2. Introductory Examples

This chapter introduces the reader to the Joule language and its programming style and illustrates many of Joule's fundamental mechanisms in the course of explaining how some simple examples work. (Joule's underlying computational model is presented in Chapter 3.) The principles illustrated here apply at all levels of granularity—the message-plumbing techniques used here to interact with very simple servers are the same as those used with complex and versatile servers—so these examples can lead to understanding of how to construct large systems in Joule.

2.1. Forwarding and Expression Syntax

Joule objects, called *servers*, interact with each other by sending messages to *ports.* Ports can be extended transparently by *channels*. A channel is a unidirectional route originating at an *acceptor* port and terminating at a *distributor* port, each of which may be held by other servers. A server that holds the acceptor of the channel can send messages through it which can be received by any server holding the distributor.

The distributor can accept a special protocol of messages that instruct it where to forward the messages originating at the acceptor. One can think of the channel as a funnel pouring into a hose. One can pour messages down the funnel, and one can direct the hose to other funnels.

Messages are sent to a port by • (send) statements of the form • *port message*. Within some scope, the ports of a channel are named by identifiers. Typically, if `A` is the acceptor of a channel, `A>` will be its corresponding distributor. The ">" suffix is a convention used to distinguish the name of a distributor.

Messages sent to the distributor cause it to change its behavior in some way. For example, the statement

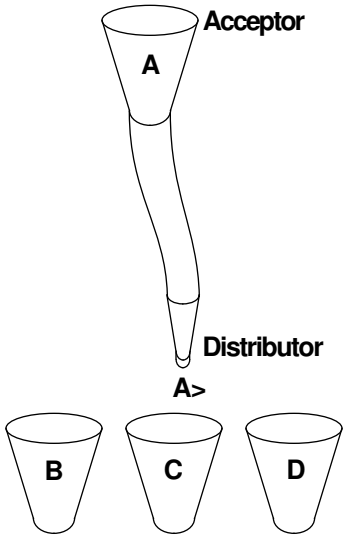
```
• A> → C
```

directs all arriving messages to the port `C`. This statement sends the forward message "→ `C`" to the distributor `A>`, instructing it to forward all messages received at `A>` to `C`.

Messages sent *through* the channel (via its acceptor) are forwarded to servers to which the corresponding distributor has been forwarded.

The same server can hold both ports of a channel—for example, in anticipation of passing one of them off to another server.

Fig. 2.1 • `A> → C`



Anything can be sent as a message, but most messages will be *Tuples* (ordered sets of ports). Tuples are the most common types of messages sent in Joule. The first element of a tuple is its name, known as the *operation*; the other elements are its *arguments*. The statement

```
• A oper: arg1 arg2
```

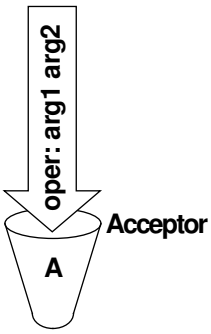


Fig. 2.2 • `A oper: arg1 arg2`

sends the tuple `oper: arg1 arg2` to the port `A` (and, if `A` is an acceptor, through the channel to be delivered by the corresponding distributor). The colon (":") as a suffix distinguishes an operation.

Operators are a special class of operations that do not require the ":" suffix. The forward statement `A> → C` is actually the sending of the forward operation "→" to `A>` (presumably a distributor) with the port `C` as its argument.

In Joule, everything is a server. Numbers are also servers that respond to a set of messages. The expression

```
• 3 + 4 x>
```

Servers make use of results by sending them messages. A print server might send a number the operation meaning "give me an ASCII representation of yourself".

sends the operation named "+" to the server "3" with two arguments: the port to "4", and the distributor `x>` on which to reveal the result. "3" acts on the addition operation by forwarding the distributor `x>` to the server "7". Results in Joule are "revealed" on a distributor rather than "returned"—the calling server retains the corresponding acceptor, which now sends to the server that is the result.

Joule allows an expression-like syntax for operations like "+" that take as their last argument the distributor of a result channel. The correct way to think of Joule's expression-like syntax is to imagine an implicit intermediate result channel `t1`. Then

```
• sum> → 3 + 4
```

The use of "reveal" rather than "return" also reinforces the awareness of security in Joule. A Joule server need reveal only what it chooses to reveal about itself. This is discussed more thoroughly in Chapter 8, "Security."

Also, the use of "return" non-orthogonally mixes data issues with control issues. "Reveal" emphasizes that it deals only with data issues.

performs the same operations as

```
• 3 + 4 t1>
• sum> → t1
```

In practice, this looks as if "3 + 4" becomes a port to which messages can be sent. A tuple-sending statement like "3 + 4" can then be used as an argument to operations, including the forward operation:

```
• sum> → 3 + 4
• Fund withdraw: (3 + 4)
```

Precedence, in Joule, reads from right to left. In `• sum> → 3 + 4`, the tuple "+ 4", sent to "3", reveals as its result an acceptor to "7". The forward operation, with the result "7" as its argument, is sent to the distributor `sum>`, causing `sum>` to also deliver to "7". The forward operator "→" routes to the server "7" all messages arriving at the distributor `sum>`.

2.2. Dispatcher

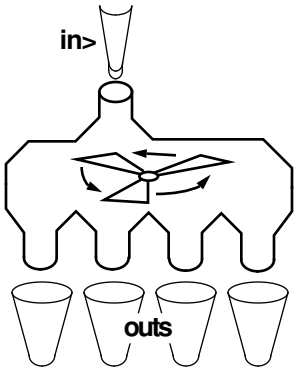


Fig. 2.3 Dispatcher

The Dispatcher server implements a simple statistical load-balancing algorithm for a number of identical servers on a network. Dispatcher receives incoming messages and forwards each one to one of the servers, chosen at random.

```
Server Dispatcher :: in> outs
    • in> → msgs
    ForAll msgs ⇒ message
        Define size
            • outs count: size>
        endDefine
        Define index
            • Random below: size index>
        endDefine
        Define out
            • outs get: index out>
        endDefine
        • out message
    endForAll
endServer
```

Dispatcher takes as arguments a distributor in> (on which the messages will arrive) and an array outs of ports to a set of servers that all provide identical services (outs is actually a port to the array, not the array server itself. For brevity, we will begin referring to acceptors interchangeably with the servers that they send to, except in cases where this could cause confusion).

Servers sending messages to outs don't know whether outs sends directly to the array, or to an array chosen randomly by another Dispatcher, and in most cases don't need to know.

Joule structures called *forms* begin with a keyword (in bold) which determines the syntactic type of the form. The **Server** form binds an identifier (in this case, Dispatcher) to a new server that executes the nested code block in response to messages from other servers. Dispatcher is a *procedural server*; it has a single method which is invoked by passing the "::" operation to the server with the appropriate number of arguments. The double colon is the simplest operation name possible in Joule; to a procedural server, it means "do what you do"—it tells the procedural server to perform its characteristic behavior.

Procedural servers are a special case of *methodical servers*. Methodical servers can accept a variety of operations (not just "**::**") and act on them in different ways. Random is a methodical server called by Dispatcher.

The **ForAll** form causes the nested block of code to be executed once for every message sent to the port defined by the **ForAll** as its first argument. Separate invocations are completely independent of one another and execute concurrently. The • in> → msgs forwards all messages received on in> to the **ForAll**'s input. The **ForAll** block is invoked for each message received, with message bound to that message.

msgs is a good example of a non-methodical server. As in this example, non-methodical servers typically provide message plumbing which is expected to terminate in methodical servers.

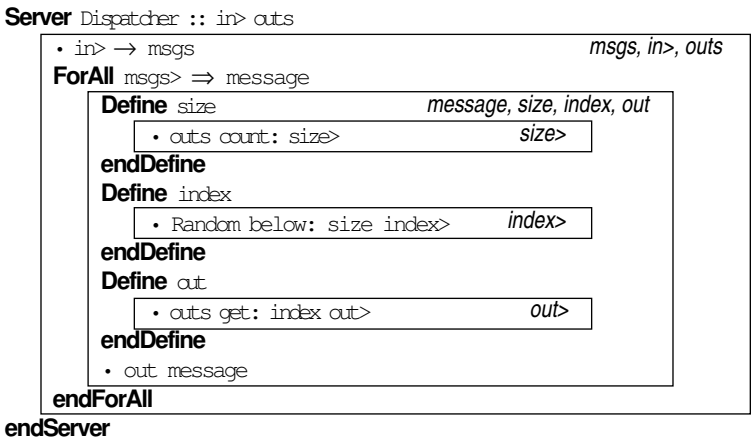
The inner scope of a Joule form consists of all lines of code between the first and last lines of the form (**Form** and **endForm**). Names defined in that block of code are visible anywhere inside that block (including the scopes of blocks nested within it), but not visible or accessible outside that block.

Lexical scoping of identifiers is discussed in detail in Section 5.3.0.

The scope of the **ForAll** form in Dispatcher includes all lines of code from **Define** size to • out message inclusive. The distributor in> is defined as a parameter by the **Server** statement and is available anywhere within the **Server** statement's scope (including inside the scopes of **ForAll** and the **Define** blocks).

The **Define** form creates a channel and binds its ports to identifiers, with the distributor bound only inside the scope of the define and the acceptor bound both inside and outside that scope. Again, distributors are distinguished by the ">" suffix. The distributor `index>` is defined inside the scope of **Define**; hence `index>` is invisible to statements at the scoping level of **ForAll**, **Server**, or `• out message`. The corresponding acceptor `index` is visible in the scope of **ForAll** because **Define** creates it in **Define**'s outer scope, but it is not generally visible in the scope of **Server**.

In the diagram below, each box represents the scope of the form surrounding it. At the upper right of each box are listed the identifiers that are visible only within that scope. Each scope also recognizes the identifiers that are visible in all boxes outside, so from the scope of the first **Define**, the visible identifiers are `size>` (visible only from within that **Define**); `message`, `size`, `index`, and `out` (since this **Define** is inside **ForAll**); `in>`, and `outs` (since **ForAll** is inside **Server**).



The nested blocks of code within the three **Define** forms do the work. First, the tuple `count: size>` is sent to the array of outputs `outs`. This is a standard operation for arrays; the number of elements in the array is revealed on the distributor `size>`—as a result, the acceptor `size` now sends to the number of elements in `outs`.

Next, the tuple `below: size index>` is sent to the server `Random`. `Random` is assumed to be an existing methodical server that provides random numbers. This tuple tells `Random` to forward the distributor `index>` to a random integer greater than zero and less than `size`. (Remember that the acceptor `size` is visible in the inner scope of **ForAll**, and hence also in the inner scope of this **Define**.)

Finally, the array `outs` is sent the tuple `get: index out>`. This is another standard operation accepted by arrays, telling `outs` to forward the distributor `out>` to the `index`th element of the array.

More briefly, "The statement `• out message` sends the original message to the randomly-chosen server `out`." The ubiquity of transparent forwarding in Joule makes descriptions of programs extremely verbose (as you see) unless we refer to acceptors as if they were the servers to which they send.

Back in the outer scope of the last **Define**, the acceptor `out` is visible and now relays messages to an acceptor randomly chosen from those in `outs`. The statement `• out message` forwards `message`, the original message received by **ForAll**, to the randomly-chosen server at the other end of `out`.

Joule belongs to the class of programming languages in which statements execute concurrently, not sequentially. If a Joule statement relies on the output of another, the code expresses the dependency and that

statement will wait for the input it needs. Each line of code in Dispatcher executes concurrently.

The statement Random below: size index> establishes communications links for messages to Random, and can execute whether or not size has yet been forwarded to its final value. When the **Define** size block completes, **Random** will use the result size to decide where to forward index>, but the **Define** index block has already done its part once the message plumbing to and from Random is established, and can cease execution.

Similarly, outs get: index out> could also execute, establishing the message plumbing to and from the array outs, without needing to wait for index> to be forwarded. The **Define** out block connects Random to outs, in a sense, and then can go away. The out message statement also executes concurrently, establishing message pathways in the same way.

The entire process "bottoms out" once outs reveals how many elements it has; then Random can generate a value for index, and the remaining forwards can take place, culminating in the final forwarding of message.

Dispatcher can be rewritten more concisely using Joule's expression-like syntax. The intermediate results channels size and out become implicit:

```
Server Dispatcher :: in> outs
    • in> → msgs
    ForAll msgs ⇒ message
        Define index = Random below: (outs count:) endDefine
        • (outs get: index) message
    endForAll
endServer
```

As a matter of programming style, this is a more attractive form of Dispatcher because the form of the code follows its function. Dispatcher does two things: pick a server at random from outs, and send message to it. This formulation has one line of code for each of these actions.

2.3. Continuous compound interest

This is a simple function that computes continuous compound interest using the formula $P + I = Pe^{\alpha t}$.

```
Reveal the interest generated by continually compounding 'principal' by 'rate' for 'time'
        time-units.
Server continuous-interest :: principal rate time total>
    • total> → principal * (e ^ (rate * time))
endServer
```

Code in italics represents comments. Like Dispatcher, the new server continuous-interest accepts the single operation " ::" ("do what you do"), with arguments bound to the principal, the interest rate, the elapsed time, and a distributor for the result channel. The expression-like syn-

Clearly, statements cannot execute concurrently on a single serial processor. In such an environment, the statements of a Joule program execute sequentially but in an order chosen by the compiler rather than in the order of their appearance in the source listing.

It's also important to note that, for every message sent to in, a separate **ForAll** is activated, and all of these activations of **ForAll** run concurrently, not sequentially. **ForAll** is not an iteration mechanism but a generator of multiple concurrent processes. See the next chapter for a more thorough introduction to the Joule computational model.

tax is used for brevity; assuming the intermediate channels have been defined, it could be written equivalently as:

```
• rate * time t1>
• e ^ t1 t2>
• principal * t2 t3>
• total> → t3
```

with the same result revealed on the channel `total`.

The `continuous-interest` server is invoked by statements like:

```
• continuous-interest :: 40000 0.15 5 result>
• result> → (continuous-interest :: 40000 0.15 5)
```

To forward a distributor is to forward all messages ever received on it.

In either case, the distributor `result>` is forwarded to the number `84680.00068`.

2.4. Factorial

The next example is the familiar factorial function.

```
Reveal the factorial of the supplied number
Server Factorial :: number result>
    If number <= 1
        • result> → 1
    else
        • result> → number * (Factorial :: number - 1)
    endIf
endServer
```

`Factorial` takes two arguments: the number whose factorial is to be computed, and the distributor of the channel on which to reveal the result. Either of these statements:

```
• Factorial :: 5 foo>
• foo> → Factorial :: 5
```

means "calculate the factorial of 5 and reveal the result on the distributor `foo>`."

The expression `number <= 1` reveals an acceptor to either the `true` server or the `false` server. The server `number` forwards that distributor to either `true` or `false` when it receives the "<=" operation with 1 as its argument. More briefly, we may say that `number <= 1` reveals either `true` or `false`.

Some extended forms like **If** have an additional layer of nested scoping between the keyword statement and the inner scopes of the nested blocks of code under the keyword and its extension keywords. The next example, `Fund`, discusses this aspect of extended forms more thoroughly.

The **If** and **else** statements and the nested blocks of code under each one are all part of the same **If**-**endIf** form. Such *extended forms* in Joule enable more complex program behavior, like conditional execution of code.

Depending on the value revealed by `number <=1`, `Factorial` forwards `result>` either to 1 or to the acceptor revealed by the expression `number * (factorial :: number - 1)`. This recursive invocation of `Factorial` causes the activation of another instance of the `Factorial` server calculating the factorial of `(number - 1)`.

2.5. Fund

`Fund` is a toy bank account used by Carl Hewitt to demonstrate properties of open systems. `Fund` is an example of a *methodical server*. Methodical servers are servers that respond to a fixed set of requests. `Fund` responds to `deposit:`, `withdraw:`, and `balance:`.

`Fund` is called a toy bank because it doesn't conserve money, nor does it prevent forging of money: the `deposit:` and `withdraw:` requests take a simple number as their argument. `Fund` is more like a rendezvous service that multiple cooperating agents could use to keep track of how much money had been used so far.

```
Server Fund
    var myBalance = 0

    return the current balance
    op balance: balance>
        • balance> → myBalance

    reduce the balance by an amount if that much is available
    op withdraw: amount flag>
        Define newBalance
            If amount > myBalance
                • newBalance> → myBalance
                • flag> → false
            orIf amount < 0
                • newBalance> → myBalance
                • flag> → false
            else
                • newBalance> → myBalance - amount
                • flag> → true
            endIf
        endDefine
        set myBalance newBalance

    increase the balance by an amount
    op deposit: amount flag>
        Define newBalance
            If amount < 0
                • newBalance> → myBalance
                • flag> → false
            else
                • newBalance> → myBalance + amount
                • flag> → true
            endIf
        endDefine
        set myBalance newBalance
endServer
```

Like procedural servers, methodical servers are defined using the **Server** form. There may be multiple **op** extensions to the **Server** form; each **op** statement defines one of the operations to which the server responds and specifies the arguments expected with that operation. The block of code under the **op** statement—the *method* corresponding to that operation—is executed whenever the server receives that operation; in this sense, each **op** statement is like a separate procedural server with a different characteristic operation. This entire program consists of one **Server** form, including its extension keywords and nested blocks of code.

The procedural form
Server `Foo` :: …

is equivalent to
Server `Foo`
op :: ...

This is for the convenience of procedures, but any operation name could be used.

The term *method* comes from Smalltalk; it corresponds to the *member function* in C++.

The **var** extensions define state variables for the server. A **var** is an identifier which can be reassigned (using the **set** statement) to a different value. (It is thus unlike an acceptor—once the corresponding distribu-

The term *instance variable* comes from Smalltalk; *member variable* is the C++ term.

tor has been forwarded, the server holding the acceptor has no control over where it sends.) A **var** is a local instance variable—it is defined only within the inner scope of the server which created it, and the value of the **var** is different in each instance of that server. In the `Fund` server, the **var** `myBalance` is set initially to zero.

There are no global variables in Joule. Servers may interact only through explicit message passing.

Joule **var**s are *not* globally-accessible locations. They are visible only to a single server and completely controlled by that server, so they do not create global synchronization problems. Furthermore, **Server** and **var** are not primitive to Joule, but are built out of more primitive constructs.

The `balance:` operation instructs `Fund` to reveal the value of `myBalance`:

```
return the current balance
op balance: balance>
    • balance> → myBalance
```

The value of `myBalance` is revealed on the distributor handed to `Fund` as the argument of the `balance:` operation.

The `withdraw:` operation reveals a `false` result if `amount` is negative or greater than the available `myBalance`.

```
reduce the balance by an amount if that much is available
op withdraw: amount flag>
    Define newBalance
        If amount > myBalance
            • newBalance> → myBalance
            • flag> → false
        orIf amount < 0
            • newBalance> → myBalance
            • flag> → false
        else
            • newBalance> → myBalance - amount
            • flag> → true
        endIf
    endDefine
    set myBalance newBalance
```

`Fund` is a simplified version of the hierarchical bank account server `Account` presented in Chapter 6. In the hierarchical account, each guard of the **If** signals a different exception, rather than merely setting a success flag to `false`.

The **set** statement, which changes the value of a **var**, executes concurrently with the **If**. **Define** introduces the intermediate acceptor `newBalance` into its outer scope, so **set** can change `myBalance` to `newBalance` even though messages sent to `newBalance` will wait to be processed until the actual value is calculated.

This gives Joule compilers on sequential computers the option of converting the guarded **If** to a nested if—if x, then foo, else if y, then bar.

Each evaluation expression (or *guard*) of the Joule **If** form executes concurrently. However, only one of the guards that succeed gets to execute its nested block of code. If, as may happen on a sequential computer running Joule, one of the guards succeeds before another has begun executing, the system need not even bother to start up the evaluation of the second guard. Joule's **If** is a race—even if the conditions of the guards are not mutually exclusive, only a single guard out of those (if any) which reveal `true` gets its block of code run.

The result port `flag>` is used as a substitute for the normal action in such a situation, which would be to raise an exception. Exception handling is beyond the scope of this example; it is discussed in detail in Section 5.7.

If `amount` is negative or greater than `myBalance`, the result channel `flag>` is set to `false`, meaning that the attempted transaction did not succeed.

The `deposit:` request increases the value of `myBalance` by `amount` (if `amount` is not negative):

```
        increase the balance by an amount
        op deposit: amount flag>
            Define newBalance
                If amount < 0
                    • newBalance> → myBalance
                    • flag> → false
                else
                    • newBalance> → myBalance + amount
                    • flag> → true
                endIf
            endDefine
            set myBalance newBalance
    endServer
```