这个方法将从某个文件中读入一个字节，而 System.in（它是 InputStream 的一个子类的预定义对象）却是从“标准输入”中读入信息，即从控制台或重定向的文件中读入信息。

InputStream 类还有若干个非抽象的方法，它们可以读入一个字节数组，或者跳过大量的字节。从 Java 9 开始，有了一个非常有用的可以读取流中所有字节的方法：

```
byte[] bytes = in.readAllBytes();
```

还有多个用来读取给定数量字节的方法，可以参见 API 说明。

这些方法都要调用抽象的 read 方法，因此，各个子类都只需覆盖这一个方法。

与此类似，OutputStream 类定义了下面的抽象方法：

```
abstract void write(int b)
```

它可以向某个输出位置写出一个字节。

如果我们有一个字节数组，那么就可以一次性地写出它们：

```
byte[] values = . . .;
out.write(values);
```

transferTo 方法可以将所有字节从一个输入流传递到一个输出流：

```
in.transferTo(out);
```

read 和 write 方法在执行时都将阻塞，直至字节确实被读入或写出。这就意味着如果流不能被立即访问（通常是因为网络连接忙），那么当前的线程将被阻塞。这使得在这两个方法等待指定的流变为可用的这段时间里，其他的线程就有机会去执行有用的工作。

available 方法使我们可以去检查当前可读入的字节数量，这意味着像下面这样的代码片段不可能被阻塞：

```
int bytesAvailable = in.available();
if (bytesAvailable > 0)
{
   var data = new byte[bytesAvailable];
   in.read(data);
}
```

当你完成对输入 / 输出流的读写时，应该通过调用 close 方法来关闭它，这个调用会释放掉十分有限的操作系统资源。如果一个应用程序打开了过多的输入 / 输出流而没有关闭，那么系统资源将被耗尽。关闭一个输出流的同时还会冲刷用于该输出流的缓冲区：所有被临时置于缓冲区中，以便用更大的包的形式传递的字节在关闭输出流时都将被送出。特别是，如果不关闭文件，那么写出字节的最后一个包可能永远也得不到传递。当然，我们还可以用 flush 方法来人为地冲刷这些输出。

即使某个输入 / 输出流类提供了使用原生的 read 和 write 功能的某些具体方法，应用系统的程序员还是很少使用它们，因为大家感兴趣的数据可能包含数字、字符串和对象，而不是原生字节。

我们可以使用众多的构建于基本的 InputStream 和 OutputStream 类之上的某个输入 / 输出类，而不只是直接使用字节。