1. 考虑使用静态工厂方法替代构造方法

一个类允许客户端获取其实例的传统方式，是提供一个公共构造方法。 其实，还有另一种技术应该成为每个程序员工具箱的一部分。 一个类可以提供一个简单的、只返回该类实例的公共静态工厂方法。 下面是一个 `Boolean` 简单的例子（基本类型 `boolean` 的包装类）。 此方法将基本类型 `boolean` 转换为 `Boolean` 对象引用：

```
public static Boolean valueOf(boolean b) {
    return b ? Boolean.TRUE : Boolean.FALSE;
}
```

注意，静态工厂方法与设计模式中的工厂方法模式不同[Gamma95]。本条目中描述的静态工厂方法在设计模式中没有直接的等价。

类可以为其客户端提供静态工厂方法，而不是公共构造方法。提供静态工厂方法来代替提供公共构造方法这种方式，有优点也有缺点。

静态工厂方法的一个优点是，与构造方法不同，它们是有名字的。 如果构造方法的参数本身并不描述被返回的对象，则具有精心选择名称的静态工厂更易于使用，并且生成的客户端代码更易于阅读。 例如，返回一个可能为素数的 `BigInteger` 的构造方法 `BigInteger(int, int, Random)` 可以更好地表示为名为 `BigInteger.probablePrime` 的静态工厂方法。 （这个方法是在 Java 1.4 中添加的。）

一个类只能有一个给定签名的构造方法。 程序员知道通过提供两个仅仅在参数类型的顺序不同的构造方法，来解决这个限制。 这是一个非常糟糕的主意。 对于这样的 API，用户将永远不会记得哪个构造方法是哪个，最终会错误地调用。 阅读使用这些构造方法的代码的人只有在参考类文档的情况下才知道代码的作用。

因为静态工厂方法有名字，所以它们不会受到上述限制。在类中似乎需要具有相同签名的多个构造方法的情况下，用静态工厂方法替换构造方法，并仔细选择名称来突出它们的差异。

静态工厂方法的第二个优点是，与构造方法不同，它们不需要每次调用时都创建一个新对象。 这允许不可变类 （详见第 17 条）使用预先构建的实例，或者在构造时缓存实例，并反复分配它们以避免创建不必要的重复对象。 `Boolean.valueof(boolean)` 方法说明了这种方法：它从不创建对象。这种技术类似于 `Flyweight` 模式[Gamma95]。如果经常请求等价对象，那么它可以极大地提高性能，特别是在创建它们的代价非常昂贵的情况下。

静态工厂方法在重复调用时，返回相同实例这个特点，可以让类在任何时候都能对实例保持严格的控制。这样做的类被称为实例控制类（instance-controlled）。有很多理由足以让我们去我们编写实例控制类。实例控制可以保证一个类是单例 的（详见第 3 条） 或不可实例化的 （详见第 4 条）。同时,它允许一个不可变的值类 （详见第 17 条） 保证不存在两个相等但不相同的实例，也就是说当且仅当 `a == b` 时才有 `a.equals(b)` 。这是 `Flyweight` 模式的基础[Gamma95]。 `Enum` 类型 （详见第 34 条） 可以做到这点。

静态工厂方法的第三个优点是，与构造方法不同，它们可以返回其返回类型的任何子类型的对象。 这为你在选择返回对象的类型时，提供了很大的灵活性。

这种灵活性的一个应用是， API 可以返回对象而不需要公开它的类。 以这种方式隐藏实现类，会使 API 非常紧凑。 这种技术适用于基于接口的框架（详见第 20 条），其中接口为静态工厂方法提供自然返回类型。

在 Java 8 之前，接口不能有静态方法。根据约定，一个名为 `Types` 的接口的静态工厂方法被放入一个不可实例化的伙伴类（companion class）（详见第 4 条） `Types` 类中。例如，Java 集合框架有 45 个接口的实用工具实现，提供不可修改的集合、同步集合等等。几乎所有这些实现都是通过静态工厂方法在一个不可实例化的类（ `java.util.collections` ）中返回的。返回对象的类都是隐藏的。

`Collections` 框架 API 的规模要比它之前返回的 45 个单独的公共类要小得多，每个类在集合框架中都有一个便利的实现。不仅精简了API，还减少了程序员为了使用 API而掌握的概念的数量和难度。程序员知道返回的对象恰好有其接口指定的 API，因此不需要为实现类读阅读额外的类文档。此外，使用这种静态工厂方法，需要客户端通过接口替代实现类，来引用返回的对象，这通常是良好的实践（详见第 64 条）。

从 Java 8 开始，接口不能包含静态方法的限制被取消了，所以通常没有理由为接口提供一个不可实例化的伴随类。 很多公开的静态成员应该放在这个接口本身。 但是，请注意，将这些静态方法的大部分实现代码放在单独的包私有类中仍然是必要的。 这是因为 Java 8 要求所有接口的静态成员都是公共的。 Java 9 允许私有静态方法，但静态字段和静态成员类仍然需要公开。

静态工厂的第四个优点是返回对象的类可以根据输入参数的不同而不同。 声明的返回类型的任何子类都是允许的。 返回对象的类也可以随每次发布而不同。

`EnumSet` 类（详见第 36 条）没有公共构造方法，只有静态工厂。 在 OpenJDK 实现中，它们根据底层枚举类型的大小返回两个子类中的一个的实例：大多数枚举类型具有 64 个或更少的元素，静态工厂将返回一个 `RegularEnumSet` 实例， 底层是 `long` 类型；如果枚举类型具有六十五个或更多元素，则工厂将返回一个 `JumboEnumSet` 实例，底层是 `long` 类型的数组。

这两个实现类的存在对于客户端而言是不可见的。 如果 `RegularEnumSet` 对于小的枚举类型不再具有性能优势，则可以在未来版本中将其淘汰，且不会产生任何不良影响。 同样，如果可以证明添加 `EnumSet` 的更多的实现可以提高性能，那么在未来的版本可能就会这样做。 客户既不知道也不关心他们从工厂返回的对象的类别; 他们只需要知道它是 `EnumSet` 的子类。

静态工厂的第五个优点是，在编写包含该方法的类时，返回的对象的类不需要存在。 这种灵活的静态工厂方法构成了服务提供者框架的基础，比如 Java 数据库连接 API（JDBC）。服务提供者框架是提供者实现服务的系统，并且系统使得实现对客户端可用，从而将客户端从实现中分离出来。

服务提供者框架中有三个基本组：服务接口，它表示实现；提供者注册 API，提供者用来注册实现；以及服务访问 API，客户端使用该 API 获取服务的实例。服务访问 API 允许客户指定选择实现的标准。在缺少这样的标准的情况下，API 返回一个默认实现的实例，或者允许客户通过所有可用的实现进行遍历。服务访问 API 是灵活的静态工厂，它构成了服务提供者框架的基础。

服务提供者框架的一个可选的第四个组件是一个服务提供者接口，它描述了一个生成服务接口实例的工厂对象。在没有服务提供者接口的情况下，必须对实现进行反射实例化（详见第 65 条）。在 JDBC 的情况下， `Connection` 扮演服务接口的一部分， `DriverManager.registerDriver` 提供程序注册 API、 `DriverManager.getConnection` 是服务访问 API， `Driver` 是服务提供者接口。

服务提供者框架模式有许多变种。 例如，服务访问 API 可以向客户端返回比提供者提供的更丰富的服务接口。 这是桥接模式[Gamma95]。 依赖注入框架（详见第 5 条）可以被看作是强大的服务提供者。 从 Java 6 开始，平台包含一个通用的服务提供者框架 `java.util.ServiceLoader`，所以你不需要，一般也不应该自己编写（详见第 59 条）。 JDBC 不使用 `ServiceLoader`，因为前者早于后者。

只提供静态工厂方法的主要限制是，没有公共或受保护构造方法的类不能被子类化。 例如，要想将 `Collections` 框架中任何遍历的实现类进行子类化，是不可能的。但是这样也会因祸得福，因为它鼓励程序员使用组合（composition）而不是继承（详见第 18 条），并且是不可变类型锁需要的（详见第 17 条）。

静态工厂方法的第二个缺点是，程序员很难找到它们。 它们不像构造方法那样在 API 文档中明确的标注出来。因此，对于提供了静态方法而不是构造器的类来说，想要查明如何实例化一个类是十分困难的。Javadoc 工具可能有一天会注意到静态工厂方法。与此同时，通过关注类或者接口的文档中静态方法，并且遵守标准的命名习惯，也可以弥补这一劣势。下面是一些静态工厂方法的常用名称。以下清单这是列出了其中的一小部分：

- from —— 类型转换方法，它接受单个参数并返回此类型的相应实例，例如：**Date d = Date.from(instant)**;
- of —— 聚合方法，接受多个参数并返回该类型的实例，并把他们合并在一起，例如：**Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING)**;
- valueOf —— from 和 to 更为详细的替代 方式，例如：**BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE)**;
- instance 或 getinstance —— 返回一个由其参数 (如果有的话) 描述的实例，但不能说它具有相同的值，例如：**StackWalker luke = StackWalker.getInstance(options)**;
- create 或 newInstance —— 与 instance 或 getInstance 类似，除此之外该方法保证每次调用返回一个新的实例，例如：**Object newArray = Array.newInstance(classObject, arrayLen)**;
- getType —— 与 getInstance 类似，但是在工厂方法处于不同的类中的时候使用。**getType** 中的 **Type** 是工厂方法返回的对象类型，例如：**FileStore fs = Files.getFileStore(path)**;
- newType —— 与 newInstance 类似，但是在工厂方法处于不同的类中的时候使用。**newType**中的 **Type** 是工厂方法返回的对象类型，例如：**BufferedReader br = Files.newBufferedReader(path)**;
- type —— getType 和 newType 简洁的替代方式，例如：**List<Complaint> litany = Collections.list(legacyLitany)**;

总之，静态工厂方法和公共构造方法都有它们的用途，并且了解它们的相对优点是值得的。通常，静态工厂更可取，因此避免在没有考虑静态工厂的情况下，直接选择使用或提供公共构造方法。

2. 当构造方法参数过多时使用 builder 模式

静态工厂和构造方法都有一个限制：它们在可选参数很多的情景下，无法很好得扩展。请考虑一个代表包装食品上的营养成分标签的例子。这些标签有几个必需的属性——每次建议的摄入量，每罐的份量和每份卡路里 ，以及超过 20 个可选的属性——总脂肪、饱和脂肪、反式脂肪、胆固醇、钠等等。大多数产品只包含这些可选字段中的少数，且具有非零值（大部分字段为空）。

应该为这样的类编写什么样的构造方法或静态工厂？传统上，程序员使用了可伸缩（telescoping constructor）构造方法模式。在这种模式中，首先提供一个只有必需参数的构造方法，接着提供增加了一个可选参数的构造函数，然后提供增加了两个可选参数的构造函数，等等；最终，在构造函数中包含所有必需和可选参数。以下就是它在实践中的样子。为了简便，只显示了四个可选属性：

```
// Telescoping constructor pattern - does not scale well!
public class NutritionFacts {
    private final int servingSize;  // (mL)             required
    private final int servings;     // (per container) required
    private final int calories;     // (per serving)   optional
    private final int fat;          // (g/serving)     optional
    private final int sodium;       // (mg/serving)    optional
    private final int carbohydrate; // (g/serving)     optional

    public NutritionFacts(int servingSize, int servings) {
        this(servingSize, servings, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories) {
        this(servingSize, servings, calories, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories, int fat) {
        this(servingSize, servings, calories, fat, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories, int fat, int sodium) {
        this(servingSize, servings, calories, fat, sodium, 0);
    }

    public NutritionFacts(int servingSize, int servings,
           int calories, int fat, int sodium, int carbohydrate) {
        this.servingSize  = servingSize;
        this.servings     = servings;
        this.calories     = calories;
        this.fat          = fat;
        this.sodium       = sodium;
        this.carbohydrate = carbohydrate;
    }
}
```

当想要创建一个实例时，可以使用包含所有你要设置的参数的构造方法：

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

通常情况下，这个构造方法的调用需要许多你不想设置的参数，但是你不得不为它们传递一个值。在这种情况下，我们为 `fat` 属性传递了 `0` 。「只有」六个参数可能看起来并不那么糟糕，但随着参数数量的增加，它很快就会失控。

简而言之，**可伸缩构造方法模式是有效的，但是当有很多参数时，很难编写客户端代码，而且很难读懂它。** 读者不知道这些值是什么意思，并且必须仔细地去数参数才能找到答案。一长串相同类型的参数可能会导致一些 bug。如果客户端不小心写反了两个这样的参数，编译器并不会报错，但是程序在运行时会出现与预期不一致的行为 （详见第 51 条）。

当在构造方法中遇到许多可选参数时，另一种选择是 JavaBeans 模式，在这种模式中，调用一个无参的构造方法来创建对象，然后调用 `setter` 方法来设置每个必需的参数和可选参数：

```
// 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; }
}
```

这种模式没有伸缩构造方法模式的缺点。有点冗长，但创建实例很容易，并且易于阅读所生成的代码:

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

不幸的是，JavaBeans 模式本身有严重的缺陷。**由于构造方法被分割成了多次调用，所以在构造过程中 JavaBean 可能处于不一致的状态。** 该类仅通过检查构造函数参数的有效性，而没有强制的一致性措施。在不一致的状态下尝试使用对象可能会导致一些错误，这些错误与平常代码的 BUG 很是不同，因此很难调试。一个相关的缺点是，**JavaBeans 模式排除了让类不可变的可能性**（详见第 17 条），并且需要程序员增加工作以确保线程安全。

通过在对象构建完成时手动「冻结」对象，并且不允许它在解冻之前使用，可以减少这些缺点，但是这种变体在实践中很难使用并且很少使用。 而且，在运行时会导致错误，因为编译器无法确保程序员会在使用对象之前调用 `freeze` 方法。

幸运的是，还有第三种选择。它结合了可伸缩构造方法模式的安全性和 JavaBean 模式的可读性。它是 Builder 模式[Gamma95] 的一种形式。客户端不直接构造所需的对象，而是调用一个包含所有必需参数的构造方法 (或静态工厂) 得到获得一个 builder 对象。然后，客户端调用 builder 对象的与 `setter` 相似的方法来设置你想设置的可选参数。最后，客户端调用 builder 对象的一个无参的 `build` 方法来生成对象，该对象通常是不可变的。Builder 通常是它所构建的类的一个静态成员类（详见第 24 条）。以下是它在实践中的示例：

```
// 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;
    }
}
```

`NutritionFacts` 类是不可变的，所有的参数默认值都在一个地方。builder 的 setter 方法返回 builder 本身，这样就可以进行链式调用，从而生成一个流畅的 API。下面是客户端代码的示例：

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

这个客户端代码很容易编写，更重要的是易于阅读。 采用 Builder 模式模拟实现的的可选参数可以在 Python 和 Scala 都可以找到。

为了简洁起见，省略了有效性检查。 要尽快检测出无效参数，检查 builder 的构造方法和方法中的参数有效性。 在 `build` 方法调用的构造方法中检查包含多个参数的不变性。为了确保这些不变性不受攻击，在从 builder 复制参数后对对象属性进行检查（详见第 50 条）。 如果检查失败，则抛出 `IllegalArgumentException` 异常（详见第 72 条），其详细消息指示哪些参数无效（详见第 75 条）。

Builder 模式非常适合类层次结构。 使用平行层次的 builder，每个 builder 嵌套在相应的类中。 抽象类有抽象的 builder；具体的类有具体的 builder。 例如，考虑代表各种比萨饼的根层次结构的抽象类：

```
// Builder pattern for class hierarchies
import java.util.EnumSet;
import java.util.Objects;
import java.util.Set;

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
    }
}
```

请注意，`Pizza.Builder` 是一个带有递归类型参数（recursive type parameter）（详见第 30 条）的泛型类型。 这与抽象的 `self` 方法一起，允许方法链在子类中正常工作，而不需要强制转换。 Java 缺乏自我类型的这种变通解决方法被称为模拟自我类型（simulated self-type）。

这里有两个具体的 `Pizza` 的子类，其中一个代表标准的纽约风格的披萨，另一个是半圆形烤乳酪馅饼。前者有一个所需的尺寸参数，而后者则允许指定酱汁是否应该在里面或在外面：

```
import java.util.Objects;

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;
    }
}
```

请注意，每个子类 builder 中的 `build` 方法被声明为返回正确的子类：`NyPizza.Builder` 的 `build` 方法返回 `NyPizza`，而 `Calzone.Builder` 中的 `build` 方法返回 `Calzone`。这种技术，其一个子类的方法被声明为返回在超类中声明的返回类型的子类型，称为协变返回类型（covariant return typing）。它允许客户端使用这些 builder，而不需要强制转换。

这些「分层 builder（hierarchical builders）」的客户端代码基本上与简单的 `NutritionFacts` builder 的代码相同。为了简洁起见，下面显示的示例客户端代码假设枚举常量的静态导入：

```
NyPizza pizza = new NyPizza.Builder(SMALL)
        .addTopping(SAUSAGE).addTopping(ONION).build();
Calzone calzone = new Calzone.Builder()
        .addTopping(HAM).sauceInside().build();
```

builder 对构造方法的一个微小的优势是，builder 可以有多个可变参数，因为每个参数都是在它自己的方法中指定的。或者，builder 可以将传递给多个调用的参数聚合到单个属性中，如前面的 `addTopping` 方法所演示的那样。

Builder 模式非常灵活。 单个 builder 可以重复使用来构建多个对象。 builder 的参数可以在构建方法的调用之间进行调整，以改变创建的对象。 builder 可以在创建对象时自动填充一些属性，例如每次创建对象时增加的序列号。

Builder 模式也有缺点。为了创建对象，首先必须创建它的 builder。虽然创建这个 builder 的成本在实践中不太可能被注意到，但在看中性能的场合下这可能就是一个问题。而且，builder 模式比伸缩构造方法模式更冗长，因此只有在有足够的参数时才值得使用它，比如四个或更多。但是请记住，你可能在以后会想要添加更多的参数。但是，如果你一开始是使用的构造方法或静态工厂，当类演化到参数数量失控的时候再转到 Builder 模式，过时的构造方法或静态工厂就会面临尴尬的处境。因此，通常最好从一开始就创建一个 builder。

总而言之，当设计类的构造方法或静态工厂的参数超过几个时，Builder 模式是一个不错的选择，特别是许多参数是可选的或相同类型的。builder 模式客户端代码比使用伸缩构造方法（telescoping constructors）更容易读写，并且 builder 模式比 JavaBeans 更安全。

3. 使用私有构造方法或枚类实现 Singleton 属性

单例是一个仅实例化一次的类[Gamma95]。单例对象通常表示无状态对象，如函数 (详见第 24 条) 或一个本质上唯一的系统组件。让一个类成为单例会使测试它的客户变得困难，因为除非实现一个作为它类型的接口，否则不可能用一个模拟实现替代单例。

有两种常见的方法来实现单例。两者都基于保持构造方法私有和导出公共静态成员以提供对唯一实例的访问。在第一种方法中，成员是 `final` 修饰的属性：

```
// Singleton with public final field
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }
    public void leaveTheBuilding() { ... }
}
```

私有构造方法只调用一次，来初始化公共静态 final `Elvis.INSTANCE` 属性。缺少一个公共的或受保护的构造方法，保证了全局的唯一性：一旦 Elvis 类被初始化，一个 Elvis 的实例就会存在——不多也不少。客户端所做的任何事情都不能改变这一点，但需要注意的是：特权客户端可以使用 `AccessibleObject.setAccessible` 方法，以反射方式调用私有构造方法（详见第 65 条）。如果需要防御此攻击，请修改构造函数，使其在请求创建第二个实例时抛出异常。

在第二个实现单例的方法中，公共成员是一个静态的工厂方法：

```
// 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() { ... }
}
```

所有对 `Elvis.getInstance` 的调用都返回相同的对象引用，并且不会创建其他的 Elvis 实例（与前面提到的警告相同）。

公共属性方法的主要优点是 API 明确表示该类是一个单例：公共静态属性是 final 的，所以它总是包含相同的对象引用。 第二个好处是它更简单。

静态工厂方法的优势之一在于，它提供了灵活性：在不改变其 API 的前提下，我们可以改变该类是否应该为单例的想法。工厂方法返回该类的唯一实例，但是，它很容易被修改，比如，改为每个调用该方法的线程返回一个唯一的实例。 第二个好处是，如果你的应用程序需要它，可以编写一个泛型单例工厂（generic singleton factory ）（详见第30 条）。 使用静态工厂的最后一个优点是，可以通过方法引用（method reference）作为提供者，例如 `Elvis::instance` 等同于 `Supplier<Elvis>`。 除非满足以上任意一种优势，否则还是优先考虑公有域（public-field）的方法。

为了将上述方法中实现的单例类变成是可序列化的 （第 12 章），仅仅将 `implements Serializable` 添加到声明中是不够的。为了保证单例模式不被破坏，必须声明所有的实例字段为 `transient`，并提供一个 `readResolve` 方法（详见第 89 条）。否则，每当序列化的实例被反序列化时，就会创建一个新的实例，在我们的例子中，导致出现新的 Elvis 实例。为了防止这种情况发生，将如下的 `readResolve` 方法添加到 Elvis 类：

```
// 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;
}
```

实现一个单例的第三种方法是声明单一元素的枚举类：

```
// Enum singleton - the preferred approach
public enum Elvis {
    INSTANCE;

    public void leaveTheBuilding() { ... }
}
```

这种方式类似于公共属性方法，但更简洁，无偿地提供了序列化机制，并提供了防止多个实例化的坚固保证，即使是在复杂的序列化或反射攻击的情况下。这种方法可能感觉有点不自然，但是 **单一元素枚举类通常是实现单例的最佳方式。**注意，如果单例必须继承 `Enum` 以外的父类（尽管可以声明一个 `Enum` 来实现接口），那么就不能使用这种方法。

4. 使用私有构造器执行非实例化

偶尔你会想写一个只包含静态方法和静态字段的类。 这些类的名声非常不好，因为有些人滥用这些类从而避免以面向对象方式思考从而编写过程化的程序，但是它们确实有着特殊的用途。 它们可以用来按照 `java.lang.Math` 或 `java.util.Arrays` 的方式，把基本类型的值或数组类型上的相关方法组织起来。我们也可以通过 `java.util.Collections` 的方式，把实现特定接口上面的静态方法进行分组，也包括工厂方法（详见第 1 条）。 （从 Java 8 开始，你也可以将这些方法放在接口中，假定该接口是你编写的并可以进行修改。）最后，这样的类可以用于在 final 类上对方法进行分组，因为不能将它们放在子类中。

这样的工具类（utility classes）不是设计用来被实例化的，因为实例化对它没有任何意义。然而，在没有显式构造器的情况下，编译器提供了一个公共的、无参的默认构造器。对于用户来说，该构造器与其他构造器没有什么区别。在已发布的 API 中经常看到无意识的被实例的类。

试图通过创建抽象类来强制执行非实例化是行不通的。 该类可以被子类化，并且子类可以被实例化。此外，它误导用户认为该类是为继承而设计的（详见第 19 条）。不过，有一个简单的方法来确保非实例化。只有当类不包含显式构造器时，才会生成一个默认构造器，**因此可以通过包含一个私有构造器来实现类的非实例化：**

```
// Noninstantiable utility class
public class UtilityClass {
    // Suppress default constructor for noninstantiability
    private UtilityClass() {
        throw new AssertionError();
    }
    ... // Remainder omitted
}
```

因为显式构造器是私有的，所以不可以在类的外部访问它。 `AssertionError` 异常不是严格要求的，但是它可以避免不小心在类的内部调用构造器。它保证类在任何情况下都不会被实例化。这个习惯用法有点违反直觉，好像构造器就是设计成不能调用的一样。因此，如前面所示，添加注释是种明智的做法。

这种习惯有一个副作用，就是使得一个类不能子类化。所有的构造器都必须显式或隐式地调用父类构造器，而在这群情况下子类则没有可访问的父类构造器来调用。

05. 依赖注入优于硬连接资源（hardwiring resources）

许多类依赖于一个或多个底层资源。例如，拼写检查器依赖于字典。将此类类实现为静态工具类并不少见 （详见第 4 条）：

```
// 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) { ... }
}
```

同样地，将它们实现为单例的做法也并不少见（详见第 3 条）：

```
// 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) { ... }
}
```

这两种方法都不令人满意，因为他们都是假设只有一本字典可用。实际上，每种语言都有自己的字典，特殊的字典被用于特殊的词汇表。另外，可能还需要用特殊的词典进行测试。想当然地认为一本字典就足够了，这是一厢情愿的想法。

可以通过使 `dictionary` 属性设置为非 `final` ，并添加一个方法来更改现有拼写检查器中的字典，从而让 `SpellChecker` 支持多个字典，但是这样的设置显得非常笨拙、容易出错、并且无法并行工作。**静态工具类和单例类不适合与需要引用底层资源的类。**

这里所需要的是能够支持类的多个实例 （在我们的示例中是指 `SpellChecker` ），每个实例都使用客户端所期望的资源（在我们的例子中是 `dictionary` ）。满足这一需求的简单模式是，**在创建新实例时将资源传递到构造器中。** 这是依赖项注入（dependency injection）的一种形式：字典是拼写检查器的一个依赖项，当它创建时被注入到拼写检查器中。

```
// 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) { ... }
}
```

依赖注入模式非常简单，许多程序员使用它多年而不知道它有一个名字。 虽然我们的拼写检查器的例子只有一个资源（字典），但是依赖项注入可以使用任意数量的资源和任意依赖图。 它保持了不变性（详见第 17 条），因此多个客户端可以共享依赖对象（假设客户需要相同的底层资源）。 依赖注入同样适用于构造器、静态工厂（详见第 1 条）和 builder 模式（详见第 2 条）。

该模式的一个有用的变体是将资源工厂传递给构造器。 工厂是可以重复调用以创建类型实例的对象。 这种工厂体现了工厂方法模式（Factory Method pattern）[Gamma95]。 Java 8 中引入的 `Supplier<T>` 接口非常适合代表工厂。 在输入上采用 `Supplier<T>` 的方法通常应该使用有界的通配符类型（bounded wildcard type）（详见第 31 条）约束工厂的类型参数，以便客户端能够传入一个工厂，来创建指定类型的任意子类型。例如，下面是一个生产马赛克的方法，它利用客户端提供的工厂来生产每一片马赛克：

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

尽管依赖注入极大地提高了灵活性和可测试性，但它可能使大型项目变得混乱，这些项目通常包含数千个依赖项。使用依赖注入框架（如 Dagger [Dagger]、Guice [Guice] 或 Spring [Spring]）可以消除这些混乱。这些框架的使用超出了本书的范围，但是请注意，为手动依赖注入而设计的 API 非常适合这些框架的使用。

总而言之，不要用单例和静态工具类来实现依赖一个或多个底层资源的类，且该资源的行为会影响到该类的行为；也不要直接用这个类来创建这些资源。而应该将这些资源或者工厂传给构造器（或者静态工厂，或者构建器），通过它们来创建类。这个实践就被称作依赖注人，它极大地提升了类的灵活性、可重用性和可测试性。

6. 避免创建不必要的对象

在每次需要时重用一个对象而不是创建一个新的相同功能对象通常是恰当的。重用可以更快更流行。如果对象是不可变的（详见第 17 条），它总是可以被重用。

作为一个不应该这样做的极端例子，请考虑以下语句：

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

语句每次执行时都会创建一个新的 String 实例，而这些对象的创建都不是必需的。String 构造方法 `("bikini")` 的参数本身就是一个 `bikini` 实例，它与构造方法创建的所有对象的功能相同。如果这种用法发生在循环中，或者在频繁调用的方法中，就会毫无必要地创建数百万个 String 实例。

改进后的版本如下：

```
String s = "bikini";
```

该版本使用单个 String 实例，而不是每次执行时创建一个新实例。此外，它可以保证对象运行在同一虚拟机上的任何其他代码重用，而这些代码恰好包含相同的字符串字面量[JLS, 3.10.5]。

通过使用静态工厂方法（详见第 1 条）和构造器，可以避免创建不需要的对象。例如，工厂方法 `Boolean.valueOf(String)` 比构造方法 `Boolean(String)` 更可取，后者在 Java 9 中被弃用。构造方法每次调用时都必须创建一个新对象，而工厂方法永远不需要这样做，在实践中也不需要。除了重用不可变对象，如果知道它们不会被修改，还可以重用可变对象。

一些对象的创建比其他对象的创建要昂贵得多。 如果要重复使用这样一个「昂贵的对象」，建议将其缓存起来以便重复使用。 不幸的是，当创建这样一个对象时并不总是很直观明显的。 假设你想写一个方法来确定一个字符串是否是一个有效的罗马数字。 以下是使用正则表达式完成此操作时最简单方法：

```
// 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})$");
}
```

这个实现的问题在于它依赖于 `String.matches` 方法。 虽然 `String.matches` 是检查字符串是否与正则表达式匹配的最简单方法，但它不适合在性能临界的情况下重复使用。 问题是它在内部为正则表达式创建一个 `Pattern` 实例，并且只使用它一次，之后它就有资格进行垃圾收集。 创建 `Pattern` 实例是昂贵的，因为它需要将正则表达式编译成有限状态机（finite state machine）。

为了提高性能，作为类初始化的一部分，将正则表达式显式编译为一个 `Pattern` 实例（不可变），缓存它，并在 `isRomanNumeral` 方法的每个调用中重复使用相同的实例：

```
// 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();
    }
}
```

如果经常调用，`isRomanNumeral` 的改进版本的性能会显著提升。 在我的机器上，原始版本在输入 8 个字符的字符串上需要 1.1 微秒，而改进的版本则需要 0.17 微秒，速度提高了 6.5 倍。 性能上不仅有所改善，而且更明确清晰了。 为不可见的 Pattern 实例创建静态 final 修饰的属性，并允许给它一个名字，这个名字比正则表达式本身更具可读性。

如果包含 `isRomanNumeral` 方法的改进版本的类被初始化，但该方法从未被调用，则 ROMAN 属性则没必要初始化。 在第一次调用 `isRomanNumeral` 方法时，可以通过延迟初始化（ lazily initializing）属性（详见第 83 条）来排除初始化，但一般不建议这样做。 延迟初始化常常会导致实现复杂化，而性能没有可衡量的改进（详见第 67 条）。

当一个对象是不可变的时，很明显它可以被安全地重用，但是在其他情况下，它远没有那么明显，甚至是违反直觉的。考虑适配器（adapters）的情况[Gamma95]，也称为视图（views）。一个适配器是一个对象，它委托一个支持对象（backing object），提供一个可替代的接口。由于适配器没有超出其支持对象的状态，因此不需要为给定对象创建多个给定适配器的实例。

例如，Map 接口的 `keySet` 方法返回 Map 对象的 Set 视图，包含 Map 中的所有 key。 天真地说，似乎每次调用 `keySet` 都必须创建一个新的 Set 实例，但是对给定 Map 对象的 `keySet` 的每次调用都返回相同的 Set 实例。 尽管返回的 Set 实例通常是可变的，但是所有返回的对象在功能上都是相同的：当其中一个返回的对象发生变化时，所有其他对象也都变化，因为它们全部由相同的 Map 实例支持。 虽然创建 `keySet` 视图对象的多个实例基本上是无害的，但这是没有必要的，也没有任何好处。

另一种创建不必要的对象的方法是自动装箱（auto boxing），它允许程序员混用基本类型和包装的基本类型，根据需要自动装箱和拆箱。 自动装箱模糊不清，但不会消除基本类型和装箱基本类型之间的区别。 有微妙的语义区别和不那么细微的性能差异（详见第 61 条）。 考虑下面的方法，它计算所有正整数的总和。 要做到这一点，程序必须使用 `long` 类型，因为 `int` 类型不足以保存所有正整数的总和：

```
// 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;
}
```

这个程序的结果是正确的，但由于写错了一个字符，运行的结果要比实际慢很多。变量 `sum` 被声明成了 `Long` 而不是 `long`，这意味着程序构造了大约 2^{31} 不必要的 `Long` 实例（大约每次往 `Long` 类型的 `sum` 变量中增加一个 `long` 类型构造的实例），把 `sum` 变量的类型由 `Long` 改为 `long`，在我的机器上运行时间从 6.3 秒降低到 0.59 秒。这个教训很明显：**优先使用基本类型而不是装箱的基本类型，也要注意无意识的自动装箱。**

这个条目不应该被误解为暗示对象创建是昂贵的，应该避免创建对象。 相反，使用构造方法创建和回收小的对象是非常廉价，构造方法只会做很少的显示工作，尤其是在现代 JVM 实现上。 创建额外的对象以增强程序的清晰度，简单性或功能性通常是件好事。

相反，除非池中的对象非常重量级，否则通过维护自己的对象池来避免对象创建是一个坏主意。对象池的典型例子就是数据库连接。建立连接的成本非常高，因此重用这些对象是有意义的。但是，一般来说，维护自己的对象池会使代码混乱，增加内存占用，并损害性能。现代 JVM 实现具有高度优化的垃圾收集器，它们在轻量级对象上轻松胜过此类对象池。

这个条目的对应点是针对条目 50 的防御性复制（defensive copying）。 目前的条目说：「当你应该重用一个现有的对象时，不要创建一个新的对象」，而条目 50 说：「不要重复使用现有的对象，当你应该创建一个新的对象时。」请注意，重用防御性复制所要求的对象所付出的代价，要远远大于不必要地创建重复的对象。 未能在需要的情况下防御性复制会导致潜在的错误和安全漏洞；而不必要地创建对象只会影响程序的风格和性能。

7. 消除过期的对象引用

如果你从使用手动内存管理的语言（如 C 或 C++）切换到像 Java 这样的带有垃圾收集机制的语言，那么作为程序员的工作就会变得容易多了，因为你的对象在使用完毕以后就自动回收了。当你第一次体验它的时候，它就像魔法一样。这很容易让人觉得你不需要考虑内存管理，但这并不完全正确。

考虑以下简单的栈实现：

```
// Can you spot the "memory leak"?
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);
        }
    }
```

这个程序没有什么明显的错误（但是对于泛型版本，请参阅条目 29）。 你可以对它进行详尽的测试，它都会成功地通过每一项测试，但有一个潜在的问题。 笼统地说，程序有一个「内存泄漏」，由于垃圾回收器的活动的增加，或内存占用的增加，静默地表现为性能下降。 在极端的情况下，这样的内存泄漏可能会导致磁盘分页（disk paging），甚至导致内存溢出（OutOfMemoryError）的失败，但是这样的故障相对较少。

那么哪里发生了内存泄漏？ 如果一个栈增长后收缩，那么从栈弹出的对象不会被垃圾收集，即使使用栈的程序不再引用这些对象。 这是因为栈维护对这些对象的过期引用（obsolete references）。 过期引用简单来说就是永远不会解除的引用。 在这种情况下，元素数组「活动部分（active portion）」之外的任何引用都是过期的。 活动部分是由索引下标小于 size 的元素组成。

垃圾收集语言中的内存泄漏（更适当地称为无意的对象保留 unintentional object retentions）是隐蔽的。 如果无意中保留了对象引用，那么不仅这个对象排除在垃圾回收之外，而且该对象引用的任何对象也是如此。 即使只有少数对象引用被无意地保留下来，也可以阻止垃圾回收机制对许多对象的回收，这对性能产生很大的影响。

这类问题的解决方法很简单：一旦对象引用过期，将它们设置为 null。 在我们的 `Stack` 类的情景下，只要从栈中弹出，元素的引用就设置为过期。 `pop` 方法的修正版本如下所示：

```
public Object pop() {
    if (size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

取消过期引用的另一个好处是，如果它们随后被错误地引用，程序立即抛出 `NullPointerException` 异常，而不是悄悄地做继续做错误的事情。尽可能快地发现程序中的错误是有好处的。

当程序员第一次被这个问题困扰时，他们可能会在程序结束后立即清空所有对象引用。这既不是必要的，也不是可取的；它不必要地搞乱了程序。**清空对象引用应该是例外而不是规范。**消除过期引用的最好方法是让包含引用的变量超出范围。如果在最近的作用域范围内定义每个变量 （详见第 57 条），这种自然就会出现这种情况。

那么什么时候应该清空一个引用呢？ `Stack` 类的哪个方面使它容易受到内存泄漏的影响？简单地说，它管理自己的内存。存储池（storage pool）由 `elements` 数组的元素组成（对象引用单元，而不是对象本身）。数组中活动部分的元素 (如前面定义的) 被分配，其余的元素都是空闲的。垃圾收集器没有办法知道这些；对于垃圾收集器来说， `elements` 数组中的所有对象引用都同样有效。只有程序员知道数组的非活动部分不重要。程序员可以向垃圾收集器传达这样一个事实，一旦数组中的元素变成非活动的一部分，就可以手动清空这些元素的引用。

一般来说，**当一个类自己管理内存时，程序员应该警惕内存泄漏问题。** 每当一个元素被释放时，元素中包含的任何对象引用都应该被清除。

另一个常见的内存泄漏来源是缓存。 一旦将对象引用放入缓存中，很容易忘记它的存在，并且在它变得无关紧要之后，仍然保留在缓存中。对于这个问题有几种解决方案。如果你正好想实现了一个缓存：只要在缓存之外存在对某个项（entry）的键（key）引用，那么这项就是明确有关联的，就可以用 `WeakHashMap` 来表示缓存；这些项在过期之后自动删除。记住，只有当缓存中某个项的生命周期是由外部引用到键（key）而不是值（value）决定时， `WeakHashMap` 才有用。

更常见的情况是，缓存项有用的生命周期不太明确，随着时间的推移一些项变得越来越没有价值。在这种情况下，缓存应该偶尔清理掉已经废弃的项。这可以通过一个后台线程 (也许是 `ScheduledThreadPoolExecutor`) 或将新的项添加到缓存时顺便清理。 `LinkedHashMap` 类使用它的 `removeEldestEntry` 方法实现了后一种方案。对于更复杂的缓存，可能直接需要使用 `java.lang.ref` 。

第三个常见的内存泄漏来源是监听器和其他回调。如果你实现了一个 API，其客户端注册回调，但是没有显式地撤销注册回调，除非采取一些操作，否则它们将会累积。确保回调是垃圾收集的一种方法是只存储弱引用（weak references），例如，仅将它们保存在 `WeakHashMap` 的键（key）中。

因为内存泄漏通常不会表现为明显的故障，所以它们可能会在系统中保持多年。 通常仅在仔细的代码检查或借助堆分析器（heap profiler）的调试工具才会被发现。 因此，学习如何预见这些问题，并防止这些问题发生，是非常值得的。

8. 避免使用 Finalizer 和 Cleaner 机制

Finalizer 机制是不可预知的，往往是危险的，而且通常是不必要的。 它们的使用会导致不稳定的行为，糟糕的性能和移植性问题。 Finalizer 机制有一些特殊的用途，我们稍后会在这个条目中介绍，但是通常应该避免它们。 从 Java 9 开始，Finalizer 机制已被弃用，但仍被 Java 类库所使用。 Java 9 中 Cleaner 机制代替了 Finalizer 机制。 Cleaner 机制不如 Finalizer 机制那样危险，但仍然是不可预测，运行缓慢并且通常是不必要的。

提醒 C++程序员不要把 Java 中的 Finalizer 或 Cleaner 机制当成的 C++ 析构函数的等价物。 在 C++ 中，析构函数是回收对象相关资源的正常方式，是与构造方法相对应的。 在 Java 中，当一个对象变得不可达时，垃圾收集器回收与对象相关联的存储空间，不需要开发人员做额外的工作。 C++ 析构函数也被用来回收其他非内存资源。 在 Java 中，try-with-resources 或 try-finally 块用于此目的（详见第 9 条）。

Finalizer 和 Cleaner 机制的一个缺点是不能保证他们能够及时执行[JLS，12.6]。 在一个对象变得无法访问时，到 Finalizer 和 Cleaner 机制开始运行时，这期间的时间是任意长的。 这意味着你永远不应该 Finalizer 和 Cleaner 机制做任何时间敏感（time-critical）的事情。 例如，依赖于 Finalizer 和 Cleaner 机制来关闭文件是严重的错误，因为打开的文件描述符是有限的资源。 如果由于系统迟迟没有运行 Finalizer 和 Cleaner 机制而导致许多文件被打开，程序可能会失败，因为它不能再打开文件了。

及时执行 Finalizer 和 Cleaner 机制是垃圾收集算法的一个功能，这种算法在不同的实现中有很大的不同。程序的行为依赖于 Finalizer 和 Cleaner 机制的及时执行，其行为也可能大不不同。 这样的程序完全可以在你测试的 JVM 上完美运行，然而在你最重要的客户的机器上可能运行就会失败。

延迟终结（finalization）不只是一个理论问题。为一个类提供一个 Finalizer 机制可以任意拖延它的实例的回收。一位同事调试了一个长时间运行的 GUI 应用程序，这个应用程序正在被一个 OutOfMemoryError 错误神秘地死掉。分析显示，在它死亡的时候，应用程序的 Finalizer 机制队列上有成千上万的图形对象正在等待被终结和回收。不幸的是，Finalizer 机制线程的运行优先级低于其他应用程序线程，所以对象被回收的速度低于进入队列的速度。语言规范并不保证哪个线程执行 Finalizer 机制，因此除了避免使用 Finalizer 机制之外，没有轻便的方法来防止这类问题。在这方面， Cleaner 机制比 Finalizer 机制要好一些，因为 Java 类的创建者可以控制自己 cleaner 机制的线程，但 cleaner 机制仍然在后台运行，在垃圾回收器的控制下运行，但不能保证及时清理。

Java 规范不能保证 Finalizer 和 Cleaner 机制能及时运行；它甚至不能能保证它们是否会运行。当一个程序结束后，一些不可达对象上的 Finalizer 和 Cleaner 机制仍然没有运行。因此，不应该依赖于 Finalizer 和 Cleaner 机制来更新持久化状态。例如，依赖于 Finalizer 和 Cleaner 机制来释放对共享资源（如数据库）的持久锁，这是一个使整个分布式系统陷入停滞的好方法。

不要相信 `System.gc` 和 `System.runFinalization` 方法。 他们可能会增加 Finalizer 和 Cleaner 机制被执行的几率，但不能保证一定会执行。 曾经声称做出这种保证的两个方法：`System.runFinalizersOnExit` 和它的孪生兄弟 `Runtime.runFinalizersOnExit`，包含致命的缺陷，并已被弃用了几十年[ThreadStop]。

Finalizer 机制的另一个问题是在执行 Finalizer 机制过程中，未捕获的异常会被忽略，并且该对象的 Finalizer 机制也会终止 [JLS, 12.6]。未捕获的异常会使其他对象陷入一种损坏的状态（corrupt state）。如果另一个线程试图使用这样一个损坏的对象，可能会导致任意不确定的行为。通常情况下，未捕获的异常将终止线程并打印堆栈跟踪（ stacktrace），但如果发生在 Finalizer 机制中，则不会发出警告。Cleaner 机制没有这个问题，因为使用 Cleaner 机制的类库可以控制其线程。

使用 finalizer 和 cleaner 机制会导致严重的性能损失。 在我的机器上，创建一个简单的 `AutoCloseable` 对象，使用 try-with-resources 关闭它，并让垃圾回收器回收它的时间大约是 12 纳秒。 使用 finalizer 机制，而时间增加到 550 纳秒。 换句话说，使用 finalizer 机制创建和销毁对象的速度要慢 50 倍。 这主要是因为 finalizer 机制会阻碍有效的垃圾收集。 如果使用它们来清理类的所有实例（在我的机器上的每个实例大约是 500 纳秒），那么 cleaner 机制的速度与 finalizer 机制的速度相当，但是如果仅将它们用作安全网（safety net），则 cleaner 机制要快得多，如下所述。 在这种环境下，创建，清理和销毁一个对象在我的机器上需要大约 66 纳秒，这意味着如果你不使用安全网的话，需要支付 5 倍（而不是 50 倍）的保险。

finalizer 机制有一个严重的安全问题：它们会打开你的类来进行 finalizer 机制攻击。finalizer 机制攻击的想法很简单：如果一个异常是从构造方法或它的序列化中抛出的——`readObject` 和 `readResolve` 方法 （第 12 章）——恶意子类的 finalizer 机制可以运行在本应该「中途夭折（died on the vine）」的部分构造对象上。finalizer 机制可以在静态字属性记录对对象的引用，防止其被垃圾收集。一旦记录了有缺陷的对象，就可以简单地调用该对象上的任意方法，而这些方法本来就不应该允许存在。从构造方法中抛出异常应该足以防止对象出现；而在 finalizer 机制存在下，则不是。这样的攻击会带来可怕的后果。Final 类不受 finalizer 机制攻击的影响，因为没有人可以编写一个 final 类的恶意子类。为了保护非 final 类不受 finalizer 机制攻击，编写一个 final 的 `finalize` 方法，它什么都不做。

那么，你应该怎样做呢？为对象封装需要结束的资源（如文件或线程），而不是为该类编写 Finalizer 和 Cleaner 机制？让你的类实现 `AutoCloseable` 接口即可，并要求客户在在不再需要时调用每个实例 close 方法，通常使用 try-with-resources 确保终止，即使面对有异常抛出情况（详见第 9 条）。一个值得一提的细节是实例必须跟踪是否已经关闭：close 方法必须记录在对象里不再有效的属性，其他方法必须检查该属性，如果在对象关闭后调用它们，则抛出 `IllegalStateException` 异常。

那么，Finalizer 和 Cleaner 机制有什么好处呢？它们可能有两个合法用途。一个是作为一个安全网（safety net），以防资源的拥有者忽略了它的 `close` 方法。虽然不能保证 Finalizer 和 Cleaner 机制会迅速运行 (或者根本就没有运行)，最好是把资源释放晚点出来，也要好过客户端没有这样做。如果你正在考虑编写这样的安全网 Finalizer 机制，请仔细考虑一下这样保护是否值得付出对应的代价。一些 Java 库类，如 `FileInputStream`、`FileOutputStream`、`ThreadPoolExecutor` 和 `java.sql.Connection`，都有作为安全网的 Finalizer 机制。

第二种合理使用 Cleaner 机制的方法与本地对等类（native peers）有关。本地对等类是一个由普通对象委托的本地 (非 Java) 对象。由于本地对等类不是普通的 Java 对象，所以垃圾收集器并不知道它，当它的 Java 对等对象被回收时，本地对等类也不会回收。假设性能是可以接受的，并且本地对等类没有关键的资源，那么 Finalizer 和 Cleaner 机制可能是这项任务的合适的工具。但如果性能是不可接受的，或者本地对等类持有必须迅速回收的资源，那么类应该有一个 `close` 方法，正如前面所述。

Cleaner 机制使用起来有点棘手。下面是演示该功能的一个简单的 `Room` 类。假设 `Room` 对象必须在被回收前清理干净。`Room` 类实现 `AutoCloseable` 接口；它的自动清理安全网使用的是一个 Cleaner 机制，这仅仅是一个实现细节。与 Finalizer 机制不同，Cleaner 机制不污染一个类的公共 API：

```
// An autocloseable class using a cleaner as a safety net
public class Room implements AutoCloseable {
    private static final Cleaner cleaner = Cleaner.create();

    // Resource that requires cleaning. Must not refer to Room!
    private static class State implements Runnable {
        int numJunkPiles; // Number of junk piles in this room

        State(int numJunkPiles) {
            this.numJunkPiles = numJunkPiles;
        }

        // Invoked by close method or cleaner
        @Override
        public void run() {
            System.out.println("Cleaning room");
            numJunkPiles = 0;
        }
    }

    // The state of this room, shared with our cleanable
    private final State state;

    // Our cleanable. Cleans the room when it’s eligible for gc
    private final Cleaner.Cleanable cleanable;
```

```
    public Room(int numJunkPiles) {
        state = new State(numJunkPiles);
        cleanable = cleaner.register(this, state);
    }

    @Override
    public void close() {
        cleanable.clean();
    }
}
```

静态内部 `State` 类拥有 Cleaner 机制清理房间所需的资源。 在这里，它仅仅包含 `numJunkPiles` 属性，它代表混乱房间的数量。 更实际地说，它可能是一个 final 修饰的 `long` 类型的指向本地对等类的指针。 `State` 类实现了 `Runnable` 接口，其 `run` 方法最多只能调用一次，只能被我们在 Room 构造方法中用 `Cleaner` 机制注册 `State` 实例时得到的 `Cleanable` 调用。 对 `run` 方法的调用通过以下两种方法触发：通常，通过调用 `Room` 的 `close` 方法内调用 `Cleanable` 的 `clean` 方法来触发。 如果在 `Room` 实例有资格进行垃圾回收的时候客户端没有调用 `close` 方法，那么 `Cleaner` 机制将（希望）调用 `State` 的 `run` 方法。

一个 `State` 实例不引用它的 `Room` 实例是非常重要的。如果它引用了，则创建了一个循环，阻止了 `Room` 实例成为垃圾收集的资格（以及自动清除）。因此， `State` 必须是静态的嵌内部类，因为非静态内部类包含对其宿主类的实例的引用（详见第 24 条）。同样，使用 lambda 表达式也是不明智的，因为它们很容易获取对宿主类对象的引用。

就像我们之前说的， `Room` 的 Cleaner 机制仅仅被用作一个安全网。如果客户将所有 `Room` 的实例放在 try-with-resource 块中，则永远不需要自动清理。行为良好的客户端如下所示：

```
public class Adult {
    public static void main(String[] args) {
        try (Room myRoom = new Room(7)) {
            System.out.println("Goodbye");
        }
    }
}
```

正如你所预料的，运行 `Adult` 程序会打印 `Goodbye` 字符串，随后打印 `Cleaning room` 字符串。但是如果时不合规矩的程序，它从来不清理它的房间会是什么样的?

```
public class Teenager {
    public static void main(String[] args) {
        new Room(99);
        System.out.println("Peace out");
    }
}
```

你可能期望它打印出 `Peace out`，然后打印 `Cleaning room` 字符串，但在我的机器上，它从不打印 `Cleaning room` 字符串；仅仅是程序退出了。 这是我们之前谈到的不可预见性。 Cleaner 机制的规范说：“ `System.exit` 方法期间的清理行为是特定于实现的。 不保证清理行为是否被调用。”虽然规范没有说明，但对于正常的程序退出也是如此。 在我的机器上，将 `System.gc()` 方法添加到 `Teenager` 类的 `main` 方法足以让程序退出之前打印 `Cleaning room`，但不能保证在你的机器上会看到相同的行为。

总之，除了作为一个安全网或者终止非关键的本地资源，不要使用 Cleaner 机制，或者是在 Java 9 发布之前的 finalizers 机制。即使是这样，也要当心不确定性和性能影响。

9. 使用 try-with-resources 语句替代 try-finally 语句

Java 类库中包含许多必须通过调用 `close` 方法手动关闭的资源。 比如 `InputStream`，`OutputStream` 和 `java.sql.Connection`。 客户经常忽视关闭资源，其性能结果可想而知。 尽管这些资源中有很多使用 finalizer 机制作为安全网，但 finalizer 机制却不能很好地工作（详见第 8 条）。

从以往来看，try-finally 语句是保证资源正确关闭的最佳方式，即使是在程序抛出异常或返回的情况下：

```
// try-finally - No longer the best way to close resources!
static String firstLineOfFile(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        br.close();
    }
}
```

这可能看起来并不坏，但是当添加第二个资源时，情况会变得更糟：

```
// try-finally is ugly when used with more than one resource!
static void copy(String src, String dst) throws IOException {
    InputStream in = new FileInputStream(src);
    try {
        OutputStream out = new FileOutputStream(dst);
        try {
            byte[] buf = new byte[BUFFER_SIZE];
            int n;
            while ((n = in.read(buf)) >= 0)
                out.write(buf, 0, n);
        } finally {
            out.close();
        }
```

```
    } finally {
        in.close();
    }
}
```

这可能很难相信，但即使是优秀的程序员，大多数时候也会犯错误。首先，我在 Java Puzzlers[Bloch05] 的第 88 页上弄错了，多年来没有人注意到。事实上，2007 年 Java 类库中使用 `close` 方法的三分之二都是错误的。

即使是用 try-finally 语句关闭资源的正确代码，如前面两个代码示例所示，也有一个微妙的缺陷。try-with-resources 块和 finally 块中的代码都可以抛出异常。 例如，在 `firstLineOfFile` 方法中，由于底层物理设备发生故障，对 `readLine` 方法的调用可能会引发异常，并且由于相同的原因，调用 `close` 方法可能会失败。 在这种情况下，第二个异常完全冲掉了第一个异常。 在异常堆栈跟踪中没有第一个异常的记录，这可能使实际系统中的调试非常复杂——通常这是你想要诊断问题的第一个异常。虽然可以编写代码来抑制第二个异常，但是实际上没有人这样做，因为它太冗长了。

当 Java 7 引入了 try-with-resources 语句时，所有这些问题一下子都得到了解决[JLS,14.20.3]。要使用这个构造，资源必须实现 `AutoCloseable` 接口，该接口由一个返回为 `void` 的 `close` 组成。Java 类库和第三方类库中的许多类和接口现在都实现或继承了 `AutoCloseable` 接口。如果你编写的类表示必须关闭的资源，那么这个类也应该实现 `AutoCloseable` 接口。

以下是我们的第一个使用 try-with-resources 的示例：

```
// try-with-resources - the the best way to close resources!
static String firstLineOfFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(
            new FileReader(path))) {
        return br.readLine();
    }
}
```

以下是我们的第二个使用 try-with-resources 的示例：

```
// try-with-resources on multiple resources - short and sweet
static void copy(String src, String dst) throws IOException {
    try (InputStream   in = new FileInputStream(src);
         OutputStream out = new FileOutputStream(dst)) {
        byte[] buf = new byte[BUFFER_SIZE];
        int n;
        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    }
}
```

不仅 try-with-resources 版本比原始版本更精简，更好的可读性，而且它们提供了更好的诊断。 考虑 `firstLineOfFile` 方法。 如果调用 `readLine` 和（不可见） `close` 方法都抛出异常，则后一个异常将被抑制（suppressed），而不是前者。 事实上，为了保留你真正想看到的异常，可能会抑制多个异常。 这些抑制的异常没有被抛弃， 而是打印在堆栈跟踪中，并标注为被抑制了。 你也可以使用 `getSuppressed` 方法以编程方式访问它们，该方法在 Java 7 中已添加到的 `Throwable` 中。

可以在 try-with-resources 语句中添加 catch 子句，就像在常规的 try-finally 语句中一样。这允许你处理异常，而不会在另一层嵌套中污染代码。作为一个稍微有些做作的例子，这里有一个版本的 `firstLineOfFile` 方法，它不会抛出异常，但是如果它不能打开或读取文件，则返回默认值：

```
// try-with-resources with a catch clause
static String firstLineOfFile(String path, String defaultVal) {
    try (BufferedReader br = new BufferedReader(
            new FileReader(path))) {
        return br.readLine();
    } catch (IOException e) {
        return defaultVal;
    }
}
```

结论很明确：在处理必须关闭的资源时，使用 try-with-resources 语句替代 try-finally 语句。 生成的代码更简洁，更清晰，并且生成的异常更有用。 try-with-resources 语句在编写必须关闭资源的代码时会更容易，也不会出错，而使用 try-finally 语句实际上是不可能的。

10. 重写 equals 方法时遵守通用约定

虽然 `Object` 是一个具体的类，但它主要是为继承而设计的。它的所有非 final 方法（equals、hashCode、toString、clone 和 finalize）都有清晰的通用约定（ general contracts），因为它们被设计为被子类重写。任何类要重写这些方法时，都有义务去遵从它们的通用约定；如果不这样做，将会阻止其他依赖于约定的类 (例如 HashMap 和 HashSet) 与此类一起正常工作。

本章论述何时以及如何重写 `Object` 类的非 final 的方法。这一章省略了 finalize 方法，因为它在条目 8 中进行了讨论。 `Comparable.compareTo` 方法虽然不是 `Object` 中的方法，因为具有很多的相似性，所以也在这里讨论。

重写 equals 方法看起来很简单，但是有很多方式会导致重写出错，其结果可能是可怕的。避免此问题的最简单方法是不覆盖 equals 方法，在这种情况下，类的每个实例只与自身相等。如果满足以下任一下条件，则说明是正确的做法：

- 每个类的实例都是固有唯一的。 对于像 Thread 这样代表活动实体而不是值的类来说，这是正确的。 Object 提供的 equals 实现对这些类完全是正确的行为。
- 类不需要提供一个「逻辑相等（logical equality）」的测试功能。例如 `java.util.regex.Pattern` 可以重写 equals 方法检查两个是否代表完全相同的正则表达式 Pattern 实例，但是设计者并不认为客户需要或希望使用此功能。在这种情况下，从 Object 继承的 equals 实现是最合适的。

- 父类已经重写了 equals 方法，则父类行为完全适合于该子类。例如，大多数 Set 从 AbstractSet 继承了 equals 实现、List 从 AbstractList 继承了 equals 实现，Map 从 AbstractMap 的 Map 继承了 equals 实现。
- 类是私有的或包级私有的，可以确定它的 equals 方法永远不会被调用。如果你非常厌恶风险，可以重写 equals 方法，以确保不会被意外调用：

```
@Override
public boolean equals(Object o) {
    throw new AssertionError(); // Method is never called
}
```

什么时候需要重写 equals 方法呢？如果一个类包含一个逻辑相等（logical equality）的概念，此概念有别于对象标识（object identity），而且父类还没有重写过 equals 方法。这通常用在值类（value classes）的情况。值类只是一个表示值的类，例如 Integer 或 String 类。程序员使用 equals 方法比较值对象的引用，期望发现它们在逻辑上是否相等，而不是引用相同的对象。重写 equals 方法不仅可以满足程序员的期望，它还支持重写过 equals 的实例作为 Map 的键（key），或者 Set 里的元素，以满足预期和期望的行为。

一种不需要 equals 方法重写的值类是使用实例控制（instance control）（详见第 1 条）的类，以确保每个值至多存在一个对象。 枚举类型（详见第 34 条）属于这个类别。 对于这些类，逻辑相等与对象标识是一样的，所以 Object 的 equals 方法作用逻辑 equals 方法。

当你重写 equals 方法时，必须遵守它的通用约定。Object 的规范如下：
equals 方法实现了一个等价关系（equivalence relation）。它有以下这些属性:

- **自反性：** 对于任何非空引用 x， `x.equals(x)` 必须返回 true。
- **对称性：** 对于任何非空引用 x 和 y，如果且仅当 `y.equals(x)` 返回 true 时 `x.equals(y)` 必须返回 true。
- **传递性：** 对于任何非空引用 x、y、z，如果 `x.equals(y)` 返回 true， `y.equals(z)` 返回 true，则 `x.equals(z)` 必须返回 true。
- **一致性：** 对于任何非空引用 x 和 y，如果在 equals 比较中使用的信息没有修改，则 `x.equals(y)` 的多次调用必须始终返回 true 或始终返回 false。
- 对于任何非空引用 x， `x.equals(null)` 必须返回 false。

除非你喜欢数学，否则这看起来有点吓人，但不要忽略它！如果一旦违反了它，很可能会发现你的程序运行异常或崩溃，并且很难确定失败的根源。套用约翰·多恩（John Donne）的说法，没有哪个类是孤立存在的。一个类的实例常常被传递给另一个类的实例。许多类，包括所有的集合类，都依赖于传递给它们遵守 equals 约定的对象。

既然已经意识到违反 equals 约定的危险，让我们详细地讨论一下这个约定。好消息是，表面上看，这并不是很复杂。一旦你理解了，就不难遵守这一约定。

那么什么是等价关系？ 笼统地说，它是一个运算符，它将一组元素划分为彼此元素相等的子集。这些子集被称为等价类（equivalence classes）。 为了使 equals 方法有用，每个等价类中的所有元素必须从用户的角度来说是可以互换（interchangeable）的。 现在让我们依次看下这个五个要求：

自反性（Reflexivity）——第一个要求只是说一个对象必须与自身相等。 很难想象无意中违反了这个规定。 如果你违反了它，然后把类的实例添加到一个集合中，那么 `contains` 方法可能会说集合中没有包含刚添加的实例。

对称性（Symmetry）——第二个要求是，任何两个对象必须在是否相等的问题上达成一致。与第一个要求不同的是，我们不难想象在无意中违反了这一要求。例如，考虑下面的类，它实现了不区分大小写的字符串。字符串被 toString 保存，但在 equals 比较中被忽略：

```
import java.util.Objects;

public final class CaseInsensitiveString {
    private final String s;

    public CaseInsensitiveString(String s) {
        this.s = Objects.requireNonNull(s);
    }

    // Broken - violates symmetry!
    @Override
    public boolean equals(Object o) {
        if (o instanceof CaseInsensitiveString)
            return s.equalsIgnoreCase(
                    ((CaseInsensitiveString) o).s);
        if (o instanceof String)  // One-way interoperability!
            return s.equalsIgnoreCase((String) o);
        return false;
    }
    ...// Remainder omitted
}
```

上面类中的 equals 试图与正常的字符串进行操作，假设我们有一个不区分大小写的字符串和一个正常的字符串：

```
CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
String s = "polish";

System.out.println(cis.equals(s)); // true
System.out.println(s.equals(cis)); // false
```

正如所料，`cis.equals(s)` 返回 true。 问题是，尽管 `CaseInsensitiveString` 类中的 equals 方法知道正常字符串，但 String 类中的 equals 方法却忽略了不区分大小写的字符串。 因此，`s.equals(cis)` 返回 false，明显违反对称性。 假设把一个不区分大小写的字符串放入一个集合中：

```
List<CaseInsensitiveString> list = new ArrayList<>();
list.add(cis);
```

`list.contains(s)` 返回了什么？谁知道呢？在当前的 OpenJDK 实现中，它会返回 false，但这只是一个实现构件。在另一个实现中，它可以很容易地返回 true 或抛出运行时异常。一旦违反了 equals 约定，就不知道其他对象在面对你的对象时会如何表现了。

要消除这个问题，只需删除 equals 方法中与 String 类相互操作的恶意尝试。这样做之后，可以将该方法重构为单个返回语句:

```
@Override
public boolean equals(Object o) {
    return o instanceof CaseInsensitiveString &&
            ((CaseInsensitiveString) o).s.equalsIgnoreCase(s);
}
```

传递性（Transitivity）—— equals 约定的第三个要求是，如果第一个对象等于第二个对象，第二个对象等于第三个对象，那么第一个对象必须等于第三个对象。同样，也不难想象，无意中违反了这一要求。考虑子类的情况， 将新值组件（value component）添加到其父类中。换句话说，子类添加了一个信息，它影响了 equals 方法比较。让我们从一个简单不可变的二维整数类型 Point 类开始：

```
public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Point))
            return false;
        Point p = (Point) o;
        return p.x == x && p.y == y;
    }

    ...  // Remainder omitted
}
```

假设想继承这个类，将表示颜色的 Color 类添加到 Point 类中：

```
public class ColorPoint extends Point {
    private final Color color;

    public ColorPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    ...  // Remainder omitted
}
```

equals 方法应该是什么样子？如果完全忽略，则实现是从 Point 类上继承的，颜色信息在 equals 方法比较中被忽略。虽然这并不违反 equals 约定，但这显然是不可接受的。假设你写了一个 equals 方法，它只在它的参数是另一个具有相同位置和颜色的 ColorPoint 实例时返回 true：

```
// Broken - violates symmetry!
@Override
public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
        return false;
    return super.equals(o) && ((ColorPoint) o).color == color;
}
```

当你比较 Point 对象和 ColorPoint 对象时，可以会得到不同的结果，反之亦然。前者的比较忽略了颜色属性，而后者的比较会一直返回 false，因为参数的类型是错误的。为了让问题更加具体，我们创建一个 Point 对象和 ColorPoint 对象：

```
Point p = new Point(1, 2);
ColorPoint cp = new ColorPoint(1, 2, Color.RED);
```

p.equals(cp) 返回 true，但是 cp.equals(p) 返回 false。你可能想使用 ColorPoint.equals 通过混合比较的方式来解决这个问题。

```
@Override
public boolean equals(Object o) {
    if (!(o instanceof Point))
        return false;

    // If o is a normal Point, do a color-blind comparison
    if (!(o instanceof ColorPoint))
        return o.equals(this);

    // o is a ColorPoint; do a full comparison
    return super.equals(o) && ((ColorPoint) o).color == color;
}
```

这种方法确实提供了对称性，但是丧失了传递性：

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

现在，`p1.equals(p2)` 和 `p2.equals(p3)` 返回了 true，但是 `p1.equals(p3)` 却返回了 false，很明显违背了传递性的要求。前两个比较都是不考虑颜色信息的，而第三个比较时却包含颜色信息。

此外，这种方法可能导致无限递归：假设有两个 Point 的子类，比如 ColorPoint 和 SmellPoint，每个都有这种 equals 方法。 然后调用 `myColorPoint.equals(mySmellPoint)` 将抛出一个 StackOverflowError 异常。

那么解决方案是什么？ 事实证明，这是面向对象语言中关于等价关系的一个基本问题。 除非您愿意放弃面向对象抽象的好处，否则无法继承可实例化的类，并在保留 equals 约定的同时添加一个值组件。

你可能听说过，可以继承一个可实例化的类并添加一个值组件，同时通过在 equals 方法中使用一个 getClass 测试代替 instanceof 测试来保留 equals 约定：

```
@Override
public boolean equals(Object o) {
    if (o == null || o.getClass() != getClass())
        return false;
    Point p = (Point) o;
    return p.x == x && p.y == y;
}
```

只有当对象具有相同的实现类时，才会产生相同的效果。这看起来可能不是那么糟糕，但是结果是不可接受的:一个 Point 类子类的实例仍然是一个 Point 的实例，它仍然需要作为一个 Point 来运行，但是如果你采用这个方法，就会失败！假设我们要写一个方法来判断一个 Point 对象是否在 unitCircle 集合中。我们可以这样做：

```
private static final Set<Point> unitCircle = Set.of(
        new Point( 1,  0), new Point( 0,  1),
        new Point(-1,  0), new Point( 0, -1));

public static boolean onUnitCircle(Point p) {
    return unitCircle.contains(p);
}
```

虽然这可能不是实现功能的最快方法，但它可以正常工作。假设以一种不添加值组件的简单方式继承 Point 类，比如让它的构造方法跟踪记录创建了多少实例：

```
public class CounterPoint extends Point {
    private static final AtomicInteger counter =
            new AtomicInteger();

    public CounterPoint(int x, int y) {
        super(x, y);
        counter.incrementAndGet();
    }

    public static int numberCreated() {
        return counter.get();
    }
}
```

里氏替代原则（Liskov substitution principle）指出，任何类型的重要属性都应该适用于所有的子类型，因此任何为这种类型编写的方法都应该在其子类上同样适用[Liskov87]。 这是我们之前声明的一个正式陈述，即 Point 的子类（如 CounterPoint）仍然是一个 Point，必须作为一个 Point 类来看待。 但是，假设我们将一个 CounterPoint 对象传递给 onUnitCircle 方法。 如果 Point 类使用基于 getClass 的 equals 方法，则无论 CounterPoint 实例的 x 和 y 坐标如何，onUnitCircle 方法都将返回 false。 这是因为大多数集合（包括 onUnitCircle 方法使用的 HashSet）都使用 equals 方法来测试是否包含元素，并且 CounterPoint 实例并不等于任何 Point 实例。 但是，如果在 Point 上使用了适当的基于 `instanceof` 的 equals 方法，则在使用 CounterPoint 实例呈现时，同样的 onUnitCircle 方法可以正常工作。

虽然没有令人满意的方法来继承一个可实例化的类并添加一个值组件，但是有一个很好的变通方法：按照条目 18 的建议，“优先使用组合而不是继承”。取代继承 Point 类的 ColorPoint 类，可以在 ColorPoint 类中定义一个私有 Point 属性，和一个公共的视图（view）（详见第 6 条）方法，用来返回具有相同位置的 ColorPoint 对象。

```
// Adds a value component without violating the equals contract
public class ColorPoint {
    private final Point point;
    private final Color color;

    public ColorPoint(int x, int y, Color color) {
        point = new Point(x, y);
        this.color = Objects.requireNonNull(color);
    }

    /**
     * Returns the point-view of this color point.
     */
    public Point asPoint() {
        return point;
    }

    @Override
    public boolean equals(Object o) {
```

```
        if (!(o instanceof ColorPoint))
            return false;
        ColorPoint cp = (ColorPoint) o;
        return cp.point.equals(point) && cp.color.equals(color);
    }

    ...    // Remainder omitted
}
```

Java 平台类库中有一些类可以继承可实例化的类并添加一个值组件。 例如，`java.sql.Timestamp` 继承了 `java.util.Date` 并添加了一个 nanoseconds 字段。 Timestamp 的等价 equals 确实违反了对称性，并且如果 Timestamp 和 Date 对象在同一个集合中使用，或者以其他方式混合使用，则可能导致不稳定的行为。 Timestamp 类有一个免责声明，告诫程序员不要混用 Timestamp 和 Date。 虽然只要将它们分开使用就不会遇到麻烦，但没有什么可以阻止你将它们混合在一起，并且由此产生的错误可能很难调试。 Timestamp 类的这种行为是一个错误，不应该被仿效。

你可以将值组件添加到抽象类的子类中，而不会违反 equals 约定。这对于通过遵循第 23 个条目中“优先考虑类层级（class hierarchies）来代替标记类（tagged classes）”中的建议而获得的类层级，是非常重要的。例如，可以有一个没有值组件的抽象类 Shape，子类 Circle 有一个 radius 属性，另一个子类 Rectangle 包含 length 和 width 属性 。 只要不直接创建父类实例，就不会出现前面所示的问题。

一致性（Consistent）——equals 约定的第四个要求是，如果两个对象是相等的，除非一个（或两个）对象被修改了， 那么它们必须始终保持相等。 换句话说，可变对象可以在不同时期可以与不同的对象相等，而不可变对象则不会。 当你写一个类时，要认真思考它是否应该设计为不可变的（详见第 17 条）。 如果你认为应该这样做，那么确保你的 equals 方法强制执行这样的限制：相等的对象永远相等，不相等的对象永远都不会相等。

不管一个类是不是不可变的，都不要写一个依赖于不可靠资源的 equals 方法。 如果违反这一禁令，满足一致性要求是非常困难的。 例如，`java.net.URL` 类中的 equals 方法依赖于与 URL 关联的主机的 IP 地址的比较。 将主机名转换为 IP 地址可能需要访问网络，并且不能保证随着时间的推移会产生相同的结果。 这可能会导致 URL 类的 equals 方法违反 equals 约定，并在实践中造成问题。 URL 类的 equals 方法的行为是一个很大的错误，不应该被效仿。 不幸的是，由于兼容性的要求，它不能改变。 为了避免这种问题，equals 方法应该只对内存驻留对象执行确定性计算。

非空性（Non-nullity）——最后 equals 约定的要求没有官方的名称，所以我冒昧地称之为“非空性”。意思是说说所有的对象都必须不等于 null。虽然很难想象在调用 `o.equals(null)` 的响应中意外地返回 true，但不难想象不小心抛出 `NullPointerException` 异常的情况。通用的约定禁止抛出这样的异常。许多类中的 equals 方法都会明确阻止对象为 null 的情况：

```
@Override
public boolean equals(Object o) {
    if (o == null)
        return false;
    ...
}
```

这个判断是不必要的。 为了测试它的参数是否相等，equals 方法必须首先将其参数转换为合适类型，以便调用访问器或允许访问的属性。 在执行类型转换之前，该方法必须使用 instanceof 运算符来检查其参数是否是正确的类型：

```
@Override
public boolean equals(Object o) {
    if (!(o instanceof MyType))
        return false;
    MyType mt = (MyType) o;
    ...
}
```

如果此类型检查漏掉，并且 equals 方法传递了错误类型的参数，那么 equals 方法将抛出 `ClassCastException` 异常，这违反了 equals 约定。 但是，如果第一个操作数为 null，则指定 instanceof 运算符返回 false，而不管第二个操作数中出现何种类型[JLS，15.20.2]。 因此，如果传入 null，类型检查将返回 false，因此不需要 明确的 null 检查。

综合起来，以下是编写高质量 equals 方法的配方（recipe）：

1. 使用 == 运算符检查参数是否为该对象的引用。如果是，返回 true。这只是一种性能优化，但是如果这种比较可能很昂贵的话，那就值得去做。
2. 使用 `instanceof` 运算符来检查参数是否具有正确的类型。 如果不是，则返回 false。 通常，正确的类型是 equals 方法所在的那个类。 有时候，改类实现了一些接口。 如果类实现了一个接口，该接口可以改进 equals 约定以允许实现接口的类进行比较，那么使用接口。 集合接口（如 Set，List，Map 和 Map.Entry）具有此特性。
3. 参数转换为正确的类型。因为转换操作在 instanceof 中已经处理过，所以它肯定会成功。
4. 对于类中的每个「重要」的属性，请检查该参数属性是否与该对象对应的属性相匹配。如果所有这些测试成功，返回 true，否则返回 false。如果步骤 2 中的类型是一个接口，那么必须通过接口方法访问参数的属性;如果类型是类，则可以直接访问属性，这取决于属性的访问权限。

对于类型为非 float 或 double 的基本类型，使用 == 运算符进行比较；对于对象引用属性，递归地调用 equals 方法；对于 float 基本类型的属性，使用静态 `Float.compare(float, float)` 方法；对于 double 基本类型的属性，使用 `Double.compare(double, double)` 方法。由于存在 `Float.NaN` ， `-0.0f` 和类似的 double 类型的值，所以需要对 float 和 double 属性进行特殊的处理；有关详细信息，请参阅 JLS 15.21.1 或 Float.equals 方法的详细文档。 虽然你可以使用静态方法 Float.equals 和 Double.equals 方法对 float 和 double 基本类型的属性进行比较，这会导致每次比较时发生自动装箱，引发非常差的性能。 对于数组属性，将这些准则应用于每个元素。 如果数组属性中的每个元素都很重要，请使用其中一个重载的 Arrays.equals 方法。

某些对象引用的属性可能合法地包含 null。 为避免出现 `NullPointerException` 异常，请使用静态方法 Objects.equals(Object, Object) 检查这些属性是否相等。

对于一些类，例如上的 `CaseInsensitiveString` 类，属性比较相对于简单的相等性测试要复杂得多。在这种情况下，你想要保存属性的一个规范形式（canonical form），这样 equals 方法就可以基于这个规范形式去做开销很小的精确比较，来取代开销很大的非标准比较。这种方式其实最适合不可变类（详见第 17 条）。一旦对象发生改变，一定要确保把对应的规范形式更新到最新。

equals 方法的性能可能受到属性比较顺序的影响。 为了获得最佳性能，你应该首先比较最可能不同的属性，开销比较小的属性，或者最好是两者都满足（derived fields）。 你不要比较不属于对象逻辑状态的属性，例如用于同步操作的 lock 属性。 不需要比较可以从“重要属性”计算出来的派生属性，但是这样做可以提高 equals 方法的性能。 如果派生属性相当于对整个对象的摘要描述，比较这个属性将节省在比较失败时再去比较实际数据的开销。 例如，假设有一个 Polygon 类，并缓存该区域。 如果两个多边形的面积不相等，则不必费心比较它们的边和顶点。

当你完成编写完 equals 方法时，问你自己三个问题：它是对称的吗?它是传递吗?它是一致的吗?除此而外，编写单元测试加以排查，除非使用 AutoValue 框架（第 49 页）来生成 equals 方法，在这种情况下可以安全地省略测试。如果持有的属性失败，找出原因，并相应地修改 equals 方法。当然，equals 方法也必须满足其他两个属性 (自反性和非空性)，但这两个属性通常都会满足。

在下面这个简单的 `PhoneNumber` 类中展示了根据之前的配方构建的 equals 方法：

```
public final class PhoneNumber {

    private final short areaCode, prefix, lineNum;

    public PhoneNumber(int areaCode, int prefix, int lineNum) {
        this.areaCode = rangeCheck(areaCode, 999, "area code");
        this.prefix = rangeCheck(prefix, 999, "prefix");
        this.lineNum = rangeCheck(lineNum, 9999, "line num");
    }

    private static short rangeCheck(int val, int max, String arg) {
        if (val < 0 || val > max)
            throw new IllegalArgumentException(arg + ": " + val);

        return (short) val;
    }

    @Override
    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;

        PhoneNumber pn = (PhoneNumber) o;

        return pn.lineNum == lineNum && pn.prefix == prefix
                && pn.areaCode == areaCode;
    }

    ... // Remainder omitted
}
```

以下是一些最后提醒：

1. **当重写 equals 方法时，同时也要重写 hashCode 方法（详见第 11 条）。**
2. **不要让 equals 方法试图太聪明。** 如果只是简单地测试用于相等的属性，那么要遵守 equals 约定并不困难。如果你在寻找相等方面过于激进，那么很容易陷入麻烦。一般来说，考虑到任何形式的别名通常是一个坏主意。例如，File 类不应该试图将引用的符号链接等同于同一文件对象。幸好 File 类并没这么做。
3. **在 equal 时方法声明中，不要将参数 Object 替换成其他类型。** 对于程序员来说，编写一个如下所示的 equals 方法，然后花上几个小时苦苦思索为什么不能正常工作的情况并不少见：

```
// Broken - parameter type must be Object!
public boolean equals(MyClass o) {
    …
}
```

问题在于这个方法并没有重写 Object.equals 方法，它的参数是 Object 类型的，这样写只是重载了 equals 方法（详见第 52 条）。 即使除了正常的方法之外，提供这种“强类型”的 equals 方法也是不可接受的，因为它可能会导致子类中的 Override 注解产生误报，提供不安全的错觉。
在这里，使用 Override 注解会阻止你犯这个错误 （详见第 40 条）。这个 equals 方法不会编译，错误消息会告诉你到底错在哪里：

```
// Still broken, but won’t compile
@Override
public boolean equals(MyClass o) {
    …
}
```

编写和测试 equals（和 hashCode）方法很繁琐，生的代码也很普通。替代手动编写和测试这些方法的优雅的手段是，使用谷歌 AutoValue 开源框架，该框架自动为你生成这些方法，只需在类上添加一个注解即可。在大多数情况下，AutoValue 框架生成的方法与你自己编写的方法本质上是相同的。

很多 IDE（例如 Eclipse，NetBeans，IntelliJ IDEA 等）也有生成 equals 和 hashCode 方法的功能，但是生成的源代码比使用 AutoValue 框架的代码更冗长、可读性更差，不会自动跟踪类中的更改，因此需要进行测试。这就是说，使用 IDE 工具生成 equals(和 hashCode) 方法通常比手动编写它们更可取，因为 IDE 工具不会犯粗心大意的错误，而人类则会。

总之，除非必须：在很多情况下，不要重写 equals 方法，从 Object 继承的实现完全是你想要的。如果你确实重写了 equals 方法，那么一定要比较这个类的所有重要属性，并且以保护前面 equals 约定里五个规定的方式去比较。

11. 重写 equals 方法时同时也要重写 hashcode 方法

在每个类中，在重写 equals 方法的时侯，一定要重写 hashcode 方法。 如果不这样做，你的类违反了 hashCode 的通用约定，这会阻止它在 HashMap 和 HashSet 这样的集合中正常工作。根据 Object 规范，以下时具体约定。

1. 如果没有修改 equals 方法中用以比较的信息，在应用程序的一次执行过程中对一个对象重复调用 hashCode 方法时，它必须始终返回相同的值。在应用程序的多次执行过程中，每个执行过程在该对象上获取的结果值可以不相同。
2. 如果两个对象根据 equals(Object) 方法比较是相等的，那么在两个对象上调用 hashCode 就必须产生的结果是相同的整数。
3. 如果两个对象根据 equals(Object) 方法比较并不相等，则不要求在每个对象上调用 hashCode 都必须产生不同的结果。 但是，程序员应该意识到，为不相等的对象生成不同的结果可能会提高散列表（hash tables）的性能。

当无法重写 hashCode 时，所违反第二个关键条款是：相等的对象必须具有相等的哈希码（hash codes）。 根据类的 equals 方法，两个不同的实例可能在逻辑上是相同的，但是对于 Object 类的 hashCode 方法，它们只是两个没有什么共同之处的对象。因此， Object 类的 hashCode 方法返回两个看似随机的数字，而不是按约定要求的两个相等的数字。

举例说明，假设你使用条目 10 中的 `PhoneNumber` 类的实例做为 HashMap 的键（key）：

```
Map<PhoneNumber, String> m = new HashMap<>();
m.put(new PhoneNumber(707, 867, 5309), "Jenny");
```

你可能期望 `m.get(new PhoneNumber(707, 867, 5309))` 方法返回 `Jenny` 字符串，但实际上，返回了 null。注意，这里涉及到两个 `PhoneNumber` 实例：一个实例插入到 HashMap 中，另一个作为判断相等的实例用来检索。 `PhoneNumber` 类没有重写 hashCode 方法导致两个相等的实例返回了不同的哈希码，违反了 hashCode 约定。put 方法把 `PhoneNumber` 实例保存在了一个哈希桶（hash bucket）中，但 get 方法却是从不同的哈希桶中去查找，即使恰好两个实例放在同一个哈希桶中，get 方法几乎肯定也会返回 null。因为 HashMap 做了优化，缓存了与每一项（entry）相关的哈希码，如果哈希码不匹配，则不会检查对象是否相等了。

解决这个问题很简单，只需要为 `PhoneNumber` 类重写一个合适的 hashCode 方法。hashCode 方法是什么样的？写一个不规范的方法的是很简单的。以下示例，虽然永远是合法的，但绝对不能这样使用：

```
// The worst possible legal hashCode implementation - never use!
@Override public int hashCode() { return 42; }
```

这是合法的，因为它确保了相等的对象具有相同的哈希码。这很糟糕，因为它确保了每个对象都有相同的哈希码。因此，每个对象哈希到同一个桶中，哈希表退化为链表。应该在线性时间内运行的程序，运行时间变成了平方级别。对于数据很大的哈希表而言，会影响到能够正常工作。

一个好的 hash 方法趋向于为不相等的实例生成不相等的哈希码。这也正是 hashCode 约定中第三条的表达。理想情况下，hash 方法为集合中不相等的实例均匀地分配 int 范围内的哈希码。实现这种理想情况可能是困难的。 幸运的是，要获得一个合理的近似的方式并不难。 以下是一个简单的配方：

1. 声明一个 int 类型的变量 result，并将其初始化为对象中第一个重要属性 `c` 的哈希码，如下面步骤 2.a 中所计算的那样。（回顾条目 10，重要的属性是影响比较相等的领域。）
2. 对于对象中剩余的重要属性 `f`，请执行以下操作：

    a. 比较属性 `f` 与属性 `c` 的 int 类型的哈希码：

-- i. 如果这个属性是基本类型的，使用 `Type.hashCode(f)` 方法计算，其中 `Type` 类是对应属性 `f` 基本类型的包装类。

-- ii. 如果该属性是一个对象引用，并且该类的 equals 方法通过递归调用 equals 来比较该属性，并递归地调用 hashCode 方法。 如果需要更复杂的比较，则计算此字段的"范式（"canonical representation）"，并在范式上调用 hashCode。 如果该字段的值为空，则使用 0（也可以使用其他常数，但通常来使用 0 表示）。

-- iii. 如果属性 `f` 是一个数组，把它看作每个重要的元素都是一个独立的属性。 也就是说，通过递归地应用这些规则计算每个重要元素的哈希码，并且将每个步骤 2.b 的值合并。 如果数组没有重要的元素，则使用一个常量，最好不要为 0。如果所有元素都很重要，则使用 `Arrays.hashCode` 方法。

b. 将步骤 2.a 中属性 c 计算出的哈希码合并为如下结果： `result = 31 * result + c;`

3. 返回 result 值。

当你写完 hashCode 方法后，问自己是否相等的实例有相同的哈希码。 编写单元测试来验证你的直觉（除非你使用 AutoValue 框架来生成你的 equals 和 hashCode 方法，在这种情况下，你可以放心地忽略这些测试）。 如果相同的实例有不相等的哈希码，找出原因并解决问题。

可以从哈希码计算中排除派生属性（derived fields）。换句话说，如果一个属性的值可以根据参与计算的其他属性值计算出来，那么可以忽略这样的属性。您必须排除在 equals 比较中没有使用的任何属性，否则可能会违反 hashCode 约定的第二条。

步骤 2.b 中的乘法计算结果取决于属性的顺序，如果类中具有多个相似属性，则产生更好的散列函数。 例如，如果乘法计算从一个 String 散列函数中被省略，则所有的字符将具有相同的散列码。 之所以选择 31，因为它是一个奇数的素数。 如果它是偶数，并且乘法溢出，信息将会丢失，因为乘以 2 相当于移位。 使用素数的好处不太明显，但习惯上都是这么做的。 31 的一个很好的特性，是在一些体系结构中乘法可以被替换为移位和减法以获得更好的性能： `31 * i == (i << 5) - i`。 现代 JVM 可以自动进行这种优化。

让我们把上述办法应用到 `PhoneNumber` 类中：

```
// Typical hashCode method
@Override
public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

因为这个方法返回一个简单的确定性计算的结果，它的唯一的输入是 `PhoneNumber` 实例中的三个重要的属性，所以显然相等的 `PhoneNumber` 实例具有相同的哈希码。 实际上，这个方法是 `PhoneNumber` 的一个非常好的 hashCode 实现，与 Java 平台类库中的实现一样。 它很简单，速度相当快，并且合理地将不相同的电话号码分散到不同的哈希桶中。

虽然在这个项目的方法产生相当好的哈希函数，但并不是最先进的。 它们的质量与 Java 平台类库的值类型中找到的哈希函数相当，对于大多数用途来说都是足够的。 如果真的需要哈希函数而不太可能产生碰撞，请参阅 Guava 框架的的com.google.common.hash.Hashing [Guava] 方法。

`Objects` 类有一个静态方法，它接受任意数量的对象并为它们返回一个哈希码。 这个名为 hash 的方法可以让你编写一行 hashCode 方法，其质量与根据这个项目中的上面编写的方法相当。 不幸的是，它们的运行速度更慢，因为它们需要创建数组以传递可变数量的参数，以及如果任何参数是基本类型，则进行装箱和取消装箱。 这种哈希函数的风格建议仅在性能不重要的情况下使用。 以下是使用这种技术编写的 `PhoneNumber` 的哈希函数：

```
// One-line hashCode method - mediocre performance
@Override
public int hashCode() {
   return Objects.hash(lineNum, prefix, areaCode);
}
```

如果一个类是不可变的，并且计算哈希码的代价很大，那么可以考虑在对象中缓存哈希码，而不是在每次请求时重新计算哈希码。 如果你认为这种类型的大多数对象将被用作哈希键，那么应该在创建实例时计算哈希码。 否则，可以选择在首次调用 hashCode 时延迟初始化（lazily initialize）哈希码。 需要注意确保类在存在延迟初始化属性的情况下保持线程安全（项目 83）。 `PhoneNumber` 类不适合这种情况，但只是为了展示它是如何完成的。 请注意，属性 hashCode 的初始值（在本例中为 0）不应该是通常创建的实例的哈希码：

```
// hashCode method with lazily initialized cached hash code
private int hashCode; // Automatically initialized to 0

@Override
public int hashCode() {
    int result = hashCode;
    if (result == 0) {
        result = Short.hashCode(areaCode);
        result = 31 * result + Short.hashCode(prefix);
        result = 31 * result + Short.hashCode(lineNum);
        hashCode = result;
    }
    return result;
}
```

不要试图从哈希码计算中排除重要的属性来提高性能。 由此产生的哈希函数可能运行得更快，但其质量较差可能会降低哈希表的性能，使其无法使用。 具体来说，哈希函数可能会遇到大量不同的实例，这些实例主要在你忽略的区域中有所不同。 如果发生这种情况，哈希函数将把所有这些实例映射到少许哈希码上，而应该以线性时间运行的程序将会运行平方级的时间。

这不仅仅是一个理论问题。 在 Java 2 之前，String 类哈希函数在整个字符串中最多使用 16 个字符，从第一个字符开始，在整个字符串中均匀地选取。 对于大量的带有层次名称的集合（如 URL），此功能正好显示了前面描述的病态行为。

不要为 hashCode 返回的值提供详细的规范，因此客户端不能合理地依赖它； 你可以改变它的灵活性。 Java 类库中的许多类（例如 String 和 Integer）都将 hashCode 方法返回的确切值指定为实例值的函数。 这不是一个好主意，而是一个我们不得不忍受的错误：它妨碍了在未来版本中改进哈希函数的能力。 如果未指定细节并在散列函数中发现缺陷，或者发现了更好的哈希函数，则可以在后续版本中对其进行更改。

总之，每次重写 equals 方法时都必须重写 hashCode 方法，否则程序将无法正常运行。你的 hashCode 方法必须遵从 Object 类指定的常规约定，并且必须执行合理的工作，将不相等的哈希码分配给不相等的实例。如果使用第 51 页的配方，这很容易实现。如条目 10 所述，AutoValue 框架为手动编写 equals 和 hashCode 方法提供了一个很好的选择，IDE 也提供了一些这样的功能。

12. 始终重写 toString 方法

虽然 Object 类提供了 toString 方法的实现，但它返回的字符串通常不是你的类的用户想要看到的。 它由类名后跟一个「at」符号（@）和哈希码的无符号十六进制表示组成，例如 `PhoneNumber@163b91`。 toString 的通用约定要求，返回的字符串应该是「一个简洁但内容丰富的表示，对人们来说是很容易阅读的」。虽然可以认为 `PhoneNumber@163b91` 简洁易读，但相比于 `707-867-5309`，但并不是很丰富 。 toString 通用约定「建议所有的子类重写这个方法」。好的建议，的确如此!

虽然它并不像遵守 equals 和 hashCode 约定那样重要 (条目 10 和 11)，但是提供一个良好的 toString 实现使你的类更易于使用，并对使用此类的系统更易于调试。当对象被传递到 println、printf、字符串连接操作符或断言，或者由调试器打印时，toString 方法会自动被调用。即使你从不调用对象上的 toString，其他人也可以。例如，有一个引用了某对象的组件，它可能在日志错误信息中包含该对象的字符串描述。如果未能重写 toString，则消息可能是无用的。

如果为 `PhoneNumber` 提供了一个很好的 toString 方法，那么生成一个有用的诊断消息就像下面这样简单：

```
System.out.println("Failed to connect to " + phoneNumber);
```

除非你重写 toString 方法，否则程序员以这种方式生成的诊断消息将一无是处。 提供一个良好的 toString 方法不仅惠及类的实例，而且有益于那些包含实例引用的对象，集合尤为明显。 当打印一个 map 时你更愿看到 `{Jenny=PhoneNumber@163b91}` 还是 `{Jenny=707-867-5309}`?

实际上，toString 方法应该返回对象中包含的所有需要关注的信息，如电话号码示例中所示。 如果对象很大或者包含不利于字符串表示的状态，这是不切实际的。 在这种情况下，toString 应该返回一个摘要，如 `Manhattan residential phone directory (1487536 listings)` 或线程 `[main, 5, main]`。 理想情况下，字符串应该是不言自明的（线程示例并没有遵守这点）。 如果未能将所有对象的值得关注的信息包含在字符串表示中，则会导致一个特别烦人的处罚：测试失败报告如下所示：

```
Assertion failure: expected {abc, 123}, but was {abc, 123}.
```

实现 toString 方法时，必须做出的一个重要决定是：在文档中指定返回值的格式。 建议你对值类进行此操作，例如电话号码或矩阵类。 指定格式的好处是它可以作为标准的，明确的，可读的对象表示。 这种表示形式可以用于输入、输出以及持久化可读性的数据对象，如 CSV 文件。 如果指定了格式，通常提供一个匹配的静态工厂或构造方法，是个好主意，所以程序员可以轻松地在对象和字符串表示之间来回转换。 Java 平台类库中的许多值类都采用了这种方法，包括 BigInteger，BigDecimal 和大部分基本类型包装类。

指定 toString 返回值的格式的缺点是，假设你的类被广泛使用，一旦指定了格式，就会终身使用。程序员将编写代码来解析表达式，生成它，并将其嵌入到持久数据中。如果在将来的版本中更改了格式的表示，那么会破坏他们的代码和数据，并且还会抱怨。但通过选择不指定格式，就可以保留在后续版本中添加信息或改进格式的灵活性。

无论是否决定指定格式，你都应该清楚地在文档中表明你的意图。如果指定了格式，则应该这样做。例如，这里有一个 toString 方法，该方法在条目 11 中使用 `PhoneNumber` 类：

```
/**
 * Returns the string representation of this phone number.
 * The string consists of twelve characters whose format is
 * "XXX-YYY-ZZZZ", where XXX is the area code, YYY is the
 * prefix, and ZZZZ is the line number. Each of the capital
 * letters represents a single decimal digit.
 *
 * If any of the three parts of this phone number is too small
 * to fill up its field, the field is padded with leading zeros.
 * For example, if the value of the line number is 123, the last
 * four characters of the string representation will be "0123".
 */
@Override
public String toString() {
    return String.format("%03d-%03d-%04d",
            areaCode, prefix, lineNum);
}
```

如果你决定不指定格式，那么文档注释应该是这样的：

```
/**
 * Returns a brief description of this potion. The exact details
 * of the representation are unspecified and subject to change,
 * but the following may be regarded as typical:
 *
 * "[Potion #9: type=love, smell=turpentine, look=india ink]"
 */
@Override
public String toString() { ... }
```

在阅读了这条注释之后，那些生成依赖于格式细节的代码或持久化数据的程序员，在这种格式发生改变的时候，只能怪他们自己。

无论是否指定格式，都可以通过编程方式访问 toString 返回的值中包含的信息。 例如，`PhoneNumber` 类应该包含 areaCode, prefix, lineNum 这三个属性。 如果不这样做，就会强迫程序员需要这些信息来解析字符串。 除了降低性能和程序员做不必要的工作之外，这个过程很容易出错，如果改变格式就会中断，并导致脆弱的系统。 由于未能提供访问器，即使已指定格式可能会更改，也可以将字符串格式转换为事实上的 API。

在静态工具类（详见第 4 条）中编写 toString 方法是没有意义的。 你也不应该在大多数枚举类型（条目 34）中写一个 toString 方法，因为 Java 为你提供了一个非常好的方法。 但是，你应该在任何抽象类中定义 toString 方法，该类的子类共享一个公共字符串表示形式。 例如，大多数集合实现上的 toString 方法都是从抽象集合类继承的。

Google 的开放源代码 AutoValue 工具在条目 10 中讨论过，它为你生成一个 toString 方法，就像大多数 IDE 工具一样。 这些方法非常适合告诉你每个属性的内容，但并不是专门针对类的含义。 因此，例如，为我们的 `PhoneNumber` 类使用自动生成的 toString 方法是不合适的（因为电话号码具有标准的字符串表示形式），但是对于我们的 `Potion` 类来说，这是完全可以接受的。 也就是说，自动生成的 toString 方法比从 Object 继承的方法要好得多，它不会告诉你对象的值。

回顾一下，除非父类已经这样做了，否则在每个实例化的类中重写 Object 的 toString 实现。 它使得类更加舒适地使用和协助调试。 toString 方法应该以一种美观的格式返回对象的简明有用的描述。

13. 谨慎地重写 clone 方法

Cloneable 接口的目的是作为一个 mixin 接口 （详见第 20 条），公布这样的类允许克隆。不幸的是，它没有达到这个目的。它的主要缺点是缺少 clone 方法，而 Object 的 clone 方法是受保护的。你不能，不借助反射 （详见第 65 条），仅仅因为它实现了 Cloneable 接口，就调用对象上的 clone 方法。即使是反射调用也可能失败，因为不能保证对象具有可访问的 clone 方法。尽管存在许多缺陷，该机制在合理的范围内使用，所以理解它是值得的。这个条目告诉你如何实现一个行为良好的 clone 方法，在适当的时候讨论这个方法，并提出替代方案。

既然 Cloneable 接口不包含任何方法，那它用来做什么？ 它决定了 Object 的受保护的 clone 方法实现的行为：如果一个类实现了 Cloneable 接口，那么 Object 的 clone 方法将返回该对象的逐个属性（field-by-field）拷贝；否则会抛出 `CloneNotSupportedException` 异常。这是一个非常反常的接口使用，而不应该被效仿。 通常情况下，实现一个接口用来表示可以为客户做什么。但对于 Cloneable 接口，它会修改父类上受保护方法的行为。

虽然规范并没有说明，但在实践中，实现 Cloneable 接口的类希望提供一个正常运行的公共 clone 方法。为了实现这一目标，该类及其所有父类必须遵循一个复杂的、不可执行的、稀疏的文档协议。由此产生的机制是脆弱的、危险的和不受语言影响的（extralinguistic）：它创建对象而不需要调用构造方法。

clone 方法的通用规范很薄弱的。 以下内容是从 Object 规范中复制出来的：

创建并返回此对象的副本。「复制（copy）」的确切含义可能取决于对象的类。一般意图是，对于任何对象 x，表达式 `x.clone() != x` 返回 true，并且 `x.clone().getClass() == x.getClass()` 也返回 true，但它们不是绝对的要求，但通常情况下，`x.clone().equals(x)` 返回 true，当然这个要求也不是绝对的。

根据约定，这个方法返回的对象应该通过调用 `super.clone` 方法获得的。如果一个类和它的所有父类（Object 除外）都遵守这个约定，情况就是如此，`x.clone().getClass() == x.getClass()`。

根据约定，返回的对象应该独立于被克隆的对象。为了实现这种独立性，在返回对象之前，可能需要修改由 super.clone 返回的对象的一个或多个属性。

这种机制与构造方法链（chaining）很相似，只是它没有被强制执行；如果一个类的 clone 方法返回一个通过调用构造方法获得而不是通过调用 super.clone 的实例，那么编译器不会抱怨，但是如果一个类的子类调用了 super.clone，那么返回的对象包含错误的类，从而阻止子类 clone 方法正常执行。如果一个类重写的 clone 方法是有 final 修饰的，那么这个约定可以被安全地忽略，因为子类不需要担心。但是，如果一个 final 类有一个不调用 super.clone 的 clone 方法，那么这个类没有理由实现 Cloneable 接口，因为它不依赖于 Object 的 clone 实现的行为。

假设你希望在一个类中实现 Cloneable 接口，它的父类提供了一个行为良好的 clone 方法。首先调用 super.clone。得到的对象将是原始的完全功能的复制品。在你的类中声明的任何属性将具有与原始属性相同的值。如果每个属性包含原始值或对不可变对象的引用，则返回的对象可能正是你所需要的，在这种情况下，不需要进一步的处理。例如，对于条目 11 中的 `PhoneNumber` 类，情况就是这样，但是请注意，不可变类永远不应该提供 clone 方法，因为这只会浪费复制。有了这个警告，以下是 `PhoneNumber` 类的 clone 方法：

```
// Clone method for class with no references to mutable state
@Override public PhoneNumber clone() {
    try {
        return (PhoneNumber) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();  // Can't happen
    }
}
```

为了使这个方法起作用，`PhoneNumber` 的类声明必须被修改，以表明它实现了 Cloneable 接口。虽然 Object 类的 clone 方法返回 Object 类，但是这个 clone 方法返回 `PhoneNumber` 类。这样做是合法和可取的，因为 Java 支持协变返回类型。换句话说，重写方法的返回类型可以是重写方法的返回类型的子类。这消除了在客户端转换的需要。在返回之前，我们必须将 Object 的 super.clone 的结果强制转换为 `PhoneNumber`，但保证强制转换成功。

super.clone 的调用包含在一个 try-catch 块中。这是因为 Object 声明了它的 clone 方法来抛出 `CloneNotSupportedException` 异常，这是一个检查时异常。由于 `PhoneNumber` 实现了 Cloneable 接口，所以我们知道调用 super.clone 会成功。这里引用的需要表明 `CloneNotSupportedException` 应该是未被检查的（详见第 71条）。

如果对象包含引用可变对象的属性，则前面显示的简单 clone 实现可能是灾难性的。例如，考虑条目 7 中的 Stack 类：

```
public class Stack {

    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        this.elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];

        elements[size] = null; // Eliminate obsolete reference
        return result;
    }

    // Ensure space for at least one more element.
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

假设你想让这个类可以克隆。 如果 clone 方法仅返回 super.clone() 调用的对象，那么生成的 Stack 实例在其 size 属性中具有正确的值，但 elements 属性引用与原始 Stack 实例相同的数组。 修改原始实例将破坏克隆中的不变量，反之亦然。 你会很快发现你的程序产生了无意义的结果，或者抛出 `NullPointerException` 异常。

这种情况永远不会发生，因为调用 Stack 类中的唯一构造方法。 实际上，clone 方法作为另一种构造方法; 必须确保它不会损坏原始对象，并且可以在克隆上正确建立不变量。 为了使 Stack 上的 clone 方法正常工作，它必须复制 stack 对象的内部。 最简单的方法是对元素数组递归调用 clone 方法：

```
// Clone method for class with references to mutable state
@Override public Stack clone() {
    try {
        Stack result = (Stack) super.clone();
        result.elements = elements.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

请注意，我们不必将 elements.clone 的结果转换为 Object[] 数组。 在数组上调用 clone 会返回一个数组，其运行时和编译时类型与被克隆的数组相同。 这是复制数组的首选习语。 事实上，数组是 clone 机制的唯一有力的用途。

还要注意，如果 elements 属性是 final 的，则以前的解决方案将不起作用，因为克隆将被禁止向该属性分配新的值。 这是一个基本的问题：像序列化一样，Cloneable 体系结构与引用可变对象的 final 属性的正常使用不兼容，除非可变对象可以在对象和其克隆之间安全地共享。 为了使一个类可以克隆，可能需要从一些属性中移除 final 修饰符。

仅仅递归地调用 clone 方法并不总是足够的。 例如，假设您正在为哈希表编写一个 clone 方法，其内部包含一个哈希桶数组，每个哈希桶都指向「键-值」对链表的第一项。 为了提高性能，该类实现了自己的轻量级单链表，而没有使用 java 内部提供的 `java.util.LinkedList`：

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;
    private static class Entry {
        final Object key;
        Object value;
        Entry  next;

        Entry(Object key, Object value, Entry next) {
            this.key   = key;
            this.value = value;
            this.next  = next;
        }
    }
    ... // Remainder omitted
}
```

假设你只是递归地克隆哈希桶数组，就像我们为 Stack 所做的那样：

```
// Broken clone method - results in shared mutable state!
@Override public HashTable clone() {
    try {
        HashTable result = (HashTable) super.clone();
        result.buckets = buckets.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

虽然被克隆的对象有自己的哈希桶数组，但是这个数组引用与原始数组相同的链表，这很容易导致克隆对象和原始对象中的不确定性行为。 要解决这个问题，你必须复制包含每个桶的链表。 下面是一种常见的方法：

```
// Recursive clone method for class with complex mutable state
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;

    private static class Entry {
        final Object key;
        Object value;
        Entry  next;

        Entry(Object key, Object value, Entry next) {
            this.key   = key;
            this.value = value;
            this.next  = next;
        }

        // Recursively copy the linked list headed by this Entry
        Entry deepCopy() {
            return new Entry(key, value,
                next == null ? null : next.deepCopy());
        }
    }

    @Override public HashTable clone() {
        try {
            HashTable result = (HashTable) super.clone();
            result.buckets = new Entry[buckets.length];
            for (int i = 0; i < buckets.length; i++)
                if (buckets[i] != null)
                    result.buckets[i] = buckets[i].deepCopy();
            return result;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
```

```
    }
    ... // Remainder omitted
}
```

私有类 HashTable.Entry 已被扩充以支持「深度复制」方法。 HashTable 上的 clone 方法分配一个合适大小的新哈希桶数组，迭代原来哈希桶数组，深度复制每个非空的哈希桶。 Entry 上的 deepCopy 方法递归地调用它自己以复制由头节点开始的整个链表。 如果哈希桶不是太长，这种技术很聪明并且工作正常。但是，克隆链表不是一个好方法，因为它为列表中的每个元素消耗一个栈帧（stack frame）。如果列表很长，这很容易导致堆栈溢出。 为了防止这种情况发生，可以用迭代来替换 deepCopy 中的递归：

```
// Iteratively copy the linked list headed by this Entry
Entry deepCopy() {
   Entry result = new Entry(key, value, next);
   for (Entry p = result; p.next != null; p = p.next)
      p.next = new Entry(p.next.key, p.next.value, p.next.next);
   return result;
}
```

克隆复杂可变对象的最后一种方法是调用 super.clone，将结果对象中的所有属性设置为其初始状态，然后调用更高级别的方法来重新生成原始对象的状态。 以 HashTable 为例，bucket 属性将被初始化为一个新的 bucket 数组，并且 put(key, value) 方法（未示出）被调用用于被克隆的哈希表中的键值映射。 这种方法通常产生一个简单，合理的优雅 clone 方法，其运行速度不如直接操纵克隆内部的方法快。 虽然这种方法是干净的，但它与整个 Cloneable 体系结构是对立的，因为它会盲目地重写构成体系结构基础的逐个属性对象复制。

与构造方法一样，clone 方法绝对不可以在构建过程中，调用一个可以重写的方法（详见第 19 条）。如果 clone 方法调用一个在子类中重写的方法，则在子类有机会在克隆中修复它的状态之前执行该方法，很可能导致克隆和原始对象的损坏。因此，我们在前面讨论的 put(key, value) 方法应该时 final 或 private 修饰的。（如果时 private 修饰，那么大概是一个非 final 公共方法的辅助方法）。

Object 类的 clone 方法被声明为抛出 CloneNotSupportedException 异常，但重写方法时不需要。公共 clone 方法应该省略 throws 子句，因为不抛出检查时异常的方法更容易使用（详见第 71 条）。

在为继承设计一个类时（详见第 19 条），通常有两种选择，但无论选择哪一种，都不应该实现 `Clonable` 接口。你可以选择通过实现正确运行的受保护的 clone 方法来模仿 Object 的行为，该方法声明为抛出 `CloneNotSupportedException` 异常。 这给了子类实现 `Cloneable` 接口的自由，就像直接继承 Object 一样。 或者，可以选择不实现工作的 clone 方法，并通过提供以下简并 clone 实现来阻止子类实现它：

```
// clone method for extendable class not supporting Cloneable
@Override
protected final Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}
```

还有一个值得注意的细节。 如果你编写一个实现了 Cloneable 的线程安全的类，记得它的 clone 方法必须和其他方法一样（详见第 78 条）需要正确的同步。 Object 类的 clone 方法是不同步的，所以即使它的实现是令人满意的，也可能需要编写一个返回 super.clone() 的同步 clone 方法。

回顾一下，实现 Cloneable 的所有类应该重写公共 clone 方法，而这个方法的返回类型是类本身。这个方法应该首先调用 super.clone，然后修复任何需要修复的属性。 通常，这意味着复制任何包含内部「深层结构」的可变对象，并用指向新对象的引用来代替原来指向这些对象的引用。虽然这些内部拷贝通常可以通过递归调用 clone 来实现，但这并不总是最好的方法。 如果类只包含基本类型或对不可变对象的引用，那么很可能是没有属性需要修复的情况。 这个规则也有例外。 例如，表示序列号或其他唯一 ID 的属性即使是基本类型的或不可变的，也需要被修正。

这么复杂是否真的有必要？很少。 如果你继承一个已经实现了 Cloneable 接口的类，你别无选择，只能实现一个行为良好的 clone 方法。 否则，通常你最好提供另一种对象复制方法。 对象复制更好的方法是提供一个复制构造方法或复制工厂。 复制构造方法接受参数，其类型为包含此构造方法的类，例如：

```
// Copy constructor
public Yum(Yum yum) { ... };
```

复制工厂类似于复制构造方法的静态工厂：

```
// Copy factory
public static Yum newInstance(Yum yum) { ... };
```

复制构造方法及其静态工厂变体与 Cloneable/clone 相比有许多优点：它们不依赖风险很大的语言外的对象创建机制；不要求遵守那些不太明确的惯例；不会与 final 属性的正确使用相冲突; 不会抛出不必要的检查异常; 而且不需要类型转换。

此外，复制构造方法或复制工厂可以接受类型为该类实现的接口的参数。 例如，按照惯例，所有通用集合实现都提供了一个构造方法，其参数的类型为 Collection 或 Map。 基于接口的复制构造方法和复制工厂（更适当地称为转换构造方法和转换工厂）允许客户端选择复制的实现类型，而不是强制客户端接受原始实现类型。 例如，假设你有一个 HashSet，并且你想把它复制为一个 TreeSet。 clone 方法不能提供这种功能，但使用转换构造方法很容易： `new TreeSet<>(s)` 。

考虑到与 Cloneable 接口相关的所有问题，新的接口不应该继承它，新的可扩展类不应该实现它。虽然实现 Cloneable 接口对于 final 类没有什么危害，但应该将其视为性能优化的角度，仅在极少数情况下才是合理的（详见第 67 条）。 通常，复制功能最好由构造方法或工厂提供。 这个规则的一个明显的例外是数组，它最好用 clone 方法复制。

14. 考虑实现 Comparable 接口

与本章讨论的其他方法不同，`compareTo` 方法并没有在 `Object` 类中声明。 相反，它是 `Comparable` 接口中的唯一方法。 它与 Object 类的 equals 方法在性质上是相似的，除了它允许在简单的相等比较之外的顺序比较，它是泛型的。 通过实现 `Comparable` 接口，一个类表明它的实例有一个自然顺序（natural ordering）。 对实现 `Comparable` 接口的对象数组排序非常简单，如下所示：

```
Arrays.sort(a);
```

它很容易查找，计算极端数值，以及维护 `Comparable` 对象集合的自动排序。例如，在下面的代码中，依赖于 String 类实现了 `Comparable` 接口，去除命令行参数输入重复的字符串，并按照字母顺序排序：

```
public class WordList {

    public static void main(String[] args) {
        Set<String> s = new TreeSet<>();
        Collections.addAll(s, args);
        System.out.println(s);
    }
}
```

通过实现 `Comparable` 接口，可以让你的类与所有依赖此接口的通用算法和集合实现进行互操作。 只需少量的努力就可以获得巨大的能量。 几乎 Java 平台类库中的所有值类以及所有枚举类型（详见第 34 条）都实现了 `Comparable` 接口。 如果你正在编写具有明显自然顺序（如字母顺序，数字顺序或时间顺序）的值类，则应该实现 `Comparable` 接口：

```
public interface Comparable<T> {
    int compareTo(T t);
}
```

`compareTo` 方法的通用约定与 `equals` 相似：

将此对象与指定的对象按照排序进行比较。 返回值可能为负整数，零或正整数，因为此对象对应小于，等于或大于指定的对象。 如果指定对象的类型与此对象不能进行比较，则引发 `ClassCastException` 异常。

下面的描述中，符号 sgn(expression) 表示数学中的 signum 函数，它根据表达式的值为负数、零、正数，对应返回-1、0 和 1。

- 实现类必须确保所有 `x` 和 `y` 都满足 `sgn(x.compareTo(y)) == -sgn(y. compareTo(x))`。（这意味着当且仅当 `y.compareTo(x)` 抛出异常时， `x.compareTo(y)` 必须抛出异常。）
- 实现类还必须确保该关系是可传递的： `(x. compareTo(y) > 0 && y.compareTo(z) > 0)` 意味着 `x.compareTo(z) > 0`。
- 最后，对于所有的 z，实现类必须确保 `x.compareTo(y) == 0` 意味着 `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`。
- 强烈推荐 `(x.compareTo(y) == 0) == (x.equals(y))`，但不是必需的。 一般来说，任何实现了 `Comparable` 接口的类违反了这个条件都应该清楚地说明这个事实。 推荐的语言是「注意：这个

类有一个自然顺序，与 `equals` 不一致」。

与 `equals` 方法一样，不要被上述约定的数学特性所退缩。这个约定并不像看起来那么复杂。 与 `equals` 方法不同， `equals` 方法在所有对象上施加了全局等价关系， `compareTo` 不必跨越不同类型的对象：当遇到不同类型的对象时， `compareTo` 被允许抛出 `ClassCastException` 异常。 通常，这正是它所做的。 约定确实允许进行不同类型间比较，这种比较通常在由被比较的对象实现的接口中定义。

正如一个违反 hashCode 约定的类可能会破坏依赖于哈希的其他类一样，违反 `compareTo` 约定的类可能会破坏依赖于比较的其他类。 依赖于比较的类，包括排序后的集合 `TreeSet` 和 TreeMap 类，以及包含搜索和排序算法的实用程序类 `Collections` 和 `Arrays` 。

我们来看看 `compareTo` 约定的规定。 第一条规定，如果反转两个对象引用之间的比较方向，则会发生预期的事情：如果第一个对象小于第二个对象，那么第二个对象必须大于第一个; 如果第一个对象等于第二个，那么第二个对象必须等于第一个; 如果第一个对象大于第二个，那么第二个必须小于第一个。 第二项约定说，如果一个对象大于第二个对象，而第二个对象大于第三个对象，则第一个对象必须大于第三个对象。 最后一条规定，所有比较相等的对象与任何其他对象相比，都必须得到相同的结果。

这三条规定的一个结果是， `compareTo` 方法所实施的平等测试必须遵守 equals 方法约定所施加的相同限制：自反性，对称性和传递性。 因此，同样需要注意的是：除非你愿意放弃面向对象抽象（详见第 10 条）的好处，否则无法在保留 `compareTo` 约定的情况下使用新的值组件继承可实例化的类。 同样的解决方法也适用。 如果要将值组件添加到实现 `Comparable` 的类中，请不要继承它；编写一个包含第一个类实例的不相关的类。 然后提供一个返回包含实例的「视图"方法。 这使你可以在包含类上实现任何 `compareTo` 方法，同时客户端在需要时，把包含类的实例视同以一个类的实例。

`compareTo` 约定的最后一段是一个强烈的建议，而不是一个真正的要求，只是声明 `compareTo` 方法施加的相等性测试，通常应该返回与 `equals` 方法相同的结果。 如果遵守这个约定，则 `compareTo` 方法施加的顺序被认为与 `equals` 相一致。 如果违反，顺序关系被认为与 `equals` 不一致。 其 `compareTo` 方法施加与 `equals` 不一致顺序关系的类仍然有效，但包含该类元素的有序集合可能不服从相应集合接口（Collection，Set 或 Map）的一般约定。 这是因为这些接口的通用约定是用 `equals` 方法定义的，但是排序后的集合使用 `compareTo` 强加的相等性测试来代替 `equals` 。 如果发生这种情况，虽然不是一场灾难，但仍是一件值得注意的事情。

例如，考虑 `BigDecimal` 类，其 `compareTo` 方法与 `equals` 不一致。 如果你创建一个空的 HashSet 实例，然后添加 `new BigDecimal("1.0")` 和 `new BigDecimal("1.00")` ，则该集合将包含两个元素，因为与 `equals` 方法进行比较时，添加到集合的两个 `BigDecimal` 实例是不相等的。 但是，如果使用 `TreeSet` 而不是 `HashSet` 执行相同的过程，则该集合将只包含一个元素，因为使用 `compareTo` 方法进行比较时，两个 BigDecimal 实例是相等的。 （有关详细信息，请参阅 BigDecimal 文档。）

编写 `compareTo` 方法与编写 `equals` 方法类似，但是有一些关键的区别。 因为 `Comparable` 接口是参数化的， `compareTo` 方法是静态类型的，所以你不需要输入检查或者转换它的参数。 如果参数是错误的类型，那么调用将不会编译。 如果参数为 null，则调用应该抛出一个 `NullPointerException` 异常，并且一旦该方法尝试访问其成员，它就会立即抛出这个异常。

在 `compareTo` 方法中，比较属性的顺序而不是相等。 要比较对象引用属性，请递归调用 `compareTo` 方法。 如果一个属性没有实现 Comparable，或者你需要一个非标准的顺序，那么使用 `Comparator` 接口。 可以编写自己的比较器或使用现有的比较器，如在条目 10 中的 `CaseInsensitiveString` 类的 `compareTo` 方法中：

```
// Single-field Comparable with object reference field
public final class CaseInsensitiveString
        implements Comparable<CaseInsensitiveString> {
    public int compareTo(CaseInsensitiveString cis) {
        return String.CASE_INSENSITIVE_ORDER.compare(s, cis.s);
    }
    ... // Remainder omitted
}
```

请注意，`CaseInsensitiveString` 类实现了 `Comparable<CaseInsensitiveString>` 接口。 这意味着 `CaseInsensitiveString` 引用只能与另一个 `CaseInsensitiveString` 引用进行比较。 当声明一个类来实现 `Comparable` 接口时，这是正常模式。

在本书第二版中，曾经推荐如果比较整型基本类型的属性，使用关系运算符「<」和 「>」，对于浮点类型基本类型的属性，使用 `Double.compare` 和 `Float.compare` 静态方法。在 Java 7 中，静态比较方法被添加到 Java 的所有包装类中。 在 `compareTo` 方法中使用关系运算符「<」和「>」是冗长且容易出错的，不再推荐。

如果一个类有多个重要的属性，那么比较他们的顺序是至关重要的。 从最重要的属性开始，逐步比较所有的重要属性。 如果比较结果不是零（零表示相等），则表示比较完成; 只是返回结果。 如果最重要的字段是相等的，比较下一个重要的属性，依此类推，直到找到不相等的属性或比较剩余不那么重要的属性。 以下是条目 11 中 `PhoneNumber` 类的 `compareTo` 方法，演示了这种方法：

```
// Multiple-field `Comparable` with primitive fields
public int compareTo(PhoneNumber pn) {
    int result = Short.compare(areaCode, pn.areaCode);
    if (result == 0) {
        result = Short.compare(prefix, pn.prefix);
        if (result == 0)
            result = Short.compare(lineNum, pn.lineNum);
    }
    return result;
}
```

在 Java 8 中 `Comparator` 接口提供了一系列比较器方法，可以使比较器流畅地构建。 这些比较器可以用来实现 `compareTo` 方法，就像 `Comparable` 接口所要求的那样。 许多程序员更喜欢这种方法的简洁性，尽管它的性能并不出众：在我的机器上排序 PhoneNumber 实例的数组速度慢了大约 10%。 在使用这种方法时，考虑使用 Java 的静态导入，以便可以通过其简单名称来引用比较器静态方法，以使其清晰简洁。 以下是 PhoneNumber 的 `compareTo` 方法的使用方法：

```
// Comparable with comparator construction methods
private static final Comparator<PhoneNumber> COMPARATOR =
        comparingInt((PhoneNumber pn) -> pn.areaCode)
          .thenComparingInt(pn -> pn.prefix)
          .thenComparingInt(pn -> pn.lineNum);

public int compareTo(PhoneNumber pn) {
    return COMPARATOR.compare(this, pn);
}
```

此实现在类初始化时构建比较器，使用两个比较器构建方法。第一个是 `comparingInt` 方法。它是一个静态方法，它使用一个键提取器函数式接口（key extractor function）作为参数，将对象引用映射为 int 类型的键，并返回一个根据该键排序的实例的比较器。在前面的示例中，`comparingInt` 方法使用 lambda 表达式，它从 `PhoneNumber` 中提取区域代码，并返回一个 `Comparator<PhoneNumber>`，根据它们的区域代码来排序电话号码。注意，lambda 表达式显式指定了其输入参数的类型 (PhoneNumber pn)。事实证明，在这种情况下，Java 的类型推断功能不够强大，无法自行判断类型，因此我们不得不帮助它以使程序编译。

如果两个电话号码实例具有相同的区号，则需要进一步细化比较，这正是第二个比较器构建方法，即 `thenComparingInt` 方法做的。 它是 `Comparator` 上的一个实例方法，接受一个 int 类型键提取器函数式接口（key extractor function）作为参数，并返回一个比较器，该比较器首先应用原始比较器，然后使用提取的键来打破连接。 你可以按照喜欢的方式多次调用 `thenComparingInt` 方法，从而产生一个字典顺序。 在上面的例子中，我们将两个调用叠加到 `thenComparingInt`，产生一个排序，它的二级键是 prefix，而其三级键是 lineNum。 请注意，我们不必指定传递给 `thenComparingInt` 的任何一个调用的键提取器函数式接口的参数类型：Java 的类型推断足够聪明，可以自己推断出参数的类型。

`Comparator` 类具有完整的构建方法。对于 long 和 double 基本类型，也有对应的类似于 comparingInt 和 `thenComparingInt` 的方法，int 版本的方法也可以应用于取值范围小于 int 的类型上，如 short 类型，如 PhoneNumber 实例中所示。对于 double 版本的方法也可以用在 float 类型上。这提供了所有 Java 的基本数字类型的覆盖。

也有对象引用类型的比较器构建方法。静态方法 `comparing` 有两个重载方式。第一个方法使用键提取器函数式接口并按键的自然顺序。第二种方法是键提取器函数式接口和比较器，用于键的排序。`thenComparing` 方法有三种重载。第一个重载只需要一个比较器，并使用它来提供一个二级排序。第二次重载只需要一个键提取器函数式接口，并使用键的自然顺序作为二级排序。最后的重载方法同时使用一个键提取器函数式接口和一个比较器来用在提取的键上。

有时，你可能会看到 `compareTo` 或 `compare` 方法依赖于两个值之间的差值，如果第一个值小于第二个值，则为负；如果两个值相等则为零，如果第一个值大于，则为正值。这是一个例子：

```
// BROKEN difference-based comparator - violates transitivity!

static Comparator<Object> hashCodeOrder = new Comparator<>() {
    public int compare(Object o1, Object o2) {
        return o1.hashCode() - o2.hashCode();
    }
};
```

不要使用这种技术！它可能会导致整数最大长度溢出和 IEEE 754 浮点运算失真的危险[JLS 15.20.1,15.21.1]。 此外，由此产生的方法不可能比使用上述技术编写的方法快得多。 使用静态 `compare` 方法：

```
// Comparator based on static compare method
static Comparator<Object> hashCodeOrder = new Comparator<>() {
    public int compare(Object o1, Object o2) {
        return Integer.compare(o1.hashCode(), o2.hashCode());
    }
};
```

或者使用 `Comparator` 的构建方法：

```
// Comparator based on Comparator construction method
static Comparator<Object> hashCodeOrder =
        Comparator.comparingInt(o -> o.hashCode());
```

总而言之，无论何时实现具有合理排序的值类，你都应该让该类实现 `Comparable` 接口，以便在基于比较的集合中轻松对其实例进行排序，搜索和使用。 比较 `compareTo` 方法的实现中的字段值时，请避免使用「<」和「>」运算符。 相反，使用包装类中的静态 `compare` 方法或 `Comparator` 接口中的构建方法。

15. 使类和成员的可访问性最小化

将设计良好的组件与设计不佳的组件区分开来的最重要的因素是，隐藏内部数据和其他实现细节的程度。一个设计良好的组件隐藏了它的所有实现细节，干净地将它的 API 与它的实现分离开来。然后，组件只通过它们的 API 进行通信，并且对彼此的内部工作一无所知。这一概念，被称为信息隐藏或封装，是软件设计的基本原则[Parnas72]。

信息隐藏很重要有很多原因，其中大部分来源于它将组成系统的组件分离开来，允许它们被独立地开发，测试，优化，使用，理解和修改。这加速了系统开发，因为组件可以并行开发。它减轻了维护的负担，因为可以更快速地理解组件，调试或更换组件，而不用担心损害其他组件。虽然信息隐藏本身并不会导致良好的性能，但它可以有效地进行性能调整：一旦系统完成并且分析确定了哪些组件导致了性能问题（条目 67），则可以优化这些组件，而不会影响别人的正确的组件。信息隐藏增加了软件重用，因为松耦合的组件通常在除开发它们之外的其他环境中证明是有用的。最后，隐藏信息降低了构建大型系统的风险，因为即使系统不能运行，各个独立的组件也可能是可用的。

Java 提供了许多机制来帮助信息隐藏。 访问控制机制（access control mechanism）[JLS，6.6] 指定了类，接口和成员的可访问性。 实体的可访问性取决于其声明的位置，以及声明中存在哪些访问修饰符（private，protected 和 public）。 正确使用这些修饰符对信息隐藏至关重要。

经验法则很简单：**让每个类或成员尽可能地不可访问。** 换句话说，使用尽可能低的访问级别，与你正在编写的软件的对应功能保持一致。

对于顶层（非嵌套的）类和接口，只有两个可能的访问级别：包级私有（package-private）和公共的（public）。如果你使用 public 修饰符声明顶级类或接口，那么它是公开的；否则，它是包级私有的。如果一个顶层类或接口可以被做为包级私有，那么它应该是。通过将其设置为包级私有，可以将其作为实现的一部分，而不是导出的 API，你可以修改它、替换它，或者在后续版本中消除它，而不必担心损害现有的客户端。如果你把它公开，你就有义务永远地支持它，以保持兼容性。

如果一个包级私有顶级类或接口只被一个类使用，那么可以考虑这个类作为使用它的唯一类的私有静态嵌套类 （详见第 24 条）。这将它的可访问性从包级的所有类减少到使用它的一个类。但是，减少不必要的公共类的可访问性要比包级私有的顶级类更重要：公共类是包的 API 的一部分，而包级私有的顶级类已经是这个包实现的一部分了。

对于成员（字段、方法、嵌套类和嵌套接口），有四种可能的访问级别，在这里，按照可访问性从小到大列出：

- private —— 该成员只能在声明它的顶级类内访问。
- package-private —— 成员可以从被声明的包中的任何类中访问。从技术上讲，如果没有指定访问修饰符（接口成员除外，它默认是公共的），这是默认访问级别。
- protected —— 成员可以从被声明的类的子类中访问（会受一些限制 [JLS, 6.6.2]），以及它声明的包中的任何类。
- public —— 该成员可以从任何地方被访问。

在仔细设计你的类的公共 API 之后，你的反应应该是让所有其他成员设计为私有的。 只有当同一个包中的其他类真的需要访问成员时，需要删除私有修饰符，从而使成员包成为包级私有的。 如果你发现自己经常这样做，你应该重新检查你的系统的设计，看看另一个分解可能产生更好的解耦的类。 也就是说，私有成员和包级私有成员都是类实现的一部分，通常不会影响其导出的 API。 但是，如果类实现 Serializable 接口（详见第 86 和 87 条），则这些字段可以「泄漏（leak）」到导出的 API 中。

对于公共类的成员，当访问级别从包私有到受保护级时，可访问性会大大增加。 受保护（protected）的成员是类导出的 API 的一部分，并且必须永远支持。 此外，导出类的受保护成员表示对实现细节的公开承诺（详见第 19 条）。 对受保护成员的需求应该相对较少。

有一个关键的规则限制了你减少方法访问性的能力。 如果一个方法重写一个超类方法，那么它在子类中的访问级别就不能低于父类中的访问级别[JLS，8.4.8.3]。 这对于确保子类的实例在父类的实例可用的地方是可用的（Liskov 替换原则，见条目 15）是必要的。 如果违反此规则，编译器将在尝试编译子类时生成错误消息。 这个规则的一个特例是，如果一个类实现了一个接口，那么接口中的所有类方法都必须在该类中声明为 public。

为了便于测试你的代码，你可能会想要让一个类，接口或者成员更容易被访问。 这没问题。 为了测试将公共类的私有成员指定为包级私有是可以接受的，但是提高到更高的访问级别却是不可接受的。换句话说，将类，接口或成员作为包级导出的 API 的一部分来促进测试是不可接受的。 幸运的是，这不是必须的，因为测试可以作为被测试包的一部分运行，从而获得对包私有元素的访问。

公共类的实例字段很少情况下采用 public 修饰（详见第 16 条）。 如果一个实例字段是非 final 的，或者是对可变对象的引用，那么通过将其公开，你就放弃了限制可以存储在字段中的值的能力。这意味着你放弃了执行涉及该字段的不变量的能力。另外，当字段被修改时，就放弃了采取任何操作的能力，**因此带有公共可变字段的类通常不是线程安全的** 。即使一个字段是 final 的，并且引用了一个不可变的对象，通过将其公开，你放弃了切换到一个新的内部数据表示的灵活性，而该字段并不存在。

同样的建议适用于静态字段，但有一个例外。 假设常量是类的抽象的一个组成部分，你可以通过 public static final 字段暴露常量。 按照惯例，这些字段的名字由大写字母组成，字母用下划线分隔（详见第 68 条）。 很重要的一点是，这些字段包含基本类型的值或对不可变对象的引用（详见第 17 条）。 包含对可变对象的引用的字段具有非 final 字段的所有缺点。 虽然引用不能被修改，但引用的对象可以被修改，并会带来灾难性的结果。

请注意，非零长度的数组总是可变的，**所以类具有公共静态 final 数组字段，或返回这样一个字段的访问器是错误的。** 如果一个类有这样的字段或访问方法，客户端将能够修改数组的内容。 这是安全漏洞的常见来源：

```
// Potential security hole!
public static final Thing[] VALUES = { ... };
```

要小心这样的事实，一些 IDE 生成的访问方法返回对私有数组字段的引用，导致了这个问题。 有两种方法可以解决这个问题。 你可以使公共数组私有并添加一个公共的不可变列表：

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final List<Thing> VALUES =
Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

或者，可以将数组设置为 private，并添加一个返回私有数组拷贝的公共方法：

```
private static final Thing[] PRIVATE_VALUES = { ... };

public static final Thing[] values() {
    return PRIVATE_VALUES.clone();
}
```

要在这些方法之间进行选择，请考虑客户端可能如何处理返回的结果。 哪种返回类型会更方便？哪个会更好的表现？

在 Java 9 中，作为模块系统（module system）的一部分引入了两个额外的隐式访问级别。模块包含一组包，就像一个包包含一组类一样。模块可以通过模块声明中的导出（export）声明显式地导出某些包 (这是 module-info.java 的源文件中包含的约定)。模块中的未导出包的公共和受保护成员在模块之外是不可访问的；在模块中，可访问性不受导出（export）声明的影响。使用模块系统允许你在模块之间共享类，而不让它们对整个系统可见。在未导出的包中，公共和受保护的公共类的成员会产生两个隐式访问级别，这是普通公共和受保护级别的内部类似的情况。这种共享的需求是相对少见的，并且可以通过重新安排包中的类来消除。

与四个主要访问级别不同，这两个基于模块的级别主要是建议（advisory）。 如果将模块的 JAR 文件放在应用程序的类路径而不是其模块路径中，那么模块中的包将恢复为非模块化行为：包的公共类的所有公共类和受保护成员都具有其普通的可访问性，不管包是否由模块导出[Reinhold，1.2]。 新引入的访问级别严格执行的地方是 JDK 本身：Java 类库中未导出的包在模块之外真正无法访问。

对于典型的 Java 程序员来说，不仅程序模块所提供的访问保护存在局限性，而且在本质上是很大程度上建议性的；为了利用它，你必须把你的包组合成模块，在模块声明中明确所有的依赖关系，重新安排你的源码树层级，并采取特殊的行动来适应你的模块内任何对非模块化包的访问[Reinhold, 3]。 现在说模块是否会在 JDK 之外得到广泛的使用还为时尚早。 与此同时，除非你有迫切的需要，否则似乎最好避免它们。

总而言之，应该尽可能地减少程序元素的可访问性（在合理范围内）。 在仔细设计一个最小化的公共 API 之后，你应该防止任何散乱的类，接口或成员成为 API 的一部分。 除了作为常量的公共静态 `final` 字段之外，公共类不应该有公共字段。 确保 `public static final` 字段引用的对象是不可变的。

16. 在公共类中使用访问方法而不是公共属性

有时候，你可能会试图写一些退化的类（degenerate classes），除了集中实例属性之外别无用处：

```
// Degenerate classes like this should not be public!
class Point {
    public double x;
    public double y;
}
```

由于这些类的数据属性可以直接被访问，因此这些类不提供封装的好处（详见第 15 条）。 如果不更改 API，则无法更改其表示形式，无法强制执行不变量，并且在访问属性时无法执行辅助操作。 坚持面向对象的程序员觉得这样的类是厌恶的，应该被具有私有属性和公共访问方法的类（getter）所取代，而对于可变类来说，它们应该被替换为 setter 设值方法：

```
// Encapsulation of data by accessor methods and mutators
class Point {
    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() { return x; }

    public double getY() { return y; }

    public void setX(double x) { this.x = x; }

    public void setY(double y) { this.y = y; }
```

```
}
```

当然，对于公共类来说，坚持面向对象是正确的：**如果一个类在其包之外是可访问的，则提供访问方法来保留更改类内部表示的灵活性。** 如果一个公共类暴露其数据属性，那么以后更改其表示形式基本上没有可能，因为客户端代码可以散布在很多地方。

但是，**如果一个类是包级私有的，或者是一个私有的内部类，那么暴露它的数据属性就没有什么本质上的错误——假设它们提供足够描述该类提供的抽象。** 在类定义和使用它的客户端代码中，这种方法比访问方法产生更少的视觉混乱。 虽然客户端代码绑定到类的内部表示，但是这些代码仅限于包含该类的包。 如果类的内部表示是可取的，可以在不触碰包外的任何代码的情况下进行更改。 在私有内部类的情况下，更改作用范围进一步限制在封闭类中。

Java 平台类库中的几个类违反了公共类不应直接暴露属性的建议。 著名的例子包括 `java.awt` 包中的 `Point` 和 `Dimension` 类。 这些类别应该被视为警示性的示例，而不是模仿的例子。 如条目 67 所述，时至今日，暴露 `Dimension` 的内部结构的决定仍然导致着严重的性能问题。

虽然公共类直接暴露属性并不是一个好主意，但是如果属性是不可变的，那么危害就不那么大了。 当一个属性是只读的时候，除了更改类的 API 外，你不能改变类的内部表示形式，也不能采取一些辅助的行为，但是可以加强不变性。例如，下面的例子中保证每个实例表示一个有效的时间：

```
// Public class with exposed immutable fields - questionable

public final class Time {
    private static final int HOURS_PER_DAY    = 24;
    private static final int MINUTES_PER_HOUR = 60;
    public final int hour;
    public final int minute;

    public Time(int hour, int minute) {
        if (hour < 0 || hour >= HOURS_PER_DAY)
           throw new IllegalArgumentException("Hour: " + hour);
        if (minute < 0 || minute >= MINUTES_PER_HOUR)
           throw new IllegalArgumentException("Min: " + minute);
        this.hour = hour;
        this.minute = minute;
    }

    ... // Remainder omitted
}
```

总之，公共类不应该暴露可变属性。 公共类暴露不可变属性的危害虽然仍然存在问题，但其危害较小。 然而，有时需要包级私有或私有内部类来暴露属性，无论此类是否是可变的。

17. 最小化可变性

不可变类简单来说是其实例不能被修改的类。 包含在每个实例中的所有信息在对象的生命周期中是固定的，因此不会观察到任何变化。 Java 平台类库包含许多不可变的类，包括 `String` 类、基本类型包装类以及 `BigInteger` 类和 `BigDecimal` 类。 有很多很好的理由：不可变类比可变类更易于设计，实现和使用。 他们不容易出错，并且更安全。

要使一个类成为不可变类，请遵循以下五条规则：

1. **不要提供修改对象状态的方法（也称为 mutators，设值方法）。**
2. **确保这个类不能被继承。** 这可以防止粗心或者恶意的子类假装对象的状态已经改变，从而破坏类的不可变行为。 防止子类化，通常是通过 `final` 修饰类，但是我们稍后将讨论另一种方法。
3. **把所有字段设置为 final。** 通过系统强制执行的方式，清楚地表达了你的意图。 另外，如果一个新创建的实例的引用在缺乏同步机制的情况下从一个线程传递到另一个线程，就必须保证正确的行为，正如内存模型[JLS，17.5; Goetz06 16] 所述。
4. **把所有的字段设置为 private。** 这可以防止客户端获得对字段引用的可变对象的访问权限，并直接修改这些对象。 虽然技术上允许不可变类具有包含基本类型数值的公有的 `final` 字段或对不可变对象的引用，但不建议这样做，因为这样使得在以后的版本中无法再改变内部的表示状态（详见第 15 和 16 条）。
5. **确保对任何可变组件的互斥访问。** 如果你的类有任何引用可变对象的字段，请确保该类的客户端无法获得对这些对象的引用。 切勿将这样的属性初始化为客户端提供的对象引用，或从访问方法返回属性。 在构造方法，访问方法和 `readObject` 方法（详见第 88 条）中进行防御性拷贝（详见第 50 条）。

以前条目中的许多示例类都是不可变的。 其中一个例子是条目 11 中的 `PhoneNumber` 类，它具有每个字段的访问方法（accessors），但没有相应的设值方法（mutators）。下面一个稍微复杂一点的例子：

```
// Immutable complex number class
public final class Complex {

    private final double re;
    private final double im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public double realPart() {
        return re;
    }

    public double imaginaryPart() {
        return im;
    }

    public Complex plus(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
```

```
    public Complex minus(Complex c) {
        return new Complex(re - c.re, im - c.im);
    }

    public Complex times(Complex c) {
        return new Complex(re * c.re - im * c.im,
                re * c.im + im * c.re);
    }

    public Complex dividedBy(Complex c) {
        double tmp = c.re * c.re + c.im * c.im;
        return new Complex((re * c.re + im * c.im) / tmp,
                (im * c.re - re * c.im) / tmp);
    }

    @Override
    public boolean equals(Object o) {
        if (o == this) {
            return true;
        }

        if (!(o instanceof Complex)) {
            return false;
        }

        Complex c = (Complex) o;

        // See page 47 to find out why we use compare instead of ==
        return Double.compare(c.re, re) == 0
                && Double.compare(c.im, im) == 0;
    }

    @Override
    public int hashCode() {
        return 31 * Double.hashCode(re) + Double.hashCode(im);
    }

    @Override
    public String toString() {
        return "(" + re + " + " + im + "i)";
    }
}
```

这个类代表了一个复数（包含实部和虚部的数字）。 除了标准的 `Object` 方法之外，它还为实部和虚部提供访问方法，并提供四个基本的算术运算：加法，减法，乘法和除法。 注意算术运算如何创建并返回一个新的 `Complex` 实例，而不是修改这个实例。 这种模式被称为函数式方法，因为方法返回将操作数应用于函数的结果，而不修改它们。 与其对应的过程式的（procedural）或命令式的

（imperative）的方法相对比，在这种方法中，将一个过程作用在操作数上，导致其状态改变。 请注意，方法名称是介词（如 plus）而不是动词（如 add）。 这强调了方法不会改变对象的值的事实。 `BigInteger` 和 `BigDecimal` 类没有遵守这个命名约定，并导致许多使用错误。

如果你不熟悉函数式方法，可能会觉得它显得不自然，但它具有不变性，具有许多优点。 **不可变对象很简单。** 一个不可变的对象可以完全处于一种状态，也就是被创建时的状态。 如果确保所有的构造方法都建立了类不变量，那么就保证这些不变量在任何时候都保持不变，使用此类的程序员无需再做额外的工作。 另一方面，可变对象可以具有任意复杂的状态空间。 如果文档没有提供由设置（mutator）方法执行的状态转换的精确描述，那么可靠地使用可变类可能是困难的或不可能的。

不可变对象本质上是线程安全的；它们不需要同步。 被多个线程同时访问它们时，不会遭到破坏。 这是实现线程安全的最简单方法。 由于没有线程可以观察到另一个线程对不可变对象的影响，所以**不可变对象可以被自由地共享。** 因此，不可变类应鼓励客户端尽可能重用现有的实例。 一个简单的方法是为常用的值提供公共的静态 final 常量。 例如， `Complex` 类可能提供这些常量：

```
public static final Complex ZERO = new Complex(0, 0);
public static final Complex ONE  = new Complex(1, 0);
public static final Complex I    = new Complex(0, 1);
```

这种方法可以更进一步。 一个不可变的类可以提供静态的工厂（详见第 1 条）来缓存经常被请求的实例，以避免在现有的实例中创建新的实例。 所有基本类型的包装类和 `BigInteger` 类都是这样做的。 使用这样的静态工厂会使客户端共享实例而不是创建新实例，从而减少内存占用和垃圾回收成本。 在设计新类时，选择静态工厂代替公共构造方法，可以在以后增加缓存的灵活性，而不需要修改客户端。

不可变对象可以自由分享的结果是，你永远不需要做出防御性拷贝（defensive copies）（详见第 50 条）。 事实上，永远不需要做任何拷贝，因为这些拷贝永远等于原始对象。 因此，你不需要也不应该在一个不可变的类上提供一个 clone 方法或拷贝构造方法（copy constructor）（详见第 13 条）。 这一点在 Java 平台的早期阶段还不是很好理解，所以 `String` 类有一个拷贝构造方法，但是它应该尽量很少使用（详见第 6 条）。

不仅可以共享不可变的对象，而且可以共享内部信息。 例如， `BigInteger` 类在内部使用符号数值表示法。 符号用 `int` 值表示，数值用 `int` 数组表示。 `negate` 方法生成了一个数值相同但符号相反的新 `BigInteger` 实例。 即使它是可变的，也不需要复制数组；新创建的 `BigInteger` 指向与原始相同的内部数组。

不可变对象为其他对象提供了很好的构件（building blocks），无论是可变的还是不可变的。 如果知道一个复杂组件的内部对象不会发生改变，那么维护复杂对象的不变性就容易多了。不可变对象是优秀的映射键和集合元素是这一原则的重要例子: 一旦不可变对象作为 `Map` 的键或 `Set` 里的元素，你就不需要担心 `Map` 或 `Set` 的不变性被这些对象的值的变化所破坏。

不可变对象无偿地提供了的原子失败机制（详见第 76 条）。 它们的状态永远不会改变，所以不可能出现临时的不一致。

不可变类的主要缺点是对于每个不同的值都需要一个单独的对象。 创建这些对象可能代价很高，特别是是大型的对象下。 例如，假设你有一个百万位的 `BigInteger` ，你想改变它的低位：

```
BigInteger moby = ...;
moby = moby.flipBit(0);
```

`flipBit` 方法创建一个新的 `BigInteger` 实例，也是一百万位长，与原始位置只有一位不同。该操作需要与 `BigInteger` 大小成比例的时间和空间。 将其与 `java.util.BitSet` 对比。 像 `BigInteger` 一样， `BitSet` 表示一个任意长度的位序列，但与 `BigInteger` 不同， `BitSet` 是可变的。 `BitSet` 类提供了一种方法，允许你在固定时间内更改百万位实例中单个位的状态：

```
BitSet moby = ...;
moby.flip(0);
```

如果执行一个多步操作，在每一步生成一个新对象，除最终结果之外丢弃所有对象，则性能问题会被放大。这里有两种方式来处理这个问题。第一种办法，先猜测一下会经常用到哪些多步的操作，然后讲它们作为基本类型提供。如果一个多步操作是作为一个基本类型提供的，那么不可变类就不必在每一步创建一个独立的对象。在内部，不可变的类可以是任意灵活的。 例如， `BigInteger` 有一个包级私有的可变的“伙伴类（companion class）”，它用来加速多步操作，比如模幂运算（modular exponentiation）。出于前面所述的所有原因，使用可变伙伴类比使用 `BigInteger` 要困难得多。 幸运的是，你不必使用它： `BigInteger` 类的实现者为你做了很多努力。

如果你可以准确预测客户端要在你的不可变类上执行哪些复杂的操作，那么包级私有可变伙伴类的方式可以正常工作。如果不是的话，那么最好的办法就是提供一个公开的可变伙伴类。 这种方法在 Java 平台类库中的主要例子是 `String` 类，它的可变伙伴类是 `StringBuilder` （及其过时的前身 `StringBuffer` 类）。

现在你已经知道如何创建一个不可改变类，并且了解不变性的优点和缺点，下面我们来讨论几个设计方案。 回想一下，为了保证不变性，一个类不得允许子类化。 这可以通过使类用 `final` 修饰，但是还有另外一个更灵活的选择。 而不是使不可变类设置为 `final` ，可以使其所有的构造方法私有或包级私有，并添加公共静态工厂，而不是公共构造方法（详见第 1 条）。 为了具体说明这种方法，下面以 `Complex` 为例，看看如何使用这种方法：

```
// Immutable class with static factories instead of constructors
public class Complex {

    private final double re;
    private final double im;

    private Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public static Complex valueOf(double re, double im) {
        return new Complex(re, im);
    }

    ... // Remainder unchanged
```

```
}
```

这种方法往往是最好的选择。 这是最灵活的，因为它允许使用多个包级私有实现类。 对于驻留在包之外的客户端，不可变类实际上是 `final` 的，因为不可能继承来自另一个包的类，并且缺少公共或受保护的构造方法。 除了允许多个实现类的灵活性以外，这种方法还可以通过改进静态工厂的对象缓存功能来调整后续版本中类的性能。

当 `BigInteger` 和 `BigDecimal` 刚被编写出来的时候，“不可变类必须是 `final` ”的说法还没有得到广泛地理解，因此它们的所有方法都可能被重写。不幸的是，为了保持向后兼容性，这一问题无法得以纠正。如果你编写一个安全性取决于来自不受信任的客户端的 `BigInteger` 或 `BigDecimal` 参数的不变类时，则必须检查该参数是否为“真实的” `BigInteger` 或者 `BigDecimal` ，而不应该是不受信任的子类的实例。如果是后者，则必须在假设可能是可变的情况下保护性拷贝（defensively copy）（详见第 50 条）：

```
public static BigInteger safeInstance(BigInteger val) {
    return val.getClass() == BigInteger.class ?
            val : new BigInteger(val.toByteArray());
}
```

在本条目开头关于不可变类的规则说明，没有方法可以修改对象，并且它的所有属性必须是 `final` 的。事实上，这些规则比实际需要的要强硬一些，其实可以有所放松来提高性能。 事实上，任何方法都不能在对象的状态中产生外部可见的变化。 然而，一些不可变类具有一个或多个非 `final` 属性，在第一次需要时将开销昂贵的计算结果缓存在这些属性中。 如果再次请求相同的值，则返回缓存的值，从而节省了重新计算的成本。 这个技巧的作用恰恰是因为对象是不可变的，这保证了如果重复的话，计算会得到相同的结果。

例如， `PhoneNumber` 类的 `hashCode` 方法（详见第 11 条）在第一次调用改方法时计算哈希码，并在再次调用时 对其进行缓存。 这种延迟初始化（详见第 83 条）的一个例子，String 类也使用到了。

关于序列化应该加上一个警告。 如果你选择使您的不可变类实现 `Serializable` 接口，并且它包含一个或多个引用可变对象的属性，则必须提供显式的 `readObject` 或 `readResolve` 方法，或者使用 `ObjectOutputStream.writeUnshared` 和 `ObjectInputStream.readUnshared` 方法，即默认的序列化形式也是可以接受的。 否则攻击者可能会创建一个可变的类的实例。 这个主题会在条目 88 中会详细介绍。

总而言之，坚决不要为每个属性编写一个 get 方法后再编写一个对应的 set 方法。 **除非有充分的理由使类成为可变类，否则类应该是不可变的。** 不可变类提供了许多优点，唯一的缺点是在某些情况下可能会出现性能问题。 你应该始终使用较小的值对象（如 `PhoneNumber` 和 `Complex` ），使其不可变。 （Java 平台类库中有几个类，如 `java.util.Date` 和 `java.awt.Point` ，本应该是不可变的，但实际上并不是）。你应该认真考虑创建更大的值对象，例如 `String` 和 `BigInteger` ，设成不可改变的。 只有当你确认有必要实现令人满意的性能（详见第 67 条）时，才应该为不可改变类提供一个公开的可变伙伴类。

对于一些类来说，不变性是不切实际的。**如果一个类不能设计为不可变类，那么也要尽可能地限制它的可变性** 。减少对象可以存在的状态数量，可以更容易地分析对象，以及降低出错的可能性。因此，除非有足够的理由把属性设置为非 `final` 的情况下，否则应该每个属性都设置为 `final` 的。把本条目的建议与条目 15 的建议结合起来，你自然的倾向就是：**除非有充分的理由不这样做，否则应该把每**

个属性声明为私有 final 的。

构造方法应该创建完全初始化的对象，并建立所有的不变性。 除非有令人信服的理由，否则不要提供独立于构造方法或静态工厂的公共初始化方法。 同样，不要提供一个“reinitialize”方法，使对象可以被重用，就好像它是用不同的初始状态构建的。 这样的方法通常以增加的复杂度为代价，仅仅提供很少的性能优势。

`CountDownLatch` 类是这些原理的例证。 它是可变的，但它的状态空间有意保持最小范围内。 创建一个实例，使用它一次，并完成：一旦 `countdown` 锁的计数器已经达到零，不能再重用它。

在这个条目中，应该添加关于 `Complex` 类的最后一个注释。 这个例子只是为了说明不变性。 这不是一个工业强度复杂的复数实现。 它对复数使用了乘法和除法的标准公式，这些公式不正确会进行不正确的四舍五入，没有为复数的 `NaN` 和无穷大提供良好的语义[Kahan91，Smith62，Thomas94]。

18. 组合优于继承

继承是实现代码重用的有效方式，但并不总是最好的工具。使用不当，会导致脆弱的软件。 在包中使用继承是安全的，其中子类和父类的实现都在同一个程序员的控制之下。对应专门为了继承而设计的，并且有文档说明的类来说（详见第 19 条），使用继承也是安全的。 然而，从普通的具体类跨越包级边界继承，是危险的。 提醒一下，本书使用「继承」一词，其含义是*实现继承*（当一个类扩展另一个类时）。 在本条目中讨论的问题不适用于*接口继承*（当类实现接口，或者当接口继承另一个接口时）。

与方法调用不同，继承打破了封装[Snyder86]。 换句话说，一个子类依赖于其父类的实现细节来保证其正确的功能。 父类的实现可能会从发布版本不断变化，如果是这样，子类可能会被破坏，即使它的代码没有任何改变。 因此，一个子类必须与其超类一起更新而变化，除非父类的作者为了继承的目的而专门设计它，并对应有文档的说明。

为了具体说明，假设有一个使用 `HashSet` 的程序。 为了调整程序的性能，需要查询 `HashSet` ，从创建它之后已经添加了多少个元素（不要和当前的元素数量混淆，当元素被删除时数量也会下降）。 为了提供这个功能，编写了一个 `HashSet` 变体，它保留了尝试元素插入的数量，并导出了这个插入数量的一个访问方法。 `HashSet` 类包含两个添加元素的方法，分别是 `add` 和 `addAll` ，所以我们重写这两个方法：

```
// Broken - Inappropriate use of inheritance!
public class InstrumentedHashSet<E> extends HashSet<E> {
    // The number of attempted element insertions
    private int addCount = 0;

    public InstrumentedHashSet() {
    }

    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }
    @Override public boolean add(E e) {
```

```
        addCount++;
        return super.add(e);
    }
    @Override public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
}
```

这个类看起来很合理，但是不能正常工作。 假设创建一个实例并使用 `addAll` 方法添加三个元素。 顺便提一句，请注意，下面代码使用在 Java 9 中添加的静态工厂方法 `List.of` 来创建一个列表；如果使用的是早期版本，请改为使用 `Arrays.asList`：

```
InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
s.addAll(List.of("Snap", "Crackle", "Pop"));
```

我们期望 getAddCount 方法返回的结果是 3，但实际上返回了 6。哪里出来问题？在 `HashSet` 内部， `addAll` 方法是基于它的 `add` 方法来实现的，即使 `HashSet` 文档中没有指名其实现细节，倒也是合理的。 `InstrumentedHashSet` 中的 `addAll` 方法首先给 `addCount` 属性设置为 3，然后使用 `super.addAll` 方法调用了 `HashSet` 的 `addAll` 实现。然后反过来又调用在 `InstrumentedHashSet` 类中重写的 `add` 方法，每个元素调用一次。这三次调用又分别给 `addCount` 加 1，所以，一共增加了 6：通过 `addAll` 方法每个增加的元素都被计算了两次。

我们可以通过消除 `addAll` 方法的重写来“修复”子类。 尽管生成的类可以正常工作，但是它依赖于它的正确方法，因为 `HashSet` 的 `addAll` 方法是在其 `add` 方法之上实现的。 这个“自我使用（self-use）”是一个实现细节，并不保证在 `Java` 平台的所有实现中都可以适用，并且可以随发布版本而变化。 因此，产生的 `InstrumentedHashSet` 类是脆弱的。

稍微好一点的做法是，重写 `addAll` 方法遍历指定集合，为每个元素调用 `add` 方法一次。 不管 `HashSet` 的 `addAll` 方法是否在其 `add` 方法上实现，都会保证正确的结果，因为 `HashSet` 的 `addAll` 实现将不再被调用。然而，这种技术并不能解决所有的问题。 这相当于重新实现了父类方法，这样的方法可能不能确定是否是自用（self-use）的，实现起来也是困难的，耗时的，容易出错的，并且可能会降低性能。 此外，这种方式并不能总是奏效，因为子类无法访问一些私有属性，所以有些方法就无法实现。

导致子类脆弱的一个相关原因是，它们的父类在后续的发布版本中可以添加新的方法。假设一个程序的安全性依赖于这样一个事实：所有被插入到集中的元素都满足一个先决条件。可以通过对集合进行子类化，然后并重写所有添加元素的方法，以确保在添加每个元素之前满足这个先决条件，来确保这一问题。如果在后续的版本中，父类没有新增添加元素的方法，那么这样做没有问题。但是，一旦父类增加了这样的新方法，则很有可能由于调用了未被重写的新方法，将非法的元素添加到子类的实例中。这不是个纯粹的理论问题。在把 `Hashtable` 和 `Vector` 类加入到 `Collections` 框架中的时候，就修复了几个类似性质的安全漏洞。

这两个问题都源于重写方法。 如果仅仅添加新的方法并且不要重写现有的方法，可能会认为继承一个类是安全的。 虽然这种扩展更为安全，但这并非没有风险。 如果父类在后续版本中添加了一个新的方法，并且你不幸给了子类一个具有相同签名和不同返回类型的方法，那么你的子类编译失败[JLS，8.4.8.3]。 如果已经为子类提供了一个与新的父类方法具有相同签名和返回类型的方法，那么你现在正在重写它，因此将遇到前面所述的问题。 此外，你的方法是否会履行新的父类方法的约定，这是值得怀疑的，因为在你编写子类方法时，这个约定还没有写出来。

幸运的是，有一种方法可以避免上述所有的问题。不要继承一个现有的类，而应该给你的新类增加一个私有属性，该属性是 现有类的实例引用，这种设计被称为组合（composition），因为现有的类成为新类的组成部分。新类中的每个实例方法调用现有类的包含实例上的相应方法并返回结果。这被称为转发（forwarding），而新类中的方法被称为转发方法。由此产生的类将坚如磐石，不依赖于现有类的实现细节。即使将新的方法添加到现有的类中，也不会对新类产生影响。为了具体说用，下面代码使用组合和转发方法替代 `InstrumentedHashSet` 类。请注意，实现分为两部分，类本身和一个可重用的转发类，其中包含所有的转发方法，没有别的方法：

```
// Reusable forwarding class
import java.util.Collection;
import java.util.Iterator;
import java.util.Set;

public class ForwardingSet<E> implements Set<E> {

    private final Set<E> s;

    public ForwardingSet(Set<E> s) {
        this.s = s;
    }

    public void clear() {
        s.clear();
    }

    public boolean contains(Object o) {
        return s.contains(o);
    }

    public boolean isEmpty() {
        return s.isEmpty();
    }

    public int size() {
        return s.size();
    }

    public Iterator<E> iterator() {
        return s.iterator();
    }
```

```
    public boolean add(E e) {
        return s.add(e);
    }

    public boolean remove(Object o) {
        return s.remove(o);
    }

    public boolean containsAll(Collection<?> c) {
        return s.containsAll(c);
    }

    public boolean addAll(Collection<? extends E> c) {
        return s.addAll(c);
    }

    public boolean removeAll(Collection<?> c) {
        return s.removeAll(c);
    }

    public boolean retainAll(Collection<?> c) {
        return s.retainAll(c);
    }

    public Object[] toArray() {
        return s.toArray();
    }

    public <T> T[] toArray(T[] a) {
        return s.toArray(a);
    }

    @Override
    public boolean equals(Object o) {
        return s.equals(o);
    }

    @Override
    public int hashCode() {
        return s.hashCode();
    }

    @Override
    public String toString() {
        return s.toString();
    }
}
```

```
// Wrapper class - uses composition in place of inheritance
import java.util.Collection;
import java.util.Set;

public class InstrumentedSet<E> extends ForwardingSet<E> {

    private int addCount = 0;

    public InstrumentedSet(Set<E> s) {
        super(s);
    }

    @Override public boolean add(E e) {
        addCount++;
        return super.add(e);
    }

    @Override public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}
```

`InstrumentedSet` 类的设计是通过存在的 `Set` 接口来实现的，该接口包含 `HashSet` 类的功能特性。除了功能强大，这个设计是非常灵活的。 `InstrumentedSet` 类实现了 `Set` 接口，并有一个构造方法，其参数也是 `Set` 类型的。本质上，这个类把 `Set` 转换为另一个类型 `Set` ， 同时添加了计数的功能。与基于继承的方法不同，该方法仅适用于单个具体类，并且父类中每个需要支持构造方法，提供单独的构造方法，所以可以使用包装类来包装任何 `Set` 实现，并且可以与任何预先存在的构造方法结合使用：

```
Set<Instant> times = new InstrumentedSet<>(new TreeSet<>(cmp));
Set<E> s = new InstrumentedSet<>(new HashSet<>(INIT_CAPACITY));
```

`InstrumentedSet` 类甚至可以用于临时替换没有计数功能下使用的集合实例：

```
static void walk(Set<Dog> dogs) {
    InstrumentedSet<Dog> iDogs = new InstrumentedSet<>(dogs);
    ... // Within this method use iDogs instead of dogs
}
```

`InstrumentedSet` 类被称为包装类，因为每个 `InstrumentedSet` 实例都包含（“包装”）另一个 Set 实例。 这也被称为装饰器模式[Gamma95]，因为 `InstrumentedSet` 类通过添加计数功能来“装饰”一个集合。 有时组合和转发的结合被不精确地地称为委托（delegation）。 从技术上讲，除非包装对象把自身传递给被包装对象，否则不是委托[Lieberman86; Gamma95]。

包装类的缺点很少。 一个警告是包装类不适合在回调框架（callback frameworks）中使用，其中对象将自我引用传递给其他对象以用于后续调用（「回调」）。 因为一个被包装的对象不知道它外面的包装对象，所以它传递一个指向自身的引用（this），回调时并不记得外面的包装对象。 这被称为 SELF 问题[Lieberman86]。 有些人担心转发方法调用的性能影响，以及包装对象对内存占用。 两者在实践中都没有太大的影响。 编写转发方法有些繁琐，但是只需为每个接口编写一次可重用的转发类，并且提供转发类。 例如， `Guava` 为所有的 `Collection` 接口提供转发类[Guava]。

只有在子类真的是父类的子类型的情况下，继承才是合适的。 换句话说，只有在两个类之间存在「is-a」关系的情况下，B 类才能继承 A 类。 如果你试图让 B 类继承 A 类时，问自己这个问题：每个 B 都是 A 吗？ 如果你不能明确的以“是的”来回答这个问题，那么 B 就不应该继承 A。如果答案是否定的，那么 B 通常包含一个 A 的私有实例，并且暴露一个不同的 API ： A 不是 B 的重要部分 ， 只是其实现细节。

在 Java 平台类库中有一些明显的违反这个原则的情况。 例如， `stacks` 实例并不是 `vector` 实例，所以 `Stack` 类不应该继承 `Vector` 类。 同样，一个属性列表不是一个哈希表，所以 `Properties` 不应该继承 `Hashtable` 类。 在这两种情况下，组合方式更可取。

如果在合适组合的地方使用继承，则会不必要地公开实现细节。由此产生的 API 将与原始实现联系在一起，永远限制类的性能。更严重的是，通过暴露其内部，客户端可以直接访问它们。至少，它可能导致混淆语义。例如，属性 `p` 指向 `Properties` 实例，那么 `p.getProperty(key)` 和 `p.get(key)` 就有可能返回不同的结果：前者考虑了默认的属性表，而后者是继承 `Hashtable` 的，它则没有考虑默认属性列表。最严重的是，客户端可以通过直接修改超父类来破坏子类的不变性。在 `Properties` 类，设计者希望只有字符串被允许作为键和值，但直接访问底层的 `Hashtable` 允许违反这个不变性。一旦违反，就不能再使用属性 API 的其他部分（ `load` 和 `store` 方法）。在发现这个问题的时候，纠正这个问题为时已晚，因为客户端依赖于使用非字符串键和值了。

在决定使用继承来代替组合之前，你应该问自己最后一组问题。对于试图继承的类，它的 API 有没有缺陷呢？ 如果有，你是否愿意将这些缺陷传播到你的类的 API 中？继承传播父类的 API 中的任何缺陷，而组合可以让你设计一个隐藏这些缺陷的新 API。

总之，继承是强大的，但它是有问题的，因为它违反封装。 只有在子类和父类之间存在真正的子类型关系时才适用。 即使如此，如果子类与父类不在同一个包中，并且父类不是为继承而设计的，继承可能会导致脆弱性。 为了避免这种脆弱性，使用组合和转发代替继承，特别是如果存在一个合适的接口来实现包装类。 包装类不仅比子类更健壮，而且更强大。

19. 要么设计继承并提供文档说明，要么禁用继承

条目 18 中提醒你注意继承没有设计和文档说明的「外来」类的子类化的危险。 那么对于专门为了继承而设计并且具有良好文档说明的类而言，这又意味着什么呢?

首先，这个类必须准确地描述重写每个方法带来的影响。 换句话说，该类必须文档说明可重写方法的自用性（self-use）。 对于每个 public 或者 protected 的方法，文档必须指明方法调用哪些可重写方法，以何种顺序调用的，以及每次调用的结果又是如何影响后续处理。 （重写方法，这里是指非 `final` 修饰的方法，无论是公开还是保护的。）更一般地说，一个类必须文档说明任何可能调用可重写方法的情况。 例如，后台线程或者静态初始化代码块可能会调用这样的方法。

调用可重写方法的方法在文档注释结束时包含对这些调用的描述。 这些描述在规范中特定部分，标记为「Implementation Requirements」，由 Javadoc 标签 `@implSpec` 生成。 这段话介绍该方法的内部工作原理。 下面是从 `java.util.AbstractCollection` 类的规范中拷贝的例子：

```
public boolean remove(Object o)

Removes a single instance of the specified element from this collection, if it is
present (optional operation). More formally, removes an element e such that
Objects.equals(o, e), if this collection contains one or more such elements. Returns
true if this collection contained the specified element (or equivalently, if this
collection changed as a result of the call).

Implementation Requirements: This implementation iterates over the collection looking
for the specified element. If it finds the element, it removes the element from the
collection using the iterator’s remove method. Note that this implementation throws an
UnsupportedOperationException if the iterator returned by this collection’s iterator
method does not implement the remove method and this collection contains the specified
object.
```

从该集合中删除指定元素的单个实例（如果存在，optional 实例操作）。 更广义地说，如果这个集合包含一个或多个这样的元素 e，就删除其中的一个满足 `Objects.equals(o, e)` 的元素 e。 如果此集合包含指定的元素（或者等同于此集合因调用而发生了更改），则返回 true。

实现要求： 这个实现迭代遍历集合查找指定元素。 如果找到元素，则使用迭代器的 `remove` 方法从集合中删除元素。 请注意，如果此集合的 `iterator` 方法返回的迭代器未实现 `remove` 方法，并且此集合包含指定的对象，则该实现将引发 `UnsupportedOperationException` 异常。

这个文档清楚地说明，重写 `iterator` 方法将会影响 `remove` 方法的行为。 它还描述了 `iterator` 方法返回的 `Iterator` 行为将如何影响 `remove` 方法的行为。 与条目 18 中的情况相反，在这种情况下，程序员继承 `HashSet` 并不能说明重写 `add` 方法是否会影响 `addAll` 方法的行为。

关于程序文档有句格言：好的 API 文档应该描述一个给定的方法做了什么工作，而不是描述它是如何做到的。那么，上面这种做法是否违背了这句格言呢？是的，它确实违背了！这正是继承破坏了封装性所带来的不幸后果。所以，为了设计一个类的文档，以便它能够被安全地子类化，你必须描述清楚那些有可能未定义的实现细节。

`@implSpec` 标签是在 Java 8 中添加的，并且在 Java 9 中被大量使用。这个标签应该默认启用，但是从 Java 9 开始，除非通过命令行开关 `-tag "apiNote:a:API Note:"`，否则 Javadoc 工具仍然会忽略它。

为了继承而进行的设计不仅仅涉及自用模式的文档设计。为了使程序员能够编写出更加有效的子类，而无须承受不必要的痛苦，**类必须以精心挑选的 protected 方法的形式，提供适当的钩子（hook），以便进入其内部工作中**。或者在罕见的情况下，提供受保护的属性。 例如，考虑 `java.util.AbstractList` 中的 `removeRange` 方法：

```
protected void removeRange(int fromIndex, int toIndex)

Removes from this list all of the elements whose index is between fromIndex,
inclusive, and toIndex, exclusive. Shifts any succeeding elements to the left (reduces
their index). This call shortens the list by (toIndex - fromIndex) elements. (If
toIndex == fromIndex, this operation has no effect.)

This method is called by the clear operation on this list and its sublists. Overriding
this method to take advantage of the internals of the list implementation can
substantially improve the performance of the clear operation on this list and its
sublists.

Implementation Requirements: This implementation gets a list iterator positioned
before fromIndex and repeatedly calls ListIterator.nextfollowed by
ListIterator.remove, until the entire range has been removed. Note: If
ListIterator.remove requires linear time, this implementation requires quadratic time.

Parameters:
    fromIndex    index of first element to be removed.
    toIndex      index after last element to be removed.
```

从此列表中删除索引介于 `fromIndex` （包含）和 `inclusive` （不含）之间的所有元素。 将任何后续元素向左移（减少索引）。 这个调用通过（ `toIndex - fromIndex` ）元素来缩短列表。 （如果 `toIndex == fromIndex` ，则此操作无效。）

这个方法是通过列表及其子类的 `clear` 操作来调用的。重写这个方法利用列表内部实现的优势，可以大大提高列表和子类的 `clear` 操作性能。

实现要求： 这个实现获取一个列表迭代器，它位于 `fromIndex` 之前，并重复调用 `ListIterator.remove` 和 `ListIterator.next` 方法，直到整个范围被删除。 **注意：如果 `ListIterator.remove` 需要线性时间，则此实现需要平方级时间。**

参数：

fromIndex 要移除的第一个元素的索引

toIndex 要移除的最后一个元素之后的索引

这个方法对 `List` 实现的最终用户来说是没有意义的。 它仅仅是为了使子类很容易提供一个快速 `clear` 方法。 在没有 `removeRange` 方法的情况下，当在子列表上调用 `clear` 方法，子类将不得不使用平方级的时间，否则，或从头重写整个 `subList` 机制——这不是一件容易的事情！

那么当你设计一个继承类的时候，你如何决定暴露哪些的受保护的成员呢？ 不幸的是，没有灵丹妙药。 所能做的最好的就是努力思考，做出最好的测试，然后通过编写子类来进行测试。 应该尽可能少地暴露受保护的成员，因为每个成员都表示对实现细节的承诺。 另一方面，你不能暴露太少，因为失去了保护的成员会导致一个类几乎不能用于继承。

测试为继承而设计的类的唯一方法是编写子类。 如果你忽略了一个关键的受保护的成员，试图编写一个子类将会使得遗漏痛苦地变得明显。 相反，如果编写的几个子类，而且没有一个使用受保护的成员，那么应该将其设为私有。 经验表明，三个子类通常足以测试一个可继承的类。 这些子类应该由父类作者以外的人编写。

当你为继承设计一个可能被广泛使用的类的时候，要意识到你永远承诺你文档说明的自用模式以及隐含在其保护的方法和属性中的实现决定。 这些承诺可能会使后续版本中改善类的性能或功能变得困难或不可能。 因此， **在发布它之前，你必须通过编写子类来测试你的类。**

另外，请注意，继承所需的特殊文档混乱了正常的文档，这是为创建类的实例并在其上调用方法的程序员设计的。 在撰写本文时，几乎没有工具将普通的 API 文档从和仅仅针对子类实现的信息，分离出来。

还有一些类必须遵守允许继承的限制。 **构造方法绝不能直接或间接调用可重写的方法。** 如果违反这个规则，将导致程序失败。 父类构造方法在子类构造方法之前运行，所以在子类构造方法运行之前，子类中的重写方法被调用。 如果重写方法依赖于子类构造方法执行的任何初始化，则此方法将不会按预期运行。 为了具体说明，这是一个违反这个规则的类：

```
public class Super {
    // Broken - constructor invokes an overridable method
    public Super() {
        overrideMe();
    }
    public void overrideMe() {
    }
}
```

以下是一个重写 `overrideMe` 方法的子类， `Super` 类的唯一构造方法会错误地调用它：

```
public final class Sub extends Super {
    // Blank final, set by constructor
    private final Instant instant;

    Sub() {
        instant = Instant.now();
    }

    // Overriding method invoked by superclass constructor
    @Override
    public void overrideMe() {
        System.out.println(instant);
    }

    public static void main(String[] args) {
```

```
        Sub sub = new Sub();
        sub.overrideMe();
    }
}
```

你可能期望这个程序打印两次 `instant` 实例，但是它第一次打印出 `null`，因为在 `Sub` 构造方法有机会初始化 `instant` 属性之前，`overrideMe` 被 `Super` 构造方法调用。 请注意，这个程序观察两个不同状态的 `final` 属性！ 还要注意的是，如果 `overrideMe` 方法调用了 instant 实例中任何方法，那么当父类构造方法调用 `overrideMe` 时，它将抛出一个 `NullPointerException` 异常。 这个程序不会抛出 `NullPointerException` 的唯一原因是 `println` 方法容忍 `null` 参数。

请注意，从构造方法中调用私有方法，其中任何一个方法都不可重写的，那么 `final` 方法和静态方法是安全的。

`Cloneable` 和 `Serializable` 接口在设计继承时会带来特殊的困难。 对于为继承而设计的类来说，实现这些接口通常不是一个好主意，因为这会给继承类的程序员带来很大的负担。 然而，可以采取特殊的行动来允许子类实现这些接口，而不需要强制这样做。 这些操作在条目 13 和条目 86 中有描述。

如果你决定在为继承而设计的类中实现 `Cloneable` 或 `Serializable` 接口，那么应该知道，由于 `clone` 和 `readObject` 方法与构造方法相似，所以也有类似的限制：**`clone` 和 `readObject` 都不会直接或间接调用可重写的方法。** 在 `readObject` 的情况下，重写方法将在子类的状态被反序列化之前运行。 在 `clone` 的情况下，重写方法将在子类的 `clone` 方法有机会修复克隆的状态之前运行。 在任何一种情况下，都可能会出现程序故障。 在 `clone` 的情况下，故障可能会损坏原始对象以及被克隆对象本身。 例如，如果重写方法假定它正在修改对象的深层结构的拷贝，但是尚未创建拷贝，则可能发生这种情况。

最后，如果你决定在为继承设计的类中实现 `Serializable` 接口，并且该类有一个 `readResolve` 或 `writeReplace` 方法，则必须使 `readResolve` 或 `writeReplace` 方法设置为受保护而不是私有。 如果这些方法是私有的，它们将被子类无声地忽略。 这是另一种情况，把实现细节成为类的 API 的一部分，以允许继承。

到目前为止，**设计一个继承类需要很大的努力，并且对这个类有很大的限制。** 这不是一个轻率的决定。 有些情况显然是正确的，比如抽象类，包括接口的骨架实现（skeletal implementations）（详见第 20 条）。 还有其他的情况显然是错误的，比如不可变的类（详见第 17 条）。

但是普通的具体类呢？ 传统上，它们既不是 `final` 的，也不是为了子类化而设计和文档说明的，但是这种情况是危险的。每次修改这样的类，则继承此类的子类将被破坏。 这不仅仅是一个理论问题。在修改非 `final` 的具体类的内部之后，接收与子类相关的错误报告并不少见，这些类没有为继承而设计和文档说明。

解决这个问题的最佳方法是禁止对在设计上和文档说明中都不支持安全子类化的类进行子类化。 这有两种方法禁止子类化。 两者中较容易的是声明类为 `final`。 另一种方法是使所有的构造方法都是私有的或包级私有的，并且添加公共静态工厂来代替构造方法。 这个方案在内部提供了使用子类的灵活性，在条目 17 中讨论过。两种方法都是可以接受的。

这个建议可能有些争议，因为许多程序员已经习惯于继承普通的具体类来增加功能，例如通知和同步等功能，或限制原有类的功能。 如果一个类实现了捕获其本质的一些接口，比如 `Set` ， `List` 或 `Map` ，那么不应该为了禁止子类化而感到愧疚。 在条目 18 中描述的包装类模式为增强功能提供了继承的优越选择。

如果一个具体的类没有实现一个标准的接口，那么你禁止继承可能给一些程序员带来不便。 如果你觉得你必须允许从这样的类继承，一个合理的方法是确保类从不调用任何可重写的方法，并文档说明这个事实。 换句话说，完全消除类的自用（self-use）的可重写的方法。 这样做，你将创建一个合理安全的子类。 重写一个方法不会影响任何其他方法的行为。

你可以机械地消除类的自我使用的重写方法，而不会改变其行为。 将每个可重写的方法的主体移动到一个私有的“帮助器方法”，并让每个可重写的方法调用其私有的帮助器方法。 然后用直接调用可重写方法的专用帮助器方法来替换每个自用的可重写方法。

简而言之，专门为了继承而设计类是一件很辛苦的工作。你必须建立文档说明其所有的自用模式，并且一旦建立了文档，在这个类的整个生命周期中都必须遵守。如果没有做到，子类就会依赖父类的实现细节，如果父类的实现发生了变化，它就有可能遭到破坏。为了允许其他人能编写出高效的子类，你还必须暴露一个或者多个受保护的方法。除非意识到真的需要子类，否则最好通过将类声明为 `final` ，或者确保没有可访问的构造器来禁止类被继承。

20. 接口优于抽象类

Java 有两种机制来定义允许多个实现的类型：接口和抽象类。 由于在 Java 8 [JLS 9.4.3] 中引入了接口的默认方法（default methods ），因此这两种机制都允许为某些实例方法提供实现。 一个主要的区别是要实现由抽象类定义的类型，类必须是抽象类的子类。 因为 Java 只允许单一继承，所以对抽象类的这种限制严格限制了它们作为类型定义的使用。 任何定义所有必需方法并服从通用约定的类都可以实现一个接口，而不管类在类层次结构中的位置。

现有的类可以很容易地进行改进来实现一个新的接口。 你只需添加所需的方法（如果尚不存在的话），并向类声明中添加一个 `implements` 子句。 例如，当 `Comparable` ， `Iterable` ， 和 `Autocloseable` 接口添加到 Java 平台时，很多现有类需要实现它们来加以改进。 一般来说，现有的类不能改进以继承一个新的抽象类。 如果你想让两个类继承相同的抽象类，你必须把它放在类型层级结构中的上面位置，它是两个类的祖先。 不幸的是，这会对类型层级结构造成很大的附带损害，迫使新的抽象类的所有后代对它进行子类化，无论这些后代类是否合适。

接口是定义混合类型（mixin）的理想选择。 一般来说，mixin 是一个类，除了它的“主类型”之外，还可以声明它提供了一些可选的行为。 例如， `Comparable` 是一个类型接口，它允许一个类声明它的实例相对于其他可相互比较的对象是有序的。 这样的接口被称为类型，因为它允许可选功能被“混合”到类型的主要功能。 抽象类不能用于定义混合类，这是因为它们不能被加载到现有的类中：一个类不能有多个父类，并且在类层次结构中没有合理的位置来插入一个类型。

接口允许构建非层级类型的框架。 类型层级对于组织某些事物来说是很好的，但是其他的事物并不是整齐地落入严格的层级结构中。 例如，假设我们有一个代表歌手的接口，和另一个代表作曲家的接口：

```
public interface Singer {
    AudioClip sing(Song s);
}

public interface Songwriter {
    Song compose(int chartPosition);
}
```

在现实生活中，一些歌手也是作曲家。 因为我们使用接口而不是抽象类来定义这些类型，所以单个类实现歌手和作曲家两个接口是完全允许的。 事实上，我们可以定义一个继承歌手和作曲家的第三个接口，并添加适合于这个组合的新方法：

```
public interface SingerSongwriter extends Singer, Songwriter {
    AudioClip strum();
    void actSensitive();
}
```

你并不总是需要这种灵活性，但是当你这样做的时候，接口是一个救星。 另一种方法是对于每个受支持的属性组合，包含一个单独的类的臃肿类层级结构。 如果类型系统中有 n 个属性，则可能需要支持 2n 种可能的组合。 这就是所谓的组合爆炸（combinatorial explosion）。 臃肿的类层级结构可能会导致具有许多方法的臃肿类，这些方法仅在参数类型上有所不同，因为类层级结构中没有类型来捕获通用行为。

接口通过包装类模式确保安全的，强大的功能增强成为可能（详见第 18 条）。 如果使用抽象类来定义类型，那么就让程序员想要添加功能，只能继承。 生成的类比包装类更弱，更脆弱。

当其他接口方法有明显的接口方法实现时，可以考虑向程序员提供默认形式的方法实现帮助。 有关此技术的示例，请参阅第 104 页的 `removeIf` 方法。如果提供默认方法，请确保使用 `@implSpec` Javadoc 标记（条目 19）将它们文档说明为继承。

使用默认方法可以提供实现帮助多多少少是有些限制的。 尽管许多接口指定了 `Object` 类中方法（如 `equals` 和 `hashCode`）的行为，但不允许为它们提供默认方法。 此外，接口不允许包含实例属性或非公共静态成员（私有静态方法除外）。 最后，不能将默认方法添加到不受控制的接口中。

但是，你可以通过提供一个抽象的骨架实现类（abstract skeletal implementation class）来与接口一起使用，将接口和抽象类的优点结合起来。 接口定义了类型，可能提供了一些默认的方法，而骨架实现类在原始接口方法的顶层实现了剩余的非原始接口方法。 继承骨架实现需要大部分的工作来实现一个接口。 这就是模板方法设计模式[Gamma95]。

按照惯例，骨架实现类被称为 `AbstractInterface`，其中 `Interface` 是它们实现的接口的名称。 例如，集合框架（ Collections Framework）提供了一个框架实现以配合每个主要集合接口：`AbstractCollection`， `AbstractSet`， `AbstractList` 和 `AbstractMap`。 可以说，将它们称为 `SkeletalCollection`， `SkeletalSet`， `SkeletalList` 和 `SkeletalMap` 是有道理的，但是现在已经确立了抽象约定。 如果设计得当，骨架实现（无论是单独的抽象类还是仅由接口上的默认方法组成）可以使程序员非常容易地提供他们自己的接口实现。 例如，下面是一个静态工厂方法，在 `AbstractList` 的顶层包含一个完整的功能齐全的 `List` 实现：

```
// Concrete implementation built atop skeletal implementation
static List<Integer> intArrayAsList(int[] a) {
    Objects.requireNonNull(a);
    // The diamond operator is only legal here in Java 9 and later
    // If you're using an earlier release, specify <Integer>
    return new AbstractList<>() {
        @Override
        public Integer get(int i) {
            return a[i];  // Autoboxing ([Item 6]
(https://www.safaribooksonline.com/library/view/effective-java-
third/9780134686097/ch2.xhtml#lev6))
        }

        @Override
        public Integer set(int i, Integer val) {
            int oldVal = a[i];
            a[i] = val;     // Auto-unboxing
            return oldVal;  // Autoboxing
        }

        @Override
        public int size() {
            return a.length;
        }
    };

}
```

当你考虑一个 `List` 实现为你做的所有事情时，这个例子是一个骨架实现的强大的演示。 顺便说一句，这个例子是一个适配器（Adapter）[Gamma95]，它允许一个 `int` 数组被看作 `Integer` 实例列表。 由于 `int` 值和整数实例（装箱和拆箱）之间的来回转换，其性能并不是非常好。 请注意，实现采用匿名类的形式（详见第 24 条）。

骨架实现类的优点在于，它们提供抽象类的所有实现的帮助，而不会强加抽象类作为类型定义时的严格约束。对于具有骨架实现类的接口的大多数实现者来说，继承这个类是显而易见的选择，但它不是必需的。如果一个类不能继承骨架的实现，这个类可以直接实现接口。该类仍然受益于接口本身的任何默认方法。此外，骨架实现类仍然可以协助接口的实现。实现接口的类可以将接口方法的调用转发给继承骨架实现的私有内部类的包含实例。这种被称为模拟多重继承的技术与条目 18 讨论的包装类模式密切相关。它提供了多重继承的许多好处，同时避免了缺陷。

编写一个骨架的实现是一个相对简单的过程，虽然有些乏味。 首先，研究接口，并确定哪些方法是基本的，其他方法可以根据它们来实现。 这些基本方法是你的骨架实现类中的抽象方法。 接下来，为所有可以直接在基本方法之上实现的方法提供接口中的默认方法，回想一下，你可能不会为诸如 `Object` 类中 `equals` 和 `hashCode` 等方法提供默认方法。 如果基本方法和默认方法涵盖了接口，那么就完成了，并且不需要骨架实现类。 否则，编写一个声明实现接口的类，并实现所有剩下的接口方法。 为了适合于该任务，此类可能包含任何的非公共属性和方法。

作为一个简单的例子，考虑一下 `Map.Entry` 接口。 显而易见的基本方法是 `getKey` ， `getValue` 和（可选的） `setValue` 。 接口指定了 `equals` 和 `hashCode` 的行为，并且在基本方面方面有一个 `toString` 的明显的实现。 由于不允许为 `Object` 类方法提供默认实现，因此所有实现均放置在骨架实现类中：

```
// Skeletal implementation class
public abstract class AbstractMapEntry<K,V>
        implements Map.Entry<K,V> {

    // Entries in a modifiable map must override this method
    @Override public V setValue(V value) {
        throw new UnsupportedOperationException();
    }

    // Implements the general contract of Map.Entry.equals
    @Override
    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry<?,?> e = (Map.Entry) o;
        return Objects.equals(e.getKey(),  getKey())
            && Objects.equals(e.getValue(), getValue());
    }

    // Implements the general contract of Map.Entry.hashCode
    @Override
    public int hashCode() {
        return Objects.hashCode(getKey())
             ^ Objects.hashCode(getValue());
    }

    @Override
    public String toString() {
        return getKey() + "=" + getValue();
    }

}
```

请注意，这个骨架实现不能在 `Map.Entry` 接口中实现，也不能作为子接口实现，因为默认方法不允许重写诸如 `equals` ， `hashCode` 和 `toString` 等 `Object` 类方法。

由于骨架实现类是为了继承而设计的，所以你应该遵循条目 19 中的所有设计和文档说明。为了简洁起见，前面的例子中省略了文档注释，但是好的文档在骨架实现中是绝对必要的，无论它是否包含 一个接口或一个单独的抽象类的默认方法。

与骨架实现有稍许不同的是简单实现，以 `AbstractMap.SimpleEntry` 为例。 一个简单的实现就像一个骨架实现，它实现了一个接口，并且是为了继承而设计的，但是它的不同之处在于它不是抽象的：它是最简单的工作实现。 你可以按照情况使用它，也可以根据情况进行子类化。

总而言之，一个接口通常是定义允许多个实现的类型的最佳方式。 如果你导出一个重要的接口，应该强烈考虑提供一个骨架的实现类。 在可能的情况下，应该通过接口上的默认方法提供骨架实现，以便接口的所有实现者都可以使用它。 也就是说，对接口的限制通常要求骨架实现类采用抽象类的形式。

21. 为后代设计接口

在 Java 8 之前，不可能在不破坏现有实现的情况下为接口添加方法。 如果向接口添加了一个新方法，现有的实现通常会缺少该方法，从而导致编译时错误。 在 Java 8 中，添加了默认方法（default method）构造[JLS 9.4]，目的是允许将方法添加到现有的接口。 但是增加新的方法到现有的接口是充满风险的。

默认方法的声明包含一个默认实现，该方法允许实现接口的类直接使用，而不必实现默认方法。 虽然在 Java 中添加默认方法可以将方法添加到现有接口，但不能保证这些方法可以在所有已有的实现中使用。 默认的方法被「注入（injected）」到现有的实现中，没有经过实现类的知道或同意。 在 Java 8 之前，这些实现是用默认的接口编写的，它们的接口永远不会获得任何新的方法。

许多新的默认方法被添加到 Java 8 的核心集合接口中，主要是为了方便使用 lambda 表达式（第 6 章）。 Java 类库的默认方法是高质量的通用实现，在大多数情况下，它们工作正常。 **但是，编写一个默认方法并不总是可能的，它保留了每个可能的实现的所有不变量。**

例如，考虑在 Java 8 中添加到 `Collection` 接口的 `removeIf` 方法。此方法删除给定布尔方法（或 `Predicate` 函数式接口）返回 `true` 的所有元素。默认实现被指定为使用迭代器遍历集合，调用每个元素的谓词，并使用迭代器的 `remove` 方法删除谓词返回 `true` 的元素。 据推测，这个声明看起来像这样：默认实现被指定为使用迭代器遍历集合，调用每个元素的 `Predicate` 函数式接口，并使用迭代器的 `remove` 方法删除 `Predicate` 函数式接口返回 `true` 的元素。 根据推测，这个声明看起来像这样：

```
// Default method added to the Collection interface in Java 8
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean result = false;
    for (Iterator<E> it = iterator(); it.hasNext(); ) {
        if (filter.test(it.next())) {
            it.remove();
            result = true;
        }
    }
    return result;
}
```

这是可能为 `removeIf` 方法编写的最好的通用实现，但遗憾的是，它在一些实际的 `Collection` 实现中失败了。 例如，考虑 `org.apache.commons.collections4.collection.SynchronizedCollection` 方法。 这个类出自 `Apache Commons` 类库中，与 `java.util` 包中的静态工厂 `Collections.synchronizedCollection` 方法返回的类相似。 Apache 版本还提供了使用客户端提供的对象进行锁定的能力，以代替集合。 换句话说，它是一个包装类（条目 18），它们的所有方法在委托给包装集合类之前在一个锁定对象上进行同步。

Apache 的 `SynchronizedCollection` 类仍然在积极维护，但在撰写本文时，并未重写 `removeIf` 方法。 如果这个类与 Java 8 一起使用，它将继承 `removeIf` 的默认实现，但实际上不能保持类的基本承诺：自动同步每个方法调用。 默认实现对同步一无所知，并且不能访问包含锁定对象的属性。 如果客户端在另一个线程同时修改集合的情况下调用 `SynchronizedCollection` 实例上的 `removeIf` 方法，则可能会导致 `ConcurrentModificationException` 异常或其他未指定的行为。

为了防止在类似的 Java 平台类库实现中发生这种情况，比如 `Collections.synchronizedCollection` 返回的包级私有的类，JDK 维护者必须重写默认的 `removeIf` 实现和其他类似的方法来在调用默认实现之前执行必要的同步。 原来不属于 Java 平台的集合实现没有机会与接口更改进行类似的改变，有些还没有这样做。

在默认方法的情况下，接口的现有实现类可以在没有错误或警告的情况下编译，但在运行时会失败。 虽然不是非常普遍，但这个问题也不是一个孤立的事件。 在 Java 8 中添加到集合接口的一些方法已知是易受影响的，并且已知一些现有的实现会受到影响。

应该避免使用默认方法向现有的接口添加新的方法，除非这个需要是关键的，在这种情况下，你应该仔细考虑，以确定现有的接口实现是否会被默认的方法实现所破坏。然而，默认方法对于在创建接口时提供标准的方法实现非常有用，以减轻实现接口的任务（详见第 20 条）。

还值得注意的是，默认方法不是被用来设计，来支持从接口中移除方法或者改变现有方法的签名的目的。在不破坏现有客户端的情况下，这些接口都不可能发生更改。

准则是清楚的。 尽管默认方法现在是 Java 平台的一部分，**但是非常悉心地设计接口仍然是非常重要的。** 虽然默认方法可以将方法添加到现有的接口，但这样做有很大的风险。 如果一个接口包含一个小缺陷，可能会永远惹怒用户。 如果一个接口严重缺陷，可能会破坏包含它的 API。

因此，在发布之前测试每个新接口是非常重要的。 多个程序员应该以不同的方式实现每个接口。 至少，你应该准备三种不同的实现。 编写多个使用每个新接口的实例来执行各种任务的客户端程序同样重要。 这将大大确保每个接口都能满足其所有的预期用途。 这些步骤将允许你在发布之前发现接口中的缺陷，但仍然可以轻松地修正它们。 **虽然在接口被发布后可能会修正一些存在的缺陷，但不要太指望这一点。**

22. 接口仅用来定义类型

当类实现接口时，该接口作为一种类型（type），可以用来引用类的实例。因此，一个类实现了一个接口，因此表明客户端可以如何处理类的实例。为其他目的定义接口是不合适的。

一种失败的接口就是所谓的常量接口（constant interface）。 这样的接口不包含任何方法; 它只包含静态 final 属性，每个输出一个常量。 使用这些常量的类实现接口，以避免需要用类名限定常量名。 这里是一个例子：

```
// Constant interface antipattern - do not use!
public interface PhysicalConstants {
    // Avogadro's number (1/mol)
    static final double AVOGADROS_NUMBER   = 6.022_140_857e23;

    // Boltzmann constant (J/K)
    static final double BOLTZMANN_CONSTANT = 1.380_648_52e-23;

    // Mass of the electron (kg)
    static final double ELECTRON_MASS      = 9.109_383_56e-31;
}
```

常量接口模式是对接口的糟糕使用。 类在内部使用一些常量，完全属于实现细节。实现一个常量接口会导致这个实现细节泄漏到类的导出 API 中。对类的用户来说，类实现一个常量接口是没有意义的。事实上，它甚至可能使他们感到困惑。更糟糕的是，它代表了一个承诺：如果在将来的版本中修改了类，不再需要使用常量，那么它仍然必须实现接口，以确保二进制兼容性。如果一个非 final 类实现了常量接口，那么它的所有子类的命名空间都会被接口中的常量所污染。

Java 平台类库中有多个常量接口，如 `java.io.ObjectStreamConstants`。 这些接口应该被视为不规范的，不应该被效仿。

如果你想导出常量，有几个合理的选择方案。 如果常量与现有的类或接口紧密相关，则应将其添加到该类或接口中。 例如，所有数字基本类型的包装类，如 `Integer` 和 `Double`，都会导出 `MIN_VALUE` 和 `MAX_VALUE` 常量。 如果常量最好被看作枚举类型的成员，则应该使用枚举类型（详见第 34 条）导出它们。 否则，你应该用一个不可实例化的工具类来导出常量（详见第 4 条）。 下是前面所示的 `PhysicalConstants` 示例的工具类的版本：

```
// Constant utility class
package com.effectivejava.science;

public class PhysicalConstants {
  private PhysicalConstants() { }  // Prevents instantiation

  public static final double AVOGADROS_NUMBER = 6.022_140_857e23;
  public static final double BOLTZMANN_CONST  = 1.380_648_52e-23;
  public static final double ELECTRON_MASS    = 9.109_383_56e-31;
}
```

顺便提一下，请注意在数字文字中使用下划线字符（_）。 从 Java 7 开始，合法的下划线对数字字面量的值没有影响，但是如果使用得当的话可以使它们更容易阅读。 无论是固定的浮点数，如果他们包含五个或更多的连续数字，考虑将下划线添加到数字字面量中。 对于底数为 10 的数字，无论是整型还是浮点型的，都应该用下划线将数字分成三个数字组，表示一千的正负幂。

通常，实用工具类要求客户端使用类名来限定常量名，例如 `PhysicalConstants.AVOGADROS_NUMBER`。 如果大量使用实用工具类导出的常量，则通过使用静态导入来限定具有类名的常量：

```
// Use of static import to avoid qualifying constants
import static com.effectivejava.science.PhysicalConstants.*;

public class Test {
    double  atoms(double mols) {
        return AVOGADROS_NUMBER * mols;
    }
    ...
    // Many more uses of PhysicalConstants justify static import
}
```

总之，接口只能用于定义类型。 它们不应该仅用于导出常量。

23. 类层次结构优于标签类

有时你可能会碰到一个类，它的实例有两个或更多的风格，并且包含一个标签字段（tag field），表示实例的风格。 例如，考虑这个类，它可以表示一个圆形或矩形：

```
// Tagged class - vastly inferior to a class hierarchy!
class Figure {
    enum Shape { RECTANGLE, CIRCLE };

    // Tag field - the shape of this figure
    final Shape shape;

    // These fields are used only if shape is RECTANGLE
    double length;
    double width;

    // This field is used only if shape is CIRCLE
    double radius;

    // Constructor for circle
    Figure(double radius) {
        shape = Shape.CIRCLE;
        this.radius = radius;
    }

    // Constructor for rectangle
    Figure(double length, double width) {
        shape = Shape.RECTANGLE;
```

```
        this.length = length;
        this.width = width;
    }

    double area() {
        switch(shape) {
          case RECTANGLE:
            return length * width;
          case CIRCLE:
            return Math.PI * (radius * radius);
          default:
            throw new AssertionError(shape);
        }
    }
}
```

这样的标签类具有许多缺点。 它们充斥着杂乱无章的样板代码，包括枚举声明，标签字段和 `switch` 语句。 可读性更差，因为多个实现在一个类中混杂在一起。 内存使用增加，因为实例负担属于其他风格不相关的领域。 字段不能成为 final，除非构造方法初始化不相关的字段，导致更多的样板代码。 构造方法在编译器的帮助下，必须设置标签字段并初始化正确的数据字段：如果初始化错误的字段，程序将在运行时失败。 除非可以修改其源文件，否则不能将其添加到标记的类中。 如果你添加一个风格，你必须记得给每个 `switch` 语句添加一个 `case` ，否则这个类将在运行时失败。 最后，一个实例的数据类型没有提供任何关于风格的线索。 总之，**标签类是冗长的，容易出错的，而且效率低下。**

幸运的是，像 Java 这样的面向对象的语言为定义一个能够表示多种风格对象的单一数据类型提供了更好的选择：子类型化（subtyping）。标签类仅仅是一个类层次的简单的模仿。

要将标签类转换为类层次，首先定义一个包含抽象方法的抽象类，该标签类的行为取决于标签值。在 `Figure` 类中，只有一个这样的方法，就是 `area` 方法。 这个抽象类是类层次的根。 如果有任何方法的行为不依赖于标签的值，把它们放在这个类中。 同样，如果有所有的方法使用的数据字段，把它们放在这个类。 `Figure` 类中不存在这种与类型无关的方法或字段。

接下来，为原始标签类的每种类型定义一个根类的具体子类。 在我们的例子中，有两个类型：圆形和矩形。 在每个子类中包含特定于改类型的数据字段。 在我们的例子中，半径字段是属于圆的，长度和宽度字段都是矩形的。 还要在每个子类中包含根类中每个抽象方法的适当实现。 这里是对应于 `Figure` 类的类层次：

```
// Class hierarchy replacement for a tagged class
abstract class Figure {
    abstract double area();
}

class Circle extends Figure {
    final double radius;

    Circle(double radius) { this.radius = radius; }

    @Override double area() { return Math.PI * (radius * radius); }
```

```
}
class Rectangle extends Figure {
    final double length;
    final double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width  = width;
    }
    @Override double area() { return length * width; }
}
```

这个类层次纠正了之前提到的标签类的每个缺点。 代码简单明了，不包含原文中的样板文件。 每种类型的实现都是由自己的类来分配的，而这些类都没有被无关的数据字段所占用。 所有的字段是 `final` 的。 编译器确保每个类的构造方法初始化其数据字段，并且每个类都有一个针对在根类中声明的每个抽象方法的实现。 这消除了由于缺少 `switch-case` 语句而导致的运行时失败的可能性。 多个程序员可以独立地继承类层次，并且可以相互操作，而无需访问根类的源代码。 每种类型都有一个独立的数据类型与之相关联，允许程序员指出变量的类型，并将变量和输入参数限制为特定的类型。

类层次的另一个优点是可以使它们反映类型之间的自然层次关系，从而提高了灵活性，并提高了编译时类型检查的效率。 假设原始示例中的标签类也允许使用正方形。 类层次可以用来反映一个正方形是一种特殊的矩形（假设它们是不可变的）：

```
class Square extends Rectangle {
    Square(double side) {
        super(side, side);
    }
}
```

请注意，上述层次结构中的字段是直接访问的，而不是通过访问器方法访问的。 这里是为了简洁起见，如果类层次是公开的（详见第 16 条），这将是一个糟糕的设计。

总之，标签类很少有适用的情况。 如果你想写一个带有显式标签字段的类，请考虑标签字段是否可以被删除，并是否能被类层次结构替换。 当遇到一个带有标签字段的现有类时，可以考虑将其重构为一个类层次结构。

24. 支持使用静态成员类而不是非静态类

嵌套类（nested class）是在另一个类中定义的类。 嵌套类应该只存在于其宿主类（enclosing class）中。 如果一个嵌套类在其他一些情况下是有用的，那么它应该是一个顶级类。 有四种嵌套类：静态成员类，非静态成员类，匿名类和局部类。 除了第一种以外，剩下的三种都被称为内部类（inner class）。 这个条目告诉你什么时候使用哪种类型的嵌套类以及为什么使用。

静态成员类是最简单的嵌套类。 最好把它看作是一个普通的类，恰好在另一个类中声明，并且可以访问所有宿主类的成员，甚至是那些被声明为私有类的成员。 静态成员类是其宿主类的静态成员，并遵循与其他静态成员相同的可访问性规则。 如果它被声明为 `private`，则只能在宿主类中访问，等等。

静态成员类的一个常见用途是作为公共帮助类，仅在与其外部类一起使用时才有用。 例如，考虑一个描述计算器支持的操作的枚举类型（详见第 34 条）。 `Operation` 枚举应该是 `Calculator` 类的公共静态成员类。 `Calculator` 客户端可以使用 `Calculator.Operation.PLUS` 和 `Calculator.Operation.MINUS` 等名称来引用操作。

在语法上，静态成员类和非静态成员类之间的唯一区别是静态成员类在其声明中具有 `static` 修饰符。 尽管句法相似，但这两种嵌套类是非常不同的。 非静态成员类的每个实例都隐含地与其包含的类的宿主实例相关联。 在非静态成员类的实例方法中，可以调用宿主实例上的方法，或者使用限定的构造[JLS，15.8.4] 获得对宿主实例的引用。 如果嵌套类的实例可以与其宿主类的实例隔离存在，那么嵌套类必须是静态成员类：不可能在没有宿主实例的情况下创建非静态成员类的实例。

非静态成员类实例和其宿主实例之间的关联是在创建成员类实例时建立的，并且之后不能被修改。通常情况下，通过在宿主类的实例方法中调用非静态成员类构造方法来自动建立关联。 尽管很少有可能使用表达式 `enclosingInstance.new MemberClass(args)` 手动建立关联。 正如你所预料的那样，该关联在非静态成员类实例中占用了空间，并为其构建添加了时间开销。

非静态成员类的一个常见用法是定义一个 `Adapter` [Gamma95]，它允许将外部类的实例视为某个不相关类的实例。 例如， `Map` 接口的实现通常使用非静态成员类来实现它们的集合视图，这些视图由 `Map` 的 `keySet`， `entrySet` 和 `values` 方法返回。 同样，集合接口（如 `Set` 和 `List`）的实现通常使用非静态成员类来实现它们的迭代器：

```
// Typical use of a nonstatic member class
public class MySet<E> extends AbstractSet<E> {
    ... // Bulk of the class omitted

    @Override
    public Iterator<E> iterator() {
        return new MyIterator();
    }

    private class MyIterator implements Iterator<E> {
        ...
    }
}
```

如果你声明了一个不需要访问宿主实例的成员类，总是把 static 修饰符放在它的声明中，使它成为一个静态成员类，而不是非静态的成员类。 如果你忽略了这个修饰符，每个实例都会有一个隐藏的外部引用给它的宿主实例。 如前所述，存储这个引用需要占用时间和空间。 更严重的是，并且会导致即使宿主类在满足垃圾回收的条件时却仍然驻留在内存中（详见第 7 条）。 由此产生的内存泄漏可能是灾难性的。 由于引用是不可见的，所以通常难以检测到。

私有静态成员类的常见用法是表示由它们的宿主类表示的对象的组件。 例如，考虑将键与值相关联的 `Map` 实例。 许多 `Map` 实现对于映射中的每个键值对都有一个内部的 `Entry` 对象。 当每个 `entry` 都与 `Map` 关联时， `entry` 上的方法（ `getKey` ， `getValue` 和 `setValue` ）不需要访问 `Map` 。 因此，使用非静态成员类来表示 `entry` 将是浪费的：私有静态成员类是最好的。 如果意外地忽略了 `entry` 声明中的 `static` 修饰符， `Map` 仍然可以工作，但是每个 `entry` 都会包含对 `Map` 的引用，浪费空间和时间。

如果所讨论的类是导出类的公共或受保护成员，则在静态和非静态成员类之间正确选择是非常重要的。 在这种情况下，成员类是导出的 API 元素，如果不违反向后兼容性，就不能在后续版本中从非静态变为静态成员类。

正如你所期望的，一个匿名类没有名字。 它不是其宿主类的成员。 它不是与其他成员一起声明，而是在使用时同时声明和实例化。 在表达式合法的代码中，匿名类是允许的。 当且仅当它们出现在非静态上下文中时，匿名类才会封装实例。 但是，即使它们出现在静态上下文中，它们也不能有除常量型变量之外的任何静态成员，这些常量型变量包括 `final` 的基本类型，或者初始化常量表达式的字符串属性[JLS，4.12.4]。

匿名类的适用性有很多限制。 除了在声明的时候之外，不能实例化它们。 你不能执行 `instanceof` 方法测试或者做任何其他需要你命名的类。 不能声明一个匿名类来实现多个接口，或者继承一个类并同时实现一个接口。 匿名类的客户端不能调用除父类型继承的成员以外的任何成员。 因为匿名类在表达式中出现，所以它们必须保持简短 —— 约十行或更少 —— 否则可读性将受到影响。

在将 lambda 表达式添加到 Java（第 6 章）之前，匿名类是创建小函数对象和处理对象的首选方法，但 lambda 表达式现在是首选（详见第 42 条）。 匿名类的另一个常见用途是实现静态工厂方法（请参阅条目 20 中的 `intArrayAsList` ）。

局部类是四种嵌套类中使用最少的。 一个局部类可以在任何可以声明局部变量的地方声明，并遵守相同的作用域规则。 局部类与其他类型的嵌套类具有共同的属性。 像成员类一样，他们有名字，可以重复使用。 就像匿名类一样，只有在非静态上下文中定义它们时，它们才会包含实例，并且它们不能包含静态成员。 像匿名类一样，应该保持简短，以免损害可读性。

回顾一下，有四种不同的嵌套类，每个都有它的用途。 如果一个嵌套的类需要在一个方法之外可见，或者太长而不能很好地适应一个方法，使用一个成员类。 如果一个成员类的每个实例都需要一个对其宿主实例的引用，使其成为非静态的; 否则，使其静态。 假设这个类属于一个方法内部，如果你只需要从一个地方创建实例，并且存在一个预置类型来说明这个类的特征，那么把它作为一个匿名类；否则，把它变成局部类。

25. 将源文件限制为单个顶级类

虽然 Java 编译器允许在单个源文件中定义多个顶级类，但这样做没有任何好处，并且存在重大风险。 风险源于在源文件中定义多个顶级类使得为类提供多个定义成为可能。 使用哪个定义会受到源文件传递给编译器的顺序的影响。

为了具体说明，请考虑下面源文件，其中只包含一个引用其他两个顶级类（ `Utensil` 和 `Dessert` 类）的成员的 Main 类：

```
public class Main {
    public static void main(String[] args) {
        System.out.println(Utensil.NAME + [Dessert.NAME](http://Dessert.NAME));
    }
}
```

现在假设在 `Utensil.java` 的源文件中同时定义了 `Utensil` 和 `Dessert`：

```
// Two classes defined in one file. Don't ever do this!
class Utensil {
    static final String NAME = "pan";
}

class Dessert {
    static final String NAME = "cake";
}
```

当然， `main` 方法会打印 `pancake` 。

现在假设你不小心创建了另一个名为 `Dessert.java` 的源文件，它定义了相同的两个类：

```
// Two classes defined in one file. Don't ever do this!
class Utensil {
    static final String NAME = "pot";
}

class Dessert {
    static final String NAME = "pie";
}
```

如果你足够幸运，使用命令 `javac Main.java Dessert.java` 编译程序，编译将失败，编译器会告诉你，你已经多次定义了类 `Utensil` 和 `Dessert` 。 这是因为编译器首先编译 `Main.java` ，当它看到对 `Utensil` 的引用（它在 `Dessert` 的引用之前）时，它将在 `Utensil.java` 中查找这个类并找到 `Utensil` 和 `Dessert` 。 当编译器在命令行上遇到 `Dessert.java` 时，它也将拉入该文件，导致它遇到 `Utensil` 和 `Dessert` 的定义。

如果使用命令 `javac Main.java` 或 `javac Main.java Utensil.java` 编译程序，它的行为与在编写 `Dessert.java` 文件（即打印 `pancake` ）之前的行为相同。 但是，如果使用命令 `javac Dessert.java Main.java 编译程序` ，它将打印 `potpie` 。 程序的行为因此受到源文件传递给编译器的顺序的影响，这显然是不可接受的。

解决这个问题很简单，将顶层类（如我们的例子中的 `Utensil` 和 `Dessert` ）分割成单独的源文件。 如果试图将多个顶级类放入单个源文件中，请考虑使用静态成员类（详见第 24 条）作为将类拆分为单独的源文件的替代方法。 如果这些类从属于另一个类，那么将它们变成静态成员类通常是更好的选择，因为它提高了可读性，并且可以通过声明它们为私有（详见第 15 条）来减少类的可访问性。下面是我们的例子看起来如何使用静态成员类：

```
// Static member classes instead of multiple top-level classes
public class Test {
    public static void main(String[] args) {
        System.out.println(Utensil.NAME + [Dessert.NAME](http://Dessert.NAME));
    }

    private static class Utensil {
        static final String NAME = "pan";
    }

    private static class Dessert {
        static final String NAME = "cake";
    }
}
```

这个教训很清楚：永远不要将多个顶级类或接口放在一个源文件中。 遵循这个规则保证在编译时不能有多个定义。 这又保证了编译生成的类文件以及生成的程序的行为与源文件传递给编译器的顺序无关。

自 Java 5 以来，泛型已经成为该语言的一部分。 在泛型之前，你必须转换从集合中读取的每个对象。 如果有人不小心插入了错误类型的对象，则在运行时可能会失败。 使用泛型，你告诉编译器在每个集合中允许哪些类型的对象。 编译器会自动插入强制转换，并在编译时告诉你是否尝试插入错误类型的对象。 这样做的结果是既安全又清晰的程序，但这些益处，不限于集合，是有代价的。 本章告诉你如何最大限度地提高益处，并将并发症降至最低。

26. 不要使用原始类型

首先，有几个术语。一个类或接口，它的声明有一个或多个类型参数（type parameters ），被称之为泛型类或泛型接口[JLS，8.1.2,9.1.2]。 例如， `List` 接口具有单个类型参数 E，表示其元素类型。 接口的全名是 `List<E>` （读作「E」的列表），但是人们经常称它为 `List` 。 泛型类和接口统称为泛型类型（generic types）。

每个泛型定义了一组参数化类型（parameterized types），它们由类或接口名称组成，后跟一个与泛型类型的形式类型参数[JLS，4.4,4.5] 相对应的实际类型参数的尖括号「<>」列表。 例如， `List<String>` （读作「字符串列表」）是一个参数化类型，表示其元素类型为 String 的列表。（ `String` 是与形式类型参数 `E` 相对应的实际类型参数）。

最后，每个泛型定义了一个原始类型（raw type），它是没有任何类型参数的泛型类型的名称[JLS，4.8]。 例如，对应于 `List<E>` 的原始类型是 `List` 。 原始类型的行为就像所有的泛型类型信息都从类型声明中被清除一样。 它们的存在主要是为了与没有泛型之前的代码相兼容。

在泛型被添加到 Java 之前，这是一个典型的集合声明。 从 Java 9 开始，它仍然是合法的，但并不是典型的声明方式了：

```
// Raw collection type - don't do this!

// My stamp collection. Contains only Stamp instances.
private final Collection stamps = ... ;
```

如果你今天使用这个声明，然后不小心把 `coin` 实例放入你的 `stamp` 集合中，错误的插入编译和运行没有错误（尽管编译器发出一个模糊的警告）：

```
// Erroneous insertion of coin into stamp collection
stamps.add(new Coin( ... )); // Emits "unchecked call" warning
```

直到您尝试从 `stamp` 集合中检索 `coin` 实例时才会发生错误：

```
// Raw iterator type - don't do this!
for (Iterator i = stamps.iterator(); i.hasNext(); )
    Stamp stamp = (Stamp) i.next(); // Throws ClassCastException
        stamp.cancel();
```

正如本书所提到的，在编译完成之后尽快发现错误是值得的，理想情况是在编译时。 在这种情况下，直到运行时才发现错误，在错误发生后的很长一段时间，以及可能远离包含错误的代码的代码中。 一旦看到 `ClassCastException` ，就必须搜索代码类库，查找将 `coin` 实例放入 `stamp` 集合的方法调用。 编译器不能帮助你，因为它不能理解那个说「仅包含 `stamp` 实例」的注释。

对于泛型，类型声明包含的信息，而不是注释：

```
// Parameterized collection type - typesafe
private final Collection<Stamp> stamps = ... ;
```

从这个声明中，编译器知道 `stamps` 集合应该只包含 `Stamp` 实例，并保证它是 `true` ，假设你的整个代码类库编译时不发出（或者抑制；参见条目 27）任何警告。 当使用参数化类型声明声明 `stamps` 时，错误的插入会生成一个编译时错误消息，告诉你到底发生了什么错误：

```
Test.java:9: error: incompatible types: Coin cannot be converted
to Stamp
    c.add(new Coin());
              ^
```

当从集合中检索元素时，编译器会为你插入不可见的强制转换，并保证它们不会失败（再假设你的所有代码都不会生成或禁止任何编译器警告）。 虽然意外地将 `coin` 实例插入 `stamp` 集合的预期可能看起来很牵强，但这个问题是真实的。 例如，很容易想象将 `BigInteger` 放入一个只包含 `BigDecimal` 实例的集合中。

如前所述，使用原始类型（没有类型参数的泛型）是合法的，但是你不应该这样做。 **如果你使用原始类型，则会丧失泛型的所有安全性和表达上的优势。** 鉴于你不应该使用它们，为什么语言设计者首先允许原始类型呢？ 答案是为了兼容性。 泛型被添加时，Java 即将进入第二个十年，并且有大量的代码没有使用泛型。 所有这些代码都是合法的，并且与使用泛型的新代码进行交互操作被认为是至关重要的。 将参数化类型的实例传递给为原始类型设计的方法必须是合法的，反之亦然。 这个需求，被称为迁移兼容性，驱使决策支持原始类型，并使用擦除来实现泛型（详见第 28 条）。

虽然不应使用诸如 `List` 之类的原始类型，但可以使用参数化类型来允许插入任意对象（如 `List<Object>` ）。 原始类型 List 和参数化类型 `List<Object>` 之间有什么区别？ 松散地说，前者已经选择了泛型类型系统，而后者明确地告诉编译器，它能够保存任何类型的对象。 虽然可以将 `List<String>` 传递给 `List` 类型的参数，但不能将其传递给 `List<Object>` 类型的参数。 泛型有子类型的规则， `List<String>` 是原始类型 `List` 的子类型，但不是参数化类型 `List<Object>` 的子类型（条目 28）。 因此，如果使用诸如 `List` 之类的原始类型，则会丢失类型安全性，但是如果使用参数化类型（例如 `List<Object>` ）则不会。

为了具体说明，请考虑以下程序：

```
// Fails at runtime - unsafeAdd method uses a raw type (List)!
public static void main(String[] args) {
    List<String> strings = new ArrayList<>();
    unsafeAdd(strings, Integer.valueOf(42));
    String s = strings.get(0); // Has compiler-generated cast
}

private static void unsafeAdd(List list, Object o) {
    list.add(o);
}
```

此程序可以编译，它使用原始类型列表，但会收到警告：

```
Test.java:10: warning: [unchecked] unchecked call to add(E) as a
member of the raw type List
    list.add(o);
             ^
```

实际上，如果运行该程序，则当程序尝试调用 `strings.get(0)` 的结果（一个 `Integer` ）转换为一个 `String` 时，会得到 `ClassCastException` 异常。 这是一个编译器生成的强制转换，因此通常会保证成功，但在这种情况下，我们忽略了编译器警告并付出了代价。

如果用 `unsafeAdd` 声明中的参数化类型 `List<Object>` 替换原始类型 `List` ，并尝试重新编译该程序，则会发现它不再编译，而是发出错误消息：

```
Test.java:5: error: incompatible types: List<String> cannot be
converted to List<Object>
    unsafeAdd(strings, Integer.valueOf(42));
```

你可能会试图使用原始类型来处理元素类型未知且无关紧要的集合。 例如，假设你想编写一个方法，它需要两个集合并返回它们共同拥有的元素的数量。 如果是泛型新手，那么您可以这样写：

```
// Use of raw type for unknown element type - don't do this!
static int numElementsInCommon(Set s1, Set s2) {
    int result = 0;
    for (Object o1 : s1)
        if (s2.contains(o1))
            result++;
    return result;
}
```

这种方法可以工作，但它使用原始类型，这是危险的。 安全替代方式是使用无限制通配符类型（unbounded wildcard types）。 如果要使用泛型类型，但不知道或关心实际类型参数是什么，则可以使用问号来代替。 例如，泛型类型 `Set<E>` 的无限制通配符类型是 `Set<?>` （读取「某种类型的集合」）。 它是最通用的参数化的 `Set` 类型，能够保持任何集合。 下面是 `numElementsInCommon` 方法使用无限制通配符类型声明的情况：

```
// Uses unbounded wildcard type - typesafe and flexible
static int numElementsInCommon(Set<?> s1, Set<?> s2) { ... }
```

无限制通配符 `Set<?>` 与原始类型 `Set` 之间有什么区别？ 问号真的给你放任何东西吗？ 这不是要点，但通配符类型是安全的，原始类型不是。 你可以将任何元素放入具有原始类型的集合中，轻易破坏集合的类型不变性（如第 119 页上的 `unsafeAdd` 方法所示）；你不能把任何元素（除 null 之外）放入一个 `Collection<?>` 中。 试图这样做会产生一个像这样的编译时错误消息：

```
WildCard.java:13: error: incompatible types: String cannot be
converted to CAP#1
    c.add("verboten");
          ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
```

不可否认的是，这个错误信息留下了一些需要的东西，但是编译器已经完成了它的工作，不管它的元素类型是什么，都不会破坏集合的类型不变性。 你不仅不能将任何元素（除 `null` 以外）放入一个 `Collection<?>` 中，并且根本无法猜测你会得到那种类型的对象。 如果这些限制是不可接受的，可以使用泛型方法（详见第 30 条）或有限制的通配符类型（详见第 31 条）。

对于不应该使用原始类型的规则，有一些小例外。 **你必须在类字面值（class literals）中使用原始类型。** 规范中不允许使用参数化类型（尽管它允许数组类型和基本类型）[JLS，15.8.2]。 换句话说，`List.class` ， `String[].class` 和 `int.class` 都是合法的，但 `List<String>.class` 和 `List<?>.class` 都是不合法的。

规则的第二个例外与 `instanceof` 操作符有关。 因为泛型类型信息在运行时被擦除，所以在无限制通配符类型以外的参数化类型上使用 `instanceof` 运算符是非法的。 使用无限制通配符类型代替原始类型，不会对 `instanceof` 运算符的行为产生任何影响。 在这种情况下，尖括号（<>）和问号（?）就显得多余。 **以下是使用泛型类型的 `instanceof` 运算符的首选方法：**

```
// Legitimate use of raw type - instanceof operator
if (o instanceof Set) {       // Raw type
    Set<?> s = (Set<?>) o;    // Wildcard type
    ...
}
```

请注意，一旦确定 `o` 对象是一个 `Set`，则必须将其转换为通配符 `Set<?>`，而不是原始类型 `Set`。 这是一个受检查的（checked）转换，所以不会导致编译器警告。

总之，使用原始类型可能导致运行时异常，所以不要使用它们。 原始类型只是为了与引入泛型机制之前的遗留代码进行兼容和互用而提供的。 作为一个快速回顾， `Set<Object>` 是一个参数化类型，表示一个可以包含任何类型对象的集合， `Set<?>` 是一个通配符类型，表示一个只能包含某些未知类型对象的集合， `Set` 是一个原始类型，它不在泛型类型系统之列。 前两个类型是安全的，最后一个不是。

为了快速参考，下表中总结了本条目（以及本章稍后介绍的一些）中介绍的术语：

术语	中文含义	举例	所在条目
Parameterized type	参数化类型	`List<String>`	条目 26
Actual type parameter	实际类型参数	`String`	条目 26
Generic type	泛型类型	`List<E>`	条目 26 和 条目 29
Formal type parameter	形式类型参数	`E`	条目 26
Unbounded wildcard type	无限制通配符类型	`List<?>`	条目 26
Raw type	原始类型	`List`	条目 26
Bounded type parameter	限制类型参数	`<E extends Number>`	条目 29
Recursive type bound	递归类型限制	`<T extends Comparable<T>>`	条目 30
Bounded wildcard type	限制通配符类型	`List<? extends Number>`	条目 31
Generic method	泛型方法	`static <E> List<E> asList(E[] a)`	条目 30
Type token	类型令牌	`String.class`	条目 33

27. 消除非检查警告

使用泛型编程时，会看到许多编译器警告：未经检查的强制转换警告，未经检查的方法调用警告，未经检查的参数化可变长度类型警告以及未经检查的转换警告。 你使用泛型获得的经验越多，获得的警告越少，但不要期望新编写的代码能够干净地编译。

许多未经检查的警告很容易消除。 例如，假设你不小心写了以下声明：

```
Set<Lark> exaltation = new HashSet();
```

编译器会提醒你你做错了什么：

```
Venery.java:4: warning: [unchecked] unchecked conversion
        Set<Lark> exaltation = new HashSet();
                               ^
  required: Set<Lark>
  found:    HashSet。
```

然后可以进行指示修正，让警告消失。 请注意，实际上并不需要指定类型参数，只是为了表明它与 Java 7 中引入的钻石运算符（「<>」）一同出现。然后编译器会推断出正确的实际类型参数（在本例中为 `Lark`）：

```
Set<Lark> exaltation = new HashSet<>();
```

但一些警告更难以消除。 本章充满了这种警告的例子。 当你收到需要进一步思考的警告时，坚持不懈！ **尽可能地消除每一个未经检查的警告。** 如果你消除所有的警告，你可以放心，你的代码是类型安全的，这是一件非常好的事情。 这意味着在运行时你将不会得到一个 `ClassCastException` 异常，并且增加了你的程序将按照你的意图行事的信心。

如果你不能消除警告，但你可以证明引发警告的代码是类型安全的，那么（并且只能这样）用 `@SuppressWarnings("unchecked")` 注解来抑制警告。 如果你在没有首先证明代码是类型安全的情况下压制警告，那么你给自己一个错误的安全感。 代码可能会在不发出任何警告的情况下进行编译，但是它仍然可以在运行时抛出 `ClassCastException` 异常。 但是，如果你忽略了你认为是安全的未经检查的警告（而不是抑制它们），那么当一个新的警告出现时，你将不会注意到这是一个真正的问题。 新出现的警告就会淹没在所有的错误警告当中。

`SuppressWarnings` 注解可用于任何声明，从单个局部变量声明到整个类。 始终在尽可能最小的范围内使用 `SuppressWarnings` 注解。 通常这是一个变量声明或一个非常短的方法或构造方法。 切勿在整个类上使用 `SuppressWarnings` 注解。 这样做可能会掩盖重要的警告。

如果你发现自己在长度超过一行的方法或构造方法上使用 `SuppressWarnings` 注解，则可以将其移到局部变量声明上。 你可能需要声明一个新的局部变量，但这是值得的。 例如，考虑这个来自 `ArrayList` 的 `toArray` 方法：

```
public <T> T[] toArray(T[] a) {
    if (a.length < size)
       return (T[]) Arrays.copyOf(elements, size, a.getClass());
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
       a[size] = null;
    return a;
}

如果编译 ArrayList 类，则该方法会生成此警告：
ArrayList.java:305: warning: [unchecked] unchecked cast
       return (T[]) Arrays.copyOf(elements, size, a.getClass());
                                  ^
  required: T[]
  found:    Object[]
```

在返回语句中设置 `SuppressWarnings` 注解是非法的，因为它不是一个声明[JLS，9.7]。 你可能会试图把注释放在整个方法上，但是不要这要做。 相反，声明一个局部变量来保存返回值并标注它的声明，如下所示：

```
// Adding local variable to reduce scope of @SuppressWarnings
public <T> T[] toArray(T[] a) {
    if (a.length < size) {
        // This cast is correct because the array we're creating
        // is of the same type as the one passed in, which is T[].
        @SuppressWarnings("unchecked") T[] result =
            (T[]) Arrays.copyOf(elements, size, a.getClass());
        return result;
    }
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

所产生的方法干净地编译，并最小化未经检查的警告被抑制的范围。

每当使用 `@SuppressWarnings(“unchecked”)` 注解时，请添加注释，说明为什么是安全的。 这将有助于他人理解代码，更重要的是，这将减少有人修改代码的可能性，从而使计算不安全。 如果你觉得很难写这样的注释，请继续思考。 毕竟，你最终可能会发现未经检查的操作是不安全的。

总之，未经检查的警告是重要的。 不要忽视他们。 每个未经检查的警告代表在运行时出现 `ClassCastException` 异常的可能性。 尽你所能消除这些警告。 如果无法消除未经检查的警告，并且可以证明引发该警告的代码是安全类型的，则可以在尽可能小的范围内使用 `@SuppressWarnings(“unchecked”)` 注解来禁止警告。 记录你决定在注释中抑制此警告的理由。

28. 列表优于数组

数组在两个重要方面与泛型不同。 首先，数组是协变的（covariant）。 这个吓人的单词意味着如果 Sub 是 Super 的子类型，则数组类型 `Sub[]` 是数组类型 `Super[]` 的子类型。 相比之下，泛型是不变的（invariant）：对于任何两种不同的类型 `Type1` 和 `Type2`，`List<Type1>` 既不是 `List<Type2>` 的子类型也不是父类型。[JLS，4.10; Naftalin07, 2.5]。 你可能认为这意味着泛型是不足的，但可以说是数组缺陷。 这段代码是合法的：

```
// Fails at runtime!
Object[] objectArray = new Long[1];
objectArray[0] = "I don't fit in"; // Throws ArrayStoreException
```

但这个不是：

```
// Won't compile!
List<Object> ol = new ArrayList<Long>(); // Incompatible types
ol.add("I don't fit in");
```

无论哪种方式，你不能把一个 `String` 类型放到一个 `Long` 类型容器中，但是用一个数组，你会发现在运行时产生了一个错误；对于列表，可以在编译时就能发现错误。 当然，你宁愿在编译时找出错误。

数组和泛型之间的第二个主要区别是数组被具体化了（reified）[JLS，4.7]。 这意味着数组在运行时知道并强制执行它们的元素类型。 如前所述，如果尝试将一个 `String` 放入 `Long` 数组中，得到一个 `ArrayStoreException` 异常。 相反，泛型通过擦除（erasure）来实现[JLS，4.6]。 这意味着它们只在编译时执行类型约束，并在运行时丢弃（或擦除）它们的元素类型信息。 擦除是允许泛型类型与不使用泛型的遗留代码自由互操作（详见第 26 条），从而确保在 Java 5 中平滑过渡到泛型。

由于这些基本差异，数组和泛型不能很好地在一起混合使用。 例如，创建泛型类型的数组，参数化类型的数组，以及类型参数的数组都是非法的。 因此，这些数组创建表达式都不合法：`new List<E>[]`，`new List<String>[]`，`new E[]`。 所有将在编译时导致泛型数组创建错误。

为什么创建一个泛型数组是非法的？ 因为它不是类型安全的。 如果这是合法的，编译器生成的强制转换程序在运行时可能会因为 `ClassCastException` 异常而失败。 这将违反泛型类型系统提供的基本保证。

为了具体说明，请考虑下面的代码片段：

```
// Why generic array creation is illegal - won't compile!
List<String>[] stringLists = new List<String>[1];  // (1)
List<Integer> intList = List.of(42);               // (2)
Object[] objects = stringLists;                    // (3)
objects[0] = intList;                              // (4)
String s = stringLists[0].get(0);                  // (5)
```

让我们假设第 1 行创建一个泛型数组是合法的。第 2 行创建并初始化包含单个元素的 `List<Integer>`。第 3 行将 `List<String>` 数组存储到 Object 数组变量中，这是合法的，因为数组是协变的。第 4 行将 `List<Integer>` 存储在 Object 数组的唯一元素中，这是因为泛型是通过擦除来实现的：`List<Integer>` 实例的运行时类型仅仅是 `List`，而 `List<String>[]` 实例是 `List[]`，所以这个赋值不会产生 `ArrayStoreException` 异常。现在我们遇到了麻烦。将一个 `List<Integer>` 实例存储到一个声明为仅保存 `List<String>` 实例的数组中。在第 5 行中，我们从这个数组的唯一列表中检索唯一的元素。编译器自动将检索到的元素转换为 `String`，但它是一个 `Integer`，所以我们在运行时得到一个 `ClassCastException` 异常。为了防止发生这种情况，第 1 行（创建一个泛型数组）必须产生一个编译时错误。

类型 `E`， `List<E>` 和 `List<String>` 等在技术上被称为不可具体化的类型（nonreifiable types）[JLS，4.7]。 直观地说，不可具体化的类型是其运行时表示包含的信息少于其编译时表示的类型。 由于擦除，可唯一确定的参数化类型是无限定通配符类型，如 `List<?>` 和 `Map<?, ?>` （详见第 26 条）。 尽管很少有用，创建无限定通配符类型的数组是合法的。

禁止泛型数组的创建可能会很恼人的。 这意味着，例如，泛型集合通常不可能返回其元素类型的数组（但是参见条目 33 中的部分解决方案）。 这也意味着，当使用可变参数方法（详见第 53 条）和泛型时，会产生令人困惑的警告。 这是因为每次调用可变参数方法时，都会创建一个数组来保存可变参数。 如果此数组的元素类型不可确定，则会收到警告。 `SafeVarargs` 注解可以用来解决这个问题（详见第 32 条）。

当你在强制转换为数组类型时，得到泛型数组创建错误，或是未经检查的强制转换警告时，最佳解决方案通常是使用集合类型 `List<E>` 而不是数组类型 `E[]`。 这样可能会牺牲一些简洁性或性能，但作为交换，你会获得更好的类型安全性和互操作性。

例如，假设你想用带有集合的构造方法来编写一个 `Chooser` 类，并且有个方法返回随机选择的集合的一个元素。 根据传递给构造方法的集合，可以使用该类的实例对象作为游戏骰子，魔力 8 号球或蒙特卡罗模拟的数据源。 这是一个没有泛型的简单实现：

```
// Chooser - a class badly in need of generics!
public class Chooser {
    private final Object[] choiceArray;


    public Chooser(Collection choices) {
        choiceArray = choices.toArray();
    }


    public Object choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceArray[rnd.nextInt(choiceArray.length)];
    }
}
```

要使用这个类，每次调用方法时，都必须将 `choose` 方法的返回值从 `Object` 转换为所需的类型，如果类型错误，则转换在运行时失败。 我们先根据条目 29 的建议，试图修改 `Chooser` 类，使其成为泛型的。

```
// A first cut at making Chooser generic - won't compile
public class Chooser<T> {
    private final T[] choiceArray;

    public Chooser(Collection<T> choices) {
        choiceArray = choices.toArray();
    }

    // choose method unchanged
}
```

如果你尝试编译这个类，会得到这个错误信息：

```
Chooser.java:9: error: incompatible types: Object[] cannot be
converted to T[]
        choiceArray = choices.toArray();
                                      ^
  where T is a type-variable:
    T extends Object declared in class Chooser
```

没什么大不了的，将 `Object` 数组转换为 `T` 数组：

```
choiceArray = (T[]) choices.toArray();
```

这没有了错误，而是得到一个警告：

```
Chooser.java:9: warning: [unchecked] unchecked cast
        choiceArray = (T[]) choices.toArray();
                                           ^
  required: T[], found: Object[]
  where T is a type-variable:
T extends Object declared in class Chooser
```

编译器告诉你在运行时不能保证强制转换的安全性，因为程序不会知道 `T` 代表什么类型——记住，元素类型信息在运行时会被泛型删除。 该程序可以正常工作吗？ 是的，但编译器不能证明这一点。 你可以证明这一点，在注释中提出证据，并用注解来抑制警告，但最好是消除警告的原因（详见第 27 条）。

要消除未经检查的强制转换警告，请使用列表而不是数组。 下面是另一个版本的 `Chooser` 类，编译时没有错误或警告：

```
// List-based Chooser - typesafe
public class Chooser<T> {
    private final List<T> choiceList;
```

```
    public Chooser(Collection<T> choices) {
        choiceList = new ArrayList<>(choices);
    }


    public T choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceList.get(rnd.nextInt(choiceList.size()));
    }
}
```

这个版本有些冗长，也许运行比较慢，但是值得一提的是，在运行时不会得到 `ClassCastException` 异常。

总之，数组和泛型具有非常不同的类型规则。 数组是协变和具体化的; 泛型是不变的，类型擦除的。 因此，数组提供运行时类型的安全性，但不提供编译时类型的安全性，对于泛型则是相反。 一般来说，数组和泛型不能很好地混合工作。 如果你发现把它们混合在一起，得到编译时错误或者警告，你的第一个冲动应该是用列表来替换数组。

29. 优先考虑泛型

参数化声明并使用 JDK 提供的泛型类型和方法通常不会太困难。 但编写自己的泛型类型有点困难，但值得努力学习。

考虑条目 7 中的简单堆栈实现：

```
// Object-based collection - a prime candidate for generics
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();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
```

```
        return result;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

这个类应该已经被参数化了，但是由于事实并非如此，我们可以对它进行泛型化。 换句话说，我们可以参数化它，而不会损害原始非参数化版本的客户端。 就目前而言，客户端必须强制转换从堆栈中弹出的对象，而这些强制转换可能会在运行时失败。 泛型化类的第一步是在其声明中添加一个或多个类型参数。 在这种情况下，有一个类型参数，表示堆栈的元素类型，这个类型参数的常规名称是 E（详见第 68 条）。

下一步是用相应的类型参数替换所有使用的 `Object` 类型，然后尝试编译生成的程序：

```
// Initial attempt to generify Stack - won't compile!
public class Stack<E> {
    private E[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new E[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(E e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public E pop() {
        if (size == 0)
            throw new EmptyStackException();
        E result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }
    ... // no changes in isEmpty or ensureCapacity
}
```

你通常会得到至少一个错误或警告，这个类也不例外。 幸运的是，这个类只产生一个错误：

```
Stack.java:8: generic array creation
        elements = new E[DEFAULT_INITIAL_CAPACITY];
                   ^
```

如条目 28 所述，你不能创建一个不可具体化类型的数组，例如类型 `E`。每当编写一个由数组支持的泛型时，就会出现此问题。 有两种合理的方法来解决它。 第一种解决方案直接规避了对泛型数组创建的禁用：创建一个 `Object` 数组并将其转换为泛型数组类型。 现在没有了错误，编译器会发出警告。 这种用法是合法的，但不是（一般）类型安全的：

```
Stack.java:8: warning: [unchecked] unchecked cast
found: Object[], required: E[]
        elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
                        ^
```

编译器可能无法证明你的程序是类型安全的，但你可以。 你必须说服自己，不加限制的类型强制转换不会损害程序的类型安全。 有问题的数组（元素）保存在一个私有属性中，永远不会返回给客户端或传递给任何其他方法。 保存在数组中的唯一元素是那些传递给 `push` 方法的元素，它们是 `E` 类型的，所以未经检查的强制转换不会造成任何伤害。

一旦证明未经检查的强制转换是安全的，请尽可能缩小范围（条目 27）。 在这种情况下，构造方法只包含未经检查的数组创建，所以在整个构造方法中抑制警告是合适的。 通过添加一个注解来执行此操作， `Stack` 可以干净地编译，并且可以在没有显式强制转换或担心 `ClassCastException` 异常的情况下使用它：

```
// The elements array will contain only E instances from push(E).
// This is sufficient to ensure type safety, but the runtime
// type of the array won't be E[]; it will always be Object[]!
@SuppressWarnings("unchecked")
public Stack() {
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
}
```

消除 `Stack` 中的泛型数组创建错误的第二种方法是将属性元素的类型从 `E[]` 更改为 `Object[]`。 如果这样做，会得到一个不同的错误：

```
Stack.java:19: incompatible types
found: Object, required: E
        E result = elements[--size];
                            ^
```

可以通过将从数组中检索到的元素转换为 `E` 来将此错误更改为警告：

```
Stack.java:19: warning: [unchecked] unchecked cast
found: Object, required: E
        E result = (E) elements[--size];
                                ^
```

因为 `E` 是不可具体化的类型，编译器无法在运行时检查强制转换。 再一次，你可以很容易地向自己证明，不加限制的转换是安全的，所以可以适当地抑制警告。 根据条目 27 的建议，我们只在包含未经检查的强制转换的分配上抑制警告，而不是在整个 `pop` 方法上：

```
// Appropriate suppression of unchecked warning
public E pop() {
    if (size == 0)
        throw new EmptyStackException();

    // push requires elements to be of type E, so cast is correct
    @SuppressWarnings("unchecked") E result =
        (E) elements[--size];

    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

两种消除泛型数组创建的技术都有其追随者。 第一个更可读：数组被声明为 `E[]` 类型，清楚地表明它只包含 `E` 实例。 它也更简洁：在一个典型的泛型类中，你从代码中的许多点读取数组；第一种技术只需要一次转换（创建数组的地方），而第二种技术每次读取数组元素都需要单独转换。 因此，第一种技术是优选的并且在实践中更常用。 但是，它确实会造成堆污染（heap pollution）（详见第 32 条）：数组的运行时类型与编译时类型不匹配（除非 `E` 碰巧是 `Object`）。 这使得一些程序员非常不安，他们选择了第二种技术，尽管在这种情况下堆的污染是无害的。

下面的程序演示了泛型 `Stack` 类的使用。 该程序以相反的顺序打印其命令行参数，并将其转换为大写。 对从堆栈弹出的元素调用 `String` 的 `toUpperCase` 方法不需要显式强制转换，而自动生成的强制转换将保证成功：

```
// Little program to exercise our generic Stack
public static void main(String[] args) {
    Stack<String> stack = new Stack<>();
    for (String arg : args)
        stack.push(arg);
    while (!stack.isEmpty())
        System.out.println(stack.pop().toUpperCase());
}
```

上面的例子似乎与条目 28 相矛盾，条目 28 中鼓励使用列表优先于数组。 在泛型类型中使用列表并不总是可行或可取的。 Java 本身生来并不支持列表，所以一些泛型类型（如 `ArrayList`）必须在数组上实现。 其他的泛型类型，比如 `HashMap`，是为了提高性能而实现的。

绝大多数泛型类型就像我们的 `Stack` 示例一样，它们的类型参数没有限制：可以创建一个 `Stack<Object>`, `Stack<int[]>` , `Stack<List<String>>` 或者其他任何对象的 `Stack` 引用类型。 请注意，不能创建基本类型的堆栈：尝试创建 `Stack<int>` 或 `Stack<double>` 将导致编译时错误。 这是 Java 泛型类型系统的一个基本限制。 可以使用基本类型的包装类（详见第 61 条）来解决这个限制。

有一些泛型类型限制了它们类型参数的允许值。 例如，考虑 `java.util.concurrent.DelayQueue` ，它的声明如下所示：

```
class DelayQueue<E extends Delayed> implements BlockingQueue<E>
```

类型参数列表（ `<E extends Delayed>` ）要求实际的类型参数 `E` 是 `java.util.concurrent.Delayed` 的子类型。 这使得 `DelayQueue` 实现及其客户端可以利用 `DelayQueue` 元素上的 `Delayed` 方法，而不需要显式的转换或 `ClassCastException` 异常的风险。 类型参数 `E` 被称为限定类型参数。 请注意，子类型关系被定义为每个类型都是自己的子类型[JLS，4.10]，因此创建 `DelayQueue<Delayed>` 是合法的。

总之，泛型类型比需要在客户端代码中强制转换的类型更安全，更易于使用。 当你设计新的类型时，确保它们可以在没有这种强制转换的情况下使用。 这通常意味着使类型泛型化。 如果你有任何现有的类型，应该是泛型的但实际上却不是，那么把它们泛型化。 这使这些类型的新用户的使用更容易，而不会破坏现有的客户端（条目 26）。

30. 优先使用泛型方法

正如类可以是泛型的，方法也可以是泛型的。 对参数化类型进行操作的静态工具方法通常都是泛型的。 集合中的所有“算法”方法（如 `binarySearch` 和 `sort` ）都是泛型的。

编写泛型方法类似于编写泛型类型。 考虑这个方法，它返回两个集合的并集：

```
// Uses raw types - unacceptable! [Item 26]
public static Set union(Set s1, Set s2) {
    Set result = new HashSet(s1);
    result.addAll(s2);
    return result;
}
```

此方法可以编译但有两个警告：

```
Union.java:5: warning: [unchecked] unchecked call to
HashSet(Collection<? extends E>) as a member of raw type HashSet
        Set result = new HashSet(s1);
                     ^
Union.java:6: warning: [unchecked] unchecked call to
addAll(Collection<? extends E>) as a member of raw type Set
        result.addAll(s2);
                     ^
```

要修复这些警告并使方法类型安全，请修改其声明以声明表示三个集合（两个参数和返回值）的元素类型的类型参数，并在整个方法中使用此类型参数。 声明类型参数的类型参数列表位于方法的修饰符和返回类型之间。 在这个例子中，类型参数列表是 `<E>`，返回类型是 `Set<E>`。 类型参数的命名约定对于泛型方法和泛型类型是相同的（详见第 29 和 68 条）：

```
// Generic method
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {
    Set<E> result = new HashSet<>(s1);
    result.addAll(s2);
    return result;

}
```

至少对于简单的泛型方法来说，就是这样。 此方法编译时不会生成任何警告，并提供类型安全性和易用性。 这是一个简单的程序来运行该方法。 这个程序不包含强制转换，并且编译时没有错误或警告：

```
// Simple program to exercise generic method
public static void main(String[] args) {
    Set<String> guys = Set.of("Tom", "Dick", "Harry");
    Set<String> stooges = Set.of("Larry", "Moe", "Curly");
    Set<String> aflCio = union(guys, stooges);
    System.out.println(aflCio);
}
```

当运行这个程序时，它会打印 `[Moe, Tom, Harry, Larry, Curly, Dick]`（输出中元素的顺序依赖于具体实现。）

`union` 方法的一个限制是所有三个集合（输入参数和返回值）的类型必须完全相同。 通过使用限定通配符类型（ bounded wildcard types）（详见第 31 条），可以使该方法更加灵活。

有时，需要创建一个不可改变但适用于许多不同类型的对象。 因为泛型是通过擦除来实现的（详见第 28 条），所以可以使用单个对象进行所有必需的类型参数化，但是需要编写一个静态工厂方法来重复地为每个请求的类型参数化分配对象。 这种被称为泛型单例工厂（generic singleton factory）的模式，用于函数对象（function objects）（详见第 42 条），比如 `Collections.reverseOrder` 方法，偶尔也用于 `Collections.emptySet` 之类的集合。

假设你想写一个恒等方法分配器（identity function dispenser）。类库提供了`Function.identity`方法，所以没有理由编写你自己的实现（详见第 59 条），但它是有启发性的。如果每次要求的时候都去创建一个新的恒等方法对象是浪费的，因为它是无状态的。如果 Java 的泛型被具体化，那么每个类型都需要一个恒等方法，但是由于它们被擦除以后，所以泛型的单例就足够了。以下是它的实例：

```
// Generic singleton factory pattern
private static UnaryOperator<Object> IDENTITY_FN = (t) -> t;

@SuppressWarnings("unchecked")
public static <T> UnaryOperator<T> identityFunction() {
    return (UnaryOperator<T>) IDENTITY_FN;
}
```

将`IDENTITY_FN`转换为（`UnaryFunction<T>`）会生成一个未经检查的强制转换警告，因为`UnaryOperator<Object>`对于每个`T`都不是一个`UnaryOperator<T>`。但是恒等方法是特殊的：它返回未修改的参数，所以我们知道，使用它作为一个`UnaryFunction<T>`是类型安全的，无论`T`的值是多少。因此，我们可以放心地抑制由这个强制生成的未经检查的强制转换警告。一旦我们完成了这些，代码编译没有错误或警告。

下面是一个示例程序，它使用我们的泛型单例作为`UnaryOperator<String>`和`UnaryOperator<Number>`。像往常一样，它不包含强制转化，编译时也没有错误和警告：

```
// Sample program to exercise generic singleton
public static void main(String[] args) {
    String[] strings = { "jute", "hemp", "nylon" };
    UnaryOperator<String> sameString = identityFunction();

    for (String s : strings)
        System.out.println(sameString.apply(s));

    Number[] numbers = { 1, 2.0, 3L };

    UnaryOperator<Number> sameNumber = identityFunction();

    for (Number n : numbers)
        System.out.println(sameNumber.apply(n));
}
```

虽然相对较少，类型参数受涉及该类型参数本身的某种表达式限制是允许的。这就是所谓的递归类型限制（recursive type bound）。递归类型限制的常见用法与`Comparable`接口有关，它定义了一个类型的自然顺序（详见第 14 条）。这个接口如下所示：

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

类型参数 T 定义了实现 `Comparable<T>` 的类型的元素可以比较的类型。 在实际中，几乎所有类型都只能与自己类型的元素进行比较。 所以，例如， `String` 类实现了 `Comparable<String>` ， `Integer` 类实现了 `Comparable<Integer>` 等等。

许多方法采用实现 `Comparable` 的元素的集合来对其进行排序，在其中进行搜索，计算其最小值或最大值等。 要做到这一点，要求集合中的每一个元素都可以与其中的每一个元素相比，换言之，这个元素是可以相互比较的。 以下是如何表达这一约束：

```
// Using a recursive type bound to express mutual comparability
public static <E extends Comparable<E>> E max(Collection<E> c);
```

限定的类型 `<E extends Comparable <E >>` 可以理解为「任何可以与自己比较的类型 E」，这或多或少精确地对应于相互可比性的概念。

这里有一个与前面的声明相匹配的方法。它根据其元素的自然顺序来计算集合中的最大值，并编译没有错误或警告：

```
// Returns max value in a collection - uses recursive type bound
public static <E extends Comparable<E>> E max(Collection<E> c) {
  if (c.isEmpty())
    throw new IllegalArgumentException("Empty collection");
  E result = null;
  for (E e : c)
    if (result == null || e.compareTo(result) > 0)
      result = Objects.requireNonNull(e);
  return result;
}
```

请注意，如果列表为空，则此方法将引发 `IllegalArgumentException` 异常。 更好的选择是返回一个 `Optional<E>` （详见第 55 条）。

递归类型限制可能变得复杂得多，但幸运的是他们很少这样做。 如果你理解了这个习惯用法，它的通配符变体（详见第 31 条）和模拟的自我类型用法（详见第 2 条），你将能够处理在实践中遇到的大多数递归类型限制。

总之，像泛型类型一样，泛型方法比需要客户端对输入参数和返回值进行显式强制转换的方法更安全，更易于使用。 像类型一样，你应该确保你的方法可以不用强制转换，这通常意味着它们是泛型的。 应该泛型化现有的方法，其使用需要强制转换。 这使得新用户的使用更容易，而不会破坏现有的客户端（详见第 26 条）。

31. 使用限定通配符来增加 API 的灵活性

如条目 28 所述，参数化类型是不变的。换句话说，对于任何两个不同类型的 `Type1` 和 `Type2`，`List<Type1>` 既不是 `List<Type2>` 的子类型也不是其父类型。尽管 `List<String>` 不是 `List<Object>` 的子类型是违反直觉的，但它确实是有道理的。 可以将任何对象放入 `List<Object>` 中，但是只能将字符串放入 `List<String>` 中。 由于 `List<String>` 不能做 `List<Object>` 所能做的所有事情，所以它不是一个子类型（条目 10 中的里氏替代原则）。

相对于提供的不可变的类型，有时你需要比此更多的灵活性。 考虑条目 29 中的 `Stack` 类。下面是它的公共 API：

```
public class Stack<E> {

    public Stack();

    public void push(E e);

    public E pop();

    public boolean isEmpty();

}
```

假设我们想要添加一个方法来获取一系列元素，并将它们全部推送到栈上。 以下是第一种尝试：

```
// pushAll method without wildcard type - deficient!
public void pushAll(Iterable<E> src) {
    for (E e : src)
        push(e);
}
```

这种方法可以干净地编译，但不完全令人满意。 如果可遍历的 `src` 元素类型与栈的元素类型完全匹配，那么它工作正常。 但是，假设有一个 `Stack<Number>`，并调用 `push(intVal)`，其中 `intVal` 的类型是 `Integer`。 这是因为 `Integer` 是 `Number` 的子类型。 从逻辑上看，这似乎也应该起作用：

```
Stack<Number> numberStack = new Stack<>();
Iterable<Integer> integers = ... ;
numberStack.pushAll(integers);
```

但是，如果你尝试了，会得到这个错误消息，因为参数化类型是不变的：

```
StackTest.java:7: error: incompatible types: Iterable<Integer>
cannot be converted to Iterable<Number>
        numberStack.pushAll(integers);
                            ^
```

幸运的是，有对应的解决方法。 该语言提供了一种特殊的参数化类型来调用一个限定通配符类型来处理这种情况。 `pushAll` 的输入参数的类型不应该是「`E` 的 `Iterable` 接口」，而应该是「`E` 的某个子类型的 `Iterable` 接口」，并且有一个通配符类型，这意味着： `Iterable<? extends E>` 。（关键字 `extends` 的使用有点误导：回忆条目 29 中，子类型被定义为每个类型都是它自己的子类型，即使它本身没有继承。）让我们修改 `pushAll` 来使用这个类型：

```
// Wildcard type for a parameter that serves as an E producer
public void pushAll(Iterable<? extends E> src) {
    for (E e : src)
        push(e);
}
```

有了这个改变， `Stack` 类不仅可以干净地编译，而且客户端代码也不会用原始的 pushAll 声明编译。 因为 `Stack` 和它的客户端干净地编译，你知道一切都是类型安全的。

现在假设你想写一个 `popAll` 方法，与 `pushAll` 方法相对应。 `popAll` 方法从栈中弹出每个元素并将元素添加到给定的集合中。 以下是第一次尝试编写 `popAll` 方法的过程：

```
// popAll method without wildcard type - deficient!
public void popAll(Collection<E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

同样，如果目标集合的元素类型与栈的元素类型完全匹配，则干净编译并且工作正常。 但是，这又不完全令人满意。 假设你有一个 `Stac<Number>` 和 `Object` 类型的变量。 如果从栈中弹出一个元素并将其存储在该变量中，它将编译并运行而不会出错。 你不应该也这样做吗？

```
Stack<Number> numberStack = new Stack<Number>();

Collection<Object> objects = ... ;

numberStack.popAll(objects);
```

如果尝试将此客户端代码与之前显示的 `popAll` 版本进行编译，则会得到与我们的第一版 `pushAll` 非常类似的错误： `Collection<Object>` 不是 `Collection<Number>` 的子类型。 通配符类型再一次提供了一条出路。 `popAll` 的输入参数的类型不应该是「E 的集合」，而应该是「E 的某个父类型的集合」（其中父类型被定义为 `E` 是它自己的父类型[JLS，4.10]）。 再次，有一个通配符类型，正是这个意思： `Collection<? super E>` 。 让我们修改 `popAll` 来使用它：

```
// Wildcard type for parameter that serves as an E consumer
public void popAll(Collection<? super E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

通过这个改动，`Stack` 类和客户端代码都可以干净地编译。

这个结论很清楚。**为了获得最大的灵活性，对代表生产者或消费者的输入参数使用通配符类型。**如果一个输入参数既是一个生产者又是一个消费者，那么通配符类型对你没有好处：你需要一个精确的类型匹配，这就是没有任何通配符的情况。

这里有一个助记符来帮助你记住使用哪种通配符类型：**PECS 代表：producer-extends，consumer-super。**

换句话说，如果一个参数化类型代表一个 `T` 生产者，使用 `<? extends T>`；如果它代表 `T` 消费者，则使用 `<? super T>`。在我们的 `Stack` 示例中，`pushAll` 方法的 `src` 参数生成栈使用的 `E` 实例，因此 `src` 的合适类型为 `Iterable<? extends E>`；`popAll` 方法的 `dst` 参数消费 `Stack` 中的 `E` 实例，因此 `dst` 的合适类型是 `Collection <? super E>`。PECS 助记符抓住了使用通配符类型的基本原则。Naftalin 和 Wadler 称之为获取和放置原则（Get and Put Principle）[Naftalin07,2.4]。

记住这个助记符之后，让我们来看看本章中以前项目的一些方法和构造方法声明。条目 28 中的 `Chooser` 类构造方法有这样的声明：

```
public Chooser(Collection<T> choices)
```

这个构造方法只使用集合选择来生产类型 T 的值（并将它们存储起来以备后用），所以它的声明应该使用一个 extends T 的通配符类型。下面是得到的构造方法声明：

```
// Wildcard type for parameter that serves as an T producer

public Chooser(Collection<? extends T> choices)
```

这种改变在实践中会有什么不同吗？是的，会有不同。假你有一个 `List<Integer>`，并且想把它传递给 `Chooser<Number>` 的构造方法。这不会与原始声明一起编译，但是它只会将限定通配符类型添加到声明中。

现在看看条目 30 中的 `union` 方法。下是声明：

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

两个参数 s1 和 s2 都是 `E` 的生产者，所以 PECS 助记符告诉我们该声明应该如下：

```
public static <E> Set<E> union(Set<? extends E> s1,  Set<? extends E> s2)
```

请注意，返回类型仍然是 `Set<E>`。不要使用限定通配符类型作为返回类型。除了会为用户提供额外的灵活性，还强制他们在客户端代码中使用通配符类型。通过修改后的声明，此代码将清晰地编译：

```
Set<Integer>  integers =  Set.of(1, 3, 5);

Set<Double>   doubles  =  Set.of(2.0, 4.0, 6.0);

Set<Number>   numbers  =  union(integers, doubles);
```

如果使用得当，类的用户几乎不会看到通配符类型。 他们使方法接受他们应该接受的参数，拒绝他们应该拒绝的参数。 如果一个类的用户必须考虑通配符类型，那么它的 API 可能有问题。

在 Java 8 之前，类型推断规则不够聪明，无法处理先前的代码片段，这要求编译器使用上下文指定的返回类型（或目标类型）来推断 `E` 的类型。 `union` 方法调用的目标类型如前所示是 `Set<Number>` 。 如果尝试在早期版本的 Java 中编译片段（以及适合的 `Set.of` 工厂替代版本），将会看到如此长的错综复杂的错误消息：

```
Union.java:14: error: incompatible types
        Set<Number> numbers = union(integers, doubles);
                                    ^
  required: Set<Number>
  found:    Set<INT#1>
  where INT#1,INT#2 are intersection types:
    INT#1 extends Number,Comparable<? extends INT#2>
    INT#2 extends Number,Comparable<?>
```

幸运的是有办法来处理这种错误。 如果编译器不能推断出正确的类型，你可以随时告诉它使用什么类型的显式类型参数[JLS，15.12]。 甚至在 Java 8 中引入目标类型之前，这不是你必须经常做的事情，这很好，因为显式类型参数不是很漂亮。 通过添加显式类型参数，如下所示，代码片段在 Java 8 之前的版本中进行了干净编译：

```
// Explicit type parameter - required prior to Java 8
Set<Number> numbers = Union.<Number>union(integers, doubles);
```

接下来让我们把注意力转向条目 30 中的 `max` 方法。这里是原始声明：

```
public static <T extends Comparable<T>> T max(List<T> list)
```

为了从原来到修改后的声明，我们两次应用了 PECS。首先直接的应用是参数列表。 它生成 T 实例，所以将类型从 `List<T>` 更改为 `List<? extends T>` 。 棘手的应用是类型参数 `T` 。这是我们第一次看到通配符应用于类型参数。 最初， `T` 被指定为继承 `Comparable<T>` ，但 `Comparable` 的 `T` 消费 `T` 实例（并生成指示顺序关系的整数）。 因此，参数化类型 `Comparable<T>` 被替换为限定通配符类型 `Comparable<? super T>` 。 `Comparable` 实例总是消费者，所以通常应该**使用** `Comparable<? super T>` **优于** `Comparable<T>` 。 `Comparator` 也是如此。因此，通常应该**使用** `Comparator<? super T>` **优于** `Comparator<T>` 。

修改后的 `max` 声明可能是本书中最复杂的方法声明。 增加的复杂性是否真的起作用了吗？ 同样，它的确如此。 这是一个列表的简单例子，它被原始声明排除，但在被修改后的版本里是允许的：

```
List<ScheduledFuture<?>> scheduledFutures = ... ;
```

无法将原始方法声明应用于此列表的原因是 `ScheduledFuture` 不实现 `Comparable<ScheduledFuture>`。 相反，它是 `Delayed` 的子接口，它继承了 `Comparable<Delayed>`。 换句话说，一个 `ScheduledFuture` 实例不仅仅和其他的 `ScheduledFuture` 实例相比较： 它可以与任何 `Delayed` 实例比较，并且足以导致原始的声明拒绝它。 更普遍地说，通配符要求来支持没有直接实现 `Comparable` （或 `Comparator` ）的类型，但继承了一个类型。

还有一个关于通配符相关的话题。 类型参数和通配符之间具有双重性，许多方法可以用一个或另一个声明。 例如，下面是两个可能的声明，用于交换列表中两个索引项目的静态方法。 第一个使用无限制类型参数（条目 30），第二个使用无限制通配符：

```
// Two possible declarations for the swap method
public static <E> void swap(List<E> list, int i, int j);
public static void swap(List<?> list, int i, int j);
```

这两个声明中的哪一个更可取，为什么？ 在公共 API 中，第二个更好，因为它更简单。 你传入一个列表（任何列表），该方法交换索引的元素。 没有类型参数需要担心。 通常， **如果类型参数在方法声明中只出现一次，请将其替换为通配符。** 如果它是一个无限制的类型参数，请将其替换为无限制的通配符; 如果它是一个限定类型参数，则用限定通配符替换它。

第二个 `swap` 方法声明有一个问题。 这个简单的实现不会编译：

```
public static void swap(List<?> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

试图编译它会产生这个不太有用的错误信息：

```
Swap.java:5: error: incompatible types: Object cannot be
converted to CAP#1
        list.set(i, list.set(j, list.get(i)));
                                         ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
```

看起来我们不能把一个元素放回到我们刚刚拿出来的列表中。 问题是列表的类型是 `List<?>`，并且不能将除 `null` 外的任何值放入 `List<?>` 中。 幸运的是，有一种方法可以在不使用不安全的转换或原始类型的情况下实现此方法。 这个想法是写一个私有辅助方法来捕捉通配符类型。 辅助方法必须是泛型方法才能捕获类型。 以下是它的定义：

```
public static void swap(List<?> list, int i, int j) {
    swapHelper(list, i, j);
}

// Private helper method for wildcard capture
private static <E> void swapHelper(List<E> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

`swapHelper` 方法知道该列表是一个 `List<E>`。 因此，它知道从这个列表中获得的任何值都是 `E` 类型，并且可以安全地将任何类型的 E 值放入列表中。 这个稍微复杂的 `swap` 的实现可以干净地编译。 它允许我们导出基于通配符的漂亮声明，同时利用内部更复杂的泛型方法。 `swap` 方法的客户端不需要面对更复杂的 `swapHelper` 声明，但他们从中受益。 辅助方法具有我们认为对公共方法来说过于复杂的签名。

总之，在你的 API 中使用通配符类型，虽然棘手，但使得 API 更加灵活。 如果编写一个将被广泛使用的类库，正确使用通配符类型应该被认为是强制性的。 记住基本规则： producer-extends, consumer-super（PECS）。 还要记住，所有 `Comparable` 和 `Comparator` 都是消费者。

32. 合理地结合泛型和可变参数

在 Java 5 中，可变参数方法（详见第 53 条）和泛型都被添加到平台中，所以你可能希望它们能够正常交互; 可悲的是，他们并没有。 可变参数的目的是允许客户端将一个可变数量的参数传递给一个方法，但这是一个脆弱的抽象（leaky abstraction）：当你调用一个可变参数方法时，会创建一个数组来保存可变参数；那个应该是实现细节的数组是可见的。 因此，当可变参数具有泛型或参数化类型时，会导致编译器警告混淆。

回顾条目 28，非具体化（non-reifiable）的类型是其运行时表示比其编译时表示具有更少信息的类型，并且几乎所有泛型和参数化类型都是不可具体化的。 如果某个方法声明其可变参数为非具体化的类型，则编译器将在该声明上生成警告。 如果在推断类型不可确定的可变参数参数上调用该方法，那么编译器也会在调用中生成警告。 警告看起来像这样：

```
warning: [unchecked] Possible heap pollution from
    parameterized vararg type List<String>
```

当参数化类型的变量引用不属于该类型的对象时会发生堆污染（Heap pollution）[JLS，4.12.2]。 它会导致编译器的自动生成的强制转换失败，违反了泛型类型系统的基本保证。

例如，请考虑以下方法，该方法是第 127 页上的代码片段的一个不太明显的变体：

```
// Mixing generics and varargs can violate type safety!
static void dangerous(List<String>... stringLists) {
    List<Integer> intList = List.of(42);
    Object[] objects = stringLists;
    objects[0] = intList;             // Heap pollution
    String s = stringLists[0].get(0); // ClassCastException
}
```

此方法没有可见的强制转换，但在调用一个或多个参数时抛出 `ClassCastException` 异常。 它的最后一行有一个由编译器生成的隐形转换。 这种转换失败，表明类型安全性已经被破坏，并且将值保存在泛型可变参数数组参数中是不安全的。

这个例子引发了一个有趣的问题：为什么声明一个带有泛型可变参数的方法是合法的，当明确创建一个泛型数组是非法的时候呢？ 换句话说，为什么前面显示的方法只生成一个警告，而 127 页上的代码片段会生成一个错误？ 答案是，具有泛型或参数化类型的可变参数参数的方法在实践中可能非常有用，因此语言设计人员选择忍受这种不一致。 事实上，Java 类库导出了几个这样的方法，包括 `Arrays.asList(T... a)` ， `Collections.addAll(Collection<? super T> c, T... elements)` ， `EnumSet.of(E first, E... rest)` 。 与前面显示的危险方法不同，这些类库方法是类型安全的。

在 Java 7 中， `@SafeVarargs` 注解已添加到平台，以允许具有泛型可变参数的方法的作者自动禁止客户端警告。 实质上， `@SafeVarargs` 注解构成了作者对类型安全的方法的承诺。 为了交换这个承诺，编译器同意不要警告用户调用可能不安全的方法。

除非它实际上是安全的，否则注意不要使用 `@SafeVarargs` 注解标注一个方法。 那么需要做些什么来确保这一点呢？ 回想一下，调用方法时会创建一个泛型数组，以容纳可变参数。 如果方法没有在数组中存储任何东西（它会覆盖参数）并且不允许对数组的引用进行转义（这会使不受信任的代码访问数组），那么它是安全的。 换句话说，如果可变参数数组仅用于从调用者向方法传递可变数量的参数——毕竟这是可变参数的目的——那么该方法是安全的。

值得注意的是，你可以违反类型安全性，即使不会在可变参数数组中存储任何内容。 考虑下面的泛型可变参数方法，它返回一个包含参数的数组。 乍一看，它可能看起来像一个方便的小工具：

```
// UNSAFE - Exposes a reference to its generic parameter array!
static <T> T[] toArray(T... args) {
    return args;
}
```

这个方法只是返回它的可变参数数组。 该方法可能看起来并不危险，但它是！ 该数组的类型由传递给方法的参数的编译时类型决定，编译器可能没有足够的信息来做出正确的判断。 由于此方法返回其可变参数数组，它可以将堆污染传播到调用栈上。

为了具体说明，请考虑下面的泛型方法，它接受三个类型 T 的参数，并返回一个包含两个参数的数组，随机选择：

```
static <T> T[] pickTwo(T a, T b, T c) {
    switch(ThreadLocalRandom.current().nextInt(3)) {
      case 0: return toArray(a, b);
      case 1: return toArray(a, c);
      case 2: return toArray(b, c);
    }
    throw new AssertionError(); // Can't get here
}
```

这个方法本身不是危险的，除了调用具有泛型可变参数的 `toArray` 方法之外，不会产生警告。

编译此方法时，编译器会生成代码以创建一个将两个 T 实例传递给 `toArray` 的可变参数数组。 这段代码分配了一个 `Object[]` 类型的数组，它是保证保存这些实例的最具体的类型，而不管在调用位置传递给 `pickTwo` 的对象是什么类型。 `toArray` 方法只是简单地将这个数组返回给 `pickTwo` ，然后 `pickTwo` 将它返回给调用者，所以 `pickTwo` 总是返回一个 `Object[]` 类型的数组。

```
public static void main(String[] args) {
    String[] attributes = pickTwo("Good", "Fast", "Cheap");
}
```

这种方法没有任何问题，因此它编译时不会产生任何警告。 但是当运行它时，抛出一个 `ClassCastException` 异常，尽管不包含可见的转换。 你没有看到的是，编译器已经生成了一个隐藏的强制转换为由 `pickTwo` 返回的值的 `String[]` 类型，以便它可以存储在属性中。 转换失败，因为 `Object[]` 不是 `String[]` 的子类型。 这种故障相当令人不安，因为它从实际导致堆污染（ `toArray` ）的方法中移除了两个级别，并且在实际参数存储在其中之后，可变参数数组未被修改。

这个例子是为了让人们认识到给另一个方法访问一个泛型的可变参数数组是不安全的，除了两个例外：将数组传递给另一个可变参数方法是安全的，这个方法是用 `@SafeVarargs` 正确标注的， 将数组传递给一个非可变参数的方法是安全的，该方法仅计算数组内容的一些方法。

这里是安全使用泛型可变参数的典型示例。 此方法将任意数量的列表作为参数，并按顺序返回包含所有输入列表元素的单个列表。 由于该方法使用 `@SafeVarargs` 进行标注，因此在声明或其调用站位置上不会生成任何警告：

```
// Safe method with a generic varargs parameter
@SafeVarargs
static <T> List<T> flatten(List<? extends T>... lists) {
    List<T> result = new ArrayList<>();
    for (List<? extends T> list : lists)
        result.addAll(list);
    return result;
}
```

决定何时使用 `@SafeVarargs` 注解的规则很简单：在每种方法上使用 `@SafeVarargs`，并使用泛型或参数化类型的可变参数，这样用户就不会因不必要的和令人困惑的编译器警告而担忧。 这意味着你不应该写危险或者 `toArray` 等不安全的可变参数方法。 每次编译器警告你可能会受到来自你控制的方法中泛型可变参数的堆污染时，请检查该方法是否安全。 提醒一下，在下列情况下，泛型可变参数方法是安全的：

1. 它不会在可变参数数组中存储任何东西
2. 它不会使数组（或克隆）对不可信代码可见。 如果违反这些禁令中的任何一项，请修复。

请注意， `SafeVarargs` 注解只对不能被重写的方法是合法的，因为不可能保证每个可能的重写方法都是安全的。 在 Java 8 中，注解仅在静态方法和 `final` 实例方法上合法; 在 Java 9 中，它在私有实例方法中也变为合法。

使用 `SafeVarargs` 注解的替代方法是采用条目 28 的建议，并用 `List` 参数替换可变参数（这是一个变相的数组）。 下面是应用于我们的 `flatten` 方法时，这种方法的样子。 请注意，只有参数声明被更改了：

```
// List as a typesafe alternative to a generic varargs parameter
static <T> List<T> flatten(List<List<? extends T>> lists) {
    List<T> result = new ArrayList<>();
    for (List<? extends T> list : lists)
        result.addAll(list);
    return result;
}
```

然后可以将此方法与静态工厂方法 `List.of` 结合使用，以允许可变数量的参数。 请注意，这种方法依赖于 `List.of` 声明使用 `@SafeVarargs` 注解：

```
audience = flatten(List.of(friends, romans, countrymen));
```

这种方法的优点是编译器可以证明这种方法是类型安全的。 不必使用 `@SafeVarargs` 注解来证明其安全性，也不用担心在确定安全性时可能会犯错。 主要缺点是客户端代码有点冗长，运行可能会慢一些。

这个技巧也可以用在不可能写一个安全的可变参数方法的情况下，就像第 147 页的 `toArray` 方法那样。它的列表模拟是 `List.of` 方法，所以我们甚至不必编写它；Java 类库作者已经为我们完成了这项工作。 `pickTwo` 方法然后变成这样：

```
static <T> List<T> pickTwo(T a, T b, T c) {
    switch(rnd.nextInt(3)) {
      case 0: return List.of(a, b);
      case 1: return List.of(a, c);
      case 2: return List.of(b, c);
    }
    throw new AssertionError();
}
```

`main` 方变成这样：

```
public static void main(String[] args) {
    List<String> attributes = pickTwo("Good", "Fast", "Cheap");
}
```

生成的代码是类型安全的，因为它只使用泛型，不是数组。

总而言之，可变参数和泛型不能很好地交互，因为可变参数机制是在数组上面构建的脆弱的抽象，并且数组具有与泛型不同的类型规则。 虽然泛型可变参数不是类型安全的，但它们是合法的。 如果选择使用泛型（或参数化）可变参数编写方法，请首先确保该方法是类型安全的，然后使用 `@SafeVarargs` 注解对其进行标注，以免造成使用不愉快。

33. 优先考虑类型安全的异构容器

泛型的常见用法包括集合，如 `Set<E>` 和 `Map<K, V>` 和单个元素容器，如 `ThreadLocal<T>` 和 `AtomicReference<T>` 。 在所有这些用途中，它都是参数化的容器。 这限制了每个容器只能有固定数量的类型参数。 通常这正是你想要的。 一个 `Set` 有单一的类型参数，表示它的元素类型; 一个 `Map` 有两个，代表它的键和值的类型；等等。

然而有时候，你需要更多的灵活性。 例如，数据库一行记录可以具有任意多列，并且能够以类型安全的方式访问它们是很好的。 幸运的是，有一个简单的方法可以达到这个效果。 这个想法是参数化键（key）而不是容器。 然后将参数化的键提交给容器以插入或检索值。 泛型类型系统用于保证值的类型与其键一致。

作为这种方法的一个简单示例，请考虑一个 `Favorites` 类，它允许其客户端保存和检索任意多种类型的 favorite 实例。 该类型的 `Class` 对象将扮演参数化键的一部分。其原因是这 `Class` 类是泛型的。 类的类型从字面上来说不是简单的 `Class` ，而是 `Class<T>` 。 例如， `String.class` 的类型为 `Class<String>` ， `Integer.class` 的类型为 `Class<Integer>` 。 当在方法中传递字面类传递编译时和运行时类型信息时，它被称为类型令牌（type token）[Bracha04]。

`Favorites` 类的 `API` 很简单。 它看起来就像一个简单 `Map` 类，除了该键是参数化的以外。 客户端在设置和获取 favorites 实例时呈现一个 `Class` 对象。 如下是 API：

```
// Typesafe heterogeneous container pattern - API
public class Favorites {
    public <T> void putFavorite(Class<T> type, T instance);
    public <T> T getFavorite(Class<T> type);
}
```

下面是一个演示 `Favorites` 类，保存，检索和打印喜欢的 `String` ， `Integer` 和 `Class` 实例：

```
// Typesafe heterogeneous container pattern - client
public static void main(String[] args) {
    Favorites f = new Favorites();
    f.putFavorite(String.class, "Java");
    f.putFavorite(Integer.class, 0xcafebabe);
    f.putFavorite(Class.class, Favorites.class);

    String favoriteString = f.getFavorite(String.class);
    int favoriteInteger = f.getFavorite(Integer.class);
    Class<?> favoriteClass = f.getFavorite(Class.class);
    System.out.printf("%s %x %s%n", favoriteString,
        favoriteInteger, favoriteClass.getName());
}
```

正如你所期望的，这个程序打印 `Java cafebabe Favorites`。 请注意，顺便说一下， `Java` 的 `printf` 方法与 C 语言的不同之处在于，你应该在 C 中使用 \n 的地方改用 %n。%n 用于生成适用于特定平台的行分隔符，在大多数平台上面的值为 \n，但并不是所有平台的分隔符都为 \n。

`Favorites` 实例是类型安全的：当你请求一个字符串时它永远不会返回一个整数。 它也是异构的：与普通 Map 不同，所有的键都是不同的类型。 因此，我们将 `Favorites` 称为类型安全异构容器（typesafe heterogeneous container）。

Favorites 的实现非常小巧。 这是完整的代码：

```
// Typesafe heterogeneous container pattern - implementation
public class Favorites {
    private Map<Class<?>, Object> favorites = new HashMap<>();

    public<T> void putFavorite(Class<T> type, T instance) {
        favorites.put(Objects.requireNonNull(type), instance);
    }

    public<T> T getFavorite(Class<T> type) {
        return type.cast(favorites.get(type));
    }
}
```

这里有一些微妙的事情发生。 每个 `Favorites` 实例都由一个名为 `favorites` 私有的 `Map<Class<?>, Object>` 来支持。 你可能认为无法将任何内容放入此 `Map` 中，因为这是无限定的通配符类型，但事实恰恰相反。 需要注意的是通配符类型是嵌套的：它不是通配符类型的 `Map` 类型，而是键的类型。 这意味着每个键都可以有不同的参数化类型：一个可以是 `Class<String>`，下一个 `Class<Integer>` 等等。 这就是异构的由来。

接下来要注意的是， `favorites` 的 `Map` 的值类型只是 `Object`。 换句话说， `Map` 不保证键和值之间的类型关系，即每个值都是由其键表示的类型。 事实上，Java 的类型系统并不足以表达这一点。 但是我们知道这是真的，并在检索一个 `favorite` 时利用了这点。

`putFavorite` 实现很简单：只需将给定的 `Class` 对象映射到给定的 `favorites` 的实例即可。如上所述，这丢弃了键和值之间的“类型联系（type linkage）”；无法知道这个值是不是键的一个实例。但没关系，因为 `getFavorites` 方法可以并且确实重新建立这种关联。

`getFavorite` 的实现比 `putFavorite` 更复杂。 首先，它从 `favorites` Map 中获取与给定 `Class` 对象相对应的值。 这是返回的正确对象引用，但它具有错误的编译时类型：它是 Object（favorites map 的值类型），我们需要返回类型 `T` 。因此， `getFavorite` 实现动态地将对象引用转换为 `Class` 对象表示的类型，使用 `Class` 的 `cast` 方法。

`cast` 方法是 Java 的 `cast` 操作符的动态模拟。它只是检查它的参数是否由 `Class` 对象表示的类型的实例。如果是，它返回参数；否则会抛出 `ClassCastException` 异常。我们知道，假设客户端代码能够干净地编译， `getFavorite` 中的强制转换不会抛出 `ClassCastException` 异常。 也就是说， `favorites map` 中的值始终与其键的类型相匹配。

那么这个 `cast` 方法为我们做了什么，因为它只是返回它的参数？ `cast` 的签名充分利用了 `Class` 类是泛型的事实。 它的返回类型是 `Class` 对象的类型参数：

```
public class Class<T> {
    T cast(Object obj);
}
```

这正是 `getFavorite` 方法所需要的。 这正是确保 `Favorites` 类型安全，而不用求助一个未经检查的强制转换的 `T` 类型。

`Favorites` 类有两个限制值得注意。 首先，恶意客户可以通过使用原始形式的 `Class` 对象，轻松破坏 `Favorites` 实例的类型安全。 但生成的客户端代码在编译时会生成未经检查的警告。 这与正常的集合实现（如 `HashSet` 和 `HashMap` ）没有什么不同。 通过使用原始类型 `HashSet` （条目 26），可以轻松地将字符串放入 `HashSet<Integer>` 中。 也就是说，如果你愿意为此付出一点代价，就可以拥有运行时类型安全性。 确保 `Favorites` 永远不违反类型不变的方法是，使 putFavorite 方法检查该实例是否由 `type` 表示类型的实例，并且我们已经知道如何执行此操作。只需使用动态转换：

```
// Achieving runtime type safety with a dynamic cast
public<T> void putFavorite(Class<T> type, T instance) {
    favorites.put(type, type.cast(instance));
}
```

`java.util.Collections` 中有一些集合包装类，可以发挥相同的诀窍。 它们被称为 `checkedSet` ， `checkedList` ， `checkedMap` 等等。 他们的静态工厂除了一个集合（或 `Map` ）之外还有一个 `Class` 对象（或两个）。 静态工厂是泛型方法，确保 `Class` 对象和集合的编译时类型匹配。 包装类为它们包装的集合添加了具体化。 例如，如果有人试图将 `Coin` 放入你的 `Collection<Stamp>` 中，则包装类在运行时会抛出 `ClassCastException` 。 这些包装类对于追踪在混合了泛型和原始类型的应用程序中添加不正确类型的元素到集合的客户端代码很有用。

Favorites 类的第二个限制是它不能用于不可具体化的（non-reifiable）类型（详见第 28 条）。 换句话说，你可以保存你最喜欢的 `String` 或 `String[]`，但不能保存 `List<String>`。 如果你尝试保存你最喜欢的 `List<String>`，程序将不能编译。 原因是无法获取 `List<String>` 的 `Class` 对象。 `List<String>.class` 是语法错误，也是一件好事。 `List<String>` 和 `List<Integer>` 共享一个 `Class` 对象，即 `List.class`。 如果“字面类型（type literals）” `List<String> .clas` s 和 `List<Integer>.class` 合法并返回相同的对象引用，那么它会对 `Favorites` 对象的内部造成严重破坏。 对于这种限制，没有完全令人满意的解决方法。

`Favorites` 使用的类型令牌 type tokens) 是无限制的： `getFavorite` 和 `putFavorite` 接受任何 `Class` 对象。 有时你可能需要限制可传递给方法的类型。 这可以通过一个有限定的类型令牌来实现，该令牌只是一个类型令牌，它使用限定的类型参数（详见第 30 条）或限定的通配符（详见第 31 条）来放置可以表示的类型的边界。

注解 API（详见第 39 条）广泛使用限定类型的令牌。 例如，以下是在运行时读取注解的方法。 此方法来自 `AnnotatedElement` 接口，该接口由表示类，方法，属性和其他程序元素的反射类型实现：

```
public <T extends Annotation>
    T getAnnotation(Class<T> annotationType);
```

参数 `annotationType` 是表示注解类型的限定类型令牌。 该方法返回该类型的元素的注解（如果它有一个）；如果没有，则返回 `null`。 本质上，注解元素是一个类型安全的异构容器，其键是注解类型。

假设有一个 `Class<?>` 类型的对象，并且想要将它传递给需要限定类型令牌（如 `getAnnotation`）的方法。 可以将对象转换为 `Class<? extends Annotation>`，但是这个转换没有被检查，所以它会产生一个编译时警告（详见第 52 条）。 幸运的是， `Class` 类提供了一种安全（动态）执行这种类型转换的实例方法。 该方法被称为 `asSubclass`，并且它转换所调用的 `Class` 对象来表示由其参数表示的类的子类。 如果转换成功，该方法返回它的参数；如果失败，则抛出 `ClassCastException` 异常。

以下是如何使用 `asSubclass` 方法在编译时读取类型未知的注解。 此方法编译时没有错误或警告：

```
// Use of asSubclass to safely cast to a bounded type token
static Annotation getAnnotation(AnnotatedElement element,
                                String annotationTypeName) {
    Class<?> annotationType = null; // Unbounded type token
    try {
        annotationType = Class.forName(annotationTypeName);
    } catch (Exception ex) {
        throw new IllegalArgumentException(ex);
    }
    return element.getAnnotation(
        annotationType.asSubclass(Annotation.class));
}
```

总之，泛型 API 的通常用法（以集合 API 为例）限制了每个容器的固定数量的类型参数。 你可以通过将类型参数放在键上而不是容器上来解决此限制。 可以使用 `Class` 对象作为此类型安全异构容器的键。 以这种方式使用的 `Class` 对象称为类型令牌。 也可以使用自定义键类型。 例如，可以有一个表示数据库行（容器）的 `DatabaseRow` 类型和一个泛型类型 `Column<T>` 作为其键。

Java 支持两种引用类型的特殊用途的系列：一种称为枚举类型的类和一种称为注解类型的接口。 本章讨论使用这些类型系列的最佳实践。

34. 使用枚举类型替代整型常量

枚举是其合法值由一组固定的常量组成的一种类型，例如一年中的季节，太阳系中的行星或一副扑克牌中的花色。 在将枚举类型添加到该语言之前，表示枚举类型的常见模式是声明一组名为 `int` 的常量，每个类型的成员都有一个常量：

```
// The int enum pattern - severely deficient!
public static final int APPLE_FUJI         = 0;
public static final int APPLE_PIPPIN       = 1;
public static final int APPLE_GRANNY_SMITH = 2;

public static final int ORANGE_NAVEL  = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD  = 2;。
```

这种被称为 `int` 枚举模式的技术有许多缺点。 它没有提供类型安全的方式，也没有提供任何表达力。 如果你将一个 `Apple` 传递给一个需要 `Orange` 的方法，那么编译器不会出现警告，还会用 == 运算符比较 `Apple` 与 `Orange`，或者更糟糕的是：

```
// Tasty citrus flavored applesauce!
int i = (APPLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN;
```

请注意，每个 `Apple` 常量的名称前缀为 `APPLE_`，每个 `Orange` 常量的名称前缀为 `ORANGE_`。 这是因为 Java 不为 `int` 枚举组提供名称空间。 当两个 `int` a 枚举组具有相同的命名常量时，前缀可以防止名称冲突，例如在 `ELEMENT_MERCURY` 和 `PLANET_MERCURY` 之间。

使用 `int` 枚举的程序很脆弱。 因为 `int` 枚举是编译时常量[JLS，4.12.4]，所以它们的 `int` 值被编译到使用它们的客户端中[JLS，13.1]。 如果与 `int` 枚举关联的值发生更改，则必须重新编译其客户端。 如果没有，客户仍然会运行，但他们的行为将是不正确的。

没有简单的方法将 `int` 枚举常量转换为可打印的字符串。 如果你打印这样一个常量或者从调试器中显示出来，你看到的只是一个数字，这不是很有用。 没有可靠的方法来迭代组中的所有 `int` 枚举常量，甚至无法获得 `int` 枚举组的大小。

你可能会遇到这种模式的变体，其中使用了字符串常量来代替 `int` 常量。 这种称为字符串枚举模式的变体更不理想。 尽管它为常量提供了可打印的字符串，但它可以导致初级用户将字符串常量硬编码为客户端代码，而不是使用属性名称。 如果这种硬编码的字符串常量包含书写错误，它将在编译时逃脱检测并导致运行时出现错误。 此外，它可能会导致性能问题，因为它依赖于字符串比较。

幸运的是，Java 提供了一种避免 `int` 和 `String` 枚举模式的所有缺点的替代方法，并提供了许多额外的好处。 它是枚举类型[JLS，8.9]。 以下是它最简单的形式：

```
public enum Apple  { FUJI, PIPPIN, GRANNY_SMITH }
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

从表面上看，这些枚举类型可能看起来与其他语言类似，比如 C，C++和 C#，但事实并非如此。 Java 的枚举类型是完整的类，比其他语言中的其他语言更强大，其枚举本质本上是 `int` 值。

Java 枚举类型背后的基本思想很简单：它们是通过公共静态 `final` 属性为每个枚举常量导出一个实例的类。 由于没有可访问的构造方法，枚举类型实际上是 `final` 的。 由于客户既不能创建枚举类型的实例也不能继承它，除了声明的枚举常量外，不能有任何实例。 换句话说，枚举类型是实例控制的（第 6 页）。 它们是单例（详见第 3 条）的泛型化，基本上是单元素的枚举。

枚举提供了编译时类型的安全性。 如果声明一个参数为 `Apple` 类型，则可以保证传递给该参数的任何非空对象引用是三个有效 `Apple` 值中的一个。 尝试传递错误类型的值将导致编译时错误，因为会尝试将一个枚举类型的表达式分配给另一个类型的变量，或者使用 `==` 运算符来比较不同枚举类型的值。

具有相同名称常量的枚举类型可以和平共存，因为每种类型都有其自己的名称空间。 可以在枚举类型中添加或重新排序常量，而无需重新编译其客户端，因为导出常量的属性在枚举类型与其客户端之间提供了一层隔离：常量值不会编译到客户端，因为它们位于 `int` 枚举模式中。 最后，可以通过调用其 `toString` 方法将枚举转换为可打印的字符串。

除了纠正 `int` 枚举的缺陷之外，枚举类型还允许添加任意方法和属性并实现任意接口。 它们提供了所有 `Object` 方法的高质量实现（第 3 章），它们实现了 `Comparable`（详见第 14 条）和 `Serializable`（第 12 章），并针对枚举类型的可任意改变性设计了序列化方式。

那么，为什么你要添加方法或属性到一个枚举类型？ 对于初学者，可能想要将数据与其常量关联起来。 例如，我们的 `Apple` 和 `Orange` 类型可能会从返回水果颜色的方法或返回水果图像的方法中受益。 还可以使用任何看起来合适的方法来增强枚举类型。 枚举类型可以作为枚举常量的简单集合，并随着时间的推移而演变为全功能抽象。

对于丰富的枚举类型的一个很好的例子，考虑我们太阳系的八颗行星。 每个行星都有质量和半径，从这两个属性可以计算出它的表面重力。 从而在给定物体的质量下，计算出一个物体在行星表面上的重量。 下面是这个枚举类型。 每个枚举常量之后的括号中的数字是传递给其构造方法的参数。 在这种情况下，它们是地球的质量和半径：

```
// Enum type with data and behavior
public enum Planet {
    MERCURY(3.302e+23, 2.439e6),
    VENUS  (4.869e+24, 6.052e6),
    EARTH  (5.975e+24, 6.378e6),
```

```
    MARS   (6.419e+23, 3.393e6),
    JUPITER(1.899e+27, 7.149e7),
    SATURN (5.685e+26, 6.027e7),
    URANUS (8.683e+25, 2.556e7),
    NEPTUNE(1.024e+26, 2.477e7);

    private final double mass;           // In kilograms
    private final double radius;         // In meters
    private final double surfaceGravity; // In m / s^2
    // Universal gravitational constant in m^3 / kg s^2
    private static final double G = 6.67300E-11;

    // Constructor
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
        surfaceGravity = G * mass / (radius * radius);
    }

    public double mass()           { return mass; }
    public double radius()         { return radius; }
    public double surfaceGravity() { return surfaceGravity; }

    public double surfaceWeight(double mass) {
        return mass * surfaceGravity;  // F = ma
    }
}
```

编写一个丰富的枚举类型比如 `Planet` 很容易。 **要将数据与枚举常量相关联，请声明实例属性并编写一个构造方法，构造方法带有数据并将数据保存在属性中。** 枚举本质上是不变的，所以所有的属性都应该是 `final` 的（详见第 17 条）。 属性可以是公开的，但最好将它们设置为私有并提供公共访问方法（详见第 16 条）。 在 `Planet` 的情况下，构造方法还计算和存储表面重力，但这只是一种优化。 每当重力被 `SurfaceWeight` 方法使用时，它可以从质量和半径重新计算出来，该方法返回它在由常数表示的行星上的重量。

虽然 `Planet` 枚举很简单，但它的功能非常强大。 这是一个简短的程序，它将一个物体在地球上的重量（任何单位），打印一个漂亮的表格，显示该物体在所有八个行星上的重量（以相同单位）：

```
public class WeightTable {
   public static void main(String[] args) {
      double earthWeight = Double.parseDouble(args[0]);
      double mass = earthWeight / Planet.EARTH.surfaceGravity();
      for (Planet p : Planet.values())
          System.out.printf("Weight on %s is %f%n",
                            p, p.surfaceWeight(mass));
      }
}
```

请注意，`Planet` 和所有枚举一样，都有一个静态 `values` 方法，该方法以声明的顺序返回其值的数组。另请注意，`toString` 方法返回每个枚举值的声明名称，使 `println` 和 `printf` 可以轻松打印。如果你对此字符串表示形式不满意，可以通过重写 `toString` 方法来更改它。这是使用命令行参数 185 运行 `WeightTable` 程序（不重写 `toString`）的结果：

```
Weight on MERCURY is 69.912739
Weight on VENUS is 167.434436
Weight on EARTH is 185.000000
Weight on MARS is 70.226739
Weight on JUPITER is 467.990696
Weight on SATURN is 197.120111
Weight on URANUS is 167.398264
Weight on NEPTUNE is 210.208751
```

直到 2006 年，在 Java 中加入枚举两年之后，冥王星不再是一颗行星。这引发了一个问题：「当你从枚举类型中移除一个元素时会发生什么？」答案是，任何不引用移除元素的客户端程序都将继续正常工作。所以，举例来说，我们的 `WeightTable` 程序将会打印一个少一行的表格。那么客户端程序引用删除的元素（在本例中是 `Planet.Pluto`）会如何？如果重新编译客户端程序，编译将会失败并在引用删除的星球的行处提供有用的错误消息; 如果无法重新编译客户端，它将在运行时从此行中引发有用的异常。这是你所希望的最好的行为，远远好于你用 `int` 枚举模式得到的结果。

一些与枚举常量相关的行为只需要在定义枚举的类或包中使用。这些行为最好以私有或包级私有方式实现。然后每个常量携带一个隐藏的行为集合，这些行为允许包含枚举的类或包在呈现常量时作出适当的反应。与其他类一样，除非你有一个令人信服的理由将枚举方法暴露给它的客户端，否则将其声明为私有的，如果需要的话将其声明为包级私有（详见第 15 条）。

如果一个枚举是广泛使用的，它应该是一个顶级类; 如果它的使用与特定的顶级类绑定，它应该是该顶级类的成员类（详见第 24 条）。例如，`java.math.RoundingMode` 枚举表示小数部分的舍入模式。`BigDecimal` 类使用了这些舍入模式，但它们提供了一种有用的抽象，它并不与 `BigDecimal` 有根本的联系。通过将 `RoundingMode` 设置为顶层枚举，类库设计人员鼓励任何需要舍入模式的程序员重用此枚举，从而提高跨 `API` 的一致性。

```
// Enum type that switches on its own value - questionable
public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;

    // Do the arithmetic operation represented by this constant
    public double apply(double x, double y) {
        switch(this) {
            case PLUS:   return x + y;
            case MINUS:  return x - y;
            case TIMES:  return x * y;
            case DIVIDE: return x / y;
        }
        throw new AssertionError("Unknown op: " + this);
    }
}
```

此代码有效，但不是很漂亮。 如果没有 `throw` 语句，就不能编译，因为该方法的结束在技术上是可达到的，尽管它永远不会被达到[JLS，14.21]。 更糟的是，代码很脆弱。 如果添加新的枚举常量，但忘记向 `switch` 语句添加相应的条件，枚举仍然会编译，但在尝试应用新操作时，它将在运行时失败。

幸运的是，有一种更好的方法可以将不同的行为与每个枚举常量关联起来：在枚举类型中声明一个抽象的 apply 方法，并用常量特定的类主体中的每个常量的具体方法重写它。 这种方法被称为特定于常量（constant-specific）的方法实现：

```
// Enum type with constant-specific method implementations
public enum Operation {
  PLUS  {public double apply(double x, double y){return x + y;}},
  MINUS {public double apply(double x, double y){return x - y;}},
  TIMES {public double apply(double x, double y){return x * y;}},
  DIVIDE{public double apply(double x, double y){return x / y;}};

  public abstract double apply(double x, double y);
}
```

如果向第二个版本的操作添加新的常量，则不太可能会忘记提供 `apply` 方法，因为该方法紧跟在每个常量声明之后。 万一忘记了，编译器会提醒你，因为枚举类型中的抽象方法必须被所有常量中的具体方法重写。

特定于常量的方法实现可以与特定于常量的数据结合使用。 例如，以下是 `Operation` 的一个版本，它重写 `toString` 方法以返回通常与该操作关联的符号：

```
// Enum type with constant-specific class bodies and data
public enum Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };

    private final String symbol;

    Operation(String symbol) { this.symbol = symbol; }

    @Override public String toString() { return symbol; }

    public abstract double apply(double x, double y);
```

```
}
```

显示的 `toString` 实现可以很容易地打印算术表达式，正如这个小程序所展示的那样：

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    for (Operation op : Operation.values())
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

以 2 和 4 作为命令行参数运行此程序会生成以下输出：

```
2.000000 + 4.000000 = 6.000000
2.000000 - 4.000000 = -2.000000
2.000000 * 4.000000 = 8.000000
2.000000 / 4.000000 = 0.500000
```

枚举类型具有自动生成的 `valueOf(String)` 方法，该方法将常量名称转换为常量本身。 如果在枚举类型中重写 `toString` 方法，请考虑编写 `fromString` 方法将自定义字符串表示法转换回相应的枚举类型。 下面的代码（类型名称被适当地改变）将对任何枚举都有效，只要每个常量具有唯一的字符串表示形式：

```
// Implementing a fromString method on an enum type
private static final Map<String, Operation> stringToEnum =
        Stream.of(values()).collect(
            toMap(Object::toString, e -> e));

// Returns Operation for string, if any
public static Optional<Operation> fromString(String symbol) {
    return Optional.ofNullable(stringToEnum.get(symbol));
}
```

请注意， `Operation` 枚举常量被放在 `stringToEnum` 的 `map` 中，它来自于创建枚举常量后运行的静态属性初始化。前面的代码在 `values()` 方法返回的数组上使用流（第 7 章）；在 Java 8 之前，我们创建一个空的 `hashMap` 并遍历值数组，将字符串到枚举映射插入到 `map` 中，如果愿意，仍然可以这样做。但请注意，尝试让每个常量都将自己放入来自其构造方法的 `map` 中不起作用。这会导致编译错误，这是好事，因为如果它是合法的，它会在运行时导致 `NullPointerException` 。除了编译时常量属性（详见第 34 条）之外，枚举构造方法不允许访问枚举的静态属性。此限制是必需的，因为静态属性在枚举构造方法运行时尚未初始化。这种限制的一个特例是枚举常量不能从构造方法中相互访问。

另请注意， `fromString` 方法返回一个 `Optional<String>` 。 这允许该方法指示传入的字符串不代表有效的操作，并且强制客户端面对这种可能性（详见第 55 条）。

特定于常量的方法实现的一个缺点是它们使得难以在枚举常量之间共享代码。 例如，考虑一个代表工资包中的工作天数的枚举。 该枚举有一个方法，根据工人的基本工资（每小时）和当天工作的分钟数计算当天工人的工资。 在五个工作日内，任何超过正常工作时间的工作都会产生加班费; 在两个周末的日子里，所有工作都会产生加班费。 使用 `switch` 语句，通过将多个 `case` 标签应用于两个代码片段中的每一个，可以轻松完成此计算：

```
// Enum that switches on its value to share code - questionable
enum PayrollDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY;

    private static final int MINS_PER_SHIFT = 8 * 60;

    int pay(int minutesWorked, int payRate) {
        int basePay = minutesWorked * payRate;

        int overtimePay;
        switch(this) {
          case SATURDAY: case SUNDAY: // Weekend
            overtimePay = basePay / 2;
            break;
          default: // Weekday
            overtimePay = minutesWorked <= MINS_PER_SHIFT ?
              0 : (minutesWorked - MINS_PER_SHIFT) * payRate / 2;
        }

        return basePay + overtimePay;
    }
}
```

这段代码无可否认是简洁的，但从维护的角度来看是危险的。 假设你给枚举添加了一个元素，可能是一个特殊的值来表示一个假期，但忘记在 `switch` 语句中添加一个相应的 `case` 条件。 该程序仍然会编译，但付费方法会默默地为工作日支付相同数量的休假日，与普通工作日相同。

要使用特定于常量的方法实现安全地执行工资计算，必须为每个常量重复加班工资计算，或将计算移至两个辅助方法，一个用于工作日，另一个用于周末，并调用适当的辅助方法来自每个常量。 这两种方法都会产生相当数量的样板代码，大大降低了可读性并增加了出错机会。

通过使用执行加班计算的具体方法替换 `PayrollDay` 上的抽象 `overtimePay` 方法，可以减少样板。 那么只有周末的日子必须重写该方法。 但是，这与 `switch` 语句具有相同的缺点：如果在不重写 `overtimePay` 方法的情况下添加另一天，则会默默继承周日计算方式。

你真正想要的是每次添加枚举常量时被迫选择加班费策略。 幸运的是，有一个很好的方法来实现这一点。 这个想法是将加班费计算移入私有嵌套枚举中，并将此策略枚举的实例传递给 `PayrollDay` 枚举的构造方法。 然后， `PayrollDay` 枚举将加班工资计算委托给策略枚举，从而无需在 `PayrollDay` 中实现 `switch` 语句或特定于常量的方法实现。 虽然这种模式不如 `switch` 语句简洁，但它更安全，更灵活：

```
// The strategy enum pattern
enum PayrollDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);

    private final PayType payType;

    PayrollDay(PayType payType) { this.payType = payType; }
    PayrollDay() { this(PayType.WEEKDAY); }  // Default

    int pay(int minutesWorked, int payRate) {
        return payType.pay(minutesWorked, payRate);
    }

    // The strategy enum type
    private enum PayType {
        WEEKDAY {
            int overtimePay(int minsWorked, int payRate) {
                return minsWorked <= MINS_PER_SHIFT ? 0 :
                  (minsWorked - MINS_PER_SHIFT) * payRate / 2;
            }
        },
        WEEKEND {
            int overtimePay(int minsWorked, int payRate) {
                return minsWorked * payRate / 2;
            }
        };

        abstract int overtimePay(int mins, int payRate);
        private static final int MINS_PER_SHIFT = 8 * 60;

        int pay(int minsWorked, int payRate) {
            int basePay = minsWorked * payRate;
            return basePay + overtimePay(minsWorked, payRate);
        }
    }
}
```

如果对枚举的 `switch` 语句不是实现常量特定行为的好选择，那么它们有什么好处呢?枚举类型的 `switch` 有利于用常量特定的行为增加枚举类型。例如，假设 `Operation` 枚举不在你的控制之下，你希望它有一个实例方法来返回每个相反的操作。你可以用以下静态方法模拟效果:

```
// Switch on an enum to simulate a missing method
public static Operation inverse(Operation op) {
    switch(op) {
        case PLUS:   return Operation.MINUS;
        case MINUS:  return Operation.PLUS;
        case TIMES:  return Operation.DIVIDE;
        case DIVIDE: return Operation.TIMES;
        default:  throw new AssertionError("Unknown op: " + op);
    }
}
```

如果某个方法不属于枚举类型，则还应该在你控制的枚举类型上使用此技术。 该方法可能需要用于某些用途，但通常不足以用于列入枚举类型。

一般而言，枚举通常在性能上与 `int` 常数相当。 枚举的一个小小的性能缺点是加载和初始化枚举类型存在空间和时间成本，但在实践中不太可能引人注意。

那么你应该什么时候使用枚举呢？ 任何时候使用枚举都需要一组常量，这些常量的成员在编译时已知。 当然，这包括“天然枚举类型”，如行星，星期几和棋子。 但是它也包含了其它你已经知道编译时所有可能值的集合，例如菜单上的选项，操作代码和命令行标志。 **一个枚举类型中的常量集不需要一直保持不变。** 枚举功能是专门设计用于允许二进制兼容的枚举类型的演变。

总之，枚举类型优于 `int` 常量的优点是令人信服的。 枚举更具可读性，更安全，更强大。 许多枚举不需要显式构造方法或成员，但其他人则可以通过将数据与每个常量关联并提供行为受此数据影响的方法而受益。 使用单一方法关联多个行为可以减少枚举。 在这种相对罕见的情况下，更喜欢使用常量特定的方法来枚举自己的值。 如果一些（但不是全部）枚举常量共享共同行为，请考虑策略枚举模式。

35. 使用实例属性替代序数

许多枚举通常与单个 `int` 值关联。所有枚举都有一个 `ordinal` 方法，它返回每个枚举常量类型的数值位置。你可能想从序数中派生一个关联的 `int` 值：

```
// Abuse of ordinal to derive an associated value - DON'T DO THIS
public enum Ensemble {
    SOLO,   DUET,   TRIO, QUARTET, QUINTET,
    SEXTET, SEPTET, OCTET, NONET,  DECTET;

    public int numberOfMusicians() { return ordinal() + 1; }
}
```

虽然这个枚举能正常工作，但对于维护来说则是一场噩梦。如果常量被重新排序，`numberOfMusicians` 方法将会中断。 如果你想添加一个与你已经使用的 `int` 值相关的第二个枚举常量，则没有那么好运了。 例如，为双四重奏（double quartet）添加一个常量可能会很好，它就像八重奏一样，由 8 位演奏家组成，但是没有办法做到这一点。

此外，如果没有给所有中间的 `int` 值添加常量，就不能为这个 `int` 值添加一个常量。例如，假设你想要添加一个常量，表示一个由 12 位演奏家组成的三重四重奏（triple quartet）。对于由 11 个演奏家组成的合奏曲，并没有标准的术语，因此你不得不为未使用的 `int` 值（11）添加一个虚拟常量（dummy constant）。最多看起来就是有些不好看。如果许多 `int` 值是未使用的，则是不切实际的。

幸运的是，这些问题有一个简单的解决方案。 **永远不要从枚举的序号中得出与它相关的值; 请将其保存在实例属性中：**

```
public enum Ensemble {
    SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),
    SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),
    NONET(9), DECTET(10), TRIPLE_QUARTET(12);

    private final int numberOfMusicians;

    Ensemble(int size) { this.numberOfMusicians = size; }
    public int numberOfMusicians() { return numberOfMusicians; }
}
```

枚举规范对此 `ordinal` 方法说道：“大多数程序员对这种方法没有用处。 它被设计用于基于枚举的通用数据结构，如 `EnumSet` 和 `EnumMap` 。“除非你在编写这样数据结构的代码，否则最好避免使用 `ordinal` 方法。

36. 使用 EnumSet 替代位属性

如果枚举类型的元素主要用于集合中，则传统上使用 int 枚举模式（详见第 34 条），将 2 的不同次幂赋值给每个常量：

```
// Bit field enumeration constants - OBSOLETE!
public class Text {
    public static final int STYLE_BOLD          = 1 << 0;  // 1
    public static final int STYLE_ITALIC        = 1 << 1;  // 2
    public static final int STYLE_UNDERLINE     = 1 << 2;  // 4
    public static final int STYLE_STRIKETHROUGH = 1 << 3;  // 8

    // Parameter is bitwise OR of zero or more STYLE_ constants
    public void applyStyles(int styles) { ... }
}
```

这种表示方式允许你使用按位或（or）运算将几个常量合并到一个称为位域（bit field）的集合中：

```
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```

位域表示还允许你使用按位算术有效地执行集合运算，如并集和交集。 但是位域具有 `int` 枚举常量等的所有缺点。 当打印为数字时，解释位域比简单的 `int` 枚举常量更难理解。 没有简单的方法遍历所有由位域表示的元素。 最后，必须预测在编写 API 时需要的最大位数，并相应地为位域（通常为 `int` 或 `long` ）选择一种类型。 一旦你选择了一个类型，你就不能在不改变 API 的情况下超过它的宽度（32 或 64 位）。

当需要传递一组常量时，一些优先使用枚举而不是 `int` 常量的程序员仍然坚持使用位域。 没有理由这样做，因为存在更好的选择。 `java.util` 包提供了 `EnumSet` 类来有效地表示从单个枚举类型中提取的值集合。 这个类实现了 `Set` 接口，提供了所有其他 `Set` 实现的丰富性，类型安全性和互操作性。 但是在内部，每个 `EnumSet` 都表示为一个位向量（bit vector）。 如果底层的枚举类型有 64 个或更少的元素，并且大多数情况下，整个 `EnumSet` 用单个 `long` 表示，所以它的性能与位域的性能相当。 批量操作（如 `removeAll` 和 `retainAll` ）是使用按位算术实现的，就像你手动操作位域一样。 但是完全避免了手动进行位操作导致的丑陋和潜在错误： `EnumSet` 为你完成这一困难工作。

下面是前一个使用枚举和枚举集合替代位域的示例。 它更短，更清晰，更安全：

```
// EnumSet - a modern replacement for bit fields
public class Text {
    public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }

    // Any Set could be passed in, but EnumSet is clearly best
    public void applyStyles(Set<Style> styles) { ... }
}
```

这里是将 `EnumSet` 实例传递给 `applyStyles` 方法的客户端代码。 `EnumSet` 类提供了一组丰富的静态工厂，可以轻松创建集合，其中一个代码如下所示：

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

请注意，`applyStyles` 方法采用 `Set<Style>` 而不是 `EnumSet<Style>` 参数。 尽管所有客户端都可能会将 `EnumSet` 传递给该方法，但接受接口类型而不是实现类型通常是很好的做法（详见第 64 条）。 这允许一个不寻常的客户端通过其他 Set 实现的可能性。

总之，**仅仅因为枚举类型将被用于集合中，就没有理由用位域来表示它。** `EnumSet` 类结合了位域的简洁性和性能以及条目 34 中所述的枚举类型的所有优点。截至 Java 9， `EnumSet` 的一个真正缺点是不能创建一个不可变的 `EnumSet`，但是在之后的发行版本中这可能会得到纠正。 同时，你可以用 `Collections.unmodifiableSet` 封装一个 `EnumSet`，但是简洁性和性能会受到影响。

37. 使用 EnumMap 替代序数索引

有时可能会看到使用 `ordinal` 方法（详见第 35 条）来索引到数组或列表的代码。 例如，考虑一下这个简单的类来代表一种植物：

```
class Plant {
    enum LifeCycle { ANNUAL, PERENNIAL, BIENNIAL }
    final String name;
    final LifeCycle lifeCycle;

    Plant(String name, LifeCycle lifeCycle) {
        this.name = name;
        this.lifeCycle = lifeCycle;
    }

    @Override public String toString() {
        return name;
    }
}
```

现在假设你有一组植物代表一个花园，想要列出这些由生命周期组织的植物 (一年生，多年生，或双年生)。为此，需要构建三个集合，每个生命周期作为一个，并遍历整个花园，将每个植物放置在适当的集合中。一些程序员可以通过将这些集合放入一个由生命周期序数索引的数组中来实现这一点：

```
// Using ordinal() to index into an array - DON'T DO THIS!
Set<Plant>[] plantsByLifeCycle =
    (Set<Plant>[]) new Set[Plant.LifeCycle.values().length];

for (int i = 0; i < plantsByLifeCycle.length; i++)
    plantsByLifeCycle[i] = new HashSet<>();

for (Plant p : garden)
    plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);
```

```
// Print the results
for (int i = 0; i < plantsByLifeCycle.length; i++) {
    System.out.printf("%s: %s%n",
        Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
}
```

这种方法是有效的，但充满了问题。 因为数组不兼容泛型（详见第 28 条），程序需要一个未经检查的转换，并且不会干净地编译。 由于该数组不知道索引代表什么，因此必须手动标记索引输出。 但是这种技术最严重的问题是，当你访问一个由枚举序数索引的数组时，你有责任使用正确的 `int` 值; `int` 不提供枚举的类型安全性。 如果你使用了错误的值，程序会默默地做错误的事情，如果你幸运的话，抛出一个 `ArrayIndexOutOfBoundsException` 异常。

有一个更好的方法来达到同样的效果。 该数组有效地用作从枚举到值的映射，因此不妨使用 `Map` 。 更具体地说，有一个非常快速的 `Map` 实现，设计用于枚举键，称为 `java.util.EnumMap` 。 下面是当程序重写为使用 `EnumMap` 时的样子：

```
// Using an EnumMap to associate data with an enum
Map<Plant.LifeCycle, Set<Plant>>  plantsByLifeCycle =
    new EnumMap<>(Plant.LifeCycle.class);

for (Plant.LifeCycle lc : Plant.LifeCycle.values())
    plantsByLifeCycle.put(lc, new HashSet<>());

for (Plant p : garden)
    plantsByLifeCycle.get(p.lifeCycle).add(p);

System.out.println(plantsByLifeCycle);
```

这段程序更简短，更清晰，更安全，运行速度与原始版本相当。 没有不安全的转换; 无需手动标记输出，因为 `map` 键是知道如何将自己转换为可打印字符串的枚举; 并且不可能在计算数组索引时出错。 `EnumMap` 与序数索引数组的速度相当，其原因是 `EnumMap` 内部使用了这样一个数组，但它对程序员的隐藏了这个实现细节，将 Map 的丰富性和类型安全性与数组的速度相结合。 请注意， `EnumMap` 构造方法接受键类 `Class` 型的 Class 对象：这是一个有限定的类型令牌（bounded type token），它提供运行时的泛型类型信息（条目 33）。

通过使用 `stream` （详见第 45 条）来管理 `Map` ，可以进一步缩短以前的程序。 以下是最简单的基于 `stream` 的代码，它们在很大程度上重复了前面示例的行为：

```
// Naive stream-based approach - unlikely to produce an EnumMap!
System.out.println(Arrays.stream(garden)
        .collect(groupingBy(p -> p.lifeCycle)));
```

这个代码的问题在于它选择了自己的 `Map` 实现，实际上它不是 `EnumMap` ，所以它不会与显式 `EnumMap` 的版本的空间和时间性能相匹配。 为了解决这个问题，使用 `Collectors.groupingBy` 的三个参数形式的方法，它允许调用者使用 mapFactory 参数指定 map 的实现：

```
// Using a stream and an EnumMap to associate data with an enum
System.out.println(Arrays.stream(garden)
        .collect(groupingBy(p -> p.lifeCycle,
() -> new EnumMap<>(LifeCycle.class), toSet())));
```

这样的优化在像这样的示例程序中是不值得的，但是在大量使用 `Map` 的程序中可能是至关重要的。

基于 `stream` 版本的行为与 `EmumMap` 版本的行为略有不同。 `EnumMap` 版本总是为每个工厂生命周期生成一个嵌套 `map` 类，而如果花园包含一个或多个具有该生命周期的植物时，则基于流的版本才会生成嵌套 `map` 类。 因此，例如，如果花园包含一年生和多年生植物但没有两年生的植物，`plantByLifeCycle` 的大小在 `EnumMap` 版本中为三个，在两个基于流的版本中为两个。

你可能会看到数组索引（两次）的数组，用序数来表示从两个枚举值的映射。例如，这个程序使用这样一个数组来映射两个阶段到一个阶段转换（phase transition）（液体到固体表示凝固，液体到气体表示沸腾等等）：

```
// Using ordinal() to index array of arrays - DON'T DO THIS!
public enum Phase {
    SOLID, LIQUID, GAS;

    public enum Transition {
        MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;
        // Rows indexed by from-ordinal, cols by to-ordinal
        private static final Transition[][] TRANSITIONS = {
            { null,    MELT,     SUBLIME },
            { FREEZE,  null,     BOIL    },
            { DEPOSIT, CONDENSE, null    }
        };

        // Returns the phase transition from one phase to another
        public static Transition from(Phase from, Phase to) {
            return TRANSITIONS[from.ordinal()][to.ordinal()];
        }
    }
}
```

这段程序可以运行，甚至可能显得优雅，但外观可能是骗人的。 就像前面显示的简单的花园示例一样，编译器无法知道序数和数组索引之间的关系。 如果在转换表中出错或者在修改 `Phase` 或 `Phase.Transition` 枚举类型时忘记更新它，则程序在运行时将失败。 失败可能是 `ArrayIndexOutOfBoundsException` ， `NullPointerException` 或（更糟糕的）沉默无提示的错误行为。 即使非空条目的数量较小，表格的大小也是 phase 的个数的平方。

同样，可以用 `EnumMap` 做得更好。 因为每个阶段转换都由一对阶段枚举来索引，所以最好将关系表示为从一个枚举（from 阶段）到第二个枚举（to 阶段）到结果（阶段转换）的 `map` 。 与阶段转换相关的两个阶段最好通过将它们与阶段转换枚举相关联来捕获，然后可以用它来初始化嵌套的 EnumMap：

```
// Using a nested EnumMap to associate data with enum pairs
public enum Phase {
   SOLID, LIQUID, GAS;

   public enum Transition {
      MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
      BOIL(LIQUID, GAS),   CONDENSE(GAS, LIQUID),
      SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);

      private final Phase from;
      private final Phase to;
      Transition(Phase from, Phase to) {
         this.from = from;
         this.to = to;
      }

      // Initialize the phase transition map
      private static final Map<Phase, Map<Phase, Transition>>
        m = Stream.of(values()).collect(groupingBy(t -> t.from,
         () -> new EnumMap<>(Phase.class),
         toMap(t -> [t.to](http://t.to), t -> t,
            (x, y) -> y, () -> new EnumMap<>(Phase.class))));

      public static Transition from(Phase from, Phase to) {
         return m.get(from).get(to);
      }
   }
}
```

初始化阶段转换的 `map` 的代码有点复杂。 `map` 的类型是 `Map<Phase, Map<Phase, Transition>>` ，意思是「从（源）阶段映射到从（目标）阶段到阶段转换映射。」这个 `map` 的 `map` 使用两个收集器的级联序列进行初始化。 第一个收集器按源阶段对转换进行分组，第二个收集器使用从目标阶段到转换的映射创建一个 `EnumMap` 。 第二个收集器 `((x, y) -> y))` 中的合并方法未使用；仅仅因为我们需要指定一个 `map` 工厂才能获得一个 `EnumMap` ，并且 `Collectors` 提供伸缩式工厂，这是必需的。 本书的前一版使用显式迭代来初始化阶段转换 `map` 。 代码更详细，但可以更容易理解。

现在假设想为系统添加一个新阶段：等离子体或电离气体。 这个阶段只有两个转变：电离，将气体转化为等离子体; 和去离子，将等离子体转化为气体。 要更新基于数组的程序，必须将一个新的常量添加到 `Phase` ，将两个两次添加到 `Phase.Transition` ，并用新的十六个元素版本替换原始的九元素阵列数组。 如果向数组中添加太多或太少的元素或者将元素乱序放置，那么如果运气不佳：程序将会编译，但在运行时会失败。 要更新基于 `EnumMap` 的版本，只需将 `PLASMA` 添加到阶段列表中，并将 `IONIZE(GAS, PLASMA)` 和 `DEIONIZE(PLASMA, GAS)` 添加到阶段转换列表中：

```
// Adding a new phase using the nested EnumMap implementation
public enum Phase {

    SOLID, LIQUID, GAS, PLASMA;

    public enum Transition {
        MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
        BOIL(LIQUID, GAS),   CONDENSE(GAS, LIQUID),
        SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID),
        IONIZE(GAS, PLASMA), DEIONIZE(PLASMA, GAS);
        ... // Remainder unchanged
    }
}
```

该程序会处理所有其他事情，并且几乎不会出现错误。 在内部，`map` 的 `map` 是通过数组的数组实现的，因此在空间或时间上花费很少，以增加清晰度，安全性和易于维护。

为了简便起见，上面的示例使用 `null` 来表示状态更改的缺失（其从目标到源都是相同的）。这不是很好的实践，很可能在运行时导致 `NullPointerException`。为这个问题设计一个干净、优雅的解决方案是非常棘手的，而且结果程序足够长，以至于它们会偏离这个条目的主要内容。

总之，使用序数来索引数组很不合适：改用 EnumMap。 如果你所代表的关系是多维的，请使用 `EnumMap <..., EnumMap <... >>`。 应用程序员应该很少使用 `Enum.ordinal`（详见第 35 条），如果使用了，也是一般原则的特例。

38. 使用接口模拟可扩展的枚举

在几乎所有方面，枚举类型都优于本书第一版中描述的类型安全模式[Bloch01]。 从表面上看，一个例外涉及可扩展性，这在原始模式下是可能的，但不受语言结构支持。 换句话说，使用该模式，有可能使一个枚举类型扩展为另一个；使用语言功能特性，它不能这样做。 这不是偶然的。 大多数情况下，枚举的可扩展性是一个糟糕的主意。 令人困惑的是，扩展类型的元素是基类型的实例，反之亦然。 枚举基本类型及其扩展的所有元素没有好的方法。 最后，可扩展性会使设计和实现的很多方面复杂化。

也就是说，对于可扩展枚举类型至少有一个有说服力的用例，这就是操作码（operation codes），也称为 opcodes。 操作码是枚举类型，其元素表示某些机器上的操作，例如条目 34 中的 `Operation` 类型，它表示简单计算器上的功能。 有时需要让 API 的用户提供他们自己的操作，从而有效地扩展 API 提供的操作集。

幸运的是，使用枚举类型有一个很好的方法来实现这种效果。基本思想是利用枚举类型可以通过为 `opcode` 类型定义一个接口，并实现任意接口。例如，这里是来自条目 34 的 `Operation` 类型的可扩展版本：

```
// Emulated extensible enum using an interface
public interface Operation {
```

```
    double apply(double x, double y);
}

public enum BasicOperation implements Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };
    private final String symbol;

    BasicOperation(String symbol) {
        this.symbol = symbol;
    }

    @Override public String toString() {
        return symbol;
    }
}
```

虽然枚举类型（`BasicOperation`）不可扩展，但接口类型（`Operation`）是可以扩展的，并且它是用于表示 API 中的操作的接口类型。 你可以定义另一个实现此接口的枚举类型，并使用此新类型的实例来代替基本类型。 例如，假设想要定义前面所示的操作类型的扩展，包括指数运算和余数运算。 你所要做的就是编写一个实现 `Operation` 接口的枚举类型：

```
// Emulated extension enum
public enum ExtendedOperation implements Operation {
    EXP("^") {
        public double apply(double x, double y) {
            return Math.pow(x, y);
        }
    },
    REMAINDER("%") {
        public double apply(double x, double y) {
            return x % y;
        }
    };

    private final String symbol;

    ExtendedOperation(String symbol) {
```

```
        this.symbol = symbol;
    }

    @Override public String toString() {
        return symbol;
    }
}
```

只要 API 编写为接口类型（`Operation`），而不是实现（`BasicOperation`），现在就可以在任何可以使用基本操作的地方使用新操作。请注意，不必在枚举中声明 `apply` 抽象方法，就像您在具有实例特定方法实现的非扩展枚举中所做的那样（第 162 页）。 这是因为抽象方法（`apply`）是接口（`Operation`）的成员。

不仅可以在任何需要「基本枚举」的地方传递「扩展枚举」的单个实例，而且还可以传入整个扩展枚举类型，并使用其元素。 例如，这里是第 163 页上的一个测试程序版本，它执行之前定义的所有扩展操作：

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(ExtendedOperation.class, x, y);
}


private static <T extends Enum<T> & Operation> void test(
        Class<T> opEnumType, double x, double y) {
    for (Operation op : opEnumType.getEnumConstants())
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

注意，扩展的操作类型的类字面文字（`ExtendedOperation.class`）从 `main` 方法里传递给了 `test` 方法，用来描述扩展操作的集合。这个类的字面文字用作限定的类型令牌（详见第 33 条）。`opEnumType` 参数中复杂的声明（`<T extends Enum<T> & Operation> Class<T>`）确保了 `Class` 对象既是枚举又是 `Operation` 的子类，这正是遍历元素和执行每个元素相关联的操作时所需要的。

第二种方式是传递一个 `Collection<? extends Operation>`，这是一个限定通配符类型（详见第 31 条），而不是传递了一个 `class` 对象：

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(Arrays.asList(ExtendedOperation.values()), x, y);
}

private static void test(Collection<? extends Operation> opSet,
        double x, double y) {
    for (Operation op : opSet)
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

生成的代码稍微不那么复杂，`test` 方法灵活一点：它允许调用者将多个实现类型的操作组合在一起。另一方面，也放弃了在指定操作上使用 `EnumSet`（详见第 36 条）和 `EnumMap`（详见第 37 条）的能力。

上面的两个程序在运行命令行输入参数 4 和 2 时生成以下输出：

```
4.000000 ^ 2.000000 = 16.000000
4.000000 % 2.000000 = 0.000000
```

使用接口来模拟可扩展枚举的一个小缺点是，实现不能从一个枚举类型继承到另一个枚举类型。如果实现代码不依赖于任何状态，则可以使用默认实现（详见第 20 条）将其放置在接口中。在我们的 `Operation` 示例中，存储和检索与操作关联的符号的逻辑必须在 `BasicOperation` 和 `ExtendedOperation` 中重复。在这种情况下，这并不重要，因为很少的代码是冗余的。如果有更多的共享功能，可以将其封装在辅助类或静态辅助方法中，以消除代码冗余。

该条目中描述的模式在 `Java` 类库中有所使用。例如，`java.nio.file.LinkOption` 枚举类型实现了 `CopyOption` 和 `OpenOption` 接口。

总之，虽然不能编写可扩展的枚举类型，但是你可以编写一个接口来配合实现接口的基本的枚举类型，来对它进行模拟。 这允许客户端编写自己的枚举（或其它类型）来实现接口。如果 `API` 是根据接口编写的，那么在任何使用基本枚举类型实例的地方，都可以使用这些枚举类型实例。

39. 注解优于命名模式

过去，通常使用命名模式（naming patterns）来指示某些程序元素需要通过工具或框架进行特殊处理。 例如，在第 4 版之前，JUnit 测试框架要求其用户通过以 test[Beck04] 开始名称来指定测试方法。这种技术是有效的，但它有几个很大的缺点。 首先，拼写错误导致失败，但不会提示。 例如，假设意外地命名了测试方法 `tsetSafetyOverride` 而不是 `testSafetyOverride`。 `JUnit 3` 不会报错，但它也不会执行测试，导致错误的安全感。

命名模式的第二个缺点是无法确保它们仅用于适当的程序元素。 例如，假设调用了 `TestSafetyMechanisms` 类，希望 `JUnit 3` 能够自动测试其所有方法，而不管它们的名称如何。 同样， `JUnit 3` 也不会出错，但它也不会执行测试。

命名模式的第三个缺点是它们没有提供将参数值与程序元素相关联的好的方法。 例如，假设想支持只有在抛出特定异常时才能成功的测试类别。 异常类型基本上是测试的一个参数。 你可以使用一些精心设计的命名模式将异常类型名称编码到测试方法名称中，但这会变得丑陋和脆弱（详见第 62 条）。 编译器无法知道要检查应该命名为异常的字符串是否确实存在。 如果命名的类不存在或者不是异常，那么直到尝试运行测试时才会发现。

注解[JLS，9.7] 很好地解决了所有这些问题， `JUnit` 从第 4 版开始采用它们。在这个项目中，我们将编写我们自己的测试框架来显示注解的工作方式。 假设你想定义一个注解类型来指定自动运行的简单测试，并且如果它们抛出一个异常就会失败。 以下是名为 Test 的这种注解类型的定义：

```
// Marker annotation type declaration
import java.lang.annotation.*;

/**
 * Indicates that the annotated method is a test method.
 * Use only on parameterless static methods.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {

}
```

`Test` 注解类型的声明本身使用 `Retention` 和 `Target` 注解进行标记。 注解类型声明上的这种注解称为元注解。 `@Retention` （ `RetentionPolicy.RUNTIME` ）元注解指示 `Test` 注解应该在运行时保留。 没有它，测试工具就不会看到 `Test` 注解。 `@Target.get（ElementType.METHOD）` 元注解表明 `Test` 注解只对方法声明合法：它不能应用于类声明，属性声明或其他程序元素。

在 `Test` 注解声明之前的注释说：“仅在无参静态方法中使用”。如果编译器可以强制执行此操作是最好的，但它不能，除非编写注解处理器来执行此操作。 有关此主题的更多信息，请参阅 `javax.annotation.processing` 文档。 在缺少这种注解处理器的情况下，如果将 `Test` 注解放在实例方法声明或带有一个或多个参数的方法上，那么测试程序仍然会编译，并将其留给测试工具在运行时来处理这个问题 。

以下是 `Test` 注解在实践中的应用。 它被称为标记注解，因为它没有参数，只是“标记”注解元素。 如果程序员错拼 `Test` 或将 `Test` 注解应用于程序元素而不是方法声明，则该程序将无法编译。

```
// Program containing marker annotations
public class Sample {

    @Test
    public static void m1() { }  // Test should pass

    public static void m2() { }
```

```
    @Test
    public static void m3() {     // Test should fail
        throw new RuntimeException("Boom");
    }

    public static void m4() { }

    @Test
    public void m5() { } // INVALID USE: nonstatic method

    public static void m6() { }

    @Test
    public static void m7() {    // Test should fail
        throw new RuntimeException("Crash");
    }

    public static void m8() { }
}
```

`Sample` 类有七个静态方法，其中四个被标注为 `Test` 。 其中两个， `m3` 和 `m7` 引发异常，两个 `m1` 和 `m5` 不引发异常。 但是没有引发异常的注解方法之一是实例方法，因此它不是注释的有效用法。 总之， `Sample` 包含四个测试：一个会通过，两个会失败，一个是无效的。 未使用 `Test` 注解标注的四种方法将被测试工具忽略。

`Test` 注解对 `Sample` 类的语义没有直接影响。 他们只提供信息供相关程序使用。 更一般地说，注解不会改变注解代码的语义，但可以通过诸如这个简单的测试运行器等工具对其进行特殊处理：

```
// Program to process marker annotations
import java.lang.reflect.*;

public class RunTests {
    public static void main(String[] args) throws Exception {
        int tests = 0;
        int passed = 0;
        Class<?> testClass = Class.forName(args[0]);
        for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                tests++;
                try {
                    m.invoke(null);
                    passed++;
                } catch (InvocationTargetException wrappedExc) {
                    Throwable exc = wrappedExc.getCause();
                    System.out.println(m + " failed: " + exc);
                } catch (Exception exc) {
                    System.out.println("Invalid @Test: " + m);
```

```
                }
            }
        }
        System.out.printf("Passed: %d, Failed: %d%n",
                          passed, tests - passed);
    }
}
```

测试运行器工具在命令行上接受完全限定的类名，并通过调用 `Method.invoke` 来反射地运行所有类标记有 `Test` 注解的方法。 `isAnnotationPresent` 方法告诉工具要运行哪些方法。 如果测试方法引发异常，则反射机制将其封装在 `InvocationTargetException` 中。 该工具捕获此异常并打印包含由 `test` 方法抛出的原始异常的故障报告，该方法是使用 `getCause` 方法从 `InvocationTargetException` 中提取的。

如果尝试通过反射调用测试方法会抛出除 `InvocationTargetException` 之外的任何异常，则表示编译时未捕获到没有使用的 `Test` 注解。 这些用法包括注解实例方法，具有一个或多个参数的方法或不可访问的方法。 测试运行器中的第二个 `catch` 块会捕获这些 `Test` 使用错误并显示相应的错误消息。 这是在 `RunTests` 在 `Sample` 上运行时打印的输出：

```
public static void Sample.m3() failed: RuntimeException: Boom
Invalid @Test: public void Sample.m5()
public static void Sample.m7() failed: RuntimeException: Crash
Passed: 1, Failed: 3
```

现在，让我们添加对仅在抛出特定异常时才成功的测试的支持。 我们需要为此添加一个新的注解类型：

```
// Annotation type with a parameter
import java.lang.annotation.*;

/**
 * Indicates that the annotated method is a test method that
 * must throw the designated exception to succeed.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Throwable> value();
}
```

此注解的参数类型是 `Class<? extends Throwable>` 。毫无疑问，这种通配符是拗口的。 在英文中，它表示“扩展 `Throwable` 的某个类的 `Class` 对象”，它允许注解的用户指定任何异常（或错误）类型。 这个用法是一个限定类型标记的例子（详见第 33 条）。 以下是注解在实践中的例子。 请注意，类名字被用作注解参数值：

```
// Program containing annotations with a parameter
```

```
public class Sample2 {

    @ExceptionTest(ArithmeticException.class)
    public static void m1() {  // Test should pass
        int i = 0;
        i = i / i;
    }

    @ExceptionTest(ArithmeticException.class)
    public static void m2() {  // Should fail (wrong exception)
        int[] a = new int[0];
        int i = a[1];
    }

    @ExceptionTest(ArithmeticException.class)
    public static void m3() { }  // Should fail (no exception)
}
```

现在让我们修改测试运行器工具来处理新的注解。 这样将包括将以下代码添加到 `main` 方法中：

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s failed: no exception%n", m);
    } catch (InvocationTargetException wrappedEx) {
        Throwable exc = wrappedEx.getCause();
        Class<? extends Throwable> excType =
            m.getAnnotation(ExceptionTest.class).value();
        if (excType.isInstance(exc)) {
            passed++;
        } else {
            System.out.printf(
                "Test %s failed: expected %s, got %s%n",
                m, excType.getName(), exc);
        }
    } catch (Exception exc) {
        System.out.println("Invalid @Test: " + m);
    }
}
```

此代码与我们用于处理 `Test` 注解的代码类似，只有一个例外：此代码提取注解参数的值并使用它来检查测试引发的异常是否属于正确的类型。 没有明确的转换，因此没有 `ClassCastException` 的危险。 测试程序编译的事实保证其注解参数代表有效的异常类型，但有一点需要注意：如果注解参数在编译时有效，但代表指定异常类型的类文件在运行时不再存在，则测试运行器将抛出 `TypeNotPresentException` 异常。

将我们的异常测试示例进一步推进，可以设想一个测试，如果它抛出几个指定的异常中的任何一个，就会通过测试。 注解机制有一个便于支持这种用法的工具。 假设我们将 `ExceptionTest` 注解的参数类型更改为 `Class` 对象数组：

```
// Annotation type with an array parameter
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception>[] value();
}
```

注解中数组参数的语法很灵活。 它针对单元素数组进行了优化。 所有以前的 `ExceptionTest` 注解仍然适用于 `ExceptionTest` 的新数组参数版本，并且会生成单元素数组。 要指定一个多元素数组，请使用花括号将这些元素括起来，并用逗号分隔它们：

```
// Code containing an annotation with an array parameter
@ExceptionTest({ IndexOutOfBoundsException.class,
                 NullPointerException.class })
public static void doublyBad() {
    List<String> list = new ArrayList<>();
    // The spec permits this method to throw either
    // IndexOutOfBoundsException or NullPointerException
    list.addAll(5, null);
}
```

修改测试运行器工具以处理新版本的 `ExceptionTest` 是相当简单的。 此代码替换原始版本：

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s failed: no exception%n", m);
    } catch (Throwable wrappedExc) {
        Throwable exc = wrappedExc.getCause();
        int oldPassed = passed;
        Class<? extends Exception>[] excTypes =
            m.getAnnotation(ExceptionTest.class).value();
        for (Class<? extends Exception> excType : excTypes) {
            if (excType.isInstance(exc)) {
                passed++;
                break;
            }
        }
        if (passed == oldPassed)
            System.out.printf("Test %s failed: %s %n", m, exc);
    }
}
```

从 Java 8 开始，还有另一种方法来执行多值注解。 可以使用 `@Repeatable` 元注解来标示注解的声明，而不用使用数组参数声明注解类型，以指示注解可以重复应用于单个元素。 该元注解采用单个参数，该参数是包含注解类型的类对象，其唯一参数是注解类型[JLS，9.6.3] 的数组。 如果我们使用 `ExceptionTest` 注解采用这种方法，下面是注解的声明。 请注意，包含注解类型必须使用适当的保留策略和目标进行注解，否则声明将无法编译：

```
// Repeatable annotation type
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(ExceptionTestContainer.class)
public @interface ExceptionTest {
    Class<? extends Exception> value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTestContainer {
    ExceptionTest[] value();
}
```

下面是我们的 `doublyBad` 测试用一个重复的注解代替基于数组值注解的方式：

```
// Code containing a repeated annotation
@ExceptionTest(IndexOutOfBoundsException.class)
@ExceptionTest(NullPointerException.class)
public static void doublyBad() { ... }
```

处理可重复的注解需要注意。重复注解会生成包含注解类型的合成注解。 `getAnnotationsByType` 方法掩盖了这一事实，可用于访问可重复注解类型和非重复注解。但 `isAnnotationPresent` 明确指出重复注解不是注解类型，而是包含注解类型。如果某个元素具有某种类型的重复注解，并且使用 `isAnnotationPresent` 方法检查元素是否具有该类型的注释，则会发现它没有。使用此方法检查注解类型的存在会因此导致程序默默忽略重复的注解。同样，使用此方法检查包含的注解类型将导致程序默默忽略不重复的注释。要使用 `isAnnotationPresent` 检测重复和非重复的注解，需要检查注解类型及其包含的注解类型。以下是 `RunTests` 程序的相关部分在修改为使用 `ExceptionTest` 注解的可重复版本时的例子：

```
// Processing repeatable annotations
if (m.isAnnotationPresent(ExceptionTest.class)
    || m.isAnnotationPresent(ExceptionTestContainer.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s failed: no exception%n", m);
    } catch (Throwable wrappedExc) {
        Throwable exc = wrappedExc.getCause();
```

```
            int oldPassed = passed;
            ExceptionTest[] excTests =
                    m.getAnnotationsByType(ExceptionTest.class);
            for (ExceptionTest excTest : excTests) {
                if (excTest.value().isInstance(exc)) {
                    passed++;
                    break;
                }
            }
            if (passed == oldPassed)
                System.out.printf("Test %s failed: %s %n", m, exc);
        }
    }
```

添加了可重复的注解以提高源代码的可读性，从逻辑上将相同注解类型的多个实例应用于给定程序元素。 如果觉得它们增强了源代码的可读性，请使用它们，但请记住，在声明和处理可重复注解时存在更多的样板，并且处理可重复的注解很容易出错。

这个项目中的测试框架只是一个演示，但它清楚地表明了注解相对于命名模式的优越性，而且它仅仅描绘了你可以用它们做什么的外观。 如果编写的工具要求程序员将信息添加到源代码中，请定义适当的注解类型。 **当可以使用注解代替时，没有理由使用命名模式。**

这就是说，除了特定的开发者（toolsmith）之外，大多数程序员都不需要定义注解类型。 **但所有程序员都应该使用 Java 提供的预定义注解类型**（详见第 27 和 40 条）。 另外，请考虑使用 IDE 或静态分析工具提供的注解。 这些注解可以提高这些工具提供的诊断信息的质量。 但请注意，这些注解尚未标准化，因此如果切换工具或标准出现，可能额外需要做一些工作。

40. 始终使用 Override 注解

Java 类库包含几个注解类型。对于典型的程序员来说，最重要的是 `@Override` 。此注解只能在方法声明上使用，它表明带此注解的方法声明重写了父类的声明。如果始终使用这个注解，它将避免产生大量的恶意 bug。考虑这个程序，在这个程序中，类 `Bigram` 表示双字母组合，或者是有序的一对字母：

```
// Can you spot the bug?
public class Bigram {
    private final char first;
    private final char second;

    public Bigram(char first, char second) {
        this.first  = first;
        this.second = second;
    }

    public boolean equals(Bigram b) {
        return b.first == first && b.second == second;
```

```
    }

    public int hashCode() {
        return 31 * first + second;
    }

    public static void main(String[] args) {
        Set<Bigram> s = new HashSet<>();
        for (int i = 0; i < 10; i++)
            for (char ch = 'a'; ch <= 'z'; ch++)
                s.add(new Bigram(ch, ch));
        System.out.println(s.size());
    }
}
```

主程序重复添加二十六个双字母组合到集合中，每个双字母组合由两个相同的小写字母组成。 然后它会打印集合的大小。 你可能希望程序打印 26，因为集合不能包含重复项。 如果你尝试运行程序，你会发现它打印的不是 26，而是 260。它有什么问题?

显然，`Bigram` 类的作者打算重写 `equals` 方法（详见第 10 条），甚至记得重写 `hashCode` （详见第 11 条）。 不幸的是，我们倒霉的程序员没有重写 `equals`，而是重载它（详见第 52 条）。 要重写 `Object.equals`，必须定义一个 `equals` 方法，其参数的类型为 `Object`，但 `Bigram` 的 `equals` 方法的参数不是 `Object` 类型的，因此 `Bigram` 继承 `Object` 的 `equals` 方法，这个 `equals` 方法测试对象的引用是否是同一个，就像 `==` 运算符一样。 每个字母组合的 10 个副本中的每一个都与其他 9 个副本不同，所以它们被 `Object.equals` 视为不相等，这就解释了程序打印 260 的原因。

幸运的是，编译器可以帮助你找到这个错误，但只有当你通过告诉它你打算重写 `Object.equals` 来帮助你。 要做到这一点，用 `@Override` 注解 `Bigram.equals` 方法，如下所示：

```
@Override public boolean equals(Bigram b) {
    return b.first == first && b.second == second;
}
```

如果插入此注解并尝试重新编译该程序，编译器将生成如下错误消息：

```
Bigram.java:10: method does not override or implement a method
from a supertype
    @Override public boolean equals(Bigram b) {
    ^
```

你会立刻意识到你做错了什么，在额头上狠狠地打了一下，用一个正确的（详见第 10 条）来替换出错的 `equals` 实现：

```
@Override public boolean equals(Object o) {
    if (!(o instanceof Bigram))
        return false;
    Bigram b = (Bigram) o;
    return b.first == first && b.second == second;
}
```

因此，应该在你认为要重写父类声明的每个方法声明上使用 `Override` 注解。 这条规则有一个小例外。 如果正在编写一个没有标记为抽象的类，并且确信它重写了其父类中的抽象方法，则无需将 `Override` 注解放在该方法上。 在没有声明为抽象的类中，如果无法重写抽象父类方法，编译器将发出错误消息。 但是，你可能希望关注类中所有重写父类方法的方法，在这种情况下，也应该随时注解这些方法。 大多数 IDE 可以设置为在选择重写方法时自动插入 `Override` 注解。

大多数 IDE 提供了是种使用 `Override` 注解的另一个理由。 如果启用适当的检查功能，如果有一个方法没有 `Override` 注解但是重写父类方法，则 IDE 将生成一个警告。 如果始终使用 `Override` 注解，这些警告将提醒你无意识的重写。 它们补充了编译器的错误消息，这些消息会提醒你无意识重写失败。 IDE 和编译器，可以确保你在任何你想要的地方和其他地方重写方法，万无一失。

`Override` 注解可用于重写来自接口和类的方法声明。 随着 `default` 默认方法的出现，在接口方法的具体实现上使用 `Override` 以确保签名是正确的是一个好习惯。 如果知道某个接口没有默认方法，可以选择忽略接口方法的具体实现上的 `Override` 注解以减少混乱。

然而，在一个抽象类或接口中，值得标记的是你认为重写父类或父接口方法的所有方法，无论是具体的还是抽象的。 例如， `Set` 接口不会向 `Collection` 接口添加新方法，因此它应该在其所有方法声明中包含 `Override` 注解以确保它不会意外地向 `Collection` 接口添加任何新方法。

总之，如果在每个方法声明中使用 `Override` 注解，并且认为要重写父类声明，那么编译器可以保护免受很多错误的影响，但有一个例外。 在具体的类中，不需要注解标记你确信可以重写抽象方法声明的方法（尽管这样做也没有坏处）。

41. 使用标记接口定义类型

标记接口（marker interface），是不包含方法声明的接口，只是指定（或「标记」）一个类实现了具有某些属性的接口。 例如，考虑 `Serializable` 接口（第 12 章）。通过实现这个接口，一个类表明它的实例可以写入 `ObjectOutputStream` （或被「序列化」）。

你可能会听说过标记注解（详见第 39 条）使得标记接口过时了。 这个断言是不正确的。 标记接口与标记注解相比具有两个优点。 首先，也是最重要的一点，**标记接口定义了一个由标记类实例实现的类型；标记注解则没有定义这样的类型。** 标记接口类型的存在允许在编译时捕获错误，如果使用标记注解，则直到运行时才能捕获错误。

Java 的序列化机制（第 6 章）使用 `Serializable` 标记接口来指示某个类型是可序列化的。 对传递给它的对象进行序列化的 `ObjectOutputStream.writeObject` 方法要求其参数可序列化。 如果此方法的参数是 `Serializable` 类型，则在编译时会检测到序列化不适当对象的尝试（通过类型检查）。 编译时错误检测是标记接口的意图，但不幸的是， `ObjectOutputStream.writeObject` API 没有利用

`Serializable` 接口：它的参数被声明为 `Object` 类型，所以尝试序列化一个不可序列化的对象直到运行时才会失败。

标记接口对于标记注解的另一个优点是可以更精确地定位目标。 如果使用目标 `ElementType.TYPE` 声明注解类型，它就可以被应用于任何类或接口。 假设有一个标记仅适用于特定接口的实现。 如果将其定义为标记接口，则可以扩展它适用的唯一接口，保证所有标记类型也是适用的唯一接口的子类型。

可以说， `Set` 接口就是这样一个受限的标记接口。 它仅适用于 `Collection` 子类型，但不会添加超出 `Collection` 定义的方法。 它通常不被认为是标记接口，因为它改进了几个 `Collection` 方法的契约，包括 `add` ， `equals` 和 `hashCode` 。 但很容易想象一个标记接口，它仅适用于某些特定接口的子类型，并且不会改进任何接口方法的契约。 这样的标记接口可以描述整个对象的一些约束条件（invariant），或者说明实例有资格被某个其他类的方法处理（就像 `Serializable` 接口指示实例有资格被 `ObjectOutputStream` 处理的方式）。

标记注解优于标记接口的主要优点是它们是更大的注解工具的一部分。因此，标记注解允许在基于注解的框架中保持一致性。

所以什么时候应该使用标记注解，什么时候应该使用标记接口？显然，如果标记是应用于除类或接口以外的任何程序元素，则必须使用注解，因为只能使用类和接口来实现或扩展接口。如果标记仅适用于类和接口，那么问自己问题：「可能我想编写一个或多个只接受具有此标记的对象的方法呢？」如果是这样，则应该优先使用标记接口而不是注解。这将使你可以将接口用作所讨论方法的参数类型，这将带来编译时类型检查的好处。如果你能说服自己，永远不会想写一个只接受带有标记的对象的方法，那么最好使用标记注解。另外，如果标记是大量使用注解的框架的一部分，则标记注解是明确的选择。

总之，标记接口和标记注释都有其用处。 如果你想定义一个没有任何关联的新方法的类型，一个标记接口是一种可行的方法。 如果要标记除类和接口以外的程序元素，或者将标记符合到已经大量使用注解类型的框架中，那么标记注解是正确的选择。 **如果发现自己正在编写目标为 `ElementType.TYPE` 的标记注解类型，那么请花时间弄清楚究竟应该用注解类型，还是标记接口更合适。**

从某种意义来说，本条目与条目 22 的的意思正好相反，条目 22 的意思是：「如果你不想定义一个类型，不要使用接口」。本条目的意思是：「如果想定义一个类型，一定要使用接口。」

42. lambda 表达式优于匿名类

在 Java 8 中，添加了函数式接口， `lambda` 表达式和方法引用，以便更容易地创建函数对象。 Stream API 随着其他语言的修改一同被添加进来，为处理数据元素序列提供类库支持。 在本章中，我们将讨论如何充分利用这些功能。

以往，使用单一抽象方法的接口（或者很少使用的抽象类）被用作函数类型。 它们的实例（称为函数对象）表示函数（functions）或行动（actions）。 自从 JDK 1.1 于 1997 年发布以来，创建函数对象的主要手段就是匿名类（详见第 24 条）。 下面是一段代码片段，按照字符串长度顺序对列表进行排序，使用匿名类创建排序的比较方法（强制排序顺序）：

```
// Anonymous class instance as a function object - obsolete!
Collections.sort(words, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
});
```

匿名类适用于需要函数对象的经典面向对象设计模式，特别是策略模式[Gamma95]。 比较器接口表示排序的抽象策略; 上面的匿名类是排序字符串的具体策略。 然而，匿名类的冗长，使得 Java 中的函数式编程成为一种不吸引人的设想。

在 Java 8 中，语言形式化了这样的概念，即使用单个抽象方法的接口是特别的，应该得到特别的对待。 这些接口现在称为函数式接口，并且该语言允许你使用 **lambda** 表达式或简称 **lambdas** 来创建这些接口的实例。 **Lambdas** 在功能上与匿名类相似，但更为简洁。 下面的代码使用 **lambdas** 替换上面的匿名类。 样板不见了，行为清晰明了：

```
// Lambda expression as function object (replaces anonymous class)
Collections.sort(words,
        (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

请注意，代码中不存在 **lambda**（`Comparator <String>`），其参数（s1 和 s2，都是 `String` 类型）及其返回值（int）的类型。 编译器使用称为类型推断的过程从上下文中推导出这些类型。 在某些情况下，编译器将无法确定类型，必须指定它们。 类型推断的规则很复杂：他们在 JLS 中占据了整个章节[JLS，18]。 很少有程序员详细了解这些规则，但没关系。 除非它们的存在使你的程序更清晰，否则省略所有 **lambda** 参数的类型。 如果编译器生成一个错误，告诉你它不能推断出 **lambda** 参数的类型，那么指定它。 有时你可能不得不强制转换返回值或整个 **lambda** 表达式，但这很少见。

关于类型推断需要注意一点。 条目 26 告诉你不要使用原始类型，条目 29 告诉你偏好泛型类型，条目 30 告诉你偏向泛型方法。 当使用 **lambda** 表达式时，这个建议是非常重要的，因为编译器获得了大部分允许它从泛型进行类型推断的类型信息。 如果你没有提供这些信息，编译器将无法进行类型推断，你必须在 **lambdas** 中手动指定类型，这将大大增加它们的冗余度。 举例来说，如果变量被声明为原始类型 `List` 而不是参数化类型 `List<String>`，则上面的代码片段将不会编译。

顺便提一句，如果使用比较器构造方法代替 **lambda**，则代码中的比较器可以变得更加简洁（详见第 14 和 43 条）：

```
Collections.sort(words, comparingInt(String::length));
```

实际上，通过利用添加到 Java 8 中的 `List` 接口的 `sort` 方法，可以使片段变得更简短：

```
words.sort(comparingInt(String::length));
```

将 **lambdas** 添加到该语言中，使得使用函数对象在以前没有意义的地方非常实用。例如，考虑条目 34 中的 `Operation` 枚举类型。由于每个枚举都需要不同的应用程序行为，所以我们使用了特定于常量的类主体，并在每个枚举常量中重写了 `apply` 方法。为了刷新你的记忆，下面是之前的代码：

```
// Enum type with constant-specific class bodies & data
public enum Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },

    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },

    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },

    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };

    private final String symbol;

    Operation(String symbol) { this.symbol = symbol; }

    @Override
    public String toString() { return symbol; }

    public abstract double apply(double x, double y);
}
```

第 34 条目说，枚举实例属性比常量特定的类主体更可取。 Lambdas 可以很容易地使用前者而不是后者来实现常量特定的行为。 仅仅将实现每个枚举常量行为的 lambda 传递给它的构造方法。 构造方法将 lambda 存储在实例属性中， `apply` 方法将调用转发给 lambda。 由此产生的代码比原始版本更简单，更清晰：

```
public enum Operation {
    PLUS  ("+", (x, y) -> x + y),
    MINUS ("-", (x, y) -> x - y),
    TIMES ("*", (x, y) -> x * y),
    DIVIDE("/", (x, y) -> x / y);

    private final String symbol;
    private final DoubleBinaryOperator op;

    Operation(String symbol, DoubleBinaryOperator op) {
        this.symbol = symbol;
        this.op = op;
    }

    @Override
```

```
        public String toString() { return symbol; }

        public double apply(double x, double y) {
            return op.applyAsDouble(x, y);
        }
    }
```

请注意，我们使用表示枚举常量行为的 lambdas 的 `DoubleBinaryOperator` 接口。 这是 `java.util.function` 中许多预定义的函数接口之一（详见第 44 条）。 它表示一个函数，它接受两个 double 类型参数并返回 double 类型的结果。

看看基于 lambda 的 `Operation` 枚举，你可能会认为常量特定的方法体已经失去了它们的用处，但事实并非如此。 与方法和类不同，lambda **没有名称和文档；如果计算不是自解释的，或者超过几行，则不要将其放入** lambda **表达式中。** 一行代码对于 lambda 说是理想的，三行代码是合理的最大值。 如果违反这一规定，可能会严重损害程序的可读性。 如果一个 lambda 很长或很难阅读，要么找到一种方法来简化它或重构你的程序来消除它。 此外，传递给枚举构造方法的参数在静态上下文中进行评估。 因此，枚举构造方法中的 lambda 表达式不能访问枚举的实例成员。 如果枚举类型具有难以理解的常量特定行为，无法在几行内实现，或者需要访问实例属性或方法，那么常量特定的类主体仍然是行之有效的方法。

同样，你可能会认为匿名类在 lambda 时代已经过时了。 这更接近事实，但有些事情你可以用匿名类来做，而却不能用 lambdas 做。 Lambda 仅限于函数式接口。 如果你想创建一个抽象类的实例，你可以使用匿名类来实现，但不能使用 lambda。 同样，你可以使用匿名类来创建具有多个抽象方法的接口实例。 最后，lambda 不能获得对自身的引用。 在 lambda 中，`this` 关键字引用封闭实例，这通常是你想要的。 在匿名类中，`this` 关键字引用匿名类实例。 如果你需要从其内部访问函数对象，则必须使用匿名类。

Lambdas 与匿名类共享无法可靠地序列化和反序列化实现的属性。**因此，应该很少 (如果有的话) 序列化一个** lambda(**或一个匿名类实例**)。 如果有一个想要进行序列化的函数对象，比如一个 `Comparator`，那么使用一个私有静态嵌套类的实例（详见第 24 条）。

综上所述，从 Java 8 开始，lambda 是迄今为止表示小函数对象的最佳方式。 **除非必须创建非函数式接口类型的实例，否则不要使用匿名类作为函数对象。** 另外，请记住，lambda 表达式使代表小函数对象变得如此简单，以至于它为功能性编程技术打开了一扇门，这些技术在 Java 中以前并不实用。

43. 方法引用优于 lambda 表达式

lambda 优于匿名类的主要优点是它更简洁。Java 提供了一种生成函数对象的方法，比 lambda 还要简洁，那就是：方法引用（method references）。下面是一段程序代码片段，它维护一个从任意键到整数值的映射。如果将该值解释为键的实例个数，则该程序是一个多重集合的实现。该代码的功能是，根据键找到整数值，然后在此基础上加 1：

```
map.merge(key, 1, (count, incr) -> count + incr);
```

请注意，此代码使用 `merge` 方法，该方法已添加到 Java 8 中的 `Map` 接口中。如果没有给定键的映射，则该方法只是插入给定值; 如果映射已经存在，则合并给定函数应用于当前值和给定值，并用结果覆盖当前值。 此代码表示 `merge` 方法的典型用例。

代码很好读，但仍然有一些样板的味道。 参数 `count` 和 `incr` 不会增加太多价值，并且占用相当大的空间。 真的，所有的 lambda 都告诉你函数返回两个参数的和。 从 Java 8 开始， `Integer` 类（和所有其他包装数字基本类型）提供了一个静态方法总和，和它完全相同。 我们可以简单地传递一个对这个方法的引用，并以较少的视觉混乱得到相同的结果：

```
map.merge(key, 1, Integer::sum);
```

方法的参数越多，你可以通过方法引用消除更多的样板。 然而，在一些 lambda 中，选择的参数名称提供了有用的文档，使得 lambda 比方法引用更具可读性和可维护性，即使 lambda 看起来更长。

只要方法引用能做的事情，就没有 lambda 不能完成的（只有一种情况例外 - 如果你好奇的话，参见 JLS，9.9-2）。 也就是说，使用方法引用通常会得到更短，更清晰的代码。 如果 lambda 变得太长或太复杂，它们也会给你一个结果：你可以从 lambda 中提取代码到一个新的方法中，并用对该方法的引用代替 lambda。 你可以给这个方法一个好名字，并把它文档记录下来。

如果你使用 IDE 编程，它将提供替换 lambda 的方法，并在任何地方使用方法引用。通常情况下，你应该接受这个提议。偶尔，lambda 会比方法引用更简洁。这种情况经常发生在方法与 lambda 相同的类中。例如，考虑这段代码，它被假定出现在一个名为 `GoshThisClassNameIsHumongous` 的类中：

```
service.execute(GoshThisClassNameIsHumongous::action);
```

这个 lambda 类似于等价于下面的代码：

```
service.execute(() -> action());
```

使用方法引用的代码段并不比使用 lambda 的代码片段更短也不清晰，所以优先选择后者。 在类似的代码行中， `Function` 接口提供了一个通用的静态工厂方法来返回标识函数 `Function.identity()` 。 不使用这种方法，而是使用等效的 lambda 内联代码： `x -> x` ，通常更短，更简洁。

许多方法引用是指静态方法，但有 4 种方法没有引用静态方法。 其中两个 Lambda 等式是特定（bound）和任意（unbound）对象方法引用。 在特定对象引用中，接收对象在方法引用中指定。 特定对象引用在本质上与静态引用类似：函数对象与引用的方法具有相同的参数。 在任意对象引用中，接收对象在应用函数对象时通过方法的声明参数之前的附加参数指定。 任意对象引用通常用作流管道（pipelines）中的映射和过滤方法（条目 45）。 最后，对于类和数组，有两种构造方法引用。 构造方法引用用作工厂对象。 下表总结了所有五种方法引用：

方法引用类型 Method Ref Type	举例 Example	Lambda 等式 Lambda Equivalent
Static	Integer::parseInt	str -> Integer.parseInt(str)
Bound	Instant.now()::isAfter	Instant then = Instant.now(); t -> then.isAfter(t)

方法引用类型 Method Ref Type	举例 Example	Lambda 等式 Lambda Equivalent
Unbound	String::toLowerCase	str -> str.toLowerCase()
Class Constructor	TreeMap<K, V>::new	() -> new TreeMap<K, V>
Array Constructor	int[]::new	len -> new int[len]

总之，方法引用通常为 lambda 提供一个更简洁的选择。 **如果方法引用看起来更简短更清晰，请使用它们；否则，还是坚持 lambda。**

44. 优先使用标准的函数式接口

现在 Java 已经有 lambda 表达式，编写 API 的最佳实践已经发生了很大的变化。 例如，模板方法模式[Gamma95]，其中一个子类重写原始方法以专门化其父类的行为，变得没有那么吸引人。 现代替代的选择是提供一个静态工厂或构造方法来接受函数对象以达到相同的效果。 通常地说，可以编写更多以函数对象为参数的构造方法和方法。 选择正确的函数式参数类型需要注意。

考虑 `LinkedHashMap` 。 可以通过重写其受保护的 `removeEldestEntry` 方法将此类用作缓存，每次将新的 key 值加入到 map 时都会调用该方法。 当此方法返回 true 时，map 将删除传递给该方法的最久条目。 以下代码重写允许 map 增长到一百个条目，然后在每次添加新 key 值时删除最老的条目，并保留最近的一百个条目：

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
   return size() > 100;
}
```

这种技术很有效，但是你可以用 lambdas 做得更好。如果 `LinkedHashMap` 是现在编写的，那么它将有一个静态的工厂或构造方法来获取函数对象。查看 `removeEldestEntry` 方法的声明，你可能会认为函数对象应该接受一个 `Map.Entry<K, V>` 并返回一个布尔值，但是这并不完全是这样：`removeEldestEntry` 方法调用 `size()` 方法来获取条目的数量，因为 `removeEldestEntry` 是 map 上的一个实例方法。传递给构造方法的函数对象不是 map 上的实例方法，无法捕获，因为在调用其工厂或构造方法时 map 还不存在。因此，map 必须将自己传递给函数对象，函数对象把 map 以及最就的条目作为输入参数。如果要声明这样一个功能接口，应该是这样的：

```
// Unnecessary functional interface; use a standard one instead.
@FunctionalInterface
interface EldestEntryRemovalFunction<K,V>{
    boolean remove(Map<K,V> map, Map.Entry<K,V> eldest);
}
```

这个接口可以正常工作，但是你不应该使用它，因为你不需要为此目的声明一个新的接口。 `java.util.function` 包提供了大量标准函数式接口供你使用。 如果其中一个标准函数式接口完成这项工作，则通常应该优先使用它，而不是专门构建的函数式接口。 这将使你的 API 更容易学习，通过减少其不必要概念，并将提供重要的互操作性好处，因为许多标准函数式接口提供了有用的默认方法。 例

如，`Predicate` 接口提供了组合判断的方法。 在我们的 `LinkedHashMap` 示例中，标准的 `BiPredicate<Map<K,V>, Map.Entry<K,V>>` 接口应优先于自定义的 `EldestEntryRemovalFunction` 接口的使用。

在 `java.util.Function` 中有 43 个接口。不能指望全部记住它们，但是如果记住了六个基本接口，就可以在需要它们时派生出其余的接口。基本接口操作于对象引用类型。 `Operator` 接口表示方法的结果和参数类型相同。 `Predicate` 接口表示其方法接受一个参数并返回一个布尔值。 `Function` 接口表示方法其参数和返回类型不同。 `Supplier` 接口表示一个不接受参数和返回值 (或「供应」) 的方法。最后， `Consumer` 表示该方法接受一个参数而不返回任何东西，本质上就是使用它的参数。六种基本函数式接口概述如下：

接口	方法	示例
`UnaryOperator<T>`	T apply(T t)	String::toLowerCase
`BinaryOperator<T>`	T apply(T t1, T t2)	BigInteger::add
`Predicate<T>`	boolean test(T t)	Collection::isEmpty
`Function<T,R>`	R apply(T t)	Arrays::asList
`Supplier<T>`	T get()	Instant::now
`Consumer<T>`	void accept(T t)	System.out::println

在处理基本类型 int，long 和 double 的操作上，六个基本接口中还有三个变体。 它们的名字是通过在基本接口前加一个基本类型而得到的。 因此，例如，一个接受 int 的 `Predicate` 是一个 `IntPredicate` ，而一个接受两个 long 值并返回一个 long 的二元运算符是一个 `LongBinaryOperator` 。 除 `Function` 接口变体通过返回类型进行了参数化，其他变体类型都没有参数化。 例如， `LongFunction<int[]>` 使用 long 类型作为参数并返回了 int[] 类型。

`Function` 接口还有九个额外的变体，当结果类型为基本类型时使用。 源和结果类型总是不同，因为从类型到它自身的函数是 `UnaryOperator` 。 如果源类型和结果类型都是基本类型，则使用带有 `SrcToResult` 的前缀 `Function` ，例如 `LongToIntFunction` (六个变体)。如果源是一个基本类型，返回结果是一个对象引用，那么带有 `ToObj` 的前缀 `Function` ，例如 `DoubleToObjFunction` (三种变体)。

有三个包含两个参数版本的基本功能接口，使它们有意义： `BiPredicate <T, U>` ， `BiFunction <T, U, R>` 和 `BiConsumer <T, U>` 。 也有返回三种相关基本类型的 `BiFunction` 变体： `ToIntBiFunction <T, U>` ， `ToLongBiFunction<T, U>` 和 `ToDoubleBiFunction <T, U>` 。 `Consumer` 有两个变量，它们带有一个对象引用和一个基本类型： `ObjDoubleConsumer <T>` ， `ObjIntConsumer <T>` 和 `ObjLongConsumer <T>` 。 总共有九个两个参数版本的基本接口。

最后，还有一个 `BooleanSupplier` 接口，它是 `Supplier` 的一个变体，它返回布尔值。 这是任何标准函数式接口名称中唯一明确提及的布尔类型，但布尔返回值通过 `Predicate` 及其四种变体形式支持。 前面段落中介绍的 `BooleanSupplier` 接口和 42 个接口占所有四十三个标准功能接口。 无可否认，这是非常难以接受的，并且不是非常正交的。 另一方面，你所需要的大部分功能接口都是为你写的，而且它们的名字是经常性的，所以在你需要的时候不应该有太多的麻烦。

大多数标准函数式接口仅用于提供对基本类型的支持。 **不要试图使用基本的函数式接口来装箱基本类型的包装类而不是基本类型的函数式接口。** 虽然它起作用，但它违反了第 61 条中的建议：「优先使用基本类型而不是基本类型的包装类」。使用装箱基本类型的包装类进行批量操作的性能后果可能是致命的。

现在你知道你应该通常使用标准的函数式接口来优先编写自己的接口。 但是，你应该什么时候写自己的接口？ 当然，如果没有一个标准模块能够满足您的需求，例如，如果需要一个带有三个参数的 `Predicate`，或者一个抛出检查异常的 `Predicate`，那么需要编写自己的代码。 但有时候你应该编写自己的函数式接口，即使与其中一个标准的函数式接口的结构相同。

考虑我们的老朋友 `Comparator <T>`，它的结构与 `ToIntBiFunction <T, T>` 接口相同。 即使将前者添加到类库时后者的接口已经存在，使用它也是错误的。 `Comparator` 值得拥有自己的接口有以下几个原因。 首先，它的名称每次在 API 中使用时都会提供优秀的文档，并且使用了很多。 其次，`Comparator` 接口对构成有效实例的构成有强大的要求，这些要求构成了它的普遍契约。 通过实现接口，就要承诺遵守契约。 第三，接口配备很多了有用的默认方法来转换和组合多个比较器。

如果需要一个函数式接口与 `Comparator` 共享以下一个或多个特性，应该认真考虑编写一个专用函数式接口，而不是使用标准函数式接口：

- 它将被广泛使用，并且可以从描述性名称中受益。
- 它拥有强大的契约。
- 它会受益于自定义的默认方法。

如果选择编写你自己的函数式接口，请记住它是一个接口，因此应非常小心地设计（详见第 21 条）。

请注意，`EldestEntryRemovalFunction` 接口（第 199 页） `标有 @FunctionalInterface` 注解。这种注解在类型类似于 `@Override`。 这是一个程序员意图的陈述，它有三个目的：它告诉读者该类和它的文档，该接口是为了实现 lambda 表达式而设计的；它使你保持可靠，因为除非只有一个抽象方法，否则接口不会编译; 它可以防止维护人员在接口发生变化时不小心地将抽象方法添加到接口中。 **始终使用 `@FunctionalInterface` 注解标注你的函数式接口。**

最后一点应该是关于在 api 中使用函数接口的问题。不要提供具有多个重载的方法，这些重载在相同的参数位置上使用不同的函数式接口，如果这样做可能会在客户端中产生歧义。这不仅仅是一个理论问题。 `ExecutorService` 的 `submit` 方法可以采用 `Callable<T>` 或 `Runnable` 接口，并且可以编写需要强制类型转换以指示正确的重载的客户端程序（详见第 52 条）。避免此问题的最简单方法是不要编写在相同的参数位置中使用不同函数式接口的重载。这是条目 52 中建议的一个特例，「明智地使用重载」。

总之，现在 Java 已经有了 lambda 表达式，因此必须考虑 lambda 表达式来设计你的 API。 在输入上接受函数式接口类型并在输出中返回它们。 一般来说，最好使用 `java.util.function.Function` 中提供的标准接口，但请注意，在相对罕见的情况下，最好编写自己的函数式接口。

45. 明智审慎地使用 Stream

在 Java 8 中添加了 Stream API，以简化串行或并行执行批量操作的任务。 该 API 提供了两个关键的抽象：流 (Stream)，表示有限或无限的数据元素序列，以及流管道 (stream pipeline)，表示对这些元素的多级计算。 Stream 中的元素可以来自任何地方。 常见的源包括集合、数组、文件、正则表达式模式匹配器、伪随机数生成器和其他流。 流中的数据元素可以是对象引用或基本类型。 支持三种基本类型：int，long 和 double。

Stream pipeline 由 Source stream（源流） 的零或多个中间操作（intermediate operations）和一个终结操作（ terminal operation）组成。每个中间操作都以某种方式转换流，例如将每个元素映射到该元素的函数，或过滤掉所有不满足某些条件的元素。中间操作都将一个流转换为另一个流，其元素类型可能与输入流相同或不同。终结操作对流执行最后一次中间操作产生的最终计算，例如将其元素存储到集合中、返回某个元素或打印其所有元素。

Stream pipeline 通常是惰性（lazily）计算求值：直到终结操作被调用后才开始计算，而为了完成终结操作而不需要的数据元素永远不会被计算出来。 这种惰性计算求值的方式，使得无限流成为可能。请注意，没有终结操作的 Stream pipine 是一个静默无操作的指令，所以不要忘记包含一个终止操作。

Stream API 流式的（fluent）：它设计允许所有组成 pipeline 的调用被链接到一个表达式中。事实上，多个管道可以链接在一起形成一个表达式。

默认情况下，流管道会按顺序（sequentially）运行。 要使管道并行执行，只需要在管道中的任何流上调用 `parallel()` 方法一样简单，但是通常不建议这么做（详见第 48 条）。

Stream API 具有足够的通用性，实际上任何计算都可以使用 Stream 执行，但是「可以」，并不意味着应该这样做。如果使用得当，流可以使程序更短更清晰；如果使用不当，它们会使程序难以阅读和维护。对于何时使用流没有硬性的规则，但是有一些启发。

考虑以下程序，该程序从字典文件中读取单词并打印其大小符合用户指定的最小值的所有变位词（anagram）组。如果两个单词由长度相通，不同顺序的相同字母组成，则它们是变位词。程序从用户指定的字典文件中读取每个单词并将单词放入 map 对象中。map 对象的键是按照字母排序的单词，因此「staple」的键是「aelpst」，「petals」的键也是「aelpst」：这两个单词就是同位词，所有的同位词共享相同的依字母顺序排列的形式（或称之为 alphagram）。map 对象的值是包含共享字母顺序形式的所有单词的列表。 处理完字典文件后，每个列表都是一个完整的同位词组。然后程序遍历 map 对象的 `values()` 的视图并打印每个大小符合阈值的列表：

```
// Prints all large anagram groups in a dictionary iteratively
public class Anagrams {
    public static void main(String[] args) throws IOException {
        File dictionary = new File(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);
        Map<String, Set<String>> groups = new HashMap<>();
        try (Scanner s = new Scanner(dictionary)) {
            while (s.hasNext()) {
                String word = s.next();
                groups.computeIfAbsent(alphabetize(word),
                        (unused) -> new TreeSet<>()).add(word);
            }
        }

        for (Set<String> group : groups.values())
```

```
            if (group.size() >= minGroupSize)
                System.out.println(group.size() + ": " + group);
        }

        private static String alphabetize(String s) {
            char[] a = s.toCharArray();
            Arrays.sort(a);
            return new String(a);
        }
    }
```

这个程序中的一个步骤值得注意。将每个单词插入到 map 中（以粗体显示）中使用了 `computeIfAbsent` 方法，该方法是在 Java 8 中添加的。这个方法在 map 中查找一个键：如果键存在，该方法只返回与其关联的值。如果没有，该方法通过将给定的函数对象应用于键来计算值，将该值与键关联，并返回计算值。 `computeIfAbsent` 方法简化了将多个值与每个键关联的 map 的实现。

现在考虑以下程序，它也能解决同样的问题，但大量过度使用了流。 请注意，整个程序（打开字典文件的代码除外）包含在单个表达式中。 在单独的表达式中打开字典文件的唯一原因是允许使用 `try-with-resources` 语句，该语句确保关闭字典文件：

```
// Overuse of streams - don't do this!
public class Anagrams {
    public static void main(String[] args) throws IOException {
        Path dictionary = Paths.get(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);
        try (Stream<String> words = Files.lines(dictionary)) {
            words.collect(
                    groupingBy(word -> word.chars().sorted()
                            .collect(StringBuilder::new,
                                    (sb, c) -> sb.append((char) c),
                                    StringBuilder::append).toString()))
                    .values().stream()
                    .filter(group -> group.size() >= minGroupSize)
                    .map(group -> group.size() + ": " + group)
                    .forEach(System.out::println);
        }
    }
}
```

如果你发现这段代码难以阅读，不要担心；你不是一个人。它更短，但是可读性也更差，尤其是对于那些不擅长使用流的程序员来说。**过度使用流使程序难于阅读和维护。**

幸运的是，有一个折中的办法。下面的程序解决了同样的问题，使用流而不过度使用它们。其结果是一个比原来更短更清晰的程序：

```
// Tasteful use of streams enhances clarity and conciseness
public class Anagrams {
```

```
    public static void main(String[] args) throws IOException {
        Path dictionary = Paths.get(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        try (Stream<String> words = Files.lines(dictionary)) {
            words.collect(groupingBy(word -> alphabetize(word)))
                    .values().stream()
                    .filter(group -> group.size() >= minGroupSize)
                    .forEach(g -> System.out.println(g.size() + ": " + g));
        }
    }
    // alphabetize method is the same as in original version
}
```

即使以前很少接触流，这个程序也不难理解。它在一个 `try-with-resources` 块中打开字典文件，获得一个由文件中的所有行组成的流。流变量命名为 words，表示流中的每个元素都是一个单词。此流上的管道没有中间操作；它的终结操作将所有单词收集到个 map 对象中，按照字母排列的形式对单词进行分组 (第 46 项)。这与之前两个版本的程序构造的 map 完全相同。然后在 map 的 `values()` 视图上打开一个新的流 `List<String>` 。当然，这个流中的元素是同位词组。对流进行过滤，以便忽略大小小于 minGroupSize 的所有组，最后由终结操作 forEach 打印剩下的同位词组。

请注意，仔细选择 lambda 参数名称。 上面程序中参数 g 应该真正命名为 group，但是生成的代码行对于本书来说太宽了。 **在没有显式类型的情况下，仔细命名 lambda 参数对于流管道的可读性至关重要。**

另请注意，单词字母化是在单独的 `alphabetize` 方法中完成的。 这通过提供操作名称并将实现细节保留在主程序之外来增强可读性。 **使用辅助方法对于流管道中的可读性比在迭代代码中更为重要，因为管道缺少显式类型信息和命名临时变量。**

字母顺序方法可以使用流重新实现，但基于流的字母顺序方法本来不太清楚，更难以正确编写，并且可能更慢。 这些缺陷是由于 Java 缺乏对原始字符流的支持（这并不意味着 Java 应该支持 char 流；这样做是不可行的）。 要演示使用流处理 char 值的危害，请考虑以下代码：

```
"Hello world!".chars().forEach(System.out::print);
```

你可能希望它打印 `Hello world!` ，但如果运行它，发现它打印 72101108108111321191111141081003 3。这是因为 `“Hello world!”.chars()` 返回的流的元素不是 char 值，而是 int 值，因此调用了 `print` 的 int 重载。无可否认，一个名为 chars 的方法返回一个 int 值流是令人困惑的。可以通过强制调用正确的重载来修复该程序：

但理想情况下，应该避免使用流来处理 char 值。 当开始使用流时，你可能会感到想要将所有循环语句转换为流方式的冲动，但请抵制这种冲动。尽管这是可能的，但可能会损害代码库的可读性和可维护性。 通常，使用流和迭代的某种组合可以最好地完成中等复杂的任务，如上面的 `Anagrams` 程序所示。 **因此，重构现有代码以使用流，并仅在有意义的情况下在新代码中使用它们。**

如本项目中的程序所示，流管道使用函数对象 (通常为 lambdas 或方法引用) 表示重复计算，而迭代代码使用代码块表示重复计算。从代码块中可以做一些从函数对象中不能做的事情:

- 从代码块中，可以读取或修改范围内的任何局部变量; 从 lambda 中，只能读取最终或有效的最终变量[JLS 4.12.4]，并且无法修改任何局部变量。
- 从代码块中，可以从封闭方法返回，中断或继续封闭循环，或抛出声明此方法的任何已检查异常; 从一个 lambda 你不能做这些事情。

如果使用这些技术最好地表达计算，那么它可能不是流的良好匹配。 相反，流可以很容易地做一些事情：

- 统一转换元素序列
- 过滤元素序列
- 使用单个操作组合元素序列 (例如添加、连接或计算最小值)
- 将元素序列累积到一个集合中，可能通过一些公共属性将它们分组
- 在元素序列中搜索满足某些条件的元素

如果使用这些技术最好地表达计算，那么使用流是这些场景很好的候选者。

对于流来说，很难做到的一件事是同时访问管道的多个阶段中的相应元素：一旦将值映射到其他值，原始值就会丢失。一种解决方案是将每个值映射到一个包含原始值和新值的 `pair` 对象，但这不是一个令人满意的解决方案，尤其是在管道的多个阶段需要一对对象时更是如此。生成的代码既混乱又冗长，破坏了流的主要用途。当它适用时，一个更好的解决方案是在需要访问早期阶段值时转换映射。

例如，让我们编写一个程序来打印前 20 个梅森素数 (Mersenne primes)。 梅森素数是一个 2p − 1 形式的数字。如果 p 是素数，相应的梅森数可能是素数; 如果是这样的话，那就是梅森素数。 作为我们管道中的初始流，我们需要所有素数。 这里有一个返回该（无限）流的方法。 我们假设使用静态导入来轻松访问 `BigInteger` 的静态成员：

```
static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

方法的名称（primes）是一个复数名词，描述了流的元素。 强烈建议所有返回流的方法使用此命名约定，因为它增强了流管道的可读性。 该方法使用静态工厂 `Stream.iterate` ，它接受两个参数：流中的第一个元素，以及从前一个元素生成流中的下一个元素的函数。 这是打印前 20 个梅森素数的程序：

```
public static void main(String[] args) {
    primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(System.out::println);
}
```

这个程序是上面的梅森描述的直接编码：它从素数开始，计算相应的梅森数，过滤掉除素数之外的所有数字（幻数 50 控制概率素性测试 the magic number 50 controls the probabilistic primality test），将得到的流限制为 20 个元素， 并打印出来。

现在假设我们想在每个梅森素数前面加上它的指数 (p)，这个值只出现在初始流中，因此在终结操作中不可访问，而终结操作将输出结果。幸运的是通过反转第一个中间操作中发生的映射，可以很容易地计算出 `Mersenne` 数的指数。 指数是二进制表示中的位数，因此该终结操作会生成所需的结果：

```
.forEach(mp -> System.out.println(mp.bitLength() + ": " + mp));
```

有很多任务不清楚是使用流还是迭代。例如，考虑初始化一副新牌的任务。假设 `Card` 是一个不可变的值类，它封装了 `Rank` 和 `Suit`，它们都是枚举类型。这个任务代表任何需要计算可以从两个集合中选择的所有元素对。数学家们称它为两个集合的笛卡尔积。下面是一个迭代实现，它有一个嵌套的 `for-each` 循环，你应该非常熟悉：

```
// Iterative Cartesian product computation
private static List<Card> newDeck() {
    List<Card> result = new ArrayList<>();

    for (Suit suit : Suit.values())
        for (Rank rank : Rank.values())
            result.add(new Card(suit, rank));

    return result;
}
```

下面是一个基于流的实现，它使用了中间操作 `flatMap` 方法。这个操作将一个流中的每个元素映射到一个流，然后将所有这些新流连接到一个流 (或展平它们)。注意，这个实现包含一个嵌套的 lambda 表达式 `（rank -> new Card(suit, rank))`：

```
// Stream-based Cartesian product computation
private static List<Card> newDeck() {
    return Stream.of(Suit.values())
        .flatMap(suit ->
            Stream.of(Rank.values())
                .map(rank -> new Card(suit, rank)))
        .collect(toList());
}
```

newDeck 的两个版本中哪一个更好？ 它归结为个人偏好和你的编程的环境。 第一个版本更简单，也许感觉更自然。 大部分 Java 程序员将能够理解和维护它，但是一些程序员会对第二个（基于流的）版本感觉更舒服。 如果对流和函数式编程有相当的精通，那么它会更简洁，也不会太难理解。 如果不确定自己喜欢哪个版本，则迭代版本可能是更安全的选择。 如果你更喜欢流的版本，并且相信其他使用该代码的程序员会与你共享你的偏好，那么应该使用它。

总之，有些任务最好使用流来完成，有些任务最好使用迭代来完成。将这两种方法结合起来，可以最好地完成许多任务。对于选择使用哪种方法进行任务，没有硬性规定，但是有一些有用的启发式方法。在许多情况下，使用哪种方法将是清楚的；在某些情况下，则不会很清楚。**如果不确定一个任务是通过流还是迭代更好地完成，那么尝试这两种方法，看看哪一种效果更好。**

46. 优先考虑流中无副作用的函数

如果你是一个刚开始使用流的新手，那么很难掌握它们。仅仅将计算表示为流管道是很困难的。当你成功时，你的程序将运行，但对你来说可能没有意识到任何好处。流不仅仅是一个 API，它是基于函数式编程的范式（paradigm）。为了获得流提供的可表达性、速度和某些情况下的并行性，你必须采用范式和 API。

流范式中最重要的部分是将计算结构化为一系列转换，其中每个阶段的结果尽可能接近前一阶段结果的纯函数（pure function）。 纯函数的结果仅取决于其输入：它不依赖于任何可变状态，也不更新任何状态。 为了实现这一点，你传递给流操作的任何函数对象（中间操作和终结操作）都应该没有副作用。

有时，可能会看到类似于此代码片段的流代码，该代码构建了文本文件中单词的频率表:

```
// Uses the streams API but not the paradigm--Don't do this!
Map<String, Long> freq = new HashMap<>();
try (Stream<String> words = new Scanner(file).tokens()) {
    words.forEach(word -> {
        freq.merge(word.toLowerCase(), 1L, Long::sum);
    });
}
```

这段代码出了什么问题？ 毕竟，它使用了流，lambdas 和方法引用，并得到正确的答案。 简而言之，它根本不是流代码; 它是伪装成流代码的迭代代码。 它没有从流 API 中获益，并且它比相应的迭代代码更长，更难读，并且更难于维护。 问题源于这样一个事实：这个代码在一个终结操作 `forEach` 中完成所有工作，使用一个改变外部状态（频率表）的 lambda。 `forEach` 操作除了表示由一个流执行的计算结果外，什么都不做，这是「代码中的臭味」，就像一个改变状态的 lambda 一样。那么这段代码应该是什么样的呢?

```
// Proper use of streams to initialize a frequency table
Map<String, Long> freq;
try (Stream<String> words = new Scanner(file).tokens()) {
    freq = words
        .collect(groupingBy(String::toLowerCase, counting()));
}
```

此代码段与前一代码相同，但正确使用了流 API。 它更短更清晰。 那么为什么有人会用其他方式写呢？ 因为它使用了他们已经熟悉的工具。 Java 程序员知道如何使用 for-each 循环，而 `forEach` 终结操作是类似的。 但 `forEach` 操作是终端操作中最不强大的操作之一，也是最不友好的流操作。 它是明确的迭代，因此不适合并行化。 `forEach` **操作应仅用于报告流计算的结果，而不是用于执行计算。** 有时，将 `forEach` 用于其他目的是有意义的，例如将流计算的结果添加到预先存在的集合中。

改进后的代码使用了收集器（collector），这是使用流必须学习的新概念。`Collectors` 的 API 令人生畏：它有 39 个方法，其中一些方法有多达 5 个类型参数。好消息是，你可以从这个 API 中获得大部分好处，而不必深入研究它的全部复杂性。对于初学者来说，可以忽略收集器接口，将收集器看作是封装缩减策略（reduction strategy）的不透明对象。在此上下文中，reduction 意味着将流的元素组合为单个对象。 收集器生成的对象通常是一个集合（它代表名称收集器）。

将流的元素收集到真正的集合中的收集器非常简单。有三个这样的收集器：`toList()`、`toSet()` 和 `toCollection(collectionFactory)`。它们分别返回集合、列表和程序员指定的集合类型。有了这些知识，我们就可以编写一个流管道从我们的频率表中提取出现频率前 10 个单词的列表。

```
// Pipeline to get a top-ten list of words from a frequency table
List<String> topTen = freq.keySet().stream()
    .sorted(comparing(freq::get).reversed())
    .limit(10)
    .collect(toList());
```

注意，我们没有对 `toList` 方法的类收集器进行限定。**静态导入收集器的所有成员是一种惯例和明智的做法，因为它使流管道更易于阅读。**

这段代码中唯一比较棘手的部分是我们把 `comparing(freq::get).reverse()` 传递给 `sort` 方法。comparing 是一种比较器构造方法（详见第 14 条），它具有一个 key 的提取方法。该函数接受一个单词，而“提取”实际上是一个表查找：绑定方法引用 `freq::get` 在 `frequency` 表中查找单词，并返回单词出现在文件中的次数。最后，我们在比较器上调用 `reverse` 方法，因此我们将单词从最频繁到最不频繁进行排序。然后，将流限制为 10 个单词并将它们收集到一个列表中就很简单了。

前面的代码片段使用 `Scanner` 的 `stream` 方法在 `scanner` 实例上获取流。这个方法是在 Java 9 中添加的。如果正在使用较早的版本，可以使用类似于条目 47 中（`streamOf(Iterable<E>)`）的适配器将实现了 `Iterator` 的 `scanner` 序转换为流。

那么收集器中的其他 36 种方法呢？它们中的大多数都是用于将流收集到 map 中的，这比将流收集到真正的集合中要复杂得多。每个流元素都与一个键和一个值相关联，多个流元素可以与同一个键相关联。

最简单的映射收集器是 toMap(keyMapper、valueMapper)，它接受两个函数，一个将流元素映射到键，另一个映射到值。在条目 34 中的 `fromString` 实现中，我们使用这个收集器从 enum 的字符串形式映射到 enum 本身:

```
// Using a toMap collector to make a map from string to enum
private static final Map<String, Operation> stringToEnum =
    Stream.of(values()).collect(
        toMap(Object::toString, e -> e));
```

如果流中的每个元素都映射到唯一键，则这种简单的 `toMap` 形式是完美的。 如果多个流元素映射到同一个键，则管道将以 `IllegalStateException` 终止。

`toMap` 更复杂的形式，以及 `groupingBy` 方法，提供了处理此类冲突 (collisions) 的各种方法。一种方法是向 `toMap` 方法提供除键和值映射器（mappers）之外的 `merge` 方法。 `merge` 方法是一个 `BinaryOperator<V>` ，其中 V是 map 的值类型。与键关联的任何附加值都使用 `merge` 方法与现有值相结合，因此，例如，如果 merge 方法是乘法，那么最终得到的结果是是值 `mapper` 与键关联的所有值的乘积。

`toMap` 的三个参数形式对于从键到与该键关联的选定元素的映射也很有用。例如，假设我们有一系列不同艺术家（artists）的唱片集（albums），我们想要一张从唱片艺术家到最畅销专辑的 map。这个收集器将完成这项工作。

```
// Collector to generate a map from key to chosen element for key
Map<Artist, Album> topHits = albums.collect(
   toMap(Album::artist, a->a, maxBy(comparing(Album::sales))));
```

请注意，比较器使用静态工厂方法 `maxBy` ，它是从 BinaryOperator 静态导入的。 此方法将 `Comparator<T>` 转换为 `BinaryOperator<T>` ，用于计算指定比较器隐含的最大值。 在这种情况下，比较器由比较器构造方法 comparing 返回，它采用 key 提取器函数 `Album::sales` 。 这可能看起来有点复杂，但代码可读性很好。 简而言之，它说，「将专辑（albums）流转换为地 map，将每位艺术家（artist）映射到销售量最佳的专辑。」这与问题陈述出奇得接近。

`toMap` 的三个参数形式的另一个用途是产生一个收集器，当发生冲突时强制执行 last-write-wins 策略。 对于许多流，结果是不确定的，但如果映射函数可能与键关联的所有值都相同，或者它们都是可接受的，则此收集器的行为可能正是您想要的：

```
// Collector to impose last-write-wins policy
toMap(keyMapper, valueMapper, (oldVal, newVal) ->newVal)
```

`toMap` 的第三个也是最后一个版本采用第四个参数，它是一个 map 工厂，用于指定特定的 map 实现，例如 `EnumMap` 或 `TreeMap` 。

`toMap` 的前三个版本也有变体形式，名为 `toConcurrentMap` ，它们并行高效运行并生成 `ConcurrentHashMap` 实例。

除了 toMap 方法之外，Collectors API 还提供了 `groupingBy` 方法，该方法返回收集器以生成基于分类器函数 （classifier function）将元素分组到类别中的 map。 分类器函数接受一个元素并返回它所属的类别。 此类别来用作元素的 map 的键。 `groupingBy` 方法的最简单版本仅采用分类器并返回一个 map，其值是每个类别中所有元素的列表。 这是我们在条目 45 中的 `Anagram` 程序中使用的收集器，用于生成从按字母顺序排列的单词到单词列表的 map：

```
Map<String, Long> freq = words
        .collect(groupingBy(String::toLowerCase, counting()));
```

`groupingBy` 的第三个版本允许指定除 `downstream` 收集器之外的 map 工厂。 请注意，这种方法违反了标准的可伸缩参数列表模式 (standard telescoping argument list pattern)： `mapFactory` 参数位于 `downStream` 参数之前，而不是之后。 此版本的 `groupingBy` 可以控制包含的 map 以及包含的集合，因此，例如，可以指定一个收集器，它返回一个 `TreeMap` ，其值是 `TreeSet` 。

`groupingByConcurrent` 方法提供了 `groupingBy` 的所有三个重载的变体。 这些变体并行高效运行并生成 `ConcurrentHashMap` 实例。 还有一个很少使用的 grouping 的亲戚称为 `partitioningBy` 。代替分类器方法，它接受 `predicate` 并返回其键为布尔值的 map。 此方法有两种重载，除了 `predicate` 之外，其中一种方法还需要 `downstream` 收集器。

通过 `counting` 方法返回的收集器仅用作下游收集器。 Stream 上可以通过 count 方法直接使用相同的功能，因此没有理由说 `collect(counting())` 。 此属性还有十五种收集器方法。 它们包括九个方法，其名称以 `summing` ， `averaging` 和 `summarizing` 开头（其功能在相应的原始流类型上可用）。它们还包括 `reduce` 方法的所有重载，以及 filter， `mapping` ， `flatMapping` 和 `collectingAndThen` 方法。 大多数程序员可以安全地忽略大多数这些方法。 从设计的角度来看，这些收集器代表了尝试在收集器中部分复制流的功能，以便下游收集器可以充当「迷你流（ministreams）」。

我们还有三种收集器方法尚未提及。 虽然他们在收 `Collectors` 类中，但他们不涉及集合。 前两个是 `minBy` 和 `maxBy` ，它们取比较器并返回比较器确定的流中的最小或最大元素。 它们是 `Stream` 接口中 `min` 和 `max` 方法的次要总结，是 `BinaryOperator` 中类似命名方法返回的二元运算符的类似收集器。 回想一下，我们在最畅销的专辑中使用了 `BinaryOperator.maxBy` 方法。

最后的 `Collectors` 中方法是 `join` ，它仅对 `CharSequence` 实例（如字符串）的流进行操作。在其无参数形式中，它返回一个简单地连接元素的收集器。 它的一个参数形式采用名为 `delimiter` 的单个 `CharSequence` 参数，并返回一个连接流元素的收集器，在相邻元素之间插入分隔符。 如果传入逗号作为分隔符，则收集器将返回逗号分隔值字符串（但请注意，如果流中的任何元素包含逗号，则字符串将不明确）。 除了分隔符之外，三个参数形式还带有前缀和后缀。 生成的收集器会生成类似于打印集合时获得的字符串，例如 `[came, saw, conquered]` 。

总之，编程流管道的本质是无副作用的函数对象。 这适用于传递给流和相关对象的所有许多函数对象。 终结操作 forEach 仅应用于报告流执行的计算结果，而不是用于执行计算。 为了正确使用流，必须了解收集器。 最重要的收集器工厂是 `toList` ， `toSet` ， `toMap` ， `groupingBy` 和 `join` 。

47. 优先使用 Collection 而不是 Stream 来作为方法的返回类型

许多方法返回元素序列（sequence）。在 Java 8 之前，通常方法的返回类型是 `Collection` ， `Set` 和 `List` 这些接口；还包括 `Iterable` 和数组类型。通常，很容易决定返回哪一种类型。规范（norm）是返回 `Collection` 接口。如果该方法仅用于启用 for-each 循环，或者返回的序列不能实现某些 `Collection` 方法 (通常是 `contains(Object)`)，则使用迭代（ `Iterable` ）接口。如果返回的元素是基本类型或有严格的性能要求，则使用数组。在 Java 8 中，将流（Stream）添加到平台中，这使得为序列返回方法选择适当的返回类型的任务变得非常复杂。

你可能听说过，流现在是返回元素序列的明显的选择，但是正如第 45 条所讨论的，流不会使迭代过时：编写好的代码需要明智地将流和迭代结合起来。如果一个 API 只返回一个流，并且一些用户想用 for-each 循环遍历返回的序列，那么这些用户肯定会感到不安。尤其令人沮丧的是， `Stream` 接口有一个和 `Iterable` 接口中一样的抽象方法，并且 `Stream` 的方法规范与 `Iterable` 中的一致。阻止程序员使用 for-each 循环在流上迭代的唯一原因是 `Stream` 无法继承 `Iterable` 。

遗憾的是，这个问题没有好的解决方法。 乍一看，似乎可以将方法引用传递给 `Stream` 的 iterator 方法。 结果代码可能有些乱，但并非不合理：

```
// Won't compile, due to limitations on Java's type inference
for (ProcessHandle ph : ProcessHandle.allProcesses()::iterator) {
    // Process the process
}
```

不幸的是，如果你试图编译这段代码，会得到一个错误信息：

```
Test.java:6: error: method reference not expected here
for (ProcessHandle ph : ProcessHandle.allProcesses()::iterator) {
```

为了使代码编译，必须将方法引用强制转换为对应参数的 `Iterable` 类型：

```
// Hideous workaround to iterate over a stream
for  (ProcessHandle ph :
(Iterable<ProcessHandle>)ProcessHandle.allProcesses()::iterator)
```

此代码可以工作，但在实践中使用它太乱。 更好的解决方法是使用适配器方法。 JDK 没有提供这样的方法，但是使用上面的代码片段中的相同技术，很容易编写一个方法。 请注意，在适配器方法中不需要强制转换，因为 Java 的类型推断在此上下文中能够正常工作：

```
// Adapter from  Stream<E> to Iterable<E>
public static <E> Iterable<E> iterableOf(Stream<E> stream) {
    return stream::iterator;
}
```

通过这个适配器，你可以使用 for-each 语句迭代任何流：

```
for (ProcessHandle p : iterableOf(ProcessHandle.allProcesses())) {
    // Process the process
}
```

注意，第 34 条中的 `Anagrams` 程序的流版本使用 `Files.lines` 方法读取字典，而迭代版本使用了 scanner。 `Files.lines` 方法优于 scanner，scanner 在读取文件时无声地吞噬所有异常。理想情况下，我们也会在迭代版本中使用 `Files.lines` 。如果 API 只提供对序列的流访问，而程序员希望使用 for-each 语句遍历序列，那么他们就要做出这种妥协。

相反，如果一个程序员想要使用流管道来处理一个序列，那么一个只提供 `Iterable` 的 API 会让他感到不安。JDK 同样没有提供适配器，但是编写这个适配器非常简单：

```
// Adapter from Iterable<E> to Stream<E>
public static <E> Stream<E> streamOf(Iterable<E> iterable) {
    return StreamSupport.stream(iterable.spliterator(), false);
}
```

如果你正在编写一个返回对象序列的方法，并且它只会在流管道中使用，那么当然可以自由地返回流。类似地，返回仅用于迭代的序列的方法应该返回一个 `Iterable` 。但是如果你写一个公共 API，它返回一个序列，你应该为用户提供哪些想写流管道，哪些想写 for-each 语句，除非你有充分的理由相信大多数用户想要使用相同的机制。

`Collection` 接口是 `Iterable` 的子类型，并且具有 `stream` 方法，因此它可以同时提供迭代和流访问的能力。 因此， `Collection` **或适当的子类型通常是公共序列返回方法的最佳返回类型。** 数组也使用 `Arrays.asList` 和 `Stream.of` 方法提供简单的迭代和流访问能力。 如果返回的序列小到足以容易地放入内存中，那么最好返回一个标准集合实现，例如 `ArrayList` 或 `HashSet` 。 **但是不要在只是为了将它作为集合返回，而在内存中存储很大的序列。**

如果你需要返回一很大但可以简洁地表示的序列，请考虑实现一个专用集合。 例如，假设返回给定集合的幂集（power set：就是原集合中所有的子集（包括全集和空集）构成的集族），该集包含其所有子集。 {a，b，c} 的幂集为 {{}，{a}，{b}，{c}，{a，b}，{a，c}，{b，c}，{a，b, c}}。 如果一个集合具有 n 个元素，则幂集具有 2n 个。 因此，你甚至不应考虑将幂集存储在标准集合实现中。 但是，在 `AbstractList` 的帮助下，很容易为此实现自定义集合。

诀窍是使用幂集中每个元素的索引作为位向量（bit vector），其中索引中的第 n 位指示源集合中是否存在第 n 个元素。 本质上，从 0 到 2^n-1 的二进制数和 n 个元素集和的幂集之间存在自然映射。 这是代码：

```
// Returns the power set of an input set as custom collection
public class PowerSet {
   public static final <E> Collection<Set<E>> of(Set<E> s) {
      List<E> src = new ArrayList<>(s);

      if (src.size() > 30)
         throw new IllegalArgumentException("Set too big " + s);

      return new AbstractList<Set<E>>() {
         @Override
         public int size() {
            return 1 << src.size(); // 2 to the power srcSize
         }

         @Override
         public boolean contains(Object o) {
            return o instanceof Set && src.containsAll((Set)o);
         }

         @Override
         public Set<E> get(int index) {
            Set<E> result = new HashSet<>();
```

```
            for (int i = 0; index != 0; i++, index >>= 1)
               if ((index & 1) == 1)
                  result.add(src.get(i));
            return result;
         }
      };
   }
}
```

请注意，如果输入集合超过 30 个元素，则 `PowerSet.of` 方法会引发异常。 这突出了使用 `Collection` 作为返回类型而不是 `Stream` 或 `Iterable` 的缺点： `Collection` 有 int 返回类型的 size 的方法，该方法将返回序列的长度限制为 `Integer.MAX_VALUE` 或 2^{31}-1。 `Collection` 规范允许 size 方法返回 2^{31} - 1，如果集合更大，甚至无限，但这不是一个完全令人满意的解决方案。

为了在 `AbstractCollection` 上编写 `Collection` 实现，除了 `Iterable` 所需的方法之外，只需要实现两种方法： `contains` 和 `size` 。 通常，编写这些方法的有效实现很容易。 如果不可行，可能是因为在迭代发生之前未预先确定序列的内容，返回 `Stream` 还是 `Iterable` 的，无论哪种感觉更自然。 如果选择，可以使用两种不同的方法分别返回。

有时，你会仅根据实现的易用性选择返回类型。例如，假设希望编写一个方法，该方法返回输入列表的所有 (连续的) 子列表。生成这些子列表并将它们放到标准集合中只需要三行代码，但是保存这个集合所需的内存是源列表大小的二次方。虽然这没有指数幂集那么糟糕，但显然是不可接受的。实现自定义集合 (就像我们对幂集所做的那样) 会很乏味，因为 JDK 缺少一个框架 `Iterator` 实现来帮助我们。

然而，实现输入列表的所有子列表的流是直截了当的，尽管它确实需要一点的洞察力（insight）。让我们调用一个子列表，该子列表包含列表的第一个元素和列表的前缀。 例如， （a，b，c）的前缀是（a）， （a，b）和（a，b，c）。 类似地，让我们调用包含后缀的最后一个元素的子列表，因此（a，b，c）的后缀是（a，b，c）， （b，c）和（c）。 洞察力是列表的子列表只是前缀的后缀（或相同的后缀的前缀）和空列表。 这一观察直接展现了一个清晰，合理简洁的实现：

```
// Returns a stream of all the sublists of its input list
public class SubLists {

   public static <E> Stream<List<E>> of(List<E> list) {
      return Stream.concat(Stream.of(Collections.emptyList()),
         prefixes(list).flatMap(SubLists::suffixes));
   }

   private static <E> Stream<List<E>> prefixes(List<E> list) {
      return IntStream.rangeClosed(1, list.size())
         .mapToObj(end -> list.subList(0, end));
   }

   private static <E> Stream<List<E>> suffixes(List<E> list) {
      return IntStream.range(0, list.size())
         .mapToObj(start -> list.subList(start, list.size()));
   }
}
```

请注意，`Stream.concat` 方法用于将空列表添加到返回的流中。 还有，`flatMap` 方法（条目 45）用于生成由所有前缀的所有后缀组成的单个流。 最后，通过映射 `IntStream.range` 和 `IntStream.rangeClosed` 返回的连续 int 值流来生成前缀和后缀。这个习惯用法，粗略地说，流等价于整数索引上的标准 for 循环。因此，我们的子列表实现似于明显的嵌套 for 循环:

```
for (int start = 0; start < src.size(); start++)
    for (int end = start + 1; end <= src.size(); end++)
        System.out.println(src.subList(start, end));
```

可以将这个 for 循环直接转换为流。结果比我们以前的实现更简洁，但可能可读性稍差。它类似于条目 45 中的笛卡尔积的使用流的代码:

```
// Returns a stream of all the sublists of its input list
public static <E> Stream<List<E>> of(List<E> list) {
   return IntStream.range(0, list.size())
      .mapToObj(start ->
         IntStream.rangeClosed(start + 1, list.size())
            .mapToObj(end -> list.subList(start, end)))
      .flatMap(x -> x);
}
```

与之前的 for 循环一样，此代码不会包换空列表。 为了解决这个问题，可以使用 `concat` 方法，就像我们在之前版本中所做的那样，或者在 `rangeClosed` 调用中用 `(int) Math.signum(start)` 替换 1。

这两种子列表的流实现都可以，但都需要一些用户使用流-迭代适配器 (Stream-to-Iterable adapte)，或者在更自然的地方使用流。流-迭代适配器不仅打乱了客户端代码，而且在我的机器上使循环速度降低了 2.3 倍。一个专门构建的 Collection 实现 (此处未显示) 要冗长，但运行速度大约是我的机器上基于流的实现的 1.4 倍。

总之，在编写返回元素序列的方法时，请记住，某些用户可能希望将元素序列作为流处理，而其他用户可能希望迭代方式来处理。 尽量满足两个群体。 如果返回集合是可行的，请执行此操作。 如果已经拥有集合中的元素，或者序列中的元素数量足够小到可以创造一个新的序列，那么返回一个标准集合，比如 `ArrayList` 。 否则，请考虑实现自定义集合，就像我们为幂集程序里所做的那样。 如果返回集合是不可行的，则返回流或可迭代的，无论哪个看起来更自然。 如果在将来的 Java 版本中，`Stream` 接口声明被修改为继承 `Iterable` ，那么你就应该返回 `Stream` ，因为它可以同时被流和迭代处理。

48. 谨慎使用流并行

在主流语言中，Java 一直处于提供简化并发编程任务的工具的最前沿。 当 Java 于 1996 年发布时，它内置了对线程的支持，包括同步和 wait / notify 机制。 Java 5 引入了 `java.util.concurrent` 类库，带有并发集合和执行器框架。 Java 7 引入了 fork-join 包，这是一个用于并行分解的高性能框架。 Java 8 引入了流，可以通过对 `parallel` 方法的单个调用来并行化。 用 Java 编写并发程序变得

越来越容易，但编写正确快速的并发程序还像以前一样困难。 安全和活跃度违规（liveness violation）是并发编程中的事实，并行流管道也不例外。

考虑条目 45 中的程序：

```
// Stream-based program to generate the first 20 Mersenne primes
public static void main(String[] args) {
    primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(System.out::println);
}

static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

在我的机器上，这个程序立即开始打印素数，运行到完成需要 12.5 秒。假设我天真地尝试通过向流管道中添加一个到 `parallel()` 的调用来加快速度。你认为它的表现会怎样?它会快几个百分点吗?慢几个百分点?遗憾的是，它不会打印任何东西，但是 CPU 使用率会飙升到 90%，并且会无限期地停留在那里 (liveness failure:活性失败)。这个程序可能最终会终止，但我不愿意去等待；半小时后我强行阻止了它。

这里发生了什么？简而言之，流类库不知道如何并行化此管道并且启发式失败（heuristics fail）。**即使在最好的情况下，如果源来自 Stream.iterate 方法，或者使用中间操作 limit 方法，并行化管道也不太可能提高其性能。** 这个管道必须应对这两个问题。更糟糕的是，默认的并行策略处理不可预测性的 `limit` 方法，假设在处理一些额外的元素和丢弃任何不必要的结果时没有害处。在这种情况下，找到每个梅森素数的时间大约是找到上一个素数的两倍。因此，计算单个额外元素的成本大致等于计算所有先前元素组合的成本，并且这种无害的管道使自动并行化算法瘫痪。这个故事的寓意很简单：不要无差别地并行化流管道（stream pipelines）。性能后果可能是灾难性的。

通常，并行性带来的性能收益在 `ArrayList` 、 `HashMap` 、 `HashSet` 和 `ConcurrentHashMap` 实例、数组、 `int` 类型范围和 `long` 类型的范围的流上最好。 这些数据结构的共同之处在于，它们都可以精确而廉价地分割成任意大小的子程序，这使得在并行线程之间划分工作变得很容易。用于执行此任务的流泪库使用的抽象是 `spliterator` ，它由 `spliterator` 方法在 `Stream` 和 `Iterable` 上返回。

所有这些数据结构的共同点的另一个重要因素是它们在顺序处理时提供了从良好到极好的引用位置（ locality of reference）： 顺序元素引用在存储器中存储在一块。 这些引用所引用的对象在存储器中可能彼此不接近，这降低了引用局部性。 对于并行化批量操作而言，引用位置非常重要：没有它，线程大部分时间都处于空闲状态，等待数据从内存传输到处理器的缓存中。 具有最佳引用位置的数据结构是基本类型的数组，因为数据本身连续存储在存储器中。

流管道终端操作的性质也会影响并行执行的有效性。 如果与管道的整体工作相比，在终端操作中完成了大量的工作，并且这种操作本质上是连续的，那么并行化管道的有效性将是有限的。 并行性的最佳终操作是缩减（reductions），即使用流的 `reduce` 方法组合管道中出现的所有元素，或者预先打包的 `reduce` (如 `min` 、 `max` 、 `count` 和 `sum`)。短路操作 `anyMatch` 、 `allMatch` 和 `noneMatch` 也可

以支持并行性。由 `Stream` 的 `collect` 方法执行的操作，称为可变缩减（mutable reductions），不适合并行性，因为组合集合的开销非常大。

如果编写自己的 `Stream`，`Iterable` 或 `Collection` 实现，并且希望获得良好的并行性能，则必须重写 `spliterator` 方法并广泛测试生成的流的并行性能。编写高质量的 `spliterator` 很困难，超出了本书的范围。

并行化一个流不仅会导致糟糕的性能，包括活性失败（liveness failures）;它会导致不正确的结果和不可预知的行为（安全故障）。 使用映射器（`mappers`），过滤器（`filters`）和其他程序员提供的不符合其规范的功能对象的管道并行化可能会导致安全故障。Stream 规范对这些功能对象提出了严格的要求。例如，传递给 `Stream` 的 `reduce` 方法操作的累加器（`accumulator`）和组合器（`combiner`）函数必须是关联的，非干扰的和无状态的。如果违反了这些要求（其中一些在第 46 项中讨论过），但按顺序运行你的管道，则可能会产生正确的结果; 如果将它并行化，它可能会失败，也许是灾难性的。

沿着这些思路，值得注意的是，即使并行的梅森素数程序已经运行完成，它也不会以正确的（升序的）顺序打印素数。为了保持顺序版本显示的顺序，必须将 `forEach` 终端操作替换为 `forEachOrdered` 操作，它保证以遇出现顺序（encounter order）遍历并行流。

即使假设正在使用一个高效的可拆分的源流、一个可并行化的或廉价的终端操作以及非干扰的函数对象，也无法从并行化中获得良好的加速效果，除非管道做了足够的实际工作来抵消与并行性相关的成本。作为一个非常粗略的估计，流中的元素数量乘以每个元素执行的代码行数应该至少是 100,000 [Lea14]。

重要的是要记住并行化流是严格的性能优化。与任何优化一样，必须在更改之前和之后测试性能，以确保它值得做（详见第 67 条）。理想情况下，应该在实际的系统设置中执行测试。通常，程序中的所有并行流管道都在公共 fork-join 池中运行。单个行为不当的管道可能会损害系统中不相关部分的其他行为。

如果在并行化流管道时，这种可能性对你不利，那是因为它们确实存在。一个认识的人，他维护一个数百万行代码库，大量使用流，他发现只有少数几个地方并行流是有效的。这并不意味着应该避免并行化流。**在适当的情况下，只需向流管道添加一个 `parallel` 方法调用，就可以实现处理器内核数量的近似线性加速。** 某些领域，如机器学习和数据处理，特别适合这些加速。

```
// 作为并行性有效的流管道的简单示例，请考虑此函数来计算π(n)，素数小于或等于 n：
// Prime-counting stream pipeline - benefits from parallelization
static long pi(long n) {
    return LongStream.rangeClosed(2, n)
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
}
```

在我的机器上，使用此功能计算π（10^8）需要 31 秒。只需添加 `parallel()` 方法调用即可将时间缩短为 9.2 秒：

```
// Prime-counting stream pipeline - parallel version
static long pi(long n) {
    return LongStream.rangeClosed(2, n)
        .parallel()
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
}
```

换句话说，在我的四核计算机上，并行计算速度提高了 3.7 倍。值得注意的是，这不是你在实践中如何计算π(n) 为 n 的值。还有更有效的算法，特别是 Lehmer's formula。

如果要并行化随机数流，请从 `SplittableRandom` 实例开始，而不是 `ThreadLocalRandom`（或基本上过时的 `Random`）。 `SplittableRandom` 专为此用途而设计，具有线性加速的潜力。 `ThreadLocalRandom` 设计用于单个线程，并将自身适应作为并行流源，但不会像 `SplittableRandom` 一样快。 `Random` 实例在每个操作上进行同步，因此会导致过度的并行杀死争用（parallelism-killing contention）。

总之，甚至不要尝试并行化流管道，除非你有充分的理由相信它将保持计算的正确性并提高其速度。不恰当地并行化流的代价可能是程序失败或性能灾难。如果您认为并行性是合理的，那么请确保您的代码在并行运行时保持正确，并在实际情况下进行仔细的性能度量。如果您的代码是正确的，并且这些实验证实了您对性能提高的怀疑，那么并且只有这样才能在生产代码中并行化流。

49. 检查参数有效性

本章（第 8 章）讨论了方法设计的几个方面：如何处理参数和返回值，如何设计方法签名以及如何记载方法文档。 本章中的大部分内容适用于构造方法和其他普通方法。 与第 4 章一样，本章重点关注可用性，健壮性和灵活性上。

大多数方法和构造方法对可以将哪些值传递到其对应参数中有一些限制。 例如，索引值必须是非负数，对象引用必须为非 `null`。 你应该清楚地在文档中记载所有这些限制，并在方法主体的开头用检查来强制执行。 应该尝试在错误发生后尽快检测到错误，这是一般原则的特殊情况。 如果不这样做，则不太可能检测到错误，并且一旦检测到错误就更难确定错误的来源。

如果将无效参数值传递给方法，并且该方法在执行之前检查其参数，则它抛出适当的异常然后快速且清楚地以失败结束。 如果该方法无法检查其参数，可能会发生一些事情。 在处理过程中，该方法可能会出现令人困惑的异常。 更糟糕的是，该方法可以正常返回，但默默地计算错误的结果。 最糟糕的是，该方法可以正常返回但是将某个对象置于受损状态，在将来某个未确定的时间在代码中的某些不相关点处导致错误。 换句话说，验证参数失败可能导致违反故障原子性（failure atomicity）（详见第 76 条）。

对于公共方法和受保护方法，请使用 Java 文档 `@throws` 注解来记在在违反参数值限制时将引发的异常（条目 74）。 通常，生成的异常是 `IllegalArgumentException`， `IndexOutOfBoundsException` 或 `NullPointerException`（条目 72）。 一旦记录了对方法参数的限制，并且记录了违反这些限制时将引发的异常，那么强制执行这些限制就很简单了。 这是一个典型的例子：

```
/**
 * Returns a BigInteger whose value is (this mod m). This method
 * differs from the remainder method in that it always returns a
 * non-negative BigInteger.
 *
 * @param m the modulus, which must be positive
 * @return this mod m
 * @throws ArithmeticException if m is less than or equal to 0
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("Modulus <= 0: " + m);

    ... // Do the computation

}
```

请注意，文档注释没有说「如果 m 为 null，mod 抛出 `NullPointerException`」，尽管该方法正是这样做的，这是调用 `m.sgn()` 的副产品。这个异常记载在类级别文档注释中，用于包含的 `BigInteger` 类。类级别的注释应用于类的所有公共方法中的所有参数。这是避免在每个方法上分别记录每个 `NullPointerException` 的好方法。它可以与 `@Nullable` 或类似的注释结合使用，以表明某个特定参数可能为空，但这种做法不是标准的，为此使用了多个注解。

在 Java 7 中添加的 `Objects.requireNonNull 方` 法灵活方便，因此没有理由再手动执行空值检查。 如果愿意，可以指定自定义异常详细消息。 该方法返回其输入的值，因此可以在使用值的同时执行空检查：

```
// Inline use of Java's null-checking facility
this.strategy = Objects.requireNonNull(strategy, "strategy");
```

你也可以忽略返回值，并使用 `Objects.requireNonNull` 作为满足需求的独立空值检查。

在 Java 9 中， `java.util.Objects` 类中添加了范围检查工具。 此工具包含三个方法： `checkFromIndexSize` ， `checkFromToIndex` 和 `checkIndex` 。 此工具不如空检查方法灵活。 它不允许指定自己的异常详细消息，它仅用于列表和数组索引。 它不处理闭合范围（包含两个端点）。 但如果它能满足你的需要，那就很方便了。

对于未导出的方法，作为包的作者，控制调用方法的环境，这样就可以并且应该确保只传入有效的参数值。因此，非公共方法可以使用断言检查其参数，如下所示：

```
// Private helper function for a recursive sort
private static void sort(long a[], int offset, int length) {
    assert a != null;
    assert offset >= 0 && offset <= a.length;
    assert length >= 0 && length <= a.length - offset;
    ... // Do the computation
}
```

本质上，这些断言声称断言条件将成立，无论其客户端如何使用封闭包。与普通的有效性检查不同，断言如果失败会抛出 `AssertionError` 。与普通的有效性检查不同的是，除非使用 `-ea` (或者-enableassertions）标记传递给 java 命令来启用它们，否则它们不会产生任何效果，本质上也不会产生任何成本。有关断言的更多信息，请参阅教程assert。

检查方法中未使用但存储以供以后使用的参数的有效性尤为重要。例如，考虑第 101 页上的静态工厂方法，它接受一个 int 数组并返回数组的 List 视图。如果客户端传入 null，该方法将抛出 `NullPointerException` ，因为该方法具有显式检查 (调用 `Objects.requireNonNull` 方法)。如果省略了该检查，则该方法将返回对新创建的 `List` 实例的引用，该实例将在客户端尝试使用它时立即抛出 `NullPointerException` 。 到那时，List 实例的来源可能很难确定，这可能会使调试任务大大复杂化。

构造方法是这个原则的一个特例，你应该检查要存储起来供以后使用的参数的有效性。检查构造方法参数的有效性对于防止构造对象违反类不变性（class invariants）非常重要。

你应该在执行计算之前显式检查方法的参数，但这一规则也有例外。 一个重要的例外是有效性检查昂贵或不切实际的情况，并且在进行计算的过程中隐式执行检查。 例如，考虑一种对对象列表进行排序的方法，例如 `Collections.sort(List)` 。 列表中的所有对象必须是可相互比较的。 在对列表进行排序的过程中，列表中的每个对象都将与其他对象进行比较。 如果对象不可相互比较，则某些比较操作抛出 `ClassCastException` 异常，这正是 `sort` 方法应该执行的操作。 因此，提前检查列表中的元素是否具有可比性是没有意义的。 但请注意，不加选择地依赖隐式有效性检查会导致失败原子性（failure atomicity）的丢失（详见第 76 条）。

有时，计算会隐式执行必需的有效性检查，但如果检查失败则会抛出错误的异常。 换句话说，计算由于无效参数值而自然抛出的异常与文档记录方法抛出的异常不匹配。 在这些情况下，你应该使用条目 73 中描述的异常翻译（exception translation）习惯用法将自然异常转换为正确的异常。

不要从本条目中推断出对参数的任意限制都是一件好事。 相反，你应该设计一些方法，使其尽可能通用。 假设方法可以对它接受的所有参数值做一些合理的操作，那么对参数的限制越少越好。 但是，通常情况下，某些限制是正在实现的抽象所固有的。

总而言之，每次编写方法或构造方法时，都应该考虑对其参数存在哪些限制。 应该记在这些限制，并在方法体的开头使用显式检查来强制执行这些限制。 养成这样做的习惯很重要。 在第一次有效性检查失败时，它所需要的少量工作将会得到对应的回报。

50. 必要时进行防御性拷贝

愉快使用 Java 的原因，它是一种安全的语言（safe language）。 这意味着在缺少本地方法（native methods）的情况下，它不受缓冲区溢出，数组溢出，野指针以及其他困扰 C 和 C++ 等不安全语言的内存损坏错误的影响。 在一种安全的语言中，无论系统的任何其他部分发生什么，都可以编写类并确切地知道它们的不变量会保持不变。 在将所有内存视为一个巨大数组的语言中，这是不可能的。

即使在一种安全的语言中，如果不付出一些努力，也不会与其他类隔离。**必须防御性地编写程序，假定类的客户端尽力摧毁类其不变量。**随着人们更加努力地试图破坏系统的安全性，这种情况变得越来越真实，但更常见的是，你的类将不得不处理由于善意得程序员诚实错误而导致的意外行为。不管怎样，花时间编写在客户端行为不佳的情况下仍然保持健壮的类是值得的。

如果没有对象的帮助，另一个类是不可能修改对象的内部状态的，但是在无意的情况下提供这样的帮助却非常地容易。例如，考虑以下类，表示一个不可变的时间期间:

```
// Broken "immutable" time period class
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param  start the beginning of the period
     * @param  end the end of the period; must not precede start
     * @throws IllegalArgumentException if start is after end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(
                start + " after " + end);

        this.start = start;
        this.end   = end;
    }

    public Date start() {
        return start;
    }

    public Date end() {
        return end;
    }
    ...    // Remainder omitted
}
```

乍一看，这个类似乎是不可变的，并强制执行不变式，即 `period` 实例的开始时间并不在结束时间之后。然而，利用 `Date` 类是可变的这一事实很容易违反这个不变式:

```
// Attack the internals of a Period instance
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
end.setYear(78);  // Modifies internals of p!
```

从 Java 8 开始，解决此问题的显而易见的方法是使用 `Instant`（或 `LocalDateTime` 或 `ZonedDateTime`）代替 `Date`，因为 `Instant` 和其他 `java.time` 包下的类是不可变的（条目 17）。**Date 已过时，不应再在新代码中使用。** 也就是说，问题仍然存在：有时必须在 API 和内部表示中使用可变值类型，本条目中讨论的技术也适用于这些时间。

为了保护 `Period` 实例的内部不受这种攻击，必须将每个可变参数的防御性拷贝应用到构造方法中，并将拷贝用作 `Period` 实例的组件，以替代原始实例:

```
// Repaired constructor - makes defensive copies of parameters
public Period(Date start, Date end) {
    this.start = new Date(start.getTime());
    this.end   = new Date(end.getTime());

    if (this.start.compareTo(this.end) > 0)
      throw new IllegalArgumentException(
          this.start + " after " + this.end);
}
```

有了新的构造方法后，前面的攻击将不会对 `Period` 实例产生影响。注意，防御性拷贝是在检查参数 (条目 49) 的有效性之前进行的，有效性检查是在拷贝上而不是在原始实例上进行的。虽然这看起来不自然，但却是必要的。它在检查参数和拷贝参数之间的漏洞窗口期间保护类不受其他线程对参数的更改的影响。在计算机安全社区中，这称为 time-of-check/time-of-use 或 TOCTOU 攻击[Viega01]。

还请注意，我们没有使用 Date 的 `clone` 方法来创建防御性拷贝。因为 Date 是非 final 的，所以 clone 方法不能保证返回类为 `java.util.Date` 的对象,它可以返回一个不受信任的子类的实例，这个子类是专门为恶意破坏而设计的。例如，这样的子类可以在创建时在私有静态列表中记录对每个实例的引用，并允许攻击者访问该列表。这将使攻击者可以自由控制所有实例。为了防止这类攻击，不要使用 `clone` 方法对其类型可由不可信任子类化的参数进行防御性拷贝。

虽然替换构造方法成功地抵御了先前的攻击，但是仍然可以对 `Period` 实例进行修改，因为它的访问器提供了对其可变内部结构的访问:

```
// Second attack on the internals of a Period instance
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
p.end().setYear(78);  // Modifies internals of p!
```

为了抵御第二次攻击，只需修改访问器以返回可变内部字属性的防御性拷贝:

```
// Repaired accessors - make defensive copies of internal fields
public Date start() {
    return new Date(start.getTime());
}

public Date end() {
    return new Date(end.getTime());
}
```

使用新的构造方法和新的访问器，`Period` 是真正不可变的。 无论程序员多么恶意或不称职，根本没有办法违反一个 `period` 实例的开头不跟随其结束的不变量（不使用诸如本地方法和反射之类的语言外方法）。 这是正确的，因为除了 `period` 本身之外的任何类都无法访问 `period` 实例中的任何可变属性。 这些属性真正封装在对象中。

在访问器中，与构造方法不同，允许使用 `clone` 方法来制作防御性拷贝。 这是因为我们知道 `Period` 的内部 `Date` 对象的类是 `java.util.Date`，而不是一些不受信任的子类。 也就是说，由于条目 13 中列出的原因，通常最好使用构造方法或静态工厂来拷贝实例。

参数的防御性拷贝不仅仅适用于不可变类。 每次编写在内部数据结构中存储对客户端提供的对象的引用的方法或构造函数时，请考虑客户端提供的对象是否可能是可变的。 如果是，请考虑在将对象输入数据结构后，你的类是否可以容忍对象的更改。 如果答案是否定的，则必须防御性地拷贝对象，并将拷贝输入到数据结构中，以替代原始数据结构。 例如，如果你正在考虑使用客户端提供的对象引用作为内部 set 实例中的元素或作为内部 map 实例中的键，您应该意识到如果对象被修改后插入，对象的 set 或 map 的不变量将被破坏。

在将内部组件返回给客户端之前进行防御性拷贝也是如此。无论你的类是否是不可变的，在返回对可拜年的内部组件的引用之前，都应该三思。可能的情况是，应该返回一个防御性拷贝。记住，非零长度数组总是可变的。因此，在将内部数组返回给客户端之前，应该始终对其进行防御性拷贝。或者，可以返回数组的不可变视图。这两项技术都记载于条目 15。

可以说，所有这些的真正教训是，在可能的情况下，应该使用不可变对象作为对象的组件，这样就不必担心防御性拷贝（详见第 17 条）。在我们的 `Period` 示例中，使用 `Instant`（或 `LocalDateTime` 或 `ZonedDateTime`），除非使用的是 Java 8 之前的版本。如果使用的是较早的版本，则一个选项是存储 `Date.getTime()` 返回的基本类型 long 来代替 Date 引用。

可能存在与防御性拷贝相关的性能损失，并且它并不总是合理的。如果一个类信任它的调用者不修改内部组件，也许是因为这个类和它的客户端都是同一个包的一部分，那么它可能不需要防御性的拷贝。在这些情况下，类文档应该明确指出调用者不能修改受影响的参数或返回值。

即使跨越包边界，在将可变参数集成到对象之前对其进行防御性拷贝也并不总是合适的。有些方法和构造方法的调用指示参数引用的对象的显式切换。当调用这样的方法时，客户端承诺不再直接修改对象。希望获得客户端提供的可变对象的所有权的方法或构造方法必须在其文档中明确说明这一点。

包含方法或构造方法的类，这些方法或构造方法的调用指示控制权的转移，这些类无法防御恶意客户端。 只有当一个类和它的客户之间存在相互信任，或者当对类的不变量造成损害时，除了客户之外，任何人都不会受到损害。 后一种情况的一个例子是包装类模式（详见第 18 条）。 根据包装类的性质，客户端可以通过在包装后直接访问对象来破坏类的不变性，但这通常只会损害客户端。

总之，如果一个类有从它的客户端获取或返回的可变组件，那么这个类必须防御性地拷贝这些组件。如果拷贝的成本太高，并且类信任它的客户端不会不适当地修改组件，则可以用文档替换防御性拷贝，该文档概述了客户端不得修改受影响组件的责任。

51. 仔细设计方法签名

这一条目是 API 设计提示的大杂烩，但它们本身并足以设立一个单独的条目。综合起来，这些设计提示将帮助你更容易地学习和使用 API，并且更不容易出错。

仔细选择方法名名称。名称应始终遵守标准命名约定（详见第 68 条）。你的主要目标应该是选择与同一包中的其他名称一致且易于理解的名称。其次是应该是选择与更广泛的共识一致的名称。避免使用较长的方法名。如果有疑问，可以从 Java 类库 API 中寻求指导。尽管类库中也存在许多不一致之处（考虑到这些类库的规模和范围，这是不可避免的），也提供了相当客观的认可和共识。

不要过分地提供方便的方法。每种方法都应该"尽其所能"。太多的方法使得类难以学习、使用、文档化、测试和维护。对于接口更是如此，在接口中，太多的方法使实现者和用户的工作变得复杂。对于类或接口支持的每个操作，提供一个功能完整的方法。只有在经常使用时，才考虑提供「快捷方式（shortcut）」。**如果有疑问，请将其删除。**

避免过长的参数列表。目标是四个或更少的参数。大多数程序员不能记住更长的参数列表。如果你的许多方法超过了这个限制，如果未经常引用其文档的情况下，那么你的 API 将无法使用。现代 IDE 编辑器会提供帮助，但是使用简短的参数列表仍然会更好。**相同类型参数的长序列尤其有害。**用户不仅不能记住参数的顺序，而且当他们意外地弄错参数顺序时，他们的程序仍然会编译和运行。只是不会按照作者的意图去执行。

有三种技术可以缩短过长的参数列表。 一种方法是将方法分解为多个方法，每个方法只需要参数的一个子集。 如果不小心，这可能会导致太多方法，但它也可以通过增加正交性（orthogonality）来减少方法个数。 例如，考虑 java.util.List 接口。 它没有提供查找子列表中元素的第一个或最后一个索引的方法，这两个索引都需要三个参数。 相反，它提供了 `subList` 方法，该方法接受两个参数并返回子列表的视图。 此方法可以与 `indexOf` 或 `lastIndexOf` 方法结合使用，这两个方法都有一个参数，以生成所需的功能。 此外， `subList` 方法可以与在 List 实例上操作的任何方法组合，以对子列表执行任意计算。 得到的 API 具有非常高的功率重量 (power-to-weight) 比。

缩短过长参数列表的第二种技术是创建辅助类来保存参数组。这些辅助类通常是静态成员类 (条目 24)。如果看到一个频繁出现的参数序列表示某个不同的实体，建议使用这种技术。例如，假设正在编写一个表示纸牌游戏的类，并且发现不断地传递一个由两个参数组成的序列，这些参数表示纸牌的点数和花色。如果添加一个辅助类来表示卡片，并用辅助类的单个参数替换参数序列的每次出现，那么 API 和类的内部结构可能会受益。

结合前两个方面的第三种技术是，从对象构造到方法调用采用 Builder 模式 (条目 2)。如果你有一个方法有许多参数，特别是其中一些是可选的，那么可以定义一个对象来表示所有的参数，并允许客户端在这个对象上进行多个「setter」调用，每次设置一个参数或较小相关的组。设置好所需的参数后，客户端调用对象的「execute」方法，该方法对参数进行最后的有效性检查，并执行实际的计算。

对于参数类型，优先选择接口而不是类（详见第 64 条）。如果有一个合适的接口来定义一个参数，那么使用它来支持一个实现该接口的类。例如，没有理由在编写方法时使用 HashMap 作为输入参数，相反，而是使用 Map 作为参数，这允许传入 HashMap、TreeMap、ConcurrentHashMap、TreeMap 的子 Map（submap）或任何尚未编写的 Map 实现。通过使用的类而不是接口，就把客户端限制在特定的实现中，如果输入数据碰巧以其他形式存在，则强制执行不必要的、代价高昂的复制操作。

与布尔型参数相比，优先使用两个元素枚举类型，除非布尔型参数的含义在方法名中是明确的。枚举类型使代码更容易阅读和编写。此外，它们还可以方便地在以后添加更多选项。例如，你可能有一个 `Thermometer` 类型的静态工厂方法，这个方法的签名是以下这个枚举：

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

`Thermometer.newInstance(TemperatureScale.CELSIUS)` 不仅比 `Thermometer.newInstance(true)` 更有意义，而且可以在将来的版本中将 `KELVIN` 添加到 `TemperatureScale` 中，而无需向 `Thermometer` 添加新的静态工厂。 此外，还可以将温度刻度（temperature-scale）依赖关系重构为枚举常量的方法（详见第 34 条）。 例如，每个刻度常量可以有一个采用 double 值并将其转换为 `Celsius` 的方法。

52. 明智审慎地使用重载

下面的程序是一个善意的尝试，根据 Set、List 或其他类型的集合对它进行分类：

```
// Broken! - What does this program print?
public class CollectionClassifier {

    public static String classify(Set<?> s) {
        return "Set";
    }

    public static String classify(List<?> lst) {
        return "List";
    }

    public static String classify(Collection<?> c) {
        return "Unknown Collection";
    }

    public static void main(String[] args) {
        Collection<?>[] collections = {
            new HashSet<String>(),
            new ArrayList<BigInteger>(),
            new HashMap<String, String>().values()
        };

        for (Collection<?> c : collections)
            System.out.println(classify(c));
    }
}
```

您可能希望此程序打印 Set，然后是 List 和 Unknown Collection 字符串，实际上并没有。 而是打印了三次 Unknown Collection 字符串。 为什么会这样？ 因为 `classify` 方法被重载了，**在编译时选择要调用哪个重载方法。** 对于循环的所有三次迭代，参数的编译时类型是相同的：`Collection<?>` 。 运行时类型在每次迭代中都不同，但这不会影响对重载方法的选择。 因为参数的编译时类型是 `Collection<?>` ，所以唯一适用的重载是第三个 `classify(Collection<?> c)` 方法，并且在循环的每次迭代中调用这个重载。

此程序的行为是违反直觉的，因为**重载（overloaded）方法之间的选择是静态的，而重写（overridden）方法之间的选择是动态的。** 根据调用方法的对象的运行时类型，在运行时选择正确版本的重写方法。 作为提醒，当子类包含与父类中具有相同签名的方法声明时，会重写此方法。 如果在子类中重写实例方法并且在子类的实例上调用，则无论子类实例的编译时类型如何，都会执行子类的重写方法。 为了具体说明，请考虑以下程序：

```
class Wine {
    String name() { return "wine"; }
}

class SparklingWine extends Wine {
    @Override String name() { return "sparkling wine"; }
}

class Champagne extends SparklingWine {
    @Override String name() { return "champagne"; }
}

public class Overriding {
    public static void main(String[] args) {
        List<Wine> wineList = List.of(
            new Wine(), new SparklingWine(), new Champagne());

        for (Wine wine : wineList)
            System.out.println(wine.name());
    }
}
```

`name` 方法在 `Wine` 类中声明，并在子类 `SparklingWine` 和 `Champagne` 中重写。 正如你所料，此程序打印出 wine，sparkling wine 和 champagne，即使实例的编译时类型在循环的每次迭代中都是 `Wine` 。 当调用重写方法时，对象的编译时类型对执行哪个方法没有影响; 总是会执行“最具体 (most specific)”的重写方法。 将此与重载进行比较，其中对象的运行时类型对执行的重载没有影响; 选择是在编译时完成的，完全基于参数的编译时类型。

在 `CollectionClassifier` 示例中，程序的目的是通过基于参数的运行时类型自动调度到适当的方法重载来辨别参数的类型，就像 Wine 类中的 name 方法一样。 方法重载根本不提供此功能。 假设需要一个静态方法，修复 `CollectionClassifier` 程序的最佳方法是用一个执行显式 `instanceof` 测试的方法替换 `classify` 的所有三个重载：

```
  public static String classify(Collection<?> c) {
    return c instanceof Set  ? "Set" :
           c instanceof List ? "List" : "Unknown Collection";
}
```

因为重写是规范，而重载是例外，所以重写设置了人们对方法调用行为的期望。 正如 `CollectionClassifier` 示例所示，重载很容易混淆这些期望。 编写让程序员感到困惑的代码的行为是不好的实践。 对于 API 尤其如此。 如果 API 的日常用户不知道将为给定的参数集调用多个方法重载中的哪一个，则使用 API 可能会导致错误。 这些错误很可能表现为运行时的不稳定行为，许多程序员很难诊断它们。 因此，**应该避免混淆使用重载。**

究竟是什么构成了重载的混乱用法还有待商榷。**一个安全和保守的策略是永远不要导出两个具有相同参数数量的重载。**如果一个方法使用了可变参数，除非如第 53 条目所述，保守策略是根本不重载它。如果遵守这些限制，程序员就不会怀疑哪些重载适用于任何一组实际参数。这些限制并不十分繁重，因为**总是可以为方法赋予不同的名称，而不是重载它们。**

例如，考虑 `ObjectOutputStream` 类。对于每个基本类型和几个引用类型，它都有其 `write` 方法的变体。这些变体都有不同的名称，例如 `writeBoolean(boolean)` 、 `writeInt(int)` 和 `writeLong(long)` ，而不是重载 `write` 方法。与重载相比，这种命名模式的另一个好处是，可以为 `read` 方法提供相应的名称，例如 `readBoolean()` 、 `readInt()` 和 `readLong()` 。 `ObjectInputStream` 类实际上提供了这样的读取方法。

对于构造方法，无法使用不同的名称：类的多个构造函数总是被重载。 在许多情况下，可以选择导出静态工厂而不是构造方法（详见第 1 条）。 此外，使用构造方法，不必担心重载和重写之间的影响，因为构造方法不能被重写。 你可能有机会导出具有相同数量参数的多个构造函数，因此知道如何安全地执行它是值得的。

如果总是清楚哪个重载将应用于任何给定的实际参数集，那么用相同数量的参数导出多个重载不太可能让程序员感到困惑。在这种情况下，每对重载中至少有一个对应的形式参数在这两个重载中具有「完全不同的」类型。如果显然不可能将任何非空表达式强制转换为这两种类型，那么这两种类型是完全不同的。在这些情况下，应用于给定实际参数集的重载完全由参数的运行时类型决定，且不受其编译时类型的影响，因此消除了一个主要的混淆。例如，ArrayList 有一个接受 int 的构造方法和第二个接受 Collection 的构造方法。很难想象在任何情况下，这两个构造方法在调用时哪个会产生混淆。

在 Java 5 之前，所有基本类型都与引用类型完全不同，但在自动装箱存在的情况下，则并非如此，并且它已经造成了真正的麻烦。 考虑以下程序：

```
public class SetList {
    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<>();
        List<Integer> list = new ArrayList<>();

        for (int i = -3; i < 3; i++) {
            set.add(i);
            list.add(i);
        }

        for (int i = 0; i < 3; i++) {
            set.remove(i);
            list.remove(i);
        }

        System.out.println(set + " " + list);
    }
```

```
}
```

首先，程序将从-3 到 2 的整数添加到有序集合和列表中。 然后，它在集合和列表上进行三次相同的 `remove` 方法调用。 如果你和大多数人一样，希望程序从集合和列表中删除非负值（0, 1 和 2）并打印[-3, -2, -1] [- 3, -2, -1]。 实际上，程序从集合中删除非负值，从列表中删除奇数值，并打印[-3, -2, -1] [-2, 0, 2]。 称这种混乱的行为是一种保守的说法。

实际情况是：调用 `set.remove(i)` 选择重载 `remove(E)` 方法，其中 `E` 是 `set (Integer)` 的元素类型，将基本类型 i 由 int 自动装箱为 Integer 中。这是你所期望的行为，因此程序最终会从集合中删除正值。另一方面，对 `list.remove(i)` 的调用选择重载 `remove(int i)` 方法，它将删除列表中指定位置的元素。如果从列表[-3, -2, -1, 0, 1, 2] 开始，移除第 0 个元素，然后是第 1 个，然后是第二个，就只剩下[-2, 0, 2]，谜底就解开了。若要修复此问题，请强制转换 `list.remove` 的参数为 `Integer` 类型，迫使选择正确的重载。或者，也可以调用 `Integer.valueOf(i)` ，然后将结果传递给 `list.remove` 方法。无论哪种方式，程序都会按预期打印[-3, -2, -1][-3, -2, -1]:

```
for (int i = 0; i < 3; i++) {
    set.remove(i);
    list.remove((Integer) i);  // or remove(Integer.valueOf(i))
}
```

前一个示例所演示的令人混乱的行为是由于 `List<E>` 接口对 `remove` 方法有两个重载：`remove(E)` 和 `remove(int)` 。在 Java 5 之前，当 List 接口被“泛型化”时，它有一个 `remove(Object)` 方法代替 `remove(E)` ，而相应的参数类型 Object 和 int 则完全不同。但是，在泛型和自动装箱的存在下，这两种参数类型不再完全不同了。换句话说，在语言中添加泛型和自动装箱破坏了 List 接口。幸运的是，Java 类库中的其他 API 几乎没有受到类似的破坏，但是这个故事清楚地表明，自动装箱和泛型在重载时增加了谨慎的重要性。

在 Java 8 中添加 lambda 表达式和方法引用以后，进一步增加了重载混淆的可能性。 例如，考虑以下两个代码片段：

```
new Thread(System.out::println).start();

ExecutorService exec = Executors.newCachedThreadPool();

exec.submit(System.out::println);
```

虽然 Thread 构造方法调用和 `submit` 方法调用看起来很相似，但是前者编译而后者不编译。参数是相同的 (`System.out::println`)，两者都有一个带有 `Runnable` 的重载。这里发生了什么?令人惊讶的答案是， `submit` 方法有一个带有 `Callable <T>` 参数的重载，而 `Thread` 构造方法却没有。你可能认为这不会有什么区别，因为 `println` 方法的所有重载都会返回 `void` ，因此方法引用不可能是 `Callable`
。这很有道理，但重载解析算法不是这样工作的。也许同样令人惊讶的是，如果 `println` 方法没有被重载，那么 `submit` 方法调用是合法的。正是被引用的方法（println）的重载和被调用的方法（submit）相结合，阻止了重载解析算法按照你所期望的方式运行。

从技术上讲，问题是 `System.out::println` 是一个不精确的方法引用[JLS，15.13.1]，并且「包含隐式类型的 lambda 表达式或不精确的方法引用的某些参数表达式被适用性测试忽略，因为在选择目标类型之前无法确定它们的含义[JLS，15.12.2]。」如果你不理解这段话也不要担心; 它针对的是编译器编写者。 关键是在同一参数位置中具有不同功能接口的重载方法或构造方法会导致混淆。 因此，**不要在相同参数位置重载采用不同函数式接口的方法**。 在此条目的说法中，不同的函数式接口并没有根本不同。 如果传递命令行开关 `-Xlint:overloads` ，Java 编译器将警告这种有问题的重载。

数组类型和 Object 以外的类是完全不同的。此外，除了 `Serializable` 和 `Cloneable` 之外，数组类型和其他接口类型也完全不同。如果两个不同的类都不是另一个类的后代[JLS, 5.5]，则称它们是不相关的。例如， `String` 和 `Throwable` 是不相关的。任何对象都不可能是两个不相关类的实例，所以不相关的类也是完全不同的。

还有其他『类型对 (pairs of types)』不能在任何方向转换[JLS, 5.1.12]，但是一旦超出上面描述的简单情况，大多数程序员就很难辨别哪些重载 (如果有的话) 适用于一组实际参数。决定选择哪个重载的规则非常复杂，并且随着每个版本的发布而变得越来越复杂。很少有程序员能理解它们所有的微妙之处。

有时候，可能觉得有必要违反这一条目中的指导原则，特别是在演化现有类时。例如，考虑 String，它从 Java 4 开始就有一个 `contenttequals (StringBuffer)` 方法。在 Java 5 中，添加了 `CharSequence` 接口，来为 `StringBuffer` 、 `StringBuilder` 、 `String` 、 `CharBuffer` 和其他类似类型提供公共接口。在添加 `CharSequence` 的同时，String 还配备了一个重载的 `contenttequals` 方法，该方法接受 `CharSequence` 参数。

虽然上面的重载明显违反了此条目中的指导原则，但它不会造成任何危害，因为当在同一个对象引用上调用这两个重载方法时，它们做的是完全相同的事情。程序员可能不知道将调用哪个重载，但只要它们的行为相同，就没有什么后果。确保这种行为的标准方法是，将更具体的重载方法调用转发给更一般的重载方法:

```
// Ensuring that 2 methods have identical behavior by forwarding
public boolean contentEquals(StringBuffer sb) {
    return contentEquals((CharSequence) sb);
}
```

虽然 Java 类库在很大程度上遵循了这一条目中的建议，但是有一些类违反了它。例如，String 导出两个重载的静态工厂方法 `valueOf(char[])` 和 `valueOf(Object)` ，它们在传递相同的对象引用时执行完全不同的操作。对此没有任何正当的理由理由，它应该被视为一种异常现象，有可能造成真正的混乱。

总而言之，仅仅可以重载方法并不意味着应该这样做。通常，最好避免重载具有相同数量参数的多个签名的方法。在某些情况下，特别是涉及构造方法的情况下，可能无法遵循此建议。在这些情况下，至少应该避免通过添加强制转换将相同的参数集传递给不同的重载。如果这是无法避免的，例如，因为要对现有类进行改造以实现新接口，那么应该确保在传递相同的参数时，所有重载的行为都是相同的。如果做不到这一点，程序员将很难有效地使用重载方法或构造方法，也无法理解为什么它不能工作。

53. 明智审慎地使用可变参数

可变参数方法正式名称称为可变的参数数量方法「variable arity methods」 [JLS, 8.4.1]，接受零个或多个指定类型的参数。 可变参数机制首先创建一个数组，其大小是在调用位置传递的参数数量，然后将参数值放入数组中，最后将数组传递给方法。

例如，这里有一个可变参数方法，它接受一系列 int 类型的参数并返回它们的总和。如你所料，`sum(1,2,3)` 的值为 6， `sum()` 的值为 0：

```
// Simple use of varargs
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```

有时，编写一个需要某种类型的一个或多个参数的方法是合适的，而不是零或更多。 例如，假设要编写一个计算其多个参数最小值的方法。 如果客户端不传递任何参数，则此方法定义不明确。 你可以在运行时检查数组长度：

```
// The WRONG way to use varargs to pass one or more arguments!
static int min(int... args) {
    if (args.length == 0)
        throw new IllegalArgumentException("Too few arguments");
    int min = args[0];
    for (int i = 1; i < args.length; i++)
        if (args[i] < min)
            min = args[i];
    return min;
}
```

该解决方案存在几个问题。 最严重的是，如果客户端在没有参数的情况下调用此方法，则它在运行时而不是在编译时失败。 另一个问题是它很难看。 必须在 args 参数上包含显式有效性检查，除非将 `min` 初始化为 `Integer.MAX_VALUE`，否则不能使用 for-each 循环，这也很难看。

幸运的是，有一种更好的方法可以达到预期的效果。 声明方法采用两个参数，一个指定类型的普通参数，另一个此类型的可变参数。 该解决方案纠正了前一个示例的所有缺陷：

```
// The right way to use varargs to pass one or more arguments
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
            min = arg;
    return min;
}
```

从这个例子中可以看出，在需要参数数量可变的方法时，可变参数是有效的。可变参数是为 `printf` 方法而设计的，该方法与可变参数同时添加到 Java 平台中，以及包括经过改造的核心反射机制。 `printf` 和反射机制都从可变参数中受益匪浅。

在性能关键的情况下使用可变参数时要小心。每次调用可变参数方法都会导致数组分配和初始化。如果你从经验上确定负担不起这个成本，但是还需要可变参数的灵活性，那么有一种模式可以让你鱼与熊掌兼得。假设你已确定 95% 的调用是三个或更少的参数的方法，那么声明该方法的五个重载。每个重载方法包含 0 到 3 个普通参数，当参数数量超过 3 个时，使用一个可变参数方法:

```
public void foo() { }

public void foo(int a1) { }

public void foo(int a1, int a2) { }

public void foo(int a1, int a2, int a3) { }

public void foo(int a1, int a2, int a3, int... rest) { }
```

现在你知道，在所有参数数量超过 3 个的方法调用中，只有 5% 的调用需要支付创建数组的成本。与大多数性能优化一样，这种技术通常不太合适，但一旦真正需要的时候，它是一个救星。

`EnumSet` 的静态工厂使用这种技术将创建枚举集合的成本降到最低。这是适当的，因为枚举集合为比特属性提供具有性能竞争力的替换（performance-competitive replacement for bit fields）是至关重要的 （详见第 36 条）。

总之，当需要使用可变数量的参数定义方法时，可变参数非常有用。 在使用可变参数前加上任何必需的参数，并注意使用可变参数的性能后果。

54. 返回空的数组或集合，不要返回 null

像如下的方法并不罕见:

```
// Returns null to indicate an empty collection. Don't do this!
private final List<Cheese> cheesesInStock = ...;

/**
 * @return a list containing all of the cheeses in the shop,
 *     or null if no cheeses are available for purchase.
 */
public List<Cheese> getCheeses() {
    return cheesesInStock.isEmpty() ? null
        : new ArrayList<>(cheesesInStock);
}
```

把没有奶酪（Cheese）可买的情况当做一种特例，这是不合常理的。这样需要在客户端中必须有额外的代码来处理 null 的返回值，如:

```
List<Cheese> cheeses = shop.getCheeses();
if (cheeses != null && cheeses.contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");
```

在几乎每次使用返回 null 来代替空集合或数组的方法时，都需要使用这种迂回的方式。 这样做很容易出错，因为编写客户端的程序员可能忘记编写特殊情况代码来处理 null 返回。 多年来这种错误可能会被忽视，因为这种方法通常会返回一个或多个对象。 此外，返回 null 代替空容器会使返回容器的方法的实现变得复杂。

有时有人认为，null 返回值比空集合或数组更可取，因为它避免了分配空容器的开销。这个论点有两点是不成立的。首先，除非测量结果表明所讨论的分配是性能问题的真正原因，否则不宜担心此级别的性能（详见第 67 条）。第二，可以在不分配空集合和数组的情况下返回它们。下面是返回可能为空的集合的典型代码。通常，这就是你所需要的:

```
//The right way to return a possibly empty collection
public List<Cheese> getCheeses() {
    return new ArrayList<>(cheesesInStock);
}
```

如果有证据表明分配空集合会损害性能，可以通过重复返回相同的不可变空集合来避免分配，因为不可变对象可以自由共享（详见第 17 条）。下面的代码就是这样做的，使用了 `Collections.emptyList` 方法。如果你要返回一个 Set，可以使用 `Collections.emptySet` ；如果要返回 Map，则使用 `Collections.emptyMap` 。但是请记住，这是一个优化，很少需要它。如果你认为你需要它，测量一下前后的性能表现，确保它确实有帮助：

```
// Optimization - avoids allocating empty collections
public List<Cheese> getCheeses() {
    return cheesesInStock.isEmpty() ? Collections.emptyList()
        : new ArrayList<>(cheesesInStock);
}
```

数组的情况与集合的情况相同。 永远不要返回 null，而是返回长度为零的数组。 通常，应该只返回一个正确长度的数组，这个长度可能为零。 请注意，我们将一个长度为零的数组传递给 `toArray` 方法，以指示所需的返回类型，即 `Cheese []` ：

```
//The right way to return a possibly empty array
public Cheese[] getCheeses() {
    return cheesesInStock.toArray(new Cheese[0]);
}
```

如果你认为分配零长度数组会损害性能，则可以重复返回相同的零长度数组，因为所有零长度数组都是不可变的：

```
// Optimization - avoids allocating empty arrays
private static final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];

public Cheese[] getCheeses() {
    return cheesesInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

在优化的版本中，我们将相同的空数组传递到每个 `toArray` 调用中，当 `cheesesInStock` 为空时，这个数组将从 `getCheeses` 返回。不要为了提高性能而预先分配传递给 `toArray` 的数组。研究表明，这样做会适得其反[Shipilev16]:

```
// Don't do this - preallocating the array harms performance!
return cheesesInStock.toArray(new Cheese[cheesesInStock.size()]);
```

总之，**永远不要返回 null 来代替空数组或集合。**它使你的 API 更难以使用，更容易出错，并且没有性能优势。

55. 明智审慎地返回 Optional

在 Java 8 之前，编写在特定情况下无法返回任何值的方法时，可以采用两种方法。要么抛出异常，要么返回 null（假设返回类型是对象是引用类型）。但这两种方法都不完美。应该为异常条件保留异常（详见第 69 条），并且抛出异常代价很高，因为在创建异常时捕获整个堆栈跟踪。返回 null 没有这些缺点，但是它有自己的缺陷。如果方法返回 null，客户端必须包含特殊情况代码来处理 null 返回的可能性，除非程序员能够证明 null 返回是不可能的。如果客户端忽略检查 null 返回并将 null 返回值存储在某个数据结构中，那么会在将来的某个时间在与这个问题不相关的代码位置上，抛出 `NullPointerException` 异常的可能性。

在 Java 8 中，还有第三种方法来编写可能无法返回任何值的方法。`Optional<T>` 类表示一个不可变的容器，它可以包含一个非 null 的 `T` 引用，也可以什么都不包含。不包含任何内容的 Optional 被称为空（empty）。非空的包含值称的 Optional 被称为存在（present）。Optional 的本质上是一个不可变的集合，最多可以容纳一个元素。`Optional<T>` 没有实现 `Collection<T>` 接口，但原则上是可以。

在概念上返回 T 的方法，但在某些情况下可能无法这样做，可以声明为返回一个 `Optional<T>`。这允许该方法返回一个空结果，以表明不能返回有效的结果。返回 Optional 的方法比抛出异常的方法更灵活、更容易使用，而且比返回 null 的方法更不容易出错。

在条目 30 中，我们展示了根据集合中元素的自然顺序计算集合最大值的方法。

```
// Returns maximum value in collection - throws exception if empty
public static <E extends Comparable<E>> E max(Collection<E> c) {
    if (c.isEmpty())
        throw new IllegalArgumentException("Empty collection");

    E result = null;
    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
            result = Objects.requireNonNull(e);
    return result;
}
```

如果给定集合为空，此方法将抛出 `IllegalArgumentException` 异常。我们在条目 30 中提到，更好的替代方法是返回 `Optional<E>` 。下面是修改后的方法:

```
// Returns maximum value in collection as an Optional<E>
public static <E extends Comparable<E>>
        Optional<E> max(Collection<E> c) {
    if (c.isEmpty())
        return Optional.empty();

    E result = null;
    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
            result = Objects.requireNonNull(e);
    return Optional.of(result);
}
```

如你所见，返回 Optional 很简单。 你所要做的就是使用适当的静态工厂创建 Optional。 在这个程序中，我们使用两个： `Optional.empty()` 返回一个空的 Optional， `Optional.of(value)` 返回一个包含给定非 null 值的 Optional。 将 null 传递给 `Optional.of(value)` 是一个编程错误。 如果这样做，该方法通过抛出 `NullPointerException` 异常作为回应。 `Optional.of(value)` 方法接受一个可能为 null 的值，如果传入 null 则返回一个空的 Optional。**永远不要通过返回 Optional 的方法返回一个空值**：它破坏 Optional 设计的初衷。

`Stream` 上的很多终止操作返回 Optional。如果我们重写 max 方法来使用一个 `Stream` ，那么 `Stream` 的 `max` 操作会为我们生成 Optional 的工作 (尽管我们还是传递一个显式的 `Comparator`):

```
// Returns max val in collection as Optional<E> - uses stream
public static <E extends Comparable<E>>
        Optional<E> max(Collection<E> c) {
    return c.stream().max(Comparator.naturalOrder());
}
```

那么，如何选择返回 Optional 而不是返回 null 或抛出异常呢？`Optional` 在本质上类似于检查异常（checked exceptions）（详见第 71 条），因为它们迫使 API 的用户面对可能没有返回任何值的事实。抛出未检查的异常或返回 null 允许用户忽略这种可能性，从而带来潜在的可怕后果。但是，抛出一个检查异常需要在客户端中添加额外的样板代码。

如果方法返回一个 Optional，则客户端可以选择在方法无法返回值时要采取的操作。 可以指定默认值：

```
// Using an optional to provide a chosen default value
String lastWordInLexicon = max(words).orElse("No words...");
```

或者可以抛出任何适当的异常。注意，我们传递的是异常工厂，而不是实际的异常。这避免了创建异常的开销，除非它真的实际被抛出：

```
// Using an optional to throw a chosen exception
Toy myToy = max(toys).orElseThrow(TemperTantrumException::new);
```

如果你能证明 Optional 非空，你可以从 Optional 获取值，而不需要指定一个操作来执行。但是如果 Optional 是空的，你判断错了，代码会抛出一个 `NoSuchElementException` 异常：

```
// Using optional when you know there’s a return value
Element lastNobleGas = max(Elements.NOBLE_GASES).get();
```

有时候，可能会遇到这样一种情况：获取默认值的代价很高，除非必要，否则希望避免这种代价。对于这些情况，Optional 提供了一个方法，该方法接受 `Supplier<T>`，并仅在必要时调用它。这个方法被称为 `orElseGet`，但是或许应该被称为 `orElseCompute`，因为它与以 `compute` 开头的三个 Map 方法密切相关。有几个 Optional 的方法来处理更特殊的用例：`filter`、`map`、`flatMap` 和 `ifPresent`。在 Java 9 中，又添加了两个这样的方法：`or` 和 `ifPresentOrElse`。如果上面描述的基本方法与你的用例不太匹配，请查看这些更高级方法的文档，并查看它们是否能够完成任务。

如果这些方法都不能满足你的需要，Optional 提供 `isPresent()` 方法，可以将其视为安全阀。如果 Optional 包含值，则返回 true；如果为空，则返回 false。你可以使用此方法对可选结果执行任何喜欢的处理，但请确保明智地使用它。`isPresent` 的许多用途都可以被上面提到的一种方法所替代。生成的代码通常更短、更清晰、更符合习惯。

例如，请考虑此代码段，它打印一个进程的父进程 ID，如果进程没有父进程，则打印 N/A. 该代码段使用 Java 9 中引入的 `ProcessHandle` 类：

```
Optional<ProcessHandle> parentProcess = ph.parent();
System.out.println("Parent PID: " + (parentProcess.isPresent() ?
    String.valueOf(parentProcess.get().pid()) : "N/A"));
```

上面的代码可以被如下代码所替代，使用了 Optional 的 `map` 方法：

```
System.out.println("Parent PID: " +
  ph.parent().map(h -> String.valueOf(h.pid())).orElse("N/A"));
```

当使用 Stream 进行编程时，通常会发现使用的是一个 `Stream<Optional<T>>`，并且需要一个 `Stream<T>`，其中包含非 Optional 中的所有元素，以便继续进行。如果你正在使用 Java 8，下面是弥补这个差距的代码:

```
streamOfOptionals
    .filter(Optional::isPresent)
    .map(Optional::get)
```

在 Java 9 中，Optional 配备了一个 `stream()` 方法。这个方法是一个适配器， 此方法是一个适配器，它将 Optional 变为包含一个元素的 Stream，如果 Optional 为空，则不包含任何元素。此方法与 Stream 的 `flatMap` 方法 (条目 45) 相结合，这个方法可以简洁地替代上面的方法：

```
streamOfOptionals.
    .flatMap(Optional::stream)
```

并不是所有的返回类型都能从 Optional 的处理中获益。**容器类型，包括集合、映射、Stream、数组和 Optional，不应该封装在 Optional 中。**与其返回一个空的 `Optional<List<T>>`，不还如返回一个空的 `List<T>`（详见第 54 条）。返回空容器将消除客户端代码处理 Optional 的需要。`ProcessHandle` 类确实有 `arguments` 方法，它返回 `Optional<String[]>`，但是这个方法应该被视为一种异常，不该被效仿。

那么什么时候应该声明一个方法来返回 `Optional<T>` 而不是 `T` 呢？ 通常，**如果可能无法返回结果，并且在没有返回结果，客户端还必须执行特殊处理的情况下，则应声明返回 Optional <T> 的方法。**也就是说，返回 `Optional<T>` 并非没有成本。 Optional 是必须分配和初始化的对象，从 Optional 中读取值需要额外的迂回。 这使得 Optional 不适合在某些性能关键的情况下使用。 特定方法是否属于此类别只能通过仔细测量来确定（详见第 67 条）。

与返回装箱的基本类型相比，返回包含已装箱基本类型的 Optional 的代价高得惊人，因为 Optional 有两个装箱级别，而不是零。因此，类库设计人员认为为基本类型 int、long 和 double 提供类似 `Option<T>` 是合适的。这些 Option 是 `OptionalInt`、`OptionalLong` 和 `OptionalDouble`。它们包含 `Optional<T>` 上的大多数方法，但不是所有方法。因此，除了「次要基本类型（minor primitive types）」Boolean，Byte，Character，Short 和 Float 之外，**永远不应该返回装箱的基本类型的 Optional。**

到目前为止，我们已经讨论了返回 Optional 并在返回后处理它们的方法。我们还没有讨论其他可能的用法，这是因为大多数其他 Optional 的用法都是可疑的。例如，永远不要将 Optional 用作映射值。如果这样做，则有两种方法可以表示键（key）在映射中逻辑上的缺失：键要么不在映射中，要么存在的话映射到一个空的 Optional。这反映了不必要的复杂性，很有可能导致混淆和错误。更通俗地说，在集合或数组中使用 Optional 的键、值或元素几乎都是不合适的。

这里留下了一个悬而未决的大问题。在实例中存储 Optional 属性是否合适吗？通常这是一种“不好的味道”：它建议你可能应该有一个包含 Optional 属性的子类。但有时这可能是合理的。考虑条目 2 中的 `NutritionFacts` 类的情况。 `NutritionFacts` 实例包含许多不需要的属性。不可能为这些属性的每个可能组合都提供一个子类。此外，属性包含基本类型，这使得很难直接表示这种缺失。对于 `NutritionFacts` 最好的 API 将为每个 Optional 属性从 getter 方法返回一个 Optional，因此将这些 Optional 作为属性存储在对象中是很有意义的。

总之，如果发现自己编写的方法不能总是返回值，并且认为该方法的用户在每次调用时考虑这种可能性很重要，那么或许应该返回一个 Optional 的方法。但是，应该意识到，返回 Optional 会带来实际的性能后果；对于性能关键的方法，最好返回 null 或抛出异常。最后，除了作为返回值之外，不应该在任何其他地方中使用 Optional。

56. 为所有已公开的 API 元素编写文档注释

如果 API 要可用，就必须对其进行文档化。传统上，API 文档是手工生成的，保持文档与代码的同步是一件苦差事。Java 编程环境使用 `Javadoc` 实用程序简化了这一任务。 `Javadoc` 使用特殊格式的文档注释 (通常称为 doc 注释)，从源代码自动生成 API 文档。

虽然文档注释约定不是 Java 语言的正式一部分，但它们构成了每个 Java 程序员都应该知道的事实上的 API。「如何编写文档注释（How to Write Doc Comments）」的网页[Javadoc-guide] 中介绍了这些约定。 虽然自 Java 4 发布以来该页面尚未更新，但它仍然是一个非常宝贵的资源。 Java 9 中添加了一个重要的文档标签， `{@ index}` ； Java 8 中有一个， `{@implSpec}` ；Java 5 中有两个， `{@literal}` 和 `{@code}` 。 上述网页中缺少这些标签的介绍，但在此条目中进行讨论。

要正确地记录 API，必须在每个导出的类、接口、构造方法、方法和属性声明之前加上文档注释。如果一个类是可序列化的，还应该记录它的序列化形式 （详见第 87 条）。在没有文档注释的情况下，Javadoc 可以做的最好的事情是将声明重现为受影响的 API 元素的唯一文档。使用缺少文档注释的 API 是令人沮丧和容易出错的。公共类不应该使用默认构造方法，因为无法为它们提供文档注释。要编写可维护的代码，还应该为大多数未导出的类、接口、构造方法、方法和属性编写文档注释，尽管这些注释不需要像导出 API 元素那样完整。

方法的文档注释应该简洁地描述方法与其客户端之间的契约。除了为继承而设计的类中的方法 （详见第 19 条）之外，契约应该说明方法做什么，而不是它如何工作的。文档注释应该列举方法的所有前置条件 (这些条件必须为真，以便客户端调用它们)，以及后置条件（这些条件是在调用成功完成后才为真）。通常，对于未检查的异常，前置条件由 `@throw` 标签隐式地描述；每个未检查异常对应于一个先决条件违反（ precondition violation）。此外，可以在受影响的参数的 `@param` 标签中指定前置条件。

除了前置条件和后置条件之外，方法还应在文档中记录它的副作用（side effort）。 副作用是系统状态的可观察到的变化，这对于实现后置条件而言显然不是必需的。 例如，如果方法启动后台线程，则文档应记录它。

完整地描述方法的契约，文档注释应该为每个参数都有一个 `@param` 标签，一个 `@return` 标签 (除非方法有 void 返回类型)，以及一个 `@throw` 标签（无论是检查异常还是非检查异常） （详见第 74 条）。如果 `@return` 标签中的文本与方法的描述相同，则可以忽略它，这取决于你所遵循的编码标准。

按照惯例，`@param` 或 `@retur` 标签后面的文本应该是一个名词短语，描述参数或返回值所表示的值。 很少使用算术表达式代替名词短语; 请参阅 `BigInteger` 的示例。 `@throw` 标签后面的文本应该包含单词「if」，后面跟着一个描述抛出异常的条件的子句。按照惯例，`@param` 、`@return` 或 `@throw` 标签后面的短语或子句不以句号结束。以下的文档注释说明了所有这些约定:

```
/**
 * Returns the element at the specified position in this list.
 *
 * <p>This method is <i>not</i> guaranteed to run in constant
 * time. In some implementations it may run in time proportional
 * to the element position.
 *
 * @param  index index of element to return; must be
 *         non-negative and less than the size of this list
 * @return the element at the specified position in this list
 * @throws IndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= this.size()})
 */
E get(int index);
```

请注意在此文档注释（`<p>` 和 `<i>`）中使用 HTML 标记。 Javadoc 实用工具将文档注释转换为 HTML，文档注释中的任意 HTML 元素最终都会生成 HTML 文档。 有时候，程序员甚至会在他们的文档注释中嵌入 HTML 表格，尽管这种情况很少见。

还要注意在 `@throw` 子句中的代码片段周围使用 Javadoc 的 `{@code}` 标签。这个标签有两个目的: 它使代码片段以代码字体形式呈现，并且它抑制了代码片段中 HTML 标记和嵌套 Javadoc 标记的处理。后一个属性允许我们在代码片段中使用小于号（<），即使它是一个 HTML 元字符。要在文档注释中包含多行代码示例，请使用包装在 HTML `<pre>` 标记中的 Javadoc `{@code}` 标签。换句话说，在代码示例前面加上字符 `<pre>{@code，然后在代码后面加上}</pre>` 。这保留了代码中的换行符，并消除了转义 HTML 元字符的需要，但不需要转义 at 符号（@），如果代码示例使用注释，则必须转义 at 符号（@）。

最后，请注意文档注释中使用的单词「this list」。按照惯例，「this」指的是在实例方法的文档注释中，指向方法调用所在的对象。

正如条目 15 中提到的，当你为继承设计一个类时，必须记录它的自用模式（self-use patterns），以便程序员知道重写它的方法的语义。这些自用模式应该使用在 Java 8 中添加的 `@implSpec` 标签来文档记录。回想一下，普通的问问昂注释描述了方法与其客户端之间的契约；相反，`@implSpec` 注释描述了方法与其子类之间的契约，如果它继承了方法或通过 super 调用方法，那么允许子类依赖于实现行为。下面是实际应用中的实例：

```
 /**
 * Returns true if this collection is empty.
 *
 * @implSpec
 * This implementation returns {@code this.size() == 0}.
 *
 * @return true if this collection is empty
 */
public boolean isEmpty() { ... }
```

从 Java 9 开始，Javadoc 实用工具仍然忽略 @implSpec 标签，除非通过命令行开关：`-tag "implSpec:a:Implementation Requirements:"`。希望在后续的版本中可以修正这个错误。

不要忘记，你必须采取特殊操作来生成包含 HTML 元字符的文档，例如小于号（<），大于号（>）和 and 符号（&）。将这些字符放入文档的最佳方法是使用 `{@literal}` 标签将它们包围起来，该标签禁止处理 HTML 标记和嵌套的 Javadoc 标记。它就像 `{@code}` 标签一样，除了不会以代码字体呈现文本以外。例如，这个 Javadoc 片段：

```
* A geometric series converges if {@literal |r| < 1}.
```

它会生成文档：「A geometric series converges if |r| < 1.」。`{@literal}` 标签可能只放在小于号的位置，而不是整个不等式，并且生成的文档是一样的，但是文档注释在源代码中的可读性较差。这说明了**文档注释在源代码和生成的文档中都应该是可读的通用原则。**如果无法实现这两者，则生成的文档的可读性要胜过在源代码中的可读性。

每个文档注释的第一个「句子」（如下定义）成为注释所在元素的概要描述。例如，第 255 页上的文档注释中的概要描述为：「返回此列表中指定位置的元素」。概要描述必须独立描述其概述元素的功能。为避免混淆，**类或接口中的两个成员或构造方法不应具有相同的概要描述。**要特别注意重载方法，为此通常使用相同的第一句话是自然的（但在文档注释中是不可接受的）。

请小心，如果预期的概要描述包含句点，因为句点可能会提前终止描述。例如，以「A college degree, such as B.S., M.S. or Ph.D.」会导致概要描述为「A college degree, such as B.S., M.S」。问题在于概要描述在第一个句点结束，然后是空格、制表符或行结束符（或第一个块标签处）[Javadoc-ref]。这里是缩写「M.S.」中的第二个句号后面跟着一个空格。最好的解决方案是用 `{@literal}` 标签来包围不愉快的句点和任何相关的文本，这样源代码中的句点后面就不会有空格了：

```
/**
 * A college degree, such as B.S., {@literal M.S.} or Ph.D.
 */
public class Degree { ... }
```

说概要描述是文档注释中的第一句子，其实有点误导人。按照惯例，它很少应该是一个完整的句子。对于方法和构造方法，概要描述应该是一个动词短语（包括任何对象），描述了该方法执行的操作。例如：

- `ArrayList(int initialCapacity)` —— 构造具有指定初始容量的空列表。
- `Collection.size()` —— 返回此集合中的元素个数。

如这些例子所示，使用第三人称陈述句时态 （“returns the number”）而不是第二人称祈使句（“return the number”）。

对于类，接口和属性，概要描述应该是描述由类或接口的实例或属性本身表示的事物的名词短语。例如：

- `Instant` —— 时间线上的瞬时点。
- `Math.PI` —— 更加接近 pi 的 double 类型数值，即圆的周长与其直径之比。

在 Java 9 中，客户端索引被添加到 Javadoc 生成的 HTML 中。这个索引以页面右上角的搜索框的形式出现，它简化了导航大型 API 文档集的任务。当你在框中键入时，得到一个匹配页面的下拉菜单。API 元素 (如类、方法和属性) 是自动索引的。有时，可能希望索引对你的 API 很重要的其他术语。为此添加了 `{@index}` 标签。对文档注释中出现的术语进行索引，就像将其包装在这个标签中一样简单，如下面的片段所示:

```
* This method complies with the {@index IEEE 754} standard.
```

泛型，枚举和注释需要特别注意文档注释。 **记录泛型类型或方法时，请务必记录所有类型参数**:

```
/**
 * An object that maps keys to values.  A map cannot contain
 * duplicate keys; each key can map to at most one value.
 *
 * (Remainder omitted)
 *
 * @param <K> the type of keys maintained by this map
 * @param <V> the type of mapped values
 */
public interface Map<K, V> { ... }
```

在记录枚举类型时，一定要记录常量，以及类型和任何公共方法。注意，如果文档很短，可以把整个文档注释放在一行:

```
/**
 * An instrument section of a symphony orchestra.
 */
public enum OrchestraSection {
    /** Woodwinds, such as flute, clarinet, and oboe. */
    WOODWIND,

    /** Brass instruments, such as french horn and trumpet. */
    BRASS,

    /** Percussion instruments, such as timpani and cymbals. */
    PERCUSSION,

    /** Stringed instruments, such as violin and cello. */
    STRING;
```

```
}
```

在为注解类型记录文档时，一定要记录任何成员，以及类型本身。用名词短语表示的文档成员，就好像它们是属性一样。对于类型的概要描述，请使用动词短语，它表示当程序元素具有此类型注解的所表示的含义:

```
/**
 * Indicates that the annotated method is a test method that
 * must throw the designated exception to pass.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
     /**
      * The exception that the annotated test method must throw
      * in order to pass. (The test is permitted to throw any
      * subtype of the type described by this class object.)
      */
    Class<? extends Throwable> value();
}
```

包级别文档注释应放在名为 package-info.java 的文件中。 除了这些注释之外，package-info.java 还必须包含一个包声明，并且可以在此声明中包含注解。 同样，如果使用模块化系统（详见第 15 条），则应将模块级别注释放在 module-info.java 文件中。

在文档中经常忽略的 API 的两个方面，分别是线程安全性和可序列化性。**无论类或静态方法是否线程安全，都应该在文档中描述其线程安全级别**，如条目 82 中所述。如果一个类是可序列化的，应该记录它的序列化形式，如条目 87 中所述。

Javadoc 具有「继承（inherit）」方法注释的能力。 如果 API 元素没有文档注释，Javadoc 将搜索最具体的适用文档注释，接口文档优先于超类文档。 搜索算法的详细信息可以在 The Javadoc Reference Guide [Javadoc-ref] 中找到。 还可以使用 `{@inheritDoc}` 标签从超类继承部分文档注释。 这意味着，除其他外，类可以重用它们实现的接口的文档注释，而不是复制这些注释。 该工具有可能减轻维护多组几乎相同的文档注释的负担，但使用起来很棘手并且有一些限制。 详细信息超出了本书的范围。

关于文档注释，应该添加一个警告说明。虽然有必要为所有导出的 API 元素提供文档注释，但这并不总是足够的。对于由多个相互关联的类组成的复杂 API，通常需要用描述 API 总体架构的外部文档来补充文档注释。如果存在这样的文档，相关的类或包文档注释应该包含到外部文档的链接。

Javadoc 会自动检查是否符合此条目中的许多建议。在 Java 7 中，需要命令行开关 `-Xdoclint` 来获得这种行为。在 Java 8 和 Java 9 中，默认情况下启用了此检查。诸如 checkstyle 之类的 IDE 插件会进一步检查是否符合这些建议[Burn01]。还可以通过 HTML 有效性检查器运行 Javadoc 生成的 HTML 文件来降低文档注释中出现错误的可能性。可以检测 HTML 标记的许多错误用法。有几个这样的检查器可供下载，可以使用 W3C markup validation service 在线验证 HTML 格式。在验证生成的 HTML 时，请记住，从 Java 9 开始，Javadoc 就能够生成 HTML5 和 HTML 4.01，尽管默认情况下仍然生成 HTML 4.01。如果希望 Javadoc 生成 HTML5，请使用 `-html5` 命令行开关。

本条目中描述的约定涵盖了基本内容。尽管撰写本文时已经有 15 年的历史，但编写文档注释的最终指南仍然是《How to Write Doc Comments》[Javadoc-guide]。

如果你遵循本项目中的指导原则，生成的文档应该提供对 API 的清晰描述。然而，唯一确定的方法，**是阅读 Javadoc 实用工具生成的 web 页面**。对于其他人将使用的每个 API，都值得这样做。正如测试程序几乎不可避免地会导致对代码的一些更改一样，阅读文档通常也会导致对文档注释的一些少许的修改。

总之，文档注释是记录 API 的最佳、最有效的方法。对于所有导出的 API 元素，它们的使用应被视为必需的。 采用符合标准惯例的一致风格 。请记住，在文档注释中允许任意 HTML，但必须转义 HTML 的元字符。

57. 最小化局部变量的作用域

这条目在性质上类似于条目 15，即"最小化类和成员的可访问性"。通过最小化局部变量的作用域，可以提高代码的可读性和可维护性，并降低出错的可能性。

较早的编程语言（如 C）要求必须在代码块的头部声明局部变量，并且一些程序员继续习惯这样做。 这是一个值得改进的习惯。 作为提醒，Java 允许你在任何合法的语句的地方声明变量（as does C, since C99）。

用于最小化局部变量作用域的最强大的技术是再首次使用的地方声明它。 如果变量在使用之前被声明，那就变得更加混乱—— 这也会对试图理解程序的读者来讲，又增加了一件分散他们注意力的事情。到使用该变量时，读者可能不记得变量的类型或初始值。

过早地声明局部变量可能导致其作用域不仅过早开始而且结束太晚。 局部变量的作用域从声明它的位置延伸到封闭块的末尾。 如果变量在使用它的封闭块之外声明，则在程序退出该封闭块后它仍然可见。如果在其预定用途区域之前或之后意外使用变量，则后果可能是灾难性的。

几乎每个局部变量声明都应该包含一个初始化器。如果还没有足够的信息来合理地初始化一个变量，那么应该推迟声明，直到认为可以这样做。这个规则的一个例外是 try-catch 语句。如果一个变量被初始化为一个表达式，该表达式的计算结果可以抛出一个已检查的异常，那么该变量必须在 try 块中初始化（除非所包含的方法可以传播异常）。如果该值必须在 try 块之外使用，那么它必须在 try 块之前声明，此时它还不能被「合理地初始化」。例如，参照条目 65 中的示例。

循环提供了一个特殊的机会来最小化变量的作用域。传统形式的 for 循环和 for-each 形式都允许声明循环变量，将其作用域限制在需要它们的确切区域。 （该区域由循环体和 for 关键字与正文之间的括号中的代码组成）。因此，如果循环终止后不需要循环变量的内容，那么**优先选择 for 循环而不是 while 循环。**

例如，下面是遍历集合的首选方式（详见第 58 条）：

```
// Preferred idiom for iterating over a collection or array
for (Element e : c) {
    ... // Do Something with e
}
```

如果需要访问迭代器，也许是为了调用它的 remove 方法，首选的习惯用法，使用传统的 for 循环代替 for-each 循环：

```
// Idiom for iterating when you need the iterator
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    Element e = i.next();
    ... // Do something with e and i
}
```

要了解为什么这些 for 循环优于 while 循环，请考虑以下代码片段，其中包含两个 while 循环和一个 bug：

```
Iterator<Element> i = c.iterator();
while (i.hasNext()) {
    doSomething(i.next());
}
...
Iterator<Element> i2 = c2.iterator();
while (i.hasNext()) {             // BUG!
    doSomethingElse(i2.next());
}
```

第二个循环包含一个复制粘贴错误：它初始化一个新的循环变量 i2，但是使用旧的变量 i，不幸的是，它仍在范围内。 生成的代码编译时没有错误，并且在不抛出异常的情况下运行，但它做错了。 第二个循环不是在 c2 上迭代，而是立即终止，给出了 c2 为空的错误印象。 由于程序无声地出错，因此错误可能会长时间无法被检测到。

如果将类似的复制粘贴错误与 for 循环（for-each 循环或传统循环）结合使用，则生成的代码甚至无法编译。第一个循环中的元素（或迭代器）变量不在第二个循环中的作用域中。下面是它与传统 for 循环的示例:

```
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    Element e = i.next();
    ... // Do something with e and i
}
...

// Compile-time error - cannot find symbol i
for (Iterator<Element> i2 = c2.iterator(); i.hasNext(); ) {
    Element e2 = i2.next();
    ... // Do something with e2 and i2
}
```

此外，如果使用 for 循环，那么发送这种复制粘贴错误的可能性要小得多，因为没有必要在两个循环中使用不同的变量名。 循环是完全独立的，因此重用元素（或迭代器）变量名称没有坏处。 事实上，这样做通常很流行。

for 循环比 while 循环还有一个优点：它更短，增强了可读性。

下面是另一种循环习惯用法，它最小化了局部变量的作用域:

```
for (int i = 0, n = expensiveComputation(); i < n; i++) {
    ... // Do something with i;
}
```

关于这个做法需要注意的重要一点是，它有两个循环变量，i 和 n，它们都具有完全相同的作用域。第二个变量 n 用于存储第一个变量的限定值，从而避免了每次迭代中冗余计算的代价。作为一个规则，如果循环测试涉及一个方法调用，并且保证在每次迭代中返回相同的结果，那么应该使用这种用法。

最小化局部变量作用域的最终技术是**保持方法小而集中。** 如果在同一方法中组合两个行为（activities），则与一个行为相关的局部变量可能会位于执行另一个行为的代码范围内。 为了防止这种情况发生，只需将方法分为两个：每个行为对应一个方法。

58. for-each 循环优于传统 for 循环

正如在条目 45 中所讨论的，一些任务最好使用 Stream 来完成，一些任务最好使用迭代。下面是一个传统的 for 循环来遍历一个集合:

```
// Not the best way to iterate over a collection!
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    Element e = i.next();
    ... // Do something with e
}
```

下面是迭代数组的传统 for 循环的实例：

```
// Not the best way to iterate over an array!
for (int i = 0; i < a.length; i++) {
    ... // Do something with a[i]
}
```

这些习惯用法比 while 循环更好（详见第 57 条），但是它们并不完美。迭代器和索引变量都很混乱——你只需要元素而已。此外，它们也代表了出错的机会。迭代器在每个循环中出现三次，索引变量出现四次，这使你有很多机会使用错误的变量。如果这样做，就不能保证编译器会发现到问题。最后，这两个循环非常不同，引起了对容器类型的不必要注意，并且增加了更改该类型的小麻烦。

for-each 循环（官方称为「增强的 for 语句」）解决了所有这些问题。它通过隐藏迭代器或索引变量来消除混乱和出错的机会。由此产生的习惯用法同样适用于集合和数组，从而简化了将容器的实现类型从一种转换为另一种的过程:

```
// The preferred idiom for iterating over collections and arrays
for (Element e : elements) {
    ... // Do something with e
}
```

当看到冒号（:） 时，请将其读作「in」。因此，上面的循环读作「对于元素 elements 中的每个元素 e」。使用 for-each 循环不会降低性能，即使对于数组也是如此：它们生成的代码本质上与手工编写的代码相同。

当涉及到嵌套迭代时，for-each 循环相对于传统 for 循环的优势甚至更大。下面是人们在进行嵌套迭代时经常犯的一个错误:

```
// Can you spot the bug?
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
            NINE, TEN, JACK, QUEEN, KING }
...
static Collection<Suit> suits = Arrays.asList(Suit.values());
static Collection<Rank> ranks = Arrays.asList(Rank.values());

List<Card> deck = new ArrayList<>();
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        deck.add(new Card(i.next(), j.next()));
```

如果没有发现这个 bug，也不必感到难过。许多专业程序员都曾犯过这样或那样的错误。问题是，对于外部集合（suit），next 方法在迭代器上调用了太多次。它应该从外部循环调用，因此每花色调用一次，但它是从内部循环调用的，因此每一张牌调用一次。在 suit 用完之后，循环抛出 `NoSuchElementException` 异常。

如果你真的不走运，外部集合的大小是内部集合大小的倍数——也许它们是相同的集合——循环将正常终止，但它不会做你想要的。 例如，考虑这种错误的尝试，打印一对骰子的所有可能的掷法：

```
// Same bug, different symptom!
enum Face { ONE, TWO, THREE, FOUR, FIVE, SIX }
...
Collection<Face> faces = EnumSet.allOf(Face.class);

for (Iterator<Face> i = faces.iterator(); i.hasNext(); )
    for (Iterator<Face> j = faces.iterator(); j.hasNext(); )
        System.out.println(i.next() + " " + j.next());
```

该程序不会抛出异常，但它只打印 6 个重复的组合（从“ONE ONE”到“SIX SIX”），而不是预期的 36 个组合。

要修复例子中的错误，必须在外部循环的作用域内添加一个变量来保存外部元素:

```
/ Fixed, but ugly - you can do better!
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); ) {
    Suit suit = i.next();
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        deck.add(new Card(suit, j.next()));
}
```

相反，如果使用嵌套 for-each 循环，问题就会消失。生成的代码也尽可能地简洁:

```
// Preferred idiom for nested iteration on collections and arrays
for (Suit suit : suits)
    for (Rank rank : ranks)
        deck.add(new Card(suit, rank));
```

但是，有三种常见的情况是你不能分别使用 for-each 循环的:

- **有损过滤（Destructive filtering）**——如果需要遍历集合，并删除指定选元素，则需要使用显式迭代器，以便可以调用其 remove 方法。 通常可以使用在 Java 8 中添加的 Collection 类中的 removeIf 方法，来避免显式遍历。
- **转换**——如果需要遍历一个列表或数组并替换其元素的部分或全部值，那么需要列表迭代器或数组索引来替换元素的值。
- **并行迭代**——如果需要并行地遍历多个集合，那么需要显式地控制迭代器或索引变量，以便所有迭代器或索引变量都可以同步进行 (正如上面错误的 card 和 dice 示例中无意中演示的那样)。

如果发现自己处于这些情况中的任何一种，请使用传统的 for 循环，并警惕本条目中提到的陷阱。

for-each 循环不仅允许遍历集合和数组，还允许遍历实现 Iterable 接口的任何对象，该接口由单个方法组成。接口定义如下:

```
public interface Iterable<E> {
    // Returns an iterator over the elements in this iterable
    Iterator<E> iterator();
}
```

如果必须从头开始编写自己的 Iterator 实现，那么实现 Iterable 会有点棘手，但是如果你正在编写表示一组元素的类型，那么你应该强烈考虑让它实现 Iterable 接口，甚至可以选择不让它实现 Collection 接口。这允许用户使用 for-each 循环遍历类型，他们会永远感激不尽的。

总之，for-each 循环在清晰度，灵活性和错误预防方面提供了超越传统 for 循环的令人注目的优势，而且没有性能损失。 尽可能使用 for-each 循环优先于 for 循环。

59. 了解并使用库

假设你想要生成 0 到某个上界之间的随机整数。面对这个常见任务，许多程序员会编写一个类似这样的小方法:

```
// Common but deeply flawed!
static Random rnd = new Random();
static int random(int n) {
    return Math.abs(rnd.nextInt()) % n;
}
```

这个方法看起来不错，但它有三个缺点。首先，如果 n 是小的平方数，随机数序列会在相当短的时间内重复。第二个缺陷是，如果 n 不是 2 的幂，那么平均而言，一些数字将比其他数字更频繁地返回。如果 n 很大，这种效果会很明显。下面的程序有力地证明了这一点，它在一个精心选择的范围内生成 100 万个随机数，然后打印出有多少个数字落在范围的下半部分：

```
public static void main(String[] args) {
    int n = 2 * (Integer.MAX_VALUE / 3);
    int low = 0;
    for (int i = 0; i < 1000000; i++)
        if (random(n) < n/2)
    low++;
    System.out.println(low);
}
```

如果 random 方法工作正常，程序将输出一个接近 50 万的数字，但是如果运行它，你将发现它输出一个接近 666666 的数字。随机方法生成的数字中有三分之二落在其范围的下半部分！

random 方法的第三个缺陷是，在极少数情况下会返回超出指定范围的数字，这是灾难性的结果。这是因为该方法试图通过调用 `Math.abs` 将 `rnd.nextInt()` 返回的值映射到非负整数。如果 `nextInt()` 返回整数。 `Integer.MIN_VALUE` 、 `Math.abs` 也将返回整数。假设 n 不是 2 的幂，那么 `Integer.MIN_VALUE` 和求模运算符 `(%)` 将返回一个负数。几乎肯定的是，这会导致你的程序失败，并且这种失败可能难以重现。

要编写一个 random 方法来纠正这些缺陷，你必须对伪随机数生成器、数论和 2 的补码算法有一定的了解。幸运的是，你不必这样做（这是为你而做的成果）。它被称为 `Random.nextInt(int)` 。你不必关心它如何工作的（尽管如果你感兴趣，可以研究文档或源代码）。一位具有算法背景的高级工程师花了大量时间设计、实现和测试这种方法，然后将其展示给该领域的几位专家，以确保它是正确的。然后，这个库经过 beta 测试、发布，并被数百万程序员广泛使用了近 20 年。该方法还没有发现任何缺陷，但是如果发现了缺陷，将在下一个版本中进行修复。**通过使用标准库，你可以利用编写它的专家的知识和以前使用它的人的经验。**

从 Java 7 开始，就不应该再使用 Random。在大多数情况下，**选择的随机数生成器现在是 ThreadLocalRandom。** 它能产生更高质量的随机数，而且速度非常快。在我的机器上，它比 Random 快 3.6 倍。对于 fork 连接池和并行流，使用 SplittableRandom。

使用这些库的第二个好处是，你不必浪费时间为那些与你的工作无关的问题编写专门的解决方案。如果你像大多数程序员一样，那么你宁愿将时间花在应用程序上，而不是底层管道上。

使用标准库的第三个优点是，随着时间的推移，它们的性能会不断提高，而你无需付出任何努力。由于许多人使用它们，而且它们是在行业标准基准中使用的，所以提供这些库的组织有很强的动机使它们运行得更快。多年来，许多 Java 平台库都被重新编写过，有时甚至是反复编写，从而带来了显著的性能改进。使用库的第四个好处是，随着时间的推移，它们往往会获得新功能。如果一个库丢失了一些

东西，开发人员社区会将其公布于众，并且丢失的功能可能会在后续版本中添加。

使用标准库的最后一个好处是，可以将代码放在主干中。这样的代码更容易被开发人员阅读、维护和重用。

考虑到所有这些优点，使用库工具而不选择专门的实现似乎是合乎逻辑的，但许多程序员并不这样做。为什么不呢？也许他们不知道库的存在。**在每个主要版本中，都会向库中添加许多特性，了解这些新增特性是值得的。** 每次发布 Java 平台的主要版本时，都会发布一个描述其新特性的 web 页面。这些页面非常值得一读 [Java8-feat, Java9-feat]。为了强调这一点，假设你想编写一个程序来打印命令行中指定的 URL 的内容（这大致是 Linux curl 命令所做的）。在 Java 9 之前，这段代码有点乏味，但是在 Java 9 中，transferTo 方法被添加到 InputStream 中。这是一个使用这个新方法执行这项任务的完整程序：

```
// Printing the contents of a URL with transferTo, added in Java 9
public static void main(String[] args) throws IOException {
    try (InputStream in = new URL(args[0]).openStream()) {
        in.transferTo(System.out);
    }
}
```

这些标准类库太庞大了，以致于不可能学完所有的文档 [Java9-api]，但是 **每个程序员都应该熟悉 java.lang、java.util 和 java.io 的基础知识及其子包。** 其他库的知识可以根据需要获得。概述库中的工具超出了本条目的范围，这些工具多年来已经发展得非常庞大。

其中有几个库值得一提。Collections 框架和 Streams 库（详见第 45 到 48 条）应该是每个程序员的基本工具包的一部分，`java.util.concurrent` 中的并发实用程序也应该是其中的一部分。这个包既包含高级的并发工具来简化多线程的编程任务，还包含低级别的并发基本类型，允许专家们自己编写更高级的并发抽象。`java.util.concurrent` 的高级部分，在第 80 条和第 81 条中讨论。

有时，类库工具可能无法满足你的需求。你的需求越特殊，发生这种情况的可能性就越大。虽然你的第一个思路应该是使用这些库，但是如果你已经了解了它们在某些领域提供的功能，而这些功能不能满足你的需求，那么可以使用另一种实现。任何有限的库集所提供的功能总是存在漏洞。如果你在 Java 平台库中找不到你需要的东西，你的下一个选择应该是寻找高质量的第三方库，比如谷歌的优秀的开源 Guava 库 [Guava]。如果你无法在任何适当的库中找到所需的功能，你可能别无选择，只能自己实现它。

总而言之，不要白费力气重新发明轮子。如果你需要做一些看起来相当常见的事情，那么库中可能已经有一个工具可以做你想做的事情。如果有，使用它；如果你不知道，检查一下。一般来说，库代码可能比你自己编写的代码更好，并且随着时间的推移可能会得到改进。这并不反映你作为一个程序员的能力。规模经济决定了库代码得到的关注要远远超过大多数开发人员所能承担的相同功能。

60. 若需要精确答案就应避免使用 float 和 double 类型

float 和 double 类型主要用于科学计算和工程计算。它们执行二进制浮点运算，该算法经过精心设计，能够在很大范围内快速提供精确的近似值。但是，它们不能提供准确的结果，也不应该在需要精确结果的地方使用。**float 和 double 类型特别不适合进行货币计算**，因为不可能将 0.1（或 10 的任意负次幂）精确地表示为 float 或 double。

例如，假设你口袋里有 1.03 美元，你消费了 42 美分。你还剩下多少钱？下面是一个简单的程序片段，试图回答这个问题：

```
System.out.println(1.03 - 0.42);
```

不幸的是，它输出了 0.6100000000000001。这不是一个特例。假设你口袋里有一美元，你买了 9 台洗衣机，每台 10 美分。你能得到多少零钱？

```
System.out.println(1.00 - 9 * 0.10);
```

根据这个程序片段，可以得到 0.0999999999999999998 美元。

你可能认为，只需在打印之前将结果四舍五入就可以解决这个问题，但不幸的是，这种方法并不总是有效。例如，假设你口袋里有一美元，你看到一个架子上有一排好吃的糖果，它们的价格仅仅是 10 美分，20 美分，30 美分，以此类推，直到 1 美元。你每买一颗糖，从 10 美分的那颗开始，直到你买不起货架上的下一颗糖。你买了多少糖果，换了多少零钱？这里有一个简单的程序来解决这个问题：

```
// Broken - uses floating point for monetary calculation!
public static void main(String[] args) {
    double funds = 1.00;
    int itemsBought = 0;
    for (double price = 0.10; funds >= price; price += 0.10) {
        funds -= price;
        itemsBought++;
    }
    System.out.println(itemsBought +"items bought.");
    System.out.println("Change: $" + funds);
}
```

如果你运行这个程序，你会发现你可以买得起三块糖，你还有 0.399999999999999999 美元。这是错误的答案！解决这个问题的正确方法是 **使用 BigDecimal、int 或 long 进行货币计算。**

这里是前一个程序的一个简单改版，使用 BigDecimal 类型代替 double。注意，使用 BigDecimal 的 String 构造函数而不是它的 double 构造函数。这是为了避免在计算中引入不准确的值 [Bloch05, Puzzle 2]：

```
public static void main(String[] args) {
    final BigDecimal TEN_CENTS = new BigDecimal(".10");
    int itemsBought = 0;
    BigDecimal funds = new BigDecimal("1.00");
    for (BigDecimal price = TEN_CENTS;funds.compareTo(price) >= 0;price =
price.add(TEN_CENTS)) {
        funds = funds.subtract(price);
        itemsBought++;
    }
    System.out.println(itemsBought +"items bought.");
    System.out.println("Money left over: $" + funds);
}
```

如果你运行修改后的程序，你会发现你可以买四颗糖，最终剩下 0 美元。这是正确答案。

然而，使用 BigDecimal 有两个缺点：它与原始算术类型相比很不方便，而且速度要慢得多。如果你只解决一个简单的问题，后一种缺点是无关紧要的，但前者可能会让你烦恼。

除了使用 BigDecimal，另一种方法是使用 int 或 long，这取决于涉及的数值大小，还要自己处理十进制小数点。在这个例子中，最明显的方法是用美分而不是美元来计算。下面是一个采用这种方法的简单改版：

```
public static void main(String[] args) {
    int itemsBought = 0;
    int funds = 100;
    for (int price = 10; funds >= price; price += 10) {
        funds -= price;
        itemsBought++;
    }
    System.out.println(itemsBought +"items bought.");
    System.out.println("Cash left over: " + funds + " cents");
}
```

总之，对于任何需要精确答案的计算，不要使用 float 或 double 类型。如果希望系统来处理十进制小数点，并且不介意不使用基本类型带来的不便和成本，请使用 BigDecimal。使用 BigDecimal 的另一个好处是，它可以完全控制舍入，当执行需要舍入的操作时，可以从八种舍入模式中进行选择。如果你使用合法的舍入行为执行业务计算，这将非常方便。如果性能是最重要的，那么你不介意自己处理十进制小数点，而且数值不是太大，可以使用 int 或 long。如果数值不超过 9 位小数，可以使用 int；如果不超过 18 位，可以使用 long。如果数量可能超过 18 位，则使用 BigDecimal。

61. 基本数据类型优于包装类

Java 有一个由两部分组成的类型系统，包括基本类型（如 int、double 和 boolean）和引用类型（如 String 和 List）。每个基本类型都有一个对应的引用类型，称为包装类型。与 int、double 和 boolean 对应的包装类是 Integer、Double 和 Boolean。

正如条目 6 中提到的，自动装箱和自动拆箱模糊了基本类型和包装类型之间的区别，但不会消除它们。这两者之间有真正的区别，重要的是你要始终意识到正在使用的是哪一种，并在它们之间仔细选择。

基本类型和包装类型之间有三个主要区别。首先，基本类型只有它们的值，而包装类型具有与其值不同的标识。换句话说，两个包装类型实例可以具有相同的值和不同的标识。第二，基本类型只有全功能值，而每个包装类型除了对应的基本类型的所有功能值外，还有一个非功能值，即 null。最后，基本类型比包装类型更节省时间和空间。如果你不小心的话，这三种差异都会给你带来真正的麻烦。

考虑下面的比较器，它的设计目的是表示 Integer 值上的升序数字排序。（回想一下，比较器的 compare 方法返回一个负数、零或正数，这取决于它的第一个参数是小于、等于还是大于第二个参数。）你不需要在实际使用中编写这个比较器，因为它实现了 Integer 的自然排序，但它提供了一个有趣的例子：

```
// Broken comparator - can you spot the flaw?
Comparator<Integer> naturalOrder =(i, j) -> (i < j) ? -1 : (i == j ? 0 : 1);
```

这个比较器看起来应该可以工作，它将通过许多测试。例如，它可以与 `Collections.sort` 一起使用，以正确地排序一个百万元素的 List，无论该 List 是否包含重复的元素。但这个比较存在严重缺陷。要使自己相信这一点，只需打印 `naturalOrder.compare(new Integer(42), new Integer(42))` 的值。两个 Integer 实例都表示相同的值 `(42)`，所以这个表达式的值应该是 0，但它是 1，这表明第一个 Integer 值大于第二个！

那么问题出在哪里呢？naturalOrder 中的第一个测试工作得很好。计算表达式 `i < j` 会使 i 和 j 引用的 Integer 实例自动拆箱；也就是说，它提取它们的基本类型值。计算的目的是检查得到的第一个 int 值是否小于第二个 int 值。但假设它不是。然后，下一个测试计算表达式 `i==j`，该表达式对两个对象引用执行标识比较。如果 i 和 j 引用表示相同 int 值的不同 Integer 实例，这个比较将返回 false，比较器将错误地返回 1，表明第一个整型值大于第二个整型值。**将 == 操作符应用于包装类型几乎都是错误的。**

在实际使用中，如果你需要一个比较器来描述类型的自然顺序，你应该简单地调用 `Comparator.naturalOrder()`，如果你自己编写一个比较器，你应该使用比较器构造方法，或者对基本类型使用静态比较方法（详见第 14 条）。也就是说，你可以通过添加两个局部变量来存储基本类型 int 值，并对这些变量执行所有的比较，从而修复损坏的比较器中的问题。这避免了错误的标识比较：

```
Comparator<Integer> naturalOrder = (iBoxed, jBoxed) -> {
    int i = iBoxed, j = jBoxed; // Auto-unboxing
    return i < j ? -1 : (i == j ? 0 : 1);
};
```

接下来，考虑一下这个有趣的小程序：

```
public class Unbelievable {
static Integer i;
public static void main(String[] args) {
    if (i == 42)
        System.out.println("Unbelievable");
    }
}
```

不，它不会打印出令人难以置信的东西，但它的行为很奇怪。它在计算表达式 `i==42` 时抛出 `NullPointerException`。问题是，i 是 Integer，而不是 int 数，而且像所有非常量对象引用字段一样，它的初值为 null。当程序计算表达式 `i==42` 时，它是在比较 Integer 与 int。**在操作中混合使用基本类型和包装类型时，包装类型就会自动拆箱**，这种情况无一例外。如果一个空对象引用自动拆箱，那么你将得到一个 `NullPointerException`。正如这个程序所演示的，它几乎可以在任何地方发生。修复这个问题非常简单，只需将 i 声明为 int 而不是 Integer。

最后，考虑条目 6 中第 24 页的程序：

```
// Hideously slow program! Can you spot the object creation?
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
        sum += i;
    }
    System.out.println(sum);
}
```

这个程序比它预期的速度慢得多，因为它意外地声明了一个局部变量 `(sum)`，它是包装类型 Long，而不是基本类型 long。程序在没有错误或警告的情况下编译，变量被反复装箱和拆箱，导致产生明显的性能下降。

在本条目中讨论的所有三个程序中，问题都是一样的：程序员忽略了基本类型和包装类型之间的区别，并承担了恶果。在前两个项目中，结果是彻底的失败；第三个例子还产生了严重的性能问题。

那么，什么时候应该使用包装类型呢？它们有几个合法的用途。第一个是作为集合中的元素、键和值。不能将基本类型放在集合中，因此必须使用包装类型。这是一般情况下的特例。在参数化类型和方法（Chapter 5）中，必须使用包装类型作为类型参数，因为 Java 不允许使用基本类型。例如，不能将变量声明为 `ThreadLocal<int>` 类型，因此必须使用 `ThreadLocal<Integer>`。最后，在进行反射方法调用时，必须使用包装类型（详见第 65 条）。

总之，只要有选择，就应该优先使用基本类型，而不是包装类型。基本类型更简单、更快。如果必须使用包装类型，请小心！**自动装箱减少了使用包装类型的冗长，但没有减少危险。** 当你的程序使用 `==` 操作符比较两个包装类型时，它会执行标识比较，这几乎肯定不是你想要的。当你的程序执行包含包装类型和基本类型的混合类型计算时，它将进行拆箱，**当你的程序执行拆箱时，将抛出** `NullPointerException`。 最后，当你的程序将基本类型装箱时，可能会导致代价高昂且不必要的对象创建。

62. 当使用其他类型更合适时应避免使用字符串

字符串被设计用来表示文本，它们在这方面做得很好。因为字符串是如此常见，并且受到 Java 的良好支持，所以很自然地会将字符串用于其他目的，而不是它们适用的场景。本条目讨论了一些不应该使用字符串的场景。

字符串是其他值类型的糟糕替代品。 当一段数据从文件、网络或键盘输入到程序时，它通常是字符串形式的。有一种很自然的倾向是保持这种格式不变，但是这种倾向只有在数据本质上是文本的情况下才合理。如果是数值类型，则应将其转换为适当的数值类型，如 int、float 或 BigInteger。如果是问题的答案，如「是」或「否」这类形式，则应将其转换为适当的枚举类型或布尔值。更一般地说，如果有合适的值类型，无论是基本类型还是对象引用，都应该使用它；如果没有，你应该写一个。虽然这条建议似乎很多余，但经常被违反。

字符串是枚举类型的糟糕替代品。 正如条目 34 中所讨论的，枚举类型常量比字符串更适合于枚举类型常量。

字符串是聚合类型的糟糕替代品。 如果一个实体有多个组件，将其表示为单个字符串通常是一个坏主意。例如，下面这行代码来自一个真实的系统标识符，它的名称已经被更改，以免引发罪责：

```
// Inappropriate use of string as aggregate type
String compoundKey = className + "#" + i.next();
```

这种方法有很多缺点。如果用于分隔字段的字符出现在其中一个字段中，可能会导致混乱。要访问各个字段，你必须解析字符串，这是缓慢的、冗长的、容易出错的过程。你不能提供 equals、toString 或 compareTo 方法，但必须接受 String 提供的行为。更好的方法是编写一个类来表示聚合，通常是一个私有静态成员类（详见第 24 条）。

字符串不能很好地替代 capabilities。 有时，字符串用于授予对某些功能的访问权。例如，考虑线程本地变量机制的设计。这样的机制提供了每个线程都有自己的变量值。自 1.2 版以来，Java 库就有了一个线程本地变量机制，但在此之前，程序员必须自己设计。许多年前，当面临设计这样一个机制的任务时，有人提出了相同的设计，其中客户端提供的字符串键，用于标识每个线程本地变量：

```
// Broken - inappropriate use of string as capability!
public class ThreadLocal {
    private ThreadLocal() { } // Noninstantiable

    // Sets the current thread's value for the named variable.
    public static void set(String key, Object value);

    // Returns the current thread's value for the named variable.
    public static Object get(String key);
}
```

这种方法的问题在于，字符串键表示线程本地变量的共享全局名称空间。为了使这种方法有效，客户端提供的字符串键必须是惟一的：如果两个客户端各自决定为它们的线程本地变量使用相同的名称，它们无意中就会共享一个变量，这通常会导致两个客户端都失败。而且，安全性很差。恶意客户端可以故意使用与另一个客户端相同的字符串密钥来非法访问另一个客户端的数据。

这个 API 可以通过用一个不可伪造的键（有时称为 capability）替换字符串来修复：

```
public class ThreadLocal {
    private ThreadLocal() { } // Noninstantiable

    public static class Key { // (Capability)
        Key() { }
}

// Generates a unique, unforgeable key
public static Key getKey() {
    return new Key();
}

public static void set(Key key, Object value);

public static Object get(Key key);
}
```

虽然这解决了 API 中基于字符串的两个问题，但是你可以做得更好。你不再真正需要静态方法。它们可以变成键上的实例方法，此时键不再是线程局部变量：而是线程局部变量。此时，顶层类不再为你做任何事情，所以你可以删除它，并将嵌套类重命名为 ThreadLocal：

```
public final class ThreadLocal {
    public ThreadLocal();
    public void set(Object value);
    public Object get();
}
```

这个 API 不是类型安全的，因为在从线程本地变量检索值时，必须将值从 Object 转换为它的实际类型。原始的基于 String 类型 API 的类型安全是不可能实现的，基于键的 API 的类型安全也是很难实现的，但是通过将 ThreadLocal 作为一个参数化的类来实现这个 API 的类型安全很简单（详见第 29 条）：

```
public final class ThreadLocal<T> {
    public ThreadLocal();
    public void set(T value);
    public T get();
}
```

粗略地说，这就是 `java.lang.ThreadLocal` 提供的 API，除了解决基于字符串的问题之外，它比任何基于键的 API 都更快、更优雅。

总之，当存在或可以编写更好的数据类型时，应避免将字符串用来表示对象。如果使用不当，字符串比其他类型更麻烦、灵活性更差、速度更慢、更容易出错。字符串经常被误用的类型包括基本类型、枚举和聚合类型。

63. 当心字符串连接引起的性能问题

字符串连接操作符 `(+)` 是将几个字符串组合成一个字符串的简便方法。对于生成单行输出或构造一个小的、固定大小的对象的字符串表示形式，它是可以的，但是它不能伸缩。使用 **字符串串联运算符重复串联 n 个字符串需要 n 的平方级时间。** 这是字符串不可变这一事实导致的结果（详见第 17 条）。当连接两个字符串时，将复制这两个字符串的内容。

例如，考虑这个方法，它通过将每个账单项目重复连接到一行来构造账单语句的字符串表示：

```
// Inappropriate use of string concatenation - Performs poorly!
public String statement() {
    String result = "";
    for (int i = 0; i < numItems(); i++)
        result += lineForItem(i); // String concatenation
    return result;
}
```

如果项的数量很大，则该方法的性能非常糟糕。**要获得能接受的性能，请使用 StringBuilder 代替 String** 来存储正在构建的语句：

```
public String statement() {
    StringBuilder b = new StringBuilder(numItems() * LINE_WIDTH);
    for (int i = 0; i < numItems(); i++)
        b.append(lineForItem(i));
    return b.toString();
}
```

自 Java 6 以来，为了使字符串连接更快，已经做了大量工作，但是这两个方法在性能上的差异仍然很大：如果 numItems 返回 100，lineForItem 返回 80 个字符串，那么第二个方法在我的机器上运行的速度是第一个方法的 6.5 倍。由于第一种方法在项目数量上是平方级的，而第二种方法是线性的，所以随着项目数量的增加，性能差异会变得越来越大。注意，第二个方法预先分配了一个足够大的 StringBuilder 来保存整个结果，从而消除了自动增长的需要。即使使用默认大小的 StringBuilder，它仍然比第一个方法快 5.5 倍。

道理很简单：**不要使用字符串连接操作符合并多个字符串**，除非性能无关紧要。否则使用 StringBuilder 的 append 方法。或者，使用字符数组，再或者一次只处理一个字符串，而不是组合它们。

64. 通过接口引用对象

条目 51 指出，应该使用接口而不是类作为参数类型。更一般地说，你应该优先使用接口而不是类来引用对象。**如果存在合适的接口类型，那么应该使用接口类型声明参数、返回值、变量和字段。** 惟一真正需要引用对象的类的时候是使用构造函数创建它的时候。为了具体说明这一点，考虑 LinkedHashSet 的情况，它是 Set 接口的一个实现。声明时应养成这样的习惯：

```
// Good - uses interface as type
Set<Son> sonSet = new LinkedHashSet<>();
```

而不是这样：

```
// Bad - uses class as type!
LinkedHashSet<Son> sonSet = new LinkedHashSet<>();
```

如果你养成了使用接口作为类型的习惯，那么你的程序将更加灵活。 如果你决定要切换实现，只需在构造函数中更改类名（或使用不同的静态工厂）。例如，第一个声明可以改为：

```
Set<Son> sonSet = new HashSet<>();
```

所有的代码都会继续工作。周围的代码不知道旧的实现类型，所以它不会在意更改。

有一点值得注意：如果原实现提供了接口的通用约定不需要的一些特殊功能，并且代码依赖于该功能，那么新实现提供相同的功能就非常重要。例如，如果围绕第一个声明的代码依赖于 `LinkedHashSet` 的排序策略，那么在声明中将 `HashSet` 替换为 `LinkedHashSet` 将是不正确的，因为 `HashSet` 不保证迭代顺序。

那么，为什么要更改实现类型呢？因为第二个实现比原来的实现提供了更好的性能，或者因为它提供了原来的实现所缺乏的理想功能。例如，假设一个字段包含一个 `HashMap` 实例。将其更改为 `EnumMap` 将为迭代提供更好的性能和与键的自然顺序，但是你只能在键类型为 enum 类型的情况下使用 `EnumMap` 。将 `HashMap` 更改为 `LinkedHashMap` 将提供可预测的迭代顺序，性能与 `HashMap` 相当，而不需要对键类型作出任何特殊要求。

你可能认为使用变量的实现类型声明变量是可以的，因为你可以同时更改声明类型和实现类型，但是不能保证这种更改会正确编译程序。如果客户端代码对原实现类型使用了替换时不存在的方法，或者客户端代码将实例传递给需要原实现类型的方法，那么在进行此更改之后，代码将不再编译。使用接口类型声明变量可以保持一致。

如果没有合适的接口存在，那么用类引用对象是完全合适的。 例如，考虑值类，如 `String` 和 `BigInteger` 。值类很少在编写时考虑到多个实现。它们通常是 final 的，很少有相应的接口。使用这样的值类作为参数、变量、字段或返回类型非常合适。

没有合适接口类型的第二种情况是属于框架的对象，框架的基本类型是类而不是接口。如果一个对象属于这样一个基于类的框架，那么最好使用相关的基类来引用它，这通常是抽象的，而不是使用它的实现类。在 `java.io` 类中许多诸如 `OutputStream` 之类的就属于这种情况。

没有合适接口类型的最后一种情况是，实现接口但同时提供接口中不存在的额外方法的类，例如， `PriorityQueue` 有一个在 `Queue` 接口上不存在的比较器方法。只有当程序依赖于额外的方法时，才应该使用这样的类来引用它的实例，这种情况应该非常少见。

这三种情况并不是面面俱到的，而仅仅是为了传达适合通过类引用对象的情况。在实际应用中，给定对象是否具有适当的接口应该是显而易见的。如果是这样，如果使用接口引用对象，程序将更加灵活和流行。**如果没有合适的接口，就使用类层次结构中提供所需功能的最底层的类**

65. 接口优于反射

核心反射机制 `java.lang.reflect` 提供对任意类的编程访问。给定一个 Class 对象，你可以获得 Constructor、Method 和 Field 实例，分别代表了该 Class 实例所表示的类的构造器、方法和字段。这些对象提供对类的成员名、字段类型、方法签名等的编程访问。

此外，Constructor、Method 和 Field 实例允许你反射性地操作它们的底层对应项：你可以通过调用 Constructor、Method 和 Field 实例上的方法，可以构造底层类的实例、调用底层类的方法，并访问底层类中的字段。例如，Method.invoke 允许你在任何类的任何对象上调用任何方法（受默认的安全约束）。反射允许一个类使用另一个类，即使在编译前者时后者并不存在。然而，这种能力是有代价的：

- **你失去了编译时类型检查的所有好处，** 包括异常检查。如果一个程序试图反射性地调用一个不存在的或不可访问的方法，它将在运行时失败，除非你采取了特殊的预防措施。
- **执行反射访问所需的代码既笨拙又冗长。** 写起来很乏味，读起来也很困难。
- **性能降低。** 反射方法调用比普通方法调用慢得多。到底慢了多少还很难说，因为有很多因素在起作用。在我的机器上，调用一个没有输入参数和返回 int 类型的方法时，用反射执行要慢 11 倍。

有一些复杂的应用程序需要反射。包括代码分析工具和依赖注入框架。即使是这样的工具，随着它的缺点变得越来越明显，人们也在逐渐远离并反思这种用法。如果你对应用程序是否需要反射有任何疑问，那么它可能不需要。

通过非常有限的形式使用反射，你可以获得反射的许多好处，同时花费的代价很少。 对于许多程序，它们必须用到在编译时无法获取的类，在编译时存在一个适当的接口或超类来引用该类（详见第 64 条）。如果是这种情况，**可以用反射方式创建实例，并通过它们的接口或超类正常地访问它们。**

例如，这是一个创建 `Set<String>` 实例的程序，类由第一个命令行参数指定。程序将剩余的命令行参数插入到集合中并打印出来。不管第一个参数是什么，程序都会打印剩余的参数，并去掉重复项。然而，打印这些参数的顺序取决于第一个参数中指定的类。如果你指定 `java.util.HashSet`，它们显然是随机排列的；如果你指定 `java.util.TreeSet`，它们是按字母顺序打印的，因为 TreeSet 中的元素是有序的：

```
// Reflective instantiation with interface access
public static void main(String[] args) {

    // Translate the class name into a Class object
    Class<? extends Set<String>> cl = null;
    try {
        cl = (Class<? extends Set<String>>) // Unchecked cast!
        Class.forName(args[0]);
    } catch (ClassNotFoundException e) {
        fatalError("Class not found.");
    }
```

```
        // Get the constructor
        Constructor<? extends Set<String>> cons = null;
        try {
            cons = cl.getDeclaredConstructor();
        } catch (NoSuchMethodException e) {
            fatalError("No parameterless constructor");
        }

        // Instantiate the set
        Set<String> s = null;
        try {
            s = cons.newInstance();
        } catch (IllegalAccessException e) {
            fatalError("Constructor not accessible");
        } catch (InstantiationException e) {
            fatalError("Class not instantiable.");
        } catch (InvocationTargetException e) {
            fatalError("Constructor threw " + e.getCause());
        } catch (ClassCastException e) {
            fatalError("Class doesn't implement Set");
        }

        // Exercise the set
        s.addAll(Arrays.asList(args).subList(1, args.length));
        System.out.println(s);
    }

    private static void fatalError(String msg) {
        System.err.println(msg);
        System.exit(1);
    }
```

虽然这个程序只是一个小把戏，但它演示的技术非常强大。这个程序可以很容易地转换成一个通用的集合测试器，通过积极地操作一个或多个实例并检查它们是否遵守 `Set` 接口约定来验证指定的 `Set` 实现。类似地，它可以变成一个通用的集合性能分析工具。事实上，该技术足够强大，可以实现一个成熟的服务提供者框架（详见第 1 条）。

这个例子也说明了反射的两个缺点。首先，该示例可以在运行时生成六个不同的异常，如果没有使用反射实例化，所有这些异常都将是编译时错误。（有趣的是，你可以通过传入适当的命令行参数，使程序生成六个异常中的每一个。）第二个缺点是，根据类的名称生成类的实例需要 25 行冗长的代码，而构造函数调用只需要一行。通过捕获 `ReflectiveOperationException`（Java 7 中引入的各种反射异常的超类），可以减少程序的长度。这两个缺点都只限于实例化对象的程序部分。实例化后，与任何其他 Set 实例将难以区分。在实际的程序中，通过这种限定使用反射的方法，大部分代码可以免受影响。

如果编译此程序，将得到 unchecked 的强制转换警告。这个警告是合法的，即使指定的类不是 `Set` 实现，`Class<? extends Set<String>>` 也会成功，在这种情况下，程序在实例化类时抛出 `ClassCastException`。要了解如何抑制警告，请阅读条目 27。

反射的合法用途（很少）是管理类对运行时可能不存在的其他类、方法或字段的依赖关系。如果你正在编写一个包，并且必须针对其他包的多个版本运行，此时反射将非常有用。该技术是根据支持包所需的最小环境（通常是最老的版本）编译包，并反射性地访问任何较新的类或方法。如果你试图访问的新类或方法在运行时不存在，要使此工作正常进行，则必须采取适当的操作。适当的操作可能包括使用一些替代方法来完成相同的目标，或者使用简化的功能进行操作。

总之，反射是一种功能强大的工具，对于某些复杂的系统编程任务是必需的，但是它有很多缺点。如果编写的程序必须在编译时处理未知的类，则应该尽可能只使用反射实例化对象，并使用在编译时已知的接口或超类访问对象。

66. 明智审慎地本地方法

Java 本地接口（JNI）允许 Java 程序调用本地方法，这些方法是用 C 或 C++ 等本地编程语言编写的。从历史上看，本地方法主要有三种用途。它们提供对特定于平台的设施（如注册中心）的访问。它们提供对现有本地代码库的访问，包括提供对遗留数据访问。最后，本地方法可以通过本地语言编写应用程序中注重性能的部分，以提高性能。

使用本地方法访问特定于平台的机制是合法的，但是很少有必要：随着 Java 平台的成熟，它提供了对许多以前只能在宿主平台中上找到的特性。例如，Java 9 中添加的流 API 提供了对 OS 流程的访问。在 Java 中没有等效库时，使用本地方法来使用本地库也是合法的。

使用本地方法来提高性能的行为很少是明智的。 在早期版本（Java 3 之前），这通常是必要的，但是从那时起 JVM 变得更快了。对于大多数任务，现在可以在 Java 中获得类似的性能。例如，在版本 1.1 中添加了 `java.math`，BigInteger 是在一个用 C 编写的快速多精度运算库的基础上实现的。在当时，为了获得足够的性能这样做是必要的。在 Java 3 中，BigInteger 则完全用 Java 重写了，并且进行了性能调优，新的版本比原来的版本更快。

这个故事的一个可悲的结尾是，除了在 Java 8 中对大数进行更快的乘法运算之外，BigInteger 此后几乎没有发生什么变化。在此期间，对本地库的工作继续快速进行，尤其是 GNU 多精度算术库（GMP）。需要真正高性能多精度算法的 Java 程序员现在可以通过本地方法使用 GMP [Blum14]。

使用本地方法有严重的缺点。由于本地语言不安全（详见第 50 条），使用本地方法的应用程序不再能免受内存毁坏错误的影响。由于本地语言比 Java 更依赖于平台，因此使用本地方法的程序的可移植性较差。它们也更难调试。如果不小心，本地方法可能会降低性能，因为垃圾收集器无法自动跟踪本地内存使用情况（详见第 8 条），而且进出本地代码会产生相关的成本。最后，本地方法需要「粘合代码」，这很难阅读，而且编写起来很乏味。

总之，在使用本地方法之前要三思。一般很少需要使用它们来提高性能。如果必须使用本地方法来访问底层资源或本地库，请尽可能少地使用本地代码，并对其进行彻底的测试。本地代码中的一个错误就可以破坏整个应用程序。

67. 明智审慎地进行优化

有三条关于优化的格言是每个人都应该知道的：

> 比起任何其他单一原因（包括盲目愚蠢），计算上的过失更多的是以效率为名（不一定能实现）而犯下的。
>
> —William A. Wulf
>
> [Wulf72]
>
> 不要去计较效率上的一些小小的得失，在 97% 的情况下，不成熟的优化才是一切问题的根源。
>
> —Donald E. Knuth
>
> [Knuth74]
>
> 在优化方面，我们应该遵守两条规则：
>
> 规则 1：不要进行优化。
>
> 规则 2 （仅针对专家）：还是不要进行优化，也就是说，在你还没有绝对清晰的未优化方案之前，请不要进行优化。
>
> —M. A. Jackson
>
> [Jackson75]

所有这些格言都比 Java 编程语言早了 20 年。它们告诉我们关于优化的一个深刻的事实：很容易弊大于利，尤其是如果过早地进行优化。在此过程中，你可能会生成既不快速也不正确且无法轻松修复的软件。

不要为了性能而牺牲合理的架构。努力编写 **好的程序，而不是快速的程序。** 如果一个好的程序不够快，它的架构将允许它被优化。好的程序体现了信息隐藏的原则：在可能的情况下，它们在单个组件中本地化设计决策，因此可以在不影响系统其余部分的情况下更改单个决策（详见第 15 条）。

这并不意味着在程序完成之前可以忽略性能问题。实现上的问题可以通过以后的优化来解决，但是对于架构缺陷，如果不重写系统，就不可能解决限制性能的问题。在系统完成之后再改变设计的某个基本方面可能导致结构不良的系统难以维护和进化。因此，你必须在设计过程中考虑性能。

尽量避免限制性能的设计决策。 设计中最难以更改的组件是那些指定组件之间以及与外部世界的交互的组件。这些设计组件中最主要的是 API、线路层协议和持久数据格式。这些设计组件不仅难以或不可能在事后更改，而且所有这些组件都可能对系统能够达到的性能造成重大限制。

考虑API设计决策的性能结果。 使公共类型转化为可变，可能需要大量不必要的防御性复制（详见第 50 条）。类似地，在一个公共类中使用继承（在这个类中组合将是合适的）将该类永远绑定到它的超类，这会人为地限制子类的性能（详见第 18 条）。最后一个例子是，在 API 中使用实现类而不是接口将你绑定到特定的实现，即使将来可能会编写更快的实现也无法使用（详见第 64 条）。

API 设计对性能的影响是非常实际的。考虑 `java.awt.Component` 中的 getSize 方法。该性能很关键方法返回 Dimension 实例的决定，加上维度实例是可变的决定，强制该方法的任何实现在每次调用时分配一个新的 Dimension 实例。尽管在现代 VM 上分配小对象并不昂贵，但不必要地分配数百万个对象也会对性能造成实际损害。

存在几种 API 设计替代方案。理想情况下，Dimension 应该是不可变的（详见第 17 条）；或者，getSize 可以被返回 Dimension 对象的原始组件的两个方法所替代。事实上，出于性能原因，在 Java 2 的组件中添加了两个这样的方法。然而，现有的客户端代码仍然使用 getSize 方法，并且仍然受到原始 API 设计决策的性能影响。

幸运的是，通常情况下，好的 API 设计与好的性能是一致的。**为了获得良好的性能而改变 API 是一个非常糟糕的想法。** 导致你改变 API 的性能问题，可能在平台或其他底层软件的未来版本中消失，但是改变的 API 和随之而来的问题将永远伴随着你。

一旦你仔细地设计了你的程序，成了一个清晰、简洁、结构良好的实现，那么可能是时候考虑优化了，假设此时你还不满意程序的性能。

记得 Jackson 的两条优化规则是「不要做」和「（只针对专家）」。先别这么做。他本可以再加一个：**在每次尝试优化之前和之后测量性能。** 你可能会对你的发现感到惊讶。通常，试图做的优化通常对于性能并没有明显的影响；有时候，还让事情变得更糟。主要原因是很难猜测程序将时间花费在哪里。程序中你认为很慢的部分可能并没有问题，在这种情况下，你是在浪费时间来优化它。一般认为，程序将 90% 的时间花费在了 10% 的代码上。

分析工具可以帮助你决定将优化工作的重点放在哪里。这些工具提供了运行时信息，比如每个方法大约花费多少时间以及调用了多少次。除了关注你的调优工作之外，这还可以提醒你是否需要改变算法。如果程序中潜伏着平方级（或更差）的算法，那么再多的调优也无法解决这个问题。你必须用一个更有效的算法来代替这个算法。系统中的代码越多，使用分析器就越重要。这就像大海捞针：大海越大，金属探测器就越有用。另一个值得特别提及的工具是 jmh，它不是一个分析器，而是一个微基准测试框架，提供了对 Java 代码性能无与伦比的预测性。

与 C 和 C++ 等更传统的语言相比，Java 甚至更需要度量尝试优化的效果，因为 Java 的性能模型更弱：各种基本操作的相对成本没有得到很好的定义。程序员编写的内容和 CPU 执行的内容之间的「抽象鸿沟」更大，这使得可靠地预测优化的性能结果变得更加困难。有很多关于性能的传说流传开来，但最终被证明是半真半假或彻头彻尾的谎言。

Java 的性能模型不仅定义不清，而且在不同的实现、不同的发布版本、不同的处理器之间都有所不同。如果你要在多个实现或多个硬件平台上运行程序，那么度量优化对每个平台的效果是很重要的。有时候，你可能会被迫在不同实现或硬件平台上的性能之间进行权衡。

自本条目首次编写以来的近 20 年里，Java 软件栈的每个组件都变得越来越复杂，从处理器到 vm 再到库，Java 运行的各种硬件都有了极大的增长。所有这些加在一起，使得 Java 程序的性能比 2001 年更难以预测，而对它进行度量的需求也相应增加。

总而言之，不要努力写快的程序，要努力写好程序；速度自然会提高。但是在设计系统时一定要考虑性能，特别是在设计API、线路层协议和持久数据格式时。当你完成了系统的构建之后，请度量它的性能。如果足够快，就完成了。如果没有，利用分析器找到问题的根源，并对系统的相关部分进行优化。第一步是检查算法的选择：再多的底层优化也不能弥补算法选择的不足。根据需要重复这个过程，在每次更改之后测量性能，直到你满意为止。

68. 遵守被广泛认可的命名约定

Java 平台有一组完善的命名约定，其中许多约定包含在《The Java Language Specification》[JLS, 6.1]。不严格地讲，命名约定分为两类：排版和语法。

有少量的与排版有关的命名约定，包括包、类、接口、方法、字段和类型变量。如果没有很好的理由，你不应该违反它们。如果 API 违反了这些约定，那么它可能很难使用。如果实现违反了这些规则，可能很难维护。在这两种情况下，违规都有可能使其他使用代码的程序员感到困惑和恼怒，并使他们做出错误的假设，从而导致错误。本条目概述了各项约定。

包名和模块名应该是分层的，组件之间用句点分隔。组件应该由小写字母组成，很少使用数字。任何在你的组织外部使用的包，名称都应该以你的组织的 Internet 域名开头，并将组件颠倒过来，例如，edu.cmu、com.google、org.eff。以 java 和 javax 开头的标准库和可选包是这个规则的例外。用户不能创建名称以 java 或 javax 开头的包或模块。将 Internet 域名转换为包名前缀的详细规则可以在《The Java Language Specification》[JLS, 6.1] 中找到。

包名的其余部分应该由描述包的一个或多个组件组成。组件应该很短，通常为 8 个或更少的字符。鼓励使用有意义的缩写，例如 util 而不是 utilities。缩写词是可以接受的，例如 awt。组件通常应该由一个单词或缩写组成。

除了 Internet 域名之外，许多包的名称只有一个组件。附加组件适用于大型工具包，这些工具包的大小要求将其分解为非正式的层次结构。例如 `javax.util` 包具有丰富的包层次结构，包的名称如 `java.util.concurrent.atomic`。这样的包称为子包，尽管 Java 几乎不支持包层次结构。

类和接口名称，包括枚举和注释类型名称，应该由一个或多个单词组成，每个单词的首字母大写，例如 List 或 FutureTask。除了缩略语和某些常见的缩略语，如 max 和 min，缩略语应该避免使用。缩略语应该全部大写，还是只有首字母大写，存在一些分歧。虽然有些程序员仍然使用大写字母，但支持只将第一个字母大写的理由很充分：即使多个首字母缩写连续出现，你仍然可以知道一个单词从哪里开始，下一个单词从哪里结束。你希望看到哪个类名，HTTPURL 还是 HttpUrl?

方法和字段名遵循与类和接口名相同的排版约定，除了方法或字段名的第一个字母应该是小写，例如 remove 或 ensureCapacity。如果方法或字段名的首字母缩写出现在第一个单词中，那么它应该是小写的。

前面规则的唯一例外是「常量字段」，它的名称应该由一个或多个大写单词组成，由下划线分隔，例如 VALUES 或 NEGATIVE_INFINITY。常量字段是一个静态的 final 字段，其值是不可变的。如果静态 final 字段具有基本类型或不可变引用类型(第17项)，那么它就是常量字段。例如，枚举常量是常量字段。如果静态 final 字段有一个可变的引用类型，那么如果所引用的对象是不可变的，那么它仍然可以是一个常量字段。注意，常量字段是唯一推荐使用下划线用法的。

局部变量名与成员名具有类似的排版命名约定，但允许使用缩写，也允许使用单个字符和短字符序列，它们的含义取决于它们出现的上下文，例如 i、denom、houseNum。输入参数是一种特殊的局部变量。它们的命名应该比普通的局部变量谨慎得多，因为它们的名称是方法文档的组成部分。

类型参数名通常由单个字母组成。最常见的是以下五种类型之一：T 表示任意类型，E 表示集合的元素类型，K 和 V 表示 Map 的键和值类型，X 表示异常。函数的返回类型通常为 R。任意类型的序列可以是 T、U、V 或 T1、T2、T3。

为了快速参考，下表显示了排版约定的示例。

Identifier Type	Example
Package or module	`org.junit.jupiter.api`, `com.google.common.collect`
Class or Interface	Stream, FutureTask, LinkedHashMap,HttpClient
Method or Field	remove, groupingBy, getCrc
Constant Field	MIN_VALUE, NEGATIVE_INFINITY
Local Variable	i, denom, houseNum
Type Parameter	T, E, K, V, X, R, U, V, T1, T2

语法命名约定比排版约定更灵活，也更有争议。包没有语法命名约定。可实例化的类，包括枚举类型，通常使用一个或多个名词短语来命名，例如 Thread、PriorityQueue 或 ChessPiece。不可实例化的实用程序类（详见第 4 条）通常使用复数名词来命名，例如 collector 或 Collections。接口的名称类似于类，例如集合或比较器，或者以 able 或 ible 结尾的形容词，例如 Runnable、Iterable 或 Accessible。因为注解类型有很多的用途，所以没有哪部分占主导地位。名词、动词、介词和形容词都很常见，例如，BindingAnnotation、Inject、ImplementedBy 或 Singleton。

执行某些操作的方法通常用动词或动词短语（包括对象）命名，例如，append 或 drawImage。返回布尔值的方法的名称通常以单词 is 或 has（通常很少用）开头，后面跟一个名词、一个名词短语，或者任何用作形容词的单词或短语，例如 isDigit、isProbablePrime、isEmpty、isEnabled 或 hasSiblings。

返回被调用对象的非布尔函数或属性的方法通常使用以 get 开头的名词、名词短语或动词短语来命名，例如 size、hashCode 或 getTime。有一种说法是，只有第三种形式（以 get 开头）才是可接受的，但这种说法几乎没有根据。前两种形式的代码通常可读性更强，例如：

```
if (car.speed() > 2 * SPEED_LIMIT)
    generateAudibleAlert("Watch out for cops!");
```

以 get 开头的表单起源于基本过时的 Java bean 规范，该规范构成了早期可重用组件体系结构的基础。有一些现代工具仍然依赖于 bean 命名约定，你应该可以在任何与这些工具一起使用的代码中随意使用它。如果类同时包含相同属性的 setter 和 getter，则遵循这种命名约定也有很好的先例。在本例中，这两个方法通常被命名为 getAttribute 和 setAttribute。

一些方法名称值得特别注意。转换对象类型（返回不同类型的独立对象）的实例方法通常称为 toType，例如 toString 或 toArray。返回与接收对象类型不同的视图（详见第 6 条）的方法通常称为 asType，例如 asList。返回与调用它们的对象具有相同值的基本类型的方法通常称为类型值，例如 intValue。静态工厂的常见名称包括 from、of、valueOf、instance、getInstance、newInstance、getType 和 newType（详见第 1 条，第 9 页）。

字段名的语法约定没有类、接口和方法名的语法约定建立得好，也不那么重要，因为设计良好的 API 包含很少的公开字段。类型为 boolean 的字段的名称通常类似于 boolean 访问器方法，省略了初始值「is」，例如 initialized、composite。其他类型的字段通常用名词或名词短语来命名，如 height、digits 和 bodyStyle。局部变量的语法约定类似于字段的语法约定，但要求更少。

总之，将标准命名约定内在化，并将其作为第二性征来使用。排版习惯是直接的，而且在很大程度上是明确的；语法惯例更加复杂和松散。引用《The Java Language Specification》[JLS, 6.1] 中的话说，「如果长期以来的传统用法要求不遵循这些约定，就不应该盲目地遵循这些约定。」，应使用常识判断。

69. 只针对异常的情况下才使用异常

假如你某一天不走运的话，可能遇到如下代码：

```
/* Horrible abuse of exceptions. Don't ever do this! */
try {
    int i = 0;
    while ( true )
        range[i++].climb();
} catch ( ArrayIndexOutOfBoundsException e ) {
}
```

这段代码是干什么的？看起来根本不明显，这正是它没有真正被使用的原因（详见 67 条）。事实证明，作为一个要对数组元素进行遍历的实现方式，它的构想是十分拙劣的。当这个循环企图访问数组边界之外的第一个数组元素的时候，使用 try-catch 并且忽略 `ArrayIndexOutOfBoundsException` 异常的手段来达到终止无限循环的目的。假定它与数组循环的标准模式是等价的，对于它的标准模式每个 Java 程序员都可以一眼辨认出来：

```
for ( Mountain m : range )
    m.climb();
```

那么为什么有人会企图使用基于异常的循环，而不是使用行之有效的模式呢？这是他们误以为可以使用 Java 的错误判断机制来提高程序性能，因为 VM 对每次数组访问都要检查越界情况，所以他们认为正常的循环终止测试被编译器隐藏了，但是在 for-each 中仍然可见，这是多余的并且应当避免。这种想法有三个错误：

- 因为异常设计的初衷适用于不正常的情形，所有几乎没有 JVM 实现试图对他们进行优化，使它们与显式的测试一样快。
- 把代码放在 try-catch 块中反而阻止了现代 JVM 实现本可能执行的某些特定优化。
- 对数据进行遍历的标准模式并不会导致冗余的检查。有些 JVM 实现会将它们优化掉。

实际上基于异常的模式比标准模式要慢得多。在我本地的机器上，对于一个有 100 个元素的数组进行遍历，标准模式比基于异常的模式快了 2 倍。

基于异常的循环模式不仅模糊了代码的意图，降低了它的性能，而且它还不能保证正常工作！如果出现了不相关的 bug，这个模式会悄悄的消失从而掩盖了这个 Bug，极大地增加了调试过程的复杂性。假设循环体的计算过程中调用了一个方法，这个方法执行了对某个不相关数组的越界访问。如果使用合理的循环模式，这个 Bug 会产生未被捕捉的异常，从而导致线程立即结束，并产生完整的堆栈轨迹。如果使用这个被误导的基于异常的循环模式，与这个 Bug 相关的异常将会被捕捉到，并且被错误的解释为正常的循环终止条件。

这个例子的教训很简单：顾名思义，**异常应该只用于异常的情况下；他们永远不应该用于正常的程序控制流程。** 一般的，应该优先使用标准的、容易理解的模式，而不是那些声称可以提供更好性能的、弄巧成拙的方法。即使真的能够改进性能，面对平台的不断改进，这种模型的性能优势也不可能一直保持。然而这种过度聪明的模式带来的微妙 Bug 和维护的痛苦将依旧存在。

这条原则对于 API 设计也有启发。**设计良好的 API 不应该强迫它的客户端为了正常的控制流程而使用异常。**如果类中具有「状态相关」（state-dependent）的方法，即只有在特定的不可预知的条件下才可以被调用的方法，这个类往往也应该具有一个单独的「状态测试」（state-testing）方法，即表明是否可以调用这个状态相关的方法。比如 Iterator 接口含有状态相关的 next 方法，以及相应的状态测试方法 hasNext。这使得利用传统的 for 循环（以及 for-each 循环，在内部使用了 hasNext 方法）对集合进行迭代的标准模式成为可能。

```
for ( Iterator<Foo> i = collection.iterator(); i.hasNext(); ){
    Foo foo = i.next();
    ...
}
```

如果 Iterator 缺少 hasNext 方法，客户端将被迫改用下面的做法：

```
/* Do not use this hideous code for iteration over a collection! */
try {
    Iterator<Foo> i = collection.iterator();
    while ( true )
    {
        Foo foo = i.next();
        ...
    }
} catch ( NoSuchElementException e ) {
}
```

这应该非常类似于本条目刚开始时对数据进行迭代的例子。除了代码繁琐令人误解之外，这个基于异常的模式可能执行起来也比标准模式更差，并且还可能掩盖系统中其他不相关部分的 Bug。

另外一种提供单独状态测试的做法是，如果「状态相关」方法无法执行想要的计算，就可以让它返回一个零长度的 optional 值（详见第 55 条），或者返回一个可被识别的返回值，比如 null。

对于「状态测试方法」和「optional 返回值或者可识别的返回值」这两种做法，有些指导原则可以帮助你在两者之间做出选择。如果对象将在缺少外部同步的情况下被并发访问，或者可被外界改变状态，就必须使用「optional 返回值或者可识别的返回值」，因为在调用「状态测试」方法和调用对应的「状态相关」方法的时间间隔之中，对象的状态有可能发生变化。如果单独的「状态测试」方法必须重复「状态相关」方法的工作，从性能的角度考虑，就必须使用可被识别的返回值。如果其他方面都是等同的，那么「状态测试」方法则优于可被识别的返回值。他提供了相对更高的可读性，对于使用不当的情形可能更加易于检测和改正：如果忘了去调用状态测试方法，状态相关的方法就会抛出异常，使得这个 Bug 变得很明显；如果忘了去检查可识别的返回值，这个 Bug 就很难被发现。optional 返回值不会有这方面的问题。

总而言之，异常是为了在异常情况下被设计和使用的。不要将它们勇于普通的控制流程，也不要编写迫使它们这么做的 API。

70. 对可恢复的情况使用受检异常，对编程错误使用运行时异常

Java 程序设计语言提供了三种 throwable：受检异常（checked exceptions）、运行时异常（runtime exceptions）和错误（errors）。程序员中存在着什么情况适合使用哪种 throwable 的困惑。虽然这种决定不总是那么清晰，但还是有一些一般性的原则提出了强有力的指导。

在决定使用受检异常还是非受检异常时，主要的原则是：**如果期望调用者能够合理的恢复程序运行，对于这种情况就应该使用受检异常。** 通过抛出受检异常，强迫调用者在一个 catch 子句中处理该异常，或者把它传播出去。因此，方法中声明要抛出的每个受检异常都是对 API 用户的一个潜在提示：与异常相关联的条件是调用这个方法一种可能结果。

API 的设计者让 API 用户面对受检异常，以此强制用户从这个异常条件条件中恢复。用户这可以忽视这样的强制要求，只需要捕获异常即可，但这往往不是个好办法（详见第 77 条）。

有两种非受检的 throwable：运行时异常和错误。在行为上两者是等同的：它们都是不需要也不应该被捕获的 throwable。如果程序抛出非受检异常或者错误，往往属于不可恢复的情形，程序继续执行下去有害无益。如果程序没有捕捉到这样的 throwable，将会导致当前线程中断（halt），并且出现适当的错误消息。

用运行时异常来表明编程错误。大多数运行时异常都表示前提违例（precondition violations）。所谓前提违例是指 API 的客户没有遵守 API 规范建立的约定。例如，数组访问的预定指明了数组的下标值必须在 0 和数组长度-1 之间。`ArrayIndexOutOfBoundsException` 表明违反了这个前提。

这个建议有一个问题：对于要处理可恢复的条件，还是处理编程错误，情况并非总是那么黑白分明。例如，考虑资源枯竭的情形，这可能是由程序错误引起的，比如分配了一块不合理的过大数组，也可能确实是由于资源不足而引起的。如果资源枯竭是由于临时的短缺，或是临时需求太大造成的，这种情况可能是可恢复的。API 设计者需要判断这样的资源枯竭是否允许恢复。如果你相信一种情况可能允许回复，就使用受检异常；如果不是，则使用运行时异常。如果不清楚是否有可能恢复，最好使用非受检异常，原因参见 71 条。

虽然 JLS（Java 语言规范）并没有要求，但是按照惯例，错误（Error）往往被 JVM 保留下来使用，以表明资源不足、约束失败，或者其他使程序无法继续执行的条件。由于这已经是个几乎被普遍接受的管理，因此最好不需要在实现任何新的 `Error` 的子类。因此，**你实现的所有非受检的 throwable 都应该是 `RuntimeExceptiond` 子类**（直接或者间接的）。不仅不应该定义 `Error` 的子类，也不应该抛出 `AssertionError` 异常。

要想定义一个不是 `Exception`、`RuntimeException` 或者 `Error` 子类的 throwable，这也是有可能的。JLS 并没有直接规定这样的 throwable，而是隐式的指定了：从行为意义上讲，他们等同于普通的受检异常（即 `Exception` 的子类，但不是 `RuntimeException` 的子类）。那么什么时候应该使用这样的 throwable？一句话，永远也不会用到。它与普通的受检异常相比没有任何益处，还会困扰 API 的使用者。

API 的设计者往往会忘记，异常也是一个完全意义上的对象，可是在它上面定义任何的方法。这些方法的主要用途是捕获异常的代码提供额外信息，特别是关于引发这个异常条件的信息。如果没有这样的方法，程序员必须要懂的如何解析「该异常的字符串表示法」，以便获得这些额外信息。这是极为不好的做法（详见 12 条）。类很少会指定它们的字符串表示法中的细节，因此对于不同的实现及不同的

版本，字符串表示法也会大相径庭。由此可见，“解析异常的字符串表示法”的代码可能是不可移植的，也是非常脆弱的。

因为受检异常往往指明了可恢复的条件，所以对于这样的异常，提供一些辅助方法尤其重要，通过这种方法调用者可以获得一些有助于程序恢复的信息。例如，假设因为用户资金不足，当他企图购买一张礼品卡时导致失败，于是抛出受检异常。这个异常应该提供一个访问方法，以便允许客户查询所缺的费用金额，使得使用者可以将这个数值传递给用户。关于这个主题的更多详情，参见 75 条。

总而言之，对于可恢复的情况，要抛出受检异常；对于程序错误，就要抛出运行时异常。不确定是否可恢复，就跑出为受检异常。不要定义任何既不是受检异常也不是运行异常的抛出类型。要在受检异常上提供方法，以便协助程序恢复。

71. 避免不必要的使用受检异常

Java 程序员不喜欢受检异常，但是如果使用得当，它们可以改善 API 和程序。不返回码和未受检异常的是，它们强迫程序员处理异常的条件，大大增强了可靠性。也就是说，过分使用受检异常会使 API 使用起来非常不方便。如果方法抛出受检异常，调用该方法代码就必须在一个或者多个 catch 块中处理这些异常，或者它必须声明抛出这些异常，并让它们传播出去。无论使用哪一种方法，都给程序员增添了不可忽视的负担。这种负担在 Java 8 中更重了，因为抛出受检异常的方法不能直接在 Stream 中使用（详见第 45 条至第 48 条）。

如果正确地使用 API 并不能阻止这种异常条件的产生，并且一旦产生异常，使用 API 的程序员可以立即采取有用的动作，这种负担就被认为是正当的。除非这两个条件都成立，否则更适合于使用未受检异常。作为一个石蕊测试（石蕊测试是指简单而具有决定性的测试），你可以试着问自己：程序员将如何处理该异常。下面的做法是最好的吗？

```
} catch ( TheCheckedException e ) {
    throw new AssertionError(); /* Can't happen! */
}
```

下面这种做法又如何？

```
} catch ( TheCheckedException e ) {
    e.printStackTrace(); /* Oh well, we lose. */
    System.exit( 1 );
}
```

如果使用 API 的程序员无法做得比这更好那么未受检的异常可能更为合适。

如果方法抛出的受检异常是唯一的，它给程序员带来的额外负担就会非常高。如果这个方法还有其他的受检异常，该方法被调用的时候，必须已经出现在一个 try 块中，所以这个异常只需要另外一个 catch 块。如果方法只抛出一个受检异常，单独这一个异常就表示：该方法必须放置于一个 try 块中，并且不能在 Stream 中直接使用。这种情况下，应该问问自己，是否还有别的途径可以避免使用受检异常。

除受检异常最容易的方法是，返回所要的结果类型的一个 optional（详见第 55 条）。这个方法不抛出受检异常，而只是返回一个零长度的 optional。这种方法的缺点是，方法无法返回任何额外的信息，来详细说明它无法执行你想要的计算。相反，异常则具有描述性的类型，并且能够导出方法，以提供额外的信息（详见第 70 条）。

「把受检异常变成未受检异常」的一种方法是，把这个抛出异常的方法分成两个方法，其中第一个方法返回一个 boolean 值，表明是否应该抛出异常。这种 API 重构，把下面的调用序列：

```
/* Invocation with checked exception */
try {
    obj.action( args );
} catch ( TheCheckedException e ) {
    ... /* Handle exceptional condition */
}
```

重构为：

```
/* Invocation with state-testing method and unchecked exception */
if ( obj.actionPermitted( args ) ) {
    obj.action( args );
} else {
    ... /* Handle exceptional condition */
}
```

这种重构并非总是恰当的，但是，凡是在恰当的地方，它都会使 API 用起来更加舒服。虽然后者的调用序列没有前者漂亮，但是这样得到的 API 更加灵活。如果程序员知道调用将会成功，或者不介意由于调用失败而导致的线程终止，这种重构还允许以下这个更为简单的调用形式：

```
obj.action(args);
```

如果你怀疑这个简单的调用序列是否符合要求，这个 API 重构可能就是恰当的。这样重构之后的 API 在本质上等同于第 69 条中的「状态测试方法」，并且同样的告诫依然适用：如果对象将在缺少外部同步的情况下被并发访问，或者可被外界改变状态，这种重构就是不恰当的，因为在 actionPermitted 和 action 这两个调用的时间间隔之中，对象的状态有可能会发生变化。如果单独的 actionPermitted 方法必须重复 action 方法的工作，出于性能的考虑，这种 API 重构就不值得去做。

总而言之，在谨慎使用的前提之下，受检异常可以提升程序的可读性；如果过度使用，将会使 API 使用起来非常痛苦。如果调用者无法恢复失败，就应该抛出未受检异常。如果可以恢复，并且想要迫使调用者处理异常的条件，首选应该返回一个 optional 值。当且仅当万一失败时，这些无法提供足够的信息，才应该抛出受检异常。

72. 优先使用标准的异常

专家级程序员与缺乏经验的程序员一个最主要的区别在于，专家追求并且通常也能够实现高度的代码重用。代码重用是值得提倡的，这是一条通用的规则，异常也不例外。Java 平台类库提供了一组基本的未受检异常，它们满足了绝大多数 API 的异常抛出需求。

重用标准的常有多个好处。其中最主要的好处是，它使 API 更易于学习和使用，因为它与程序员已经熟悉的习惯用法一致。第二个好处是，对于用到这些 API 程序而言，它们的可读性会更好，因为它们不会出现很多程序员不熟悉的异常。最后（也是最不重要的）一点是，异常类越少，意味着内存占用（footprint）就越小，装载这些类的时间开销也越少。

最经常被重用的异常类型是 `IllegalArgumentException`（详见第 49 条）。当调用者传递的参数值不合适的时候，往往就会抛出这个异常。比如，假设某一个参数代表了“某个动作的重复次数”，如果程序员给这个参数传递了一个负数，就会抛出这个异常。

另一个经常被重用的异常是 `IllegalStateException`。如果因为接收对象的状态而使调用非法，通常就会抛出这个异常。例如，如果在某个对象被正确地初始化之前，调用者就企图使用这个对象，就会抛出这个异常。

可以这么说，所有错误的方法调用都可以被归结为非法参数或者非法状态，但是，还有一些其他的标准异常也被用于某些特定情况下的非法参数和非法状态。如果调用者在某个不允许 null 值的参数中传递了 null，习惯的做法就是抛出 `NullPointerException` 异常，而不是 `IllegalArgumentException`。同样地，如果调用者在表示序列下标的参数中传递了越界的值，应该抛出的就是 `IndexOutOfBoundsException` 异常，而不是 `IllegalArgumentException`。

另一个值得了解的通用异常是 `ConcurrentModificationException`。如果检测到一个专门设计用于单线程的对象，或者与外部同步机制配合使用的对象正在（或已经）被并发地修改，就应该抛出这个异常。这个异常顶多就是一个提示，因为不可能可靠地侦测到并发的修改。

最后一个值得注意的标准异常是 `UnsupportedOperationException`。如果对象不支持所请求的操作，就会抛出这个异常。很少用到它，因为绝大多数对象都会支持它们实现的所有方法。如果类没有实现由它们实现的接口所定义的一个或者多个可选操作（optional operation），它就可以使用这个异常。例如，对于只支持追加操作的 List 实现，如果有人试图从列表中删除元素，它就会抛出这个异常。

不要直接重用 Exception、RuntimeException、Throwable 或者 Error。 对待这些类要像对待抽象类一样。你无法可靠地测试这些异常，因为它们是一个方法可能抛出的其他异常的超类。

下表概括了最常见的可重用异常：

异常	使用场合
IllegalArgumentException	非 null 的参数值不正确
IllegalStateException	不适合方法调用的对象状态
NullPointerException	在禁止使用 null 的情况下参数值为 null
IndexOutOfBoundsExecption	下标参数值越界
ConcurrentModificationException	在禁止并发修改的情况下，检测到对象的并发修改
UnsupportedOperationException	对象不支持用户请求的方法

然这些都是 Java 平台类库中迄今为止最常被重用的异常，但是，在条件许可的情况下，其他的异常也可以被重用。例如，如果要实现诸如复数或者有理数之类的算术对象，也可以重用 `ArithmeticException` 和 `NumberFormatException`。如果某个异常能够满足你的需要，就不要犹豫，使用就是，不过一定要确保抛出异常的条件与该异常的文档中描述的条件一致。这种重用必须建立在语义的基础上，而不是建立在名称的基础之上。而且，如果希望稍微增加更多的失败一捕获（failure-capture）信息（详见第 75 条），可以放心地子类化标准异常，但要记住异常是可序列化的（详见第 12 章）。这也正是「如果没有非常正当的理由，千万不要自己编写异常类」的原因。

选择重用哪一种异常并非总是那么精确，因为上表中的"使用场合"并不是相互排斥的比如，以表示一副纸牌的对象为例。假设有一个处理发牌操作的方法，它的参数是发一手牌的纸牌张数。假设调用者在这个参数中传递的值大于整副纸牌的剩余张数。这种情形既可以被解释为 `IllegalArgumentException`（handSize 参数的值太大），也可以被解释为 `IllegalStateException`（纸牌对象包含的纸牌太少）。在这种情况下，**如果没有可用的参数值，就抛出 `IllegalStateException`，否则就抛出 `IllegalArgumentException`。**

73. 抛出与抽象对应的异常

如果方法抛出的异常与它所执行的任务没有明显的联系，这种情形将会使人不知所措。当方法传递由低层抽象抛出的异常时，往往会发生这种情况。除了使人感到困惑之外，这也"污染"了具有实现细节的更高层的 API 。如果高层的实现在后续的发行版本中发生了变化，它所抛出的异常也可能会跟着发生变化，从而潜在地破坏现有的客户端程序。

为了避免这个问题，**更高层的实现应该捕获低层的异常，同时抛出可以按照高层抽象进行解释的异常。**这种做法称为异常转译（exception translation），如下代码所示：

```
/* Exception Translation */
try {
    ... /* Use lower-level abstraction to do our bidding */
} catch (LowerLevelException e) {
    throw new HigherLevelException(...);
}
```

下面的异常转译例子取自于 `AbstractSequentialList` 类，该类是 List 接口的一个骨架实现（skeletal implementation），详见第 20 条。在这个例子中，按照 `List<E>` 接口中 get 方法的规范要求，异常转译是必需的：

```
/**
 * Returns the element at the specified position in this list.
 * @throws IndexOutOfBoundsException if the index is out of range
 * ({@code index < 0 || index >= size()}).
 */
public E get(int index) {
    ListIterator<E> i = listIterator(index);
    try {
        return(i.next() );
    } catch (NoSuchElementException e) {
        throw new IndexOutOfBoundsException("Index: " + index);
    }
}
```

一种特殊的异常转译形式称为异常链（exception chaining），如果低层的异常对于调试导致高层异常的问题非常有帮助，使用异常链就很合适。低层的异常（原因）被传到高层的异常，高层的异常提供访问方法 (Throwable 的 getCause 方法）来获得低层的异常：

```
// Exception Chaining
try {
... // Use lower-level abstraction to do our bidding
} catch (LowerLevelException cause) {
    throw new HigherLevelException(cause);
}
```

高层异常的构造器将原因传到支持链（chaining-aware）的超级构造器，因此它最终将被传给 Throwable 的其中一个运行异常链的构造器，例如 Throwable(Throwable) :

```
/* Exception with chaining-aware constructor */
class HigherLevelException extends Exception {
    HigherLevelException( Throwable cause ) {
        super(cause);
    }
}
```

大多数标准的异常都有支持链的构造器。对于没有支持链的异常，可以利用 Throwable 的 initCause 方法设置原因。异常链不仅让你可以通过程序（用 getCause）访问原因，还可以将原因的堆栈轨迹集成到更高层的异常中。

尽管异常转译与不加选择地从低层传递异常的做法相比有所改进，但是也不能滥用它。 如有可能，处理来自低层异常的最好做法是，在调用低层方法之前确保它们会成功执行，从而避免它们抛出异常。有时候，可以在给低层传递参数之前，检查更高层方法的参数的有效性，从而避免低层方法抛出异常。

如果无法阻止来自低层的异常，其次的做法是，让更高层来悄悄地处理这些异常，从而将高层方法的调用者与低层的问题隔离开来。在这种情况下，可以用某种适当的记录机制（如 java.util.logging）将异常记录下来。这样有助于管理员调查问题，同时又将客户端代码和最终用户与问题隔离开来。

总而言之，如果不能阻止或者处理来自更低层的异常，一般的做法是使用异常转译，只有在低层方法的规范碰巧可以保证“它所抛出的所有异常对于更高层也是合适的”情况下，才可以将异常从低层传播到高层。异常链对高层和低层异常都提供了最佳的功能：它允许抛出适当的高层异常，同时又能捕获低层的原因进行失败分析（详见第 75 条）。

74. 每个方法抛出的异常都需要创建文档

描述一个方法所抛出的异常，是正确使用这个方法时所需文档的重要组成部分。因此，花点时间仔细地为每个方法抛出的异常建立文档是特别重要的。

始终要单独地声明受检异常， 并且利用 Javadoc 的 `@throws` 标签， **准确地记录下抛出每个异常的条件。** 如果一个公有方法可能抛出多个异常类，则不要使用“快捷方式”声明它会抛出这些异常类的某个超类。永远不要声明一个公有方法直接「throws Exception」，或者更糟糕的是声明它直接「throws Throwable」，这是非常极端的例子。这样的声明不仅没有为程序员提供关于“这个方法能够抛出哪些异常”的任何指导信息，而且大大地妨碍了该方法的使用，因为它实际上掩盖了该方法在同样的执行环境下可能抛出的任何其他异常。这条建议有一个例外，就是 main 方法它可以被安全地声明抛出 Exception，因为它只通过虚拟机调用。

虽然 Java 语言本身并没有要求程序员为一个方法声明它可能会抛出的未受检异常，但是，如同受检异常一样，仔细地为它们建立文档是非常明智的。未受检异常通常代表编程上的错误(详见第 70 条)，让程序员了解所有这些错误都有助于帮助他们避免犯同样的错误。对于方法可能抛出的未受检异常，如果将这些异常信息很好地组织成列表文档，就可以有效地描述出这个方法被成功执行的前提条件。每个方法的文档应该描述它的前提条件（详见第 56 条），这是很重要的，在文档中记录下未受检异常是满足前提条件的最佳做法。

对于接口中的方法，在文档中记录下它可能抛出的未受检异常显得尤为重要。这份文档构成了该接口的通用约定（general contract）的一部分，它指定了该接口的多个实现必须遵循的公共行为。

使用 Javadoc 的 `@throws` 标签记录下一个方法可能抛出的每个未受检异常，但是不要使用 throws 关键字将未受检的异常包含在方法的声明中。 使用 API 的程序员必须知道哪些异常是需要受检的，哪些是不需要受检的，因为他们有责任区分这两种情形。当缺少由 throws 声明产生的方法标头时，由 Javadoc 的 `@throws` 标签所产生的文档就会提供明显的提示信息，以帮助程序员区分受检异常和未受检异常。

应该注意的是，为每个方法可能抛出的所有未受检异常建立文档是很理想的，但是在实践中并非总能做到这一点。当类被修订之后，如果有个导出方法被修改了，它将会抛出额外的未受检异常，这不算违反源代码或者二进制兼容性。假设一个类调用了另一个独立编写的类中的方法。第一个类的编写者可能会为每个方法抛出的未受检异常仔细地建立文档，但是，如果第二个类被修订了，抛出了额外的未受检异常，很有可能第一个类(它并没有被修订)就会把新的未受检异常传播出去，尽管它并没有声明这些异常。

如果一个类中的许多方法出于同样的原因而抛出同一个异常，在该类的文档注释中对这个异常建立文档，这是可以接受的， 而不是为每个方法单独建立文档。一个常见的例子是 `NullPointerException`。若类的文档注释中有这样的描述：「All methods in this class throw a NullPointerException if a null object reference is passed in any parameter」（如果 null 对象引用被传

递到任何一个参数中，这个类中的所有方法都会抛出 `NullPointerException` ），或者有其他类似的语句，这是可以的。

总而言之，要为你编写的每个方法所能抛出的每个异常建立文档。对于未受检异常和受检异常，以及抽象的方法和具体的方法一概如此。这个文档在文档注释中应当采用 `@throws` 标签的形式。要在方法的 throws 子句中为每个受检异常提供单独的声明，但是不要声明未受检的异常。如果没有为可以抛出的异常建立文档，其他人就很难或者根本不可能有效地使用你的类和接口。

75. 在细节消息中包含失败—捕获信息

当程序由于未被捕获的异常而失败的时候，系统会自动地打印出该异常的堆栈轨迹。在堆栈轨迹中包含该异常的字符串表示法 （string representation），即它的 toString 方法的调用结果。它通常包含该异常的类名，紧随其后的是细节消息 （detail message）。通常，这只是程序员或者网站可靠性工程师在调查软件失败原因时必须检查的信息。如果失败的情形不容易重现，要想获得更多的信息会非常困难，甚至是不可能的。因此，异常类型的 toString 方法应该尽可能多地返回有关失败原因的信息，这一点特别重要。换句话说，异常的字符串表示法应该捕获失败，以便于后续进行分析。

为了捕获失败，异常的细节信息应该包含“对该异常有贡献”的所有参数和字段的值。 例如，`IndexOutOfBoundsException` 异常的细节消息应该包含下界、上界以及没有落在界内的下标值。该细节消息提供了许多关于失败的信息。这三个值中任何一个或者全部都有可能是错的。实际的下标值可能小于下界或等于上界（「越界错误」），或者它可能是个无效值，太小或太大。下界也有可能大于上界（严重违反内部约束条件的一种情况） 。每一种情形都代表了不同的问题，如果程序员知道应该去查找哪种错误，就可以极大地加速诊断过程。

对安全敏感的信息有一条忠告。由于在诊断和修正软件问题的过程中，许多人都可以看见堆栈轨迹， **因此千万不要在细节消息中包含密码、密钥以及类似的信息！**

虽然在异常的细节消息中包含所有相关的数据是非常重要的，但是包含大量的描述信息往往没有什么意义。堆栈轨迹的用途是与源文件结合起来进行分析，它通常包含抛出该异常的确切文件和行数，以及堆栈中所有其他方法调用所在的文件和行数。关于失败的冗长描述信息通常是不必要的，这些信息可以通过阅读源代码而获得。

异常的细节消息不应该与“用户层次的错误消息”混为一谈，后者对于最终用户而言必须是可理解的。与用户层次的错误消息不同，异常的字符串表示法主要是让程序员或者网站可靠性工程师用来分析失败的原因。因此，信息的内容比可读性要重要得多。用户层次的错误消息经常被本地化，而异常的细节消息则几乎没有被本地化。

为了确保在异常的细节消息中包含足够的失败 - 捕捉信息， 一种办法是在异常的构造器而不是字符串细节消息中引入这些信息。然后，有了这些信息，只要把它们放到消息描述中，就可以自动产生细节消息。例如 `IndexOutOfBoundsException` 使用如下构造器代替 `String` 构造器：

```
/**
 * Constructs an IndexOutOfBoundsException.
 *
 * @param lowerBound the lowest legal index value
 * @param upperBound the highest legal index value plus one
```

```
     * @param index the actual index value
     */
    public IndexOutOfBoundsException( int lowerBound, int upperBound,
                                      int index ) {
        // Generate a detail message that captures the failure
        super(String.format(
                  "Lower bound: %d, Upper bound: %d, Index: %d",
                  lowerBound, upperBound, index ) );
        // Save failure information for programmatic access
        this.lowerBound = lowerBound;
        this.upperBound = upperBound;
        this.index = index;
    }
```

从 Java 9 开始， `IndexOutOfBoundsException` 终于获得了一个构造器，它可以带一个类型为 int 的 index 参数值，但遗憾的是，它删去了 lowerBound 和 upperBound 参数。更通俗地说， Java 平台类库并没有广泛地使用这种做法，但是，这种做法仍然值得大力推荐。它使程序员更加易于抛出异常以捕获失败。实际上，这种做法使程序员不想捕获失败都难！ 这种做法可以有效地把代码集中起来放在异常类中，由这些代码对异常类自身中的异常产生高质量的细节消息，而不是要求类的每个用户都多余地产生细节消息。

正如第 70 条中所建议的， 为异常的失败 - 捕获信息（在上述例子中为 lowerBound 、upperBound 和 index）提供一些访问方法是合适的。提供这样的访问方法对受检的异常，比对未受检异常更为重要，因为失败一捕获信息对于从失败中恢复是非常有用的。程序员希望通过程序的手段来访问未受检异常的细节，这很少见 （尽管也是可以想象的） 。然而，即使对于未受检异常，作为一般原则提供这些访问方法也是明智的 （详见第 12 条） 。

76. 保持失败原子性

当对象抛出异常之后，通常我们期望这个对象仍然保持在一种定义良好的可用状态之中， 即使失败是发生在执行某个操作的过程中间。对于受检异常而言，这尤为重要，因为调用者期望能从这种异常中进行恢复。**一般而言，失败的方法调用应该使对象保持在被调用之前的状态。** 具有这种属性的方法被称为具有失败原子性 （failure atomic） 。

有几种途径可以实现这种效果。最简单的办法莫过于设计一个不可变的对象 （详见第 17 条） 。如果对象是不可变的，失败原子性就是显然的。如果一个操作失败了，它可能会阻止创建新的对象，但是永远也不会使已有的对象保持在不一致的状态之中，因为当每个对象被创建之后它就处于一致的状态之中，以后也不会再发生变化。

对于在可变对象上执行操作的方法，获得失败原子性最常见的办法是，在执行操作之前检查参数的有效性 （详见第 49 条） 。这可以使得在对象的状态被修改之前，先抛出适当的异常。比如，以第 7 条中的 Stack.pop 方法为例：

```
public Object pop() {
    if ( size == 0 )
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; /* Eliminate obsolete reference */
    return(result);
}
```

如果取消对初始大小（size）的检查，当这个方法企图从一个空栈中弹出元素时，它仍然会抛出异常。然而，这将会导致 size 字段保持在不一致的状态（负数）之中，从而导致将来对该对象的任何方法调用都会失败。此外，那时， pop 方法抛出的 `ArrayIndexOutOfBoundsException` 异常对于该抽象来说也是不恰当的（详见第 73 条）。

一种类似的获得失败原子性的办法是，调整计算处理过程的顺序，使得任何可能会失败的计算部分都在对象状态被修改之前发生。如果对参数的检查只有在执行了部分计算之后才能进行，这种办法实际上就是上一种办法的自然扩展。比如，以 `TreeMap` 的情形为例，它的元素被按照某种特定的顺序做了排序。为了向 `TreeMap` 中添加元素，该元素的类型就必须是可以利用 `TreeMap` 的排序准则与其他元素进行比较的。如果企图增加类型不正确的元素，在 tree 以任何方式被修改之前，自然会导致 `ClassCastException` 异常。

第三种获得失败原子性的办法是，在对象的一份临时拷贝上执行操作，当操作完成之后再用临时拷贝中的结果代替对象的内容。如果数据保存在临时的数据结构中，计算过程会更加迅速，使用这种办法就是件很自然的事。例如，有些排序函数会在执行排序之前，先把它的输入列表备份到一个数组中，以便降低在排序的内循环中访问元素所需要的开销。这是出于性能考虑的做法，但是，它增加了一项优势：即使排序失败，它也能保证输入列表保持原样。

最后一种获得失败原子性的办法远远没有那么常用，做法是编写一段恢复代码 （recovery code），由它来拦截操作过程中发生的失败，以及便对象回滚到操作开始之前的状态上。这种办法主要用于永久性的（基于磁盘的） 数据结构。

虽然一般情况下都希望实现失败原子性，但并非总是可以做到。举个例子，如果两个线程企图在没有适当的同步机制的情况下，并发地修改同一个对象，这个对象就有可能被留在不一致的状态之中。因此，在捕获了 `ConcurrentModificationException` 异常之后再假设对象仍然是可用的，这就是不正确的。错误通常是不可恢复的，因此，当方法抛出 `AssertionError` 时，不需要努力去保持失败原子性。

即使在可以实现失败原子性的场合，它也并不总是人们所期望的。对于某些操作，它会显著地增加开销或者复杂性。也就是说， 一旦了解了这个问题，获得失败原子性往往既简单又容易。

总而言之，作为方法规范的一部分，它产生的任何异常都应该让对象保持在调用该方法之前的状态。如果违反这条规则， API 文档就应该清楚地指明对象将会处于什么样的状态。遗憾的是，大量现有的 API 文档都未能做到这一点。

77. 不要忽略异常

尽管这条建议看上去是显而易见的，但是它却常常被违反，因而值得再次提出来。当 API 的设计者声明一个方法将抛出某个异常的时候，他们等于正在试图说明某些事情。所以，请不要忽略它！要忽略一个异常非常容易，只需将方法调用通过 try 语句包围起来，并包含一个空的 catch 块：

```
// Empty catch block ignores exception - Highly suspect!
try {
    ...
} catch ( SomeException e ) {
}
```

空的 catch 块会使异常达不到应有的目的， 即强迫你处理异常的情况。忽略异常就如同忽略火警信号一样——如果把火警信号器关掉了，当真正有火灾发生时，就没有人能看到火警信号了。或许你会侥幸逃过劫难，或许结果将是灾难性的。每当见到空的 catch 块时，应该让警钟长鸣。

有些情形可以忽略异常。比如，关闭 `FileinputStream` 的时候。因为你还没有改变文件的状态，因此不必执行任何恢复动作，并且已经从文件中读取到所需要的信息，因此不必终止正在进行的操作。即使在这种情况下，把异常记录下来还是明智的做法，因为如果这些异常经常发生，你就可以调查异常的原因。 **如果选择忽略异常，** **catch 块中应该包含一条注释，说明为什么可以这么做，并且变量应该命名为** ignored:

```
Future<Integer> f = exec.submit(planarMap::chromaticNumber);
int numColors = 4; // Default: guaranteed sufficient for any map
try {
    numColors = f.get( 1L, TimeUnit.SECONDS );
} catch ( TimeoutException | ExecutionException ignored ) {
    // Use default: minimal coloring is desirable, not required
}
```

本条目中的建议同样适用于受检异常和未受检异常。不管异常代表了可预见的异常条件，还是编程错误，用空的 catch 块忽略它，都将导致程序在遇到错误的情况下悄然地执行下去。然后，有可能在将来的某个点上，当程序不能再容忍与错误源明显相关的问题时，它就会失败。正确地处理异常能够彻底避免失败。只要将异常传播给外界，至少会导致程序迅速失败，从而保留了有助于调试该失败条件的信息。

78. 同步访问共享的可变数据

关键字 `synchronized` 可以保证在同一时刻，只有一个线程可以执行某一个方法，或者某一个代码块。许多程序员把同步的概念仅仅理解为一种互斥（ mutual exclusion ）的方式，即，当一个对象被一个线程修改的时候，可以阻止另一个线程观察到对象内部不一致的状态。按照这种观点，对象被创建的时候处于一致的状态（详见第 17 条），当有方法访问它的时候，它就被锁定了。这些方法观察到对象的状态，并且可能会引起状态转变（ statetransition ），即把对象从一种一致的状态转换到另一种一致的状态。正确地使用同步可以保证没有任何方法会看到对象处于不一致的状态中。

这种观点是正确的，但是它并没有说明同步的全部意义。如果没有同步，一个线程的变化就不能被其他线程看到。同步不仅可以阻止一个线程看到对象处于不一致的状态之中，它还可以保证进入同步方法或者同步代码块的每个线程，都能看到由同一个锁保护的之前所有的修改效果。

Java 语言规范保证读或者写一个变量是原子的（ atomic ），除非这个变量的类型为 long 或者 double [JLS , 17.4, 17.7] 。换句话说，读取一个非 long 或 double 类型的变量，可以保证返回值是某个线程保存在该变量中的， 即使多个线程在没有同步的情况下并发地修改这个变量也是如此。

你可能听说过，为了提高性能，在读或写原子数据的时候，应该避免使用同步。这个建议是非常危险而错误的。虽然语言规范保证了线程在读取原子数据的时候，不会看到任意的数值，但是它并不保证一个线程写入的值对于另一个线程将是可见的。 **为了在线程之间进行可靠的通信，也为了互斥访问，同步是必要的。** 这归因于 Java 语言规范中的内存模型（memory model），它规定了一个线程所做的变化何时以及如何变成对其他线程可见[JLS ,17.4; Goetz06, 16]。

如果对共享的可变数据的访问不能同步，其后果将非常可怕，即使这个变量是原子可读写的。以下面这个阻止一个线程妨碍，另一个线程的任务为例。Java 的类库中提供了 `Thread.stop` 方法，但是在很久以前就不提倡使用该方法了，因为它本质上是不安全的——使用它会导致数据遭到破坏。 **千万不要使用 `Thread.stop` 方法。** 要阻止一个线程妨碍另一个线程，建议的做法是让第一个线程轮询（ poll ）一个 boolean 字段，这个字段一开始为 false ，但是可以通过第二个线程设置为 true ，以表示第一个线程将终止自己。由于 boolean 字段的读和写操作都是原子的，程序员在访问这个字段的时候不再需要使用同步：

```
// Broken! - How long would you expect this program to run?
public class StopThread {
    private static Boolean stopRequested;

    public static void main(String[] args)
            throws InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested)
            i++;
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

你可能期待这个程序运行大约一秒钟左右，之后主线程将 stopRequested 设置为 true ，致使后台线程的循环终止。但是在我的机器上，这个程序永远不会终止：因为后台线程永远在循环！

问题在于，由于没有同步，就不能保证后台线程何时'看到'主线程对 stopRequested 的值所做的改变。没有同步，虚拟机将以下代码：

```
while (!stopRequested)
    i++;
```

转变成这样：

```
if (!stopRequested)
    while (true)
        i++;
```

这种优化称作提升（hoisting），正是 OpenJDK Server VM 的工作。结果是一个活性失败（liveness failure）：这个程序并没有得到提升。修正这个问题的一种方式是同步访问 stopRequested 字段。这个程序会如预期般在大约一秒之内终止：

```
// Properly synchronized cooperative thread termination
public class StopThread {
    private static Boolean stopRequested;
    private static synchronized void requestStop() {
        stopRequested = true;
    }
    private static synchronized Boolean stopRequested() {
        return stopRequested;
    }
    public static void main(String[] args)
            throws InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested())
                i++;
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}
```

注意写方法（requestStop）和读方法（stopRequested）都被同步了。只同步写方法还不够！**除非读和写操作都被同步，否则无法保证同步能起作用。** 有时候，会在某些机器上看到只同步了写（或读）操作的程序看起来也能正常工作，但是在这种情况下，表象具有很大的欺骗性。

`StopThread` 中被同步方法的动作即使没有同步也是原子的。换句话说，这些方法的同步只是为了它的通信效果，而不是为了互斥访问。虽然循环的每个迭代中的同步开销很小，还是有其他更正确的替代方法，它更加简洁，性能也可能更好。如果 stopRequested 被声明为 volatile，第二种版本的 `StopThread` 中的锁就可以省略。虽然 volatile 修饰符不执行互斥访问，但它可以保证任何一个线程在读取该字段的时候都将看到最近刚刚被写入的值：

```
// Cooperative thread termination with a volatile field
public class StopThread {
    private static volatile Boolean stopRequested;
    public static void main(String[] args)
            throws InterruptedException {
```

```
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested)
                i++;
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

在使用 volatile 的时候务必要小心。以下面的方法为例，假设它要产生序列号：

```
// Broken - requires synchronization!
private static volatile int nextSerialNumber = 0;

public static int generateSerialNumber() {
    return nextSerialNumber++;
}
```

这个方法的目的是要确保每个调用都返回不同的值（只要不超过 2^{32} 个调用）。这个方法的状态只包含一个可原子访问的字段： nextSerialNumber ，这个字段的所有可能的值都是合法的。因此，不需要任何同步来保护它的约束条件。然而，如果没有同步，这个方法仍然无法正确地工作。

问题在于，增量操作符（++）不是原子的。它在 nextSerialNumber 字段中执行两项操作：首先它读取值，然后写回一个新值，相当于原来的值再加上 1。如果第二个线程在第一个线程读取旧值和写回新值期间读取这个字段第二个线程就会与第一个线程一起看到同一个值，并返回相同的序列号。这就是安全性失败（ safety failure ）：这个程序会计算出错误的结果。

修正 `generateSerialNumber` 方法的一种方法是在它的声明中增加 `synchronized` 修饰符。这样可以确保多个调用不会交叉存取，确保每个调用都会看到之前所有调用的效果。一旦这么做，就可以且应该从 nextSerialNumber 中删除 volatile 修饰符。为了保护这个方法，要用 long 代替 int ，或者在 nextSerialNumber 要进行包装时抛出异常。

最好还是遵循第 59 条中的建议，使用 `AtomicLong` 类，它是 `java.util.concurrent.atomic` 的组成部分。这个包为在单个变量上进行免锁定、线程安全的编程提供了基本类型。虽然 volatile 只提供了同步的通信效果，但这个包还提供了原子性。这正是你想让 generateSerialNumber 完成的工作，并且它可能比同步版本完成得更好：

```
// Lock-free synchronization with java.util.concurrent.atomic
private static final Atomiclong nextSerialNum = new Atomiclong();

public static long generateSerialNumber() {
    return nextSerialNum.getAndIncrement();
}
```

避免本条目中所讨论到的问题的最佳办法是不共享可变的数据。要么共享不可变的数据（详见第 17 条），要么压根不共享。换句话说， **将可变数据限制在单个线程中。** 如果采用这一策略，对它建立文档就很重要，以便它可以随着程序的发展而得到维护。深刻地理解正在使用的框架和类库也很重要，因为它们引入了你不知道的线程。

让一个线程在短时间内修改一个数据对象，然后与其他线程共享，这是可以接受的，它只同步共享对象引用的动作。然后其他线程没有进一步的同步也可以读取对象，只要它没有再被修改。这种对象被称作高效不可变（ effectively immutable) [Goetz06, 3.5.4] 。将这种对象引用从一个线程传递到其他的线程被称作安全发布（ safe publication) [Goetz06, 3.5.3] 。安全发布对象引用有许多种方法：可以将它保存在静态字段巾，作为类初始化的一部分；可以将它保存在 volatile 字段、final 字段或者通过正常锁定访问的字段中；或者可以将它放到并发的集合中（详见第 81 条）。

总而言之， **当多个线程共享可变数据的时候，每个读或者写数据的线程都必须执行同步。** 如果没有同步，就无法保证一个线程所做的修改可以被另一个线程获知。未能同步共享可变数据会造成程序的活跃性失败（ liveness failure ）和安全性失败（ safety failure ） 。这样的失败是最难调试的。它们可能是间歇性的，且与时间相关，程序的行为在不同的虚拟机上可能根本不同。如果只需要线程之间的交互通信，而不需要互斥， volatile 修饰符就是一种可以接受的同步形式，但要正确地使用它可能需要一些技巧。

79. 避免过度同步

第 78 条告诫过我们缺少同步的危险性。本条目则关注相反的问题。依据情况的不同，过度同步则可能导致性能降低、死锁，甚至不确定的行为。

为了避免活性失败和安全性失败，在一个被同步的方法或者代码块中，永远不要放弃对客户端的控制。 换句话说，在一个被同步的区域内部，不要调用设计成要被覆盖的方法，或者是由客户端以函数对象的形式提供的方法（详见第 24 条） 。从包含该同步区域的类的角度来看，这样的方法是外来的（ alien ） 。这个类不知道该方法会做什么事情，也无法控制它。根据外来方法的作用，从同步区域中调用它会导致异常、死锁或者数据损坏。

为了对这个过程进行更具体的说明，以下面的类为例，它实现了一个可以观察到的集合包装（ set wrapper ） 。该类允许客户端在将元素添加到集合中时预订通知。这就是观察者（Observer ）模式［Gamma95 ］ 。为了简洁起见，类在从集合中删除元素时没有提供通知，但要提供通知也是一件很容易的事情。这个类是在第 18 条中可重用的 ForwardingSet 上实现的：

```
// Broken - invokes alien method from synchronized block!
public class ObservableSet<E> extends ForwardingSet<E> {
    public ObservableSet(Set<E> set) { super(set); }

    private final List<SetObserver<E>> observers= new ArrayList<>();

    public void addObserver(SetObserver<E> observer) {
        synchronized(observers) {
            observers.add(observer);
        }
    }
```

```
    public Boolean removeObserver(SetObserver<E> observer) {
        synchronized(observers) {
            return observers.remove(observer);
        }
    }

    private void notifyElementAdded(E element) {
        synchronized(observers) {
            for (SetObserver<E> observer : observers)
            observer.added(this, element);
        }
    }

    @Override
    public Boolean add(E element) {
        Boolean added = super.add(element);
        if (added)
        notifyElementAdded(element);
        return added;
    }

    @Override
    public Boolean addAll(Collection<? extends E> c) {
        Boolean result = false;
        for (E element : c)
        result |= add(element);
        // Calls notifyElementAdded
        return result;
    }
}
```

观察者通过调用 addObserver 方法预订通知，通过调用 removeObserver 方法取消预订。在这两种情况下，这个回调（ callback ）接口的实例都会被传递给方法：

```
@FunctionalInterface
public interface SetObserver<E> {
    // Invoked when an element is added to the observable set
    void added(ObservableSet<E> set, E element);
}
```

这个接口的结构与 `BiConsumer<Obser vableSet<E> ,E>` 一样。我们选择定义一个定制的函数接口，因为该接口和方法名称可以提升代码的可读性，且该接口可以发展整合多个回调。也就是说，还可以设置合理的参数来使用 BiConsumer （详见第 44 条）。

如果只是粗略地检验一下， `ObservableSet` 会显得很正常。例如，下面的程序打印出 0 ~ 99 的数字：

```
public static void main(String[] args) {
    ObservableSet<Integer> set = new ObservableSet<>(new HashSet<>());
    set.addObserver((s, e) -> System.out.println(e));
    for (int i = 0; i < 100; i++)
        set.add(i);
}
```

现在我们来尝试一些更复杂点的例子。假设我们用一个 `addObserver` 调用来代替这个调用，用来替换的那个 `addObserver` 调用传递了一个打印 `Integer` 值的观察者，这个值被添加到该集合中，如果值为 23 ，这个观察者要将自身删除：

```
set.addObserver(new SetObserver<>() {
    public void added(ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23)
            s.removeObserver(this);
    }
});
```

注意，这个调用以一个匿名类 `SetObserver` 实例代替了前一个调用中使用的 lambda 。这是因为函数对象需要将自身传给 `s.removeObserver` ，而 lambda 则无法访问它们自己（详见第 42 条） 。

你可能以为这个程序会打印数字 0 ~ 23 ，之后观察者会取消预订，程序会悄悄地完成它的工作。实际上却是打印出数字 0 ~ 23 ，然后抛出 `ConcurrentModificationException` 。问题在于，当 `notifyElementAdded` 调用观察者的 added 方法时，它正处于遍历 observers 列表的过程中。added `方法` 调用可观察集合的 `removeObserver` 方法，从而调用 `observers.remove` 。现在我们有麻烦了。我们正企图在遍历列表的过程中，将一个元素从列表中删除，这是非法的。 `notifyElementAdded` 方法中的迭代是在一个同步的块中，可以防止并发的修改，但是无法防止迭代线程本身回调到可观察的集合中，也无法防止修改它的 observers 列表。

现在我们要尝试一些比较奇特的例子： 我们来编写一个试图取消预订的观察者，但是不直接调用 `removeObserver` ，它用另一个线程的服务来完成。这个观察者使用了一个 executor service （详见第 80 条） ：

```
// Observer that uses a background thread needlessly
set.addObserver(new SetObserver<>() {
    public void added(ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23) {
            ExecutorService exec =
            Executors.newSingleThreadExecutor();
            try {
                exec.submit(() -> s.removeObserver(this)).get();
            }
            catch (ExecutionException | InterruptedException ex) {
                throw new  AssertionError (ex);
            }
```

```
            finally {
                exec.shutdown();
            }
        }
    }
});
```

顺便提一句，注意看这个程序在一个 catch 子句中捕获了两个不同的异常类型。这个机制是在 Java 7 中增加的，不太正式地称之为多重捕获（multi-catch）。它可以极大地提升代码的清晰度，行为与多异常类型相同的程序，其篇幅可以大幅减少。

运行这个程序时，没有遇到异常，而是遭遇了死锁。后台线程调用 `s.removeObserver`，它企图锁定 observers，但它无法获得该锁，因为主线程已经有锁了。在这期间，主线程一直在等待后台线程来完成对观察者的删除，这正是造成死锁的原因。

这个例子是刻意编写用来示范的，因为观察者实际上没理由使用后台线程，但这个问题却是真实的。从同步区域中调用外来方法，在真实的系统中已经造成了许多死锁，例如 GUI 工具箱。

在前面这两个例子中（异常和死锁），我们都还算幸运。调用外来方法（added）时，同步区域（observers）所保护的资源处于一致的状态。假设当同步区域所保护的约束条件暂时无效时，你要从同步区域中调用一个外来方法。由于 Java 程序设计语言中的锁是可重入的（reentrant），这种调用不会死锁。就像在第一个例子中一样，它会产生一个异常，因为调用线程已经有这个锁了，因此当该线程试图再次获得该锁时会成功，尽管概念上不相关的另一项操作正在该锁所保护的数据上进行着。这种失败的后果可能是灾难性的。从本质上来说，这个锁没有尽到它的职责。可重入的锁简化了多线程的面向对象程序的构造，但是它们可能会将活性失败变成安全性失败。

幸运的是，通过将外来方法的调用移出同步的代码块来解决这个问题通常并不太困难。对于 `notifyElementAdded` 方法，这还涉及给 observers 列表拍张“快照”，然后没有锁也可以安全地遍历这个列表了。经过这一修改，前两个例子运行起来便再也不会出现异常或者死锁了：

```
// Alien method moved outside of synchronized block - open calls
private void notifyElementAdded(E element) {
    List<SetObserver<E>> snapshot = null;
    synchronized(observers) {
        snapshot = new ArrayList<>(observers);
    }
    for (SetObserver<E> observer : snapshot)
        observer.added(this, element);
}
```

事实上，要将外来方法的调用移出同步的代码块，还有一种更好的方法。Java 类库提供了一个并发集合（concurrent collection），详见第 81 条，称作 `CopyOnWriteArrayList`，这是专门为此定制的。这个 `CopyOnWriteArrayList` 是 `ArrayList` 的一种变体，它通过重新拷贝整个底层数组，在这里实现所有的写操作。由于内部数组永远不改动，因此迭代不需要锁定，速度也非常快。如果大量使用，`CopyOnWriteArrayList` 的性能将大受影响，但是对于观察者列表来说却是很好的，因为它们几乎不改动，并且经常被遍历。

如果将这个列表改成使用 `CopyOnWriteArrayList` ，就不必改动 `ObservableSet` 的 `add` 和 `addAll` 方法。下面是这个类的其余代码。注意其中并没有任何显式的同步：

```
// Thread-safe observable set with CopyOnWriteArrayList
private final List<SetObserver<E>> observers = new CopyOnWriteArrayList<>();

public void addObserver(SetObserver<E> observer) {
    observers.add(observer);
}

public Boolean removeObserver(SetObserver<E> observer) {
    return observers.remove(observer);
}

private void notifyElementAdded(E element) {
    for (SetObserver<E> observer : observers)
        observer.added(this, element);
}
```

在同步区域之外被调用的外来方法被称作“开放调用”（open call）[Goetz06, 10 .1.4]。除了可以避免失败之外，开放调用还可以极大地增加并发性。外来方法的运行时间可能为任意时长。如果在同步区域内调用外来方法，其他线程对受保护资源的访问就会遭到不必要的拒绝。

通常来说，应该在同步区域内做尽可能少的工作。 获得锁，检查共享数据，根据需要转换数据，然后释放锁。如果你必须要执行某个很耗时的动作，则应该设法把这个动作移到同步区域的外面，而不违背第 78 条中的指导方针。

本条目的第一部分是关于正确性的。接下来，我们要简单地讨论一下性能。虽然自 Java 平台早期以来，同步的成本已经下降了，但更重要的是，永远不要过度同步。在这个多核的时代，过度同步的实际成本并不是指获取锁所花费的 CPU 时间；而是指失去了并行的机会，以及因为需要确保每个核都有一个 - 致的内存视图而导致的延迟。过度同步的另一项潜在开销在于，它会限制虚拟机优化代码执行的能力。

如果正在编写一个可变的类，有两种选择：省略所有的同步，如果想要并发使用，就允许客户端在必要的时候从外部同步，或者通过内部同步，使这个类变成是线程安全的（详见第 82 条），你还可以因此获得明显比从外部锁定整个对象更高的并发性。 `java.util` 中的集合（除了已经废弃的 `Vector` 和 `Hashtable` 之外）采用了前一种方法，而 `java.util.concurrent` 中的集合则采用了后一种方法（详见第 81 条）。

在 Java 平台出现的早期，许多类都违背了这些指导方针。例如， `StringBuffer` 实例几乎总是被用于单个线程之中，而它们执行的却是内部同步。为此， `StringBuffer` 基本上都由 `StringBuilder` 代替，它是一个非同步的 `StringBuffer` 。同样地， `java.util.Random` 中线程安全的伪随机数生成器，被 `java.util.concurrent.ThreadLocalRandom` 中非同步的实现取代，主要也是出于上述原因。当你不确定的时候，就不要同步类，而应该建立文档，注明它不是线程安全的。

如果你在内部同步了类，就可以使用不同的方法来实现高并发性，例如分拆锁、分离锁和非阻塞并发控制。这些方法都超出了本书的讨论范围，但有其他著作对此进行了阐述[Goetz06, Herlihy12] 。

如果方法修改了静态字段，并且该方法很可能要被多个线程调用，那么也必须在内部同步对这个字段的访问（除非这个类能够容忍不确定的行为）。多线程的客户端要在这种方法上执行外部同步是不可能的，因为其他不相关的客户端不需要同步也能调用该方法。字段本质上就是一个全局变量，即使是私有的也一样，因为它可以被不相关的客户端读取和修改。第 78 条中的 `generateSerialNumber` 方法使用的 `nextSerialNumber` 字段就是这样的一个例子。

总而言之，为了避免死锁和数据破坏，千万不要从同步区字段内部调用外来方法。更通俗地讲，要尽量将同步区字段内部的工作量限制到最少。当你在设计一个可变类的时候，要考虑一下它们是否应该自己完成同步操作。在如今这个多核的时代，这比永远不要过度同步来得更重要。只有当你有足够的理由一定要在内部同步类的时候，才应该这么做，同时还应该将这个决定清楚地写到文档中（详见第 82 条）。

80. executor 、 task 和 stream 优先于线程

本书第 1 版中阐述了简单的工作队列（work queue）[Bloch01，详见第 49 条] 代码。它利用一个后台线程，允许客户端可以插入异步处理任务到队列中。当不再需要这个工作队列时，客户端可以调用一个方法，让后台线程在完成队列中所有工作后，优雅的终止自己。这个实现仅比玩具复杂一点，即使这样，它依然需要整整一页精妙的代码，如果你不能正确的实现，将很容易会出现安全性和活性失败。幸运的是，你再不需要写这类代码了。

到本书第二版出版的时候， Java 平台中已经增加了 `java.util.concurrent` 包。它包含了一个很灵活的基于接口的任务执行工具 —— Executor Framework 。只需一行代码就可以创建了一个在各方面都比本书第一版更好的工作队列：

```
ExecutorService exec = Executors.newSingleThreadExecutor();
```

下面是为执行而提交一个 runnable 的方法：

```
exec.execute(runnable);
```

下面是告诉 executor 如何优雅地终止（如果你没有这么做，虚拟机可能不会退出）：

```
exec.shutdown();
```

你可以利用 executor service 完成更多的工作。例如，可以等待完成一项特殊的任务（就如第 79 条中的 get 方法一样），你可以等待一个任务集合中的任何任务或者所有任务完成（利用 `invokeAny` 或者 `invokeAll` 方法），可以等待 executor service 优雅地完成终止（利用 `awaitTermination` 方法），可以在任务完成时逐个地获取这些任务的结果（利用 `ExecutorCompletionService`），可以调度在某个特殊的时间段定时运行或者阶段性地运行的任务（利用 `ScheduledThreadPoolExecutor`），等等。

如果想让不止一个线程来处理来自这个队列的请求，只要调用一个不同的静态工厂，这个工厂创建了一种不同的 executor service ，称作线程池（thread pool ） 。你可以用固定或者可变数目的线程创建一个线程池。 `java.util.concurrent.Executors` 类包含了静态工厂，能为你提供所需的大多数 executor 。然而，如果你想来点特别的，可以直接使用 `ThreadPoolExecutor` 类。这个类允许你控制线程池操作的几乎每个方面。

为特殊的应用程序选择 executor service 是很有技巧的。如果编写的是小程序，或者是轻量负载的服务器，使用 `Executors.newCachedThreadPool` 通常是个不错的选择，因为它不需要配置，并且一般情况下能够「正确地完成工作」。但是对于大负载的服务器来说，缓存的线程池就不是很好的选择了！在缓存的线程池中，被提交的任务没有排成队列，而是直接交给线程执行。如果没有线程可用，就创建一个新的线程。如果服务器负载得太重，以致它所有的 CPU 都完全被占用了，当有更多的任务时，就会创建更多的线程，这样只会使情况变得更糟。因此，在大负载的产品服务器中，最好使用 `Executors.newFixedThreadPool` ，它为你提供了一个包含固定线程数目的线程池，或者为了最大限度地控制它，就直接使用 `ThreadPoolExecutor` 类。

不仅应该尽量不要编写自己的工作队列，而且还应该尽量不直接使用线程。当直接使用线程时，Thread 是既充当工作单元，又是执行机制。在 Executor Framework 中，工作单元和执行机制是分开的。现在关键的抽象是工作单元，称作任务（task） 。任务有两种： `Runnable` 及其近亲 `Callable` （它与 Runnable 类似，但它会返回值，并且能够抛出任意的异常）。执行任务的通用机制是 executor service 。如果你从任务的角度来看问题，并让一个 executor service 替你执行任务，在选择适当的执行策略方面就获得了极大的灵活性。本质上，Executor 框架执行的功能与 `Collections` 框架聚合（aggregation）的功能相同。

在 Java 7 中， Executor 框架被扩展为支持 fork-join 任务，这些任务是通过一种称作 fork-join 池的特殊 executor 服务运行的。fork-join 任务用 ForkJoinTask 实例表示，可以被分成更小的子任务，包含 `ForkJoinPool` 的线程不仅要处理这些任务，还要从另一个线程中“偷”任务，以确保所有的线程保持忙碌，从而提高 CPU 使用率、提高吞吐量，并降低延迟。fork-join 任务的编写和调优是很有技巧的。并行流 Parallel streams （详见第 48 条）是在 fork join 池上编写的，我们不费什么力气就能享受到它们的性能优势，前提是假设它们正好适用于我们手边的任务。

Executor Framework 的完整处理方法超出了本书的讨论范围，但是有兴趣的读者可以参阅《Java Concurrency in Practice》一书［Goetz06］ 。

81. 并发工具优于 wait 和 notify

本书第 1 版中专门用了一个条目来说明如何正确地使用 `wait` 和 `notify` （Bloch01，详见第 50 条） 。它提出的建议仍然有效，并且在本条目的最后也对此做了概述，但是这条建议现在远远没有之前那么重要了。这是因为几乎没有理由再使用 `wait` 和 `notify` 了。自从 Java 5 发行版本开始， Java 平台就提供了更高级的并发工具，它们可以完成以前必须在 `wait` 和 `notify` 上手写代码来完成的各项工作。 **既然正确地使用 wait 和 notify 比较困难，就应该用更高级的并发工具来代替。**

`java.util.concurrent` 中更高级的工具分成三类： Executor Framework 、并发集合（Concurrent Collection）以及同步器（Synchronizer） ，Executor Framework 只在第 80 条中简单地提到过，并发集合和同步器将在本条目中进行简单的阐述。

并发集合为标准的集合接口（如 `List` 、 `Queue` 和 `Map` ）提供了高性能的并发实现。为了提供高并发性，这些实现在内部自己管理同步（详见第 79 条） 。因此， **并发集合中不可能排除并发活动；将它锁定没有什么作用，只会使程序的速度变慢。**

因为无法排除并发集合中的并发活动，这意味着也无法自动地在并发集合中组成方法调用。因此，有些并发集合接口已经通过依赖状态的修改操作（state-dependent modify operation）进行了扩展，它将几个基本操作合并到了单个原子操作中。事实证明，这些操作在并发集合中已经够用，它们通过缺省方法（详见第 21 条）被加到了 Java 8 对应的集合接口中。

例如， `Map` 的 `putIfAbsent(key, value)` 方法，当键没有映射时会替它插入一个映射，并返回与键关联的前一个值，如果没有这样的值，则返回 null 。这样就能很容易地实现线程安全的标准 `Map` 了。例如，下面这个方法模拟了 `String.intern` 的行为：

```
// Concurrent canonicalizing map atop ConcurrentMap - not optimal
private static final ConcurrentMap<String, String> map = new ConcurrentHashMap<>();
public static String intern(String s) {
    String previousValue = map.putIfAbsent(s, s);
    return previousValue == null ? s : previousValue;
}
```

事实上，你还可以做得更好。 `ConcurrentHashMap` 对获取操作（如 get）进行了优化。因此，只有当 `get` 表明有必要的时候，才值得先调用 `get` ，再调用 `putIfAbsent` ：

```
// Concurrent canonicalizing map atop ConcurrentMap - faster!
public static String intern(String s) {
    String result = map.get(s);
    if (result == null) {
        result = map.putIfAbsent(s, s);
        if (result == null)
            result = s;
    }
    return result;
}
```

`ConcurrentHashMap` 除了提供卓越的并发性之外，速度也非常快。在我的机器上，上面这个优化过的 `intern` 方法比 `String.intern` 快了不止 6 倍（但是记住， `String.intern` 必须使用某种弱引用，避免随着时间的推移而发生内存泄漏）。并发集合导致同步的集合大多被废弃了。**比如， 应该优先使用 `ConcurrentHashMap` ，而不是使用 `Collections.synchronizedMap` 。** 只要用并发 Map 替换同步 Map ，就可以极大地提升并发应用程序的性能。

有些集合接口已经通过阻塞操作（blocking operation ）进行了扩展，它们会一直等待（或者阻塞）到可以成功执行为止。例如， `BlockingQueue` 扩展了 `Queue` 接口，并添加了包括 `take` 在内的几个方法，它从队列中删除并返回了头元素，如果队列为空，就等待。这样就允许将阻塞队列用于工作队列（work queue），也称作生产者—消费者队列 (producer-consumer queue），一个或者多个生产者线程（producer thread）在工作队列中添加工作项目，并且当工作项目可用时，一个或者多个消费者线程（consumer thread ）则从工作队列中取出队列并处理工作项目。不出所料，大多数

`ExecutorService` 实现（包括 `ThreadPoolExecutor`）都使用了一个 `BlockingQueue`（详见第 80 条）。

同步器（Synchronizer）是使线程能够等待另一个线程的对象，允许它们协调动作。最常用的同步器是 `CountDownLatch` 和 `Semaphore`。较不常用的是 `CyclicBarrier` 和 `Exchanger`。功能最强大的同步器是 `Phaser`。

倒计数锁存器（Countdown Latch）是一次性的障碍，允许一个或者多个线程等待一个或者多个其他线程来做某些事情。`CountDownLatch` 的唯一构造器带有一个 int 类型的参数，这个 int 参数是指允许所有在等待的线程被处理之前，必须在锁存器上调用 `countDown` 方法的次数。

要在这个简单的基本类型之上构建一些有用的东西，做起来是相当容易。例如，假设想要构建一个简单的框架，用来给一个动作的并发执行定时。这个框架中只包含单个方法，该方法带有一个执行该动作的 executor，一个并发级别（表示要并发执行该动作的次数），以及表示该动作的 runnable。所有的工作线程（worker thread）自身都准备好，要在 timer 线程启动时钟之前运行该动作。当最后一个工作线程准备好运行该动作时，timer 线程就「发射发令枪（fires the starting gun）」，同时允许工作线程执行该动作。一旦最后一个工作线程执行完该动作，timer 线程就立即停止计时。直接在 wait 和 notify 之上实现这个逻辑会很棍乱，而在 `CountDownLatch` 之上实现则相当简单：

```
// Simple framework for timing concurrent execution
public static long time(Executor executor, int concurrency,
                        Runnable action) throws InterruptedException {
    CountDownLatch ready = new CountDownLatch(concurrency);
    CountDownLatch start = new CountDownLatch(1);
    CountDownLatch done = new CountDownLatch(concurrency);
    for (int i = 0; i < concurrency; i++) {
        executor.execute(() -> {
            ready.countDown();
            // Tell timer we're ready
            try {
                start.await();
                // Wait till peers are ready
                action.run();
            }
            catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            finally {
                done.countDown();
                // Tell timer we're done
            }
        });
    }
    ready.await();
    // Wait for all workers to be ready
    long startNanos = System.nanoTime();
    start.countDown();
    // And they're off!
    done.await();
```

```
    // Wait for all workers to finish
    return System.nanoTime() - startNanos;
}
```

注意这个方法使用了三个倒计数锁存器。第一个是 `ready` ，工作线程用它来告诉 `timer` 线程它们已经准备好了。然后工作线程在第二个锁存器 `start` 上等待。当最后一个工作线程调用 `ready.countDown` 时， `timer` 线程记录下起始时间，并调用 `start.countDown` 允许所有的工作线程继续进行。然后 timer 线程在第三个锁存器 `done` 上等待，直到最后一个工作线程运行完该动作，并调用 `done.countDown` 。一旦调用这个， `timer` 线程就会苏醒过来，并记录下结束的时间。

还有一些细节值得注意。传递给 time 方法的 executor 必须允许创建至少与指定并发级别一样多的线程，否则这个测试就永远不会结束。这就是线程饥饿死锁（thread starvationdeadlock） [Goetz06, 8.1.1] 。如果工作线程捕捉到 `InterruptedException` ，就会利用习惯用法 `Thread.currentThread().interrupt()` 重新断言中断，并从它的 `run` 方法中返回。这样就允许 executor 在必要的时候处理中断，事实上也理应如此。注意，我们利用了 `System.nanoTime` 来给活动定时。对于间歇式的定时，始终应该优先使用 `System.nanoTime` ，而不是使用 `System.currentTimeMillis` 。因为 `System.nanoTime` 更准确，也更精确，它不受系统的实时时钟的调整所影响。最后，注意本例中的代码并不能进行准确的定时，除非 action 能完成一定量的工作，比如一秒或者一秒以上。众所周知，准确的微基准测试十分困难，最好在专门的框架如 jmh 的协助下进行 [JMH] 。

本条目仅仅触及了并发工具的一些皮毛。例如，前一个例子中的那三个倒计数锁存器其实可以用一个 `CyclicBarrier` 或者 `Phaser` 实例代替。这样得到的代码更加简洁，但是理解起来比较困难。虽然你始终应该优先使用并发工具，而不是使用 wait 方法和 notify 方法，但可能必须维护使用了 wait 方法和 notify 方法的遗留代码。wait 方法被用来使线程等待某个条件。它必须在同步区域内部被调用，这个同步区域将对象锁定在了调用 wait 方法的对象上。下面是使用 wait 方法的标准模式：

```
// The standard idiom for using the wait method
synchronized (obj) {
    while (<condition does not hold>)
    obj.wait();
    // (Releases lock, and reacquires on wakeup)
    ... // Perform action appropriate to condition
}
```

始终应该使用 wait 循环模式来调用 wait 方法；永远不要在循环之外调用 wait 方法。循环会在等待之前和之后对条件进行测试。

在等待之前测试条件，当条件已经成立时就跳过等待，这对于确保活性是必要的。如果条件已经成立，并且在线程等待之前， `notify` （或者 `notifyAll` ）方法已经被调用， 则无法保证该线程总会从等待中苏醒过来。

在等待之前测试条件，如果条件不成立的话继续等待，这对于确保安全性是必要的。当条件不成立的时候，如果线程继续执行，则可能会破坏被锁保护的约束关系。当条件不成立时，有以下理由可使一个线程苏醒过来：

- 另一个线程可能已经得到了锁，并且从一个线程调用 `notify` 方法那一刻起，到等待线程苏醒过来的这段时间中，得到锁的线程已经改变了受保护的状态。

- 条件并不成立，但是另一个线程可能意外地或恶意地调用了 `notify` 方法。在公有可访问的对象上等待，这些类实际上把自己暴露在了这种危险的境地中。公有可访问对象的同步方法中包含的 `wait` 方法都会出现这样的问题。
- 通知线程（ notifying thread ）在唤醒等待线程时可能会过于「慷慨」 。例如，即使只有某些等待线程的条件已经被满足，但是通知线程可能仍然调用 `notifyAll` 方法。
- 在没有通知的情况下，等待线程也可能（但很少）会苏醒过来。这被称为"伪唤醒"(spurious wakeup) [POSIX, 11.4.3.6.1; Java9-api] 。

一个相关的话题是，为了唤醒正在等待的线程，你应该使用 `notify` 方法还是 `notifyAll` 方法（回忆一下， notify 方法唤醒的是单个正在等待的线程，假设有这样的线程存在，而 `notifyAll` 方法唤醒的则是所有正在等待的线程） 。一种常见的说法是，应该始终使用 `notifyAll` 方法。这是合理而保守的建议。它总会产生正确的结果，因为它可以保证你将会唤醒所有需要被唤醒的线程。你可能也会唤醒其他一些线程，但是这不会影响程序的正确性。这些线程醒来之后，会检查它们正在等待的条件，如果发现条件并不满足，就会继续等待。

从优化的角度来看，如果处于等待状态的所有线程都在等待同一个条件，而每次只有一个线程可以从这个条件中被唤醒，那么你就应该选择调用 `notify` 方法，而不是 `notifyAll` 方法。

即使这些前提条件都满足，也许还是有理由使用 `notifyAll` 方法而不是 `notify` 方法。就好像把 `wait` 方法调用放在一个循环中，以避免在公有可访问对象上的意外或恶意的通知一样，与此类似，使用 `notifyAll` 方法代替 `notify` 方法可以避免来自不相关线程的意外或恶意的等待。否则，这样的等待会"吞掉"一个关键的通知，使真正的接收线程无限地等待下去。

简而言之，直接使用 `wait` 方法和 `notify` 方法就像用"并发汇编语言"进行编程一样，而 `java.util.concurrent` 则提供了更高级的语言。 **没有理由在新代码中使用 `wait` 方法和 `notify` 方法，即使有，也是极少的。** 如果你在维护使用 `wait` 方法和 `notify` 方法的代码，务必确保始终是利用标准的模式从 `while` 循环内部调用 `wait` 方法。一般情况下，应该优先使用 `notifyAll` 方法，而不是使用 `notify` 方法。如果使用 `notify` 方法，请一定要小心，以确保程序的活性。

82. 文档应包含线程安全属性

类在其方法并发使用时的行为是其与客户端约定的重要组成部分。如果你没有记录类在这一方面的行为，那么它的用户将被迫做出假设。如果这些假设是错误的，生成的程序可能缺少足够的同步（详见 78 条）或过度的同步（详见 79 条）。无论哪种情况，都可能导致严重的错误。

你可能听说过，可以通过在方法的文档中查找 `synchronized` 修饰符来判断方法是否线程安全。这个观点有好些方面是错误的。在正常操作中，Javadoc 的输出中没有包含同步修饰符，这是有原因的。方法声明中 `synchronized` 修饰符的存在是实现细节，而不是其 API 的一部分。**它不能可靠地表明方法是线程安全的。**

此外，声称 `synchronized` 修饰符的存在就足以记录线程安全性，这个观点是对线程安全性属性的误解，认为要么全有要么全无。实际上，线程安全有几个级别。**要启用安全的并发使用，类必须清楚地记录它支持的线程安全级别。** 下面的列表总结了线程安全级别。它并非详尽无遗，但涵盖以下常见情况：

- **不可变的** — 这个类的实例看起来是常量。不需要外部同步。示例包括 `String`、`Long` 和 `BigInteger`（详见第 17 条）。
- **无条件线程安全** — 该类的实例是可变的，但是该类具有足够的内部同步，因此无需任何外部同步即可并发地使用该类的实例。例如 `AtomicLong` 和 `ConcurrentHashMap`。
- **有条件的线程安全** — 与无条件线程安全类似，只是有些方法需要外部同步才能安全并发使用。示例包括 `Collections.synchronized` 包装器返回的集合，其迭代器需要外部同步。
- **非线程安全** — 该类的实例是可变的。要并发地使用它们，客户端必须使用外部同步来包围每个方法调用（或调用序列）。这样的例子包括通用的集合实现，例如 `ArrayList` 和 `HashMap`。
- **线程对立** — 即使每个方法调用都被外部同步包围，该类对于并发使用也是不安全的。线程对立通常是由于在不同步的情况下修改静态数据而导致的。没有人故意编写线程对立类；此类通常是由于没有考虑并发性而导致的。当发现类或方法与线程不相容时，通常将其修复或弃用。第 78 条中的 `generateSerialNumber` 方法在没有内部同步的情况下是线程对立的，如第 322 页所述。

这些类别（不包括线程对立类）大致对应于《Java Concurrency in Practice》中的线程安全注解，分别为 `Immutable`、`ThreadSafe` 和 `NotThreadSafe` [Goetz06, Appendix A]。上面分类中的无条件线程安全和有条件的线程安全都包含在 `ThreadSafe` 注解中。

在文档中记录一个有条件的线程安全类需要小心。你必须指出哪些调用序列需要外部同步，以及执行这些序列必须获得哪些锁（在极少数情况下是锁）。通常是实例本身的锁，但也有例外。例如，`Collections.synchronizedMap` 的文档提到：

It is imperative that the user manually synchronize on the returned map when iterating over any of its collection views:

当用户遍历其集合视图时，必须手动同步返回的 Map：

```
Map<K, V> m = Collections.synchronizedMap(new HashMap<>());
Set<K> s = m.keySet(); // Needn't be in synchronized block
...
synchronized(m) { // Synchronizing on m, not s!
    for (K key : s)
        key.f();
}
```

不遵循这个建议可能会导致不确定的行为。

类的线程安全的描述通常属于该类的文档注释，但是具有特殊线程安全属性的方法应该在它们自己的文档注释中描述这些属性。没有必要记录枚举类型的不变性。除非从返回类型可以明显看出，否则静态工厂必须记录返回对象的线程安全性，正如 `Collections.synchronizedMap` 所演示的那样。

当一个类使用公共可访问锁时，它允许客户端自动执行一系列方法调用，但是这种灵活性是有代价的。它与诸如 `ConcurrentHashMap` 之类的并发集合所使用的高性能内部并发控制不兼容。此外，客户端可以通过长时间持有可公开访问的锁来发起拒绝服务攻击。这可以是无意的，也可以是有意的。

为了防止这种拒绝服务攻击，你可以使用一个私有锁对象，而不是使用同步方法（隐含一个公共可访问的锁）：

因为私有锁对象在类之外是不可访问的，所以客户端不可能干扰对象的同步。实际上，我们通过将锁对象封装在它同步的对象中，是在应用条目 15 的建议。

注意，lock 字段被声明为 final。这可以防止你无意中更改它的内容，这可能导致灾难性的非同步访问（详见 78 条）。我们正在应用条目 17 的建议，最小化锁字段的可变性。**Lock 字段应该始终声明为 final。** 无论使用普通的监视器锁（如上所示）还是 `java.util.concurrent` 包中的锁，都是这样。

私有锁对象用法只能在无条件的线程安全类上使用。有条件的线程安全类不能使用这种用法，因为它们必须在文档中记录，在执行某些方法调用序列时要获取哪些锁。

私有锁对象用法特别适合为继承而设计的类（详见第 19 条）。如果这样一个类要使用它的实例进行锁定，那么子类很容易在无意中干扰基类的操作，反之亦然。通过为不同的目的使用相同的锁，子类和基类最终可能「踩到对方的脚趾头」。这不仅仅是一个理论问题，它就发生在 `Thread` 类中 [Bloch05, Puzzle 77]。

总之，每个类都应该措辞严谨的描述或使用线程安全注解清楚地记录其线程安全属性。`synchronized` 修饰符在文档中没有任何作用。有条件的线程安全类必须记录哪些方法调用序列需要外部同步，以及在执行这些序列时需要获取哪些锁。如果你编写一个无条件线程安全的类，请考虑使用一个私有锁对象来代替同步方法。这将保护你免受客户端和子类的同步干扰，并为你提供更大的灵活性，以便在后续的版本中采用复杂的并发控制方式。

83. 明智审慎的使用延迟初始化

延迟初始化是延迟字段的初始化，直到需要它的值。如果不需要该值，则不会初始化字段。这种技术既适用于静态字段，也适用于实例字段。虽然延迟初始化主要是一种优化，但是它也可以用于破坏类中的有害循环和实例初始化 [Bloch05, Puzzle 51]。

与大多数优化一样，延迟初始化的最佳建议是「除非需要，否则不要这样做」（详见第 67 条）。延迟初始化是一把双刃剑。它降低了初始化类或创建实例的成本，代价是增加了访问延迟初始化字段的成本。根据这些字段中最终需要初始化的部分、初始化它们的开销以及初始化后访问每个字段的频率，延迟初始化实际上会损害性能（就像许多「优化」一样）。

延迟初始化也有它的用途。如果一个字段只在类的一小部分实例上访问，并且初始化该字段的代价很高，那么延迟初始化可能是值得的。唯一确定的方法是以使用和不使用延迟初始化的效果对比来度量类的性能。

在存在多个线程的情况下，使用延迟初始化很棘手。如果两个或多个线程共享一个延迟初始化的字段，那么必须使用某种形式的同步，否则会导致严重的错误（详见第 78 条）。本条目讨论的所有初始化技术都是线程安全的。

在大多数情况下，常规初始化优于延迟初始化。 下面是一个使用常规初始化的实例字段的典型声明。注意 final 修饰符的使用（详见第 17 条）：

如果您使用延迟初始化来取代初始化的循环（circularity），请使用同步访问器，因为它是最简单、最清晰的替代方法：

```
// Lazy initialization of instance field - synchronized accessor
private FieldType field;
private synchronized FieldType getField() {
    if (field == null)
        field = computeFieldValue();
    return field;
}
```

这两种习惯用法（使用同步访问器进行常规初始化和延迟初始化）在应用于静态字段时都没有改变，只是在字段和访问器声明中添加了 static 修饰符。

如果需要在静态字段上使用延迟初始化来提高性能，使用 lazy initialization holder class 模式。 这个用法可保证一个类在使用之前不会被初始化 [JLS, 12.4.1]。它是这样的：

```
// Lazy initialization holder class idiom for static fields
private static class FieldHolder {
    static final FieldType field = computeFieldValue();
}
private static FieldType getField() { return FieldHolder.field; }
```

第一次调用 getField 时，它执行 FieldHolder.field，导致初始化 FieldHolder 类。这个习惯用法的优点是 getField 方法不是同步的，只执行字段访问，所以延迟初始化实际上不会增加访问成本。典型的 VM 只会同步字段访问来初始化类。初始化类之后，VM 会对代码进行修补，这样对字段的后续访问就不会涉及任何测试或同步。

如果需要使用延迟初始化来提高实例字段的性能，请使用双重检查模式。这个模式避免了初始化后访问字段时的锁定成本（详见第 79 条）。这个模式背后的思想是两次检查字段的值（因此得名 double check）：一次没有锁定，然后，如果字段没有初始化，第二次使用锁定。只有当第二次检查指示字段未初始化时，调用才初始化字段。由于初始化字段后没有锁定，因此将字段声明为 volatile 非常重要（详见第 78 条）。下面是这个模式的示例：

```
// Double-check idiom for lazy initialization of instance fields
private volatile FieldType field;
private FieldType getField() {
    FieldType result = field;
    if (result == null) { // First check (no locking)
        synchronized(this) {
            if (field == null) // Second check (with locking)
                field = result = computeFieldValue();
        }
    }
    return result;
}
```

这段代码可能看起来有点复杂。特别是不清楚是否需要局部变量（result）。该变量的作用是确保 field 在已经初始化的情况下只读取一次。

虽然不是严格必需的，但这可能会提高性能，而且与低级并发编程相比，这更优雅。在我的机器上，上述方法的速度大约是没有局部变量版本的 1.4 倍。虽然您也可以将双重检查模式应用于静态字段，但是没有理由这样做：the lazy initialization holder class idiom 是更好的选择。

双重检查模式的两个变体值得注意。有时候，您可能需要延迟初始化一个实例字段，该字段可以容忍重复初始化。如果您发现自己处于这种情况，您可以使用双重检查模式的变体来避免第二个检查。毫无疑问，这就是所谓的「单检查」模式。它是这样的。注意，field 仍然声明为 volatile：

```
// Single-check idiom - can cause repeated initialization!
private volatile FieldType field;
private FieldType getField() {
    FieldType result = field;
    if (result == null)
        field = result = computeFieldValue();
    return result;
}
```

本条目中讨论的所有初始化技术都适用于基本字段和对象引用字段。当双检查或单检查模式应用于数值基本类型字段时，将根据 0（数值基本类型变量的默认值）而不是 null 检查字段的值。

如果您不关心每个线程是否都会重新计算字段的值，并且字段的类型是 long 或 double 之外的基本类型，那么您可以选择在单检查模式中从字段声明中删除 volatile 修饰符。这种变体称为原生单检查模式。它加快了某些架构上的字段访问速度，代价是需要额外的初始化（每个访问该字段的线程最多需要一个初始化）。这绝对是一种奇特的技术，不是日常使用的。

总之，您应该正常初始化大多数字段，而不是延迟初始化。如果必须延迟初始化字段以实现性能目标或为了破坏有害的初始化循环，则使用适当的延迟初始化技术。对于字段，使用双重检查模式；对于静态字段，则应该使用the lazy initialization holder class idiom。例如，可以容忍重复初始化的实例字段，您还可以考虑单检查模式。

84. 不要依赖线程调度器

当许多线程可以运行时，线程调度器决定哪些线程可以运行以及运行多长时间。任何合理的操作系统都会尝试公平地做出这个决定，但是策略可能会有所不同。因此，编写良好的程序不应该依赖于此策略的细节。**任何依赖线程调度器来保证正确性或性能的程序都可能是不可移植的。**

编写健壮、响应快、可移植程序的最佳方法是确保可运行线程的平均数量不显著大于处理器的数量。这使得线程调度器几乎没有选择：它只运行可运行线程，直到它们不再可运行为止。即使在完全不同的线程调度策略下，程序的行为也没有太大的变化。注意，可运行线程的数量与线程总数不相同，后者可能更高。正在等待的线程不可运行。

保持可运行线程数量低的主要技术是让每个线程做一些有用的工作，然后等待更多的工作。**如果线程没有做有用的工作，它们就不应该运行。** 对于 Executor 框架（详见第 80 条），这意味着适当调整线程池的大小 [Goetz06, 8.2]，并保持任务短小（但不要太短），否则分派的开销依然会损害性能。

线程不应该处于忙等待状态（busy-wait），而应该反复检查一个共享对象，等待它的状态发生变化。除了使程序容易受到线程调度器变化无常的影响之外，繁忙等待还大大增加了处理器的负载，还影响其他人完成工作。作为反面的极端例子，考虑一下 `CountDownLatch` 的不正确的重构实现：

```
// Awful CountDownLatch implementation - busy-waits incessantly!
public class SlowCountDownLatch {

    private int count;

    public SlowCountDownLatch(int count) {
        if (count < 0)
            throw new IllegalArgumentException(count + " < 0");
        this.count = count;
    }

    public void await() {
        while (true) {
            synchronized(this) {
                if (count == 0)
                return;
            }
        }
    }

    public synchronized void countDown() {
        if (count != 0)
            count--;
    }
}
```

在我的机器上，当 1000 个线程等待一个锁存器时，`SlowCountDownLatch` 的速度大约是 Java 的 `CountDownLatch` 的 10 倍。虽然这个例子看起来有点牵强，但是具有一个或多个不必要运行的线程的系统并不少见。性能和可移植性可能会受到影响。

当面对一个几乎不能工作的程序时，而原因是由于某些线程相对于其他线程没有获得足够的 CPU 时间，那么 **通过调用 `Thread.yield` 来「修复」程序** 你也许能勉强让程序运行起来，但它是不可移植的。在一个 JVM 实现上提高性能的相同的 `yield` 调用，在第二个 JVM 实现上可能会使性能变差，而在第三个 JVM 实现上可能没有任何影响。 `Thread.yield` **没有可测试的语义。** 更好的做法是重构应用程序，以减少并发运行线程的数量。

一个相关的技术是调整线程优先级，类似的警告也适用于此技术，即，线程优先级是 Java 中最不可移植的特性之一。通过调整线程优先级来调优应用程序的响应性并非不合理，但很少情况下是必要的，而且不可移植。试图通过调整线程优先级来解决严重的活性问题是不合理的。在找到并修复潜在原因之前，问题很可能会再次出现。

总之，不要依赖线程调度器来判断程序的正确性。生成的程序既不健壮也不可移植。因此，不要依赖 `Thread.yield` 或线程优先级。这些工具只是对调度器的提示。线程优先级可以少量地用于提高已经工作的程序的服务质量，但绝不应该用于「修复」几乎不能工作的程序。

85. 优先选择 Java 序列化的替代方案

当序列化在 1997 年添加到 Java 中时，它被认为有一定的风险。这种方法曾在研究语言（Modula-3）中尝试过，但从未在生产语言中使用过。虽然程序员不费什么力气就能实现分布式对象，这一点很吸引人，但代价也不小，如：不可见的构造函数、API 与实现之间模糊的界线，还可能会出现正确性、性能、安全性和维护方面的问题。支持者认为收益大于风险，但历史证明并非如此。

在本书之前的版本中描述的安全问题，和人们担心的一样严重。21 世纪初仅停留在讨论的漏洞在接下来的 10 年间变成了真实严重的漏洞，其中最著名的包括 2016 年 11 月对旧金山大都会运输署市政铁路（SFMTA Muni）的勒索软件攻击，导致整个收费系统关闭了两天 [Gallagher16]。

序列化的一个根本问题是它的可攻击范围太大，且难以保护，而且问题还在不断增多：通过调用 ObjectInputStream 上的 readObject 方法反序列化对象图。这个方法本质上是一个神奇的构造函数，可以用来实例化类路径上几乎任何类型的对象，只要该类型实现 Serializable 接口。在反序列化字节流的过程中，此方法可以执行来自任何这些类型的代码，因此所有这些类型的代码都在攻击范围内。

攻击可涉及 Java 平台库、第三方库（如 Apache Commons collection）和应用程序本身中的类。即使坚持履行实践了所有相关的最佳建议，并成功地编写了不受攻击的可序列化类，应用程序仍然可能是脆弱的。引用 CERT 协调中心技术经理 Robert Seacord 的话：

> Java 反序列化是一个明显且真实的危险源，因为它被应用程序直接和间接地广泛使用，比如 RMI（远程方法调用）、JMX（Java 管理扩展）和 JMS（Java 消息传递系统）。不可信流的反序列化可能导致远程代码执行（RCE）、拒绝服务（DoS）和一系列其他攻击。应用程序很容易受到这些攻击，即使它们本身没有错误 [Seacord17]。

攻击者和安全研究人员研究 Java 库和常用的第三方库中的可序列化类型，寻找在反序列化过程中调用的潜在危险活动的方法称为 gadget。多个小工具可以同时使用，形成一个小工具链。偶尔会发现一个小部件链，它的功能足够强大，允许攻击者在底层硬件上执行任意的本机代码，允许提交精心设计的字节流进行反序列化。这正是 SFMTA Muni 袭击中发生的事情。这次袭击并不是孤立的。不仅已经存在，而且还会有更多。

不使用任何 gadget，你都可以通过对需要很长时间才能反序列化的短流进行反序列化，轻松地发起拒绝服务攻击。这种流被称为反序列化炸弹 [Svoboda16]。下面是 Wouter Coekaerts 的一个例子，它只使用哈希集和字符串 [Coekaerts15]：

```
static byte[] bomb() {
    Set<Object> root = new HashSet<>();
    Set<Object> s1 = root;
    Set<Object> s2 = new HashSet<>();
    for (int i = 0; i < 100; i++) {
        Set<Object> t1 = new HashSet<>();
        Set<Object> t2 = new HashSet<>();
        t1.add("foo"); // Make t1 unequal to t2
        s1.add(t1); s1.add(t2);
        s2.add(t1); s2.add(t2);
        s1 = t1;
        s2 = t2;
    }
```

```
        return serialize(root); // Method omitted for brevity
    }
```

对象图由 201 个 HashSet 实例组成，每个实例包含 3 个或更少的对象引用。整个流的长度为 5744 字节，但是在你对其进行反序列化之前，资源就已经耗尽了。问题在于，反序列化 HashSet 实例需要计算其元素的哈希码。根哈希集的 2 个元素本身就是包含 2 个哈希集元素的哈希集，每个哈希集元素包含 2 个哈希集元素，以此类推，深度为 100。因此，反序列化 Set 会导致 hashCode 方法被调用超过 2100 次。除了反序列化会持续很长时间之外，反序列化器没有任何错误的迹象。生成的对象很少，并且堆栈深度是有界的。

那么你能做些什么来抵御这些问题呢？当你反序列化一个你不信任的字节流时，你就会受到攻击。**避免序列化利用的最好方法是永远不要反序列化任何东西。** 用 1983 年电影《战争游戏》（WarGames）中名为约书亚（Joshua）的电脑的话来说，「唯一的制胜绝招就是不玩。」**没有理由在你编写的任何新系统中使用 Java 序列化。** 还有其他一些机制可以在对象和字节序列之间进行转换，从而避免了 Java 序列化的许多危险，同时还提供了许多优势，比如跨平台支持、高性能、大量工具和广泛的专家社区。在本书中，我们将这些机制称为跨平台结构数据表示。虽然其他人有时将它们称为序列化系统，但本书避免使用这种说法，以免与 Java 序列化混淆。

以上所述技术的共同点是它们比 Java 序列化简单得多。它们不支持任意对象图的自动序列化和反序列化。相反，它们支持简单的结构化数据对象，由一组「属性-值」对组成。只有少数基本数据类型和数组数据类型得到支持。事实证明，这个简单的抽象足以构建功能极其强大的分布式系统，而且足够简单，可以避免 Java 序列化从一开始就存在的严重问题。

领先的跨平台结构化数据表示是 JSON 和 Protocol Buffers，也称为 protobuf。JSON 由 Douglas Crockford 设计用于浏览器与服务器通信，Protocol Buffers 由谷歌设计用于在其服务器之间存储和交换结构化数据。尽管这些技术有时被称为「中性语言」，但 JSON 最初是为 JavaScript 开发的，而 protobuf 是为 c++ 开发的；这两种技术都保留了其起源的痕迹。

JSON 和 protobuf 之间最显著的区别是 JSON 是基于文本的，并且是人类可读的，而 protobuf 是二进制的，但效率更高；JSON 是一种专门的数据表示，而 protobuf 提供模式（类型）来记录和执行适当的用法。虽然 protobuf 比 JSON 更有效，但是 JSON 对于基于文本的表示非常有效。虽然 protobuf 是一种二进制表示，但它确实提供了另一种文本表示，可用于需要具备人类可读性的场景（pbtxt）。

如果你不能完全避免 Java 序列化，可能是因为你需要在遗留系统环境中工作，那么你的下一个最佳选择是 **永远不要反序列化不可信的数据。** 特别要注意，你不应该接受来自不可信来源的 RMI 流量。Java 的官方安全编码指南说：「反序列化不可信的数据本质上是危险的，应该避免。」这句话是用大号、粗体、斜体和红色字体设置的，它是整个文档中唯一得到这种格式处理的文本。[Java-secure]

如果无法避免序列化，并且不能绝对确定反序列化数据的安全性，那么可以使用 Java 9 中添加的对象反序列化筛选，并将其移植到早期版本（java.io.ObjectInputFilter）。该工具允许你指定一个过滤器，该过滤器在反序列化数据流之前应用于数据流。它在类粒度上运行，允许你接受或拒绝某些类。默认接受所有类，并拒绝已知潜在危险类的列表称为黑名单；在默认情况下拒绝其他类，并接受假定安全的类的列表称为白名单。**优先选择白名单而不是黑名单，** 因为黑名单只保护你免受已知的威胁。一个名为 Serial Whitelist Application Trainer（SWAT）的工具可用于为你的应用程序自动准备一个白名单 [Schneider16]。过滤工具还将保护你免受过度内存使用和过于深入的对象图的影响，但它不能保护你免受如上面所示的序列化炸弹的影响。

不幸的是，序列化在 Java 生态系统中仍然很普遍。如果你正在维护一个基于 Java 序列化的系统，请认真考虑迁移到跨平台的结构化数据，尽管这可能是一项耗时的工作。实际上，你可能仍然需要编写或维护一个可序列化的类。编写一个正确、安全、高效的可序列化类需要非常小心。本章的其余部分将提供何时以及如何进行此操作的建议。

总之，序列化是危险的，应该避免。如果你从头开始设计一个系统，可以使用跨平台的结构化数据，如 JSON 或 protobuf。不要反序列化不可信的数据。如果必须这样做，请使用对象反序列化过滤，但要注意，它不能保证阻止所有攻击。避免编写可序列化的类。如果你必须这样做，一定要非常小心。

86. 非常谨慎地实现 Serializable

使类的实例可序列化非常简单，只需实现 `Serializable` 接口即可。因为这很容易做到，所以有一个普遍的误解，认为序列化只需要程序员付出很少的努力。而事实上要复杂得多。虽然使类可序列化的即时代价可以忽略不计，但长期代价通常是巨大的。

实现 `Serializable` 接口的一个主要代价是，一旦类的实现被发布，它就会降低更改该类实现的灵活性。 当类实现 `Serializable` 时，其字节流编码（或序列化形式）成为其导出 API 的一部分。一旦广泛分发了一个类，通常就需要永远支持序列化的形式，就像需要支持导出 API 的所有其他部分一样。如果你不努力设计自定义序列化形式，而只是接受默认形式，则序列化形式将永远绑定在类的原始内部实现上。换句话说，如果你接受默认的序列化形式，类中私有的包以及私有实例字段将成为其导出 API 的一部分，此时最小化字段作用域（详见第 15 条）作为信息隐藏的工具，将失去其有效性。

如果你接受默认的序列化形式，然后更改了类的内部实现，则会导致与序列化形式不兼容。试图使用类的旧版本序列化实例，再使用新版本反序列化实例的客户端（反之亦然）程序将会失败。当然，可以在维护原始序列化形式的同时更改内部实现（使用 `ObjectOutputStream.putFields` 或 `ObjectInputStream.readFields`），但这可能会很困难，并在源代码中留下明显的缺陷。如果你选择使类可序列化，你应该仔细设计一个高质量的序列化形式，以便长期使用（详见第 87 和 90 条）。这样做会增加开发的初始成本，但是这样做是值得的。即使是设计良好的序列化形式，也会限制类的演化；而设计不良的序列化形式，则可能会造成严重后果。

可序列化会使类的演变受到限制，施加这种约束的一个简单示例涉及流的唯一标识符，通常称其为串行版本 UID。每个可序列化的类都有一个与之关联的唯一标识符。如果你没有通过声明一个名为 `serialVersionUID` 的静态 final long 字段来指定这个标识符，那么系统将在运行时对类应用加密散列函数（SHA-1）自动生成它。这个值受到类的名称、实现的接口及其大多数成员（包括编译器生成的合成成员）的影响。如果你更改了其中任何一项，例如，通过添加一个临时的方法，生成的序列版本 UID 就会更改。如果你未能声明序列版本 UID，兼容性将被破坏，从而在运行时导致 `InvalidClassException`。

实现 `Serializable` 接口的第二个代价是，增加了出现 bug 和安全漏洞的可能性（详见第 85 条）。 通常，对象是用构造函数创建的；序列化是一种用于创建对象的超语言机制。无论你接受默认行为还是无视它，反序列化都是一个「隐藏构造函数」，其他构造函数具有的所有问题它都有。由于没有与反序列化关联的显式构造函数，因此很容易忘记必须让它能够保证所有的不变量都是由构造函数建立的，并且不允许攻击者访问正在构造的对象内部。依赖于默认的反序列化机制，会让对象轻易地遭受不变性破坏和非法访问（详见第 88 条）。

实现 `Serializable` 接口的第三个代价是，它增加了与发布类的新版本相关的测试负担。 当一个可序列化的类被修改时，重要的是检查是否可以在新版本中序列化一个实例，并在旧版本中反序列化它，反之亦然。因此，所需的测试量与可序列化类的数量及版本的数量成正比，工作量可能很大。你必须确保「序列化-反序列化」过程成功，并确保它生成原始对象的无差错副本。如果在第一次编写类时精心设计了自定义序列化形式，那么测试的工作量就会减少（详见第 87 和 90 条）。

实现 `Serializable` 接口并不是一个轻松的决定。 如果一个类要参与一个框架，该框架依赖于 Java 序列化来进行对象传输或持久化，这对于类来说实现 `Serializable` 接口就是非常重要的。此外，如果类 A 要成为另一个类 B 的一个组件，类 B 必须实现 `Serializable` 接口，若类 A 可序列化，它就会更易于被使用。然而，与实现 `Serializable` 相关的代价很多。每次设计一个类时，都要权衡利弊。历史上，像 `BigInteger` 和 `Instant` 这样的值类实现了 `Serializable` 接口，集合类也实现了 `Serializable` 接口。表示活动实体（如线程池）的类很少情况适合实现 `Serializable` 接口。

为继承而设计的类（详见第 19 条）很少情况适合实现 `Serializable` 接口，接口也很少情况适合扩展它。 违反此规则会给扩展类或实现接口的任何人带来很大的负担。有时，违反规则是恰当的。例如，如果一个类或接口的存在主要是为了参与一个要求所有参与者都实现 `Serializable` 接口的框架，那么类或接口实现或扩展 Serializable 可能是有意义的。

在为了继承而设计的类中，`Throwable` 类和 `Component` 类都实现了 `Serializable` 接口。正是因为 `Throwable` 实现了 `Serializable` 接口，RMI 可以将异常从服务器发送到客户端；`Component` 类实现了 `Serializable` 接口，因此可以发送、保存和恢复 GUI，但即使在 Swing 和 AWT 的鼎盛时期，这个工具在实践中也很少使用。

如果你实现了一个带有实例字段的类，它同时是可序列化和可扩展的，那么需要注意几个风险。如果实例字段值上有任何不变量，关键是要防止子类覆盖 finalize 方法，可以通过覆盖 finalize 并声明它为 final 来做到。最后，如果类的实例字段初始化为默认值（整数类型为 0，布尔值为 false，对象引用类型为 null），那么必须添加 `readObjectNoData` 方法：

```
// readObjectNoData for stateful extendable serializable classes
private void readObjectNoData() throws InvalidObjectException {
    throw new InvalidObjectException("Stream data required");
}
```

这个方法是在 Java 4 中添加的，涉及将可序列化超类添加到现有可序列化类 [Serialization, 3.5] 的特殊情况。

关于不实现 `Serializable` 的决定，有一个警告。如果为继承而设计的类不可序列化，则可能需要额外的工作来编写可序列化的子类。子类的常规反序列化，要求超类具有可访问的无参数构造函数 [Serialization, 1.10]。如果不提供这样的构造函数，子类将被迫使用序列化代理模式（详见第 90 条）。

内部类（详见第 24 条）不应该实现 `Serializable`。 它们使用编译器生成的合成字段存储对外围实例的引用，并存储来自外围的局部变量的值。这些字段与类定义的对应关系，就和没有指定匿名类和局部类的名称一样。因此，内部类的默认序列化形式是不确定的。但是，静态成员类可以实现 `Serializable` 接口。

87. 考虑使用自定义的序列化形式

当你在时间紧迫的情况下编写类时，通常应该将精力集中在设计最佳的 API 上。有时，这意味着发布一个「一次性」实现，你也知道在将来的版本中会替换它。通常这不是一个问题，但是如果类实现 `Serializable` 接口并使用默认的序列化形式，你将永远无法完全摆脱这个「一次性」的实现。它将永远影响序列化的形式。这不仅仅是一个理论问题。这种情况发生在 Java 库中的几个类上，包括 `BigInteger` 。

在没有考虑默认序列化形式是否合适之前，不要接受它。 接受默认的序列化形式应该是一个三思而后行的决定，即从灵活性、性能和正确性的角度综合来看，这种编码是合理的。一般来说，设计自定义序列化形式时，只有与默认序列化形式所选择的编码在很大程度上相同时，才应该接受默认的序列化形式。

对象的默认序列化形式，相对于它的物理表示法而言是一种比较有效的编码形式。换句话说，它描述了对象中包含的数据以及从该对象可以访问的每个对象的数据。它还描述了所有这些对象相互关联的拓扑结构。理想的对象序列化形式只包含对象所表示的逻辑数据。它独立于物理表征。

如果对象的物理表示与其逻辑内容相同，则默认的序列化形式可能是合适的。 例如，默认的序列化形式对于下面的类来说是合理的，它简单地表示一个人的名字：

```
// Good candidate for default serialized form
public class Name implements Serializable {
    /**
     * Last name. Must be non-null.
     * @serial
     */
    private final String lastName;

    /**
     * First name. Must be non-null.
     * @serial
     */
    private final String firstName;

    /**
     * Middle name, or null if there is none.
     * @serial
     */
    private final String middleName;
    ... // Remainder omitted
}
```

从逻辑上讲，名字由三个字符串组成，分别表示姓、名和中间名。Name 的实例字段精确地反映了这个逻辑内容。

即使你认为默认的序列化形式是合适的，你通常也必须提供 readObject 方法来确保不变性和安全性。 对于 `Name` 类而言，`readObject` 方法必须确保字段 lastName 和 firstName 是非空的。第 88 条和第 90 条详细讨论了这个问题。

注意，虽然 lastName、firstName 和 middleName 字段是私有的，但是它们都有文档注释。这是因为这些私有字段定义了一个公共 API，它是类的序列化形式，并且必须对这个公共 API 进行文档化。`@serial` 标记的存在告诉 Javadoc 将此文档放在一个特殊的页面上，该页面记录序列化的形式。

与 `Name` 类不同，考虑下面的类，它是另一个极端。它表示一个字符串列表（使用标准 List 实现可能更好，但此时暂不这么做）：

```
// Awful candidate for default serialized form
public final class StringList implements Serializable {
    private int size = 0;
    private Entry head = null;
    private static class Entry implements Serializable {
        String data;
        Entry next;
        Entry previous;
    }
    ... // Remainder omitted
}
```

从逻辑上讲，这个类表示字符串序列。在物理上，它将序列表示为双向链表。如果接受默认的序列化形式，该序列化形式将不遗余力地镜像出链表中的所有项，以及这些项之间的所有双向链接。

当对象的物理表示与其逻辑数据内容有很大差异时，使用默认的序列化形式有四个缺点：

- **它将导出的 API 永久地绑定到当前的内部实现。** 在上面的例子中，私有 `StringList.Entry` 类成为公共 API 的一部分。如果在将来的版本中更改了实现，`StringList` 类仍然需要接受链表形式的输出，并产生链表形式的输出。这个类永远也摆脱不掉处理链表项所需要的所有代码，即使不再使用链表作为内部数据结构。
- **它会占用过多的空间。** 在上面的示例中，序列化的形式不必要地表示链表中的每个条目和所有链接关系。这些链表项以及链接只不过是实现细节，不值得记录在序列化形式中。因为这样的序列化形式过于庞大，将其写入磁盘或通过网络发送将非常慢。
- **它会消耗过多的时间。** 序列化逻辑不知道对象图的拓扑结构，因此必须遍历开销很大的图。在上面的例子中，只要遵循 next 的引用就足够了。
- **它可能导致堆栈溢出。** 默认的序列化过程执行对象图的递归遍历，即使对于中等规模的对象图，这也可能导致堆栈溢出。用 1000-1800 个元素序列化 `StringList` 实例会在我的机器上生成一个 `StackOverflowError`。令人惊讶的是，序列化导致堆栈溢出的最小列表大小因运行而异（在我的机器上）。显示此问题的最小列表大小可能取决于平台实现和命令行标志；有些实现可能根本没有这个问题。

`StringList` 的合理序列化形式就是列表中的字符串数量，然后是字符串本身。这构成了由 `StringList` 表示的逻辑数据，去掉了其物理表示的细节。下面是修改后的 `StringList` 版本，带有实现此序列化形式的 `writeObject` 和 `readObject` 方法。提醒一下，`transient` 修饰符表示要从类的默认序列化表单中省略该实例字段：

```
// StringList with a reasonable custom serialized form
public final class StringList implements Serializable {
    private transient int size = 0;
    private transient Entry head = null;
```

```
    // No longer Serializable!

    private static class Entry {
        String data;
        Entry next;
        Entry previous;
    }
    // Appends the specified string to the list
    public final void add(String s) { ... }

    /**
     * Serialize this {@code StringList} instance.
     *
     * @serialData The size of the list (the number of strings
     * it contains) is emitted ({@code int}), followed by all of
     * its elements (each a {@code String}), in the proper
     * sequence.
     */
    private void writeObject(ObjectOutputStream s) throws IOException {
        s.defaultWriteObject();
        s.writeInt(size);
        // Write out all elements in the proper order.
        for (Entry e = head; e != null; e = e.next)
            s.writeObject(e.data);
    }

    private void readObject(ObjectInputStream s) throws IOException,
ClassNotFoundException {
        s.defaultReadObject();
        int numElements = s.readInt();
        // Read in all elements and insert them in list
        for (int i = 0; i < numElements; i++)
            add((String) s.readObject());
    }

    ... // Remainder omitted
}
```

`writeObject` 做的第一件事是调用 `defaultWriteObject`，`readObject` 做的第一件事是调用 `defaultReadObject`，即使 `StringList` 的所有字段都是 `transient` 的。你可能听说过，如果一个类的所有实例字段都是 `transient` 的，那么你可以不调用 `defaultWriteObject` 和 `defaultReadObject`，但是序列化规范要求你无论如何都要调用它们。这些调用的存在使得在以后的版本中添加非瞬态实例字段成为可能，同时保留了向后和向前兼容性。如果实例在较晚的版本中序列化，在较早的版本中反序列化，则会忽略添加的字段。如果早期版本的 `readObject` 方法调用 `defaultReadObject` 失败，反序列化将失败，并出现 `StreamCorruptedException`。

注意，`writeObject` 方法有一个文档注释，即使它是私有的。这类似于 Name 类中私有字段的文档注释。这个私有方法定义了一个公共 API，它是序列化的形式，并且应该对该公共 API 进行文档化。与字段的 `@serial` 标记一样，方法的 `@serialData` 标记告诉 Javadoc 实用工具将此文档放在序列化形式页面上。

为了给前面的性能讨论提供一定的伸缩性，如果平均字符串长度是 10 个字符，那么经过修改的 `StringList` 的序列化形式占用的空间大约是原始字符串序列化形式的一半。在我的机器上，序列化修订后的 `StringList` 的速度是序列化原始版本的两倍多，列表长度为 10。最后，在修改后的形式中没有堆栈溢出问题，因此对于可序列化的 `StringList` 的大小没有实际的上限。

虽然默认的序列化形式对 `StringList` 不好，但是对于某些类来说，情况会更糟。对于 `StringList`，默认的序列化形式是不灵活的，并且执行得很糟糕，但是它是正确的，因为序列化和反序列化 `StringList` 实例会生成原始对象的无差错副本，而所有不变量都是完整的。对于任何不变量绑定到特定于实现的细节的对象，情况并非如此。

例如，考虑哈希表的情况。物理表示是包含「键-值」项的哈希桶序列。一个项所在的桶是其键的散列代码的函数，通常情况下，不能保证从一个实现到另一个实现是相同的。事实上，它甚至不能保证每次运行都是相同的。因此，接受哈希表的默认序列化形式将构成严重的 bug。对哈希表进行序列化和反序列化可能会产生一个不变量严重损坏的对象。

无论你是否接受默认的序列化形式，当调用 `defaultWriteObject` 方法时，没有标记为 `transient` 的每个实例字段都会被序列化。因此，可以声明为 `transient` 的每个实例字段都应该做这个声明。这包括派生字段，其值可以从主数据字段（如缓存的哈希值）计算。它还包括一些字段，这些字段的值与 JVM 的一个特定运行相关联，比如表示指向本机数据结构指针的 long 字段。**在决定使字段非 `transient` 之前，请确信它的值是对象逻辑状态的一部分。** 如果使用自定义序列化表单，大多数或所有实例字段都应该标记为 `transient`，如上面的 `StringList` 示例所示。

如果使用默认的序列化形式，并且标记了一个或多个字段为 `transient`，请记住，当反序列化实例时，这些字段将初始化为默认值：对象引用字段为 null，数字基本类型字段为 0，布尔字段为 false [JLS, 4.12.5]。如果这些值对于任何 `transient` 字段都是不可接受的，则必须提供一个 readObject 方法，该方法调用 `defaultReadObject` 方法，然后将 `transient` 字段恢复为可接受的值（详见第 88 条）。或者，可以采用延迟初始化（详见第 83 条），在第一次使用这些字段时初始化它们。

无论你是否使用默认的序列化形式，**必须对对象序列化强制执行任何同步操作，就像对读取对象的整个状态的任何其他方法强制执行的那样。** 例如，如果你有一个线程安全的对象（详见第 82 条），它通过同步每个方法来实现线程安全，并且你选择使用默认的序列化形式，那么使用以下 write-Object 方法：

```
// writeObject for synchronized class with default serialized form
private synchronized void writeObject(ObjectOutputStream s) throws IOException {
    s.defaultWriteObject();
}
```

如果将同步放在 `writeObject` 方法中，则必须确保它遵守与其他活动相同的锁排序约束，否则将面临资源排序死锁的风险 [Goetz06, 10.1.5]。

无论选择哪种序列化形式，都要在编写的每个可序列化类中声明显式的序列版本 UID。 这消除了序列版本 UID 成为不兼容性的潜在来源（详见第 86 条）。这么做还能获得一个小的性能优势。如果没有提供序列版本 UID，则需要执行高开销的计算在运行时生成一个 UID。

声明序列版本 UID 很简单，只要在你的类中增加这一行：

```
private static final long serialVersionUID = randomLongValue;
```

如果你编写一个新类，为 `randomLongValue` 选择什么值并不重要。你可以通过在类上运行 serialver 实用工具来生成该值，但是也可以凭空选择一个数字。串行版本 UID 不需要是唯一的。如果修改缺少串行版本 UID 的现有类，并且希望新版本接受现有的序列化实例，则必须使用为旧版本自动生成的值。你可以通过在类的旧版本上运行 serialver 实用工具（序列化实例存在于旧版本上）来获得这个数字。

如果你希望创建一个新版本的类，它与现有版本不兼容，如果更改序列版本 UID 声明中的值，这将导致反序列化旧版本的序列化实例的操作引发 `InvalidClassException`。**不要更改序列版本 UID，除非你想破坏与现有序列化所有实例的兼容性。**

总而言之，如果你已经决定一个类应该是可序列化的（详见第 86 条），那么请仔细考虑一下序列化的形式应该是什么。只有在合理描述对象的逻辑状态时，才使用默认的序列化形式；否则，设计一个适合描述对象的自定义序列化形式。设计类的序列化形式应该和设计导出方法花的时间应该一样多，都应该严谨对待（详见第 51 条）。正如不能从未来版本中删除导出的方法一样，也不能从序列化形式中删除字段；必须永远保存它们，以确保序列化兼容性。选择错误的序列化形式可能会对类的复杂性和性能产生永久性的负面影响。

88. 保护性的编写 readObject 方法

第 50 条介绍了一个不可变的日期范围类，它包含可变的私有变量 `Date`。该类通过在其构造器和访问方法（accessor）中保护性的拷贝 `Date` 对象，极力维护其约束条件和不可变性。该类代码如下所示：

```
// Immutable class that uses defensive copying
public final class Period {
    private final Date start;
    private final Date end;
    /**
     * @param start the beginning of the period
     * @param end the end of the period; must not precede start
     * @throws IllegalArgumentException if start is after end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());
        if (this.start.compareTo(this.end) > 0)
```

```
            throw new IllegalArgumentException(start + " after " + end);
    }
    public Date start () {
        return new Date(start.getTime());
    }
    public Date end () {
        return new Date(end.getTime());
    }
    public String toString() {
        return start + " - " + end;
    }
    ... // Remainder omitted
}
```

假设你决定要把这个类成为可序列化的。因为 Period 对象的物理表示法正好反映了它的逻辑数据内容，所以，使用默认的序列化形式是合理的（详见 87 条）。因此，为了使这个类成为可序列化的，似乎你所需要做的也就是在类的声明中增加 implements Serializable 字样。然而，如果你真的这么做，那么这个类就不保证它的关键约束了。

问题在于 `readObject` 方法实际上相当于另外一个公有的构造器，它要求同其他构造器一样警惕所有的注意事项。构造器必须检查其参数的有效性（详见 49 条），并且在必要的时候对参数进行保护性拷贝（详见 50 条），同样的，`readObject` 方法也需要这样做。如果 `readObject` 方法无法做到这两者之一，对于攻击者来说要违反这个类的约束条件就相对容易很多。

不严格的说，`readObject` 方法是一个「用字节流作为唯一参数」的构造器。在正常使用的情况下，对一个正常构造的实例进行序列化可以产生字节流。但是，当面对一个人工仿造的字节流时，`readObject` 产生的对象会违反它所属类的约束条件，这时问题就产生了。这种字节流可以用来创建一个不可能的对象（impossible object），这时利用普通构造器无法创建的。

假设我们仅仅在 Period 类的声明加上了 `implements Serializable` 字样。那么这个丑陋的程序代码将会产生一个 Period 实例，他的结束时间比起始时间还早。对于高位 byte 值进行强制类型转换是 Java 缺少 byte 并且做出 byte 类型签名的不幸决定的后果：

```
public class BogusPeriod {
    // Byte stream couldn't have come from a real Period instance!
    private static final byte[] serializedForm = {
        (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x06,
        0x50, 0x65, 0x72, 0x69, 0x6f, 0x64, 0x40, 0x7e, (byte)0xf8,
        0x2b, 0x4f, 0x46, (byte)0xc0, (byte)0xf4, 0x02, 0x00, 0x02,
        0x4c, 0x00, 0x03, 0x65, 0x6e, 0x64, 0x74, 0x00, 0x10, 0x4c,
        0x6a, 0x61, 0x76, 0x61, 0x2f, 0x75, 0x74, 0x69, 0x6c, 0x2f,
        0x44, 0x61, 0x74, 0x65, 0x3b, 0x4c, 0x00, 0x05, 0x73, 0x74,
        0x61, 0x72, 0x74, 0x71, 0x00, 0x7e, 0x00, 0x01, 0x78, 0x70,
        0x73, 0x72, 0x00, 0x0e, 0x6a, 0x61, 0x76, 0x61, 0x2e, 0x75,
        0x74, 0x69, 0x6c, 0x2e, 0x44, 0x61, 0x74, 0x65, 0x68, 0x6a,
        (byte)0x81, 0x01, 0x4b, 0x59, 0x74, 0x19, 0x03, 0x00, 0x00,
        0x78, 0x70, 0x77, 0x08, 0x00, 0x00, 0x00, 0x66, (byte)0xdf,
        0x6e, 0x1e, 0x00, 0x78, 0x73, 0x71, 0x00, 0x7e, 0x00, 0x03,
        0x77, 0x08, 0x00, 0x00, 0x00, (byte)0xd5, 0x17, 0x69, 0x22,
```

```
        0x00, 0x78
    };
    public static void main(String[] args) {
        Period p = (Period) deserialize(serializedForm);
        System.out.println(p);
    }
    // Returns the object with the specified serialized form
    static Object deserialize(byte[] sf) {
    try {
            return new ObjectInputStream(
        new ByteArrayInputStream(sf)).readObject();
    }
    catch (IOException | ClassNotFoundException e) {
        throw new IllegalArgumentException(e);
    }
    }
}
```

被用来初始化 serializedForm 的 byte 常量数组是这样产生的：首先对一个正常的 Period 实例进行序列化，然后对得到的字节流进行手工编辑。对于这个例子而言，字节流的细节并不重要，如果你对此十分好奇，可以在《Java Object Serialization Specification》[Serialization, 6] 中查到有关序列化字节流格式的描述信息。如果运行这个程序，它会打印出「Fri Jan 01 12:00:00 PST 1999 - Sun Jan 01 12:00:00 PST 1984」。主要把 Period 类声明成为可序列化的，这会使我们创建出其违反类约束条件的对象。

为了修整这个问题，可以为 `Period` 提供一个 `readObject` 方法，该方法首先调用 `defaultReadObject`，然后检查被反序列化之后的对象有效性。如果有效性检查失败，`readObject` 方法就会抛出一个 `InvalidObjectException` 异常，这使得反序列化过程不能成功的完成：

```
// readObject method with validity checking - insufficient!
private void readObject(ObjectInputStream s)
        throws IOException, ClassNotFoundException {
    s.defaultReadObject();
    // Check that our invariants are satisfied
    if (start.compareTo(end) > 0)
    throw new InvalidObjectException(start +" after "+ end);
}
```

尽管这样的修成避免了攻击者创建无效的 `Period` 实例，但是这里依旧隐藏着一个更为微妙的问题。通过伪造字节流，要想创建可变的 `Period` 实例仍是有可能的，做法是：字节流以一个有效的 `Period` 实例开头，然后附加上两个额外的引用，指向 `Period` 实例中两个私有的 `Date` 字段。攻击者从 `ObjectInputStream` 读取 `Period` 实例，然后读取附加在其后面的「恶意编制的对线引用」。这些对象引用使得攻击者能够访问到 `Period` 对象内部的私有 `Date` 字段所引用的对象。通过改变这些 `Date` 实例，攻击者可以改变 `Period` 实例。如下的类演示了这种攻击方式：

```
public class MutablePeriod {
    // A period instance
```

```
    public final Period period;
    // period's start field, to which we shouldn't have access
    public final Date start;
    // period's end field, to which we shouldn't have access
    public final Date end;

    public MutablePeriod() {
        try {
            ByteArrayOutputStream bos =
            new ByteArrayOutputStream();
            ObjectOutputStream out =
            new ObjectOutputStream(bos);
            // Serialize a valid Period instance
            out.writeObject(new Period(new Date(), new Date()));
            /*
             * Append rogue "previous object refs" for internal
             * Date fields in Period. For details, see "Java
             * Object Serialization Specification," Section 6.4.
            */
            byte[] ref = { 0x71, 0, 0x7e, 0, 5 };
            // Ref #5
            bos.write(ref);
            // The start field
            ref[4] = 4;
            // Ref # 4
            bos.write(ref);
            // The end field
            // Deserialize Period and "stolen" Date references
            ObjectInputStream in = new ObjectInputStream(
            new ByteArrayInputStream(bos.toByteArray()));
            period = (Period) in.readObject();
            start = (Date) in.readObject();
            end = (Date) in.readObject();
        }
        catch (IOException | ClassNotFoundException e) {
            throw new AssertionError(e);
        }
    }
}
```

要查看正在进行的攻击，请运行以下程序：

```
public static void main(String[] args) {
    MutablePeriod mp = new MutablePeriod();
    Period p = mp.period;
    Date pEnd = mp.end;
    // Let's turn back the clock
    pEnd.setYear(78);
    System.out.println(p);
    // Bring back the 60s!
    pEnd.setYear(69);
    System.out.println(p);
}
```

在我本地机器上运行这个程序产生的输出结果如下：

```
Wed Nov 22 00:21:29 PST 2017 - Wed Nov 22 00:21:29 PST 1978
Wed Nov 22 00:21:29 PST 2017 - Sat Nov 22 00:21:29 PST 1969
```

虽然 Period 实例被创建之后，他的约束条件没有被破坏。但是要随意修改它的内部组件仍然是有可能的。一旦攻击者获得了一个可变的 `Period` 实例，就可以将这个实例传递给一个「安全性依赖于 Period 的不可变性」的类，从而造成更大的危害。这种推断并不牵强：实际上，有许多类的安全性就是依赖于 String 的不可变性。

问题的根源在于， `Period` 的 `readObject` 方法并没有完成足够的保护性拷贝。 **当一个对象被反序列化的时候，对于客户端不应该拥有的对象引用，如果那个字段包含了这样的对象引用，就必须做保护性拷贝，这是非常重要的。** 因此，对于每个可序列化的不可变类，如果它包含了私有的可变字段，那么在它的 `readObject` 方法中，必须要对这些字段进行保护性拷贝。下面的这些 `readObject` 方法可以确保 `Period` 类的约束条件不会遭到破坏，以保持它的不可变性：

```
// readObject method with defensive copying and validity checking
private void readObject(ObjectInputStream s)
            throws IOException, ClassNotFoundException {
    s.defaultReadObject();
    // Defensively copy our mutable components
    start = new Date(start.getTime());
    end = new Date(end.getTime());
    // Check that our invariants are satisfied
    if (start.compareTo(end) > 0)
    throw new InvalidObjectException(start +" after "+ end);
}
```

注意，保护性拷贝是在有效性检查之前进行的。我们没有使用 `Date` 的 `clone` 方法来执行保护性拷贝机制。这两个细节对于保护 `Period` 类免受攻击是必要的（详见 50 条）。同时也注意到，对于 final 字段，保护性字段是不可能的。为了使用 `readObject` 方法，我们必须要将 start 和 end 字段声明成为非 final 的。很遗憾的是，这还算是相对比较好的做法。有了这新的 `readObject` 方法，并且取消了 start 和 end 的 final 修饰符之后， `MutablePeriod` 类将不再有效。此时，上面的攻击程序会产生如下输出：

```
Wed Nov 22 00:23:41 PST 2017 - Wed Nov 22 00:23:41 PST 2017
Wed Nov 22 00:23:41 PST 2017 - Wed Nov 22 00:23:41 PST 2017
```

有一个简单的「石蕊」测试，可以用来确定默认的 `readObject` 方法是否可以被接受。测试方法：增加一个公有的构造器，其参数对应于该对象中每个非 transient 的字段，并且无论参数的值是什么，都是不进行检查就可以保存到相应的字段中。对于这样的做法，你是否会感到很舒适？如果你对这个问题的回答是否定的，就必须提供一个显式的 `readObject` 方法，并且它必须执行构造器所要求的所有有效性检查和保护性拷贝。另一种方法是，可以使用序列化代理模式（serialization proxy pattern），详见第 90 条。强烈建议使用这个模式，因为它分担了安全反序列化的部门工作。

对于非 final 的可序列化的类，在 `readObject` 方法和构造器之间还有其他类似的地方。与构造器一样，`readObject` 方法不可以调用可被覆盖的方法，无论是直接调用还是间接调用都不可以（详见 19 条）。如果违反了这条规则，并且覆盖了该方法，被覆盖的方法将在子类的状态被反序列化之前先运行。这个程序很可能会失败[Bloch05, Puzzle 91]。

总而言之，在编写 `readObject` 方法的时候，都要这样想：你正在编写一个公有的构造器，无论给它传递什么样的字节流，它都必须产生一个有效的实例。不要假设这个字节流一定代表着一个真正被序列化的实例。虽然在本条目的例子中，类使用了默认的序列化形式，但是所有讨论到的有可能发生的问题也同样适用于自定义序列化形式的类。下面以摘要的形式给出一些指导方针，有助于编写出更健壮的 `readObject` 方法。

- 类中的对象引用字段必须保持为私有属性，要保护性的拷贝这些字段中的每个对象。不可变类中的可变组件就属于这一类别
- 对于任何约束条件，如果检查失败就抛出一个 `InvalidObjectException` 异常。这些检查动作应该跟在所有的保护性拷贝之后。
- 如果整个对象图在被反序列化之后必须进行验证，就应该使用 `ObjectInputValidation` 接口（本书没有讨论）。
- 无论是直接方法还是间接方法，都不要调用类中任何可被覆盖的方法。

89. 对于实例控制，枚举类型优于 readResolve

第 3 条讲述了 Singletion（单例）模式，并且给出了以下这个 Singletion 示例。这个类限制了对其构造器的访问，以确保永远只创建一个实例。

```
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();

    private Elvis() { ... }
    public void leaveTheBuilding() { ... }
}
```

正如在第 3 条中所提到的，如果在这个类上面增加 `implements Serializable` 的字样，它就不是一个单例。无论该类使用了默认的序列化形式，还是自定义的序列化形式（详见 87 条），都没有关系；也跟它是否使用了显式的 `readObject` （详见 88 条）无关。任何一个 `readObject` 方法，不管是显式的还是默认的，都会返回一个新建的实例，这个新建的实例不同于类初始化时创建的实例。

`readResolve` 特性允许你用 `readObject` 创建的实例代替另外一个实例[Serialization, 3.7]。对于一个正在被反序列化的对象，如果它的类定义了一个 `readResolve` 方法，并且具备正确的声明，那么在反序列化之后，新建对象上的 `readResolve` 方法就会被调用。然后，该方法返回的对象引用将会被回收，取代新建的对象。这个特性在绝大多数用法中，指向新建对象的引用不会再被保留，因此成为垃圾回收的对象。

如果 Elvis 类要实现 `Serializable` 接口，下面的 `readResolve` 方法就足以保证它的单例属性：

```
// readResolve for instance control - you can do better!
private Object readResolve() {
    // Return the one true Elvis and let the garbage collector
    // take care of the Elvis impersonator.
    return INSTANCE;
}
```

该方法忽略了被反序列化的对象，只返回类初始化创建的那个特殊的 `Elvis` 实例。因此 `Elvis` 实例的序列化形式不应该包含任何实际的数据；所有的实例字段都应该被声明为 `transient` 。事实上，**如果依赖 `readResolve` 进行实例控制，带有对象引用类型的所有实例字段都必须声明为 `transient`** 。 否则，那种破釜沉舟式的攻击者，就有可能在 `readResolve` 方法运行之前，保护指向反序列化对象的引用，采用的方式类似于在第 88 条中提到的 `MutablePeriod` 攻击。

这种攻击有点复杂，但是背后的思想十分简单。如果单例包含一个非 `transient` 的对象引用字段，这个字段的内容就可以在单例的 `readResolve` 方法之前被反序列化。当对象引用字段的内容被反序列化时，它就允许一个精心制作的流「盗用」指向最初被反序列化的单例对象引用。

以下是它更详细的工作原理。首先编写一个「盗用者」类，它既有 `readResolve` 方法，又有实例字段，实例字段指向被序列化的单例的引用，「盗用者」就「潜伏」在其中。在序列化流中，用「盗用者」的 `readResolve` 方法运行时，它的实例字段仍然引用部分反序列化（并且还没有被解析）的 Singletion。

「盗用者」的 `readResolve` 方法从它的实例字段中将引用复制到静态字段中，以便该引用可以在 `readResolve` 方法运行之后被访问到。然后这个方法为它所藏身的那个域返回一个正确的类型值。如果没有这么做，当序列化系统试着将「盗用者」引用保存到这个字段时，虚拟机就会抛出 `ClassCastException` 。

为了更具体的说明这一点，我们以如下这个单例模式为例：

```
// Broken singleton - has nontransient object reference field!
public class Elvis implements Serializable {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { }

    private String[] favoriteSongs =
```

```
        { "Hound Dog", "Heartbreak Hotel" };
    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }

    private Object readResolve() {
        return INSTANCE;
    }
}
```

如下「盗用者」类，是根据以上描述构造的：

```
public class ElvisStealer implements Serializable {
    static Elvis impersonator;
    private static final long serialVersionUID = 0;
    private Elvis payload;

    private Object readResolve() {
        // Save a reference to the "unresolved" Elvis instance
        impersonator = payload;

        // Return object of correct type for favoriteSongs field
        return new String[] { "A Fool Such as I" };
    }
}
```

下面是一个不完整的程序，它反序列化一个手工制作的流，为那个有缺陷的单例产生两个截然不同的实例。这个程序省略了反序列化方法，因为它与第 88 条一样。

```
public class ElvisImpersonator {
    // Byte stream couldn't have come from a real Elvis instance!
    private static final byte[] serializedForm = {
            (byte) 0xac, (byte) 0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x05, 0x45,
            0x6c, 0x76, 0x69, 0x73, (byte) 0x84, (byte) 0xe6, (byte) 0x93, 0x33,
            (byte) 0xc3, (byte) 0xf4, (byte) 0x8b, 0x32, 0x02, 0x00, 0x01, 0x4c,
            0x00, 0x0d, 0x66, 0x61, 0x76, 0x6f, 0x72, 0x69, 0x74, 0x65, 0x53,
            0x6f, 0x6e, 0x67, 0x73, 0x74, 0x00, 0x12, 0x4c, 0x6a, 0x61, 0x76,
            0x61, 0x2f, 0x6c, 0x61, 0x6e, 0x67, 0x2f, 0x4f, 0x62, 0x6a, 0x65,
            0x63, 0x74, 0x3b, 0x78, 0x70, 0x73, 0x72, 0x00, 0x0c, 0x45, 0x6c,
            0x76, 0x69, 0x73, 0x53, 0x74, 0x65, 0x61, 0x6c, 0x65, 0x72, 0x00,
            0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x01, 0x4c,
            0x00, 0x07, 0x70, 0x61, 0x79, 0x6c, 0x6f, 0x61, 0x64, 0x74, 0x00,
            0x07, 0x4c, 0x45, 0x6c, 0x76, 0x69, 0x73, 0x3b, 0x78, 0x70, 0x71,
            0x00, 0x7e, 0x00, 0x02
        };

    public static void main(String[] args) {
        // Initializes ElvisStealer.impersonator and returns
```

```
        // the real Elvis (which is Elvis.INSTANCE)
        Elvis elvis = (Elvis) deserialize(serializedForm);
        Elvis impersonator = ElvisStealer.impersonator;
        elvis.printFavorites();
        impersonator.printFavorites();
    }
}
```

这个程序会产生如下的输出，最终证明可以创建两个截然不同的 `Elvis` 实例（包含两种不同的音乐品味）：

```
[Hound Dog, Heartbreak Hotel]
[A Fool Such as I]
```

通过将 `favoriteSongs` 字段声明为 `transient`，可以修复这个问题，但是最好把 `Elvis` 做成一个单元素的枚举类型（详见第 3 条）。就如 `ElvisStealer` 攻击所示范的，用 `readResolve` 方法防止“临时”被反序列化的实例收到攻击者的访问，这种方法十分脆弱需要万分谨慎。

如果将一个可序列化的实例受控的类编写为枚举，Java 就可以绝对保证除了所声明的常量之外，不会有其他实例，除非攻击者恶意的使用了享受特权的方法。如 `AccessibleObject.setAccessible`。能够做到这一点的任何一位攻击者，已经拥有了足够的特权来执行任意的本地代码，后果不堪设想。将 Elvis 写成枚举的例子如下所示：

```
// Enum singleton - the preferred approach
public enum Elvis {
    INSTANCE;

    private String[] favoriteSongs = { "Hound Dog", "Heartbreak Hotel" };

    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }
}
```

用 readResolve 进行实例控制并不过时。如果必须编写可序列化的实力受控的类，在编译时还不知道它的实例，你就无法将类表示成为一个枚举类型。

readResolve 的可访问性（accessibility）十分重要。 如果把 `readResolve` 方法放在一个 `final` 类上面，它应该是私有的。如果把 `readResolve` 方法放在一个非 `final` 类上，就必须认真考虑它的的访问性。如果它是私有的，就不适用于任何一个子类。如果它是包级私有的，就适用于同一个包内的子类。如果它是受保护的或者是公开的，并且子类没有覆盖它，对序列化的子类进行反序列化，就会产生一个超类实例，这样可能会导致 `ClassCastException` 异常。

总而言之，应该尽可能的使用枚举类型来实施实例控制的约束条件。如果做不到，同时又需要一个即可序列化又可以实例受控的类，就必须提供一个 `readResolve` 方法，并确保该类的所有实例化字段都被基本类型，或者是 `transient` 的。

90. 考虑用序列化代理代替序列化实例

正如 85 条和第 86 条提到的，以及本章一直在讨论的，决定实现 Serializable 接口，会增加出错和出现安全问题的可能性，因为它允许利用语言之外的机制来创建实例，而不是使用普通的构造器。然而，有一种方法可以极大的减少这些风险。就是序列化代理模式（seralization proxy pattern）。

序列化代理模式相当简单。首先，为可序列化的类设计一个私有的静态嵌套类，精确地表示外围类的逻辑状态。这个嵌套类被称为序列化代理（seralization proxy），它应该有一个单独的构造器，其参数类型就是那个外围类。这个构造器只是从它的参数中复制数据：它不需要进行任何一致性检验或者保护性拷贝。从设计的角度看，序列化代理的默认序列化形式是外围类最好的序列化形式。外围类及其序列代理都必须声明实现 `Serializable` 接口。

例如，以第 50 条中编写不可变的 `Period` 类为例，它在第 88 条中被作为可序列化的。以下是一个类的序列化代理。 `Period` 类是如此简单，以致于它的序列化代理有着与类完全相同的字段。

```
// Serialization proxy for Period class
private static class SerializationProxy implements Serializable {
    private final Date start;
    private final Date end;
    SerializationProxy(Period p) {
        this.start = p.start;
        this.end = p.end;
    }
    private static final long serialVersionUID =
                        234098243823485285L; // Any number will do (Item 87)
}
```

接下来，将下面的 `writeReplace` 方法添加到外围类中。通过序列化代理，这个方法可以被逐字的复制到任何类中。

```
// writeReplace method for the serialization proxy pattern
private Object writeReplace() {
    return new SerializationProxy(this);
}
```

这个方法的存在就是导致系统产生一个 `SerializationProxy` 实例，代替外围类的实例。换句话说 `writeReplace` 方法在序列化之前，将外围类的实例转变成了它的序列化代理。

有了 `writeReplace` 方法之后，序列化系统永远不会产生外围类的序列化实例，但是攻击者有可能伪造企图违反该类约束条件的示例。为了防止此类攻击，只需要在外围类中添加如下 `readObject` 方法。

```
// readObject method for the serialization proxy pattern
private void readObject(ObjectInputStream stream)
        throws InvalidObjectException {
    throw new InvalidObjectException("Proxy required");
}
```

最后在 `SerializationProxy` 类中提供一个 `readResolve` 方法，他返回一个逻辑上等价的外围类的实例。这个方法的出现，导致序列化系统在反序列化的时候将序列化代理转为外围类的实例。

这个 `readResolve` 方法仅仅利用它的公有 API 创建外围类的一个实例，这正是该模式的魅力所在它极大的消除了序列化机制中语言之外的特征，因为反序列化实例是利用与任何其他实例相同的构造器、静态工厂和方法而创建的。这样你就不必单独确保被反序列化的实例一定要遵守类的约束条件。如果该类的静态工厂或者构造器建立了这些约束条件，并且它的实例方法保持着这些约束条件，你就可以确信序列化也确保着这些约束条件。

以下是上述的 `Period.SerializationProxy` 的 `readResolve` 方法：

```
// readResolve method for Period.SerializationProxy
private Object readResolve() {
    return new Period(start, end); // Uses public constructor
}
```

正如保护性拷贝方法一样（详见 88 条），序列化代理方式可以阻止伪字节流的攻击（详见 88 条）以及内部字段的盗用攻击（详见 88 条）。与前两种方法不同，这种方法允许 `Period` 类的字段为 final，为了确保 `Period` 类是真正不可变的（详见 17 条），这一点非常重要。与前两种方法不同的还有，这种方法不需要太费心思。你不必知道哪些字段可能受到狡猾的序列化攻击的威胁，你也不必显式的执行有效性检查，作为反序列化的一部分。

还有另外一种方法，使用这种方法时，序列化代理模式的功能比保护性拷贝的更加强大。序列化代理模式允许反序列化实例有着与原始序列化实例不同的类。你可能认为这在实际应用中没有什么作用，其实不然。

以 `EnumSet` 的情况为例（详见 36 条）。这个类没有公有的构造器，只有静态工厂。从客户端的角度来看，他们返回 `EnumSet` 实例，但是在 OpenJDK 的实现，它们返回的是两种子类之一，具体取决于底层枚举类型的大小。如果底层的枚举类型有 64 个或者少于 64 个的元素，静态工厂就返回一个 `RegularEnumSet`；它们就返回一个 `JunmboEnumSet`。

现在考虑这种情况：如果序列化一个枚举类型，它的枚举有 60 个元素，然后给这个枚举类型再增加 5 个元素，之后反序列化这个枚举集合。当它被序列化的时候，是一个 `RegularEnumSet` 实例，但是它一旦被反序列化，他就变成了 `JunmboEnumSet` 实例。实际发生的情况也正是如此，因为 `EnumSet` 使用序列化代理模式如果你感兴趣，可以看看如下的 `EnumSet` 序列化代理，它实际上就是这么简单：

```
// EnumSet's serialization proxy
private static class SerializationProxy<E extends Enum<E>>
    implements Serializable {
    private static final long serialVersionUID = 362491234563181265L;
```

```
    // The element type of this enum set.
    private final Class<E> elementType;

    // The elements contained in this enum set.
    private final Enum<?>[] elements;

    SerializationProxy(EnumSet<E> set) {
        elementType = set.elementType;
        elements = set.toArray(new Enum<?>[0]);
    }

    private Object readResolve() {
        EnumSet<E> result = EnumSet.noneOf(elementType);

        for (Enum<?> e : elements)
            result.add((E) e);

        return result;
    }
}
```

序列化代理模式有两个局限性。它不能与可以被客户端拓展的类兼容（详见 19 条）。它也不能与对象图中包含循环的某些类兼容：如果你企图从一个对象的序列化代理的 readResovle 方法内部调用这个对象的方法，就会得到一个 `ClassCastException` 异常，因为你还没有这个对象，只有它的序列化代理。

最后一点，序列化代理模式所增强的功能和安全性不是没有代价。在我的机器上，通过序列化代理来序列化和反序列化 `Period` 实例的开销，比使用保护性拷贝增加了 14%。

总而言之，当你发现必须在一个不能被客户端拓展的类上面编写 `readObject` 或者 `writeObject` 方法时，就应该考虑使用序列化代理模式。想要稳健的将带有重要约束条件的对象序列化时，这种模式是最容易的方法。