本书主旨就是帮助你更有效的使用java语言和他自己的基础库，这些基础库包括java.lang, java.util,和java.io,以及这些库的字库比如 java.util.concurrent and java.util.function。其他的一些库我们也会有所提及。

本书由90个条款组成，每个条款都阐述了一种规则。这些规则都应该在实践中得到，最优秀最有经验的程序员都认为这些规则是非常有用的。这些规则松散的被分成了11章，每章都涵盖软件设计的一个宽的范围。本书不推荐从头按顺阅读，因为每项或多或少都可以独立存在的。这些规则被高度穿插引用因此你可以很方便来规划自己正本书的阅读的进程。

自从本书发行的上一个版本之后，java平台增加了许多新的特性。本书的大部分规则将一些方式去运用这些特性。下面的表展示了这些关键特性的主要内容都分布在哪些规则中，以及支持这些特性的java平台。

大部分规则通过程序示例进行阐明。本书的一个关键特性就是代码示例包含有许多设计模式和惯用条款。在合适的地方，他们进行交叉引用，可以在标准的Gamma95中找到标准的引用。

Many items contain one or more program examples illustrating some practice to be avoided. Such examples, sometimes known as antipatterns, are clearly labeled with a comment such as // Never do this!. In each case, the item explains why the example is bad and suggests an alternative approach.

许多规则包含了一个或者多个程序代码，这些程序代码是应该避免出现的。例如有些例子，有时候作为反模式被熟知，这些例子的被清楚的以一个注释锁标明：永远不用这么做。每个例子，这些规则项解释为什么这个例子是错误的，并且给出一个代替的方式。

This book is not for beginners: it assumes that you are already comfortable

with Java. If you are not, consider one of the many fine introductory texts, such as Peter Sestoft’s Java Precisely [Sestoft16]. While Effective Java is designed to be accessible to anyone with a working knowledge of the language, it should provide

food for thought even for advanced programmers.

这本书不适合初学者：它会假设读者已经非常熟悉java。如果你不符合要求，参考许多优秀的引导性内容，比如peter的java精粹。而effectjava是提供给任何一个以java语言为工作内容的读者，他可以提供思考上的帮助，甚至是对于一些高级程序员。

Most of the rules in this book derive from a few fundamental principles.Clarity and simplicity are of paramount importance. The user of a component should never be surprised by its behavior. Components should be as small as possible but no smaller. (As used in this book, the term component refers to any reusable software element, from an individual method to a complex framework consisting of multiple packages.) Code should be reused rather than copied. The dependencies between components should be kept to a minimum. Errors should be detected as soon as possible after they are made, ideally at compile time.

本书的大部分规则都是衍生与一少部分基础原则.清晰简单是至关重要的。组件的使用者不应该被他行为感到惊奇，组件应该尽可能小但是不能太小。本书中使用的术语"组件"是指任可以重用的软件对象，从一个独立的方法到一个包括多个包的复杂的框架。代码应该被重用而不是复制。组件之间的依赖应该保持最小。错误在他们产生之后应该尽可能快的被检测出来,理想状态是在编译器就能被发现。

While the rules in this book do not apply 100 percent of the time, they do

characterize best programming practices in the great majority of cases. You

should not slavishly follow these rules, but violate them only occasionally and

with good reason. Learning the art of programming, like most other disciplines,

consists of first learning the rules and then learning when to break them.

本书中这些规则不适合所以的情况，然后，他们在大部分示例中表现为最好的编程实践。你不应该完成顺从这些规则，但是如果违背这些规则应该是很偶然并且需要合适的理由。学习编程的艺术（适用于大部分原则），包括第一次学习使用规则，然后学习打破规则。

For the most part, this book is not about performance. It is about writing

programs that are clear, correct, usable, robust, flexible, and maintainable. If you can do that, it’s usually a relatively simple matter to get the performance you need(Item 67). Some items do discuss performance concerns, and a few of these items provide performance numbers. These numbers, which are introduced with the

phrase “On my machine,” should be regarded as approximate at best.

本书大部分章节，不是关于性能的。本书是提供编写清晰，正确，可用，健壮，可扩展和可维护代码的指南。如果你能做到这些，那么提升你需要的性能通常就是一件很简单的事情了。有些条款讨论了性能的问题，但是很少的条款提供了性能指标。这些指标是使用在“On my machine”机器上，被考虑是接近最好的选择。

For what it’s worth, my machine is an aging homebuilt 3.5GHz quad-core

Intel Core i7-4770K with 16 gigabytes of DDR3-1866 CL9 RAM, running Azul’s

Zulu 9.0.0.15 release of OpenJDK, atop Microsoft Windows 7 Professional SP1

(64-bit).

下面是值得参考，我个人机器是一个自组装的老机器:3.5GH 四核 16G内存，openjdk，win7

When discussing features of the Java programming language and its libraries,

it is sometimes necessary to refer to specific releases. For convenience, this book

uses nicknames in preference to official release names. This table shows the mapping between release names and nicknames:

在讨论java程序设计语言的特性和他函数库的时候，是非常有必要明确一下发型版本。为了方便，本书使用别名而不是官方的版本名称。下面的表展示了发行版本及其对应的别名。

The examples are reasonably complete, but favor readability over completeness.

They freely use classes from packages java.util and java.io. In order to

compile examples, you may have to add one or more import declarations, or other

such boilerplate. The book’s website, http://joshbloch.com/effectivejava,

contains an expanded version of each example, which you can compile and run.

书中的例子是适度完整，可读性超过了完整性，他们自由使用来自util和io包中的类。为了去编译这些例子，需要自己添加一个或者多个import声明，或者其他引用。本书的官网包括了每个例子的扩展版本，这些版本的例子，可以直接编译和执行。

For the most part, this book uses technical terms as they are defined in The

Java Language Specification, Java SE 8 Edition [JLS]. A few terms deserve

special mention. The language supports four kinds of types: interfaces (including

annotations), classes (including enums), arrays, and primitives. The first three are known as reference types. Class instances and arrays are objects; primitive values are not. A class’s members consist of its fields, methods, member classes, and member interfaces. A method’s signature consists of its name and the types of its formal parameters; the signature does not include the method’s return type.

大部情况下，本书使用的技术术语来自语音java8的语言规范上的术语。很少的术语会有特殊的说明。java语音支持四种类型：接口(包括注解),类（包括枚举），数组和原生态类型。前三种是引用类型。类的示例和数组都是对象，原生态数据类型不是。一个类的成员包括：属性字段，方法，成员类，成员接口。一个方法签名签名包括它的名称，形式参数，方法签名不包括方法返回类型。

This book uses a few terms differently from The Java Language Specification.

Unlike The Java Language Specification, this book uses inheritance as a synonym

for subclassing. Instead of using the term inheritance for interfaces, this book simply states that a class implements an interface or that one interface extends

another. To describe the access level that applies when none is specified, this book uses the traditional package-private instead of the technically correct package access [JLS, 6.6.1].

本书使用了一些术语与java规范不太一样。与java规范相比不同的是，本书使用继承作为子类化的同义词。为了替代术语“接口继承”，本书简单的把类实现接口和接口继承接口通称为继承。为描述没有指定访问级别，本书使用了传统的包私有替代专业上正确的包访问。

This book uses a few technical terms that are not defined in The Java Language

Specification. The term exported API, or simply API, refers to the classes,

interfaces, constructors, members, and serialized forms by which a programmer

accesses a class, interface, or package. (The term API, which is short for application programming interface, is used in preference to the otherwise preferable term interface to avoid confusion with the language construct of that name.) A programmer who writes a program that uses an API is referred to as a user of the API. A class whose implementation uses an API is a client of the API.

本书还使用了一些java规范中没有定义的专业术语。对API(或者是指普通的API)术语进行扩展，扩展的术语API指接口，构造函数，成员变量，和程序员可以访问类，接口或者包的可序列化的形式。（API是应用程序接口的缩写，他比其他可取的术语---“接口”更优先使用，因为可以避免与java语言本身结构中的接口术语相混淆）

Classes, interfaces, constructors, members, and serialized forms are collectively known as API elements. An exported API consists of the API elements that are accessible outside of the package that defines the API. These are the API elements that any client can use and the author of the API commits to support. Not coincidentally, they are also the elements for which the Javadoc utility generates documentation in its default mode of operation. Loosely speaking, the exported API of a package consists of the public and protected members and constructors of every public class or interface in the package.

类，接口，构造函数按的序列化形式都的被以API元素的形式所知晓。一个扩展的API包括的API元素，可以访问包以为定义的API。这里的API可以被人任何客户端使用并且API提供者会对它提供支持。无独有偶，这些API就是JAVADOC工具以默认形式产生的文档。笼统说，扩展AIP是包括了包中每个类的公共构造函数，公共和保护成员以及接口。

In Java 9, a module system was added to the platform. If a library makes use of the module system, its exported API is the union of the exported APIs of all the packages exported by the library’s module declaration.

在java9中，一个模块化的系统被加到平台中，如果一个库函数使用模块化系统，他的扩展API是一个并集，这个集合中包括所有的扩展包的所以扩展API，这些扩展包都是通过库的模块化声明进行扩展。

Creating and Destroying Objects

对象的创建和销毁

This chapter concerns creating and destroying objects: when and how to create

them, when and how to avoid creating them, how to ensure they are destroyed in a

timely manner, and how to manage any cleanup actions that must precede their

destruction.

本章关注的是对象的创建和销毁:何时怎样创建对象，什么时间以及怎样避免创建对象，如何确保他们及时进行销毁，最后怎样管理对象的清理工作，这些清理工作必须在他们销毁执行进行。

**Item 1: Consider static factory methods instead of constructors**

第1款:考虑用静态工厂方法替代构造函数

The traditional way for a class to allow a client to obtain an instance is to provide a public constructor. There is another technique that should be a part of every programmer’s toolkit. A class can provide a public static factory method, which is simply a static method that returns an instance of the class. Here’s a simple example from Boolean (the boxed primitive class for boolean). This method translates a boolean primitive value into a Boolean object reference:

public static Boolean valueOf(boolean b) {

return b ? Boolean.TRUE : Boolean.FALSE;

}

Note that a static factory method is not the same as the Factory Method pattern from Design Patterns [Gamma95]. The static factory method described in this item has no direct equivalent in Design Patterns.

传统的获取类实例的方法是允许客户端访问类提供的公共构造函数。还有另外一种技术，应该作为每个程序员的工具包里的一部分：一个类应该提供一个静态工厂方法，这个方法是普通的静态方法，它返回一个类的实例对象。这里有一个来自Boolean（原始数据类型boolean的包装类）的简单示例。该方法把boolean的原始值转变为Boolean类的引用类型：

public static Boolean valueOf(boolean b) {

return b ? Boolean.TRUE : Boolean.FALSE;

}

注意，这静态工厂方法与设计模式中的静态工厂方法不一样。这本条款中的静态工厂方法在设计模式没有直接等效。

A class can provide its clients with static factory methods instead of, or in addition to, public constructors. Providing a static factory method instead of a public constructor has both advantages and disadvantages.

一个类为客户端除了公共构造方法之前，还应该提供静态工厂方法。利用静态构造方法来代替公共构造函数优缺点并存。

One advantage of static factory methods is that, unlike constructors, they have names. If the parameters to a constructor do not, in and of themselves, describe the object being returned, a static factory with a well-chosen name is easier to use and the resulting client code easier to read. For example, the constructor BigInteger(int, int, Random), which returns a BigInteger that is probably prime, would have been better expressed as a static factory method

named BigInteger.probablePrime. (This method was added in Java 4.)

第一个优点，与构造方法不同，静态工厂方法有自己名字。如果通过构造函数的参数本身不能够描述该构造方法将要返回的对象，此时，拥有恰当名字静态工厂方法更加易于使用，并且客户端代码易读性也强。例如，构造函数BigInteger(int, int, Random)可能返回BigInteger对象是一个素数，但如果使用静态工厂方法BigInteger.probablePrime，那么描述的就更清晰。

A class can have only a single constructor with a given signature. Programmers have been known to get around this restriction by providing two constructors whose parameter lists differ only in the order of their parameter types. This is a really bad idea. The user of such an API will never be able to remember which constructor is which and will end up calling the wrong one by mistake. People reading code that uses these constructors will not know what the code does without referring to the class documentation.

一个类的一个函数签名只能定义一个构造函数。大家都知道程序员为了应对这种限制，提供两个构造函数，这两个构造函数的参数列表仅仅可能是参数类型顺序不同。这是一个比较差的办法。这种API使用者永远记不住哪个构造方法对应哪个实例，并且无法避免由于失误导致调用了错误的构造函数。用户在阅读使用这样的构造方法的代码时，如果没有相关的类说明文档，将不会明白代码所表达的含义。

Because they have names, static factory methods don’t share the restriction discussed in the previous paragraph. In cases where a class seems to require multiple constructors with the same signature, replace the constructors with static factory methods and carefully chosen names to highlight their differences.

由于静态工厂方法有自己的名称，静态工厂方法就不会有上一段落所讨论的那些限制。在这样的情况下，一个类似乎需要多个拥有相同签名的构造函数，使用静态工厂方法取代构造函数，需要给静态工厂方法仔细选择能够区别彼此的名称。

A second advantage of static factory methods is that, unlike constructors, they are not required to create a new object each time they’re invoked. This allows immutable classes (Item 17) to use preconstructed instances, or to cache instances as they’re constructed, and dispense them repeatedly to avoid creating unnecessary duplicate objects. The Boolean.valueOf(boolean) method illustrates this technique: it never creates an object. This technique is similar to the Flyweight pattern [Gamma95]. It can greatly improve performance if equivalent objects are requested often, especially if they are expensive to create.

静态工厂方法的第二个优点是，在他每次调用的时候，不需要创建一个新的对象，这一点与构造方法不同。这种方式允许不可变类（条款17）使用预构造的实例或者是之前缓存的构造实例，这样可以反复的避免创建多个不必要的重复对象。Boolean.valueOf(boolean)这个方法也证明了这种技巧：它不会创建新的对象。这个技巧似乎与享元模式类似。当每次需要相同的对象，并且他们的创建成本很高，那么这种方式非常有助于提高性能。

The ability of static factory methods to return the same object from repeated invocations allows classes to maintain strict control over what instances exist at any time. Classes that do this are said to be instance-controlled. There are several reasons to write instance-controlled classes. Instance control allows a class to guarantee that it is a singleton (Item 3) or noninstantiable (Item 4). Also, it allows an immutable value class (Item 17) to make the guarantee that no two equal instances exist: a.equals(b) if and only if a == b. This is the basis of the Flyweight pattern [Gamma95]. Enum types (Item 34) provide this guarantee.

静态工厂方法拥有重复调用静态工厂方法产生同一个对象的能力，这种的能力允许类可以严格控制什么时候出现什么实例。这被称为类的实例控制。编写实例控制类有几个原因：1.可以保证类的单例或者不能实例化。2.保证类实例的不变性，可以保证不会存在两个相等的对象实例：如果a.equals(b)那么有且仅有a==b。这是享员模式的最基本原则。枚举类型提供了这种的保证。

**A third advantage of static factory methods is that, unlike constructors, they can return an object of any subtype of their return type.** This gives yougreat flexibility in choosing the class of the returned object.

与构造函数不同的第三个优点：他们可以返回实例类的子类。这就给了开发人员在选择类的返回值是提供了很大的灵活性。

One application of this flexibility is that an API can return objects without making their classes public. Hiding implementation classes in this fashion leads to a very compact API. This technique lends itself to *interface-based frameworks*(Item 20), where interfaces provide natural return types for static factory methods.

灵活性的一个应用是一个API可以返回非public声明的对象。隐藏类的实现的方式，可以使API设计变的紧凑。这种技术导致自己成为面向接口编程的框架，在这些地方静态工厂方法自然而然就会返回接口类型。

Prior to Java 8, interfaces couldn’t have static methods. By convention, static factory methods for an interface named *Type* were put in a *noninstantiable companion* *class* (Item 4) named *Types.* For example, the Java Collections Framework has forty-five utility implementations of its interfaces, providing unmodifiable collections, synchronized collections, and the like. Nearly all of these implementations are exported via static factory methods in one noninstantiable class (java.util.Collections). The classes of the returned objects are all nonpublic.

在java8之前，接口类型是不能拥有静态方法的。按照约定，Type的接口类型的静态工厂方法，将会被放到一个以Types命名且不能实例化的Type接口伴侣类里。例如，java Collections Framework有45个辅助实现，这些静态工厂方法产生不可变的,同步的等等集合类型。几乎所有这些实现都是在不可变类(java.util.Collections)中的静态工厂方法中发布的。这些方法返回的对象所属的类都是非公有的。

The Collections Framework API is much smaller than it would have been had it exported forty-five separate public classes, one for each convenience implementation. It is not just the *bulk* of the API that is reduced but the *conceptual weight:* the number and difficulty of the concepts that programmers must master in order to use the API. The programmer knows that the returned object has precisely the API specified by its interface, so there is no need to read additional class documentation for the implementation class. Furthermore, using such a static factory method requires the client to refer to the returned object by interface rather than implementation class, which is generally good practice (Item 64).

JAVA集合框架API比他能导出的45个独立的公共类少的多，每个类都针对于一个的便利的实现。这不仅仅是API数量的减少，而是概念上数量减少了：要求程序员必须掌握的概念数量和难度都降低了。程序员知道这些返回的对象是由接口API所清晰描述的，因此不需要在阅读附加在类上的文档和实现类。另外，使用静态工厂方法需要客户端去指定接口作为返回对象，而不是实现类，这个在实践中很适用。

As of Java 8, the restriction that interfaces cannot contain static methods was eliminated, so there is typically little reason to provide a noninstantiable companion class for an interface. Many public static members that would have been at home in such a class should instead be put in the interface itself. Note, however, that it may still be necessary to put the bulk of the implementation code behind these static methods in a separate package-private class. This is because Java 8 requires all static members of an interface to be public. Java 9 allows private static methods, but static fields and static member classes are still required to be public.

作为JAVA8，接口不能包括静态方法的限制已经消失了，因此这里就不需要提供一个接口的不可变伴侣类。类中大部分公共静态成员应该被放到接口定义中。注意，然后，独立的包私有类中有大量的静态方法的实现，这仍然是很有必要的。这是因为java8需要接口中所有的静态成员必须是public类型。java9允许private静态方法，但是静态属性和静态成员类仍然必须是public。

**A fourth advantage of static factories is that the class of the returned object can vary from call to call as a function of the input parameters.** Any subtypeof the declared return type is permissible. The class of the returned object canalso vary from release to release.

**静态工厂方法的第四个优点是方法返回对象的类型可以根据每次调用的输入参数不同而变化。任何返回类型的子类型都可以被允许。返回对象的类型，也可以随着每次发布不同而不同。**

The EnumSet class (Item 36) has no public constructors, only static factories. In the OpenJDK implementation, they return an instance of one of two subclasses, depending on the size of the underlying enum type: if it has sixty-four or fewer elements, as most enum types do, the static factories return a RegularEnumSet instance, which is backed by a single long; if the enum type has sixty-five or more elements, the factories return a JumboEnumSet instance, backed by a long array.

就行EnumSet类没有公共构造方法，只要静态工厂方法。在OpenJDK实现中，这些方法返回实例是属于两个子类型中哪个类型，依赖于底层enum类型的大小：如果它的元素小于等于64，并且大部分enum类型都是如此，那么静态方法返回RegularEnumSet实例，后台是用一个long型实现。如果enum类型有超过了65个元素，工厂方法返回JumboEnumSet类型实例，后台是用一个long数组实现。

The existence of these two implementation classes is invisible to clients. If RegularEnumSet ceased to offer performance advantages for small enum types, it could be eliminated from a future release with no ill effects. Similarly, a future release could add a third or fourth implementation of EnumSet if it proved beneficial for performance. Clients neither know nor care about the class of the object they get back from the factory; they care only that it is some subclass of EnumSet.

对于这两个实现类的存在，客户是不可见。如果RegularEnumSet对于小enum类型在性能上不再有优势，在没有不良影响下他可能会从将来的发行版本中移除。同样，将来的发型版本可能增加第三个，或者第四个EnumSet的实现类，如果他能提供更优越的性能。客户端不需要知道也不必去关心这些来自工厂方法返回的对象类型。他们只需要关心它是EnumSet的一个子类型即可。

**A fifth advantage of static factories is that the class of the returned object need not exist when the class containing the method is written.** Such flexiblestatic factory methods form the basis of *service provider frameworks*, like the JavaDatabase Connectivity API (JDBC). A service provider framework is a system inwhich providers implement a service, and the system makes the implementationsavailable to clients, decoupling the clients from the implementations.

**第五个优点是返回对象所属的类，在编写包含该静态方法是可以不必存在。像这些具有灵活性的静态工厂方法构成服务提供框架的基础，例如java数据库连接API（jdbc）。一个服务提供框架是一个系统，服务提供者此系统中实现了这种服务，这种系统为客户提供了可用的，去耦合的服务实现。**

There are three essential components in a service provider framework: a*service interface*, which represents an implementation; a *provider registration API*, which providers use to register implementations; and a *service access API*,which clients use to obtain instances of the service. The service access API mayallow clients to specify criteria for choosing an implementation. In the absence ofsuch criteria, the API returns an instance of a default implementation, or allowsthe client to cycle through all available implementations. The service access APIis the flexible static factory that forms the basis of the service provider framework.

**在服务提供框架中有3个重要的组件：1.服务接口，他代表了一个实现。2，提供者需要注册的API，服务提供者需要使用该API注册自己的实现。3.服务访问API，用户通过该API获取提供服务的实例。服务访问API为用户提供了选择一个实现的详细的标准说明,或者允许用户在所有可用的实现中循环使用。服务访问API是一个可扩展的今天工厂方法，它构成了服务提供框架的基础。**

An optional fourth component of a service provider framework is a *service provider interface*, which describes a factory object that produce instances of theservice interface. In the absence of a service provider interface, implementationsmust be instantiated reflectively (Item 65). In the case of JDBC, Connectionplays the part of the service interface, DriverManager.registerDriver is theprovider registration API, DriverManager.getConnection is the service accessAPI, and Driver is the service provider interface.

第四个组件是可选择的——服务提供者接口。他描述了提供的服务接口实例的工厂对象。如果没有服务提供者接口，实现者必须通过反射实例化。比如JDBC, Connection扮演了服务接口的角色，DriverManager.registerDriver是服务提供者注册API，DriverManager.getConnection是服务访问API，Driver是服务提供者接口。

There are many variants of the service provider framework pattern. For example, the service access API can return a richer service interface to clients than the one furnished by providers. This is the *Bridge* pattern [Gamma95]. Dependency injection frameworks (Item 5) can be viewed as powerful service providers. Since Java 6, the platform includes a general-purpose service provider framework, java.util.ServiceLoader, so you needn’t, and generally shouldn’t, write your own (Item 59). JDBC doesn’t use ServiceLoader, as the former predates the latter.

有许多种服务提供者框架模型。例如，服务访问接口可能返回比服务提供者供应的服务接口更为丰富的服务接口。这个被称为桥接模式。注入依赖模式被认为是最强大的服务提供者。自从java6开始，java平台包括了通用服务提供者框架——java.util.ServiceLoader，因此你不需要也不应该在编写自己的服务提供者框架。JDBC没有使用ServiceLoader，因为前者在后者出来之前就已经存在了。

**The main limitation of providing only static factory methods is that classes without public or protected constructors cannot be subclassed.** Forexample, it is impossible to subclass any of the convenience implementationclasses in the Collections Framework. Arguably this can be a blessing in disguisebecause it encourages programmers to use composition instead of inheritance(Item 18), and is required for immutable types (Item 17).

一个类在没有公共或者保护构造函数的情况下，只有静态工厂方法情况下，有一个重要的局限性。例如，在Collections框架中，任何适用的实现类都不能被子类化。可以认为这一点可能假装认为成一种优势，因为根据条款18，鼓励程序员适用组合来代替继承，并且在不可变类中需要这种的静态工厂方法。

**A second shortcoming of static factory methods is that they are hard for programmers to find.** They do not stand out in API documentation in the way that constructors do, so it can be difficult to figure out how to instantiate a class that provides static factory methods instead of constructors. The Javadoc tool may someday draw attention to static factory methods. In the meantime, you can reduce this problem by drawing attention to static factories in class or interface documentation and by adhering to common naming conventions. Here are some common names for static factory methods. This list is far from exhaustive:

静态工厂方法的第二个缺点是他们很难被程序员发现。在API文档中，静态工厂方法很难像构造方法那样突出，因此很难断定通过他提供的静态方法替代构造方法是如何实例化对象的。JAVADOC工具可能在某天会给类或者接口中的静态工厂方法画一个标注，通过附着在普通命名惯例基础上。这里提供了一些普通静态方法命名规范。下面的列表远远不够：

• **from**—A *type-conversion method* that takes a single parameter and returns a corresponding instance of this type, for example:

Date d = Date.from(instant);

from-一个类型转换方法，它只有一个参数，返回一个与该类型相应的实例。例如：

Date d = Date.from(instant);

• **of**—An *aggregation method* that takes multiple parameters and returns an instance of this type that incorporates them, for example:

Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);

聚合方法，拥有多个参数，返回一个把多个参数聚合在一起的实例。例如：

Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);

• **valueOf**—A more verbose alternative to from and of, for example:

BigInteger prime = BigInteger.valueOf(Integer.MAX\_VALUE);

valueOf: 一个更详尽的由from和of的代替方案，比如

BigInteger prime = BigInteger.valueOf(Integer.MAX\_VALUE);

• **instance** or **getInstance**—Returns an instance that is described by its parameters (if any) but cannot be said to have the same value, for example: StackWalker luke = StackWalker.getInstance(options);

instance 或者 getInstance—返回一个实例，这个实例是通过参数来描述(如果有参数)，但是不是代表需要返回一个相同的值。例如:

StackWalker luke = StackWalker.getInstance(options);

• **create** or **newInstance**—Like instance or getInstance, except that the method guarantees that each call returns a new instance, for example: Object newArray = Array.newInstance(classObject, arrayLen);

create或者newInstance——与instance 或者getInstance类似，希望静态方法保证每次调用都会返回一个新的实例对象了，例如：

Object newArray = Array.newInstance(classObject, arrayLen);

• **get*Type***—Like getInstance, but used if the factory method is in a different class.

*Type* is the type of object returned by the factory method, for example: FileStore fs = Files.getFileStore(path);

getType——与getInstance类似，但是当相同的工厂方法存在于不同的类中时使用。Type是调用工厂方法返回对象的类型。例如:

for example: FileStore fs = Files.getFileStore(path);

• **new*Type***—Like newInstance, but used if the factory method is in a different class. *Type* is the type of object returned by the factory method, for example: BufferedReader br = Files.newBufferedReader(path);

newType——与newInstance类似，但是当相同的工厂方法存在于不同的类中时使用。Type是调用工厂方法返回对象的类型。例如

BufferedReader br = Files.newBufferedReader(path);

• ***type***—A concise alternative to get*Type* and new*Type*, for example: List<Complaint> litany = Collections.list(legacyLitany);

type——get*Type 和*new*Type*的简明方式，例如：

List<Complaint> litany = Collections.list(legacyLitany);

In summary, static factory methods and public constructors both have their uses, and it pays to understand their relative merits. Often static factories are preferable, so avoid the reflex to provide public constructors without first considering static factories.

总的来说，静态工厂方法和公共构造函数二者都有他们的自己的用途，理解他们优缺点是有好处的。通常静态工厂方法会更胜一筹,因此避免第一反应就采用公共构造函数，而没有考虑静态工厂方法。

**Item 2: Consider a builder when faced with many constructor parameters**

**条款2：当构造函数需要很多参数时，考虑创建一个builder方法**

Static factories and constructors share a limitation: they do not scale well to large numbers of optional parameters. Consider the case of a class representing the Nutrition Facts label that appears on packaged foods. These labels have a few required fields—serving size, servings per container, and calories per serving—and more than twenty optional fields—total fat, saturated fat, trans fat, cholesterol, sodium, and so on. Most products have nonzero values for only a few of these optional fields.

静态工厂方法和构造函数有一个共同的局限性：他们不能很好平衡数量比较多的可选参数。考虑一种情况，用一个类表示实物包装上的营养成分表。这些标签有一些必须的属性——分量数量，每个包有多少分量，每个分量包括多少卡洛里，还有超过20中可选的属性——总脂肪，饱和脂肪，反式脂肪，胆固醇，钠等等。这些可选字段中，对于大部分产品，只有期中一小部分是有非0值。

What sort of constructors or static factories should you write for such a class? Traditionally, programmers have used the *telescoping constructor* pattern, in which you provide a constructor with only the required parameters, another with a single optional parameter, a third with two optional parameters, and so on, culminating in a constructor with all the optional parameters. Here’s how it looks in practice. For brevity’s sake, only four optional fields are shown:

面对这样的类，你应该怎么编写构造函数和静态工厂方法呢？一般，程序员使用重叠构造器模式，在这方式程序员提供一个构造函数，该构造函数值必须的参数，第二构造函数包括一个可选参数，第三个构造函数使用两个构造函数，以此类推,最后一个构造函数包括了所以的可选参数。下面展示了上面说的方法在实践中的样子。为了简短，下面只展示四个可选属性。

具体代码略

When you want to create an instance, you use the constructor with the shortest parameter list containing all the parameters you want to set:

NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);

当你想要创建一个实例的时候，使用下面这个构造函数，它的参数列表只包括了你所要设置的参数。

NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);

Typically this constructor invocation will require many parameters that you don’t want to set, but you’re forced to pass a value for them anyway. In this case, we passed a value of 0 for fat. With “only” six parameters this may not seem so bad, but it quickly gets out of hand as the number of parameters increases.

典型的情况下，调用这个构造函数将需要很多参数，而这些参数你并不想设置，但是最后你无论如何得被强迫给这些参数传递一个值。在这样情况下，我们需要传递0给fat。这个只有6个参数的构成函数似乎不那么糟糕，但是当大量参数增加的时候，它很快就会脱离你的控制。

In short, **the telescoping constructor pattern works, but it is hard to write client code when there are many parameters, and harder still to read it.** Thereader is left wondering what all those values mean and must carefully countparameters to find out. Long sequences of identically typed parameters can causesubtle bugs. If the client accidentally reverses two such parameters, the compilerwon’t complain, but the program will misbehave at runtime (Item 51).

**简而言之，重叠构造器模式可以使用，但当有很多参数的时候，非常难去编写客户端代码，也非常以阅读。读者会想知道这些值意味着什么，必须仔细研究这些参数去寻找这些含义。比较长的相同类型参数序列，可能会导致微妙的bug，如果一个客户端突然颠倒了这样的两个参数，编译器将不会发现，但是程序可能会产生运行错误。**

A second alternative when you’re faced with many optional parameters in a constructor is the *JavaBeans* pattern, in which you call a parameterless constructor to create the object and then call setter methods to set each required parameter and each optional parameter of interest:

第二种情况，你面对一个有很多可选择参数的构造方法时，比如在一个javabean模式中，这这种模式中，你需要调用一个无参构造函数去创建一个对象然后调用set方法为每个必须的参数和自己感兴趣的可选参数设值：

**// JavaBeans Pattern - allows inconsistency, mandates mutability**

public class NutritionFacts {

// Parameters initialized to default values (if any)

private int servingSize = -1; // Required; no default value

private int servings = -1; // Required; no default value

private int calories = 0;

private int fat = 0;

private int sodium = 0;

private int carbohydrate = 0;

public NutritionFacts() { }

// Setters

public void setServingSize(int val) { servingSize = val; }

public void setServings(int val) { servings = val; }

public void setCalories(int val) { calories = val; }

public void setFat(int val) { fat = val; }

public void setSodium(int val) { sodium = val; }

public void setCarbohydrate(int val) { carbohydrate = val; }

}

This pattern has none of the disadvantages of the telescoping constructor pattern. It is easy, if a bit wordy, to create instances, and easy to read the resulting code:

这种模式没有**重叠构造器模式的缺点，很容易创建对象和阅读编写后的代码，除了有点冗长。**

NutritionFacts cocaCola = new NutritionFacts();

cocaCola.setServingSize(240);

cocaCola.setServings(8);

cocaCola.setCalories(100);

cocaCola.setSodium(35);

cocaCola.setCarbohydrate(27);

Unfortunately, the JavaBeans pattern has serious disadvantages of its own. Because construction is split across multiple calls, **a JavaBean may be in an** **inconsistent state partway through its construction.** The class does not have the option of enforcing consistency merely by checking the validity of the constructor parameters. Attempting to use an object when it’s in an inconsistent state may cause failures that are far removed from the code containing the bug and hence difficult to debug. A related disadvantage is that **the JavaBeans pattern** **precludes the possibility of making a class immutable** (Item 17) and requires added effort on the part of the programmer to ensure thread safety.

不幸的是，JavaBeans模式自身就有很严重的缺点。因为构建对象被分成了多次调用，一个javabean在它的整个构造过程中，部分出现一个不一致的状态。类仅仅通过检查构造方法的参数合法性，是没有强一致性的功能。试图使用一个处在非一致性状态中的对象时，可能会导致程序失败，这种失败与普通代码中包括bug有有很大区别，他增加debug的困难。一个相关的缺点是JavaBeans模式不能创建不变对象并且需要在这个上面增强对程序的维护，确保对象的线程安全。

It is possible to reduce these disadvantages by manually “freezing” the object when its construction is complete and not allowing it to be used until frozen, but this variant is unwieldy and rarely used in practice. Moreover, it can cause errors at runtime because the compiler cannot ensure that the programmer calls the freeze method on an object before using it.

为了消除这些弱点，在这些对象构造函数完成之后，手动的去“冻结”对象，一直到“冻结”完成之后，才能使用。但是这种变通，在实践中是比较笨的，而且不常用。而且，在运行中，会引起错误。因为编译器不能确保程序员在使用对象之前会调用这些“冻结”方法。

Luckily, there is a third alternative that combines the safety of the telescoping constructor pattern with the readability of the JavaBeans pattern. It is a form of the *Builder* pattern [Gamma95]. Instead of making the desired object directly, the client calls a constructor (or static factory) with all of the required parameters and gets a *builder object*. Then the client calls setter-like methods on the builder object to set each optional parameter of interest. Finally, the client calls a parameterless build method to generate the object, which is typically immutable. The builder is typically a static member class (Item 24) of the class it builds. Here’s how it looks in practice:

幸运的，有第三种选择，它把层叠构造函数模式的安全性和Javabean模式的可读性结合在一起。它是Builder模式的一种形式。客户端通过调用一个包含所有必须参数构造函数或者静态工厂方法创建一个builder对象，来代替直接创建一个想要的对象。然后，客户端调用builder对象中类似于setter的方法去设置每一个感兴趣的对象。最后，客户端调用一个没有参数的build方法去生产一个对象，这个是典型的不可变对象。builder是他所构建的类的静态成员类。下面是几种在实践中的样式：

**// Builder Pattern**

public class NutritionFacts {

private final int servingSize;

private final int servings;

private final int calories;

private final int fat;

private final int sodium;

private final int carbohydrate;

public static class Builder {

// Required parameters

private final int servingSize;

private final int servings;

// Optional parameters - initialized to default values

private int calories = 0;

private int fat = 0;

private int sodium = 0;

private int carbohydrate = 0;

public Builder(int servingSize, int servings) {

this.servingSize = servingSize;

this.servings = servings;

}

public Builder calories(int val)

{ calories = val; return this; }

public Builder fat(int val)

{ fat = val; return this; }

public Builder sodium(int val)

{ sodium = val; return this; }

public Builder carbohydrate(int val)

{ carbohydrate = val; return this; }

public NutritionFacts build() {

return new NutritionFacts(this);

}

}

private NutritionFacts(Builder builder) {

servingSize = builder.servingSize;

servings = builder.servings;

calories = builder.calories;

fat = builder.fat;

sodium = builder.sodium;

carbohydrate = builder.carbohydrate;

}

}

The NutritionFacts class is immutable, and all parameter default values are in one place. The builder’s setter methods return the builder itself so that invocations can be chained, resulting in a *fluent API*. Here’s how the client code looks:

NutritionFacts cocaCola = new NutritionFacts.Builder(240,8) .calories(100).sodium(35).carbohydrate(27).build();

This client code is easy to write and, more importantly, easy to read. **The Builder pattern simulates named optional parameters** as found in Python and Scala.

NutritionFacts类是不可变的，所有参数的默认值在只在一个地方。builder的setter方法返回builder对象本身，目的是为了进行链式调用，这是就是流式API。下面是客户端的代码:

NutritionFacts cocaCola = new NutritionFacts.Builder(240,8) .calories(100).sodium(35).carbohydrate(27).build();

客户端代码是易于编写，而且更重要的是易于阅读。Builer模式是模仿Python和scala，在它们的语法中称为可变参数。

Validity checks were omitted for brevity. To detect invalid parameters as soonas possible, check parameter validity in the builder’s constructor and methods.Check invariants involving multiple parameters in the constructor invoked by thebuild method. To ensure these invariants against attack, do the checks on objectfields after copying parameters from the builder (Item 50). If a check fails, throwan IllegalArgumentException (Item 72) whose detail message indicates whichparameters are invalid (Item 75).

合法性坚持为了简洁被省略了。为了尽可能快检测到非法的参数，检查参数合法性放在了builder的构造函数或者builder方法中。涉及多个参数不变性的检查是在build方法中调用的构造器中完成。为了确保这些不可变变量避免攻击，从buidler对象拷贝出这些参数到对象属性之后做检查，如果检查失败，抛出IllegalArgumentException异常，用这些异常的详细信息去提示哪些参数是非法。

**The Builder pattern is well suited to class hierarchies.** Use a parallel hierarchy of builders, each nested in the corresponding class. Abstract classes have abstract builders; concrete classes have concrete builders. For example, consider an abstract class at the root of a hierarchy representing various kinds of pizza:

builder模式非常适合类的继承机构。使用builder的一个并行层次结构，每个builder都嵌入到相应的类中。例如，考虑一个抽象类作为所有pizza类的根类。

**// Builder pattern for class hierarchies**

public abstract class Pizza {

public enum Topping { HAM, MUSHROOM, ONION, PEPPER, SAUSAGE }

final Set<Topping> toppings;

abstract static class **Builder<T extends Builder<T>>** {

EnumSet<Topping> toppings = EnumSet.noneOf(Topping.class);

public T addTopping(Topping topping) {

toppings.add(Objects.requireNonNull(topping));

**return self();**

}

abstract Pizza build();

**// Subclasses must override this method to return "this"**

**protected abstract T self();**

}

Pizza(Builder<?> builder) {

toppings = builder.toppings.clone(); // See Item 50

}

}

Note that Pizza.Builder is a *generic type* with a *recursive type parameter* (Item 30). This, along with the abstract self method, allows method chaining towork properly in subclasses, without the need for casts. This workaround for thefact that Java lacks a self type is known as the *simulated self-type* idiom.

注意，Pizza.Builder是一个递归类型的泛型。这样，使用抽象方法self，不需要进行类型转换，就能允许链式方法在子类中调用。实际上该方法是模仿了self-type语法，因为Java本身不包括self-type语法。

Here are two concrete subclasses of Pizza, one of which represents a standard New-York-style pizza, the other a calzone. The former has a required size parameter, while the latter lets you specify whether sauce should be inside or out:

这里有两具体的pizza实现类，一个代表了标准的New-York pizza，一个是calzone pizza。前者需要一个大小参数，后者让你指定沙拉汁是在内还是在外：

public class NyPizza extends Pizza {

public enum Size { SMALL, MEDIUM, LARGE }

private final Size size;

public static class Builder extends Pizza.Builder<Builder> {

private final Size size;

public Builder(Size size) {

this.size = Objects.requireNonNull(size);

}

@Override public NyPizza build() {

return new NyPizza(this);

}

@Override protected Builder self() { return this; }

}

private NyPizza(Builder builder) {

super(builder);

size = builder.size;

}

}

public class Calzone extends Pizza {

private final boolean sauceInside;

public static class Builder extends Pizza.Builder<Builder> {

private boolean sauceInside = false; // Default

public Builder sauceInside() {

sauceInside = true;

return this;

}

@Override public Calzone build() {

return new Calzone(this);

}

@Override protected Builder self() { return this; }

}

private Calzone(Builder builder) {

super(builder);

sauceInside = builder.sauceInside;

}

}

Note that the build method in each subclass’s builder is declared to return the correct subclass: the build method of NyPizza.Builder returns NyPizza, while the one in Calzone.Builder returns Calzone. This technique, where in a subclass method is declared to return a subtype of the return type declared in the superclass, is known as *covariant return typing*. It allows clients to use these builders without the need for casting.

注意每个子类中的builder对象里的build方法声明一个正确的返回子类：NyPizza.Builder中的build方法返回NyPizza，而Calzone.Builder中的build方法返回Calzone。子类中的方法调用返回一个子类型，这个子类型是在超累里定义的返回类型的子类型, 这种技术被称为协变返回类型。这种技术允许客户端使用这些builder而不需要做类型转换。

A minor advantage of builders over constructors is that builders can have multiple varargs parameters because each parameter is specified in its own method. Alternatively, builders can aggregate the parameters passed into multiple calls to a method into a single field, as demonstrated in the addTopping method earlier.

builder相比构造函数的还有一个小的有点，就是builder可以拥有多个可变参数，因为每个参数可以拥有自己的方法。或者，builder能聚集所有的参数到一个字段中，可以多次调用一个方法把参数传给方法，例如上面的addTopping方法。

The Builder pattern is quite flexible. A single builder can be used repeatedly to build multiple objects. The parameters of the builder can be tweaked between invocations of the build method to vary the objects that are created. A builder can fill in some fields automatically upon object creation, such as a serial number that increases each time an object is created.

Builder模式相当灵活. 一个简单的builder能重复构建多个对象。builder的参数可以在build方法调用的时候进行调整，以便能去改变所创建的对象。在对象创建的时候，builder可以自动填充某些字段，比如自增序号，每次对象被创建的时候，会自动增加一个序号。

The Builder pattern has disadvantages as well. In order to create an object, you must first create its builder. While the cost of creating this builder is unlikely to be noticeable in practice, it could be a problem in performance-critical situations. Also, the Builder pattern is more verbose than the telescoping constructor pattern, so it should be used only if there are enough parameters to make it worthwhile, say four or more. But keep in mind that you may want to add more parameters in the future. But if you start out with constructors or static factories and switch to a builder when the class evolves to the point where the number of parameters gets out of hand, the obsolete constructors or static factories will stick out like a sore thumb. Therefore, it’s often better to start with a builder in the first place.

builder模式同样也存在缺点。为了创建一个对象，你必须首先创建他的builder。实践中，创建builder的性能代价通常没有被考虑到，在对性能很敏感的系统中的这确实是一个问题。第二，builder模式相比层次构造函数更冗长，因此，他应该仅仅被用在有足够多的参数的时候才能显出他的价值，比如4个或者4个以上参数。但是记住，在未来你可能会增加参数。但是，如果你以构造函数或者静态工厂方法开始，然后向builder转向，当类发展到一个时间点，这个点是参数的个数已经失控，淘汰的构造函数和静态方法将会像一个疼痛的大拇指一样，格外的突出。因此，首要就是使用builder模式。

In summary, **the Builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters**, especially if many of the parameters are optional or of identical type.Client code is much easier to read and write with builders than with telescopingconstructors, and builders are much safer than JavaBeans.

**总的来说，builder模式是一个很好的选择，在所要设计的类的构造函数和静态工厂方法有很多参数的时候，尤其是有很多可选参数或者是同一类型的参数。相比层叠式的构造函数，客户端代码使用builder非常容易理解和编写，而且比javabeans更加安全。**

**Item 3: Enforce the singleton property with a private constructor or an enum type**

条款3 利用私有构造函数或者枚举实现单例

A *singleton* is simply a class that is instantiated exactly once [Gamma95]. Singletons typically represent either a stateless object such as a function (Item 24) or a system component that is intrinsically unique. **Making a class a singleton can** **make it difficult to test its clients** because it’s impossible to substitute a mock implementation for a singleton unless it implements an interface that serves as its type.

单例是一个只能被实例化一次的类。单例典型代表一个无状态的对象，比如一个函数或者一个系统组件，此组件是独一。创建的单例类使得测试它的客户端变的很困难，因为不能创建一个mock实现去替代单例，除非它实现的是一个接口，该接口作为提供服务的类型。

There are two common ways to implement singletons. Both are based on keeping the constructor private and exporting a public static member to provide access to the sole instance. In one approach, the member is a final field:

有两种方法去实现单例。着两种方法都是需要保证构造函数的私有，开放一个公共的静态方法，提供方法单例实例。第一种方式如下，使用成员是final类型。

**// Singleton with public final field**

public class Elvis {

**public static final Elvis INSTANCE = new Elvis();**

private Elvis() { ... }

public void leaveTheBuilding() { ... }

}

The private constructor is called only once, to initialize the public static final field Elvis.INSTANCE. The lack of a public or protected constructor *guarantees* a “monoelvistic” universe: exactly one Elvis instance will exist once the Elvis class is initialized—no more, no less. Nothing that a client does can change this, with one caveat: a privileged client can invoke the private constructor reflectively (Item 65) with the aid of the AccessibleObject.setAccessible method. If you need to defend against this attack, modify the constructor to make it throw an exception if it’s asked to create a second instance.

为了初始化公共静态final域INSTANCE，私有构造函数只能被调用一次。没有公共或者保护构造函数保证了唯一性:当Elvis实例化的时候，恰好只有一个Elvis实例存在，不多不少。在任何情况下客户端都不能对此进行改变，这里有一个警告：一个特权客户可以通过反射AccessibleObject.setAccessible调用私有构造函数.如果你需要保护这种攻击，修改构造函数使得它能够在第二次创建对象的时候抛出异常。

In the second approach to implementing singletons, the public member is a static factory method:

第二种方式实现单例，通过公开的静态工厂方法：

**// Singleton with static factory**

public class Elvis {

**private** static final Elvis INSTANCE = new Elvis();

private Elvis() { ... }

**public static Elvis getInstance()** { return INSTANCE; }

public void leaveTheBuilding() { ... }

}

All calls to Elvis.getInstance return the same object reference, and no other Elvis instance will ever be created (with the same caveat mentioned earlier).

所以调用Elvis.getInstance方法将返回同一个对象的引用。无论如何都没有新的Elvis实例被创建（相同的警告见上文）

The main advantage of the public field approach is that the API makes it clear that the class is a singleton: the public static field is final, so it will always contain the same object reference. The second advantage is that it’s simpler.

public域的方式的优点是:单例的API很清楚表达此类是单例，因为该静态public域是final，因此总是保护了同一个对象的引用。第二个优点是它很简单。

One advantage of the static factory approach is that it gives you the flexibility to change your mind about whether the class is a singleton without changing its API. The factory method returns the sole instance, but it could be modified to return, say, a separate instance for each thread that invokes it. A second advantage is that you can write a *generic singleton factory* if your application requires it(Item 30). A final advantage of using a static factory is that a *method reference* can be used as a supplier, for example Elvis::instance is a Supplier<Elvis>.Unless one of these advantages is relevant, the public field approach is preferable.

静态工厂方法的优点是：它给你提供了可扩展性，根据自己的想法决定类是否是单例，而不需修改API。静态工厂方法返回一个单例，但是他可以在修改后返回，比如说，每个线程调用他之后返回一个不同的实例。第二个优点是，你可以变现一个通用的单例工厂方法，如果你的应用需要单例。使用静态工厂方法的最后优点是方法引用可以作为supplier被使用，例如：Elvis::instance是一个Supplier<Elvis>，除了以上这些优点，使用public域方式更为合适。

To make a singleton class that uses either of these approaches *serializable*(Chapter 12), it is not sufficient merely to add implements Serializable to itsdeclaration. To maintain the singleton guarantee, declare all instance fieldstransient and provide a readResolve method (Item 89). Otherwise, each time aserialized instance is deserialized, a new instance will be created, leading, in thecase of our example, to spurious Elvis sightings. To prevent this from happening, add this readResolve method to the Elvis class:

如果想要通过上面两张方法创建的单例可以序列化，在声明中增加Serializable接口是不够的。为了保证单例，需要声明所有的域是transient，提供一个readResolve方法。否则每次序列化后的实例在反序列化后，一个新的实例被创建，在我们的例子中，导致看见一个伪造的Elvis。防止这样的事情的反射，需要Elvis给增加一个readResolve方法。

**// readResolve method to preserve singleton property**

private Object readResolve() {

// Return the one true Elvis and let the garbage collector

// take care of the Elvis impersonator.

return INSTANCE;

}

A third way to implement a singleton is to declare a single-element enum:

第三种方法是声明一个单例元素的枚举。

**// Enum singleton - the preferred approach**

public enum Elvis {

INSTANCE;

public void leaveTheBuilding() { ... }

}

This approach is similar to the public field approach, but it is more concise, provides the serialization machinery for free, and provides an ironclad guarantee against multiple instantiation, even in the face of sophisticated serialization or reflection attacks. This approach may feel a bit unnatural, but **a single-element** **enum type is often the best way to implement a singleton**. Note that you can’t use this approach if your singleton must extend a superclass other than Enum(though you *can* declare an enum to implement interfaces).

这种方法与public域方式类似，但是它提供了更多的选择，提供了一个免费的序列化机制,并且为防止多实例化提供了坚固的保证。即使面对复杂的序列化和反射攻击。虽然这种方法让人感觉有点不自然，但是单例元素的枚举类型是最好的实现单子方式。注意，如果当你的单例类必须继承父类（除枚举以外，但你可以声明一个枚举实现接口）的时候，不能使用这种方式。

**Item 4: Enforce noninstantiability with a private constructor**

条款4 使用私有构造函数执行不可实例化

Occasionally you’ll want to write a class that is just a grouping of static methods and static fields. Such classes have acquired a bad reputation because some people abuse them to avoid thinking in terms of objects, but they do have valid uses. They can be used to group related methods on primitive values or arrays, in the manner of java.lang.Math or java.util.Arrays. They can also be used to group static methods, including factories (Item 1), for objects that implement some interface, in the manner of java.util.Collections. (As of Java 8, you can also put such methods in the interface, assuming it’s yours to modify.) Lastly, such classes can be used to group methods on a final class, since you can’t put them in a subclass.

偶尔，你创建一个类，这个类只包括静态方法和静态域。这些类拥有不好的名声，因为有些人滥用这些类，而没有仔细考虑过这些对象。但是，这些类在用的时候，确实比较有效。这些类被根据有关联的方法进行分类，这些方法用来操作原始值或者数组，例如Math和Arrays。他们可以根据静态方法（包括静态工厂），这些类型通常是实现了某些接口，比如java.util.Collections。(在java8中，你能把这些方法放到接口中，假设这些类是自己定义的)最后，这些类可以把方法聚合在一个final类中，因为你不能把这些放放到子类中。

Such utility classes were not designed to be instantiated: an instance would be nonsensical. In the absence of explicit constructors, however, the compiler provides a public, parameterless default constructor. To a user, this constructor is indistinguishable from any other. It is not uncommon to see unintentionally instantiable classes in published APIs.

这些工具类设计之初不是为了实例化:因为实例对象是没有意义的。当缺少明确的构造函数的时，编译器会提供一个公共的，无参的默认构造函数。对于使用者，这个构造函数与气体构造函数没有区别。当看到并非成心使用这些已经发布的APIs创建的实例对象也不稀奇。

**Attempting to enforce noninstantiability by making a class abstract does not work.** The class can be subclassed and the subclass instantiated. Furthermore,it misleads the user into thinking the class was designed for inheritance (Item 19).There is, however, a simple idiom to ensure noninstantiability. A default constructoris generated only if a class contains no explicit constructors, so **a class can be made noninstantiable by including a private constructor**:

**通过创建abstarct类的方法来阻止类的实例化，是这方法并不好。这个类可以被子类化，而子类是可以被实例化。另外，这种方法会误导用户以为这个类被设计是用来继承。这里有一种实现非实例化的管用方法。默认的构造函数被生成仅当一个类没有提供明确的构造函数，因此一个类可以通过提供一个私有的构造函数来实现非实例化。**

**// Noninstantiable utility class**

public class UtilityClass {

**// Suppress default constructor for noninstantiability**

private UtilityClass() {

throw new AssertionError();

}

... // Remainder omitted

}

Because the explicit constructor is private, it is inaccessible outside the class. The AssertionError isn’t strictly required, but it provides insurance in case the constructor is accidentally invoked from within the class. It guarantees the class will never be instantiated under any circumstances. This idiom is mildly counterintuitive because the constructor is provided expressly so that it cannot be invoked. It is therefore wise to include a comment, as shown earlier.

因为明确提供私有的构造函数，它不能在类的外部访问。AssertionError也不是一定要提供，但是AssertionError提供了一种保障，以防私有构造函数由于失误在类的内部被调用。它保证了类在任何情况下永远不能被实例化。使用这种方法，明确的提供一个构造函数目的就是不能被调用，这一点稍微有点违反直觉。明智的做法就是增加注释，例如上文所示。

As a side effect, this idiom also prevents the class from being subclassed. All constructors must invoke a superclass constructor, explicitly or implicitly, and a subclass would have no accessible superclass constructor to invoke.

此种方法有一种副作用，它阻止了此类被继承。所有的构造函数必须显式或者隐式的调用父类的构造构造函数，这种方式子类失去了访问父类构造函数的权限。

**Item 5: Prefer dependency injection to hardwiring resources**

优先选择依赖注入而不是硬编码的资源文件

Many classes depend on one or more underlying resources. For example, a spell checker depends on a dictionary. It is not uncommon to see such classes implemented as static utility classes (Item 4):

许多类依赖于一个或者多个基础的资源文件。例如，一个拼写检查器依赖于字典。经常见到这些类作为静态的工具类实现。

**// Inappropriate use of static utility - inflexible & untestable!**

public class SpellChecker {

private static final Lexicon dictionary = ...;

private SpellChecker() {} // Noninstantiable

public static boolean isValid(String word) { ... }

public static List<String> suggestions(String typo) { ... }

}

Similarly, it’s not uncommon to see them implemented as singletons (Item 3):

同样，经常看到这些类做为单例而实现。

**// Inappropriate use of singleton - inflexible & untestable!**

public class SpellChecker {

private final Lexicon dictionary = ...;

private SpellChecker(...) {}

public static INSTANCE = new SpellChecker(...);

public boolean isValid(String word) { ... }

public List<String> suggestions(String typo) { ... }

}

Neither of these approaches is satisfactory, because they assume that there is only one dictionary worth using. In practice, each language has its own dictionary, and special dictionaries are used for special vocabularies. Also, it may be desirable to use a special dictionary for testing. It is wishful thinking to assume that a single dictionary will suffice for all time.

这些方法都不是很符合要求，因为他们都假定只有一个字典值可用。实际中，每种语音都有自己的字段，并且特殊的字典会用在特殊的词汇上。另外，使用特殊的字段去测试可能才会满足需要。理想的想法假定是任何时候只需要一个字典就能满足所有的需求。

You could try to have SpellChecker support multiple dictionaries by making the dictionary field nonfinal and adding a method to change the dictionary in an existing spell checker, but this would be awkward, error-prone, and unworkable in a concurrent setting. **Static utility classes and singletons are inappropriate for** **classes whose behavior is parameterized by an underlying resource.**

你能够尝试拥的spellchecker可以支持多个字段通过把字典字典定义称为非final，并且增加一个方法使用存在的拼写检查器去改变字典，但是这种方法是笨的，易出错的，并且在并发设置的时候无法工作。静态工具类和单例对于这些类是不恰当的，这些静态工具类或者单子的行为是通过基础配置文件进行参数化。

What is required is the ability to support multiple instances of the class (in our example, SpellChecker), each of which uses the resource desired by the client (in our example, the dictionary). A simple pattern that satisfies this requirement is to **pass the resource into the constructor when creating a new instance**. This is one form of *dependency injection*: the dictionary is a *dependency* of the spell checker and is *injected* into the spell checker when it is created.

类需要一种可以支持类多个实例的能力（在上面的例子中，SpellChecker），根据客户（这里是，dictionary）的期望使用每种资源文件。下面这个简单的模式满足这个需求：当创建实例的时候把资源传递给构造器。这就是一种依赖注入的形式：spell checker依赖dictionary,并且当他被创建的时候注入到spellchecker中。

**// Dependency injection provides flexibility and testability**

public class SpellChecker {

private final Lexicon dictionary;

**public SpellChecker(Lexicon dictionary)** {

this.dictionary = Objects.requireNonNull(dictionary);

}

public boolean isValid(String word) { ... }

public List<String> suggestions(String typo) { ... }

}

The dependency injection pattern is so simple that many programmers use it for years without knowing it has a name. While our spell checker example had only a single resource (the dictionary), dependency injection works with an arbitrary number of resources and arbitrary dependency graphs. It preserves immutability (Item 17), so multiple clients can share dependent objects (assuming the clients desire the same underlying resources). Dependency injection is equally applicable to constructors, static factories (Item 1), and builders (Item 2).

依赖注入这种模式非常简单，以至于许多程序员使用了好多年却不知道它的名字。当我们的例子中的拼写检测器只有单个资源（字典）的时候，依赖注入工作在任意数量的资源和任意依赖图中。依赖注入保护了不变性。因此，多客户端可以共享依赖对象（假设多个客户端使用同一个基础资源）。依赖注入同样适合于构造函数，静态方法和builder。

A useful variant of the pattern is to pass a resource *factory* to the constructor. A factory is an object that can be called repeatedly to create instances of a type. Such factories embody the *Factory Method* pattern [Gamma95]. The Supplier<T> interface, introduced in Java 8, is perfect for representing factories. Methods that take a Supplier<T> on input should typically constrain the factory’s type parameter using a *bounded wildcard type* (Item 31) to allow the client to pass in a factory that creates any subtype of a specified type. For example, here is a method that makes a mosaic using a client-provided factory to produce each tile:

这种模式的另外一种形式是传递资源工厂给构造函数。这个工厂是一个对象，它能够被重复调用去创建一个类型的实例。这种工厂把工厂方法设计模式具体化了。java8中引入的Supplier<T>接口是这种工厂的完美实现。这使用Supplier<T>作为参数的方法，应该典型的强迫工厂类型的参数使用泛型，使得可以允许客户传递给工厂类型的参数是他指定的类型的子类型。例如，这个的方法，它创建一个Mosaic对象，使用客户提供的工厂去创建每个tile。

Mosaic create(Supplier<? extends Tile> tileFactory) { ... }

Although dependency injection greatly improves flexibility and testability, it can clutter up large projects, which typically contain thousands of dependencies. This clutter can be all but eliminated by using a *dependency injection framework*, such as Dagger [Dagger], Guice [Guice], or Spring [Spring]. The use of these frameworks is beyond the scope of this book, but note that APIs designed for manual dependency injection are trivially adapted for use by these frameworks.

虽然依赖注入极大的改善了灵活性和易测试性，但是他工程变得杂乱，这些工程中包含了数以千计的依赖。这种杂乱可以通过使用注入框架，比如Dagger，Guice，Spring而消除。对于这些框架的使用超过了本书的范围，但是，注意，这些手动注入API的设计完全能够适合这些框架。

In summary, do not use a singleton or static utility class to implement a class that depends on one or more underlying resources whose behavior affects that of the class, and do not have the class create these resources directly. Instead, pass the resources, or factories to create them, into the constructor (or static factory or builder). This practice, known as dependency injection, will greatly enhance the flexibility, reusability, and testability of a class.

总的来说，当一个类依赖一个或多个基础资源时候，这些资源的行为会影响整个类。不要使用单例或者静态工厂去实现该类，也不要让这类直接去创建这些资源。而是把资源，或者通过工厂创建资源直接传入构造器（或者静态工厂方法或者builder）。这种实践方式，众所周知的依赖注入形式，将很好的增加类的扩展性，可重用性和易测试性。

**Item 6:** **Avoid creating unnecessary objects**

**条款6 避免创建不必要的对象**

It is often appropriate to reuse a single object instead of creating a new functionally equivalent object each time it is needed. Reuse can be both faster and more stylish. An object can always be reused if it is immutable (Item 17).

通常复用单例来代替创建一个新的具有相同功能性的对象是很合理的。复用即使快速又流行。如果一个类不可变，那么他总能被复用。

As an extreme example of what not to do, consider this statement:

那么是一个什么也没有做的极端例子，考虑下面的语句：

String s = new String("bikini"); // DON'T DO THIS!

The statement creates a new String instance each time it is executed, and none of those object creations is necessary. The argument to the String constructor ("bikini") is itself a String instance, functionally identical to all of the objects created by the constructor. If this usage occurs in a loop or in a frequently invoked method, millions of String instances can be created needlessly. The improved version is simply the following:

String s = "bikini";

这个语句每次调用的时候都会创建了一个新的String实例，但是这些对象的创建都不是必须的。String的构造函数的参数bikini本身就是一个字符串，功能上讲所有的对象都一样,他们的创建都是通过构造函数。如果这种用法发生在一个循环中，或者一个频繁调用的方法中，成千上万的不必要的String实例将被创建。改进的版本是下面简单的语句：

String s = "bikini";

This version uses a single String instance, rather than creating a new one each time it is executed. Furthermore, it is guaranteed that the object will be reused by any other code running in the same virtual machine that happens to contain the same string literal [JLS, 3.10.5].

这个版本中使用的是一个单个String实例，而不是每次调用的时候，创建一个新的String实例。另外，它还保证了对象将会被其他运行在同一台java虚拟机上的代码复用，当其他代码碰巧使用了这个相同的string字面量。

You can often avoid creating unnecessary objects by using *static factory methods* (Item 1) in preference to constructors on immutable classes that provide both.For example, the factory method Boolean.valueOf(String) is preferable to theconstructor Boolean(String), which was deprecated in Java 9. The constructor *must* create a new object each time it’s called, while the factory method is neverrequired to do so and won’t in practice. In addition to reusing immutable objects,you can also reuse mutable objects if you know they won’t be modified.

你可以通过优先使用静态方法而不是构造函数避免创建不必要的对象，如果这二者类都提供。例如使用静态工厂方法Boolean.valueOf(String)就由于使用构造函数Boolean(String)，这个构造方法在java9中已经抛弃。构造函数在每次调用的时候必须创建一个新对象，而静态工厂方法从来不需要这样做而且在实践中也不会这样做。除了为了重用不变对象，如果你知道他们在使用的时候不会改变，你也可以重用可变对象。

Some object creations are much more expensive than others. If you’re goingto need such an “expensive object” repeatedly, it may be advisable to cache it forreuse. Unfortunately, it’s not always obvious when you’re creating such an object.Suppose you want to write a method to determine whether a string is a validRoman numeral. Here’s the easiest way to do this using a regular expression:

有些对象的创建比其他对象花费更多的代价。如果你重复需要这样的代价高的对象，明智的选择就是为了重复使用，需要把它缓存起来。不幸的，当你创建这样的对象的时候，其实并不总是很明显。假设你想要写一个方法去判断一个字符串是否是合肥的罗马数字。这里有最简单的方法去实现这个功能——使用正则表达式：

**// Performance can be greatly improved!**

static boolean isRomanNumeral(String s) {

return s.matches("^(?=.)M\*(C[MD]|D?C{0,3})"

+ "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

}

The problem with this implementation is that it relies on the String.matches method. **While String.matches is the easiest way to check if a string matches** **a regular expression, it’s not suitable for repeated use in performance-critical** **situations.** The problem is that it internally creates a Pattern instance for the regular expression and uses it only once, after which it becomes eligible for garbage collection. Creating a Pattern instance is expensive because it requires compiling the regular expression into a finite state machine.

这种实现方式的问题是它依赖于String.matches的方法。虽然String.matches是最简单的方法去检查一个字符是不是与该正则表达式相匹配，但是它不适合在一个对性能要求很严的环境中重复调用。它存在的问题是该方法内部会为正则表达式创建一个Pattern实例并且只是使用一次，使用之后，就会成为垃圾回收的对象。创建一个Pattern是非常消耗资源的，因为它需要把正则表达式编译一个有限状态机。

To improve the performance, explicitly compile the regular expression into a Pattern instance (which is immutable) as part of class initialization, cache it, and reuse the same instance for every invocation of the isRomanNumeral method:

为了提升性能，需要明确的把正则表达式编译成为Patten实例作为类实现的一部分，这个实例是不可变的，把它缓存起来，然后每次在调用isRomanNumeral方法的时候重用这个实例。

**// Reusing expensive object for improved performance**

public class RomanNumerals {

private static final Pattern ROMAN = Pattern.compile(

"^(?=.)M\*(C[MD]|D?C{0,3})"

+ "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

static boolean isRomanNumeral(String s) {

return ROMAN.matcher(s).matches();

}

}

The improved version of isRomanNumeral provides significant performance gains if invoked frequently. On my machine, the original version takes 1.1 μs on an 8-character input string, while the improved version takes 0.17 μs, which is 6.5 times faster. Not only is the performance improved, but arguably, so is clarity. Making a static final field for the otherwise invisible Pattern instance allows us to give it a name, which is far more readable than the regular expression itself.

如果重复调用isRomanNumeral，它的改进版本提供了重要的性能提升。在我的机器上，原始版本在处理8个字符的时候花费1.1us秒，而在改进版本中只花费0.17us,速度提升了6.5倍。不仅性能得到了很大的提升，无可厚非的是，这种也更加清晰。把其他不可见的Pattern实例设置为final静态域，这样允许我们给他命名一个名称，这个将比正则表达式本身更具有可读性。

If the class containing the improved version of the isRomanNumeral method is initialized but the method is never invoked, the field ROMAN will be initialized needlessly. It would be possible to eliminate the initialization by *lazily initializing* the field (Item 83) the first time the isRomanNumeral method is invoked, but this is *not* recommended. As is often the case with lazy initialization, it would complicate the implementation with no measurable performance improvement (Item 67).

如果一个保护改进版本的isRomanNumeral方法的类被初始化，但是这个方法从未被调用，那么字段ROMAN就没有必要被初始化。通过在第一次调用isRomanNumeral方法的时候，使用字段的懒惰式初始化的方法可以消除这种无用的初始化，但是并不推荐这么做。与惰性初始化方式一样，这种方式使得实现变得复杂，而且对性能的提升也没有多少改变。

When an object is immutable, it is obvious it can be reused safely, but there are other situations where it is far less obvious, even counterintuitive. Consider the case of *adapters* [Gamma95]*,* also known as *views*. An adapter is an object that delegates to a backing object, providing an alternative interface. Because an adapter has no state beyond that of its backing object, there’s no need to create more than one instance of a given adapter to a given object.

当一个对象是不可变的，很显然它可以安全的被重用，但是有其他不常见的情况，甚至是违反直觉的。细想一下适配器，通常也称为试图。一个适配器是一个对象，它代表了他所适配的对象，提供了一个可选择的接口。因为一个适配器除去他适配的对象它没有状态，所以不需要给一个既定的对象创建更多的适配器实例。

For example, the keySet method of the Map interface returns a Set view of the Map object, consisting of all the keys in the map. Naively, it would seem that every call to keySet would have to create a new Set instance, but every call to keyset on a given Map object may return the same Set instance. Although the returned Set instance is typically mutable, all of the returned objects are functionally identical: when one of the returned objects changes, so do all the others, because they’re all backed by the same Map instance. While it is largely harmless to create multiple instances of the keySet view object, it is unnecessary and has no benefits.

例如，Map接口的keyset方法返回一个包括所有key的Map对象的set视图。天真地认为，每次调用keyset方法都会创建一个新的set实例，但是在一个给定的map对象上每次调用keyset方法可能只会返回同一个set实例。虽然返回的set实例是可变的，但是所有的返回对象在功能上都是相同的：当一个返回对象发生改变，这些改变会体现在其他返回对象上。因此他们的背后是同一个map实例。虽然当创建多个keyset视图对象是没有什么危害，但是这也不是必需且没有易处。

Another way to create unnecessary objects is *autoboxing*, which allows the programmer to mix primitive and boxed primitive types, boxing and unboxing automatically as needed. **Autoboxing blurs but does not erase the distinction** **between primitive and boxed primitive types.** There are subtle semantic distinctions and not-so-subtle performance differences (Item 61). Consider the following method, which calculates the sum of all the positive int values. To do this, the program has to use long arithmetic because an int is not big enough to hold the sum of all the positive int values:

另一种创建多个不必要的对象是自动装箱操作，自动装箱操作允许程序员把原始数据类型和原始数据类型的装箱类型混合使用，装箱和拆箱都是根据需要自动完成。s虽然自动装箱比较模糊但是不能消除原始数据类型和包装类之间的区别。他们之间有微妙的语义上的差别，也有不那么微妙的性能上的差别。仔细思考下面的方法，计算所有int型的正数和。为了实现这个功能，程序需要使用Long型，因为一个int型不足以保存所有int型的正数和。

**// Hideously slow! Can you spot the object creation?**

private static long sum() {

Long sum = 0L;

for (long i = 0; i <= Integer.MAX\_VALUE; i++)

sum += i;

return sum;

}

This program gets the right answer, but it is *much* slower than it should be, due to a one-character typographical error. The variable sum is declared as a Long instead of a long, which means that the program constructs about 231 unnecessary Long instances (roughly one for each time the long i is added to the Long sum). Changing the declaration of sum from Long to long reduces the runtime from 6.3 seconds to 0.59 seconds on my machine. The lesson is clear: **prefer primitives to** **boxed primitives, and watch out for unintentional autoboxing.**

这个程序可以获得正确的答案，但是他比他能达到的速度要慢很多，因为一个字符的印刷错误。变量sum被声明为了一个Long代替long，它的意思是程序构建了2的31次方个不必要的Long变量（虽然每次long i增加到了Long型的sum和中）。把Long修改为long的声明，运行时间从6.3秒减少到了0.59秒在我的机器上。教训是很清晰：使用原始数据类型去替代包装类型,并且小心无意识的自动装箱操作。

This item should not be misconstrued to imply that object creation is expensive and should be avoided. On the contrary, the creation and reclamation of small objects whose constructors do little explicit work is cheap, especially on modern JVM implementations. Creating additional objects to enhance the clarity, simplicity, or power of a program is generally a good thing.

这个条款不应该被误认为是创建对象是非常昂贵的，应该避免去创建对象。正相反，小对象的创建与回收，由于他们的构造函数做的东西很少，因此明白的说，这些消耗资源很少，尤其是面对现代的JVM虚拟机实现。创建附加对象会使程序变的清晰，简单，有力，通常来说这也是一件好事。

Conversely, avoiding object creation by maintaining your own *object pool* is a bad idea unless the objects in the pool are extremely heavyweight. The classic example of an object that *does* justify an object pool is a database connection. The cost of establishing the connection is sufficiently high that it makes sense to reuse these objects. Generally speaking, however, maintaining your own object pools clutters your code, increases memory footprint, and harms performance. Modern JVM implementations have highly optimized garbage collectors that easily outperform such object pools on lightweight objects.

相反，通过创建自己的对象池来避免对象的创建是一个糟糕的注意，除非这些池中的对象都是重量级的对象。满足对象池的合适的对象例子就是数据库连接对象。建立数据库的连接的消耗是非常巨大，因此减少这些对象的创建才变的有意义。通常来说，维持自己定义的连接池会使得自己的代码变得混乱,增加了内存的占用，并且对性能有害。现在JVM虚拟机的实现有很好的对垃圾回收器，这些垃圾回收器性能早就已经超过了这些轻量级对象的对象池

The counterpoint to this item is Item 50 on *defensive copying*. The present item says, “Don’t create a new object when you should reuse an existing one,” while Item 50 says, “Don’t reuse an existing object when you should create a new one.” Note that the penalty for reusing an object when defensive copying is called for is far greater than the penalty for needlessly creating a duplicate object. Failing to make defensive copies where required can lead to insidious bugs and security holes; creating objects unnecessarily merely affects style and performance.

与此条款相对的是条款50“保卫性复制”。当前的条款说的是不要创建新对象当你应该重复使用一个存在的对象。而条款50描述得是：“不要重用一个已经存在的对象，当你应该创建新一个新对象”。注意,当保卫行复制被调用的时候，重用对象的坏处远远大于创建不必要的重复对象的坏处。当在需要进行保护性复制的地方，创建保护性复制失败会导致隐藏的bug和安全漏洞；而新建不必要的对象仅仅只是影响了风格和性能。

**Item 7: Eliminate obsolete object references**

条款7 消除无用的对象引用

If you switched from a language with manual memory management, such as C or C++, to a garbage-collected language such as Java, your job as a programmer was made much easier by the fact that your objects are automatically reclaimed when you’re through with them. It seems almost like magic when you first experience it. It can easily lead to the impression that you don’t have to think about memory management, but this isn’t quite true.

如果你从手动内存管理的语言（比如C++）转到垃圾自动回收的语言中（比如java）你作为程序员的工作变得非常简单，这是因为一个事实：你自己定义的对象可以自动的被清理当你使用完成这些对象之后。当你第一次使用的时候，会感觉非常神奇。它会给人一种感觉，就是你不需要考虑内存管理，然后这并不是真的。

Consider the following simple stack implementation:

考虑下面的简单的stact的实现。

public class Stack {

private Object[] elements;

private int size = 0;

private static final int DEFAULT\_INITIAL\_CAPACITY = 16;

public Stack() {

elements = new Object[DEFAULT\_INITIAL\_CAPACITY];

}

public void push(Object e) {

ensureCapacity();

elements[size++] = e;

}

public Object pop() {

if (size == 0)

throw new EmptyStackException();

return elements[--size];

}

/\*\*

\* Ensure space for at least one more element, roughly

\* doubling the capacity each time the array needs to grow.

\*/

private void ensureCapacity() {

if (elements.length == size)

elements = Arrays.copyOf(elements, 2 \* size + 1);

}

}

There’s nothing obviously wrong with this program (but see Item 29 for a generic version). You could test it exhaustively, and it would pass every test with flying colors, but there’s a problem lurking. Loosely speaking, the program has a “memory leak,” which can silently manifest itself as reduced performance due to increased garbage collector activity or increased memory footprint. In extreme cases, such memory leaks can cause disk paging and even program failure with an OutOfMemoryError, but such failures are relatively rare.

这个程序没有明显的错误（但是请看条款29的普通版本）。你可以耗尽所有去测试它，最后每次都能通过测试。但是这里有一个潜伏的问题。简单的说，程序存在内存泄漏，随着垃圾回收和内存占用的不断增加，性能逐步下降，使得它的问题会慢慢显露出来。极端情况下，这些内存泄漏会导致磁盘分页并且严重的时候，会引起OutOfMemoryError错误，当然了这些文件是相当罕见的。

So where is the memory leak? If a stack grows and then shrinks, the objects that were popped off the stack will not be garbage collected, even if the program using the stack has no more references to them. This is because the stack maintains obsolete references to these objects. An obsolete reference is simply a reference that will never be dereferenced again. In this case, any references outside of the “active portion” of the element array are obsolete. The active portion consists of the elements whose index is less than size.

那么，内存泄漏发生在哪里？如果一个stack增长之后又缩小——对象被从stack中弹出将不会出发垃圾回收，即使程序中使用的stack已经没有对这些对象的引用了。这是因为stack保持着这些对象废弃的引用。一个废弃的引用是一个普通的引用，这个引用不会被再次引用。这种情况下，任何引用在数组活跃索引位置之外的引用都是废弃引用。数组的活跃位置组成的数据元素，他们的索引位置是小于数组大小的。

Memory leaks in garbage-collected languages (more properly known as *unintentional*

*object retentions*) are insidious. If an object reference is unintentionally retained, not only is that object excluded from garbage collection, but so too are any objects referenced by that object, and so on. Even if only a few object references are unintentionally retained, many, many objects may be prevented from being garbage collected, with potentially large effects on performance.

内存泄漏隐藏在提供垃圾回收的编程语言中（更多是没有意识留下的对象引用）。如果一个对象引用无意中被保留，那么这个对象不但将会从垃圾回收中逃脱，而且被这个对象所引用的其他队都会从垃圾回收中逃脱。即使是很少的对象被无意识的引用，那么将会有很多很多的对象被阻止在垃圾回收之外，这样会对性能产生潜在的影响。

The fix for this sort of problem is simple: null out references once they

become obsolete. In the case of our Stack class, the reference to an item becomes

obsolete as soon as it’s popped off the stack. The corrected version of the pop

method looks like this:

对这类问题的修正非常简单：当这些引用成为废弃引用的时候，给他们赋null。在我们的stack类中，当数组中的对象从stack中弹出后，对应的引用变为废弃，正确的pop方法应该如下所示：

public Object pop() {

if (size == 0)

throw new EmptyStackException();

Object result = elements[--size];

**elements[size] = null; // Eliminate obsolete reference**

return result;

}

An added benefit of nulling out obsolete references is that if they are subsequently dereferenced by mistake, the program will immediately fail with a NullPointerException, rather than quietly doing the wrong thing. It is always beneficial to detect programming errors as quickly as possible.

把null赋值给废弃引用的好处是，当他们后面被间接误用,程序会立即抛出NullPointerException异常，而不是悄无声息的去做错误的事情。对于尽可能快的探测程序的错误总是有益处的。

When programmers are first stung by this problem, they may overcompensate by nulling out every object reference as soon as the program is finished using it. This is neither necessary nor desirable; it clutters up the program unnecessarily. **Nulling out object references should be the exception rather than the norm.** The best way to eliminate an obsolete reference is to let the variable that contained the reference fall out of scope. This occurs naturally if you define each variable in the narrowest possible scope (Item 57).

当程序员遭遇过这样的问题之后，他们可能会过度使用null给对象应用的赋值，当程序一完成，就给程序对象赋null。这种做法既不是必要的，效果也不令人满意。它会给程序带来不必要的混乱。给对象赋null应该是作为一种异常处理方式，而不应该作为一种规范。消除废弃的引用的最好方式是让包含引用的变量离开他的作用域。这个应该是很自然的发生，如果你定义每个变量在一个尽可能窄的作用域中。

So when should you null out a reference? What aspect of the Stack class makes it susceptible to memory leaks? Simply put, it *manages its own memory*. The *storage pool* consists of the elements of the elements array (the object reference cells, not the objects themselves). The elements in the active portion of the array (as defined earlier) are *allocated*, and those in the remainder of the array are *free*. The garbage collector has no way of knowing this; to the garbage collector, all of the object references in the elements array are equally valid. Only the programmer knows that the inactive portion of the array is unimportant. The programmer effectively communicates this fact to the garbage collector by manually nulling out array elements as soon as they become part of the inactive portion.

什么时候对引用赋null值？stack类的哪些方面使得stack容易引起内存泄漏？简单说，它自己管理自己的内存。存储池是由数据数组的元素组成，数组中的对象引用的单元，不是对象自己本身。元素被分配到了数组的活跃索引上，数组中剩下的元素是可以被释放的。垃圾回收器没有办法知道这些，对于垃圾回收器来说，所有的这些在数组中对象引用的元素都是合法的。只有程序员知道不处于数组中活跃位置的元素是不重要的。程序员最有有效的方法去通知垃圾回收器，只有尽可能快的通过收到给那些数组中处于不活跃位置的元素赋null值。

Generally speaking, **whenever a class manages its own memory, the programmer**

**should be alert for memory leaks**. Whenever an element is freed, any object references contained in the element should be nulled out.

总的来说，无论何时一个类自己管理自己的内存，程序员应该警惕内存泄漏。无论何时元素被释放，任何对象的引用保护数组元素应该被赋null。

**Another common source of memory leaks is caches.** Once you put an object reference into a cache, it’s easy to forget that it’s there and leave it in the cache long after it becomes irrelevant. There are several solutions to this problem. If you’re lucky enough to implement a cache for which an entry is relevant exactly so long as there are references to its key outside of the cache, represent the cache as a WeakHashMap; entries will be removed automatically after they become obsolete. Remember that WeakHashMap is useful only if the desired lifetime of cache entries is determined by external references to the key, not the value.

另外引起内存泄漏的公共资源是缓存。一旦你把一个对象引用加入到一个缓存中，那么该对象在缓存中或者当他变成无效之后，仍然长时间留在缓存中这些都非常容易被遗忘。这里有几个方法去解决这个问题。如果你足够幸运为适合的对象去实现了一个缓存，只要在缓存外面去引用它的key，那么就需要使用WeakHashMap去实现缓存。这些对象在他们成为废弃对象后，会自动被移除掉。记住WeakHashMap是非常有用的，仅当缓存中的项的生命周期是由键的外部引用决定的，而不是值

More commonly, the useful lifetime of a cache entry is less well defined, with entries becoming less valuable over time. Under these circumstances, the cache should occasionally be cleansed of entries that have fallen into disuse. This can be done by a background thread (perhaps a ScheduledThreadPoolExecutor) or as a side effect of adding new entries to the cache. The LinkedHashMap class facilitates the latter approach with its removeEldestEntry method. For more sophisticated caches, you may need to use java.lang.ref directly.

**A third common source of memory leaks is listeners and other callbacks.** If you implement an API where clients register callbacks but don’t deregister themexplicitly, they will accumulate unless you take some action. One way to ensurethat callbacks are garbage collected promptly is to store only *weak references* tothem, for instance, by storing them only as keys in a WeakHashMap.

Because memory leaks typically do not manifest themselves as obviousfailures, they may remain present in a system for years. They are typicallydiscovered only as a result of careful code inspection or with the aid of adebugging tool known as a *heap profiler*. Therefore, it is very desirable to learn toanticipate problems like this before they occur and prevent them from happening.

**Item 8:**

**Item 9:**