5. Use a delimiter character to end the input line as follows:

   ```
   read -d delim_char var
   ```

   For example:

   ```
   $ read -d ":" var
   hello:#var is set to hello
   ```

Running a command until it succeeds

When using your shell for everyday tasks, there will be cases where a command might succeed only after some conditions are met, or the operation depends on an external event (such as a file being available to download). In such cases, one might want to run a command repeatedly until it succeeds.

How to do it...

Define a function in the following way:

```
repeat()
{
  while true
  do
    $@ && return
  done
}
```

Or, add this to your shell's `rc` file for ease of use:

```
repeat() { while true; do $@ && return; done }
```

How it works...

We create a function called `repeat` that has an infinite `while` loop, which attempts to run the command passed as a parameter (accessed by `$@`) to the function. It then returns if the command was successful, thereby exiting the loop.

There's more...

We saw a basic way to run commands until they succeed. Let us see what we can do to make things more efficient.

A faster approach

On most modern systems, true is implemented as a binary in `/bin`. This means that each time the aforementioned `while` loop runs, the shell has to spawn a process. To avoid this, we can use the `:` shell built-in, which always returns an exit code 0:

```
repeat() { while :; do $@ && return; done }
```

Though not as readable, this is certainly faster than the first approach.

Adding a delay

Let's say you are using `repeat()` to download a file from the Internet which is not available right now, but will be after some time. An example would be:

```
repeat wget -c http://www.example.com/software-0.1.tar.gz
```

In the current form, we will be sending too much traffic to the web server at `www.example.com`, which causes problems to the server (and maybe even to you, if say the server blacklists your IP for spam). To solve this, we can modify the function and add a small delay as follows:

```
repeat() { while :; do $@ && return; sleep 30; done }
```

This will cause the command to run every 30 seconds.

Field separators and iterators

The **internal field separator** (**IFS**) is an important concept in shell scripting. It is very useful while manipulating text data. We will now discuss delimiters that separate different data elements from single data stream. An internal field separator is a delimiter for a special purpose. An internal field separator is an environment variable that stores delimiting characters. It is the default delimiter string used by a running shell environment.

Consider the case where we need to iterate through words in a string or **comma separated values** (**CSV**). In the first case we will use `IFS=" "` and in the second, `IFS=","`. Let us see how to do it.

Getting ready

Consider the case of CSV data:

```
data="name,sex,rollno,location"
To read each of the item in a variable, we can use IFS.
oldIFS=$IFS
IFS=, now,
for item in $data;
do
```