arguments and in addition it is possible to designate an argument as being optional and provide a default value.
It is useful if the number of arguments may be varying length and also we may wish to specify an argument by name rather than by its position in the list.
How this is done is discussed below.

Default and optional arguments

In the examples so far all arguments to the function where required and the function call will produce unless all are provided. If the argument type is not given a type of **`Any`** is passed. It is up to the body or the function to treat an **`Any`** argument for all the cases which might occur or possibly trap the error and raise an exception
For example multiplying two integers results in an integer and two reals in a real. If we multiply an integer with a real we get a real number. The integer is said to be promoted to a real. Similarly when a real is multiplied with a complex number, the real is promoted to a complex and the result is complex.
When a real and an array are multiplied the result will be a real array, unless of course, it is an array of complex numbers. However two arrays are multiplied we get an exception raised, similar to:

```
julia> sq(x) = x*x
sq (generic function with 1 method)
julia> a = [1.0,2,3];
julia> sq(a)
ERROR: DimensionMismatch("Cannot multiply two vectors")
```

```
Stacktrace:
 [1] sq(::Array{Float64,1}) at ./REPL[10]:1
```

However we saw previously that we can definitely the square function using the `.*` construct and this will now work and the elements will all be promoted to reals

```
julia> sqq(x) = x.*x;
julia> a = [1.0,2,3];
julia> sqq(a)
3-element Array{Float64,1}:
 1.0
 4.0
 9.0
```

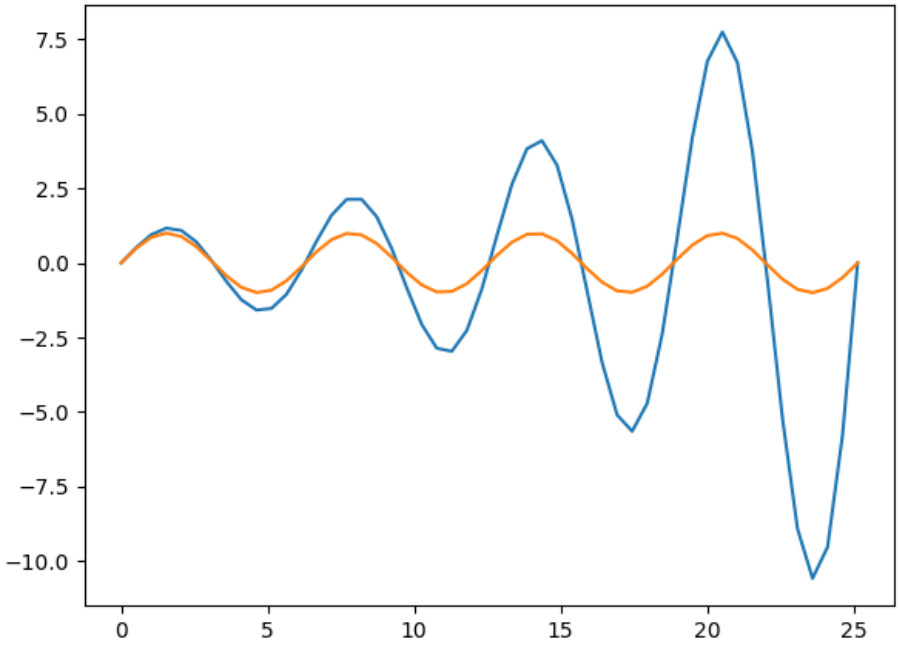
Typing of arguments is a good idea not only because it restricts function behaviour but also it aids the compiler. Just how this is done in Julia without over-loading a function for every possible combination of argument types, we will see later in this chapter.
Sometimes we wish for some (or all) of a function's argument to take default values if they are not provided. This is done by using an a ***`arg = value`*** syntax, such as

```
# Define a function and locate some array space
foo(x, p = 0.0) = exp(p*x)*sin(x);
t = linspace(0.0,8pi);
w = zeros(length(t));

# This can be done using a list comprehension such as
# w = [foo(u) for u in t]
#
for i = 1:length(w)
  w[i] = foo(t[i], 0.1)
end

using PyPlot
plot (t, w)
```

The figure 3.1 shows a plot of this function (p = 0.1) using PyPlot to display the result together with the default case (*viz* the sine wave)



In the call p is given the value p=0.1, however we still could pass a value such as p = 3 as this would be promoted in the function body to a real.
Looking at the methods for **`foo()`**

```
julia> methods(foo)
# 2 methods for generic function "f":
foo(x) at none:1
f00(x,p) at none:1
```

In fact we could pass a rational or even a complex number

```
julia> foo(2.0,3//4); # => 0.22313016
julia> foo(2.0,2 + 3im); # => 0.01758613 + 0.0051176733im
```

Because of the complex argument the result in the second case is complex too.
Optional arguments must come after required ones as otherwise the meaning would be ambiguous. Also when there are two optional parameters values for the all preceding ones must be provided in order to specify ones further down the list.
So defining a linear function:

```
julia> foo(x, y , a=2.5, b=4.0, c=1.0) = a*x + y *b +c
```

```
julia> foo(1,1);   # =>7.5 : all parameters are defaulted
julia> foo(1,1,2); #=> 7.0 : sets a equal to 2.0
julia> foo(1,1,2.5,4.0,3.0); # => 9.5
```

The final example sets *c = 3.0* but both *a* and *b* must be also be specified even though they are passing their default values. For long argument lists this is not practicable and it is better to use named parameters rather than simple optional ones.

Variable argument list

First we can look at the case where we wish to define a function which can take a variable number of arguments. We know that these type of functions exist as **+** is an example of one such.
The definition takes the form: g(a,b,c…) where a and b are required arguments but g can also take zero or more arguments represented by c…

In this case c will be returned as a tuple of values as the following illustrates:

```
function g(a ,b, c...)
  n = length(c)
  if n > 0 then
    x = zeros(n)
    for i = 1:n
      x[i] = a + b*c[i]
    end
    return x
  else
    return nothing
  end
end

julia> g(1.,2.); # => return 'nothing'
julia> g(1.,2.,3.,4.)
2-element Array{Float64,1}: #=> [ 7.0, 9.0 ]
```

The function needs to be 'sensible' in terms of its arguments but a call using rationals will work with this definition as they promote to reals

```
julia> g(1.0, 2.0, 3//5, 5//7)
2-element Array{Float64,1}:
 2.2
 2.42857
```

Since functions are first class objects these may be passed as arguments, so modifying the definition of g slightly gives a (very poor) map function:

```
function g(a ,b...)
  n = length(b)
  if n == 0 then
    return nothing
  else
    x = zeros(n)
    for i = 1:n
      x[i] = a(b[i])
    end
    return x
  end
end

julia> g(x -> x*x, 1. , 2., 3., 4.)
4-element Array{Float64,1}:
 1.0
 4.0
 9.0
 16.0
```

Note that in the cases where there was no variable arguments I chose to return ***nothing*** ; this is a special variable defined by Julia of type Nothing.

We will meet another special type ***missing*** when discussing Julia's implementation of dataframes.

Named parameters

Previously we defined a linear function in two variables (x,y) with three default parameters (a,b,c) but met the problem that to set the parameter c we need to supply values for a and b To do this we can use the following syntax:

```
julia> f(x, y; a=2.5, b=4.0, c=1.0) = a*x + b*y + c;
julia> f(1.,1.,c=1.); # => 7.5
```

The only difference is that the final three arguments are separated from the first two by a semicolon rather than a comma. Now a,b and c are named parameters and we can pass the value of c which out knowing the those of a and b. We can combine variable arguments and named parameters in a meaningful way as:

```
function foo(x...; mu=0.0, sigma=1.0)
  n = length(x)
  (n == 0) ? (return nothing) : begin
  a = zeros(n);
  [a[i] = (mu + sigma*rand())*x[i] for i = 1:n]
  a
```

```
  end
end

julia> foo(1.0,2.0,3.0, sigma=0.5)
3-element Array{Float64,1}:
 0.342006
 0.70062
 1.47813
```

So **`foo()`** returns a Gaussian variable with mean **`mu`** *and* standard deviation **`sigma`**.

Because Julia supports the Unicode character set it is possible to define the function using appropriate symbols μ and σ : i.e as *foo(x...; μ=0.0, σ=1.0)*

Scope

In the previous example we used ***`condition?statement-1:statement-2`*** notation as a short-hand for **`if-then-else-end`** . However it was necessary to wrap the code following the colon in **`begin-end`** .
This is a form of block, as are **`if`** statements, **`for`** and **`while`** loops.

Julia always signals the termination of the most recent block by way of the **`end`** statement.

Other examples of blocks we have met so far are those introduced by **`module`** , **`function`** and **`struct`** ***`(type)`*** definitions and also by **`try`** and **`catch`** statements.
The question we need to consider:- Is if a variable is declared inside a block is it visible outside it? -- this is controlled by Julia's scoping rules.
Since **`if-then-else`** or **`begin-end`** blocks do not affect a variables visibility it better to refer to the current scope rather than current block.

There are new scoping rules applying to the visibility of variables declared at the top-level.

These were discussed in chapter 1 and the reader is asked to read these if necessary, they will not be repeated here.

Certain constructs will introduce new variables into the current innermost scope when a

variable is introduced into a scope, it is also inherited by any inner scopes unless one of that scope explicitly overrides it.

- The rules are reasonably clear:
- A declaration local introduces a new local variable.
- A const is now only allowed at the top level
- A declaration global makes a variable in the current scope (and inner) scopes refer to the global variable of that name.
- A function's arguments are introduced as new local variables into the scope of the function's body.
- An assignment x = 1 (say) introduces a new local variable x only if x is neither declared global nor introduced as local by any enclosing scope before or after the current line of code.

To clarify the last statement in a function foo() such as :

```
function foo()
x = y = 0;
while (x < 5)
  y = x += 1;
end
println(y)
end
foo() ; # returns (y) => 5

function foo()
  x = y = 0;
  while (x < 5)
    local y = x += 1;
  end
  return y
 end
 f() ; # returns (y) => 0
```

Notice that the variable y in the while loop is local to it and so returned by the function is 0 not 5.
There is a further construct which Julia provides in passing anonymous function definitions as argument, which is do - end and one we will find convenient when working with file IO in the next chapter.
Consider mapping an array to its squares when the value is 0.3 or more

```
a = rand(5)
map(x -> begin
  if (x < 0.3)
```

```
    return(0)
  else
    return(x*x)
  end
end, a)
5-element Array{Real,1}:
# => [0.503944 , 0.711046, 0 , 0.214098 , 0]

map(a) do x
  if (x < 0.3)
    return(0)
  else
    return(x*x)
  end
end
5-element Array{Real,1}:
 # => [0.503944 , 0.711046, 0 , 0.214098 , 0]
```

Both produce the same result but the second is cleaner and more compact. The use of the **`do x`** syntax creates an anonymous function with argument *x* and passes it as the first argument to map.
Similarly, **`do a,b`** would create a two-argument anonymous function and a plain do would declare that what follows is an anonymous function of the form () -> ....
Note that Julia does not (*as yet*) have a switch statement (*as in C*) which would be equivalent to successive **`if-elseif-else-end`** statements. There are a packages which introduce a macro which will setup to generate multiple if-elseif-else statements, one of such is Match.jl
To illustrate let us consider the *Mathematicians* proof that all odd numbers are prime!, `(see discussion in http://rationalwiki.org)`

We can code this concisely using pattern matching as:

```
# First add the package: i.e. Pkg.add("Match")
using Match
allodds(x) = @match x begin
  !isinteger(x) || iseven(x) || (x < 3) => "Not a valid choice"
  3 || 5 || 7 => "$x is prime"
  _ => "By induction all numbers are prime"
end

# and running it on a select few gives:
for i in [1:2:9]
  @printf "%d : %s\n" i allodds(i)
end
1 : Not a valid choice
3 : 3 is prime
5 : 5 is prime
7 : 7 is prime
```

```
9 : By induction all odd numbers are prime
```

The Queen's problem

Finally I will introduce a function which we will use later for timing macros. This is to solve the Queens problem, which was first introduced by Max Bezzel in 1848 and the first solutions were published by Franz Nauck in 1850.
In 1972 Edsger Dijkstra used this problem to illustrate the power of what he called structured programming and published a highly detailed description of a depth-first backtracking algorithm..

The problem was originally to place 8 queens on a chessboard so that no queen could take any other, although this was later generated to N queens on an N by N board. An analysis of the problem is given in `Wikipedia`. The solution to the case N=1 is tivial and there are no solutions for N = 2 or 3. For a standard chess board there are 92 solutions, out of a possible 4.4 billion combinations of placing the queens randomly on the board, so an exhaustive solution is out of the question.
The Julia implementation of the solution uses quite a few of the constructs we have discussed:

```
struct Queen
    x::Integer
    y::Integer
end

qhorz(qa, qb) = qa.x == qb.x;
qvert(qa, qb) = qa.y == qb.y;
qdiag(qa, qb) = abs(qa.x - qb.x) == abs(qa.y - qb.y);

qhvd(qa, qb) = qhorz(qa, qb)||qvert(qa, qb)||qdiag(qa, qb);
qany(testq, qs) = any(q -> qhvd(testq, q), qs);

function qsolve(nsqsx, nsqsy, nqs, presqs = ())
    nqs == 0 && return presqs
    for xsq in 1:nsqsx
        for ysq in 1:nsqsy
            testq = Queen(xsq, ysq)
            if !qany(testq, presqs)
                tryqs = (presqs..., testq)
                maybe = qsolve(nsqsx, nsqsy, nqs - 1, tryqs)
                maybe !== nothing && return maybe
            end
        end
    end
```

```
    return nothing
end

# Usual case is a square board with the same number of queens
qsolve(nqs) = qsolve(nqs, nqs, nqs)

julia> qsolve(8)
Queen(1, 1), Queen(2, 5), Queen(3, 8), Queen(4, 6),
Queen(5, 3), Queen(6, 7), Queen(7, 2), Queen(8, 4))
```

The code has a matrix [**nsqsx** ***by*** **nsqsy** represent the board and so can be applied to non-square boards.

qhoriz(), qvert() ***and*** **qdiag()** return **true** if an horizontal, vertical or diagonal line contain more than a single queen.

qsolve() is the main function which calls itself recursively and uses tree pruning to reduce the amount of computation involved.

This computation slows down markedly with increasing 'n' and I'll use this function at the end of the chapter to give some benchmarks.

Julia's Type system

Julia implements and composite-aggregation object model rather than the most common inheritance ones which all for sub-typing and polymorphism.
While this might seem restrictive it allow use of a multiple dispatch call mechanism rather than the single dispatch one employed in the usual object orientated ones.
Coupled with Julia's system of types, multiple dispatch is extremely powerful. Moreover it is a more logical approach for data scientists and scientific programmers and if for no other reason exposing this to you the analyst/programmer is a reason to use Julia. In fact there are lots of other reasons as well, as we will see later in this chapter.

A look at the Rational types

The rational number type was introduced in the previous chapter and like most of Julia , it is implemented in the language itself and the source is in base/rational.jl and is available to inspection.

Because Rational is a base type it does not need to be included explicitly so we can explore it immediately

```
julia> fieldnames(Rational)
 2-element Array{Symbol,1}:
 :num
 :den
```

The **`fieldnames()`** function lists what in object-orientated parlance would be termed properties but what Julia lists as an array of symbols. Julia uses the `:` character as a prefix to denote a symbol and there will be much more to say on symbols when we consider macros.

`:num` num corresponds to the numerator of the rational and **`:den`** to its denominator

Be careful to distinguish between `:` *and* `::`.

The first denotes symbols while the latter indicates a variable's type.

To see how we can construct a Rational we can use the **`methods()`** function

```
julia> methods(Rational)
# 12 methods for generic function "(::Type)":
[1] (::Type{T})(z::Complex) where T<:Real in Base at complex.jl:37
[2] (::Type{Rational})(n::Integer) in Base at rational.jl:20
[3] (::Type{Rational})(n::T, d::T) where T<:Integer in Base at
rational.jl:18
[4] (::Type{Rational})(n::Integer, d::Integer) in Base at rational.jl:19
. . .
```

Obviously we wish to construct rationals from integers not floating-point numbers but in Julia there are different size integers ranging from Int8 to Int128 and we also have signed and unsigned flavours of integers.
Rather than providing a recipe for making a rational for every combination of these Julia provides various ways which are generic and **`methods()`** helpfully lists the line in the source file in which they occur.

The constructors [2] and [4] above are the simplest to understand
Rational(n::Integer) is used for converting an integer to a rational; i.e. when the denominator of 1 (and not passed). Rational(n::Integer, d::Integer) is used when both a numerator and denominator are provided and are integers.
Case [3] is similar to [4] but defined in terms of a parametric type T which is then constrained to be a subtype of the abstract type Integer, so includes all concrete integers, both signed and unsigned.

Case [1] is an extension to the original definition (in earlier version of Julia) to cover complex rational *where the complex coefficients are integer*:

```
julia> z1 = 5 + 1im;
julia> z2 = 3 + 2im;
julia> z1//z2
17//13 - 7//13*im
```

Parametric definitions are very useful for establishing the rules for manipulating types, as we will see later.

The entire source for the Rational type is quite long but the first few lines are informative and reproduced here:

Try the following to view the file

julia> less(string(Sys.BINDIR,"/../share/julia/base/rational.jl

```
struct Rational{T<:Integer} <: Real
 num::T
 den::T
 function Rational{T}(num::Integer, den::Integer) where T<:Integer
 num == den == zero(T) && throw(ArgumentError("invalid rational:
zero($T)//zero($T)"))
 num2, den2 = (sign(den) < 0) ? divgcd(-num, -den) : divgcd(num, den)
 new(num2, den2)
 end
end
Rational(n::T, d::T) where {T<:Integer}
 = Rational{T}(n,d)
Rational(n::Integer, d::Integer)
 = Rational(promote(n,d)...)
Rational(n::Integer) = Rational(n,one(n))

function divgcd(x::Integer,y::Integer)
 g = gcd(x,y)
 div(x,g), div(y,g)
end
```

Rationals are defined using the **`struct`** . This creates a number whose numerator and denominator are immutable, i.e. can not be altered once the number is defined. If we will need to modify a type's fields then if need to be defined with **`mutable struct`** .

```
julia> r = 5//7; r.num = 3; println(r)
ERROR: type Rational is immutable
Stacktrace:
 [1] setproperty!(::Rational{Int64}, ::Symbol, ::Int64) at ./sysimg.jl:9
 [2] top-level scope
```

The function following the type definition:

function Rational{T}(num::Integer, den::Integer) where T<:Integer
. . .
. . .
end

is termed the constructor and provides a recipe for creating the value. In cases where a constructor is no defined the default is just to fill in the type's fields with the values passed.

However for rationals this will not suffice. We need to check that the denominator is not zero (or else generate an error) and also to reduce the rational to its least possible form by dividing by an common factors between the numerator and the denominator. To this end a 'helper' function is defined **`divgcd()`** which itself uses the Base functions **`gcd()`** and **`div()`** . The special function **`gnew()`** is called return a value for the datatype.

The remaining Rational() constructors are special cases which will use the first definition to create the value.

Also we must provide rules for combining Rationals with Integers and what we might mean when passing real of complex numbers

```
function //(x::Rational, y::Integer)
  xn,yn = divgcd(x.num,y)
  xn//checked_mul(x.den,yn)
end

function //(x::Integer, y::Rational)
  xn,yn = divgcd(x,y.num)
  checked_mul(xn,y.den)//yn
end

function //(x::Rational, y::Rational)
  xn,yn = divgcd(x.num,y.num)
  xd,yd = divgcd(x.den,y.den)
  checked_mul(xn,yd)//checked_mul(xd,yn)
end

//(x::Complex, y::Real) = complex(real(x)//y,imag(x)//y)
//(x::Number, y::Complex) = x*y'//abs2(y)
function //(x::Complex, y::Complex)
 xy = x*y'
 yy = real(y*y')
 complex(real(xy)//yy, imag(xy)//yy)
 end
```

We see that rational numbers can be defined for complex numbers as long as **all** of the real and imaginary parts of **both** complex numbers are integer. Then this is easy to achieve by multiplying by the complex conjugate of the denominator which splits the number into a real (rational) and imaginary (rational) part.

Note the use of Real in the definition refers to non-imaginary rather than floating-point numbers; Integers are a type of Real as are Floats.

```
julia> (1+2im) / (3 + 4im)
 0.44 + 0.08im
julia> (1+2im) // (3 + 4im)
11/25 + 2//25*im
```

Just defining a type does not mean that we can do anything with it. For a numeric type we need to define the usual arithmetic functions (+, -, *, /) and also comparison operations (==, <, <=, >, >=)

The rest of the code in rational.jl deals with with definitions and we will look at how this is done when discussing parametric types and multiple dispatch.

Our own "Vehicle" data type

Let us build a data type from scratch and as an example let us look at data structures which describe vehicles and their ownership. This is in a file provided vehicles.jl which needs to be added to the Julia environment using the **`include`** statement or alternatively encapsulated in a module.

First we define the types and their associated fields as **`struct`** and the type hierarchy using the **`abstract type ... end`** syntax.

```
# Contact details for the vehicle's owner
struct Contact
  name::String
  email::String
  phone::String
end

# Vehicle is the top of the abstract type chain
abstract type Vehicle end

# Define Car, Bike and Boat as subtypes of Vehicle
abstract type Car <: Vehicle end
abstract type Bike <: Vehicle end
abstract type Boat <: Vehicle end
```

```
# Add Powerboat as subtype of Boat, and so of Vehicle
abstract type Powerboat <: Boat end
```

Now we will define some *concrete* types of cars, bikes and boats.

- These will have fields defined within them, so are no further subtypeable
- The field can not be changed once a variable has been constructed is the type is defined using **`struct`**
- To be able change the value of a field use the syntax **`mutable struct`**

```
struct Ford <: Car
  owner::Contact
  make::String
  fuel::String
  color::String
  engine_cc::Int64
  speed_mph::Float64
  function Ford(owner, make, engine_cc,speed_mph)
    new(owner,make,"Petrol","Black",engine_cc,speed_mph)
  end
end

mutable struct BMW <: Car
  owner::Contact
  make::String
  fuel::String
  color::String
  engine_cc::Int64
  speed_mph::Float64
  function BMW(owner,make,engine_cc,speed_mph)
    new(owner,make,"Petrol","Blue",engine_cc,speed_mph)
  end
end

struct VW <: Car
  owner::Contact
  make::String
  fuel::String
  color::String
  engine_cc::Int64
  speed_mph::Float64
end

struct MotorBike <: Bike
  owner::Contact
  make::String
  engine_cc::Int64
```

```
  speed_mph::Float64
end

struct Scooter <: Bike
  owner::Contact
  make::String
  engine_cc::Int64
  speed_mph::Float64
end

mutable struct Yacht <: Boat
 owner::Contact
 make::String
 length_m::Float64
end

mutable struct Speedboat <: Powerboat
  owner::Contact
  make::String
  fuel::String
  engine_cc::Int64
  speed_knots::Float64
  length_m::Float64
end
```

This defines three kinds of cars: Ford, BMW and VW.

Ford's and BMW's have constructors which set the fuel type and colour. In the case of a Ford this can not be changed (as Henry Ford said " You can have any colour as long as it is black") but BMW is defined as **`mutable struct`** so this can changed. The VW type, as with Ford, is also immutable.

Further we define two concrete types of bikes: Motorbike and Scooter and two types of boats: Yacht and Speedboat.

The Speedboat is a subtype of Powerboat, so we could define other concrete types such as DutchBarge, NarrowBoat etc., and assign different fields and create to these their own constructors.

With these definitions we can start to instant some variables corresponding to vehicles and their owners..

```
malcolm = Contact("Malcolm",
                  "mal@abc.net",
                  "+44 7777 555999");
myCar = Ford(malcolm, "Model T", 1000, 50.0);
myBike = Scooter(malcolm, "Vespa", 125, 35.0);
```

```
james = Contact("James",
                "jim@abc.net",
                "+44 7777666888");
jmCar = BMW(james,"Series 500", 3200, 125.0);
jmCar.color = “Black”;
jmBoat = Yacht(james,"Oceanis 44",14.6);
jmBike = MotorBike(james, "Harley", 850, 120.0);

david = Contact("David",
                "dave@abc.net",
                "+30 7777 222444");
dvCar = VW(david,"Golf", "diesel", "red", 1800, 85.0);
dvBoat = Speedboat(david,"Sealine 28","petrol",
                   600, 45.0, 8.2);
```

Note that James' BMW is black so after creating variable I changed the colour from blue to black by assigning: **`jmCar.color = "Black"`**

Given the type structure, we can already do something with these

```
cs = [myCar, jmCar, dvCar]
3-element Array{Car,1}:
Ford(Contact("Malcolm","malcolm@abc.net","07777555999",
             "Model-T","Petrol","Black",1000,50.0)
BMW(Contact("James","james@abc.net","07777666888"),
            "Series 500","Petrol","Black",3200,125.0)
VW(Contact("David","dave@abc.net","07777222444"),"Golf"
           ,"diesel","red",1800,85.0)
```

This defines the array **`cs`** since the Ford, BMW and VW are all subtypes of Car and Julia creates the appropriate array and we can iterate over it:

```
for c in cs
  who = c.owner.name
  model = c.model
  make = typeof(c);
  println("$who has a $make $model")
end

Malcolm has a Ford Model-T
James has a BMW series 500
David has a VW Golf
```

Similarly we could define the vehicles that James owns as:

```
vs = [jmCar, jmBike, jmBoat]
```

This will create an **`Any`** array of type **`Vehicle`** since that is the nearest common super

type and we can use this array to list the vehicles

```
println("James owns the following:")
for v in vs
  model = v.model
  make = typeof(v);
  mtype = super(make);
 print("$mtype:$make:$model\n")
end

Car:BMW:Series 500
Bike:MotorBike:Harley
Boat:Yacht:Oceanis 44
```

Next we can define a set of functions to be used with the Vehicle class.

```
function is_quicker(a::VW, b::BMW)
  if (a.speed_mph == b.speed_mph)
    return nothing
  else
    return(a.speed_mph > b.speed_mph ? a : b)
  end
end
```

Also we can compare different sort of vehicles such as a boat and a bike, taking account of the fact that a boat's speed is in expressed in knots and a bike or car in mph or kph.

```
function is_quicker(a::Speedboat, b::Scooter)
  const KNOTS_TO_MPH = 1.151
  # Bike speeds are defined in terms of MPH ...
  # whereas with Powerboats they are expressed as Knots.
  # We need to convert these using a local variable 'a_mph'
  a_mph = KNOTS_TO_MPH * a.speed_knots
  if (a_mph == b.speed_mph)
    return nothing
  else
    return(a_mph > b.speed_mph ? a : b)
  end
end

function is_longer(a::Yacht, b::Speedboat)
  if (a.length_m == b.length_m)
    return nothing
  else
    return(a.length > b.length_m ? a : b)
  end
end
```

```
is_quicker(a::BMW, b::VW) = is_quicker(b,a)
is_quicker(a::Scooter, b::Speedboat) = is_quicker(b,a)
is_longer(a::Speedboat, b::Yacht) = is_longer(b,a)

using Printf
@printf "%s %s\n" isquicker(dvCar,jmCar) "has the faster car"
James has the faster car

@printf "The faster vehicle is the %s\n" isquicker(msBike,dsBoat)
The faster vehicle is the Sealine 28.
```

So are we are not able to compare two BMW's or two VW's nor any car with a Ford!

While it would be possible to cover all the possibilities with only three types of car, it is hardly practicable to do this for all makes of car, let alone including bikes and boats. Julia solves this by using parametric types which we will discuss later in this chapter.

Secondly we require all vehicles to have a speed field and if defining rules by use of parametric types it the speed field will need to be the same symbol (i.e. have the same name) in each case.

Any defined functions can check for the existence of a field before trying to use it or alternatively use a **`try-catch`** block to trap the error but the problem largely goes away if concrete types can inherit fields from their supertype(s).

The vehicle type and accompanying constructors and function definitions are defined in a file vehicles.jl

There is some merit in treating this as a module and for this we would add the lines to the beginning of the file and, since **`module`** defines a block, terminate the whole with an **`end`** statement.

```
module Vehicles
export Contact, Vehicle, Car, MotorBike, Yacht, Powerboat, Boat
export Ford, BMW, VW, Scooter, Speedboat, isquicker, islonger
```

This is now accessed by means of the using **`using Vehicles`** statement and it will be picked up from the current directory or any that or in a special array called **`LOAD_PATH`** .

We can add our own directory(s) to this array by using a **`push!()`** call such as:

```
# .myjulia is a hidden directory in my home directory
# on a Mac
```

```
push!(LOAD_PATH,"/Users/malcolm/.myjulia)";
println(LOAD_PATH)
```

Putting this a **.juliarc** file will ensure that it happens each time that Julia is started; again this is a file located in my home directory.

In the next chapter I will look more closely at setting up a *.juliarc* file and some of the other things you may wish to include each time Julia starts up.

Note that all the types and functions which we wish to be visible must be included in an export statement. This can come at the beginning of the module (as is usual) or the end, it does not matter. Any function that's is not exported can still be called by has to be explicitly referenced as **Vehicles.islonger(jmBoat, dvBoat)**

Modules can also have using statements, so **using myModule** means that myModule will be available for resolving names as needed without the need to full reference the name and with the proviso that a function name does not class with any existing names.

Since modules are first-class objects in Julia, it is possible to shorten the syntax somewhat by using a variable as an alias:

using Vehicles;
vh=Vehicle;
vh.islonger(jmBoat,dvBoat)

When including a set of functions rather than the entire set such as **using myModule.foo1, myModule.foo2, myModule.foo3** and this can be shorted to **using myModule: foo1, foo2, foo3**

As well as **using** statements modules can also use which reference functions from other modules but not add them to the main namespace. The **importall** statement will import all functions exported from a module rather than individual ones.

Modules can also have **import** statements. These support all the same syntax as **using** but only operate on single name at a time although the **importall** statement will import all functions exported from a module rather than individual ones.

Importing does not add modules to be searched the way **using** does and also differs in that functions must be imported with the import statement to be extended with new

methods.

One last feature of incorporating our type system in a module is more a convenience than a necessity. When developing using the console REPL, any types defined are fixed and once defined we are unable to change them without restarting Julia and redefining them. However modules can be reused and this will effective redefine all they contain including any defined types.

Typealias and Unions

It is often convenient to introduce a new name for an already expressible type and for this Julia provides a typealias mechanism.

In version 1.0 the syntax has changed and now uses the following form:

```
julia> const Vector{T} = Array{T,1}
Array{T,1} where T

julia> const Matrix{T} = Array{T,2}
Array{T,2} where T
```

Type aliases are useful when defining umbrella type as a union of simpler ones. Union types are extensively used in the Base and there are many examples in the code listing there

```
julia> const Signed64 = Union{Int8, Int16, Int32, Int64}
Union{Int16, Int32, Int64, Int8}

julia> const Unsigned64 = Union{UInt8, UInt16, UInt32, UInt64}
Union{UInt16, UInt32, UInt64, UInt8}

julia> const Integer64 = Union{Signed64,Unsigned64}
Union{Sigend64,Unsigned64}
```

Recall in our vehicle type, we provided contact details as a name, email, phone type however alternatively it might be more appropriate to use a postal address. To accommodate both we can write:

```
julia> multable struct Address
  name::String
  street::String
  city::String
  country::String
  postcode::String
end

julia> postal = Address("Malcolm Sherrington","1 Main
```

```
Street","London","UK","WC2N 9ZZ")
Address("Malcolm Sherrington", "1 Main Street", "London", "UK", "WC2N 9ZZ")

julia> const Owner = Union{Contact,Address}
Union{Address, Contact}
```

The alias allows us to supply the owner field either as contact or postal details

```
struct Yacht <: Boat
  owner::Owner
  make::String
  length_m::Float64
end
julia> y1 = Yacht(me,"Moody 36", 11.02)
Yacht(Contact("malcolm","malcolm@abc.com","07777555999"),
"Moody 36", 11.02)
julia> y2 = Yacht(postal,"Dufour 44", 13.47)
Yacht(Address("Malcolm Sherrington","1 Main Street","London",
"UK","EC1A 9ZZ"), "Dufour 44". 13.47)julia>

c1.owner.name; # => "malcolm"
```

Enumerations

One problem with our vehicle type is that the fuel is defined as a string whereas it would be better to restrict the choice to set of values. Julia provides one approach to enumerations im terms of enumerations.

First we are going to a vector of types {Any} to hold the enumerated values. These could be consts using integers or Strings but I'll restrict to a list of symbols and create a file vnum.jl to hold the following:

```
const VecAny = Array{Any,1}

function vnum(syms::Symbol...)
  aa = Any[]
  for v in syms
    push!(aa,v)
  end
  aa
end

function vidx(aa::VecAny, s::Symbol)
  for (i, v) in enumerate(aa)
    if v == s then
      return (i - 1)
    end
```

```
    Nothing
  end
end
vin(aa::VecAny, s::Symbol) = (vidx(aa,s) >= 0 ? true : false)
```

The vnum is created by pushing a variable list of symbols onto an empty Any vector. Additionally there is a function vidx() which returns the position in the enumeration, holding with convention this is zero-based, and a one-line vin() checks that a symbol is on the v-nmeration.

We can use this to define our fuels types as:

```
fs = vnum(:NONE,:PETROL,:DIESEL,:LPG);
vidx(fs,:DIESEL); # => 2
vin(fs,:NONE);    # => true
vin(fs,:GASOIL);  # => false

julia> c2.owner.name; # => "Malcolm Sherrington"
julia> c1.owner.email; # => "malcolm@abc.com"
julia> c2.owner.email
ERROR: type Address has no field email

julia> typeof(c1.owner); # => Contact
julia> typeof(c2.owner); # => Address

julia> isa(c1.owner,Contact); # => true
julia> isa(c1.owner,Address); # => false
```

Multidimensional Vectors and PI revisited

We will start by defining a vector in 3 dimensions, this is a vector in mechanics sense and not the Julia vector, i.e. a one dimensional array.

We will define a module V3D as:

```
# This module uses Float64 components but could use a parameterised
# type {T} as we will see later
#
module V3D

# import operators from Base and Linear Algebra
import Base: +, *, /, ==, <, >, zero, one, iszero
import LinearAlgebra: norm, dot

# and export the type Vec3 and the norm, dist functions (for Vec3)
export Vec3, norm, dist
```

```
# now define a simple structure to hold the coordinates of the vector
struct Vec3
 x::Float64
 y::Float64
 z::Float64
end

# and the 'usual' runs for manipulating vectors
(+)(a::Vec3, b::Vec3) = Vec3(a.x+b.x, a.y+b.y, a.z+b.z)
(*)(p::Vec3, s::Real) = Vec3(p.x*s, p.y*s, p.z*s)
(*)(s::Real, p::Vec3) = p*s
(/)(p::Vec3, s::Real) = (1.0/s)*p

(==)(a::Vec3, b::Vec3) = (a.x==b.x)&&(a.y==b.y)&&(a.z==b.z) ? true : false;

# here are the scalar (dot) product and the norm of then vector
dot(a::Vec3, b::Vec3) = a.x*b.x + a.y*b.y + a.z*b.z;
norm(a::Vec3) = sqrt(dot(a,a));

zero(Vec3) = Vec3(0.0,0.0,0.0);
one(Vec3)  = Vec3(1.0,1.0,1.0);
iszero(a::Vec3) = (v == zero(Vec3))

(<)(a::Vec3, b::Vec3) = norm(a) < norm(b) ? true : false;
(>)(a::Vec3, b::Vec3) = norm(a) > norm(b) ? true : false;

# also define the distance between two vectors
dist(a::Vec3, b::Vec3) = sqrt((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y) +
(a.z-b.z)*(a.z-b.z))

end
```

We can now use this module to define a couple of 3-D vectors and output the distance between them as:

```
using Main.V3D
using Printf

v1 = Vec3(1.2,3.4,5.6);
v2 = Vec3(2.1,4.3,6.5);
@printf "Distance between vectors is %.3f \n" dist(v1,v2
Distance between vectors is 1.559
```

It is also possible to create matrix of 3D-vectors (quickly)

```
@elapsed begin
 vv = [Vec3(rand(),rand(),rand()) for i = 1:1000000];
 vs = reshape(vv,1000,1000);
```

```
end
0.063921754
julia> vs
1000×1000 Array{Vec3,2}:
Vec3(0.175379,0.930732,0.265873) Vec3(0.932432,0.495334,0.684214) . . .
Vec3(0.476968,0.407128,0.41125)  Vec3(0.00346352,0.213962,0.622112)
. . . .
```

Lead ins

```
We can now use the volume of the unit sphere (i.e 4πr/3) to estimate PI,
recall that the norm of the vectors will lie in the octant of the sphere:k
= 0
for i in 1:length(vv)
 if norm(vv[i]) < 1.0 k +=1 end
end
@printf "Estimate of PI is %9.5f\n" 6.0*k/length(vv)
3.14281
```

Parameterisation

One obvious problem with the module as defined above is that the components of the vector are defined in terms of the concert type Float64. As we saw earlier it is possible to define a structure in terms os a parameterised type and we wish to ensure the all the component s are of the same type {T}.

The following rewriting of the module does this:

```
module V3P
#
# import Base.+, Base.*, Base./, Base.norm, Base.==, Base.<, Base.>
#
import Base: +, *, /, ==, <, >
import LinearAlgebra: norm, dot
export Vec3P, norm, dist

struct Vec3P{T<:Number}
 x::T
 y::T
 z::T
end

(+)(a::Vec3P, b::Vec3P) = Vec3P(a.x+b.x, a.y+b.y, a.z+b.z)
(*)(p::Vec3P, s::Real) = Vec3P(p.x*s, p.y*s, p.z*s)
(*)(s::Real, p::Vec3P) = p*s
(/)(p::Vec3P, s::Real) = (1.0/s)*p
```

```
(==)(a::Vec3P, b::Vec3P) = (a.x==b.x)&&(a.y==b.y)&&(a.z==b.z) ? true :
false;
dot(a::Vec3P, b::Vec3P) = a.x*b.x + a.y*b.y + a.z*b.z;
norm(a::Vec3P) = sqrt(dot(a,a));

(<)(a::Vec3P, b::Vec3P) = norm(a) < norm(b) ? true : false;
(>)(a::Vec3P, b::Vec3P) = norm(a) > norm(b) ? true : false;

dist(a::Vec3P, b::Vec3P) =
   sqrt((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y) + (a.z-b.z)*(a.z-b.z))

end
```

Now we can pass other numeric types, viz. complex numbers, rationals etc

```
julia> using Main.V3P
julia> z1 = Vec3P{Complex}(1 + 2im, 2 + 3im, 3 + 4im);
julia> z2 = Vec3P{Complex}(3 - 2im, 4 - 3im,5 - 4im);

julia> z1 + z2
Vec3P{Complex{Int64}}(4 + 0im, 6 + 0im, 8 + 0im)

julia> zn = norm(z1 + z2)     # => sqrt(116) but it IS a complex number.
10.770329614269007 + 0.0im
julia> @assert zn == sqrt(116)

julia> r1 = Vec3P{Rational}(11//7, 13//5, 8//17)
Vec3P{Rational}(11//7, 13//5, 8//17)
julia> r2 = Vec3P{Rational}(17//9, 23//15, 28//17)
Vec3P{Rational}(17//9, 23//15, 28//17)

julia> r1 + r2
Vec3P{Rational{Int64}}(218//63, 62//15, 36//17)
julia> norm(r1 + r2)    # this is REAL, since sqrt(Rational) is a Float
5.791603442602598
```

Higher dimensional vectors

We are able to extend the 3-D vector to higher dimensions, in which case we will need to use an array to store the vector's components and pass the number of dimensions as a second parameter.

As an example we will use a secondary package **`StaticArrays`** which provides an **`SVector`** structure to hold the components. The following is not a full definition, just defining sufficient operations to calculate the distance between to N-Vectors and give, yet another, estimate of PI.

```
module VNX
using StaticArrays

import Base: +, *, /, ==, <, >
import LinearAlgebra: norm, dot

export VecN, norm, dist
struct VecN
  sv::SVector;
end

sizeof(a::VecN) = length(a.sv)
sOK(a::VecN, b::VecN) =
  (sizeof(a) == sizeof(b)) ? true : throw(BoundsError("Vector of different
lengths"));
(+)(a::VecN, b::VecN) = [a.sv[i] + b.sv[i] for i in 1:sizeof(a) if
sOK(a,b)]
(*)(x::Real, a::VecN) = [a.sv[i]*x for i in 1:sizeof(a)]
(*)(a::VecN, x::Real) = x*a
(/)(a::VecN, x::Real) = [a.sv[i]/x for i in 1:sizeof(a)]
(==)(a::VecN, b::VecN) = any([(a.sv[i] == b.sv[i]) for i in 1:sizeof(a) if
sOK(a,b)])
dot(a::VecN, b::VecN) = sum([a.sv[i]*b.sv[i] for i in 1:sizeof(a) if
sOK(a,b)])
norm(a::VecN) = sqrt(dot(a,a));
(<)(a::VecN, b::VecN) = norm(a) < norm(b) ? true : false;
(>)(a::VecN, b::VecN) = norm(a) > norm(b) ? true : false;

dist(a::VecN, b::VecN) = sum(map(x -> x*x,[a.sv[i]-b.sv[i]
                                  for i in 1:sizeof(a) if sOK(a,b)]))
end
```

The extension of the sphere to higher dimension is termed the n-ball (n>3) and the volume of n-balls is available via this `Wikipedia page`.

Here is an version for 4-balls; recall that now the 'volume' comprising all vectors with positive components is 2^N,
i.e 1/16th for the 4-sphere.

```
# Generate K 4-vectors
#
using Main.VNX, StaticArrays
K = 10^5;
vv = Array{VecN}(undef,K);

for j = 1:K
 vv[j] = VecN(@SVector [rand() for i = 1:4])
end
```

```
# Sum up the vectors which lie within the 4-ball
#
s = 0;
for j = 1:K
 if (norm(vv[j]) < 1.0)
   global s += 1
 end
end

# Volume of the unit 4-ball is 2*π*π
# The count is 1/16th of the volume
#
mypi = sqrt(32*s/K); # => 3.14592Summary
```

Summary

In this chapter we have looked how the Julia type system defines common numeric and string types and the how the use of parametric types provides an efficient mechanism for formulating more complex type structures.

We developed a set of types for a class of vehicle types and then added data to create and manipulate some specific instances.

Finally we looked at a numeric example of 3-D vectors and showed how this could be extended to greater dimensions. We applied these types to an example we saw early, viz. the Monte Carlo estimation of PI

Index