of time and then test the condition again. For example, consider the following code that uses polling to wait until data has been provided in a pipe:

```
class ReadFromPipe extends Thread
{
  private Pipe pipe;

  //...
  public void run()
  {
    int data;
    while(true)
    {
      synchronized(pipe) {
        while((data = pipe.getData()) == 0)
        {
          //No data, so sleep for a while and try again.
          try {
            sleep(200);
          }
          catch(InterruptedException e){}  //Exception is ignored
        }                                  //purposefully.
        //Process data
      }
    }
  }
}
```

An object of the `ReadFromPipe` class runs on a separate thread and performs polling. The `run` method contains an infinite loop that continually queries the `Pipe` class to see if any data is available to be read. If there is no data, the thread sleeps for `200` milliseconds and then queries again.

This code works, but it is inefficient because the polling loop takes up processor cycles. When there is no data in the pipe, this thread still requires processor cycles to query the pipe for data.

A more efficient implementation uses the `wait` method with `notify` or `notifyAll`. Proper use of these methods eliminates the need to waste processor cycles on polling. For example, the previous code rewritten to avoid polling and use `wait` and `notifyAll` looks like this:

```
class ReadFromPipe extends Thread
{
  private Pipe pipe;
  //...
```