Reprint from
The *Mathematica* Conference

June, 1992 Boston, MA

653.12.10.1992

Programming in *Mathematica*

Roman E. Maeder

1 Introduction

Mathematica includes a rich and powerful programming language. Unlike the usual languages like Basic, C, etc., it is not restricted to having a small number of data types but allows users to perform all symbolic computations of *Mathematica*.

The manual can only hint at the possibilities. It explains all the features but does not show which ones to use for a particular problem or how to fit things together into larger programs.

Through design and tradition each programming language has developed a certain preferred style of good programming. It is possible to solve the same problem in many different ways, but there is usually some idea about what is a good or a bad program. *Mathematica* is new and there are, as yet, few good programs that have established such a style. In this tutorial, I want to present examples of what the designers of *Mathematica* think is good programming style.

First, we look at mechanisms for encapsulating program modules, so-called packages. In the second part of the tutorial we compare various programming styles, especially procedural programming, functional programming, and mathematical programming.

These tutorial notes are adapted from the second edition of *Programming in Mathematica*, Addison-Wesley, 1991. All sample programs mentioned are part of the standard *Mathematica* distribution from Wolfram Research (Version 2.0). Cross-references in this tutorial refer to *Programming in Mathematica*, Second Edition.

2 Encapsulation Mechanisms for Packages

The first important concept is that of a *context*. Contexts provide a way of keeping variables in different packages separate. Another benefit of contexts is the ability to hide information from the user of our package. We want to keep local variables and auxiliary functions hidden from the user.

Next we present a *skeletal package*. It contains all the commands for setting up the proper contexts and can be used as a starting point for your own packages. Using such a template guarantees a certain uniformity in the overall appearance of a package, making it easier to understand packages that somebody else has written.

■ 2.1 All about Contexts

There are two global variables inside *Mathematica* that control the creation of new symbols and the look-up of existing ones.

`$Context`	the current context
`$ContextPath`	the list of contexts to search

These two variables govern the look-up of contexts.

When *Mathematica* encounters a symbol in the input that you type or that is read from a file, it searches the current context and then all the contexts on the context path for this symbol. If it cannot find one, a new symbol is created in the current context. See also Sections 2.6.8 through 2.6.10 of The *Mathematica* Book for additional explanations on contexts.

Normally, the values of the variables `$Context` or `$ContextPath` should not be changed directly. It is better to use the commands provided for context manipulation. They make the necessary changes to these variables and also perform error checking.

`BeginPackage["`*Context*`` ` ``"]`	start a package
`EndPackage[ ]`	end a package
`Begin["`*Context*`` ` ``"]`	change the current context
`End[ ]`	return to the previous context

Commands to manipulate contexts.

To see how these commands change the values of `$Context` and `$ContextPath`, let us look in detail at what happens when a package is read in. We simulate reading in a stripped-down version of the example from Chapter 1. We take out all the actual code and are left with just a few declarations:

```
BeginPackage["ComplexMap`"]

CartesianMap::usage = "CartesianMap[f,...] plots a map."

Begin["`Private`"]

huge = 10.0^6

MakeLines[points_] :=
        Module[{lines, newpoints},
                        .
                        .
                        .
        ]

CartesianMap[ f_,... ] := ...

End[ ]
EndPackage[ ]
```

An excerpt from `ComplexMap.m`.

In the following table, we show the effect of each line in the package on the values of the variables `$Context` and `$ContextPath`. The entries are filled in only if the command changed them. We also show the fully qualified names of the symbols encountered in the input. The first line is not part of the package. It simulates some computations that we did before reading in the package. At the end, we return to this global context and use the function `CartesianMap[ ]` that we have just defined.

	Command	Symbols	`$Context`	`$ContextPath`
1	`Factor[y^2-1]`	`` System`Factor ``, `` Global`y ``	`` Global` ``	`` {Global`, System`} ``
2	`<<ComplexMap.m`			
the following lines are read from the package				
3	`` BeginPackage["ComplexMap`"] ``	`` System`BeginPackage ``	`` ComplexMap` ``	`` {ComplexMap`, System`} ``
4	`CartesianMap::usage = "..."`	`` ComplexMap`CartesianMap ``		
5	``Begin["`Private`"]``	`` System`Begin ``	`` ComplexMap`Private` ``	
6	`huge = 10.0^6`	`` ComplexMap`Private`huge ``		
7	`MakeLines[points_] :=`	`` ComplexMap`Private`MakeLines `` `` ComplexMap`Private`points ``		
8	`Module[{lines,...},...]`	`` ComplexMap`Private`lines ``		
9	`CartesianMap[ f_,...] :=`	`` ComplexMap`CartesianMap `` `` ComplexMap`Private`f ``		
10	`End[ ]`		`` ComplexMap` ``	
11	`EndPackage[ ]`		`` Global` ``	`` {ComplexMap`, Global`, System`} ``
here we return to the top level				
12	`CartesianMap[Sin,...]`	`` ComplexMap`CartesianMap `` `` System`Sin ``		
13	`MakeLines[...]`	`` Global`MakeLines ``		

Here is a line-by-line discussion of what happens exactly in the example above:

1. This line shows a typical calculation that could have been done just before reading in the package `ComplexMap.m`. The values of the variables `$Context` and `$ContextPath` are the default ones when *Mathematica* starts up.
2. Here we give the command to read in the package. The following lines are the contents of the package; they are not typed in by the user.
3. `BeginPackage[ ]` sets the value of `$Context`. `$ContextPath` always becomes `{`*PackageContext*`` ` ``, `` System`} `` independent of what it was before. Note especially that ``Global` `` is not on the context search path. In a package there is, therefore, no danger of wrongly accessing any objects that have been defined in the *Mathematica* session so far. The package always starts in a clean state.
4. Defining a usage message for `CartesianMap` creates the symbol since it is not built in and does not exist in the current context. It is, therefore, created in the current context, which is the package context ``ComplexMap` ``.
5. The command `Begin["`*Context*`` `"] `` changes the current context. It does not affect the context path. The argument ``"`Private`"`` begins with a context mark `` ` `` and the context is, therefore, a subcontext of the current context, with full name ``ComplexMap`Private` ``. The consequence is that newly created symbols will be put in the context ``ComplexMap`Private` ``.
6. The variable `huge` is local to the package. Its value is, however, preserved from one call of `CartesianMap[ ]` to the next, since it is not declared as a local variable to `CartesianMap[ ]`. Such variables are sometimes called *static variables*.
7. The command `MakeLines[ ]` too is local to the package.
8. The variable `lines` is local to the command `MakeLines[ ]`. Its value is not preserved across calls.
9. Finally, here is the definition of the function `CartesianMap[ ]`. The symbol `CartesianMap` has already been created (in line 4) in the context ``ComplexMap` ``. Since this context is on the context path the symbol is found there and the definition is for the existing symbol.
10. The command `End[ ]` undoes the previous `Begin[ ]` and restores the current context to ``ComplexMap` ``. The context ``ComplexMap`Private` `` is not on the context path, and any symbols defined in this context are therefore no longer accessible.
11. `EndPackage[ ]` restores the current context to what it was before the command `BeginPackage[ ]`. The context path is also restored, but the package context is added in front of it.
12. Now we are back in our interactive *Mathematica* session and can use the command `CartesianMap[ ]` since the context in which it was defined appears in the context path.
13. The command `MakeLines[ ]`, however, cannot be used. The symbol is not found in its context ``ComplexMap`Private` ``. A new symbol is created in the global context that has nothing to do with the other one.

The net effect of reading in a package is to add a new context in front of the context path and to define some functions in this context. Functions that are made available in a self-contained program unit for use outside of it are said to be *exported* from it. The package controls which of its functions it wants to export. This mechanism of explicitly exporting objects from a program unit is found in one form or another in most modern programming languages. It is an important software engineering tool.

Context names themselves are not symbols and must be quoted when used as arguments of a command. A symbol in any context can be specified with its fully qualified name as *Context*`` ` ``*symbol*. It is therefore not entirely true that the auxiliary function `MakeLines[ ]` is inaccessible. By using ``ComplexMap`Private`MakeLines``, we could access it even though the context in which it is defined is not on the search path. This practice is, of course, strongly discouraged.

2.2 Packages That Use Other Packages

Recall that the command `BeginPackage["`*Context*`` ` ``"]` resets the context path to `{`*Context*`` `, System`} ``, independent of its former value. One consequence is that symbols that the user of your package has defined before reading in the package do not get in the way since such symbols are defined in the global context. But it also means that any other packages that have been read in are not accessible inside your package. You might want to use a command defined in another package inside your own package, however.

Importing Another Package

`BeginPackage[ ]` has optional arguments that specify contexts that are to be left on the search path. If they are not already on the search path, then they are first read in using the command `Needs["`*Context*`` ` ``"]` implicitly. An example of such a package is the standard package `Graphics/Shapes.m`. It defines a command to rotate a three-dimensional graphics object and uses the standard package `Geometry/Rotations.m` to compute the rotation matrix. Here is a small excerpt showing how this is set up:

```
BeginPackage["Graphics`Shapes`", "Geometry`Rotations`"]

RotateShape::usage = "..."

Begin["`Private`"]

RotateShape[ shape_, phi_, theta_, psi_ ] :=
        Block[{rotmat = RotationMatrix3D[N[phi], N[theta], N[psi]]},
                        .
                        .
                        .
        ]

End[ ]
EndPackage[ ]
```

Part of `Graphics/Shapes.m`

The function `RotationMatrix3D[ ]` used inside `RotateShape[ ]` comes from the package `Geometry/Rotations.m`. Here is the detailed analysis of the context changes in the same format as in Section 2.1.

	Command	Symbols	$Context	$ContextPath
1	``BeginPackage["Graphics`Shapes`", "Geometry`Rotations`"]``		``Graphics`Shapes` ``	``{Graphics`Shapes`, Geometry`Rotations`, System`}``
2	`RotateShape::usage = "..."`	``Graphics`Shapes`RotateShape``		
3	``Begin["`Private`"]``		``Graphics`Shapes`Private` ``	
4	`RotateShape[shape_,...] :=`	``Graphics`Shapes`RotateShape`` ``Graphics`Shapes`Private`shape``		
5	`Block[{rotmat= RotationMatrix3D[..]},..]`	``Graphics`Shapes`Private`rotmat`` ``Geometry`Rotations`RotationMatrix3D``		
6	`End[ ]`		``Graphics`Shapes` ``	
7	`EndPackage[ ]`		``Global` ``	``{Graphics`Shapes`, Geometry`Rotations`, Global`, System`}``

As with the command `Needs[ ]` itself, no package is read in if the context given as an optional argument to `BeginPackage[ ]` is already on the search path. A package is therefore read in only once even if it is used inside several packages that you read in your session one after the other.

The optional arguments of `BeginPackage[ ]` specify those packages that are needed inside your package. Such packages are said to be *imported* in your package. Mentioning imported packages at the start of your package is also an important element of documentation. It makes all dependencies on other packages explicit. Stating exactly what your package depends on is another important principle of software design.

Hidden Import

Any package read in through the mechanism just explained is left on the search path after your package has been read in. It is therefore also made available to the user of your package. This is sometimes desirable or does not matter very much. But it could also hide other functions that the user of your package has defined or read in before your package. The user should not have to be concerned about other packages that are made available as a consequence of reading in yours. There is a way of making another package available to your own without leaving it on the search path after the end of your package. Instead of mentioning the package as as an optional argument of `BeginPackage[ ]`, you can also read it in with a `Needs[ ]` command inside your package, after the call to `BeginPackage[ ]`. The fact that your package imports this other package is then hidden from the user, and this method is therefore termed *hidden import*. Here is the modified code fragment of `Shapes.m` followed by the detailed analysis of the context changes.

```
BeginPackage["Graphics`Shapes`"]

Needs["Geometry`Rotations`"]

RotateShape::usage = "..."

Begin["`Private`"]

RotateShape[ shape_Graphics3D, phi_, theta_, psi_ ] :=
        Block[{rotmat = RotationMatrix3D[N[phi], N[theta], N[psi]]},
                        .
                        .
                        .
        ]

End[ ]
EndPackage[ ]
```

A modified version of `Graphics/Shapes.m`

	Command	Symbols	$Context	$ContextPath
1	`BeginPackage["Graphics`Shapes`"]`		`Graphics`Shapes``	`{Graphics`Shapes`, System`}`
2	`Needs["Geometry`Rotations`"]`			`{Geometry`Rotations`, Graphics`Shapes`, System`}`
3	`RotateShape::usage = "..."`	`Graphics`Shapes`RotateShape`		
4	`Begin["`Private`"]`		`Graphics`Shapes`Private``	
5	`RotateShape[ shape_,...] :=`	`Graphics`Shapes`RotateShape` `Graphics`Shapes`Private`shape`		
6	`Block[{rotmat= RotationMatrix3D[..]},..]`	`Shapes`Private`rotmat` `Geometry`Rotations`RotationMatrix3D`		
7	`End[ ]`		`Graphics`Shapes``	
8	`EndPackage[ ]`		`Global``	`{Graphics`Shapes`, Global`, System`}`

After reading in this version of `Shapes.m` the context `Rotations`` is not on the search path and cannot be accessed by the user. Note that inside `Shapes.m` the order of the contexts `Graphics`Shapes`` and `Geometry`Rotations`` on the search path is reversed. This poses no problems since the current context is still either `Graphics`Shapes`` or `Graphics`Shapes`Private`` and so new symbols like `RotateShape` are still created in the correct context and not in `Geometry`Rotations``.

There is one potential problem with this hidden import of packages. If another package also tries to import a package that has already been read in somewhere else, then it would be read in twice. Many packages are not designed to be read in more than once. If packages are requested only through the command `BeginPackage[ ]` or through `Needs[ ]` *before* the command `BeginPackage[ ]` and not through hidden import or the explicit command `<<`*Package*, this cannot happen. Hidden import is therefore best used only for packages that were explicitly designed to be used that way, for example auxiliary packages for one of your own packages.

Context Names and Package Names

When a context name is specified in the `Needs[ ]` command or as one of the optional arguments of `BeginPackage[ ]`, *Mathematica* has to derive the name of a file to read in. By convention, the name of the file is of the form `Context.m` where *Context* is the context name without the final `` ` ``. If the context name is composed of several parts separated by context marks, then these context marks are translated into appropriate path separators for your file system (/ under Unix, for example). The command ``Needs["Geometry`Rotations`"]`` would try to read in the package `Geometry/Rotations.m`. This is the usual case since all standard packages are put into subdirectories of the **Package** directory. The translation between context names and file names is performed by the function `ContextToFilename[ ]` that is set up correctly on each different machine. To further allow machine-independent programming, the `Get[ ]` command also accepts a context name instead of a file. You can therefore read in the package `Geometry/Rotations.m`, for example, using ``<<Geometry`Rotations` ``. Under Unix, this context name would then be translated back to `Geometry/Rotations.m`. See also Subsection 2.10.4 of The *Mathematica* Book.

We turn on tracing for the function `ContextToFilename[ ]`.

```
In[1]:= On[ContextToFilename]
```

When we specify a context name as an argument of `Get[ ]` we see that it calls `ContextToFilename[ ]` to obtain a valid file name for the computer currently in use.

```
In[2]:= << Geometry`Rotations`

   ContextToFilename::trace:
   ContextToFilename[Geometry`Rotations`] —>
    Geometry/Rotations.m.
```

`Needs[ ]` has an optional second argument that allows you to specify a different file name. This, however, is not possible in the `BeginPackage[ ]` command. One way around it is to precede `BeginPackage[ ]` by a call to `Needs[ ]`. If the package context ``Rotations` `` were in the file `TestRotations.m` instead of `Rotations.m`, then the following is a possible way to import this package into `Shapes.m` in the same way as was done in Subsection 2.2.1:

```
Needs["Rotations`", "TestRotations.m"]

BeginPackage["Graphics`Shapes`", "Rotations`"]
                   .
                   .
                   .
EndPackage[ ]
```

Another variant of `Graphics/Shapes.m`

■ 2.3 Protection of Symbols in a Package

Protecting Exported Symbols

Built-in commands are protected against accidental modification by a user. The commands `Protect[ ]` and `Unprotect[ ]` are discussed in Section 2.4.12 of The *Mathematica* book. The commands defined in a package can be protected in the same way and will then behave just like built-in ones. All you have to do is to give the command `Protect[`$symb_1$`, `$symb_2$`, ...]` at the end of the package (between `End[ ]` and `EndPackage[ ]`), where the $symb_i$ are all the symbols to be exported from the package. You do not have to protect auxiliary functions defined in the private context since they cannot be accessed by the user anyway. Here is a version of the example `ComplexMap.m` from Chapter 1 with added protection:

```
BeginPackage["ComplexMap`"]

CartesianMap::usage = "..."
PolarMap::usage = "..."
```

```
Begin["`Private`"]
                 .
                 .
                 .
End[ ]

Protect[CartesianMap, PolarMap]

EndPackage[ ]
```

Protecting symbols in `ComplexMap.m`

A package with protected symbols cannot be read in twice. The second time around the symbols would already be protected when the rules and usage messages are defined, giving you error messages. Therefore, you usually do not include the `Protect[ ]` command while a package is still under development. On the other hand, you might want to put in a statement `Clear[`$symb_1$`, `$symb_2$`, ...]` near the beginning of the package. This makes sure that all old rules are forgotten when you modify the rules and read in the modified package into the same *Mathematica* session many times during debugging. After having found the next-to-last bug in your code, you then insert the protection commands and remove the `Clear[ ]` at the beginning. (The *last* bug in any piece of code is, of course, invariably found by the first user of the code and never by the programmer.) If you develop your package as a Notebook you can enable or disable the `Protect[ ]` command easily by marking the cell that contains it as either *active* or *inactive*. If you prepare it as an ordinary file, you can comment out the command with `(* Protect[...] *)`.

The protection of these symbols can still be removed by using `Unprotect[ ]`, just as for built-in symbols. If you really want to disable this, you can lock the symbols in addition to protecting them. The attributes of a locked symbol cannot be changed at all so this action is final. In the above example you could add the command

```
SetAttributes[{CartesianMap, PolarMap}, Locked]
```

after the call to `Protect[ ]`.

Unprotecting System Symbols

A complementary case arises if a package defines additional rules for built-in functions. In this case, the affected symbols have to be unprotected at the beginning of the package and should be protected again at the end.

Here is an example package `ReIm.m` that defines additional rules for the built-in functions `Re[ ]` and `Im[ ]` that allow simplification of symbolic expressions involving them. In *Mathematica*, variables that do not have a value are treated as standing for any quantity, including complex numbers. The expression `Re[x]` does not, therefore, simplify to `x`. In some applications, you assume that variables stand for real numbers only and then you would like to simplify `Re[x]` to `x`.

To declare a variable *var* to be real, set its imaginary part to 0 with the command *var*`/: Im[`*var*`] = 0`. Rules for `Re[ ]` and `Im[ ]` defined in `ReIm.m` will then perform the simplifications. Here are a few easy simplification rules for addition and multiplication:

```
BeginPackage["ReIm`"]

Begin["`Private`"]

Unprotect[Re, Im]

Re[x_] := x  /; Im[x] == 0

Re[x_+y_] := Re[x] + Re[y]
Im[x_+y_] := Im[x] + Im[y]

Re[x_ y_] := Re[x] Re[y] - Im[x] Im[y]
Im[x_ y_] := Re[x] Im[y] + Im[x] Re[y]
```

```
Protect[Re, Im]

End[ ]

EndPackage[ ]
```

The first version of ReIm.m

In[3]:= << ReIm1.m

Nothing is known about x. It does not simplify.

In[4]:= Re[x]i

Out[4]= Re[x]

a is declared to be real. This declaration should be associated with a.

In[5]:= a/: Im[a] = 0;

It does what we expect.

In[6]:= Re[a]

Out[6]= a

This is partially simplified using the knowledge we have about a.

In[7]:= Im[a x]

Out[7]= a Im[x]

One feature of ReIm.m is not quite the way we want it. If the user had already unprotected Re, for example, then reading in our package would protect it again! This should not happen and there is a way around it. The function Unprotect[ ] returns as its value a list of all those of its arguments that it did actually unprotect. Arguments that were already unprotected will be skipped. Instead of protecting all the symbols again at the end, we only protect those that were returned by Unprotect[ ].

```
BeginPackage["ReIm`"]

Begin["`Private`"]

protected = Unprotect[Re, Im]

Re[x_] := x  /; Im[x] == 0

Re[x_+y_] := Re[x] + Re[y]
Im[x_+y_] := Im[x] + Im[y]

Re[x_ y_] := Re[x] Re[y] - Im[x] Im[y]
Im[x_ y_] := Re[x] Im[y] + Im[x] Re[y]

Protect[ Evaluate[protected] ]

End[ ]

EndPackage[ ]
```

A better version of ReIm.m

The return value of the Unprotect[ ] command is saved in a local variable. Its value is then used in the command Protect[ ] and only those symbols are then protected. Protect[ ] does not evaluate its arguments because it operates on the symbols themselves and not on their values. It is thus necessary to force evaluation with the command Evaluate[ ], otherwise the symbol protected itself would be protected.

Here we unprotect the built-in symbol Re.

In[1]:= Unprotect[Re]

Out[1]= {Re}

Re no longer has the attribute Protected.

In[2]:= Attributes[Re]

Out[2]= {Listable}

Im is still protected.

In[3]:= Attributes[Im]

Out[3]= {Listable, Protected}

Now let's read in the package and see what happens to the protection of Re and Im.

In[4]:= << ReIm2.m

Both are as before.

```
In[5]:= Attributes[Re]

Out[5]= {Listable}
In[6]:= Attributes[Im]

Out[6]= {Listable, Protected}
```

The standard package `Algebra/ReIm.m` defines many more such rules for simplification of `Re[ ]`, `Im[ ]`, and `Conjugate[ ]`.

Changes from the First Edition

Mathematica Version 2.0 contains a new command `ComplexExpand[ ]` that can be used in place of `Algebra/ReIm.m` in many cases. It assumes variables as real, unless declared complex. With it, the example from page 9 could be written as follows:

This simplifies `Im[a x]` assuming that `x` is complex-valued, all other variables, including `a`, are assumed real.

```
In[1]:= ComplexExpand[ Im[a x], x ]

Out[1]= a Im[x]
```

■ 2.4 A Skeletal Package

If you look at the examples presented so far and at other packages, you will notice that the framework of a package is mostly the same for all of them. We have developed this framework in Chapter 1 and the preceding sections of this chapter. We are now ready to collect everything together in a template or skeleton for packages. (As with real skeletons, the flesh is missing and has to be provided by the author of the package.) The cornerstones of every package are the context manipulation commands. Then come the usage messages for the functions that are to be exported, then the details about imports of other packages and the protection of symbols.

The package `Skeleton.m` is syntactically correct so it can be read into *Mathematica*. To avoid error messages, you need to provide the three files `Package1.m`, `Package2.m`, and `Package3.m` as well. They need to contain only a few lines that define the contexts.

```
BeginPackage["Package1`"]
EndPackage[]
```

`Package1.m`: one of the imported packages.

If you are writing your own package, you can take `Skeleton.m` as a starting point. Change all the names of the functions and the package itself (including the context name in `BeginPackage[ ]`). You should also make sure to delete any of the features that you do not use, for example, the statements for importing other packages or for defining rules for built-in objects. As explained in Subsection 2.3.1, you might want to comment out the statement to protect the exported symbols while your package is still being debugged. (Under the Notebook front end, the cell containing it can be marked inactive.)

```
(* Skeleton.m -- a skeletal package *)

(* set up the package context, included any imports *)

BeginPackage["Skeleton`", "Package1`", "Package2`"]

Needs["Package3`"]    (* read in any hidden imports *)


(* usage messages for the exported functions and the context itself *)

Skeleton::usage = "Skeleton.m is a package that does nothing."

Function1::usage = "Function1[n] does nothing."

Function2::usage = "Function2[n, (m:17)] does even more nothing."
```

```
Begin["`Private`"]    (* begin the private context *)

(* unprotect any system functions for which rules will be defined *)
protected = Unprotect[ Sin, Cos ]

(* definition of auxiliary functions and local (static) variables *)
Aux[f_] := Do[something]
staticvar = 0

(* error messages for the exported objects *)
Skeleton::badarg = "You twit, you called `1` with argument `2`!"

(* definition of the exported functions *)
Function1[n_] := n
Function2[n_, m_:17] := n m /; n < 5 || Message[Skeleton::badarg, Function2, n]

(* rules for system functions *)
Sin/: Sin[x_]^2 := 1 - Cos[x]^2

Protect[ Evaluate[protected] ]      (* restore protection of system symbols *)
End[]         (* end the private context *)
Protect[ Function1, Function2 ]    (* protect exported symbols *)
EndPackage[]  (* end the package context *)
```

`Skeleton.m`: a template for packages.

3 Procedures and Local Variables

Contexts and packages organize the larger units of a program. We now turn to the smaller units, the procedures and functions that make up a larger program. The principles of information hiding, abstraction, and encapsulation of internal workings apply on this level, too.

■ 3.1 Procedures in *Mathematica*

Strictly speaking, there are no procedures, functions, or subroutines in *Mathematica*. Any definition of the form `f[`*args*`] :=` *body* is a rewrite rule. Whenever the evaluator sees an expression that matches the left side, it is replaced by the right side with the values of the pattern variables substituted. This corresponds closely to a procedure call of a traditional language, a similarity that is intended. The benefits of this setup are that a parameter list of a procedure is not restricted to the usual `proc[`arg_1`,` arg_2`, ...,` arg_n`]`, but can be any pattern. The procedure `CartesianMap[ ]` from chapter 1, for example, uses the form (or *calling sequence*)

```
CartesianMap[ f_, {x0_, x1_, dx_:Automatic}, {y0_, y1_, dy_:Automatic}, opts___ ].
```

It declares optional arguments, a variable number of extra arguments (options), and combines the two ranges into lists, thus adding clarity to the argument structure. Depending on the form of such a rewrite rule, we usually call it either a procedure, function, or transformation rule.

```
SplitLine[vl_] :=
        Module[{vll, pos, linelist = {}, low, high},
                vll = If[NumberQ[#], #, Indeterminate]& /@ vl;
                pos = Flatten[ Position[vll, Indeterminate] ];
                pos = Union[ pos, {0, Length[vll]+1} ];
                Do[ low = pos[[i]]+1;
                    high = pos[[i+1]]-1;
                    If[ low < high, AppendTo[linelist, Take[vll, {low, high}]] ],
                   {i, 1, Length[pos]-1}];
                linelist
        ]
```

A typical procedure.

```
RandomPoly[x_, n_] := Sum[ Random[Integer, {-10, 10}] x^i, {i, 0, n} ]
```

A typical function.

```
log[a_ b_] := log[a] + log[b]
```

A typical transformation rule.

■ 3.2 Local Variables

Local variables are variables that exist only within a procedure definition (or rewrite rule in *Mathematica*). Using local variables to hold intermediate results within the body of a procedure is an important program design principle. We have already seen that the use of *global* variables can lead to interaction between the procedure and the rest of the program. The part of the program in which a declared variable is visible and can be used is called the *scope* of the variable. A *block* is a part of a program that declares local variables whose scope is that part of the program. A language that supports blocks of scope is called a block-structured language. Examples are ALGOL, C, and Pascal. Fortran also allows local variables, but scopes cannot be nested.

The construct `Module[{`*variables*`}, `*body*`]` declares the variables in the list *variables* as local. Their scope is the expression *body*, usually a compound statement of the form $stmt_1$`;` $statm_2$`;`$\ldots$`;` $stmt_n$. `Block[ ]` also introduces local variables. It is used for special purposes. The differences are discussed in Section 2.6.7 of The *Mathematica* Book.

In most block-structured languages formal parameters of a procedure are treated as initialized local variables, often called "call-by-value". When the procedure is called they are initialized with the value of the actual argument passed to the procedure. Some languages also allow "call-by-reference" parameters. Any occurrence of such a parameter stands for the actual argument. Any assignment to it modifies the actual argument. *Mathematica* has no special mechanism for procedure parameters, but uses pattern matching. Superficially, the names given to patterns play a role similar to parameters, but they are not variables. The value of the actual argument is simply *substituted* for every occurrence of the pattern variable in the body of the procedure. They behave therefore in a similar manner as call-by-reference parameters.

■ 3.3 Local Constants

Local constants are less widely known. Consider the following expression:

$$x(y-1)(1-x^2) + (y-1)^2 - y(1-x^2).$$

The subterms $y-1$ and $1-x^2$ occur in several places. This expression can be written in a simpler way by introducing two auxiliary variables $a = y-1$ and $b = 1-x^2$ and then writing

$$xab + a^2 - yb.$$

The variables a and b should, of course, be local to this expression, just like local variables of a function, but we don't really need local *variables*, since we do not try to assign to them. We only look at their initial values. Within our expression, a and b are constants. The construct

`With[{`$const_1$ `=` val_1`, ..., `$const_n$ `=` val_n`}, `*body*`]`

behaves exactly like that. Every occurrence of the declared constants is replaced by their values. As with pattern variables, it is not possible to assign to them. Their scope is *body*. Such a construct is usually available in functional languages. In LISP it is called `let`.

This is the translation of the formula given above into *Mathematica*. An added benefit is that the terms `y - 1` and `1 - x^2` are evaluated only once.

```
In[2]:= With[ {a = y - 1, b = 1 - x^2}, x a b + a^2 - y b ]

                 2                    2         2
Out[2]= x (1 - x ) (-1 + y) + (-1 + y)  - (1 - x ) y
```

`With[ ]` helps to write concise functional code.

```
In[3]:= ShiftedSum[x_, s_, n_] :=
            Module[{i},
                With[{y = x - s}, Sum[ y^i, {i, 0, n} ] ]
            ]
```

`Module[ ]` and `With[ ]` also prevents conflicts of names of symbols (here for `y` and `i`).

```
In[4]:= ShiftedSum[y, i, 4]

                                 2            3            4
Out[4]= 1 - i + y + (-i + y)  + (-i + y)  + (-i + y)
```

4 Functional and Procedural Programming

Mathematica allows you to program in a variety of styles. The most commonly identified programming styles are procedural programming, functional programming, declarative programming, and object-oriented programming. If you have done your programming so far in one programming language exclusively, I could probably tell you which one you are used to by looking at your first *Mathematica* programs.

Rather than just continue in the style you are used to, I would like to show you how to use *Mathematica*'s commands in the best possible way, choosing the style that is best suited to the problem to be solved. Here, we put the emphasis on procedural and functional programming. In Chapter 6 we will look in detail into the unique style called *mathematical programming* that *Mathematica* offers.

Section 1 deals with the basic forms of iteration, available in almost any programming language. The next section then introduces some iteration constructs that are unique. They correspond very closely to the way we think about mathematics and should be used whenever possible.

■ 4.1 Loops

Loops and iterations are fundamental to any programming language. There are two basic kinds of iterations: repeating a statement a fixed number of times and iterating statements as long as a some condition is satisfied. This section introduces the commands `Do[ ]`, `While[ ]`, and `For[ ]` that are available in one way or another in all procedural programming languages. As we will see later, *Mathematica* offers alternatives that are better suited for many applications.

Iteration

The most basic form of a loop is `Do[`*statement*`, ` *iterator*`]`. It allows you to perform a statement over and over again with the iterator variable taking on successive values. The `Do[ ]` statement itself returns no value.

This prints the squares of the first five positive integers. The output is a side effect of the `Print[ ]` statement. No value is returned; there is no `Out[ ]` line.

```
In[5]:= Do[ Print[i^2], {i, 1, 5} ]

1
4
9
16
25
```

Sometimes you do not need the loop variable at all. The following piece of code computes the n^{th} Fibonacci number f_n. Each Fibonacci number is defined as the sum of the previous two. The first and second one are defined to be one. The sequence begins with $1, 1, 2, 3, 5, 8, 13, \ldots$.

```
Fibonacci[n_Integer?Positive] :=
        Module[{fn1=1, fn2=0},
                Do[ {fn1, fn2} = {fn1 + fn2, fn1}, {n-1} ];
                fn1
        ]
```

Fibonacci1.m: iterative computation of Fibonacci numbers.

The local variables fn1 and fn2 are initialized to the first two Fibonacci numbers. In the loop we repeatedly replace fn1 with the sum of the previous two numbers that are stored in fn1 and fn2 at any time. fn2 is then set to the old value of fn1. The parallel assignment of both fn1 and fn2 makes this very easy without using an additional local variable. Before the loop fn1 is initialized to $f_1 = 1$. Therefore, the loop has to be repeated $n - 1$ times to compute f_n.

This gives the Fibonacci number f_{100}. Loops like this are quite fast.

```
In[2]:= Fibonacci[100]

Out[2]= 354224848179261915075
```

Note that the number of iterations to be performed is computed when the iterator is evaluated before the loop is started. Changing the value of the upper bound in the iterator from inside the loop will therefore not change the number of iterations to be performed. The start, stop, and increment values can be general expressions. The only condition is that the number of iterations to perform evaluates to a number (other than a complex number). The increment need not divide evenly into the interval from start to stop. The number of iterations in the loop Do[*body*, {*start*, *stop*, *incr*}] is $(stop - start)/incr + 1$, rounded down to the nearest integer.

The number of steps is (4a - 2a)/a + 1 ⟶ 3, an integer.

```
In[3]:= Do[ Print[i], {i, 2a, 4a, a} ]

2 a
3 a
4 a
```

The number of steps, (3.5 - 0.0)/1 + 1 ⟶ 4.5 is rounded to 4.

```
In[5]:= Do[ Print[r], {r, 0.0, 3.5} ]

0.
1.
2.
3.
```

In a nested loop, the iterator for the inner variable j is evaluated for each value of the outer variable i.

```
In[7]:= Do[ Print[{i, j}], {i, 3}, {j, i} ]

{1, 1}
{2, 1}
{2, 2}
{3, 1}
{3, 2}
{3, 3}
```

We have already encountered an example for the use of the Do[ ] in the function SplitLines[ ] in the package ComplexMap.m.

Do[ ] in *Mathematica* is similar to the DO loop in Fortran or BASIC and to the for loop in Pascal.

Conditional Repetition of Statements

Often we want to perform a calculation repeatedly while a certain condition is true and stop as soon it becomes false. In this case, we do not know the number of iterations that are to be performed in advance and cannot use the Do[ ] loop, but use the While[ ] loop instead. Its form is While[*condition*, *body*]. Before each iteration the *condition* is tested. If the test returns True, the body of the loop is evaluated one more time, otherwise the loop terminates without returning a value. If the test does not return True the first time it is tested, the loop body is not executed at all.

For an example we look at the function $\pi(x)$ that, given a number x as argument, finds the number of primes less than x. Prime numbers are positive integers that have no divisors except 1 and themselves. The sequence starts with $2, 3, 5, 7, 11, \ldots$. The function `Prime[`n`]` returns the n^{th} prime number p_n. We use iteration to find an n such that $p_n \leq x < p_{n+1}$. To avoid name conflicts with the built-in function `PrimePi[ ]` we call our version `MyPrimePi[ ]`.

```
Attributes[MyPrimePi] = {Listable}

MyPrimePi[x_] := 0 /; x < 2

MyPrimePi[x_] :=
        Module[{li, n0, n1, m, nx = N[x]},
                li = LogIntegral[nx];
                n0 = Floor[li - LogIntegral[Sqrt[nx]]];
                n1 = Ceiling[li];
                While[ n1 - n0 > 1,
                        m = Floor[(n0 + n1)/2];                 (* midpoint *)
                        If[ Prime[m] <= nx, n0 = m, n1 = m ]
                ];
                n0
        ] /; x >= 2
```

`PrimePi.m`: find the index of a prime number.

The function `MyPrimePi[ ]` first gets an initial guess for n. It is a famous theorem in mathematics that the logarithmic integral gives a rather good guess for the n. In the range in which `Prime[ ]` is implemented this guess is always too large. On the other hand, subtracting the logarithmic integral of $\sqrt{x}$ gives an estimate that is too low. We therefore maintain two variables n_0 and n_1 that bracket the correct value. At the beginning, we know that the n we are looking for lies between n_0 and n_1. At each iteration, we check whether the midpoint between them is too high or too low. If it is too low, we set n_0 to this midpoint otherwise we set n_1 to it. In each iteration, the interval between n_0 and n_1 is cut in half and after a few steps the two will differ by at most 1 and we have found n. Our algorithm does not work for arguments smaller than 2. We give a separate rule for this case. There are no primes smaller than 2 and so the value of `MyPrimePi[ ]` is 0 in this case. (The built-in function `PrimePi[ ]` works similarly to ours but uses a more sophisticated guess to start the iteration.)

The number 1993 is the 301^{st} prime.

```
In[2]:= MyPrimePi[1993]

Out[2]= 301
```

This computes the $100{,}000^{th}$ prime.

```
In[3]:= Prime[100000]

Out[3]= 1299709
```

And so we can test our function.

```
In[4]:= MyPrimePi[%]

Out[4]= 100000
```

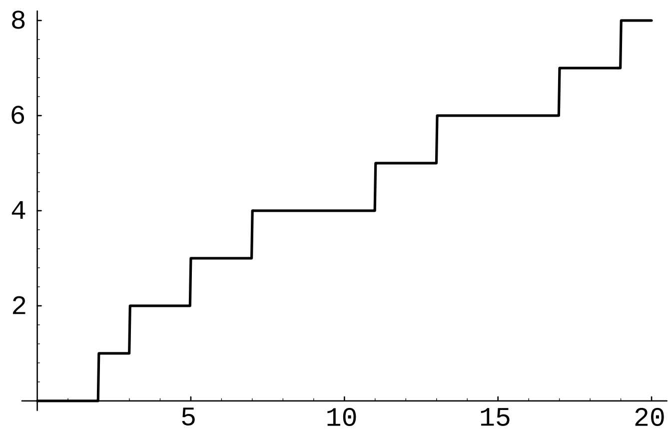
`MyPrimePi[ ]` is listable and it can therefore compute $\pi(x)$ for a whole list of numbers in one call.

```
In[5]:= MyPrimePi[Range[10]]

Out[5]= {0, 1, 2, 2, 3, 3, 4, 4, 4, 4}
```

`MyPrimePi[ ]` is defined for all real numbers. We can therefore even plot it. The function jumps by one at prime values and is constant in between.

```
In[6]:= Plot[MyPrimePi[x], {x, 0, 20}]
```



`While[]` in *Mathematica* is similar to the `while` loop in Pascal and C.

The For Loop

The `For[ ]` loop is patterned after the corresponding loop in the language C. It does not have an equivalent in other languages. C lacks the equivalent of the `Do[ ]` loop and so expert C programmers often end up using `For[ ]` instead of `Do[ ]` to iterate over the values of a variable in *Mathematica*. Instead of `Do[`*body*`, {`*var*`, `*start*`, `*stop*`}]`, they write `For[`*var*`=`*start*`, `*var*`<=`*stop*`, `*var*`++, `*body*`]`, which looks a bit clumsy. The `For[ ]` loop is useful in more complicated instances where the iteration involves several variables and end conditions. The `For[ ]` loop can easily be expressed in terms of a `While[ ]` loop. Instead of

`For[`*start*`, `*test*`, `*step*`, `*body*`]`

we can write

start`; While[`*test*`, `*body*`; `*step*`]`.

If you are not familiar with `For[ ]`, this correspondence can help you understand how it works.

Testing the Exit Condition at the End of a Loop

Besides the `While[ ]` loop that checks a condition at the beginning of each iteration, many programming languages also offer a loop that checks the condition *after* each iteration. Here are several ways in which we could implement a hypothetical `Until[`*body*`, `*test*`]` that repeats *body* until *test* becomes `True`.

`While[True,`*body*`;If[`*test*`,Break[ ]]]`	break from an infinite `While[ ] as soon as` *test* `is true`
`t=False;While[!t,`*body*`;t=`*test*`]`	remember the value of *test* for the next iteration
`For[t=False,!t,t=`*test*`,`*body*`]`	use a `For[ ] loop`

Loops that check their test at the end of an iteration.

If we wanted, we could define our own command `Until[ ]` in terms of one of the possibilities outlined.

```
Until::usage = "Until[body, test] evaluates body until test becomes true."

Begin["`Private`"]

Attributes[Until] = {HoldAll}

Until[body_, test_] := Module[ {t}, For[ t=False, !t, t=test, body ] ]

End[]
Null
```

`Until.m`: a loop that checks its test after each iteration.

The attribute `HoldAll` prevents evaluation of the parameters of `Until[ ]`. They are only evaluated inside `For[ ]`. All loops should behave like this. The definition of `Until[ ]` is so simple that it is not even worth creating a separate package context for it. We do, however, isolate the local symbols `body`, `test`, and `t` in a private subcontext of the current context.

This is a fixed point iteration. It sets x to $1 + 1/x$ until x is equal to $1 + 1/x$. This number is called the *golden ratio*. Note that the precision of the result is greater than the precision of the starting point.

```
In[2]:= x = N[1, 20]; Until[ x=1+1/x, x==1+1/x ]; x

Out[2]= 1.61803398874989484820458683436563811772030917980\
  57628621354486228
```

Loops that test their condition at the end are particularly useful for interactive input from the user. We will repeatedly prompt the user for input until he has given a valid answer. The following code will insist on the user typing a number that satisfies a given predicate. (A predicate is a function that returns `True` or `False`.)

```
GetNumber[prompt_String, predicate_:(True&)] :=
        Module[{answer},
                While[ True,
                        answer = Input[prompt];
                        If[ NumberQ[answer] && predicate[answer], Break[] ];  (* good   *)
                        If[ answer === EndOfFile, Break[] ];                  (* escape *)
                        Print["Please enter a number that satisfies ", predicate]
                ];
                answer
        ]
```

`GetNumber.m`: repeatedly prompting the user for input.

It is a good idea to allow some escape mechanism from the loop that prompts for input. In this case, generating an end-of-file character will exit the loop. If the answer was rejected, a friendly message is printed. The default value for the predicate is the pure function `True&` that returns `True` independent of its argument, and so it will accept any number.

An input that does not satisfy the predicate will issue a message and prompt again.

```
In[1]:= input = GetNumber["Enter a prime: ", PrimeQ]

Enter a prime: 9
Please enter a number that satisfies PrimeQ
Enter a prime: 11

Out[1]= 11
```

Mathematica can read any expression. You can perform arbitrary calculations in the input you type. If it does evaluate to a number, it will be accepted.

```
In[2]:= GetNumber["Enter any number: "]

Enter any number: junk
Please enter a number that satisfies True &
Enter any number: (-1 + 5 I)^2

Out[2]= -24 - 10 I
```

`Until[ ]` is similar to `repeat ... until` in Pascal and to `do ... while` in C (with the truth value of the test reversed!).

4.2 Structured Iteration

In many programming languages the loops presented in Section 4.1 are all that is available. The structured iteration commands of this section are perhaps new to you. If you have programmed in LISP or APL, then you will, however, recognize many familiar and useful commands. The flow-control statements of traditional languages were not designed with the applications in mind that are now possible in *Mathematica*, but rather they were selected for ease of implementation and the need of applications in computer science itself.

Since *Mathematica* does offer the traditional looping constructs, as we have just seen, it is rather tempting to simply continue using these familiar means of flow-control instead of using a more natural, problem-oriented approach. In this section, I would like to show you the transformation from the old approach to *Mathematica*'s way of functional programming.

Sums and Products

Given the problem of adding the square roots of the first 500 integers, the solution in most programming languages is to use an auxiliary variable that is incremented by the square root of a loop index iterating from 1 to 500.

```
sum = 0.0;
Do[ sum = sum + N[Sqrt[i]], {i, 1, 500} ];
sum
```

The procedural way of adding numbers.

In *Mathematica*, this loop reduces to a single statement that directly corresponds to the mathematical formula $\sum_{i=1}^{500} \sqrt{i}$.

```
Sum[ N[Sqrt[i]], {i, 1, 500} ]
```

The mathematical way of adding numbers.

Mathematica does not force you to think about *how* to implement a summation, but lets you focus on the concept itself instead. `Product[ ]` works in the same way, multiplying its terms together instead of adding them up.

This computes the product $\prod_{i=0}^{5}(x-i)$ and expands it.

```
In[3]:= Product[ x-i, {i, 0, 5} ] // Expand

Out[3]= 4.25232921501135639390462188851108328620660858097\
  76108893715487471
```

The iterator used for sums and products is the same as that used for loops (see Subsection 4.1.1). This allows a rather sophisticated way of computing the same product.

```
In[4]:= Product[ e, {e, x, x-5, -1} ] // Expand

Out[4]= 4.25232921501135639390462188851108328620660858097\
  76108893715487471
```

Tables

`Sum[]` and `Product[]` are two examples of a class of commands that evaluate an expression several times varying one or several index variables and then collecting the results in a specific way, either adding them up or multiplying them together. The `Table[]` command is the simplest of them. It just collects its results in a list.

This generates a list of the first 10 powers of x.

```
In[1]:= Table[ x^i, {i, 0, 9} ]

                2    3    4    5    6    7    8    9
Out[1]= {1, x, x , x , x , x , x , x , x , x }
```

This generates a list of the first six of the polynomials used in the previous example. Note how the inner index (in the product) depends on the outer index (of the table).

```
In[2]:= Table[ Product[x-j, {j, 0, i}], {i, 0, 5} ]

Out[2]= {x, (-1 + x) x, (-2 + x) (-1 + x) x,
  (-3 + x) (-2 + x) (-1 + x) x,
  (-4 + x) (-3 + x) (-2 + x) (-1 + x) x,
  (-5 + x) (-4 + x) (-3 + x) (-2 + x) (-1 + x) x}
```

`TableForm[]` prints the elements of a list on separate lines as a table.

```
In[3]:= TableForm[%]

Out[3]//TableForm=
 x
 (-1 + x) x
 (-2 + x) (-1 + x) x
 (-3 + x) (-2 + x) (-1 + x) x
 (-4 + x) (-3 + x) (-2 + x) (-1 + x) x
 (-5 + x) (-4 + x) (-3 + x) (-2 + x) (-1 + x) x
```

`Table[]`, like all of these structured iterators, can have more than one iterator specification, giving you multidimensional tables. Two iterators generate matrices. The first iterator is the outermost one. `Table[`*expr*`,` *iterator*$_1$`,` *iterator*$_2$`]` is therefore equivalent to `Table[Table[`*expr*`,` *iterator*$_2$`],` *iterator*$_1$`]`.

The matrix of the first n powers of n different variables is called Vandermonde's matrix.

```
In[4]:= Table[ x[i]^j, {i, 1, 5}, {j, 0, 4} ] // MatrixForm

Out[4]//MatrixForm=
                   2      3      4
    1   x[1]   x[1]   x[1]   x[1]

                   2      3      4
    1   x[2]   x[2]   x[2]   x[2]

                   2      3      4
    1   x[3]   x[3]   x[3]   x[3]

                   2      3      4
    1   x[4]   x[4]   x[4]   x[4]

                   2      3      4
    1   x[5]   x[5]   x[5]   x[5]
```

Its determinant in expanded form has $n!$ terms. Here n is 5, and we get 120 terms—too much to print out in full.

```
In[5]:= Det[%] // Short

Out[5]//Short=
     4     3     2                                2     3     4
 x[1]  x[2]  x[3]  x[4] + <<118>> + x[2] x[3]  x[4]  x[5]
```

But it factors in this nice way.

```
In[6]:= Factor[%]

Out[6]= (-x[1] + x[2]) (-x[1] + x[3]) (-x[2] + x[3])
  (-x[1] + x[4]) (-x[2] + x[4]) (-x[3] + x[4])
  (-x[1] + x[5]) (-x[2] + x[5]) (-x[3] + x[5])
  (-x[4] + x[5])
```

Arrays

To understand the difference between `Array[]` and the other structured iterators we have discussed so far, let's have a closer look at how these iterators work. Assume the following iterator: `Table[`*expr*`, {i, `*start*`, `*final*`}]`, where the iterator variable is `i`. First the variable is set to the first value, *start*. This is done in much the same way as if you had typed `i=`*start*. Then *expr* is evaluated, using the current value for `i`, wherever `i` occurs in *expr*. For the next iteration, the next value *start*`+1` is assigned to `i` and *expr* is again evaluated.

Another way to look at this is to say that the expression *expr* describes a function of the iterator variable `i`. For each value of `i`, we get a value *f*`[i]` for some function *f*. The iterator `Array[]` takes this point of view. In `Array[`*f*`, `*n*`]`, *f* is the *name* of a function that is applied to each value of the iterator in turn. We do not need a named iterator variable as the name of the parameter of a function does not matter at all. *Mathematica* simply generates the expressions *f*`[1]`, *f*`[2]`, ..., *f*`[`*n*`]` and evaluates them.

You could express this action using `Table[ ]`: The expression `Array[`*f*`, `*n*`]` is equivalent to `Table[`*f*`[i], {i, `*n*`}]` (assuming that there is no conflict of variable names, *i.e.*, `i` does not appear as a free variable in the definition of *f*).

This generates a list of the first 10 prime numbers.

```
In[1]:= Array[ Prime, 10 ]

Out[1]= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

This is the equivalent `Table[]` command.

```
In[2]:= Table[ Prime[i], {i, 10} ]

Out[2]= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

You can also express a `Table[]` in terms of an `Array[]`. If the expression to be tabulated is not of the form *f*`[`*var*`]`, where *var* is the iterator variable, you need to express it as a pure function. For example, `Table[i^2, {i, 10}]` can be expressed as `Array[#^2&, 10]`.

`Array[]` is not as flexible as `Table[]`. The increment of the iterator is always one. The iterator starts at 1 unless a different starting point is given as a third argument, as in `Array[f, 4, 0]` $\longrightarrow$ `{f[0], f[1], f[2], f[3]}`. A multidimensional array has the form `Array[`*f*`, {`n_1`, `n_2`, ..., `n_k`}]` and corresponds to the table

$$\texttt{Table[}\ f\texttt{[}i_1,\ i_2,\ldots,\ i_n\texttt{]},\ \texttt{\{}i_1,\ n_1\texttt{\}},\ \texttt{\{}i_2,\ n_2\texttt{\}},\ldots,\ \texttt{\{}i_k,\ n_k\texttt{\}}\ \texttt{]}.$$

This array gives again Vandermonde's matrix from Subsection 4.2.2. We have two iterators and so we need a pure function of two variables.

```
In[4]:= Array[ x[#1]^(#2-1)&, {5, 5} ] // MatrixForm
```

Out[4]//MatrixForm=

$$\begin{matrix} 1 & x[1] & x[1]^2 & x[1]^3 & x[1]^4 \\ 1 & x[2] & x[2]^2 & x[2]^3 & x[2]^4 \\ 1 & x[3] & x[3]^2 & x[3]^3 & x[3]^4 \\ 1 & x[4] & x[4]^2 & x[4]^3 & x[4]^4 \\ 1 & x[5] & x[5]^2 & x[5]^3 & x[5]^4 \end{matrix}$$