Java中从函数式编程中引入的两个核心思想：将方法和Lambda（匿名函数）作为一等值,以及在没有可变共享状态时,函数或方法可以有效、安全地并行执行。前面说到的新的Stream API把这两种思想都用到了。

常见的函数式语言，如SML、OCaml、Haskell，还提供了进一步的结构来帮助程序员。其中之一就是通过更多的描述性数据类型来避免null。

在Java 8里有一个Optional<T>类，如果你能一直地使用它的话，就可以帮助你避免出现NullPointer异常。它是一个容器对象，可以包含，也可以不包含一个值。Optional<T>中有方法来明确处理值不存在的情况，这样就可以避免NullPointer异常了。换句话说，它使用类型系统，允许你表明我们知道一个变量可能会没有值。

第二个想法是（结构）模式匹配。这在数学中也有使用，例如：

f(0) = 1

f(n) = n\*f(n-1) otherwise

在Java中，你可以在这里写一个if-then-else语句或一个switch语句。其他语言表明，对于更复杂的数据类型，模式匹配可以比if-then-else更简明地表达编程思想。对于这种数据类型，你也可以使用多态和方法重载来替代if-then-else，但对于哪种方式更适合，就语言设计而言仍有一些争论。我们认为两者都是有用的工具，你都应该掌握。不幸的是，Java 8对模式匹配的支持并不完全，虽然我们会在第14章中介绍如何对其进行表达。与此同时，我们用一个以 Scala语言(另一个使用JVM的类Java语言，启发了Java在一些方面的发展;请参阅第15章)表达的例子加以描述。比方说，你要写一个程序对描述算术表达式的树做基本的简化。给定一个数据类型Expr代表这样的表达式，在Scala里你可以写以下代码，把Expr分解给它的各个部分，然后返回另一个Expr:

def simplifyExpression(expr: Expr): Expr = expr match {

case BinOp("+", e, Number(0)) => e

case BinOp("\*", e, Number(1)) => e

case BinOp("/", e, Number(1)) => e

case \_ => expr

}

这里，Scala的语法expr match就对应于Java中的switch（expr）。

为什么Java中的switch语句应该限于原始类型值和Stings呢？函数式语言倾向于允许switch用在更多的数据类型上，包括允许模式匹配（在Scala代码中是通过match操作实现）。在面向对象设计中，常用的访客模式可以用来遍历一组类（如汽车的不同组件：车轮、发动机、底盘等），并对每个访问的对象执行操作。模式匹配的一个优点是编译器可以报告常见错误，如：“Brakes类属于用来表示Car类的组件的一族类。你忘记了要显式处理它。”