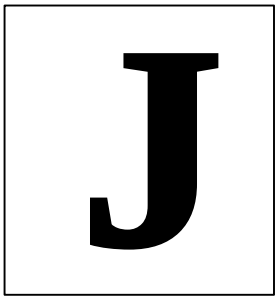


Primer

Eric B. Iverson

J Primer

Iverson Software Inc.
33 Major St.
Toronto, Ontario
Canada M5S 2K9

www.jsoftware.com

ISBN 1-895721-17-2

Table of Contents

Start here 1
Why J 1
Purpose of this book 2
Your background 2
How to use this book 2
Environment 3
Get started 4
Experiment 5
Standard profile 6
Terminology 7
Alphabet 7
Word 8
Sentence 8
Verb 8
Noun 8
Number 9
Negative number 9
Primitive 10
Name 10
Comment 11
Error 12
Ambivalence 12
Dyad 12
Monad 13
Vocabulary 13
Checkpoint A 16

Numeric constant 16
String 16
Word formation 17
Space 19
Precedence 20
Parentheses 21
Order of evaluation 21
Verb definition 23
Monad and dyad definition 25
Script file 26
Local 30
Global 31

Debug global 33
When =. and =: are the same 34
When they aren't 34
Locale 35
z locale 38
Script load 38
Checkpoint B 39

Debug - stepping through a verb 40
Debug - an error 42
Comparative 43
Control structure 44
Checkpoint C 46

Basic way of adding lists 47
J way of adding lists 50
A few more primitives 52
Plot 54
Plot locale 55
Print precision 56
Inexact numbers 58
Tolerance 59
Checkpoint D 61

Atom 62
List 62
Table 62
Array 63
Axis 64
Shape 64
Rank 65
Empty Array 65
Single atom array 66
Verb arguments 67
Frame and cell 68
Item 69
k-cell 70
Verb rank 71
Agreement 72
Rank conjunction " 74
Result shape 76

Checkpoint E 78

Adverb 78
Insert adverb 79
Table adverb 81
Conjunction 85
Order of execution - adverbs & conjunctions 85
Box - monad < 86
Link - dyad ; 87
Open - monad > 89
From - dyad { (selecting items) 91
From (boxed indexes) 92
From (scattered indexing) 95
Amend } (modify selected) 95
Selecting without from 98
Cut ;. 100
Each 102
Hook 104
Fork 104
Tacit definition 105
Explicit-to-tacit translator 109

Checkpoint F 111

Foreign !: 112
Files 113
Component files 118
Graphical user interface 120
Data processing 127
GUI 132

Where to go from here 134

J Online Documentation 135
J Dictionary 135
J Phrases 136

Start here

J is a general purpose, high-level programming language. If you are new to J and want to be a J programmer, this is a good place to start. Even if you have considerable programming experience, there is much that is unique to J, and it is worthwhile to at least skim this book before jumping into the deep end.

Why J

J is a very rich language. You could study and use it for years, and still consider yourself a beginner. This is in sharp contrast to simpler languages like Basic or Java, where months of concerted study and use would make you an expert. The effort required to become an expert J programmer is closer to that required to become an expert C++ programmer.

The good news is that the essence of J is so simple and consistent, that you can quickly learn enough to start solving real and interesting problems.

It is easier to learn enough Basic or Java to solve trivial problems, but it is easier to learn enough J to solve more interesting and challenging problems. And once you have that level of skill under your belt, you are not at the end of the road, but can continue to improve, making yourself a better and more formidable programmer.

J is particularly strong in the mathematical, statistical, and logical analysis of arrays of data. It is a powerful tool in building new and better solutions to old problems and even better at finding solutions where the problem is not already well understood.

As well as being a general purpose programming language, the J system also provides:

- an integrated development environment
- standard libraries, utilities, and packages
- a form designer for your application forms (windows)
- an event-driven graphical user interface to your application
- several methods of interfacing with other programming languages and applications
- rapid application prototyping and development
- royalty-free distribution of run-time versions of your application

If you are interested in programming solutions to challenging data processing problems, then the time you invest in learning J will be well spent.

Purpose of this book

The *J Dictionary* is the authoritative and definitive specification of the J language. It can be used to learn J, but the fact that it covers all of the language concisely, yet completely and rigorously, with more emphasis on the complex than the mundane, does scare some of us away.

This book provides a kinder, gentler start for beginners. This book takes you along a path in easy steps to the point where you can write an application in J. Along the way you will be introduced to all the key ideas in J by seeing them in simplified and specific contexts. At the end, you will be able to write real programs in J, and you will also be comfortable in using the *J Dictionary* as a reference for your work as a J programmer. The purpose of this book is to get you up to speed where you can use the *J Dictionary* in a manner that makes you wonder why you ever bothered with this simple stuff.

You should be able to work your way through this book fairly quickly, and at the end you will be an entry-level J programmer. As such, you will have far more programming power at your fingertips than even the most experienced Basic or Java programmer.

Your background

This book assumes that you are familiar with another programming language such as Basic, Java, or C. However, this is not a prerequisite, and you shouldn't have particular problems if J is your first computer language (in fact, congratulations!).

It is also assumed that you are familiar with running Windows applications, in particular MDI (multiple-document interface) applications such as Word and Excel.

Most things can be done in J much as they are done in other languages, and in several areas a topic is introduced just as it would be introduced in other languages. If you are familiar with other languages this makes it easier to follow how it works in J. In some cases there is a much better *J way* to solve a problem, and that is also covered.

How to use this book

The book is a series of small, bite-size sections that are intended to be read in order from the start of the book to the end. Sections typically depend on most or even all of the previous sections having been read. Jumping around is pointless and likely frustrating.

The book is self-contained and could be read without access to a system. In particular, examples of interactions with the J system show both what you enter and how the system responds. However, it is intended to be read with access to a system and with as much use of a

system along with the book as possible. It is **strongly** recommended that you eventually type in all the examples and play around as much as you can with variations on them.

Sometimes a section uses terms and concepts that aren't defined until later. This requires you to proceed with a soft understanding the first time through that becomes more concrete on a second reading.

This book is probably best read by reading it three times:

- Skim the whole book. Try some examples, but it is better to just plow on and get the big picture.
- The second time read it carefully and try all the examples.
- The third time try your own examples to clarify your understanding and to increase your comfort with the mechanics of actually using the system (instead of just reading about it and following instructions).

Environment

This book assumes that you have installed and will be using J for Windows or J for Macintosh. Any edition at Release 4 or later is appropriate.

There are minor differences in J for Macintosh, but they should not be a problem in using this book to learn J.

J for UNIX is not as directly usable with this book. The J language is identical, so the differences are all in areas such as how to edit text files. With a reasonable knowledge of UNIX and some extra effort this book can be used in learning J on a UNIX system. Some sections deal with facilities that are not part of J for UNIX and you will just have to read those sections without being able to execute the examples or experiment. In particular, J for UNIX does not have a plot facility or a form designer.

Get started

Double-click the J icon to start J. Your J session should look something like this:

The window labeled 1.ijx is an execution window. You type J sentences into the ijx window and J executes them when you press enter and displays the result.

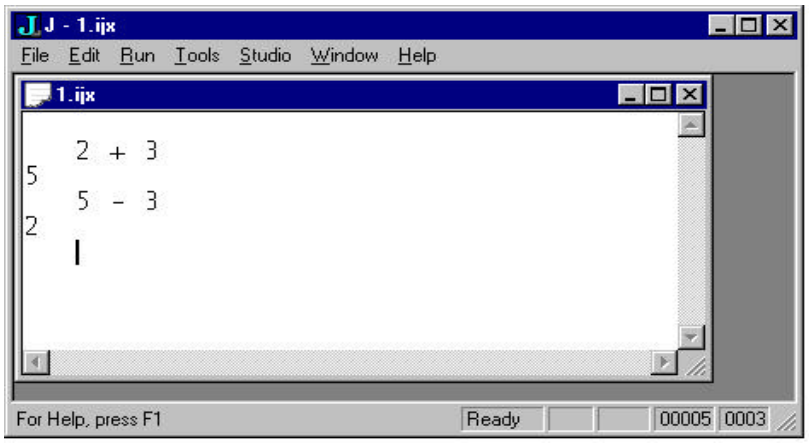
Type the following line into the ijx window and press Enter.

```
2 + 3
```

The sentence is executed and the result is displayed. Type the following line and press Enter.

```
5 - 3
```

Your session should look something like this:



Lines that you enter are indented three spaces and the J answers start at the margin.

Experiment

You are encouraged to experiment. Try entering similar lines with different numbers. It is clear that `+` is plus and `-` is minus. Enter lines that use `*` for times and `%` for divide. From using `%` you will pretty quickly see that numbers such as `2.5` can be the result, and that they can also be used as arguments.

Until you have more experience, you might sometimes be surprised or even disconcerted by what you observe. Take things in small steps. Try examples where you are pretty sure you already know the answer, and do the experiment to confirm your understanding. If a result puzzles you too much, don't spend time on it in these early stages.

```
   3 - 5
_2
```

The `_2`, instead of `-2`, might confuse you. Don't worry, it will be explained in a bit.
Most examples in this book show what you should type, indented by three spaces, and also show the result the system displays. This means that you can read the book in a casual manner, without having to use the system to see results. However, the only way you will really learn is by eventually trying the examples and experimenting with your own.

Examples are shown in a fixed-pitch font much as they would appear in the ijx window of your system. A larger font is used in some examples to make it easier for you to read and type the example into the system. This is done where you might mistype something because of being unfamiliar with some of the words, or where a typo could have confusing results.

While experimenting, you frequently want to execute minor variations on sentences you have already tried. There are several shortcuts that make this easier. In the ijx window you can move the cursor to any line in the window and press Enter to recall that line as a new line at the bottom of the window ready for editing. You can recall previous input lines for editing by holding down the Ctrl key and pressing the up arrow key until you see the line you want to work with.

The examples in most sections are self-contained, but a later part of a section might depend on steps taken in an earlier part. A few sections depend on steps taken in previous sections, but this should be fairly obvious.

Standard profile

The examples typically assume that your system runs the standard profile when it is started. This configures your system and makes some standard utilities available.

The standard profile is loaded by default. The start up option /noprofile prevents the profile from being loaded.

Enter CR (capital letter C followed by capital letter R) into the ijx window as a quick check on whether the profile has been run.

```
   CR
```

If the result is a blank line, then profile has been run and you can skip the rest of this section.

If instead you see:

```
    CR
value error: CR
```

you will have to change your J icon to remove the /noprofile option.

Terminology

All programming languages have things in common with the English language. Where the analogy is close, J tends to use English language terms in preference to terms used in math and other programming languages.

You could, as in other languages, say *line of code*, but in J you tend to say *sentence* instead. Similarly you could refer to the + *function*, but you usually say *verb*.

Some English language terms used in J are: alphabet, word, sentence, verb, noun, adverb, and conjunction.

There are several reasons for this approach. One problem it deals with is the plethora of related, but subtly different, uses of traditional terms in math and numerous programming languages. For example: function, subfunction, operator, program, routine, and subroutine are all used in slightly different ways in different programming languages. Rather than inherit this confusion, J adopts its own terms, and defines them precisely within its context.

Using English terms gives you a good idea of what the general meaning of the term is in J. In addition, using natural language terms encourages and facilitates taking the English statement of a problem and more directly writing the corresponding J sentences.

The use of the J terms is encouraged, but certainly isn't mandatory, and using the term function instead of verb is quite OK.

Alphabet

The J alphabet is the ASCII alphabet and consists of:

26 lowercase letters (a to z)
26 uppercase letters (A to Z)

```
0 1 2 3 4 5 6 7 8 9
= < > _
+ * - %
^ $ ~ |
. : , ;
# ! / \
[ ] { }
" ` @ & ?
( )
'
```

There are a few characters that sometimes cause confusion. The - (minus) character is different from the _ (underbar) character and there are three different quote characters:

- ' quote
- " double-quote
- ` back-quote

If you try an example and type " (a double-quote) instead of '' (two quotes) you will be disappointed that your result is not the same as in the book.

The . (period) is usually called dot.

Word

A word is a group of characters from the alphabet that has a meaning.

```
2.5 + 5
```

The sentence has three words: the number 2.5, the +, and the number 5.

The rules for forming words from the characters in a sentence are simple, and for now, common sense will suffice in recognizing the words in a sentence. There are some complications that will be dealt with in later sections.

Sentence

A sentence is a group of words that form a complete instruction. Unlike English sentences, J sentences do not end with a period or other punctuation mark. Instead a J sentence is usually a complete line.

Verb

In the following sentence the character + is a verb (a word that expresses an action).

```
2 + 5
```

Noun

In the previous sentence the numbers 2 and 5 are both nouns.

Number

The following are numbers: `0, 1, 2, 2.5, 12.75, 0.5, 7e6`.

Enter these, and other numbers, in simple sentences with the verbs `+ - *` and %.

```
   2 * 12.75
25.5
```

An important rule is that a number does not start with a dot.

```
   0.5
0.5

   0.5 + 3
3.5

   .5
+-+-+
|.|5|
+-+-+

   .5 + 3
|syntax error
|        .5+3
```

Clearly `.5` is not the same as `0.5`. You don't need to know now what `.5` is, but it is important to understand that a number does not start with a dot.

The _ (underbar) is infinity and is a number.

Negative number

A negative number is indicated by a _ (underbar), not a - (minus).

```
   5 + _3
2

   _7
_7
```

```
   _5e_3
_0.005

   5 - 9
_4
```

Leaving out the blank between the `-` and the `9` does not change the meaning.

```
   5 -9
_4
```

The `-` is always a verb. This simplifies the rules for evaluating sentences as there is no special case for `-` when it is used immediately to the left of a number. But if `-` is always a verb, then another character, the _ (underbar) is required to spell a negative number.

The _ is used in spelling numbers, along with the digits, dot, and the e for exponential notation; it indicates a negative number.

Remember: _ in front of a number is part of the number and indicates that it is negative, and `-` is always a verb and is not part of a number.

The __ (two underbars) is negative infinity and is a number.

Primitive

A primitive is a word that is defined by the system. For example, `+` is a primitive. The meaning of a primitive is fixed and cannot be changed.

A primitive is spelled with a graphic character (such as `+`) or with a graphic modified by an inflection (a dot or colon), as in `+.` or `+:`.

A primitive is also spelled by one or more letters followed by a dot or colon. For example, `i.` is a primitive that is called *index* or *index of* depending on how it is used.

Name

Whereas a primitive is a word defined by the system, a name is a word defined by you. The primitive `=.` defines a name.

```
   v =. 23
```

The sentence can be read as *v is 23*. The word `=.` is called copula (another good English language term). The name `v` is defined as the number `23` and can be used in other sentences.

```
   5 + v
28
```

Unlike a primitive, a name can be redefined.

```
   v =. 45
   5 + v
50
```

The system does not display the result of the sentence when it begins with a copula. A sentence that contains only a name shows a display form of the definition of the name.

```
   abc =. 123
   abc
123
```

You can give anything a name. For example, you could give a name to the verb `+`.

```
   plus =. +
   23 plus 45
68
```

The preferred way to read `abc =. def` is: *abc is def* or *abc is defined as def*. However, borrowing from other computer languages, it is also common to say: *abc is assigned the value def* or *the value def is assigned to abc*.

Comment

The primitive NB. starts a comment that runs to the end of the line.

```
   2 + 23     NB. this is a comment and is not executed
25
```

Error

An error in a sentence is reported and the execution stops.

```
   123 foo 234
|value error
|   123     foo 234
```

A vertical bar marks the error report. When appropriate, the sentence is displayed with extra spaces marking where the error was detected.

Ambivalence

Every verb has two definitions: a *dyad* used when it has both a left and right argument, and a *monad* used when it has only a right argument. The two definitions are usually related, as in `-` with a dyad definition of minus and a monad definition of negate.

```
   5 - 3
2
   - 7
_7
```

The dyad is also referred to as the dyadic case of the verb and the monad is referred to as the monadic case of the verb.

The term ambivalence is used in the chemical sense of *both valences* to indicate that a verb can react with both a single argument and with two arguments.

Remember: every verb has both a monad and dyad definition.

Dyad

The dyadic case of a verb is used if the verb has both a left and right argument.

The dyad `%` (divided by) is defined as *the left argument divided by the right argument*.

```
   5 % 2
2.5
```

Monad

The monadic case of a verb is used if the verb has only a right argument.

The monad % (reciprocal) is defined as *1 divided by the right argument*.

```
   % 2
0.5
```

The relationship between the monad % and dyad % where the monad is the dyad with a fixed left argument is quite common, and you will see this in a number of the other primitive verbs.

Vocabulary

The *J Dictionary* is **the** J reference book and is your ultimate, authoritative source of information on J. The sooner you become familiar with using it the better, and a good way to start is by looking up the primitives that have been introduced so far.

The back cover of the *J Dictionary* (or last page, depending on the edition) is called the vocabulary. It is a concise listing of all the primitives and gives a page reference to the detailed definition. Learning how to use the vocabulary is your key to learning how to read and write J.

If you don't have a printed copy of the *J Dictionary*, or if you prefer it, you can use the online version instead. Press F1 to show the vocabulary and click a primitive to jump to the entry. If you press F1 while holding down Ctrl you get context sensitive help. If the caret is at a +, pressing Ctrl+F1 will jump to the + entry.

In the vocabulary take a look at the row for + , which is the fifth row. The first entry in this row contains: Conjugate • Plus. The word + is a verb and its monad is called conjugate, and its dyad is called plus. The dyad + is plus as defined in arithmetic.

The monad + is interesting. In math, the conjugate of a complex number is the number with the same real part and an imaginary part of opposite sign. On real numbers the conjugate has no effect and the result is the argument, and on complex numbers it changes the sign of the imaginary part. J supports complex numbers just as directly as integers or real numbers. A complex number is indicated by a j separating the real and imaginary parts.

```
   int =. 23
   + int
23
```

```
   float =. 23.5
   + float
23.5

   imagine =. 2j3      NB. 2 real part, 3 imaginary part
   + imagine           NB. change sign of imaginary part
2j_3
```

Many primitives support complex numbers and the *J Dictionary* must document this, which means there is a bit of extra complexity in some of the descriptions. If you need complex numbers in your application this is fantastic. But if you are a beginner and are not concerned with complex numbers, then you have to know enough to be able to ignore these bits and not get distracted or confused.

The page number of the definition for + is in the rightmost column of the + row. Turn to that page now.

The header line of the page gives the monad name Conjugate on the left side, and the dyad name Plus on the right. The formal name of the primitive + is near the center. The 0 0 0 to the right of the + will be explained later.

Below this header are two boxes (or columns, in the online documentation) . The left box has the monad definition and the right box has the dyad definition. The page then continues with general discussion and examples.

The *J Dictionary* is a concise, rigorous, and complete reference suitable for the most experienced users. This can make it difficult for beginners who don't know what to expect. For example, in the general discussion and examples for + there is considerable discussion of complex numbers and not a single example of just adding a couple of integers. Great for the experienced user who would be insulted if told how to add integers, but a bit of a reading challenge for the real beginner. You have to learn how to tune out, for now, the bits that are too advanced or are not relevant to your current interest, and concentrate on the parts that are.

Go back to the vocabulary page and take a look at the +. entry (fifth row, second entry) which contains: Real/Imaginary • GCD (Or). There is a lot of information here, and again much of it is not relevant at this early stage. Let's look at the definition. The page number in the hardcopy manual is for the first entry in the row and the other entries are on the following pages.

Glance at the left box for the monad definition and notice that it is for complex numbers. File it for future reference, but give it a pass for now.

The dyad case in the header is described as: GCD (Or). This gives two informal names, GCD and Or, and indicates it can be used in two different ways. The dyad definition is in the right box. Note that GCD stands for *greatest common divisor* (which should at least ring a bell of math memories). Further on in the definition you will find that if the arguments are boolean then the `+.` is the logical or function. The GCD is a useful extension of the domain of the or function to non-boolean arguments. This extension of the domain of primitives is common in J. For now, it is interesting to note that `+.` has this larger domain, but it is also easy to limit it to boolean arguments.

```
   0 +. 0      NB. 0 or 0
0

   0 +. 1
1

   1 +. 0
1

   1 +. 1
1
```

The vocabulary page entry for `+:` contains: Double • Not-Or. The definition page, which follows the `+` and `+.` definition pages, gives quite simple definitions for both the monad and dyad.

The monadic case is called double and does just what you'd expect.

```
   +:  3
6
```

The dyadic case is the logical negation of the or of the arguments.

Again, for both `+.` and `+:` much of the general discussion and examples are perhaps beyond your capabilities right now. But the key is to know how to navigate and to get the information that is relevant.

After the row for `?` there are additional rows for primitives that are spelled with names that are inflected with a dot or colon.

Checkpoint A

At this point you should understand:

- how to use the *J Dictionary* vocabulary
- terms such as word, sentence, noun, verb, ambivalence, dyad, monad

Check your understanding by doing the following exercises:

- look up the definition of the monads `+: *: -: %:` in the *J Dictionary*
- experiment with these new monads

Numeric constant

You have seen the use of single numbers. It is also possible to have a list of numbers.

```
   num  =. 5
   nums =. 23 0.5 12.5 7e6 _12 7
```

You'll do lots more with numeric lists, but for now we just want to establish that there is such a thing.

String

Characters bracketed by quotes create a string.

```
   char  =. 'Q'
   chars =. 'this is a list of characters'
```

The quotes are not displayed when the string is displayed.

```
   char
Q
   chars
this is a list of characters
```

The quote starts and ends a string. A pair of quotes indicates that the quote character itself is in the string.

```
   'put 2 ''s to get 1 '' in the string'
put 2 's to get 1 ' in the string
```

An unmatched quote is an error.

```
   abc =. 'asdf
|open quote
|   abc =. 'asdf
|          ^
```

A string can be empty.

```
   abc =. ''
```

A string is also referred to as a literal constant or as a character constant.

Word formation

The monad `;:` can be useful in figuring out what the words are in a sentence. The word formation primitive takes a string as its right argument, splits it into words, and returns a result with each word in a box. For now, don't worry about what the boxes are, just note how visually helpful they are. You'll learn about boxes in later sections.

```
   ;: '2 + 3'
+-+-+-+
|2|+|3|
+-+-+-+
```

If you don't see the boxes when you try this on your system, then you are using a font that does not have the line drawing characters. Use the **Edit|Configure...** menu command to select a font that does show the boxes. Alternatives are ISIJ, Terminal, MS LineDraw, Courier New, and Lucida Console. Some require OEM and some Default to get the boxes. When you get one you like, be sure to check Save Config so that it will be used the next time you start J.

```
   ;: '2.5 + 3e4'
+---+-+---+
|2.5|+|3e4|
+---+-+---+
```

```
   ;: 'a =. 1 2 3'
+-+--+-----+
|a|=.|1 2 3|
+-+--+-----+

   ;: 'test + 123    NB. this is a comment'
+----+-+---+--------------------+
|test|+|123|NB. this is a comment|
+----+-+---+--------------------+

   ;: 'def =. ''testing 1 2 3'''
+---+--+---------------+
|def|=.|'testing 1 2 3'|
+---+--+---------------+
```

Note that the following are all J words and each goes in its own box:

```
2.5
3e4
=.
1 2 3
test
NB. this is a comment
'testing 1 2 3'
```

It might surprise you that constants such as `1 2 3` and `'testing 1 2 3'` are J words. This is an important point and understanding it is necessary in reading and writing J sentences.

If you are ever puzzled by a J sentence (it could happen), one of the things you can do is apply `;:` to it to be sure you know the words. You can then worry about the meanings of those words.

Look up `;:` in the *J Dictionary*. The informal name for `;:` is word formation. Turn to the page for `;` and turn pages until you come to the page that has `;:` in its heading.

Space

Spaces are not required to separate primitives from other words. On the other hand extra spaces don't change the meaning.

```
   2+3
5
   2 + 3
5
   2    +    3
5
```

Most examples in this book have spaces around all primitives. This makes the individual words stand out and allows you to concentrate on the meaning of the words without the additional problem of first figuring out what the words are.

However, most J programmers, as they become more experienced, reach a point where they can easily read the words in a sentence, and the extra spaces become a nuisance and hindrance to understanding, rather than an aid. You will notice that in some of the later examples that some of these unnecessary spaces are left out.

There are some cases where a space is essential.

A space must separate names.

```
   a=.3
   plus=.+
   a plus a
6

   aplusa
|value error
```

The `.` (dot) and `:` (colon), used as inflections, change the word immediately in front of them into a new word. When used as a primitive or as the start of a primitive, they must have a space in front so that they are not treated as an inflection.

```
   ;: 'a . b'       NB. 3 words
+-+-+-+
|a|.|b|
+-+-+-+
```

```
   ;: 'a .b'       NB. same 3 words
+-+-+-+
|a|.|b|
+-+-+-+

   ;: 'a. b'       NB. 2 words
+--+-+
|a.|b|
+--+-+
```

A space must separate a `.` or `:` , that is not being used as an inflection, from the previous word.

Numbers, for example `1e7`, can contain letters. There are in fact several letters that can be used in spelling numbers in J. Letters that immediately follow a number are treated as part of the spelling of the number.

```
   plus =. +
   1plus 3
|ill-formed number

   1 plus 3
4
```

A space must separate a number from a letter that is not a part of the number.

Precedence

Math traditionally gives multiplication precedence over addition. In math class (or a skill testing question from a cereal box contest), if you were asked what 2 + 3 * 4 was, you would know the answer was 14.

This is not too confusing if there are only a few functions and only a few levels of precedence (division | multiplication | addition and subtraction). But it gets awkward in languages such as C which have many functions and many levels of precedence.

With the large number of verbs in J it would have been difficult to define the precedence, let alone trying to remember it when reading or writing. Moreover, being able to name things means you would also have to figure out what to do with the sentence:

```
   2 plus 3 times 4
```

In a break with traditional math and in contrast to most other programming languages, J has no verb precedence. It will take you a little while to stop doing the multiplication first, but the overall simplification is worthwhile.

Remember: there is no verb precedence.

Parentheses

Math and most programming languages, including J, use parentheses to control the evaluation of a sentence. If a sentence is *fully parenthesized* then the order of evaluation is identical in most languages and is independent of verb precedence or any other rules.

```
   2 + (3 * 4)
14

   (2 + 3) * 4
20

   10 - (4 - 3)
9

   (10 - 4) - 3
3
```

There isn't any confusion about these answers.

Order of evaluation

What is the answer if the parentheses are left out?

```
   10 - 4 - 3
9
```

J evaluates the sentence as:

```
   10 - (4 - 3)
9
```

Most other languages would evaluate it as:

```
   (10 - 4) - 3
3
```

In the absence of explicit parentheses, J implicitly provides them from the right towards the left. Other languages provide them from the left towards the right. A longer sentence will make this visually clearer.

```
   10 - 4 - 3 - 1
8

   10 - (4 - (3 - 1))         NB. J right-to-left
8

   ((10 - 4) - 3) – 1         NB. others left-to-right
2
```

Now consider a sequence of monadic verbs.

```
   - - - 4
_4
```

Everyone knows how to parenthesize this, and every language does it the same.

```
   - (- (- 4))
4
```

The grouping is done right-to-left and in this case the other languages agree with J. J always parenthesizes from right-to-left, whereas other languages have different rules for different situations.

J has a right-to-left order of evaluation. Most other languages have a left-to-right order of evaluation for dyads, right-to-left for monads; and this is modified by the relative precedence of the verbs involved.

With nouns and verbs the J evaluation rule from *J Dictionary* section E is:

> Execution proceeds from right to left, except that when a right parenthesis is encountered, the segment enclosed by it and its matching left parenthesis is executed, and its result replaces the entire segment and its enclosing parentheses.

There are things in J, other than nouns and verbs, that you have not yet met that complicate this rule by adding a few more. It is these additional classes that largely justify the J break with tradition and adoption of a right-to-left evaluation.

To further quote from the *J Dictionary* section E:

> One important consequence of these rules is that in an unparenthesized expression the right argument of any verb is the result of the entire phrase to its right.

This is due to the lack of verb precedence as well as right-to-left evaluation.

No verb precedence, right-to-left evaluation, and the rules for the other classes make the overall evaluation rules simple, reduce the need for parentheses, and make sentences easier for an experienced J user to read and write.

Read the following sentences, evaluate them in your head, and understand how the no precedence and right-to-left rules explain the answer.

```
   2 * 4 + 5
18

   2 + 4 * 5
22

   2 - 4 - 5
3

   8 % 2 + 2
2
```

Remember: no verb precedence and right-to-left evaluation.

Verb definition

The art of programming lies not so much in using the primitives, as in defining your own verbs, tailored to your requirements. In defining your own verbs you are extending the language to build an application that solves a particular set of problems.

Let's assume that the problem is to convert temperatures between Fahrenheit and centigrade. You need to define a verb that does that.

The following is a definition of the verb centigrade that will convert its argument from a Fahrenheit value to a centigrade value.

```
   centigrade =. 3 : 0
t1 =. y. - 32
t2 =. t1 * 5
t3 =. t2 % 9
)
```

The font size in the above has been upped a bit to make it easier for you to type it exactly into the ijx window. At this point you want to do something that works, rather than deal with problems arising from typos, so transcribe it carefully.

In the ijx window enter the first line:

```
   centigrade =. 3 : 0
```

Type it exactly as shown. There must be a blank between the `3` and the `:` . The `3` indicates that you are defining a verb and the `0` indicates that the definition is in the subsequent input lines.

After you enter the above line the cursor is at the left margin and has not been indented the three spaces as it normally is. This indicates that the system is waiting for you to enter the rest of the definition.

Type in the lines following the definition of centigrade as shown. They are at the left margin, and so look like they might have been displayed by the system, but in fact they are your entries of the lines required to define the verb.

The final line that contains just the `)` ends this special definition input mode. After you enter this final line, the system again indents the three spaces indicating that it is ready to execute a sentence.

If you entered the definition correctly, you should be able to experiment with your new verb.

```
   centigrade 32
0

   centigrade _40
_40

   centigrade 212
100
```

Let's look at the definition to understand how it works. The `y.` in the first sentence of the definition you type is the name of the argument of the verb. When you execute the verb with an argument the first line will subtract `32` from the argument and define `t1`. When the first line is finished, execution proceeds to the next line, which defines `t2` as the result of `t1` times `5`. Execution proceeds to the next line and defines `t3` as `t2` divided by `9`. There are no more lines, so the execution of the verb is finished. The result of the verb is the last result that was evaluated.

We used to define the verb. The phrase is equivalent and some find it easier to read. However, it hides information and we will use the form.

Monad and dyad definition

As discussed in the earlier section on ambivalence, all verbs had two definitions, a monad and a dyad. You have defined only a monad for centigrade. What about the dyad?

```
   23 centigrade 32
|domain error
|   23     centigrade 32
```

Since you didn't provide a dyad definition, it is empty and this is treated as if the dyad had no arguments in its domain, and any arguments you give will cause a domain error.

Let's examine some simple examples of defining dyadic, monadic, and both cases.

```
   monadminus =. 3 : 0
- y.
)

   monadminus 5
_5

   5 monadminus 3
|domain error
|   5     monadminus 3
```

The above defines the monad of the verb named monadminus. Applying it monadically works and applying it dyadically fails.

In one-line definitions like this you can take a shortcut and make the definition on a single line and avoid entering the special input mode that needs to be ended with the `)`. The following is an equivalent way of doing the above definition:

```
   monadminus =. 3 : '- y.'
```

The string contains the single line that makes up the definition. It is provided directly as the right argument of `:` instead of the `0` used earlier.

So far you have defined just the monadic case of a verb. You can also define a verb with just a dyadic definition. Instead of 3 as the left argument to `:` use a 4 to define the dyadic case.

```
   dyadminus =. 4 : 'x. - y.'
   5 dyadminus 3
2

   dyadminus 5
|domain error
|       dyadminus 5
```

In the monad case the `y.` name is the right argument and in the dyad case `x.` is the left argument and `y.` is the right.

What if you want to define both cases of a verb?

```
   minus =. 3 : 0
- y.
:
x. - y.
)
```

The `:` by itself on a line separates the monad and dyad definitions.

```
   3 minus 5
_2

   5 minus 3
2

   minus 5
_5
```

Script file

When you close J you lose the definitions of all the names. What you execute in the ijx window affects the current session, but is not permanent. This is fine when experimenting, but when you start defining things like your centigrade verb you want to record the definition so that you can use it in another session.

Close J and restart it.

```
   centigrade
|value error
```

You have a clean slate. The definition of centigrade, and all the other names you defined, in the previous session are lost.

At least the primitives are still there!

```
   2 + 5
7
```

As you would expect, to maintain a permanent record of your definitions, you save them in files. Files with J sentences and definitions are called script files and you can edit them just as you would edit any other text file. Script files typically have a suffix of .ijs.

Remember: a script file is a source file for definitions.

Although you can use any text editor to work with script files, the J system provides a simple editor that is integrated in ways that make it convenient.

The **File|New IJS** menu command creates a new script file and a window for editing it. Do this now and you will see that your J session has both an ijx window and a new ijs window. Use **Window|Tile Across** so that you can see them both side by side.

The ijs window is an edit window on the file with the name in its titlebar. Enter in an ijs window does not execute the line, it just moves to the start of a new line.

Type your centigrade definition into the ijs window.

```
centigrade =: 3 : 0
t1 =. y. - 32
t2 =. t1 * 5
t3 =. t2 % 9
)
```

Be sure to use `=:` instead of `=.` in the first line. The makes a global definition. If you use it is a local definition. This important difference is explained shortly.

Because this is an ijs window the system has not provided a three space indent for the first line.

So far you have just edited changes into the window. The file has not been changed and the verb is still not defined. You have to *run* the script in order to execute the sentences.

With the ijs window active (titlebar highlighted), run it with **Run|Window**. Running the window with **Run|Window**, saves changes that have been edited in the window to the file, and then executes each of the sentences in the file. This is similar to your typing the contents of the file into the ijx window, except the sentences and results are not displayed. The only display in

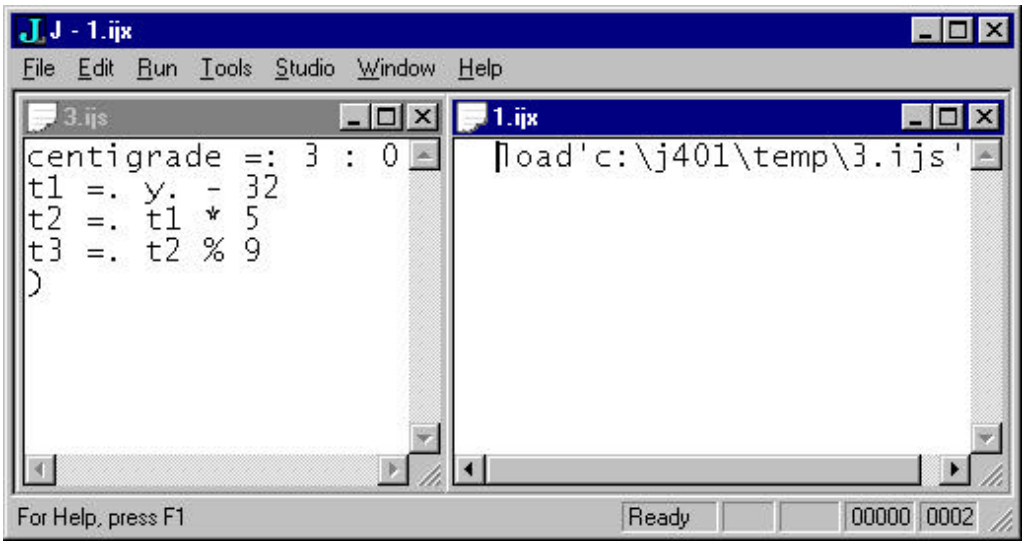
the ijx window is the sentence that causes the file sentences to be executed. This sentence will be something like: `load'c:\j4\temp\1.ijs'`

If an error is reported (output in the ijx window with a vertical bar on the left) then you have a typo in your script. Correct the text in the ijs window and run it again.

The sentences in the script file have been executed and centigrade is now defined. In the ijx window try using centigrade.

```
   centigrade 32
0
```

Your screen should look something like:



The file created with **File|New IJS** is in the TEMP directory and has a temporary format name (a number with an ijs suffix). If you close J now, it will ask if you want to delete that temporary file. If you replied no, you could restart and open that temporary file and run it to define `centigrade`. However, it would be better to resave it now with a more appropriate name in the USER directory. Use **File|Save As...**, change to the USER directory, and set the file name as cf.ijs. The file name in the ijs window titlebar will change to the new name.

Close the cf.ijs window and erase your centigrade definition. You erase the definition of a name by using the utility verb `erase` with an argument that is the string of the name you want to erase. The result of 1 indicates the erase was successful.

```
   erase 'centigrade'
1
   centigrade 212
|value error
|       centigrade 212
```

Use **File|Open** to open the cf.ijs window and use **Run|Window** to run the script to define `centigrade`.

```
   centigrade 212
100
```

Let's add a definition for `fahrenheit` to the cf.ijs window. Type in the following after your centigrade definition. **Again, be sure to use `=:` .**

```
fahrenheit =: 3 : 0
t1 =. y. * 9
t2 =. t1 % 5
t3 =. t2 + 32
)
```

Use **Run|Window** to run the sentences in the cf.ijs script. Because these are the first changes to a permanent (non-temporary) file you are prompted to see if you want to save the changes to file. Reply yes, and then test your new verb.

```
   fahrenheit 0
32

   fahrenheit 451
843.8
```

Close J and restart it.

```
   centigrade
|value error
   fahrenheit
|value error
```

You can run the sentences in the cf.ijs file without opening the file for editing. Use **Run|File** and select your cf.ijs file. A line similar to `load'c:\j4\user\cf.ijs'` appears in the ijx window to run the sentences in the file.

```
   centigrade 0
32
```

```
   fahrenheit 212
100
```

The line that starts with load that appears in the ijx window is in fact the sentence that causes the sentences in the file to be executed. The menu command is just a short cut way of executing this sentence. The string is the full path name to the file to run. You can shorten this full path name to a relative path name when you type it manually.

To check this, close J, restart it, and verify that centigrade is undefined. In the ijx window execute the following sentence.

```
   load'user\cf.ijs'
```

Now check that your verbs are defined.

Use File|Open to open your cf.ijs file for editing.

What if there is an error in the script? Let's add an intentional error to the script to see what happens. Add the line `foo 123` at the end of the script and run the script again.

```
   load'c:\j4\user\cf.ijs'
|value error
|       foo 123
|[-13]
```

An error is reported and the execution of the sentences in the script stops. The number at the end of the error report is the line number in the script that had the error. The statusbar shows the line number in the script and you can use this to find the error in the script.

Remove the error from the script and run it again.

Local

The verb `centigrade` uses names `t1`, `t2`, and `t3` in its definition, but if you refer to them outside the verb they are not defined.

```
   centigrade 0
32

   t1
|value error

   t1 =. 123
   t1
123
```

```
   centigrade 100
212

   t1
123
```

The use of `t1` inside the definition of `centigrade` has not conflicted with your use of `t1` outside the definition. The verb `centigrade` does not define a `t1` outside of itself, as indicated by the value error, and setting a value into its `t1` does not change the value of `t1` outside the definition.

The `t1` used inside `centigrade` is a *local* name. A local name exists only inside the verb. The `t1` used outside `centigrade` is a *global* name. A name defined in the execution of a verb with the copula `=.` is a local name.

Global

A name defined outside the execution of a verb is a *global* name.

In the previous section, the `t1` defined in the ijx window is a global name that is completely different from the `t1` defined inside the verb centigrade.

Let's try some experiments. Create a temporary script file with **File|New IJS** and type into it the definition:

```
fooa =: 3 : 0      NB. =: is important
zzz + y.
)
```

Run the script with **Run|Window**. In the ijx window:

```
   fooa 5
|value error
|       zzz+y.
```

Let's define the global zzz to see what happens. Defining it outside a verb means it is a global. In the ijx window:

```
   zzz =. 23            NB. define global zzz
   fooa 5
28
```

The verb `fooa` uses the global `zzz`. So, a verb can use globals.

Edit the script to add `foob` and then run the script.

```
foob =: 3 : 0
zzz =. 7
zzz + y.
)
```

In the ijx window:

```
   foob 3
10

   zzz
23
```

The verb `foob` uses its local `zzz` and ignores the global. So, a verb can use locals and ignore globals of the same name.

Inside a verb the copula `=.` defines a local name. Once a name is defined as a local, references to that name are to the local name.

What if you wanted to define a global name? The global copula `=:` (= with a colon inflection) defines a global name. Edit the script to add `fooc` and then run the script.

```
fooc =: 3 : 0
gw =: y.
lz =. y.
)
```

In the ijx window:

```
   fooc 3
3

   gw
3
```

```
   lz
|value error
   gw =. 24
   fooc 5
5

   gw
5
```

Defining `gw` with `=:` defines the global name.

In general, it is good practice to only define locals in a verb and to not define globals. This is an important part of what is sometimes called a *functional style* of programming. Verbs that define globals are said to have *side effects* and are more likely to cause bugs and make it harder to read the application to understand what is happening.

It is possible to define a verb that uses both the global and local definitions of a name. With few exceptions this is VERY bad practice.

Debug global

Sometimes when trying to debug or better understand a verb it is useful to see the values of its local names or other intermediate results. A quick way of doing this is to add a line to the verb definition that does a global definition.

Open the cf.ijs file and add a line to `centigrade` to define global `gt1` as `t1`.

```
centigrade =: 3 : 0     NB. =:
t1 =. y. - 32
gt1 =: t1               NB. temp line for debugging info
t2 =. t1 * 5
t3 =. t2 % 9
)
```

Run the script and in the ijx window:

```
   centigrade 124
51.1111

   t1
|value error
```

```
   gt1
92
```

After `centigrade` finished execution you can't see what value the local `t1` had, but you can see a copy of the value in `gt1`.

Remove the line from the script and run the script to redefine `centigrade` without the debug line.

When =. and =: are the same

You have seen how `=.` and `=:` are different when used in a verb definition.

When you execute sentences in the ijx window you are not executing them inside a verb so the `=.` and `=:` have the same effect. In the ijx window:

```
   a =. 123
   a
123

   a =: 234
   a
234
```

In the ijx window the `=.` and `=:` copulas are the same and it doesn't matter which you use to define a name as they both define the global name. The `=.` is easier to type and tends to be the one that is used. In a strict sense it would be better to explicitly use `=:` when defining a global name.

When they aren't

You have seen that `=.` and `=:` in the ijx window are the same. And you have seen that inside a verb they are different. It is important to realize there is also a difference in scripts.

When you run a script, the load sentence is executed in the ijx window and the verb load executes the sentences in the script. So, the sentences in the script are executed in the load verb. This means that names defined with `=.` are defined as locals of the verb load. If you want to define a global in the script you must use `=:` . This is why the lines which define globals such as centigrade and fahrenheit in your script cf.ijs must use `=:` . If you used `=.` , they would be local to load and would disappear as soon as load finished execution.

Always think about whether a definition is global or local and use `=:` and `=.` accordingly.

Locale

First of all, note that the term locale is very different from local.

A locale is a set of global names. There can be several locales, so there can be several sets of global names.

A global name in a locale is distinguished from the same name in other locales by qualifying the name with the addition of the locale name bracketed by _ (underbar) characters. A name qualified by a locale is always a global name.

```
   abc_def_ =: 2
```

The above sentence can be read as *global abc in locale def is 2*.

```
   abc_base_ =: 4
```

The above sentence can be read as *global abc in locale base is 4*.

If the locale name is elided, it is assumed to be base.

```
   abc__   NB. the same as abc_base_
4
```

If a global name is not qualified with a locale name, then it is in the *current locale*. The base locale is the current locale unless it has been explicitly changed by executing a verb in a different locale. The following defines `abc` in the base locale:

```
   abc =. 6
   abc_base_
6
   abc
6
```

Since the base locale is the current locale, the names `abc` and `abc_base_` are the same.

The name `abc_def_` is clearly different from `abc`, but so far there is no way of telling that anything special is going on. In what sense are `abc` and `foo` in the same (base) locale? And `abc` and `abc_def_` in different locales?

One way of distinguishing is to use the `names` utility verb that lists global names.

```
   a =. 23
   b =. 24
```

```
   a_q_ =. 25
   w_q_ =. 26
   names 0         NB. 0 lists nouns
a abc b
```

Your `names` result may be different, but it will include all global nouns you have defined in the base locale. You should see the `a` and `b` that you defined above and note that you do not see the `w` that was defined in locale `q`.

To see the names defined in locale `q` you can do the following:

```
   names_q_ 0      NB. names in locale q
a w
```

Nouns `a` and `w` are defined in the `q` locale.

Locales partition global names into different sets, and utilities, such as `names`, can work with globals in a particular locale.

The real power of locales comes into play with verbs defined in a locale. When a verb executes *in a locale* it executes with that locale, not the base locale, as the current locale.

Let's define a simple verb in the `q` locale to see how this works.

```
   f_q_ =. 3 : 'a =: y.'
```

This verb defines global `a` with its right argument. There can be many different locales, each with their own global `a`. But when `f_q_` executes, it executes in the `q` locale and the `q` locale is the current local, and global names it uses are from the `q` locale. Try the following experiments:

```
   a =. 23     NB. define a in the base locale
   a_q_ =. 24  NB. define a in locale q
   f_q_ 100    NB. execute f in locale q
100
   a
23
   a_q_
100
```

Executing `f_q_ 100` defined global `a_q_` as `100`. It did not affect the global `a` in the base locale.

If a verb explicitly references a name in a locale then that is the global that is affected. For example, define verb `g_q_` that defines `a` in the base locale. You will see that the `a` in the base locale is defined and the `a` in the `q` locale is not changed:

```
   g_q_ =. 3 : 'a_base_ =: y.'    NB. explicit locale name
   g_q_ 200
200
   a
200
   a__
200
   a_q_
100
```

Locales partition global names into separate sets. In particular, related nouns and verbs, say in a set of utilities, can be defined in their own locale. Their names don't conflict with names in the base or other locales. When you look at your application you can look at just the related globals that are in a particular locale. When a verb runs in a locale it uses globals from that same locale.

The `names` verb with an argument of `0` lists nouns, with `3` it lists verbs, and with `6` it lists locale names.

```
   names_q_ 0
a w

   names_q_ 3      NB. verbs
f g

   names 6         NB. locale names
base j q z
```

The list of locale names is interesting. and you know about, but what about and ?

The globals in the `j` and `z` locales are defined when J starts up and runs the profile.ijs script. The `j` locale contains things which are useful in building an application and is discussed in the *J Online Documentation*.

The `z` locale is very interesting indeed.

z locale

The z locale is the *parent locale* of all other locales.

If a name is not found in the current locale, and there is a definition for it in the z locale, then that definition is used **as if it were in the current locale**.

The z locale is for common utilities that you want to be available everywhere. From the z locale, they are available for execution in any locale as if they were in that locale, yet there is only a single copy, and the names in the z locale don't clutter up the names in the other locales.

The profile.ijs that runs when you start J defines many standard utilities in the z locale. You have used both the erase and the names verbs which are defined in the z locale. You can tell this by the following:

```
   names 3       NB. verbs in the base locale
...
```

The above does not list names as a name, yet you are able to execute it. This is because when it is not found in the base locale, its definition from the z locale is used as if it were in fact defined in the base locale.

```
   names_z_ 3    NB. verbs in z locale
...
```

The result is too long to list here. The verb names has a dyadic definition that takes a left argument which indicates the first letter of names to return.

```
    'n' names_z_ 3
nameclass namelist names nc nl
```

names is defined in the z locale and that is the definition that is executed.

Script load

In addition to the utilities loaded with the standard profile, there are several additional scripts of standard utilities provided with the system. These standard utilities are documented in the *J Online Documentation* available from the J help menu. You could run these scripts directly, but you would need to remember the path to the script, as well as which locale to run them in. The standard profile provides utilities to make this easier for you. The scripts verb lists scripts that can be loaded with the load verb.

```
   scripts ''
. . .
parts    plot     profile  scripts  stdlib   strings
trig     validate winlib   winutil
...
```

The `scripts` verb with an argument of `'v'` lists the scripts with their full path and locale.

```
   scripts 'v'
. . .
compare  c:\j4\system\main\compare.ijs        z
convert  c:\j4\system\main\convert.ijs        z
 . .
```

The convert script contains conversion utilities.

```
   load 'convert'

   [n=. av 'Sir Richard'
83 105 114 32 82 105 99 104 97 114 100

   av n
Sir Richard
```

Checkpoint B

At this point you should understand:

- a text file that is a source of sentences is called a script file
- a script file defines global names
- how to create a new temporary script file
- how to save a temporary script file as a permanent file
- how to run a script file to execute its sentences
- how to define a verb in a script file
- how to define the monadic and dyadic cases of a verb
- the difference between `=.` and `=:`
- the difference between local and global
- that a locale is a set of global names
- that there can be more than one locale
- that the base locale is the one you normally work with

Check your understanding by doing the following exercises:

- create a new temporary script file
- in the script define `square` as a monad that uses `*:` to square its argument
- save the script in the user directory with the name square.ijs
- run the script and test the verb square
- close J, restart, use **Run|File** to run user\square.ijs and test it

Debug - stepping through a verb

In an earlier section you added a debugging line to a verb definition that allowed you to see the results of intermediate steps when the verb was run. Sometimes you need more powerful tools than that.

Use `load` to load the debug utilities.

```
   load 'debug'
```

Open your script file cf.ijs and run it.

Let's execute `centigrade`, but with a stop on each line so that you can take a look at exactly what is going on.

```
   dbss 'centigrade *:*'
```

The `dbss` (Set Stop) argument requests a stop before executing all, indicated by `*:*`, lines in `centigrade`.

```
   dbr 1
```

`dbr` with an argument of `1` requests that the system suspend execution when an error or stop occurs. When a verb is suspended it is halted in mid execution. You can examine definitions, change definitions, and you can resume execution of the suspended verb.

```
   centigrade 212
|stop
|       t1=.y.-32
|centigrade[0]
```

The error report (bars at the left margin) indicates execution stopped on line 0 of `centigrade` and shows the sentence from that line.

The execution of `centigrade` is suspended and the indent of six spaces, rather than three, indicates the suspension. The variable y. is the argument.

```
      y.
212
```

The stop occurs before the line is executed, so `t1` has not been defined and if you try to look at it you will get a value error.

Use `dbrun` to continue execution. It will run the current line, and because stops are set on all lines it will then stop on the next line.

```
      dbrun ''
|stop
|       t2=.t1*5
|centigrade[1]

      t1
180

      t1*5
900
```

`centigrade` is now stopped on line 1, and as you can see, you are able to check the value of local `t1` that was defined in line 0. Step through the next lines and examine locals.

```
      dbrun ''
|stop
|       t3=.t2%9
|centigrade[2]

      t2
900

      t2%9
100

      dbrun ''
100
```

You are no longer suspended in `centigrade` and you are back to the normal indent of three spaces.

Turn off the request for debug suspensions and reset to have no stops.

```
   dbr 0
   dbss ''
```

Debug - an error

Let's introduce an error into your `centigrade` verb to see how that looks and how you would find and fix it.

Open your cf.ijs script and edit the first line to have an error by adding quotes around the expression to the right of the copula `.`

```
t1 =. 'y. - 32'
```

Instead of `t1` being defined as the result of `y. - 32` , it will be defined as the string `'y. - 32'` .

Run the script to make the new definition. Turn off debug suspension and request no stops and then run your buggy `centigrade`. Be sure to load the debug utilities if they are not already loaded.

```
   dbr 0   NB. disable suspension
   dbss ''
   centigrade 212
|domain error
|   t2=.t1    *5
```

You are executing with suspension disabled (`dbr 0`) so execution did not suspend in `centigrade` and you have the normal 3 space indent.

If you look at the line in error it is clear that the `5` is a valid argument to times, so there must be something wrong with `t1`. But you don't know the value of `t1`. You could stare at the source for the error, but, in a complex situation, it might be quicker to use debug.
Enable suspension and rerun.

```
   dbr 1   NB. enable suspension
   centigrade 212
|domain error
|   t2=.t1    *5
|centigrade[1]
```

There is a 6 space indent indicating suspension, and because `centigrade` is suspended you can look at the value of `t1`.

```
      t1
y. - 32
```

From the display of `t1` it is clear that it is a string, not the number from the desired calculation. You can now look at the source to see where `t1` was defined and see that the quotes should not be there.

Edit the source to fix the definition by removing the quotes and run the script to redefine `centigrade`.

You want to run line 0 again to properly define `t1`. You can do this by using `dbjmp` to continue execution at line 0.

```
      dbjmp 0
100
```

Since no stops are set and there are no other errors, line 0 of `centigrade` is executed, which sets a proper value into local `t1` and execution continues until finished.

Comparative

The dyad = has a result of 1 if its left and right argument are equal, and a result of 0 if they are different.

```
   23 = 34
0

   23 = 23
1

   a =. 'd'
   a = 'c'
0

   7 + a = 'c'
7

   7 + a = 'd'
8
```

Some programming languages treat the results of comparative primitives such as = as True and False values that are not numbers. In J the results of comparatives are just numbers.

There are several other comparative verbs: less-than < , less-or-equal <: , larger-than > , and larger-or-equal >: . These comparative primitives are sometimes called relationals.

```
   7 < 8
1

   7 < 7
0

   7 <: 7
1
```

Control structure

In `centigrade` the sentences in the definition are just executed sequentially, one after the other. To conditionally control which sentences are executed you use control structures.

Control structures are built with control words and sentences. The following is an example of a control structure:

```
if. x. = 'c'
 do. centigrade y.
 else. fahrenheit y.
end.
```

The `if.` control word starts the control structure and the `end.` control word ends it. The result of the `x. = 'c'` sentence is the test result and it determines which of the other sentences in the control structure are executed. If the test result is `1`, then the sentence after the `do.` control word is executed. If the test result is any other value then the sentence after the `else.` control word is executed. In English: if the left argument equals the letter c, then execute `centigrade`, otherwise execute `fahrenheit`.

Use this capability to add a new verb to your cf.ijs script that will convert a number from Fahrenheit to centigrade or from centigrade to Fahrenheit depending on the value of the left argument. Open your cf.ijs script and add the following definition at the end.

```
NB. convert f to c if x. is 'c', otherwise c to f
convert =: dyad : 0
if. x. = 'c'
 do. centigrade y.
 else. fahrenheit y.
end.
)
```

This defines the dyadic case of the verb. The dyad has a left argument with the name `x.` and a right argument with the name `y.`. The verb `convert` takes a left argument of `'c'` to convert a Fahrenheit value to centigrade. Any left argument other than `'c'` will convert a centigrade value to Fahrenheit.

Note that you use your verbs `fahrenheit` and `centigrade` just as you would use primitive verbs.

Run the script and test `convert`.

```
   'c' convert  212
100

   'f' convert 100
212
```

Normally a sentence is a line in a script. However, with control words separating a line into several sentences it is possible to have more than one sentence on a line.

The following line is equivalent to the multiple lines used earlier.

```
if. x. = 'c' do. centigrade y. else. fahrenheit y. end.
```

Control structures are only allowed in definitions and you cannot type one directly into the ijx window for execution.

There are nine control structure patterns:

```
      if. T do. B end.

      if. T do. B else. B1 end.

      if. T do. B elseif. T1 do. B1 elseif. T2 do. B2 end.

      try. B catch. B1 end.

      while. T do. B end.

      whilst. T do. B end.

      for. T do. B end.

      for_i. T do. B end.
```

```
select. T
  case. T0 do. B0
  case. T1 do. B1
  fcase.T2 do. B2
  case. T3 do. B3
end.
```

A control structure starts with `if.` , `try.` , `while.` , `whilst.`, `for.` , `for_i.` , or `select.` and ends with a matching `end.` .

Words beginning with T or B denote a block of 0 or more sentences and can contain nested control structures.

The result of the last sentence in a T block determines which block is executed next and whether execution in the control structure is finished.

Often the T block is a single sentence that makes a simple test like the one in the example.

The `try.` control structure is an interesting one. It executes the B block of sentences, and if there are no errors it skips to the end of the structure. However, if there is an error, then the B1 block is executed.

The `while.` control structure executes the T block and if its result is not 0 then it executes the B block and continues this until the T block has a 0 result. If the T block is 0 the first time, then the B block is not executed.

The `whilst.` control structure is the same as `while.` except that the T block is skipped the first time. This means that the B block is always executed at least once.

See the *J Dictionary* for more information on control structures.

Checkpoint C

At this point you should understand:

- that `load 'debug'` loads debug utilities
- the general idea of verb debugging
- how control words create control structures by grouping sentences into blocks
- what the T block test result is
- how the test result determines which B block to execute
- how the test result determines when control structure execution is finished

Check your understanding by doing the following exercises:

- debug step through your `convert` verb
- create a temporary script file and define a verb called `conv` that is similar to `convert`, but insists on a `'f'` argument to do the conversion to Fahrenheit and gives a string result indicating there was an error if the left argument is neither `'c'` nor `'f'`. Hint: use the control structure sketched out here:

```
if. x. = 'c' do. ...
  elseif. x. = 'f' do. ...
  elseif. 1 do. 'left arg not c or f'
end.
```

  or try a structure.

- create a temporary script file and define a dyad called plus that adds its left argument to its right. But, if there is an error, it should give a string result. Hints: use dyad : 0; `4 plus 9` should return `13`; `'a' plus 9` should return your error string (perhaps, `'there was an error'`); use a `try.` control structure to catch the error and give the string result.

Basic way of adding lists

You can have lists of numbers.

```
a =. 12 24 47
b =. 12 34 45
```

If you wanted to add two lists of numbers in a language like Basic you would have to get each number in turn from each list, add them together, and then stick this new result at the end of the result list.

To add two lists of numbers together in this way you need a few new primitives. You need a way to get one number from a list. The verb `{` (from) can do this.

```
   0 { 7 9 2 4    NB. index 0 value
7
   1 { 7 9 2 4    NB. index 1
9
   2 { 7 9 2 4    NB. index 2
2
   3 { 7 9 2 4    NB. index 3
4
```

You need to be able to append a new result value to the result list. The verb , can do this.

```
   7 9 2 , 4     NB. append 4 to the list
7 9 2 4

   7 9 2 4 , 7   NB. append 7 to the list
7 9 2 4 7

   a =. 7 9 2 4
   a =. a , 23
   a
7 9 2 4 23
```

You need to know how many numbers there are in the list so that you will know when you are done. The monad # (tally) tells us how many numbers are in the list.

```
   # 7 9 2 4
4
   # 7 9 2 4 7
5
```

You also need a way to create an empty result to which you will add each new result. An empty string will do this.

```
   r =. ''       NB. an empty string
```

With these new verbs, combined with what you already know, you can write a Basic or Java style program that adds two lists.

Create a temporary script file and add the `addlists` definition.

```
addlists =: dyad : 0
r =. ''
count =. # x.
i =. 0
while. i < count do.
 left =. i { x.
 right =. i { y.
 sum =. left + right
 r =. r , sum
 i =. i + 1
end.
r
)
```

The local `i` is the index to select numbers from each list. It starts with `0` to select the first number from the left and right arguments. At the end of the `while.` control structure the `i` is incremented by `1` so that the next time the block is executed it will select the next number. The `while.` structure tests to see if `i` is less than the count of the argument. The control structure is finished when `i` is equal to the count of numbers to be added together. The `left` and `right` locals are defined as the next pair of numbers. They are added together and are appended to the end of the result `r`. The final line in the definition is `r` and that is the result.

Run the script and test your definition of `addlists`.

```
   2 3 4 addlists 4 5 6
6 8 10
```

If you made a typo in the definition you will get an error or a wrong answer. In that case, you should check carefully that you have typed the definition in correctly.

Certain errors (such as omitting the line that incremented the value of `i`) give you a `while.` that runs forever, and the statusbar indicates running and you won't get any result displayed. This is because the `while.` never ends and the program keeps adding the first element of the left and right arguments and never steps to the next element. If you are in a loop like this, press Ctrl+Break to interrupt the execution.

In fact, it is worthwhile seeing how this looks. Edit the `addlists` definition so that `i` is not incremented. The easiest way to do this is to add `NB.` in front of the `i =. i + 1` sentence. Run the script and test the verb. You should see that the statusbar indicates running and that there is no result. Press Ctrl+Break to stop the execution.

For such a simple thing, this definition seems overly verbose in taking eleven lines. The definition can be compacted a bit by combining sentences. In the temporary script file create a second version of the definition.

```
adda =: dyad : 0
r =. ''
count =. # x.
i =. 0
while. i < count do.
 r =. r , (i { x.) + (i { y.)
 i =. i + 1
end.
r
)
```

Run the script and test this new version.

```
   2 3 4 adda 4 5 6
6 8 10
```

This is essentially how programmers in most languages add two lists of numbers. The program could be further streamlined, but it would still have to be a control structure that dealt with each number, one at a time. Most languages only know how to add a single number to a single number, and to add lists of numbers, you need to write a control structure that loops and explicitly adds each element of the list in turn.

J way of adding lists

J knows how to do things to single numbers, but it also knows how to do things with lists. Since J knows how to add lists, you can write a third, simpler version of the definition.

```
addb =: 4 : 'x. + y.'
```

Add this definition to your temporary script file, run the script, and test it.

```
   2 3 4 addb 4 5 6
6 8 10
```

At this point you probably realize that addb is so simple as to be unnecessary.

```
   2 3 4 + 4 5 6
6 8 10
```

In J you can just add the lists of numbers because the + verb knows all about lists of numbers.

Years of research and thought have gone into how J verbs work with lists. For example, if you wanted to add 1 to each number in a list.

```
   1 adda 2 3 4
3
```

The Basic style `adda` verb gives an answer, but it is the wrong answer. What happens is that the `while.` uses the count of the left argument (which is 1) to determine how many elements to generate in the result, and so calculates only the first result number.

```
   2 3 4 adda 1
|index error
|   r=.r,(i{x.)+(i    {y.)
```

This gives an error because the `while.` uses a count of 3 (the count of numbers in the left argument) but the right argument doesn't have that many so you get an error.

But the J + handles both these cases the way you would like, and would expect!

```
   1 + 2 3 4
3 4 5

   2 3 4 + 1
3 4 5
```

Thank goodness the `addlists` and `adda` verbs are in a temporary file and are easy to get rid of, because clearly you don't need them in J. Close and delete that temporary script now.

The simple concept of working with lists, instead of just single things, extends throughout J.

```
   2 4 6 * 7 8 9
14 32 54

   2 * 2 3 4
4 6 8

   2 3 4 * 2
4 6 8

   5 6 7 % 2
2.5 3 3.5

   2 3 4 - 3
_1 0 1
```

It works for the comparatives as well.

```
   2 3 4 = 7 3 8
0 1 0

   2 < 0 1 2 3 4 5
0 0 0 1 1 1

   't' = 'testing'
1 0 0 1 0 0 0
```

If this works for primitives, what about verbs such as `centigrade`?

Run your cf.ijs script to define your verbs and see what happens.

```
   centigrade _40 32 212
_40 0 100
```

You can apply your `centigrade` verb to a list of numbers and get a list of results. This also work for `fahrenheit`.

```
   fahrenheit _40 0 100
_40 32 212
```

What about your dyad convert?

```
   'c' convert _40 32 212
_40 0 100

   'f' convert _40 0 32 100 212
_40 32 89.6 212 413.6
```

The extension of verbs to work consistently on lists is very powerful and significantly distinguishes J from most other languages. It is important that you assimilate this into the way you think about solving problems.

A few more primitives

The monad `i.` (integers) result is the argument-length list of integers from `0` up to one less than the argument.

```
   i. 2
0 1

   i. 4
0 1 2 3

   i. 6
0 1 2 3 4 5
```

The dyad `$` is called shape.

```
   5 $ 7   NB. a list of 5 7's
7 7 7 7 7
   8 $ 23  NB. a list of 8 23's
23 23 23 23 23 23 23 23
```

The monad ? (roll) generates a random number in the range 0 up to 1 less than the argument. The answers vary depending on how the die rolls.

```
   ? 10
6

   ? 10
0
   ? 10 10 10 10      NB. 4 numbers in range 0 to 9
3 0 4 6

   ? 5 $ 100          NB. 5 numbers in range 0 to 99
58 93 84 52 9
   ? i. 5
|domain error
|       ?i.5
```

The `? i. 5` fails because the `?` on `0` fails because there are no integers in the range `0` to `_1` and so there is no answer for the argument of 0. If you add `1` to the result of `i.` we will get an answer.

```
   ? 1 + i. 20
0 1 0 1 0 3 3 2 2 4 7 3 0 13 1 5 16 6 3 19
```

The dyad ^ (power) result is the left argument multiplied by itself the number of times given by the right argument.

```
   3 ^ 2              NB. 3 * 3
9

   2 ^ 3              NB. 2 * 2 * 2
8

   2 ^ 4              NB. 2 * 2 * 2 * 2
16

   2 2 2 2 2 2 ^ 0 1 2 3 4 5
1 2 4 8 16 32

   2 ^ 0 1 2 3 4 5
1 2 4 8 16 32

   2 ^ i. 6
1 2 4 8 16 32
```

The dyad `o.` (circle) is the letter o inflected with a dot, and it provides the circular (trigonometric) functions. In particular, the `o.` verb with a left argument of `1` gives the sine of the right argument.

```
   1 o. i.7
0 0.841471 0.909297 0.14112 _0.756802 _0.958924 _0.279415
```

It is hard to tell whether this makes sense or not and it would be better to see this data with a plot.

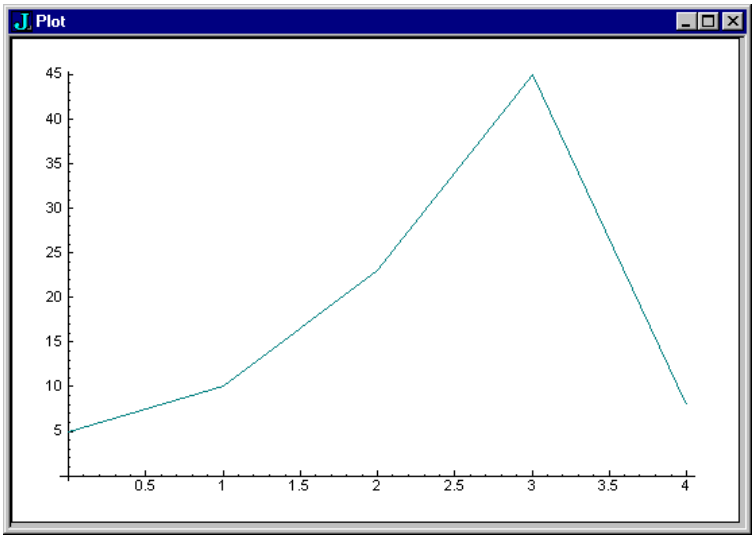
Plot

To use the plot facility you need to load it.

```
load 'plot'
```

Try a simple plot.

```
plot 5 10 23 45 8
```



The plot is in a separate window that stays on top of your session. Close the plot window to get rid of it.

Now that you can plot data, let's take a look at some of the data you were generating in the previous section.

```
   plot 2 ^ i. 5
   plot ? 5 $ 100
   plot ? 50 $ 100
   plot ? 100 $ 100
   plot ? 1 + i. 50
   plot ? 1 + i. 100
```

A left argument customizes the plot.

```
   'TITLE myplot;TYPE bar' plot i.5
```

Or try the sine values you calculated earlier.

```
   plot 1 o. i. 16
```

There is a family of utilities defined in script trig.ijs you could make use of here. Use `load` to load that script.

```
   load 'trig'
   plot sin i. 16
```

To produce a finer plot you need to provide more results over a similar range.

```
   plot sin 0.2 * i.60
   plot cos 0.2 * i.60
```

Plot locale

Plot also illustrates a common technique in the use of locales. Close J and restart it to get a clean slate. The `names` in the base locale is empty and there are just the two standard locales that are populated by the profile.ijs script.

```
   names 3           NB. verbs

   names 6           NB. locales
base j    jcfg z
```

The load verb (defined in the z locale) loads the plot system into the jwplot and jzplot locales.

```
   load 'plot'
   plot ? 5 $ 100
   names 3

   names 6
base  j   jwplot  jzplot  z
   plot
+------------+
|plot_jwplot_|
+------------+
```

The verb `plot` is not defined in the base locale, but is defined in the `z` locale. When it is executed in the base locale the definition from the `z` locale is executed as if it were in the base locale. Entering the name `plot` displays its definition. The interesting thing about the definition of `plot` is that it executes the `plot` verb in the `jwplot` locale. So the `plot` in the `z` locale is a cover that works in all locales and has the result of executing `plot` in the `jwplot` locale.

This technique of loading a facility like the plot package into its own locale, and then defining cover verbs in the `z` locale so that plain name references invoke the desired verb in the facility locale is common. The definitions in the `jwplot` and `jzplot` locales can be viewed as the *private* implementation of the facility and the names that are *exposed* by being defined in the `z` locale are the *public* or *published* interface.

Print precision

Print precision is the number of digits shown when a number is displayed.

```
   1 % 3
0.333333

   10 % 3
3.33333
   100 % 3
33.3333
```

Using lists and some of the new primitives you can now see this more concisely:

```
   (10 ^ i. 6) % 3
0.333333 3.33333 33.3333 333.333 3333.33 33333.3
```

You can guess that the default print precision is 6 because in each case 6 digits are shown.

```
   a =. 1e5 + 10 % 3
   b =. a + 0.1
   a
100003

   b
100003

   a = b
0
```

With only 6 digits shown, a and b look the same even though they aren't. In a situation like this you need to see more digits. The print precision can be changed by use of the following verb.

```
   pps =. 9!:11       NB. print precision set
```

Don't worry about the curious appearance of this verb, just use it.

```
   pps 9

   a
100003.333

   b
100003.433
```

With print precision of 9 there is enough detail to see what is going on.

```
   pps 6

   %3 9 13
0.333333 0.111111 0.0769231

   pps 12

   %3 9 13
0.333333333333 0.111111111111 0.0769230769231
```

The default print precision of 6 is adequate for most situations because you don't usually have to see all those extra digits of detail. However, it is important to know that they really are there, and that the display has been abbreviated as a matter of convenience.

Inexact numbers

The way numbers are stored in a computer limits the maximum number of digits in the number. This maximum depends on the hardware, but typically a `pps` argument of `20` guarantees that all the digits available for a value will be displayed.

```
   pps 6

   %3
0.333333
   pps 12

   %3
0.333333333333
   pps 20

   %3
0.33333333333333331
```

At `20` digits of precision the result of `%3` displays all the detail it has on the number in `17` digits. The result of `%3` is not the exact mathematical result, but is the closest number to that exact result that can be stored in the computer. This difference between what you would expect from exact math and the limitations on how numbers are stored in computers can be confusing.

```
   3 * % 3
1
```

The result of `%3` is not the exact value, but it is so close that when multiplied by `3` it gives the exact expected value of `1`.

```
   3 * 10 * % 3
10
```

Multiplying the inexact result of `%3` by `10` magnifies the error, but it is still close enough to the exact value that when multiplied by `3` it gives the exact expected value of 10.

```
   3 * 100 * % 3
99.999999999999986
```

However, multiplying `%3` by `100` magnifies the error enough so that when multiplied by `3` you do not get the exact expected answer of `100`, but instead get a number that is very close to `100`.

Using lists you can combine the above examples.

```
   3 * 1 10 100 * % 3
1 10 99.999999999999986
```

The fact the `%3` isn't stored exactly in the computer may not surprise you too much if you realize that an exact representation in decimal digits would take an infinite numbers of 3's.

There is an additional source of confusion due to the fact that computers store numbers internally in a binary format where each digit is a 0 or 1, rather than the decimal format you are familiar with where the digits range in value from 0 to 9. A consequence of this is that even very simple decimal numbers, exactly expressed with a few digits, when converted to the computer's binary format, are stored as an inexact value.

```
   pps 20

   0.5 0.25 0.1
0.5 0.25 0.10000000000000001
```

The `0.5` and `0.25` are stored exactly, but the `0.1` is stored inexactly, and when displayed with maximum precision shows as `0.10000000000000001 .`

These are facts of life with the way computers store floating point numbers and apply to all computer languages, not just J. Usually you can ignore these details, but they can sometimes cause problems or confusion if you don't have an idea about what is going on.

Tolerance

This section is a bit advanced and understanding it is not critical. If it makes sense, great. If not, don't worry about it, and just move on to the next section.

For some kinds of work with floating point numbers, this section is important, along with a more detailed understanding of how numbers are stored and manipulated by the hardware. For most work, however, this section can be ignored.

Let's consider the calculation at the end of the last section in more detail.

```
   pps 20

   a =. 3 * 100 * % 3
   a
99.999999999999986
   a = 100
1
```

By exact math you would expect `a` to be `100`. But because the computer can't exactly represent the value `%3`, you get a value for `a` that is very close to `100`, but not exactly, as you can see by its detailed display with a print precision of `20`. However, note that `a` is considered to be equal to `100`, even though you can see that it is not exactly equal. This is because the comparison is *tolerant*. That is, numbers do not have to be exactly identical to be considered equal.

Let's experiment to get an idea for how tolerant the comparison is by gradually taking the value further away from 100. The input line recall shortcut with Ctrl+up arrow is very useful for playing with things like this.

```
   100 = 100
1

   100 = 99.999999999999986
1

   100 = 99.99999999999998
1

   100 = 99.9999999999999
1

   100 = 99.999999999999
1

   100 = 99.99999999999
0
```

In the last example you crossed the line and the value is far enough away from 100 that it is no longer considered to be equal. Let's look at another example.

```
   a =. 23
   b =. a - 1e_12
   c =. a - 1e_11
   a
23

   b
22.999999999999002
   c
22.999999999989999

   a = b
1
```

```
   a = c
0
```

The values of a and b are close enough to be considered equal. The values of a and c are not close enough to be considered equal. Close enough refers to the difference between the two numbers.

```
   a - b
9.9831254374294076e_13

   a - c
1.000088900582341e_11
```

In both cases the difference is small, but b is closer than c to a. Reading the *J Dictionary* definition for = you will see that the dividing line between close enough and not close enough is determined by the result of multiplying the larger of the numbers times the default tolerance value of 2^_44. That is, close enough is *relative* to the size of the numbers.

```
   tolerance =. a * 2 ^ _44
   tolerance
1.3073986337985843e_12
```

Check both differences against this tolerance:

```
   (a - b , c) <: tolerance
1 0
```

The difference between a and b is less or equal to the tolerance, whereas the difference between a and c is not.

Checkpoint D

At this point you should understand:

- primitives work with lists
- your own verbs work with lists
- how to use several new verbs
- how to use the plot facility
- comparatives such as = that give numeric 0 and 1 results

Check your understanding by doing the following exercises:

- look up the *J Dictionary* definitions of the integers, shape, roll, power, and circle verbs; in most cases only a part of their capabilities have been introduced, so you will have to read the definitions carefully to be able to ignore the parts not yet relevant, and to pick out the parts that are
- experiment with the new primitives

Atom

A noun that is a single entity is an atom.

```
23
'a'
b =. 23
a =. 'q'
```

An atom has 0 dimensions.

List

A noun that is a list of atoms is a list.

```
23 24 25
'this is a string'
b =. 7 14 21
a =. 'another string'
```

A list has one dimension. The length of the dimension is the count of atoms in the dimension.

Table

A table is a two dimensional array of atoms. Tables cannot be written directly as a constant as can an atom or a list, but instead must be created with a primitive. The dyad $ (shape) can create tables. The left argument indicates the count of items in each dimension and the right argument provides items to populate the table.

```
   2 3 $ 7       NB. a 2 by 3 table of 7's
7 7 7
7 7 7
```

```
   2 3 $ 7 8 9 10 11 12
 7  8  9
10 11 12

   2 3 $ 7 8    NB. cycle through atoms to get enough
7 8 7
8 7 8

   3 4 $ 'abcdefghijklmnopqrstuv'
abcd
efgh
ijkl
```

The monad `$` (shape of) gives the shape of its argument. The shape is the list of the count of atoms in each dimension of the argument.

```
   a =. 2 3 $ 7
   $ a
2 3
```

The monad `i.` (integers), introduced earlier for creating a list of integers, can be used to create tables of integers.

```
   i.5          NB. list of 5 integers
0 1 2 3 4

   i. 2 3       NB. 2 by 3 table of integers
0 1 2
3 4 5
```

Array

Atoms, lists, and tables are all arrays. All nouns in J are arrays. Atoms have 0 dimensions, lists have 1 dimension, and tables have 2 dimensions. This extends to higher dimension arrays.

The primitive $, discussed earlier with lists and tables also works with higher dimension arrays.

```
   2 3 4 $ i. 24
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23
```

The above is a 3-dimensional array. The blank line indicates the break between the 1st dimension and the 2nd and 3rd.

```
   b =. 2 3 4 $ 'abcdef'
   $ b
2 3 4
```

The monad `i.` also works with higher dimension arrays.

```
   i. 2 3 4
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23
```

The J terms atom, list, and table are analogous to the math terms of scalar, vector, and matrix.

Axis

The term axis is used in slight preference to the term dimension. Atoms have 0 axes, lists have 1 axis, and tables have 2 axes.

Shape

The shape of a noun is the list of the count of atoms in each of its axes. The monad $ gives the shape of a noun.

```
   a =. 4 2 3 $ 'abcdef'
   $ a
4 2 3

   $ 4            NB. atom has 0 axes and its shape is empty

   $ i. 5
5
```

Rank

The rank of a noun is the count of its axes. An atom has rank 0, a list rank 1, a table rank 2, and an array with 5 axes has rank 5. The rank of an array is very important and determines in a significant way how verbs act upon it.

The shape of an array is a list with as many numbers as the array has axes. This means that the count of the shape of an array is the rank of the array.

```
   # $ 4              NB. atom has rank 0
0
   # $ 4 5 6          NB. list has rank 1
1
   # $ 2 3 $ 'a'      NB. table has rank 2
2
```

Empty Array

An array is empty if it contains no atoms. An empty array has a `0` in its shape.

```
   a =. 0 $ 0         NB. empty list
   $a
0

   $ ''               NB. empty list
0

   b =. 2 0 3 $ 'a'   NB. empty rank 3 array
   $b
2 0 3
```

Empty arrays have no atoms to display and their display can be confusing if you don't know what to expect. An empty list displays as a blank line. A table with 0 rows displays as 0 lines; a table with 3 rows, but 0 columns, displays as 3 blank lines.

```
   2 $ 5
5 5

   1 $ 5
5

   0 $ 5          NB. empty list displays as blank line

   2 2 $ 5
5 5
5 5

   1 2 $ 5
5 5

   0 2 $ 5        NB. 0 rows displays 0 lines
   2 0 $ 5        NB. 2 rows displays 2 lines
```

The display of lists and tables with 1 row can look the same, and you have to look at their shapes to distinguish them.

```
   a =. 2 $ 5
   b =. 1 2 $ 5
   a
5 5
   b              NB. b displays the same as a
5 5
   $ a
2
   $ b            NB. but b has a different shape
1 2
```

Single atom array

An array with a single atom is referred to as a *singleton*. All singletons with the same atom display the same way. However, the fact that they have different ranks affects how verbs act on them. This can be a pitfall for beginners. It is important to remember that if it displays like an atom, but does not behave like one, then check its rank.

```
   atom =. 5
   list =. 1 $ 5
   atom
5

   list               NB. list looks like atom
5

   atom + 23 23 23
28 28 28

   list + 23 23 23    NB. but does not behave like atom
|length error
|   list    +23 23 23
   # $ atom           NB. rank of atom is 0
0

   # $ list           NB. rank of list is 1
1
```

Verb arguments

Much of the power of J lies in the ability of a verb to treat its arguments as a series of parts. The verb applies itself to each of the parts, creating a series of partial results, and then assembles the partial results into the final result. Exactly how this works and what you can do with it is described in the next several sections.

Let's look at a few examples to get an idea of where you are heading.

```
   m =. i. 2 2
   m
0 1
2 3
```

You can add arrays together that have the same rank and shape.

```
   m + 2 2 $ 10 11 12 13
10 12
14 16
```

You can add a single number to an array.

```
  10 + m
10 11
12 13
```

What if you wanted to add one number to the first row and a different number to the second row?

```
   10 20 + m
10 11
22 23
```

But what if you wanted to add those numbers to the columns instead? You have to indicate that you want to add to the columns not the rows.

```
   10 20 +"1 m
10 21
12 23
```

Frame and cell

So far nouns have been considered in their entirety. However, it is useful to think of an array as consisting of cells, parts of the array (subarrays) that when placed in a frame, make up the entire array.

```
   a =. 2 3 $ i. 6
   a
0 1 2
3 4 5
```

The array `a` can be thought of as having 6 cells, where each cell was an atom. The frame would be the shape `2 3` that structures the 6 individual cells into the array `a`. Visually:

cells are atoms

`0`	cell 0
`1`	cell 1
...	
`5`	cell 5

frame is shape `2 3` that structures the cell atoms into the array

The array `a` can also be thought of as having 2 cells, where each cell was a list. The frame would be the shape `2` that structures the cells into the array `a`. Visually:

cells are lists

`0 1 2`	cell 0
`3 4 5`	cell 1

frame is shape `2` that structures the cell lists into the array

Finally, the array `a` can be thought of as having 1 cell, where the cell was a table. The frame would be the shape empty that structures the cells into the array `a`. Visually:

```
cells are tables
0 1 2       cell 0
2 3 4
```

frame is shape empty that structures the cell table into the array

A table with shape `2 3` can be thought of as:

- a `2 3` frame of cells that are atoms
- a `2` frame of cells that are lists of shape `3`
- an empty frame of a cell that is a table of shape `2 3`

Similarly, an array with shape `4 3 2` can be thought of as:

- a `4 3 2` frame of cells that are atoms
- a `4 3` frame of cells that are lists of shape `2`
- a `4` frame of cells that are tables of shape `3 2`
- an empty frame of a cell that is a rank `3` array of shape `4 3 2`

The frame is a prefix of the shape of the array. It can be the entire shape (a prefix of all), in which case the cells are atoms. It can be empty (a prefix of none) in which case there is a single cell which is the array. Or anything in between.

The cell shape is the array shape with the frame prefix removed. The length of the cell shape is the *cell rank*.

The cells of an array are the subarrays that, when assembled into the corresponding frame, create the entire array.

Item

Arrays are frequently treated as having a frame of length 1. With this frame, the array has cells of rank 1 less than the rank of the array. These cells are the *items* of the array.

The items of a list are the atoms in the list. The items of a table are the rows in the table. The items of a rank 3 array are the tables in the array. An array is the list of its items.

An atom has one item, itself.

The # (tally) of a noun is the number of items in the noun.

```
   # 23
1

   # 1 $ 5
1

   # i. 5
5

   # i.2 3
2
```

k-cell

A cell of rank k is also called a rank-k cell or k-cell. A 0-cell is an atom, a 1-cell is a list, a 2-cell is a table, and so on. If the rank of the cells of a noun is given, then the frame is whatever is left over of the shape of the noun.

Negative numbers are also used, as in _2-cell and _1-cell; the frames of such cells have length indicated by the magnitude of the numbers. You have seen _1-cells before: they are items.

```
  abc =. 4 3 2 $ i. 24
```

The noun `abc` can be thought of as:

- a `4 3 2` frame of 0-cells
- a `4 3` frame of 1-cells
- a `4` frame of 2-cells
- an empty frame of a 3-cell

A more general way of phrasing this is:

- a rank 3 frame of 0-cells
- a rank 2 frame of 1-cells
- a rank 1 frame of 2-cells
- a rank 0 frame of 3-cells

Verb rank

A verb has a rank that determines how it applies to its arguments. A monad of rank k applies to the k-cells of its argument. A dyad of left rank kl and right rank kr applies to the kl-cells of its left argument and the kr-cells of its right argument. Verb rank is a powerful tool that controls the way a verb applies to arrays.

The ranks of a primitive verb are given in the *J Dictionary* definition. For example, look up the definition of + . The rank information follows the word in the header. For + this is `0 0 0`. The monad rank is 0 which indicates the monad + applies to the atoms , or 0-cells, in its argument. The dyad ranks are 0 for the left argument (indicating it applies to the atoms in its left argument), and 0 for the right argument (indicating it applies to the atoms in the right argument).

Let's see how this works when adding two tables.

```
   a =. i. 2 3
   b =. 6 + a
   a
0 1 2
3 4 5

   b
6  7  8
9 10 11

   a + b
 6  8 10
12 14 16
```

The dyad + has left rank 0. This means it applies to the atoms of its left argument. Similarly the right rank is 0 and it applies to the atoms of its right argument. The verb takes an atom from its left argument, an atom from its right argument, and adds them together to create a partial result. It does this for each atom from the left and right argument and creates an appropriate number of partial results, which are then assembled into the result frame to create the final result.

In the example above the verb + has a left rank of 0. This means the left argument is treated as a `2 3` frame of atoms. Similarly, a right rank of 0 means that the right argument is treated as a `2 3` frame of atoms.

The frame of the result is determined by the frames of the arguments, and so its frame is also `2 3` and each cell is the result of adding an atom from the left argument with an atom from the right argument.

Agreement

For a dyad the left rank of the verb and the rank of the left argument determine the frame of the left argument. Similarly the right rank of the verb and the rank of the right argument determine the frame of the right argument. If the left and right frames are the same, then there are the same number of cells in each argument, and it is simply a matter of taking each cell in turn from the left and right arguments, applying the verb, and putting the result into the frame of the result.

```
   a =. i. 2 3
   b =. 2 3 $ 7
   a + b
 7  8  9
10 11 12
```

Visually you can see how each atom from the left is used with the corresponding atom from the right.

```
0 1 2   +   7 7 7   gives   7  8  9
3 4 5       7 7 7          10 11 12
```

You have also seen that the following works.

```
   a + 7
 7  8  9
10 11 12
```

Visually you can see how each atom from the left is used with the corresponding atom from the right.

```
0 1 2   +   7 ...      gives   7  8  9
3 4 5       ...               10 11 12
```

The ... indicates that the cell is repeated to provide the required arguments. The ... to the right and below the 7 indicates it is repeated in 2 axes.

But what about the following?

```
   a + 3 4
3 4 5
7 8 9
```

Again you can see how the cells of the right argument repeat to provide the required verb arguments.

```
0 1 2   +   3 ...      gives    3 4 5
3 4 5       4 ...               7 8 9
```

But there must be some agreement between the cells in the arguments.

```
   a + 3 4 5
|length error
|   a    +3 4 5
```

Visually what is happening:

```
0 1 2   +   3 ...      gives    3 4 5
3 4 5       4 ...               7 8 0
            5 ...      error - ran out of lefts
```

The above cases are simple enough, but consider the following with a rank 3 noun.

```
   b =. i. 2 3 4
   b + a
 0  1  2  3
 5  6  7  8
10 11 12 13

15 16 17 18
20 21 22 23
25 26 27 28
```

This is more complicated to visualize.

```
 0  1  2  3   +   0 ...   gives   0  1  2  3
 4  5  6  7       1 ...           5  6  7  8
 8  9 10 11       2 ...          10 11 12 13

12 13 14 15       3 ...          15 16 17 18
16 17 18 19       4 ...          20 21 22 23
20 21 22 23       5 ...          25 26 27 28
```

Similarly:

```
   b + 2 3
 2  3  4  5
 6  7  8  9
10 11 12 13

15 16 17 18
19 20 21 22
23 24 25 26
```

Visually:

```
 0  1  2  3   +   2 ...   gives  2  3  4  5
 4  5  6  7       ...            6  7  8  9
 8  9 10 11                     10 11 12 13

12 13 14 15       3 ...         15 16 17 18
16 17 18 19       ...           19 20 21 22
20 21 22 23                     23 24 25 26
```

The agreement rule is quite simple. If the left and right frames are the same then there is no problem. Otherwise, one frame must be a prefix of the other, and its cells are repeated into its trailing axes to provide the required arguments.

Rank conjunction "

The primitive " (double-quote, not two quotes) is the rank conjunction. Conjunctions haven't been introduced yet and there is more detail in a later section. For now, just think of a conjunction as similar to a dyad verb in that it takes a left and right argument and has a result. The particular use of " of interest here is when the left argument is a verb and the right argument is a noun. Yes, conjunctions can take verb arguments, as well as noun, whereas a verb can take only noun arguments.

In the section on names there was an example where you directly defined a name as a verb.

```
plus =. +
```

This style of definition is more direct than the type you used to define `centigrade`. It is called tacit definition and is dealt with in more detail in a later section. The name `plus` is defined as the primitive + and thus has the same rank as + of `0 0 0` .
The rank conjunction produces a new verb from its left argument with the rank information from its right argument.

```
plus000 =. + " 0 0 0
```

The right argument for " is the rank information for the primitive + that is given in the *J Dictionary* (look up + in the vocabulary, turn to the definition page, and note the rank information in the heading). The first 0 is the rank of the monad argument. The second and third 0's are respectively the rank of the dyad left and right arguments.

Since `plus000` is + with same ranks as the primitive + it should behave just as does + or `plus` . You can verify this with a few experiments borrowed from the previous section on agreement.

```
   a =. i. 2 3
   a plus000 a
0 2  4
6 8 10

   a plus000 1 2 3
|length error
|   a   plus000 1 2 3
```

The length error occurs because the arguments do not agree as per the previous section. The left frame is `2 3` and the right frame `3`, and `3` is not a prefix of `2 3`; there are extra cells from the left argument without corresponding cells from the right argument.

However, it seems reasonable to want to add the list 1 2 3 to each list in the left argument. You know what you want it to do. Visually:

```
0 1 2   +   1 2 3      gives   1 3 5
3 4 5       ...                4 6 8
```

You want a variation of + that adds lists from its left argument to lists from its right. You can do that by changing the arguments to the " conjunction to indicate that the dyad left and right ranks are lists.

```
   plus011 =. + " 0 1 1
   a plus011 1 2 3
1 3 5
4 6 8

   1 2 3 plus011 a
1 3 5
4 6 8
```

In practice you wouldn't bother to give a name to such a specific application of + and you would instead use the expression directly.

```
   1 2 3 +" 0 1 1 a
1 3 5
4 6 8
```

Since + is applied dyadically and both ranks are 1, you can use the shorter form of `+"1` which uses `1` for the rank of all arguments.

```
   1 2 3 +"1 a
1 3 5
4 6 8
```

In this case, the left frame is empty with a cell shape of `3` and the right frame is `2` with a cell shape of `3`. Empty is a prefix of `2`, and so the frames agree.

There is one thing you have to be aware of.

```
   a +"1 1 2 3
|length error
|   a    +"1 1 2 3
```

The problem is that J doesn't know that you want the first `1` to be the argument to `"` and the second `1` to be part of the constant `1 2 3`. What happens is that the constant `1 1 2 3` is used as the right argument of `"` and since `"` is defined to allow only arguments of 1 2 or 3 numbers, there is a length error. You need to let J know that the `1` belongs to the `"` and that the `1 2 3` is a constant.

```
   a (+"1) 1 2 3
1 3 5
4 6 8

   a +"1 (1 2 3)
1 3 5
4 6 8
```

Result shape

In the previous sections the question of the shape of the result was glossed over. For a monad the frame of the result is the same as the frame of the argument. For a dyad the frame of the result is the frame of the longer of the frames of the arguments (or either frame if they are the same).

With a verb like + that has an atom result for each atom argument this is straightforward. Things get more interesting with verbs that have more complicated behavior.

Consider the verb $. Look it up in the *J Dictionary* and you'll see it has rank of _ 1 _ . The _ indicates an infinite (unbounded) rank and means that the verb applies to the entire argument. The monad has unbounded rank and so applies to the entire right argument. If you think about the monad $ with a result that is the shape of its entire right argument this makes sense. The dyad left rank is 1 and this means that it applies to lists from the left argument. The dyad right rank is unbounded and so applies to the entire right argument.

```
   2 4 $ i.3
0 1 2 0
1 2 0 1

   2 4 $"1 0 i.3
0 0 0 0
0 0 0 0

1 1 1 1
1 1 1 1

2 2 2 2
2 2 2 2
```

The first example is what you have seen before, but what is going on in the second? The $"1 0 means that $ will get cell arguments as a list (1-cells) on the left and as an atom (0-cell) on the right. The left frame is empty (nothing is left of the shape of the left argument after a 1-cell is taken) and the right frame is 3 (there are 3 0-cells in the right argument). So the result frame is 3.

```
2 4 $ 0   gives   0 0 0 0   left 1-cell $ right 0-cell
                  0 0 0 0

... $ 1   gives   1 1 1 1   repeat 1-cell $ next 0-cell
                  1 1 1 1

... $ 2   gives   2 2 2 2   repeat 1-cell $ next 0-cell
                  2 2 2 2
```

The frame of the result is 3 and the things in that frame are 2 by 4 tables, so the shape of the final result is 3 2 4.

```
   $ 2 4 $"1 0 i.3
3 2 4
```

Rank (noun rank, verb rank, frames, cells, and the rank conjunction) applies to all verbs and greatly increases the ways in which you can use any verb.

Checkpoint E

At this point you should understand:

- the terms atom, list, table, array, axis, rank, shape, item, frame, and cell
- noun rank
- verb rank
- `"` (rank conjunction)
- agreement
- how rank determines which cell arguments a verb applies to
- how the result is built up of the partial results

Check your understanding by doing the following exercises:

- experiment with the primitives you know and use `"` to apply them to cells and see how the partial results build up the final result
- don't limit your experiments to verbs like `+`, but also try verbs such as `$` (shape) and `,` (append)

Adverb

An adverb is similar to a verb, but differs in the following:

- an adverb has only a left argument (a verb is ambivalent, and has either a right argument or both a left and a right argument)
- an adverb can apply to nouns or verbs (a verb applies only to nouns)
- an adverb typically has a verb as its result (a verb always has a noun result)

The verb result of an adverb is referred to as a *derived* verb.

The primitive `/` is an adverb. Its result is a new verb. If the monadic case of the derived verb is used, then the `/` is referred to as *insert*. If the dyadic case of the derived verb is used, then the `/` is referred to as *table*.

Insert adverb

A / is referred to as *insert* if it is applied to a verb and the derived verb is then used monadically. The derived verb applies itself monadically by inserting the original verb between the items of the argument.

```
   sumover =. +/
```

The adverb / takes the verb argument on its left, which is +, and creates a new verb named sumover.

```
   sumover 7 5 10
22
```

The items of the argument 7 5 10 are the three atoms 7, 5, and 10 and the definition of sumover is that it inserts its original verb between the items of the argument.

```
   7 + 5 + 10
22

   sumover i. 8
28
```

What if you do this to a table?

```
   a =. i. 2 3
   a
0 1 2
3 4 5

   sumover a
3 5 7
```

Interesting, but let's take a closer look. The items of a are the two lists 0 1 2 and 3 4 5. The verb sumover is defined to put the + (the original argument of /) between the items of its argument.

```
   0 1 2 + 3 4 5
3 5 7
```

What if there were more rows?

```
   a=. i. 3 4
   a
0 1  2  3
```

```
4 5  6  7
8 9 10 11

   sumover a
12 15 18 21
```

The items of `a` are the three lists and with the `+` inserted between them you have:

```
   0 1 2 3 + 4 5 6 7 + 8 9 10 11
12 15 18 21
```

The verb `sumover` applied to a table gives the sum over the columns. What if you wanted the sum over the rows?

```
   sumover"1 a
6 22 38
```

The above is worth thinking about. First give your new verb a name to make it easier to talk about.

```
   sumrows =. sumover"1
   sumrows a
6 22 38
```

Look up `/` in the *J Dictionary* and note that the rank information is `_ _ _`. The rank information for an adverb gives the rank of the derived verb. So, `sumover` has monadic rank `_` (unbounded). The verb `sumover` applies to its entire argument and so inserts its original verb of `+` between the items of the argument.

The verb `sumrows` has monadic rank `1` and applies to the 1-cells of its argument. It is applied to each of the 1-cells of the argument, giving a partial result, and these partial results are then assembled into the result frame. Instead of the entire argument being fed to the verb `sumrows`, 1-cells are fed in, so `sumrows` inserts `+` between the items of the 1-cells. The 1-cells of the table argument are the rows of the table, so the `+` is inserted between the items of the rows. Visually:

```
sumrows   0 1 2 3     (first 1-cell)  gives  6
sumrows   4 5 6 7     (next 1-cell)   gives  22
sumrows   8 9 10 11   (next 1-cell)   gives  38
```

The partial results of `6`, `22`, and `38` are then assembled into the list result.

What about a rank 3 argument?

```
   a =. i. 2 3 4
   a
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23

   sumover a
12 14 16 18
20 22 24 26
28 30 32 34
```

The items are the two tables and putting the + between them gives the result. Because this is the sum over the items, and in this case is the sum over the tables of a rank 3 array, it can be described as the sum over the planes of the array.

The name `sumover` was used because it made it clearer in the beginning what was being done. In practice it is probably better to just use the primitives directly.

```
   a =. i. 2 3 4
   +/ a          NB. sum over planes
12 14 16 18
20 22 24 26
28 30 32 34

   +/"2 a        NB. sum over tables
12 15 18 21
48 51 54 57

   +/"1 a        NB. sum over rows
 6 22 38
54 70 86
```

Table adverb

The dyad `v/` computes a table for the verb `v` (a function table).

```
   a =. i. 5
   a +/ a        NB. addition table
0 1 2 3 4
1 2 3 4 5
```

```
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

   a */ a        NB. times table
0 0 0  0  0
0 1 2  3  4
0 2 4  6  8
0 3 6  9 12
0 4 8 12 16

  0 1 +/ 0 1     NB. + table on booleans
0 1
1 2
  0 1 +./ 0 1    NB. or
0 1
1 1
   0 1 +:/ 0 1   NB. not-or
1 0
0 0
   0 1 */ 0 1    NB. times
0 0
0 1
   0 1 *./ 0 1   NB. and
0 0
0 1
   0 1 *:/ 0 1   NB. not-and
1 1
1 0
```

The derived verb `v/` has a left rank that is the left rank of `v` and a right rank of _ ; it applies `v` between each cell of the left argument and the entire right argument.

```
   additiontable =. +/
```

The verb `additiontable` is defined as the result of the `/` adverb applied to `+` as an argument. The definition is the same as the definition for `sumover` that you used in the previous section. The name `additiontable` makes sense when the dyad is used, and `sumover` makes sense when the monad is used. Nothing prevents using either name in a misleading way.

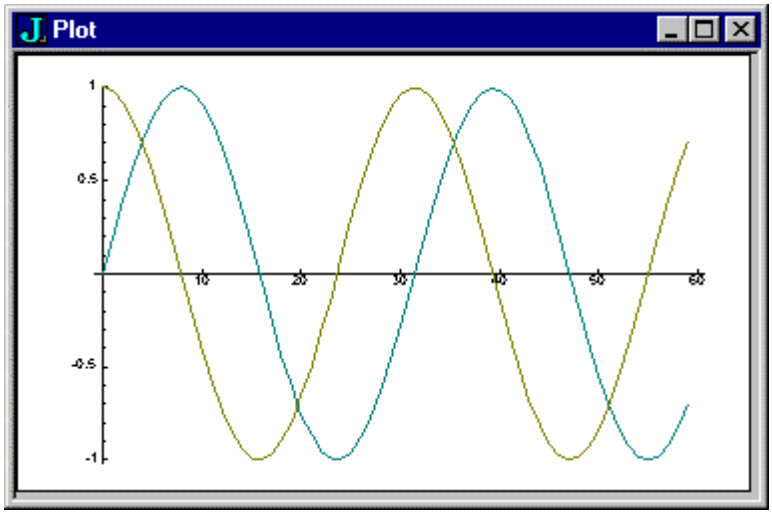
```
   a additiontable a
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

The left rank of + is `0`, so the rank of the derived verb is `0` and its cell arguments are atoms. The right rank of the derived verb is _ and its cell arguments are the entire right argument. Visually the above works as follows:

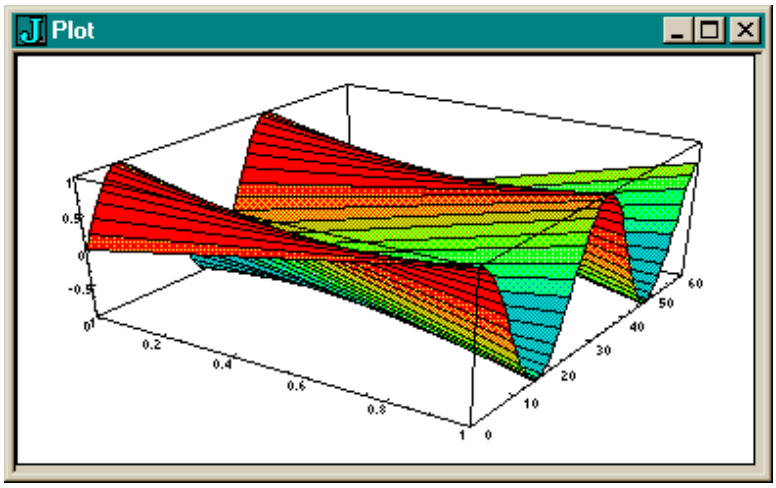
```
0 (first atom from left) + entire right   gives  0 1 2 3 4
1 (next atom from left)  + entire right   gives  1 2 3 4 5
2 (next atom from left)  + entire right   gives  2 3 4 5 6
3 (next atom from left)  + entire right   gives  3 4 5 6 7
4 (next atom from left)  + entire right   gives  4 5 6 7 9
```

The following is an interesting use of table together with the plotting from an earlier section. The plot facility plots each row in a table argument as a separate series of data.

```
   load 'plot'
   plot 1 2 o. / 0.2 * i.60
```




```
   'surface' plot 1 2 o. / 0.2 * i.60
```



The `1` from the left argument used as the left argument of `o.` gives sine values in the first row. The `2` from the left argument for `o.` gives cosine values in the second row.

In the examples above the left rank of the original verb is always 0 and so the cells of the left argument of the derived verb are simply the atoms of the left argument. Let's look at an example where the left rank of the original verb is not 0.

```
   f =. ,"1
```

The verb `f` is append with a left and right rank of `1`.

```
   p=. 2 2 $'abcd'
   p
ab
cd

   q=.3 3$'ABCD'
   q
ABC
DAB
CDA
```

```
   p f/ q
abABC
abDAB
abCDA

cdABC
cdDAB
cdCDA
```

Visually:

```
ab (1st list from left) , entire right   gives:
    abABC
    abDAB
    abCDA

cd (next list)          , entire right   gives:
    cdABC
    cdDAB
    cdCDA
```

Conjunction

A conjunction is similar to an adverb, except that it takes both a left and right argument. The rank conjunction was introduced informally in an earlier section. In addition, the `:` used to define verbs such as `centigrade` can now be recognized as a conjunction. In defining `centigrade`, the `:` takes a left argument of `3` and a right argument of `0`. So far it could be a verb, but the fact that its result is a verb proves it is a conjunction.

The verb result of a conjunction is referred to as a *derived* verb.

Order of execution - adverbs & conjunctions

Adverbs and conjunctions have higher precedence than verbs. This means that an adverb or conjunction is executed before a verb. Furthermore, the left argument of an adverb or conjunction is the entire verb phrase that precedes it. The exact rules for parsing and execution are given in section E of the *J Dictionary*.

For practical purposes, the following examples illustrate the rules.

```
   a =. i.2 3
   +/"1 - a
_3 _12
```

Like all J sentences, the above sentence executes from right-to-left. Before the + can be parsed as being a dyad or a monad, the higher precedence " conjunction executes. The " conjunction takes the 1 as its right argument and the *entire verb phrase to its left* as its left argument. The verb phrase to the left is the adverb / which takes the + as its left argument. The following uses parentheses to make clear the order of execution that follows from the rules.

```
((+/)"1) (- a)
```

As mentioned earlier, simple examples with constants may require that you separate the constant that is the conjunction argument from the constant that is the argument of the derived verb.

```
a (+"1) 1 2 3
a +"1 (1 2 3)
a +"1 [ 1 2 3
```

The last one uses the monad [(same) that is defined to just return its argument. This is a bit shorter and avoids the use of parentheses.

Box - monad <

So far you have dealt with atoms that are either numeric or character. The monad < (box) introduces a new type of atom called *boxed*. The monad < applies to any noun and returns an atom that is a box which contains the argument.

An array is either a numeric array that contains numbers, or a literal array that contains characters, or a boxed array that contains boxes. Arrays of numbers and characters are referred to as *open* to distinguish them from boxed arrays.

```
b =. < 2 3 4
$b      NB. an atom has empty shape
```

A boxed array is displayed in a box.

```
   b
+-----+
|2 3 4|
+-----+
```

```
   c =. < 4 7 9
   d =. b , c          NB. append
   d
+-----+-----+
|2 3 4|4 7 9|
+-----+-----+

   $d                  NB. list with shape 2
2

   (< 2 3 4) , < 4 7 9
+-----+-----+
|2 3 4|4 7 9|
+-----+-----+
```

Arrays of different types (numeric, character, and boxed) cannot be appended to one another.

```
   'a' , 3
|domain error
|   'a'    ,3

   3 , < 2 3 4
|domain error
|   3    ,<2 3 4
```

Boxed arrays are of the same type and can be appended no matter what they contain.

```
   (< 'abc') , < 4 5 6
+---+-----+
|abc|4 5 6|
+---+-----+

   (< 2 3 $ 'abcdef') , (< i. 3 4) , < 23
+---+---------+--+
|abc|0 1  2  3|23|
|def|4 5  6  7|  |
|   |8 9 10 11|  |
+---+---------+--+
```

Link - dyad ;

The dyad `;` (link) makes it easy to create a list of boxed nouns.

```
   2 3 4 ; 4 7 9
+-----+-----+
|2 3 4|4 7 9|
+-----+-----+

   'abc' ; 5 7 9
+---+-----+
|abc|5 7 9|
+---+-----+

   (2 3 $ 'abcdef') ; (i. 3 4) ; 23
+---+---------+--+
|abc|0 1  2  3|23|
|def|4 5  6  7|  |
|   |8 9 10 11|  |
+---+---------+--+
```

Look link up in the *J Dictionary*. Note that its dyad rank is _ _ so it applies to its entire left and entire right argument. It always boxes its left argument, and its right argument is boxed only if not already boxed. Boxing its right argument only if not already boxed makes the following work:

```
   'abc' ; 'defg' ; 'hijkl'
+---+----+-----+
|abc|defg|hijkl|
+---+----+-----+
```

This is evaluated from right-to-left as:

```
   'abc' ; ('defg' ; 'hijkl')
```

So the right argument to the leftmost link is already boxed, and will not be boxed again. Boxed again? What would that look like?

```
   'abc' ; < 'defg' ; 'hijkl'
+---+------------+
|abc|+----+-----+|
|   ||defg|hijkl||
|   |+----+-----+|
+---+------------+
```

Boxes can be nested!

Open - monad >

The monad > (open) is the inverse of box. That is, it takes the contents out of a box. Applied to a noun that is already open it has no effect.

```
   > 23 5 7
23 5 7

   a =. i.5
   a
0 1 2 3 4

   < a
+---------+
|0 1 2 3 4|
+---------+

   > < a
0 1 2 3 4
```

Look > up in the *J Dictionary*. The monad > has a rank of `0` so it applies to each atom in its argument. Each atom is opened, creating a partial result, that is then assembled with all the other partial results into the final result.

```
   a =. 1 2 3 ; 5 6 7
   a
+-----+-----+
|1 2 3|5 6 7|
+-----+-----+

   > a
1 2 3
5 6 7
```

The assembly of the partial results is straightforward if they all have the same rank and shape. But with open the partial results could have very different ranks and shapes.

```
   a =. 1 ; 2 3 ; 4 5 6
   a
+-+---+-----+
|1|2 3|4 5 6|
+-+---+-----+
```

```
   > a
1 0 0
2 3 0
4 5 6
```

Each atom is opened and the partial results are extended to fit into the result frame.

```
> 1     gives 1     (atom)
> 2 3   gives 2 3   (list)
> 4 5 6 gives 4 5 6 (list)
```

The result frame has to have partial results that all have the same rank and shape. To do this, each partial result is extended to have the same rank as the rank of the highest rank result and a shape that is the maximum along any axis of all the partial results.

In the example above, the atom `1` is extended to a list and is padded with `0`'s to have a shape of `3`. The list `2 3` is padded with a `0` to have a shape of `3`. The `4 5 6` list is already OK. And the final result has a frame of `3` where each result cell is a list of `3` atoms.

What if the boxes contain characters?

```
   > 'abc' ; 'defg' ; 'hijkl'
abc
defg
hijkl
```

The result is a `3` by `5` table. The character partial results are extended in a similar manner, but blanks are used as fill, rather than `0`. You cannot open a boxed list that contains both character and numeric data.

```
   > 1 2 3 ; 'asdf'
|domain error
|       >1 2 3;'asdf'
```

There is no way to put the numeric and character partial results together in the frame so there is a domain error.

From - dyad { (selecting items)

The dyad { (from) is used to select items from an array. This is sometimes called indexing an array.

```
   a =. 5 7 9 4 8
   3 { a         NB. select item at index 3
4

   0 { a         NB. indexes start at 0
5

   2 4 { a       NB. select 2 items
9 8

   2 0 4 { a     NB. select 3 items
9 5 8
```

From selects items from its argument, and therefore selects rows from tables.

```
   m =. i. 3 4
   m
0 1  2  3
4 5  6  7
8 9 10 11

   0 { m
0 1 2 3

   2 1 { m
8 9 10 11
4 5  6  7
```

From selects tables from rank 3 arrays.

```
   1 { i. 2 3 4
12 13 14 15
16 17 18 19
20 21 22 23
```

From (boxed indexes)

If the left argument of `{` is boxed, then it is opened and each of its items gives the indexes along successive axes of the right argument. This can be used to select any subarray from an array.

```
   m =. i. 3 4
   m
0 1  2  3
4 5  6  7
8 9 10 11

   (< 1 2) { m
6
```

The opened left argument is the list `1 2`. The first item is `1,` and it is used as the index of the 1st axis; the second item is `2,` and it is used as the index of the 2nd axis. The `1` selects the list `4 5 6 7` and the `2` selects `6` from that list.

If there are fewer items in the list than there are axes, then all of the trailing axes are selected.

```
   (<1) { m
4 5 6 7
```

This is more interesting with a higher rank array.

```
   d =. i.2 3 4
   d
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23

   (<1 2 3) { d   NB. plane 1, row 2, column 3
23

   (<1 2) { d  NB. plane 1, row 2, all columns
20 21 22 23
```

So far the items in the list of indexes for each axis has been an atom and selects only one index. What if you want more than one index?

If an item in the list of indexes is boxed, then it is a list of indexes for that axis.

Suppose you want to select from `m` the table of atoms that are in rows `0 2`, and columns `0 2 3`. That is, the table:

```
      col-0  col-2  col-3
row-0   0      2      3
row-2   8     10     11
```

The indexes for the axes are the list:

```
   0 2 ; 0 2 3
+---+-----+
|0 2|0 2 3|
+---+-----+
```

That list of indexes needs to be boxed so that each item will be treated as indexes into the successive axes. The `0 2` selects rows (1st axis) and the `0 2 3` selects columns (2nd axis).

```
   < 0 2 ; 0 2 3
+-----------+
|+---+-----+|
||0 2|0 2 3||
|+---+-----+|
+-----------+

   (< 0 2 ; 0 2 3) { m
0  2  3
8 10 11

   (< 0 1 ; 0 2 ; 2 3) { d
 2  3
10 11

14 15
22 23
```

Again, if there are fewer items than axes, then all of the trailing axes are selected.

```
   (< 0 1 ; 0 2) { d
 0  1  2  3
 8  9 10 11

12 13 14 15
20 21 22 23
```

Frequently the desired subarray includes all of an axis that is not a trailing axis. This could be done by giving all indexes for that axis.

```
   (< 0 1 2 ; 2 3) { m
 2  3
 6  7
10 11
```

This may be inconvenient in a real application where it would necessary to calculate the indexes. For this reason, a boxed empty list, `<''`, indicates that all indexes in the axis are selected.

```
   < (<'') ; 2 3
+--------+
|+--+---+|
||++|2 3||
|||||   ||
||++|   ||
|+--+---+|
+--------+
```

The above, used as the left argument will select all indexes along the first axis because the first item is a boxed empty list, and indexes `2` and `3` along the 2nd axis.

```
  (< (<'');2 3) { m
 2  3
 6  7
10 11
```

The boxed empty list is so useful that the primitive `a:` is defined as `<''` . So, the above can be simplified.

```
  (< a: ; 2 3) { m
 2  3
 6  7
10 11
```

The above can be even more easily expressed with the rank conjunction.

```
   2 3 {"1 m
 2  3
 6  7
10 11
```

From (scattered indexing)

If the left argument is a singleton, it is opened and its items are indexes along successive axes.

```
   (<0 2 ; 2 3) { m
 2  3
10 11
```

What if the left argument wasn't a singleton?

```
   (0 2 ; 2 3) { m
2 11
```

What is going on here? Nothing special, as this is just your old friend rank. The dyad `{` has a left rank of `0` and a right rank of _ . This means that the left argument is taken as 0-cells and the right argument is taken in its entirety. Visually:

```
(<0 2)  (first left cell)  { m   gives 2
(<2 3)  (next left cell)   { m   gives 11
```

The result is assembled from the `2` and `11` partial results.

This is called scattered indexing.

```
   (0 0 ; 1 1 ; 2 2) { m      NB. scatter index a diagonal
0 5 10
```

Amend } (modify selected)

Amend is an adverb whose result is a dyad that is used to modify an array. The left argument of amend is usually a noun. Let's look at an example:

```
   change_index_two =. 2}
```

The verb `change_index_two` is used dyadically. Its right argument is the original data and the left argument is a new value for index position `2`.

```
   15 change_index_two 5 6 7 8
5 6 15 8

   30 change_index_two 23 18 17
23 18 30

   'b' change_index_two 'cat'
cab

   15 (2}) 5 6 7 8
5 6 15 8

   30 (2}) 23 18 17
23 18 30

   'b' 2} 'cat'
cab
```

This extends in ways that you might expect.

```
   23 (1 4}) 7 7 7 7 7
7 23 7 7 23

   23 24 (1 4}) 7 7 7 7 7
7 23 7 7 24

   'bet' 2 5 8} 'cattumbiz'
cabtuebit
```

In general an amend `x s} y` is defined as:

The result is formed by replacing by `x` those parts of `y` that are selected by s. The s argument to `}` is treated the same way as the left argument of the verb `{` .

Amend allows us to give selected parts of an array new values. The amend argument gives the indexing information about what data to modify. This selects the same elements to be modified as it would if used as the left argument to the dyad verb `{` .

If you understand from, then amend is quite simple.

```
   m =. i. 3 4
   1 { m
4 5 6 7

   23 23 23 23 (1}) m
 0  1  2  3
23 23 23 23
 8  9 10 11

   23 (1}) m
 0  1  2  3
23 23 23 23
 8  9 10 11
```

You can first use the selection information to see what data is to be modified.

```
   1 2 { m
4 5  6  7
8 9 10 11
```

The selected data is a subarray of shape `2 4`. So you need a subarray of shape `2 4` to replace the selected data.

```
  (2 4 $ 23 23 23 23 24 24 24 24) (1 2}) m
 0  1  2  3
23 23 23 23
24 24 24 24
```

Suppose you want to modify the subarray that is selected as rows 1 and 2, and columns 0 and 3 with the value 12.

```
   12 ((<1 2;0 3)}) m
 0 1  2  3
12 5  6 12
12 9 10 12
```

Modify that subarray by replacing it with the array in the left argument.

```
   (2 2 $ 23 24 25 26) ((<1 2;0 3)}) m
 0 1  2  3
23 5  6 24
25 9 10 26
```

Selecting without from

Some situations where you could use { (from) are so common that they have their own primitives. These primitives are not dealt with in any detail here, and are mentioned so that you are aware of them and can look up their definitions and make use of them in your own work. They are shown here by example.

The monad {. (head) takes the first item of its argument and is similar to { with a left argument of 0.

```
   {. 5 6 7
5

   {. i. 3 4
0 1 2 3
```

The monad {: (tail) takes the last item of its argument and is similar to { with a left of _1 (oh yes, forgot to mention earlier that negative indexes simply index from the end of the axis).

```
   {: 5 6 7
7

   {: i. 3 4
8 9 10 11
```

The monad }. (behead) drops the first item of its argument.

```
   }. 5 6 7
6 7

   }. i. 3 4
4 5  6  7
8 9 10 11
```

The monad }: (curtail) drops the last item of its argument.

```
   }: 5 6 7
5 6
   }: i. 3 4
0 1 2 3
4 5 6 7
```

The dyad `{.` (take) takes the indexes from axes as indicated by the left argument.

```
   3 {. i. 8
0 1 2

   2 3 {. i. 3 4
0 1 2
4 5 6

   2 {. i. 3 4
0 1 2 3
4 5 6 7
```

The dyad `}.` (drop) drops the indexes from axes as indicated by the left argument.

```
   1 }. 2 3 4
3 4

   1 2 }. i. 3 4
 6  7
10 11
```

An interesting way to think of `{. {: }.` and `}:` is that they are indexing corners of the array.

One capability that the dyad `{.` has that is not so directly related to from is it can create an array that is larger than the selected corner. It does this by filling in with `0`, `' '`, or `a:` as appropriate.

```
   5 {. 5 6 7
5 6 7 0 0

   4 5 {. i. 2 3
0 1 2 0 0
3 4 5 0 0
0 0 0 0 0
0 0 0 0 0

    5 {. 'abc'
abc
   <"0 [ 5 {. 'abc'        NB. make sure they are there
+-+-+-+-+-+
|a|b|c| | |
+-+-+-+-+-+
```

```
   7 {. (<"0) 5 {. 'abc'      NB. fill with a:
+-+-+-+-+-+++
|a|b|c| | |||
+-+-+-+-+-+++
```

The dyad # (copy) is also fairly directly related to from. Its left argument is a list of how many times to repeat the corresponding item from the right argument.

```
   3 2 1 2 3 # 'abcde'
aaabbcddeee
```

The above is equivalent to the following:

```
   0 0 0 1 1 2 3 3 4 4 4 { 'abcde'
aaabbcddeee
```

Cut ;.

The cut conjunction applies a verb to partitions of its argument. In discussing cut we will use the expression `f =: u ;. n` to give names to the various elements. The left argument of cut is the verb `u` that will be applied to partitions of the right argument of the derived verb `f`. The right argument of cut is the noun `n` that indicates the kind of partitions.

Let's consider cases where `n` is `1`, `_1`, `2`, or `_2` and where the derived verb is used monadically.

If `n` is `2` or `_2`, the items of the right argument are partitioned into arguments that end with the last item in the argument. The item that is used to mark the partitions is called the *fret*. A negative `n` indicates that the fret is not included in the partition. Each partition is passed to the verb `u` and the partial results are assembled into the final result.

```
   cut2  =. < ;. 2
```

The definition of `cut2` could be read as: *box cut last*.

```
   cut2 'how now brown cow '
+----+----+------+----+
|how |now |brown |cow |
+----+----+------+----+
```

In this example the items of the right argument are characters. The fret is the last character, which is a blank. The fret is used to break the entire argument into a series of arguments to which the verb `<` is applied. Visually:

```
< 'how '  (first partition)  gives  first partial result
< 'now '  (next partition)   gives  next partial result
< 'brown '
< 'cow '
```

The partial results are assembled into the final result.

```
   <;._2 'how now brown cow '
+---+---+-----+---+
|how|now|brown|cow|
+---+---+-----+---+
```

The `cut2` boxed results include the fret and the `<;._2` boxed results do not include the fret.

The following applies to the same partitions of the right argument, but applies the dyad # (tally) instead. This gives us the count of each partition.

```
   #;._2 'how now brown cow '
3 3 5 3
```

This applies in the same way to numeric data. In the following `_1` is the fret.

```
   a =. 2 3 4 _1 5 3 2 _1 23 45 65 132 _1
   <;._2 a
+-----+-----+------------+
|2 3 4|5 3 2|23 45 65 132|
+-----+-----+------------+

   #;._2 a
3 3 4

   +/;._2 a
9 10 265
```

A `1` or `_1` uses the first item as the fret. If positive, the fret is included in the result, if negative it is not included.

```
   <;._1 'madam i''m adam'
+---+---+----++
|ada| i'| ada||
+---+---+----++
```

Sometimes the partition information is separate from the data. Instead of frets in the data, the partition information can be provided in a left argument to the derived verb. The partition information is boolean data where a 1 indicates the start (with 1 or _1) or end (with 2 or _2) of the partitions.

```
   1 0 0 0 1 0 0 <;.1 'abcdefg'
+----+---+
|abcd|efg|
+----+---+

   d =.  'the test is the thing'
   'the' E. d
1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0

   ('the' E. d) <;.1 d
+------------+---------+
|the test is |the thing|
+------------+---------+
```

The example above uses a new primitive E. that you can look up in the *J Dictionary* if you want additional information at this time. However, without worrying about the details you should get the idea of what is happening.

Chopping up character lists into boxes is so useful that there is a standard utility called cutopen that handles many of the common cases. For example:

```
   cutopen 'testing testing 1 2 3'
+-------+-------+-+-+-+
|testing|testing|1|2|3|
+-------+-------+-+-+-+
```

Each

Frequently with boxed data it is useful to be able to do something to the contents of each of the boxes. This is so useful that the standard profile defines an adverb called each that does exactly this. The definition of each involves a little more than you have covered so far, but don't worry about the details, just use it.

The adverb each takes a verb as its left argument. The derived verb is applied to the contents of the boxes of its arguments.

```
   a =. 10 12 13 ; 2 3 ; 4 5 6 8 3
   a
+--------+---+---------+
|10 12 13|2 3|4 5 6 8 3|
+--------+---+---------+

   +/ each a           NB. sum over each
+--+-+--+
|35|5|26|
+--+-+--+

   */ each a           NB. times over each
+----+-+----+
|1560|6|2880|
+----+-+----+

   |. each a           NB. reverse each
+--------+---+---------+
|13 12 10|3 2|3 8 6 5 4|
+--------+---+---------+

   >./ each a          NB. max over each
+--+-+-+
|13|3|8|
+--+-+-+
```

The previous examples all used the derived verb monadically. The following use the derived verb dyadically.

```
   23 , each a         NB. append each
+-----------+------+------------+
|23 10 12 13|23 2 3|23 4 5 6 8 3|
+-----------+------+------------+

   5 < each a
+-----+---+---------+
|1 1 1|0 0|0 0 1 1 0|
+-----+---+---------+

   1 |. each a
+--------+---+---------+
|12 13 10|3 2|5 6 8 3 4|
+--------+---+---------+
```

Did you catch the new verb `|.` (reverse • rotate) that slipped in above? Did you look it up in the *J Dictionary*?

Hook

A *train* is a sequence of two or three words in a row that is given a special meaning. A train of two verbs is a *hook* and is evaluated as follows:

```
  (f g) y  evaluates as   y f g y
x (f g) y  evaluates as   x f g y
```

Suppose you wanted to scale a list of numbers such that the result was each number divided by the maximum number in the list. The maximum over a list is given by the verb derived by applying the adverb `/` to the verb `>.` .

```
   a =. 3 5 8 2 7
   maxover =. >./
   maxover a
8
```

To divide an argument by the maximum over the argument you can use:

```
   a % maxover a
0.375 0.625 1 0.25 0.875
```

The above can be written as a hook.

```
   (% maxover) a
0.375 0.625 1 0.25 0.875
```

You can now define `scale` as a hook with `% >./` .

```
   scale =. % >./
   scale a
0.375 0.625 1 0.25 0.875
```

Fork

A train of three verbs is a *fork* and is evaluated as follows:

```
  (f g h) y    evaluates as    (f y) g (h y)
x (f g h) y    evaluates as    (x f y) g (x h y)
```

A simple example of a fork is the sequence of three verbs `+/ % #` . The `/` adverb takes its left argument `+` and returns a verb, so there is a sequence of three verbs. Let's examine the use of this fork monadically.

```
   (f  g h) y  evaluates as    (f  y) g (h y)
   (+/ % #) y  evaluates as    (+/ y) % (# y)
```

This can be read as: *sum over the argument divided by the count of the argument*. This is the definition of the mean or average.

```
   (+/ % #) 5 9 12
8.66667

   (+/ % #) i.9
4
```

You can now define `mean` as a fork with `+/  %  #` .

```
   mean =. +/ % #
   mean i.9
4
```

Tacit definition

The `centigrade` verb was defined explicitly with the `:` conjunction. The term explicit indicates that the arguments to the verb in the definition are referred to explicitly by their names of `x.` and `y.` .

In a tacit definition the arguments are not named and do not appear explicitly in the definition. The arguments are referred to implicitly by the syntactic requirements of the definition. You have already used several tacit definitions.

```
   plus =. +
   sumover =. +/
   maxover =. >./
   scale =. % >./
   mean =. +/ % #
```

The above are all tacit definitions. They do not use `:` and do not refer to arguments by name. In some cases the tacit form of definition is much simpler and more obvious than the equivalent explicit definition. In more complicated situations, it may take a bit of experience before you are comfortable with a tacit definition. This is partly because you probably have experience with explicit forms of definitions and very little with tacit definitions. In addition,

tacit definitions tend to be more concise and mathematical expressions of a definition, and it may be necessary to go through the more detailed steps of creating an explicit definition before the equivalent tacit definition becomes clear.

Let's revisit `fahrenheit` to see how it could be defined tacitly. Open the cf.ijs script and look at the `fahrenheit` definition.

```
fahrenheit =: 3 : 0
t1 =. y. * 9
t2 =. t1 % 5
t3 =. t2 + 32
)
```

You can start by cleaning up the explicit definition. Now that you are more comfortable with J you can combine these calculations into a single line.

```
   fa =: 3 : '(y. * 9 % 5) + 32'
```

The parentheses are required because the calculation inside them must be done before the 32 is added.

Let's shuffle the definition a bit to make the steps in building a tacit definition a bit clearer.

```
    fb =: 3 : '32 + ((9%5) * y.)'
```

The above could be read as: *add 32 to nine-fifths times the argument.*

So, you need an add32 verb and a ninefifthstimes verb. You can use the bond conjunction & to build these verbs tacitly. The bond conjunction with a constant left argument returns a derived monad that is the verb in its right argument with the constant left argument.

```
   add32 =: 32 & +
```

This defines `add32` as a monad that adds 32 to its argument.

```
   add32 12
44

   ninefifthstimes =: (9%5) & *
```

This gives a monad which multiplies its argument by `9%5`.

```
   ninefifthstimes 20
36
```

Combining these you have:

```
   add32 ninefifthstimes 100
212
```

The atop conjunction @ combines two verbs into a derived verb that applies the right verb to its argument and then applies the left verb to that result.

`(u @ v) y` evaluates as `u v y`

Use the atop conjunction to combine your two verbs to create the final definition.

```
   fc =: add32 @ ninefifthstimes

   fc 100
212

   fc _40
_40

   fc 0
32
```

Display the verb fc and note that its definition is dependent on the other two definitions.

```
   fc
+-----+-+---------------+
|add32|@|ninefifthstimes|
+-----+-+---------------+
```

Sometimes after you have built up a tacit definition from smaller building blocks you realize you really don't want all those smaller definitions hanging around. The `f.` adverb takes a tacit definition and replaces names with their definitions.

```
   fz =. fc f.
```

The adverb `f.` , like all adverbs, takes its argument on its left.

Look at `fz` to see the final definition.

```
   fz
+--------+-+---------+
|+--+-+-+|@|+---+-+-+|
||32|&|+||  ||1.8|&|*||
|+--+-+-+|  |+---+-+-+|
+--------+-+---------+
```

The system can display tacit definitions in several different forms. These options can be selected from the Edit|Configure... menu dialog. With box display you get the preceding display. The *Box Display* can be very useful in understanding tacit definitions. However, for now select *Linear Display* so that you will see the following:

```
   fz
32&+@(1.8&*)
```

In comparing something as simple as a verb defined as +, the tacit definition is much simpler than the equivalent explicit definition. In the `fahrenheit` example it could be argued that the explicit definition was simpler, especially if you used the `1.8` directly instead of the `9%5` as does the tacit definition.

```
fx =: 3 : '32+1.8*y.'
```

vs.

```
fz =: 32&+@(1.8&*)
```

The real strength in tacit programming comes in more complicated transformations of the arguments, particularly when the arguments must be referenced several times. The following illustrates another use of tacit definition.

```
   xmean =: 3 : '(+/y.) % #y.'
```

This is the mean that you ran across in the Fork section.

```
   mean =: +/ % #
```

The tacit definition just uses the fork directly.

The fork could also have been used in the explicit definition, but would have required parentheses around the fork.

```
   xmean =: 3 : '(+/ % #)y.'
```

One advantage of tacit definitions is that they are more easily manipulated in formal ways than are explicit definitions. For example, J can automatically derive the inverse of many tacit definitions. Let's try this with the `fz` tacit definition. The inverse of the Fahrenheit conversion is the centigrade conversion. The standard profile defines an adverb `inverse`.

```
   fz =: 32&+@(1.8&*)
   cz =: fz inverse
   fz 100
212

   cz 212
100

   fz 0
32

   cz 32
0
```

Tacit programming is very powerful, but there is no need to leap into it. It is important to know what it is and to start using it in simple cases as this is the best way to become more familiar with it.

Explicit-to-tacit translator

There is a primitive which automatically converts one-line explicit definitions to an equivalent tacit definition. You can learn a lot about tacit programming by writing one line explicit definitions, converting them to tacit form, and studying the resulting tacit definition.

Let's do this with an explicit `fahrenheit` definition. A left argument of `3` to `:` creates an explicit definition. A left argument of `13` to `:` creates a tacit definition.

```
   fx =:  3 : '32 + y. * 9 % 5'       NB.  3 explicit
   ft =: 13 : '32 + y. * 9 % 5'       NB. 13 tacit
```

Use Edit|Configure... to select Linear Display.

```
   ft
32"_ + ] * 1.8"_
```

At first glance this is confusing as it introduces several new things at once. The first thing to do is to look at the boxed display.

Select the Box Display.

```
   ft
+--------+-+---------------+
|+--+-+-+|+|+-+-+---------+|
||32|"|_|| |||]||*|+---+-+-+||
|+--+-+-+| || | ||1.8|"|_|||
|        | || | |+---+-+-+||
|        | |+-+-+---------+|
+--------+-+---------------+
```

At the top level of boxing there are 3 boxes. This is a train with three elements and is in fact a fork. You can take this thing apart by giving names to the parts and looking at them separately.

The first element of the fork is the phrase `32"_ .` Give this a name and experiment with it a bit as a monadic verb.

```
   left =: 32"_
   left 123
32

   left i.5
32
```

Whatever you give `left` as an argument, it just returns `32`. You've seen the `"` conjunction before, but not with a constant left argument. Let's look this up in the *J Dictionary*. When you turn to the definition for rank you will notice that there are three pages of definitions, each with its own header. The three headings are:

```
    Rankm " n
    Ranku " n
                          Assign rank    m " v    u " v    mv lv rv
```

The different definitions are for the rank conjunction used with different types of arguments. In the headings `m` and `n` indicate noun arguments and `u` and `v` indicate verb arguments. Your earlier use of `"` involved a verb left argument and a noun right argument and is covered by the second definition. Both `32` and _ (infinity) are nouns so it is the first definition that is relevant.

Reading the definition for `m " n` makes it clear that the observations are correct. With a right rank of _ , the derived verb applies to its entire right argument, and no matter what it is, it returns the left argument, which is `32`.

Let's look at the right element of the fork.

```
   right =: ] * 1.8"_
   right 23
41.4

   right 10
18
```

Let's not worry about the details of the definition, but again, by observation, what the verb `right` does is to multiply its argument by `1.8` (which is `9%5`).

The final definition is a fork.

```
   ff =: left + right
   ff 100
212

   ff 0
32
```

The result of the fork, `ff` , does work. Let's look in more detail at the definition of the fork.

```
left + right   evaluates as    (left y) + (right y)
                  32     + (1.8 * y)
```

Which is the Fahrenheit conversion!

Compare your custom built tacit definition with the automatically translated one and note how different they are.

```
32&+@(1.8&*)
     vs
32"_ + ] * 1.8"_
```

Tacit programming is very rich and varied and is tightly tied to adverbs and conjunctions such as bond, atop, and rank, and to trains such as hook and fork.

Checkpoint F

At this point you should understand:

- the terms adverb and conjunction
- how to evaluate sentences with adverbs and conjunctions

- how to select subarrays from arrays
- how to modify subarrays in arrays
- boxed data
- hooks and forks
- tacit definition

Check your understanding by doing the following exercises:

- experiment with dyads `{ {. }.` (negative numbers in the left arguments)
- look up `a.` in the *J Dictionary* and use it as the right argument in experiments with the dyads `{ {. }.`
- experiment with the adverbs and conjunctions introduced so far
- experiment with the hook `= <.` , determine what it does and why it does it (hint: give it a name and apply it to `2 4 5.2 7 6.5`)
- do the same with the hook `= +`

Foreign !:

All J interfaces with the environment are provided by the conjunction `!:` (foreign). In most cases the foreign result is a verb that provides a specific service. The left argument of foreign selects a general family of services and the right argument selects a specific service from that family. The families of services selected by the left argument are:

0 Scripts
1 Files
2 Host Commands
3 Storage Types
4 Name Classes and Lists
5 Representations
6 Time
7 Space
8 PC DOS Facilities
9 Global Parameters
11 Windows
13 Debug
14 Data Driver
15 Dynamic Link Library
16 Sockets
17 Regular Expression
18 Locales
128 Numerical Functions

The foreign conjunction is documented in the *J Dictionary* and in the *J Online Documentation.*

You have run across a few specific uses of the foreign conjunction in earlier sections.

The derived verb `9!:11` was used in the Print precision section. The `9` selects the Global Parameters family and the `11` gives a derived verb that sets the print precision.

The load verb uses `0!:0` to load scripts. The left argument selects the scripts family. Look up the definition. The `0` right argument is treated as three decimal digits 000 and therefore executes sentences from a file, stops on an error, and is silent (does not display sentences or results).

Files

Many applications require reading and writing files. Like all J interfaces with the environment, files are accessed with the foreign conjunction. The `1` family of foreigns work with files. First define a few verbs for convenience.

```
   readfile =: 1!:1
   writefile =: 1!:2
```

Let's create a file with some data in it. You'll be using the filename several times, so give it a name. The file foreigns require that the file name be a boxed string.

```
   fn =. < 'user\test.txt'
   'testing 1 2 3' writefile fn
```

Use whatever file editor you like to take a look at the file `test.txt` that was created in the J user directory. You could also open it as a script file in J by using the **File|Open** command (you will have to change the "List files of type" combobox to list all files in order to see your test.txt file).

You can read the data from this file.

```
   data =. readfile fn
   data
testing 1 2 3
```

You can rewrite the file and read the new data.

```
   'new stuff for the file' writefile fn
   readfile fn
new stuff for the file
```

Use an editor to change and resave the `test.txt` file and read it again to see that you get the new data. Again, you could do this by opening the file as a script file in J, editing it, and closing it and saving the changes.

Let's assume you had a numeric table that you wanted to write out as text file.

```
   numtab =. i. 4 5
   numtab writefile fn
|domain error
|   data     writefile fn
```

If you try to write `numtab` out you get a domain error because `writefile` requires a string as its left argument. So, you need to convert the numeric table to a string. The first step is to convert the numeric data to character data. The primitive `":` (format) does this.

```
   cdata =. ": numtab
   cdata
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19

   $cdata
4 14
```

The display of `cdata` looks like the numeric data, but its shape of `4 14` makes it clear that it is a character table. However, you still can't write this out to file because a file must be a list, not a table.

The monad `,` (ravel) puts all the atoms of an array into a list.

```
   crdata =. , cdata
   $ crdata
56

   crdata writefile fn
   readfile fn
 0  1  2  3  4 5  6  7  8  910 11 12 13 1415 16 17 18 19
```

The data has been written to the file. However, reading the data from the file shows there are still some problems. The fact that there were four rows of numbers has been lost and some of the numbers from the end of a row (such as 9) run right into the first number of the next row. Important information has been lost. The character list should indicate that it has four lines of data.

Lines in a text file are separated by two special characters called CR (carriage return) and LF (line feed). These characters are defined by the standard profile. The list of these two characters used to separate lines is called CRLF. On the Macintosh, a CR alone is used to separate lines, and if you are working on a Mac you will have to take this into account. UNIX systems use just the LF character to separate lines.

```
   'abc' , CRLF , 'defghi'
abc
defghi
```

To each item (list) in `cdata` you want to append the list CRLF. You need do this with a rank 1 version of append.

```
   ddata =. cdata ,"1 CRLF
   ddata
 0  1  2  3  4

 5  6  7  8  9

10 11 12 13 14

15 16 17 18 19
```

The blank lines in the display occur because the CRLF characters cause a new line, but the end of the row of a table also causes a new line. However, when you ravel this to create a list, the system won't have any rows to worry about and the display will again look OK.

```
   ldata =. , ddata
   ldata
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
```

Now you have a string with complete information about the original data that you can write to the file.

```
  ldata writefile fn
```

Open the file in an editor, or as a script file, to see that the data is there.

What if you had this file and wanted to get the numbers in it into J for processing? You need to reverse the previous process.

```
   rdata =. readfile fn
   rdata
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19

   $rdata
64
```

Getting the raw character data in is easy. But notice from the shape that it is a list of character data.

You know that each line of data ends with CRLF. The fact that this is two characters, instead of 1 is a nuisance, so the first thing to do is to get rid of the CR characters and to leave just the LF as the delimiter. The following expression uses `-.` (look it up in the *J Dictionary)* to remove the CR characters from the data. Character data with just CR, just LF, or with CRLF separating lines displays the same.

```
   dlf =. rdata -. CR
   dlf
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
```

You can use the `cutopen` verb to partition the data.

```
   bdata =. cutopen dlf
   bdata
+---------+---------+--------------+--------------+
|0 1 2 3 4|5 6 7 8 9|10 11 12 13 14|15 16 17 18 19|
+---------+---------+--------------+--------------+
```

Each box contains the character data for the corresponding line. You need a primitive that converts strings to numbers. The dyad ". can be used to convert characters to numbers.

```
   0 ". '5 2 7'
5 2 7

   a =. 0 ". '5 2 7'
   3 + a
8 5 10
```

The left argument of `".` is the value used if a conversion of a number fails.

```
   0 ". '5 7.5 23.b 8'
5 7.5 0 8
```

Use the each adverb to convert each of the boxes to numbers.

```
   ndata =. 0 ". each bdata
   ndata
+---------+---------+--------------+--------------+
|0 1 2 3 4|5 6 7 8 9|10 11 12 13 14|15 16 17 18 19|
+---------+---------+--------------+--------------+
```

The display of bdata and ndata look the same, but the bdata boxes contain characters and the ndata boxes contain numbers. Open the ndata boxes to get the numeric table result.

```
   d =. > ndata
   d
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
```

You can wrap this all together by creating a new script file, entering the following definitions, and saving it with a permanent name.

```
writetable =: dyad : 0
d =. ": x.
d =. d ,"1 CRLF
d =. , d
d 1!:2 y.
)

readtable =: 3 : 0
d =. 1!:1 y.
d =. d -. CR
d =. cutopen d
d =. 0 ". each d
d =. > d
)
```

Run the script file and test your definitions.

```
   (i. 3 7) writetable fn
   1 + readtable fn
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
```

As you gain more experience with J you will start combining sentences together. A more experienced J programmer would probably write the above definitions as follows:

```
writetable =: 4 : '(,(": x.),"1 CRLF) 1!:2 y.'
readtable =: 3 : '>0 ". each cutopen (1!:1 y.)-.CR'
```

The script files.ijs provide many useful utilities for working with files. Look them up in the *J Online Documentation.*

```
   load 'files'
   (": i. 3 9) fwrites 'newtest.txt'
84

   0 ". freadr 'newtest.txt'
 0  1  2  3  4  5  6  7  8
 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26
```

Component files

A component file (jfile) can be thought of as a boxed list stored in a file. An item of the boxed list on file is referred to as a component. The script jfiles.ijs provides the utilities for working with jfiles.

```
   load 'jfiles'
   f =. 'user\data.ijf'
   jcreate f
1

   'first component' jappend f
0

   (1.5+i.2 3) jappend f
1
```

```
   ('asdf';23) jappend f
2 3

   (<'mum';'dad') jappend f
4
```

The `jsize` result gives file information, including the indexes of the first and last items.

```
   jsize f
0 5 1408 0

   jread f;0
+---------------+
|first component|
+---------------+

   jread f;1
+-----------+
|1.5 2.5 3.5|
|4.5 5.5 6.5|
+-----------+

   jread f;2
+----+
|asdf|
+----+

   jread f;3
+--+
|23|
+--+

   jread f;4
+---------+
|+---+---+|
||mum|dad||
|+---+---+|
+---------+

   'new' jreplace f;3
3
```

```
   jread f;i.5
+--------------+-----------+----+---+---------+
|first component|1.5 2.5 3.5|asdf|new|+---+---+|
|               |4.5 5.5 6.5|    |   ||mum|dad||
|               |           |    |   |+---+---+|
+---------------+-----------+----+---+---------+
```

Jfiles are documented in the *J Online Documentation*.

Graphical user interface

These days almost no programming task is complete until it is packaged in a graphical user interface (GUI).

Let's add a GUI to your `centigrade` and `fahrenheit` verbs.

There are many steps in building a form and an application. The exact steps you should follow are contained in the series of indented, bulleted items. General discussion and background information is provided in text between these bulleted items.

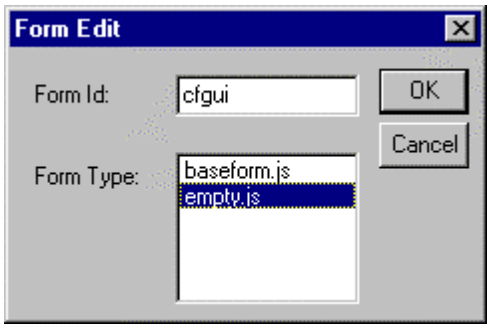
Run your cf.ijs script and make sure that `centigrade` and `fahrenheit` work.

The first step in creating a GUI is to create a form definition. A form definition is stored in a script file just as are all your other definitions.

Create a new script file, save it as a permanent file in the user directory, and start the form editor. The form editor is covered in more detail in the *J Online Documentation* available on the help menu and you should refer to that if you have problems with the following steps, or want more information at this time.

- create a new script file with **File|New IJS**
- save it in the user directory as cfgui.ijs with **File|Save As...**
- start the form editor with **Edit|Form Edit**

You should now have the Form Edit dialog box on the screen.



- type cfgui for your form name
- select the empty form item in the listbox
- press OK to close the dialog

You should now have two new windows on the screen, one in the upper left corner and one in the center of the screen that look something like:

The small form in the corner is the new form you are editing. The Design dialog allows you to customize the form and is in the middle of the screen. The script file cfgui.ijs has had text added to it that defines the form.

Create a static control in your form with the text centigrade. A static control is used to label other controls.

- press the New... button in the Design dialog
- in the New Control dialog select static from the listbox
- type centigrade into the caption edit box
- press OK

New controls are created in the upper left corner of the form. You can drag a control with the mouse. To drag a control, point at it with the mouse, hold down the left mouse button and move it.

- drag the centigrade label down and to the right a bit

Create an edit control with an id of `cid` for the centrigrade value. The id is very important as it is used as the name of the noun used for the control as well as being used in commands to indicate which control they affect.

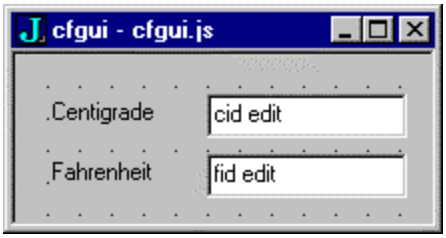
- press the New... button
- select a class of edit
- type in `cid` as the control id
- press OK
- drag the `cid` edit control to the right of the label control

Create a static control with the text Fahrenheit and an edit control with an id of `fid`.

- repeat steps similar to the above to create a Fahrenheit static label and an `fid` edit control

Experiment a bit with moving the controls around. Grab edges or corners to resize them. If you make a mistake you can select a control with the mouse and press the Delete key to delete it and then recreate it.

Your form should now look something like the following:



The form design is finished. Let's exit the form editor and try running the form.

- press OK in the Design dialog

The form definition is now in the cfgui.ijs script. Let's take a look at what the form editor put into the script. The numbers giving screen coordinate will be different, but your script should look something like:

```
NB. base form

CFGUI=: 0 : 0
pc cfgui;
xywh 12 18 40 10;cc ccstatic static;cn "centigrade:";
xywh 56 16 40 14;cc cid edit ws_border es_autohscroll;
xywh 12 40 40 10;cc ccstatic static;cn "Fahrenheit:";
xywh 56 36 40 14;cc fid edit ws_border es_autohscroll;
pas 6 6;pcenter;
rem form end;
)

cfgui_run=: 3 : 0
wd CFGUI
NB. initialize form here
wd 'pshow;'
)
cfgui_close=: 3 : 0
wd'pclose'
)
```

All interactions with forms are done with the `wd` (Window Driver) verb. The `wd` argument is always a string that starts with a command. A string can contain multiple commands separated by semicolons.

The noun `CFGUI` is defined by the conjunction `:` in a manner similar to how verbs are defined. The left argument of `0` creates a noun. It is defined as the lines of characters up to the line which contains only the `)`. It contains the commands that will create the form. Don't worry about the details now, but most of it should make some sense. Commands are followed by parameters and multiple commands on a line are separated by `;`. The `pc` command is a parent create (a form is referred to as a parent). The next line has an `xywh` command that sets a rectangular area on the form and is followed by a `cc` command (create child) that creates one of the controls you put on the form.

After the definition of `CFGUI` you will see that the editor has created a verb called `cfgui_run.` This verb ignores its argument. It executes the `wd` verb with `CFGUI` as an argument. This creates the form, but doesn't show it. The final `wd` with the argument `'pshow;'` will show the parent (form).
At this point the cfgui.ijs script has not been run so the definitions are just text in the script file and have not been defined. After you run the script you are ready to run your application.

- run the cfgui.ijs script with **Run|Window**
- in the ijx window: `cfgui_run 0`

When you execute `cfgui_run 0` you should see your form in the middle of the screen. Typing into the controls and pressing Enter has no effect because you have no code connected to the events yet.

You can close the form manually by executing the `wd` command `reset` that closes all forms.

- in the ijx window: `wd 'reset'`

When you type a value in the centigrade edit control and press Enter you cause an event. An event is identified by the form and the control in which it occurs and the type of the event. An Enter in an edit control is a button event (pressing enter in an edit field is analogous to pressing a button control). So, the event of interest here is for form cfgui, control `cid`, and is a button event.

When an event occurs, a verb called the event handler is executed. The name of the verb that is executed is determined by the event. The name of an event handler is made up of three parts: `formid_controlid_event`. So, the event handler of interest has the name `cfgui_cid_button`.

The event handler `cfgui_cid_button` should convert the value from the `cid` edit control to Fahrenheit and then display that result in the `fid` edit control.

The form editor can automatically create a skeleton of this event handler for you. In the form editor, hold down the Ctrl key and click a control, and you will be switched to editing in the script at the definition of the verb that handles the event for that control.

You closed the form editor, so the first thing is to restart the form editor. Select the cfgui.ijs script and start the form editor with **Edit|Form Edit**.

- select cfgui.ijs (titlebar highlighted)
- start form editor with **Edit|Form Edit**

Your form should again appear open for editing in the corner of your screen.

- hold down the Ctrl key and click the `cid` control

You should be positioned at the skeleton definition of `cfgui_cid_button` in the cfgui.ijs script. You need to define that verb. When the event handler is executed the noun `cid` will automatically have the value of the contents of the edit field. It will be a string and you need to convert that to a number with the `". ` primitive.

```
t =. 0 ". cid
```

The next thing is to convert that centigrade value to Fahrenheit.

```
t =. fahrenheit t
```

The noun `t` is the number you want to display in the `fid` edit control. The number must be converted to a string before it can be shown in an edit field. Use `":` (format) for this.

```
t =. ": t
```

Finally, write the text string to the `fid` edit field.

```
wd 'set fid *' , t
```

The `wd` argument has a command of `set`, the id of the control to set, and `t` contains the data to set. The indicates all the following data is the text to set in the control.
Add these sentences to the definition in the cfgui.ijs script.

```
cfgui_cid_button=: 3 : 0
t =. 0 ". cid
t =. fahrenheit t
t =. ": t
wd 'set fid *', t
)
```

Be careful to type this correctly into your script.

You return to the form editor by holding down the Ctrl key and clicking the script window.

- add the sentences to the definition of `cfgui_cid_button`
- hold the Ctrl key and click the script to return to the form editor
- press OK in the Design dialog

At this point the cfgui.ijs script has not been run so the changes are just text in the script file.

- run the script with **Run|Window**
- in the ijx window run the application: `cfgui_run 0`

You should see your form in the middle of the screen. Type a number into the centigrade field and press Enter. The Fahrenheit value should display in its field.

If you type into the Fahrenheit field and press Enter nothing happens. This is because you have not provided a handler for that event. If an event handler verb is not defined, the event is ignored. Let's define the event handler for Enter in the Fahrenheit field now. Start the form editor and hold down the Ctrl key and click the `fid` control to get to the definition of the verb

for that event. The definition for `cfgui_fid_button` is similar to that of `cfgui_cid_button`.

```
cfgui_fid_button=: 3 : 0
t =. 0 ". fid
t =. centigrade t
wd'set cid *', ": t
)
```

- select cfgui.ijs (titlebar highlighted)
- start the form editor with **Edit|Form Edit**
- hold down the Ctrl key and click the `fid` control
- add the `cfgui_fid_button` definition to the script
- hold the Ctrl key and click the script to return to the form editor

- press OK in the Design dialog

- run the script with **Run|Window**
- in the ijx window: `cfgui_run 0`

Now when you type a value in the Fahrenheit field and press Enter it will be converted and display in the centigrade field.

Finally, add a close button so that the form will be able to close itself. The event handler will be as follows:

```
cfgui_close_button=: 3 : 0
wd 'pclose'
)
```

The `wd` command `pclose` (parent close) closes the form.

- in the ijx window: `wd 'reset'`
- start the form editor with **Edit|Form Edit**
- press the New... button
- select a class of button
- type `close` as the control id
- type Close in the caption field
- press OK

- drag the Close button to the right side of the form

- hold down the Ctrl key and click the Close button
- add the `wd 'pclose'` sentence to the definition

- hold the Ctrl key and click the script to return to the form editor
- press OK in the Design dialog
- run the script with **Run|Window**
- in the ijx window: `cfgui_run 0`

When you tire of doing conversions you can press the Close button to close your form.

Congratulations! you have written a GUI application in J. It is simple and has rough edges, but you are over the high hurdles.

Data processing

Applications typically have a GUI part and a *data processing* (DP) part. The DP part is the actual calculations and data manipulation. A good application implementation will be modular and this implies a clear distinction between the GUI and the DP parts.

In this section you will develop the DP part of a simple application. In the next section you will develop the GUI part.

The DP part of the application is specified as follows:

> The input is the name of a text file. The output is a string that displays as a table that contains: the file name, a count of lines, a count of characters, and a row for each distinct character in the file and a count of how many times it appears in the file. The rows of distinct characters should be sorted by their counts.

You'll be working with files, so load the file utilities.

```
   load 'files'
```

Create a simple text file to use as test data.

```
   fn =. 'user\text.txt'
   data =. 'abc' , LF, 'bc' , LF, 'b' , LF
   data fwrite fn
9
   fread fn
abc
bc
b
```

You need to define a verb `report` that takes a filename as an argument and returns the specified result. You'll build pieces of the definition in the ijx window and then put them all together into the definition in a script.

The input is a filename and in the `report` verb it will have the name `y.` , so start by working with y. in the ijx window.

```
   y. =. 'user\text.txt'
```

Read the file.

```
   d =. fread y.
```

The report will have two columns. The first column will be the labels 'File:', 'Lines:', 'Chars:', and each distinct character in the file. The second column will be the value for that row. Since the data is a mixture of text and numbers it makes sense to build the result as boxed data.

Create a noun with the fixed labels.

```
   r =. 'File:' ; 'Lines:' ; 'Chars:'
   r
+-----+------+------+
|File:|Lines:|Chars:|
+-----+------+------+
```

The values for those labels are calculated as follows:

```
   y. ; (+/ LF = d) ; #d
+-------------+-+-+
|user\text.txt|3|9|
+-------------+-+-+
```

The dyad `,.` (stitch) can connect these two lists into a table.

```
   r =. r ,.  y. ; (+/ LF = d) ; #d
   r
+------+-------------+
|File: |user\text.txt|
+------+-------------+
|Lines:|3            |
+------+-------------+
|Chars:|9            |
+------+-------------+
```

The next thing is to add the rows with the characters and their frequency counts. The letter is the label and the count is the value, so it just adds more items to `r`. Let's postpone that part of the problem, and work instead on converting the boxed table to the string result required by the spec. Use a comment to mark the bit we are skipping over for now.

```
   NB. need to add frequency rows to r here
```

The numbers in the second column need to be converted to characters. The easiest way to do this is to convert the contents of each box to characters. The characters are already characters and are not affected, but any numbers will be converted.

```
   r  =. ":each r
   r
+------+-------------+
|File: |user\text.txt|
+------+-------------+
|Lines:|3            |
+------+-------------+
|Chars:|9            |
+------+-------------+
```

The display of `r` with all characters looks the same, but each box now contains characters.

The next step is interesting and the details are left for you to puzzle out. It adds a TAB after each label and an LF after each value. In the final result the TAB separates the label from its value, and the LF causes a new line for the next label. The boxed display shows the TAB and LF as blanks, but they really are in there.

```
   r =. r ,each"1 1 TAB;LF
   r
+-------+--------------+
|File:  |user\text.txt |
+-------+--------------+
|Lines: |3             |
+-------+--------------+
|Chars: |9             |
+-------+--------------+
```

The monad `;` (raze) opens all the boxes and assembles a string result.

```
   ; r
File:  user\text.txt
Lines: 3
Chars: 9
```

You are ready to define your verb `report`. Create a new script and save it as user\textdp.ijs. Putting together the ijx experiments, add the following definition for `report` to the script.

```
report =: 3 : 0
d =. fread y.
r =. 'File:' ; 'Lines:' ; 'Chars:'
r =. r ,. y. ; (+/ LF = d) ; #d
NB. need to add frequency rows to r here
r =. ":each r
r =. r ,each"1 1 TAB;LF
; r
)
```

Run the script and test `report`.

```
   report fn
File:          user\text.txt
Lines:         3
Chars:         9
```

Now calculate the frequency rows. You need a verb `freq` that returns a table of boxes where the first column is the distinct characters and the second column is the count of times they are in the file. The argument to `freq` is the file data and inside `freq` it will have the name `y.`, so let's start with `y.` defined as the file data.

```
   y. =. fread fn
```

The data can include TAB, CR, and LF characters and they should be removed. The dyad `-.` (less) can remove these unwanted characters.

```
   d =. y. -. TAB,CR,LF
   d
abcbcb
```

The utility `nubcount`, defined by script misc.ijs, returns a table of boxes with a first column containing the distinct items in its argument and the second column containing the counts.

```
load 'misc'
   nc =. nubcount d
   nc
+-+-+
|a|1|
+-+-+
|b|3|
+-+-+
|c|2|
+-+-+
```

To sort the items by the counts you need to get the counts into a list.

```
   > 1 {"1 nc
1 3 2
```

The dyad `\:` (sort down) sorts the items of its left argument based its right argument.

```
   nc \: > 1 {"1 nc
+-+-+
|b|3|
+-+-+
|c|2|
+-+-+
|a|1|
+-+-+
```

Put this all together and add the following definition to your script.

```
freq =: 3 : 0
d =. y. -. TAB,CR,LF
nc =. nubcount d
nc \: > 1 {"1 nc
)
```

Run the script and test `freq`.

```
   freq fread fn
+-+-+
|b|3|
+-+-+
|c|2|
+-+-+
|a|1|
+-+-+
```

You can now use `freq` in your `report` verb. Modify the `NB.` comment line in `report` to be:

```
r =. r , freq d
```

Run the script and test `report`.

```
   report fn
File:          user\text.txt
Lines:         3
Chars:         9
b              3
c              2
a              1
```

Try it on other text files.

You have finished the data processing part.

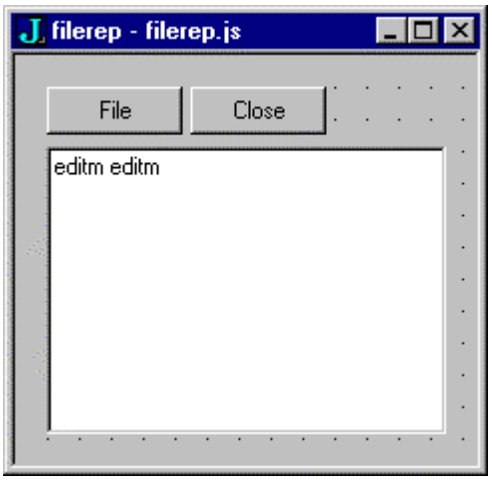
GUI

The GUI part of the application is specified as follows:

> The form should have a File button, a Close button, and a multiline edit control. The File button allows the user to select a text file. The report on the selected text file is displayed in the multiline edit control.

You need to design the form and define the event handlers.

The GUI definitions will be in a different script from the DP definitions to keep clear the distinction between the two parts. Create a new script file and save it as user\textgui.ijs. Start the form editor and design the form. The File button should have an id of `file` and the Close button should have an id of `close`. The large edit control is a multiline edit control that has a class of editm in the New Control dialog. The multiline edit control should have an id of `editm` (the default is , so you must specify `editm )`. The form should look like the following:



You need to add event handlers for the Close and File buttons. The code for the event handlers is in the following listing. This listing should be similar to your final textgui.ijs script.

```
FILEREP=: noun define
pc filerep;
xywh 9 7 34 14;cc file button;cn "File";
xywh 47 7 34 14;cc close button;cn "Close";
xywh 9 27 119 134;cc editm editm ws_border es_autovscroll;
pas 6 6;pcenter;
rem form end;
)

filerep_run=: 3 : 0
wd FILEREP
NB. initialize form here
wd 'pshow;'
)
filerep_close_button=: 3 : 0
wd'pclose'
)

filerep_file_button=: 3 : 0
p =. '"" "" "" "Text(*.txt)|*.txt" ofn_filemustexist'
fn =. wd 'mbopen ' , p
if. 0 ~: #fn do.
 wd 'set editm *' , report fn end.
)
```

The only part that is new is the use of the `wd` command `mbopen`. This command brings up the *common file open dialog* box that allows the user to select a file. Local `p` contains the parameters for the `mbopen` command. These parameters are critical and must be defined properly. If you want to know more about the `mbopen` parameters, you can check in the *J Online Documentation.*

The result of the `mbopen` command is the file name selected by the user. If the user pressed cancel in the open dialog the result will be an empty string and there is nothing to do. If `fn` is not empty then you execute `report fn` to generate the report and set it into the `editm` control.

The `*` in the line `wd 'set editm *' , report fn end.` indicates that the rest of the string, which is the result of `report fn,` is the data to set into the editm multiline edit control.

Run the textdp.ijs and textgui.ijs scripts and then start the application.

```
   filerep_run 0
```

Press the File button and select your user\text.txt file and press OK. Try other text files.

The application uses definitions from four scripts: textdp.ijs, textgui.ijs, files.ijs, and misc.ijs. It makes sense to create a single script that will load all the scripts and then run the application.

Create a new script file, save it as user\textapp.ijs, and add the following lines.

```
NB. this application reports file character frequencies
load 'files'
load 'misc'
load < 'user\textdp.ijs'
load < 'user\textgui.ijs'
filerep_run 0
```

Save the script. Close J and restart it to get a clean slate. Run the application by using Run|File to run the script user\textapp.ijs.

Where to go from here

If this is your first skim reading, or your second more detailed reading, or if you feel you haven't quite mastered all the material, then this is the point to go back to the beginning and have another go in more detail, and perhaps use a bit more elbow grease and get those hands working on the keyboard.

If you have mastered this material, then it is time to move on. You can start digging in on your own, but a bit more time with the other manuals is probably worthwhile.

J Online Documentation

Take the time to familiarize yourself with the documentation available from the J help menu. It is worthwhile to quickly browse through all the material so you'll know where to look when something comes up.

There is information on how to use the standard libraries and packages that are provided with the system. If you prefer reinventing the wheel, then ignore it, but if you want a head start, take a look.

There is material on GUI programming and the window driver. If you are content with the ijx and ijs windows and are your own end user, then ignore it. If you want to build more complete applications and possibly provide them for other end users, then you need to spend time learning this stuff.

J Dictionary

By now you should be familiar with the vocabulary and definition part of the *J Dictionary*.

This primer has introduced most of the *J Dictionary* concepts, although in a simpler way that is restricted to specific situations. As mentioned before, the *J Dictionary* is a reference book and its descriptions are both as concise as possible, and at the same time as complete as possible, with more emphasis on complex and tricky situations for experts, than on the mundane ones for beginners.

It would be worthwhile to read the sections that preceed the individual definitions. Much of it will make sense, and some of it will either answer questions that had arisen in your mind, or just as likely raise interesting new questions. Based on the experience of seasoned J programmers, you'll find yourself reading and rereading this material, and on every reading you'll learn something new.

You are also now more than ready to attack the Introduction. You should definitely work your way through the first 20 lessons. It shouldn't be too tough and you'll learn a lot.

Especially if you work through the exercises!

The remaining lessons are more difficult and will take more effort to master.

Working your way through the Sample Topics is one of the best ways to meet some of the primitives in action. You'll be continuously challenged and constantly referring to the

definitions and experimenting with the system to try to understand what is going on. In the end you'll be a much better J programmer, to say nothing of learning a fair bit of math and computer science!

J Phrases

The earlier you take a look at the J Phrases book and find that it starts to make sense, the better. There are few better ways to learn a language than by reading material written by experts, and that is exactly what this book is. Think of it as a collection of the greatest J short stories; you'll delight and marvel as you read through it, but it can also be a practical addition to your kit of software tools. If you run across a particular requirement in one of your applications for a tight little kernel of math, statistics, or data manipulation, chances are good that you can refer to the J Phrases and find what you want, or at least a starting point.

end.

Finally, the end of the structured learning! Welcome to the ranks of J programmers. You are now free to put this stuff to use in your own way. Good luck and enjoy.

INDEX

A

A few more primitives, 52
adverb, 85
Adverb, 78
Agreement, 72
Alphabet, 7
Ambivalence, 12
Amend (modify selected), 95
arguments, 67
Array, 63
Atom, 62
Axis, 64

B

back-quote, 8
Basic way of adding lists, 47
Box - monad <, 86

C

cell, 68, 70
centigrade, 23
Checkpoint A, 16
Checkpoint B, 39
Checkpoint C, 46
Checkpoint D, 61
Checkpoint E, 78
Checkpoint F, 111
Comment, 11
Comparative, 43
Component files, 118
conjunction, 74, 85
Conjunction, 85
Control structure, 44
copula, 11
Cut, 100

D

Data processing, 127
Debug - an error, 42
Debug - stepping through a verb, 40
Debug global, 33
definition, 23, 25, 105
dimension, 64
divided by, 12
dot, 8
double-quote, 8
dyad, 12, 25
Dyad, 12

E

Each, 102
Empty Array, 65
end., 136
Environment, 3
error, 42
Error, 12
exact, 58
Experiment, 5
Explicit, 109
Explicit-to-tacit translator, 109

F

Fahrenheit, 23
file, 26
files, 118
Files, 113
Foreign !:, 112
Fork, 104
Frame and cell, 68
From - dyad { (selecting items), 91
From (boxed indexes), 92
From (scattered indexing), 95
function, 7

G

Get started, 4
Global, 31
Graphical user interface, 120
GUI, 132

H

Hook, 104
How to use this book, 2

I

ijx window, 4
indexing, 91
Inexact numbers, 58
Insert adverb, 79
Item, 69

J

J Dictionary, 135
j locale, 37
J Online Documenationl, 135
J Phrases, 136
J way of adding lists, 50
jfile, 118

K

k-cell, 70

L

line of code, 7
Link - dyad ;, 87
List, 62
load, 38
Local, 30
Locale, 35

M

Macintosh, 3
minus, 8
modify selected, 95
monad, 12
Monad, 13, 25
Monad and dyad definition, 25

N

Name, 10
NB., 11
Negative number, 9
Noun, 8, 9
numbers, 58
Numeric constant, 16

O

Open - monad >, 89
Order of evaluation, 21
Order of execution - adverbs & conjunctions, 85

P

Parentheses, 21
period, 8
Phrases, 136
Plot, 54
Plot locale, 55
Precedence, 20
precision, 56
Primitive, 10
Print precision, 56
profile, 6
Purpose of this book, 2

Q

quote, 8

R

rank, 71
Rank, 65
, 74
reciprocal, 13
Result shape, 76

S

Script file, 26
Script load, 38
Selecting without from, 98
sentence, 7
Sentence, 8
shape, 76
Shape, 64
Single atom array, 66
singleton, 66
Space, 19
Standard profile, 6
Start here, 1
stepping through a verb, 40
String, 16

structure, 44

T

Table, 62
Table adverb, 81
tacit, 109
Tacit definition, 105
Terminology, 7
Tolerance, 59

U

underbar, 8
UNIX, 3
user interface, 120

V

verb, 7
Verb, 8
Verb arguments, 67
Verb definition, 23
Verb rank, 71
Vocabulary, 13

W

When =. and =: are the same, 34
When they aren't, 34
Where to go from here, 134
Why J, 1
Windows, 3
Word, 8
Word formation, 17

Y

Your background, 2

Z

z locale, 38