L9 Defensive Programming, Immutability

Make Bugs Impossible

Java also gives us *immutable references*: variables declared with the keyword `final` ,which can be assigned once but never reassigned

Localize bug

If we can't prevent bugs, we can try to localize them to a small part of the program, so that we don't have to look too hard to find the cause of a bug. When localized to a single method or small module, bugs may be found simply by studying the program text.

We already talked about **fail fast**: the earlier a problem is observed (the closer to its cause), the easier it is to fix.

Checking preconditions is an example of **defensive programming**. Real programs are rarely bug-free. Defensive programming offers a way to mitigate the effects of bugs even if you don't know where they are.

Assertions

It is common practice to define a procedure for these kinds of defensive checks, usually called `assert`:

```
assert (x >= 0);
```

Incremental Development

A great way to localize bugs to a tiny part of the program is incremental development. Build only a bit of your program at a time, and test that bit thoroughly before you move on. That way, when you discover a bug, it's more likely to be in the part that you just wrote, rather than anywhere in a huge pile of code.

将错误定位到程序的一小部分的一个好方法是增量开发。一次只构建一小部分程序，并在继续之前彻底测试该部分。这样，当你发现一个 bug 时，它更有可能出现在你刚刚编写的部分，而不是一大堆代码中的任何地方。

- Unit testing: when you test a module in isolation, you can be confident that any bug you find is in that unit – or maybe in the test cases themselves.
- Regression testing: when you're adding a new feature to a big system, run the regression test suite as often as possible. If a test fails, the bug is probably in the code you just changed.

Modularity & Encapsulation

Modularity. Modularity means dividing up a system into components, or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system. The opposite of a modular system is a monolithic system – big and with all of its pieces tangled up and dependent on each other.

模块化。模块化意味着将系统划分为组件或模块，每个组件或模块都可以与系统的其余部分分开设计、实现、测试、推理和重用。模块化系统的反面是一个整体系统——很大，所有的部分都纠缠在一起，相互依赖

A program consisting of a single, very long main() function is monolithic – harder to understand, and harder to isolate bugs in. By contrast, a program broken up into small functions and classes is more modular.

由单个很长的 main() 函数组成的程序是整体的——更难理解，也更难隔离其中的错误。相比之下，分解成小函数和类的程序更加模块化。

One kind of encapsulation is access control, using `public` and `private` to control the visibility and accessibility of your variables and methods. A public variable or method can be accessed by any code (assuming the class containing that variable or method is also public). A private variable or method can only be accessed by code in the same class. Keeping things private as much as possible, especially for variables, provides encapsulation, since it limits the code that could inadvertently cause bugs

一种封装是访问控制，使用公共和私有来控制变量和方法的可见性和可访问性。任何代码都可以访问公共变量或方法（假设包含该变量或方法的类也是公共的）。私有变量或方法只能由同一类中的代码访问。尽可能保持私有，特别是对于变量，提供了封装，因为它限制了可能无意中导致错误的代码

Another kind of encapsulation comes from **variable scope**. The *scope* of a variable is the portion of the program text over which that variable is defined, in the sense that expressions and statements can refer to the variable. A method parameter's scope is the body of the method. A local variable's scope extends from its declaration to the next closing curly brace. Keeping variable scopes as small as possible makes it much easier to reason about where a bug might be in the program. For example, suppose you have a loop like this:

另一种封装来自变量作用域。变量的范围是定义该变量的程序文本部分，从表达式和语句可以引用该变量的意义上说。方法参数的范围是方法的主体。局部变量的范围从它的声明扩展到下一个右花括号。保持变量范围尽可能小，可以更容易地推断程序中可能存在错误的位置。例如，假设您有一个这样的循环

Minimizing the scope of variables is a powerful practice for bug localization. Here are a few rules that are good for Java:

- **Always declare a loop variable in the for-loop initializer.** So rather than declaring it before the loop:

Mutability

就是可变变量

Risks of mutation

Mutable types seem much more powerful than immutable types. If you were shopping in the Datatype Supermarket, and you had to choose between a boring immutable `String` and a super-powerful-do-anything mutable `StringBuilder`, why on earth would you choose the immutable one? `StringBuilder` should be able to do everything that `String` can do, plus `set()` and `append()` and everything else.

The answer is that **immutable types are safer from bugs, easier to understand, and more ready for change**. Mutability makes it harder to understand what your program is doing, and much harder to enforce contracts. Here are two examples that illustrate why.

答案是不可变类型更安全，更容易理解，更容易改变。可变性使得理解你的程序在做什么变得更加困难，并且更加难以执行契约。这里有两个例子可以说明原因。

Risky example #1: passing mutable values

Risky example #2: returning mutable values