值的 Stream，这固然会造成困扰。修正方法是利用转换强制调用正确的覆盖：

```
"Hello world!".chars().forEach(x -> System.out.print((char) x));
```

但是，**最好避免利用 Stream 来处理 char 值。**

刚开始使用 Stream 时，可能会冲动到恨不得将所有的循环都转换成 Stream，但是切记，千万别冲动。这可能会破坏代码的可读性和易维护性。一般来说，即使是相当复杂的任务，最好也结合 Stream 和迭代来一起完成，如上面的 `Anagrams` 程序范例所示。因此，**重构现有代码来使用 Stream，并且只在必要的时候才在新代码中使用。**

如本条目中的范例程序所示，Stream pipeline 利用函数对象（一般是 Lambda 或者方法引用）来描述重复的计算，而迭代版代码则利用代码块来描述重复的计算。下列工作只能通过代码块，而不能通过函数对象来完成：

- 从代码块中，可以读取或者修改范围内的任意局部变量；从 Lambda 则只能读取 final 或者有效的 final 变量 [JLS 4.12.4]，并且不能修改任何 local 变量。
- 从代码块中，可以从外围方法中 `return`、`break` 或 `continue` 外围循环，或者抛出该方法声明要抛出的任何受检异常；从 Lambda 中则完全无法完成这些事情。

如果某个计算最好要利用上述这些方法来描述，它可能并不太适合 Stream。反之，Stream 可以使得完成这些工作变得易如反掌：

- 统一转换元素的序列
- 过滤元素的序列
- 利用单个操作（如添加、连接或者计算其最小值）合并元素的顺序
- 将元素的序列存放到一个集合中，比如根据某些公共属性进行分组
- 搜索满足某些条件的元素的序列

如果某个计算最好是利用这些方法来完成，它就非常适合使用 Stream。

利用 Stream 很难完成的一件事情就是，同时从一个 pipeline 的多个阶段去访问相应的元素：一旦将一个值映射到某个其他值，原来的值就丢失了。一种解决办法是将每个值都映射到包含原始值和新值的一个对象对（pair object），不过这并非万全之策，当 pipeline 的多个阶段都需要这些对象对时尤其如此。这样得到的代码将是混乱、繁杂的，违背了 Stream 的初衷。最好的解决办法是，当需要访问较早阶段的值时，将映射颠倒过来。

例如，编写一个打印出前 20 个梅森素数（Mersenne primes）的程序。解释一下，梅森素数是一个形式为 2^p-1 的数字。如果 p 是一个素数，相应的梅森数字也是素数；那么它就是一个梅森素数。作为 pipeline 的第一个 Stream，我们想要的是所有素数。下面的方法将返回（无限）Stream。假设使用的是静态导入，便于访问 `BigInteger` 的静态成员：