github: https://github.com/sjsdfg/effctive-java-3rd-chinese

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公共静态工厂方法，它只是一个返回类实例的静态方法。 下面是一个 `Boolean` 简单的例子（`boolean` 基本类型的包装类）。 此方法将 `boolean` 基本类型转换为 `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。提供静态工厂方法而不是公共构造方法有优点也有缺点。

静态工厂方法的一个优点是，不像构造方法，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构造方法来解决这个限制，这两个构造方法的参数列表只有它们的参数类型的顺序不同。 这是一个非常糟糕的主意。 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。

因为他们有名字，所以静态工厂方法不会受到上面讨论中的限制。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。 这允许不可变的类 (条目 17) 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，特别是如果在创建它们非常昂贵的情况下。

静态工厂方法从重复调用返回相同对象的能力允许类保持在任何时候存在的实例的严格控制。这样做的类被称为实例控制（instance-controlled）。编写实例控制类的原因有很多。实例控制允许一个类来保证它是一个单例 (3) 项或不可实例化的 (条目 4)。同时,它允许一个不可变的值类 (条目 17) 保证不存在两个相同的实例：当且仅当 `a == b` 时 `a.equals(b)`。这是享元模式的基础[Gamma95]。 `Enum` 类型 (条目 34) 提供了这个保证。

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l。 这种技术适用于基于接口的框架（条目 20），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 `Type` 的接口的静态工厂方法被放入一个非实例化的伙伴类 (companion class)(条目 4) `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非实例类 (`java .util.collections`) 中导出的。返回对象的类都是非公开的。

`Collections` 框架 API 的规模要比它之前输出的 45 个单独的公共类要小得多，每个类有个便利类的实现。不仅是 API 的大部分减少了，还包括概念上的权重：程序员必须掌握的概念的数量和难度，才能使用 API。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而不是实现类来引用返回的对象，这通常是良好的实践 (条目 64)。

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（条目 36）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：如果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， 返回一个 `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，返回一个 `long` 类型的数组。

这两个实现类的存在对于客户是不可见的。 如果 `RegularEnumSet` 不再为小枚举类型提供性能优势，则可以在未来版本中将其淘汰，而不会产生任何不良影响。 同样，未来的版本可能会添加 `EnumSet` 的第三个或第四个实现，如果它证明有利于性能。 客户既不知道也不关心他们从工厂返回的对象的类别; 他们只关心它是 `EnumSet` 的一些子类。

静态工厂的第 5 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 (条目 65)。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（条目 5）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（条目 59）。 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，在 `Collections` 框架中不可能将任何方便实现类子类化。可以说，这可能是因祸得福，因为它鼓励程序员使用组合而不是继承 (条目 18)，并且是不可变类型 (条目 17)。

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 文档中突出，因此很难找出如何实例化一个提供静态工厂方法而不是构造方法的类。Javadoc 工具可能有一天会引起对静态工厂方法的注意。与此同时，可以通过将注意力吸引到类或接口文档中的静态工厂以及遵守通用的命名约定来减少这个问题。下面是一些静态工厂方法的常用名称。以下清单并非完整：

- from——A 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——一个聚合方法，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 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，除了该方法保证每个调用返回一个新的实例，例如：**Object newArray = Array.newInstance(classObject, arrayLen)**;
- getType——与 getInstance 类似，但是如果在工厂方法中不同的类中使用。**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，但是如果在工厂方法中不同的类中使用。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 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：它们不能很好地扩展到很多可选参数的情景。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。大多数产品都有非零值，只有少数几个可选属性。

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，只提供了一个只所需参数的构造函数，另一个只有一个可选参数，第三个有两个可选参数，等等，最终在构造函数中包含所有可选参数。这就是它在实践中的样子。为了简便起见，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，可以使用包含所有要设置的参数的最短参数列表的构造方法：

```
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 值。 『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，它会很快失控。

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，并且必须仔细地计算参数才能找到答案。一长串相同类型的参数可能会导致一些细微的 bug。如果客户端意外地反转了两个这样的参数，编译器并不会抱怨，但是程序在运行时会出现错误行为 (条目 51)。

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，调用一个无参数的构造函数来创建对象，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。由于构造方法在多次调用中被分割，所以在构造过程中 JavaBean 可能处于不一致的状态。该类没有通过检查构造参数参数的有效性来执行一致性的选项。在不一致的状态下尝试使用对象可能会导致与包含 bug 的代码大相径庭的错误，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 (条目 17)，并且需要在程序员的部分增加工作以确保线程安全。

当它的构造完成时，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。客户端不直接调用所需的对象，而是调用构造方法 (或静态工厂)，并使用所有必需的参数，并获得一个 builder 对象。然后，客户端调用 builder 对象的 `setter` 相似方法来设置每个可选参数。最后，客户端调用一个无参的 `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 (条目 24)。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，这样调用就可以被链接起来，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。 Builder 模式模拟 Python 和 Scala 中的命名可选参数。

为了简洁起见，省略了有效性检查。 要尽快检测无效参数，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（条目 50）。 如果检查失败，则抛出 `IllegalArgumentException` 异常（条目 72），其详细消息指示哪些参数无效（条目 75）。

Builder 模式非常适合类层次结构。 使用平行层次的 builder，每个嵌套在相应的类中。 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）（条目 30）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）的习惯用法。

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：

```java
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"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，但在性能关键的情况下可能会出现问题。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，如果希望在将来添加更多的参数。但是，如果从构造方法或静态工厂开始，并切换到 builder，当类演化到参数数量失控的时候，过时的构造方法或静态工厂就会面临尴尬的处境。因此，所以，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，特别是如果许多参数是可选的或相同类型的。客户端代码比使用伸缩构造方法（telescoping constructors）更容易读写，并且 builder 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 (条目 24) 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 (条目 65)。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。

静态工厂方法的一个优点是，它可以灵活地改变你的想法，无论该类是否为单例而不必更改其 API。 工厂方法返回唯一的实例，但是可以修改，比如，返回调用它的每个线程的单独实例。 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 ）（条目 30）。 使用静态工厂的最后一个优点是方法引用可以用 `supplier`，例如 `Elvis::instance` 等同于 `Supplier<Elvis>`。 除非与这些优点相关的，否则公共属性方法是可取的。

创建一个使用这两种方法的单例类 (第 12 章)，仅仅将 `implements Serializable` 添加到声明中是不够的。为了维护单例的保证，声明所有的实例属性为 `transient`，并提供一个 `readResolve` 方法 (条目 89)。否则，每当序列化实例被反序列化时，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，将这个 `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，提供了免费的序列化机制，并提供了针对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. 使用私有构造方法执行非实例化

偶尔你会想写一个类，它只是一组静态方法和静态属性。 这样的类获得了不好的名声，因为有些人滥用这些类而避免以面向对象方式思考，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，在基本类型的数值或数组上组织相关的方法。 它们也可以用于将静态方法（包括工厂（条目 1））分组，用于实现某个接口的对象，其方式为 `java.util.Collections`。 （从 Java 8 开始，你也可以将这些方法放在接口中，假如它是你自己修改的。）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。

这样的实用类（utility classes）不是设计用来被实例化的：一个实例是没有意义的。然而，在没有显式构造方法的情况下，编译器提供了一个公共的、无参的默认构造方法。对于用户来说，该构造方法与其他构造方法没有什么区别。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，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 (条目 19)。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方法时，才会生成一个默认构造方法，**因此可以通过包含一个私有构造方法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
}
```

因为显式构造方法是私有的，所以在类之外是不可访问的。`AssertionError` 异常不是严格要求的，但是它提供了一种保证，以防在类中意外地调用构造方法。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，好像构造方法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，阻止了类的子类化。所有的构造方法都必须显式或隐式地调用父类构造方法，而子类则没有可访问的父类构造方法来调用。

05. 使用依赖注入取代硬连接资源（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实用工具类并不少见 (条目 4):

```
// 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，将它们实现为单例也并不少见 (条目 3)：

```
// 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，因为他们假设只有一本字典值得使用。在实际中，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，使用专门的字典来进行测试也是可取的。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，从而让拼写检查器支持多个字典，但是在并发环境中，这是笨拙的、容易出错的和不可行的。静态实用类和单例对于那些行为被底层资源参数化的类来说是不合适的。

所需要的是能够支持类的多个实例 (在我们的示例中，即 `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方法中。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（条目 17），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方法，静态工厂（条目 1）和 builder 模式（条目 2）。

该模式的一个有用的变体是将资源工厂传递给构造方法。 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)（条目 31）约束工厂的类型参数，以允许客户端传入工厂，创建指定类型的任何子类型。 例如，下面是一个使用客户端提供的工厂生成 tile 的方法：

```
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。

总之，不要使用单例或静态的实用类来实现一个类，该类依赖于一个或多个底层资源，这些资源的行为会影响类的行为，并且不让类直接创建这些资源。相反，将资源或工厂传递给构造方法 (或静态工厂或 builder 模式)。这种称为依赖注入的实践将极大地增强类的灵活性、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 (条目 17)，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，就可以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 (static factory methods(条目 1)，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（条目 83）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（条目 67）。

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（autoboxing），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（条目 61）。 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 231 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。

考虑以下简单的堆栈实现：

```
// 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 (条目 57)，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。Link`此处输入代码`edHashMap 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（条目 9）。

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（条目 9）。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 (条目 24)。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（条目 8）。

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。任何类都有义务重写这些方法，以遵从他们的通用约定；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）（条目 1）的类，以确保每个值至多存在一个对象。 枚举类型（条目 34）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<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，和一个公共的试图（view）（条目 6）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（条目 17）。 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（条目 17）。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（条目 11）。**
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，编写一个看起来像这样的 equals 方法并不少见，然后花上几个小时苦苦思索为什么它不能正常工作：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，编写一个看起来像这样的 equals 方法并不少见，然后花上几个小时苦苦思索为什么它不能正常工作。

```
// 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（Item 52）。 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 (条目 40)。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. 当在一个应用程序执行过程中，如果在 equals 方法比较中没有修改任何信息，在一个对象上重复调用 hashCode 方法时，它必须始终返回相同的值。从一个应用程序到另一个应用程序的每一次执行返回的值可以是不一致的。
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，对对象有引用的组件可能包含在日志错误消息中对象的字符串表示。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);
```

程序员将以这种方式生成诊断消息，不管你是否重写 toString，但是除非你这样做，否则这些消息将不会有用。 提供一个很好的 toString 方法的好处不仅包括类的实例，同样有益于包含实例引用的对象，特别是集合。 打印 map 对象时你会看到哪一个，`{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（条目 4）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 (条目 20)，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 (条目 65)，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（条目 71）。

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（条目 19）。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（条目 71）。

在为继承设计一个类时（条目 19），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（条目 78）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（条目 67）。 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（条目 34）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（条目 10）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](http://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](http://Short.compare(areaCode), pn.areaCode);
    if (result == 0)  {
        result = [Short.compare(prefix](http://Short.compare(prefix), pn.prefix);
        if (result == 0)
            result = [Short.compare(lineNum](http://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，组件将其内部数据和其他组件的其他实现细节隐藏起来。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 (条目 24)。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 (属性、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 (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（条目 86 和 87），则这些属性可以“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（条目 19）。 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。

公共类的实例属性很少公开 (条目 16)。 如果一个实例属性是非 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属性中的值的能力。这意味着你放弃了执行涉及该属性的不变量的能力。另外，当属性被修改时，就放弃了采取任何操作的能力，**因此公共可变属性的类通常不是线程安全的** 。即使属性是 final 的，并且引用了一个不可变的对象，通过使它公开，你就放弃切换到不存在属性的新的内部数据表示的灵活性。

同样的建议适用于静态属性，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 属性暴露常量。 按照惯例，这些属性的名字由大写字母组成，字母用下划线分隔（条目 68）。 很重要的一点是，这些属性包含基本类型的值或对不可变对象的引用（条目 17）。 包含对可变对象的引用的属性具有非 final 属性的所有缺点。 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 数组属性，或返回这样一个属性的访问器是错误的。** 如果一个类有这样的属性或访问方法，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属性的引用，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` 属性之外，公共类不应该有公共属性。 确保 `public static final` 属性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（条目 15）。 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。 着名的例子包括 `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，暴露 `Dimension` 的内部结构的决定是一个严重的性能问题，这个问题在今天仍然存在。

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。 公共累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它的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容易设计，实现和使用。 他们不太容易出错，更安全。

要使一个类不可变，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）。**

2. **确保这个类不能被继承。** 这可以防止粗心的或恶意的子类，假设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. **把所有属性设置为 final。** 通过系统强制执行，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从一个线程传递到另一个线程而没有同步，就必须保证正确的行为，正如内存模型[JLS，17.5; Goetz06,16]所述。
4. **把所有的属性设置为 private。** 这可以防止客户端获得对属性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共 `final` 属性或对不可变对象的引用，但不建议这样做，因为它不允许在以后的版本中更改内部表示（条目 15 和 16）。
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属性，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（条目 88）中进行防御性拷贝（条目 50）。

以前条目中的许多示例类都是不可变的。 其中这样的类是条目 11 中的 `PhoneNumber` 类，它具有每个属性的访问方法（accessors），但没有相应的设值方法（mutators）。 这是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（procedural）或命令（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并不会被破坏。 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（条目 1）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）（条目 50）。 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）（条目 13）。 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（条目 6）。

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量就容易多了。这一原则的特例是，不可变对象可以构成 `Map` 对象的键和 `Set` 的元素，一旦不可变对象作为 `Map` 的键或 `Set` 里的元素，即使破坏了 `Map` 和 `Set` 的不可变性，但不用担心它们的值会发生变化。

不可变对象提供了免费的原子失败机制（条目 76）。 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，特别是如果是大型的对象下。 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（条目 1）。 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` 被写入时，不可变类必须是有效的 `final`，因此它们的所有方法都可能被重写。不幸的是，在保持向后兼容性的同时，这一事实无法纠正。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“真实的” `BigInteger` 还是 `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）（条目 50）：

```
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（第 53 页的条目 11）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（条目 83）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（条目 67）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（条目 19），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”一词来表示实现继承（当一个类继承另一个类时）。 在这个项目中讨论的问题不适用于接口继承（当类实现接口或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到底是否时自用（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，则很有肯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？ 如果你不能如实回答这个问题，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，使用合成和转发代替继承，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. 如使用继承则设计，应当文档说明，否则不该使用

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。 那么为了继承而设计和文档说明一个类是什么意思呢？

首先，这个类必须准确地描述重写这个方法带来的影响。 换句话说，该类必须文档说明可重写方法的自用性（self-use）。 对于每个公共或受保护的方法，文档必须指明方法调用哪些重写方法，以何种顺序以及每次调用的结果如何影响后续处理。 （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。 本节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）。 更正式地说，如果这个集合包含一个或多个这样的元素，删除使得 `Objects.equals(o, e)` 的一个元素 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，则此实现将引发 `UnsupportedOperationException` 异常。

这个文档毫无疑问地说明，重写 `iterator` 方法会影响 `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。

但是，这是否违背了一个良好的 API 文档应该描述给定的方法是什么，而不是它是如何做的呢？ 是的，它确实！这是继承违反封装这一事实的不幸后果。要文档说明一个类以便可以安全地进行子类化，必须描述清楚那些没有详细说明的实现细节。

`@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 实用工具仍然会忽略它。

设计继承涉及的不仅仅是文档说明自用的模式。 为了让程序员能够写出有效的子类而不会带来不适当的痛苦，一个类可能以明智选择的受保护方法的形式提供内部工作，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）（条目 20）。 还有其他的情况显然是错误的，比如不可变的类（条目 17）。

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。

解决这个问题的最好办法是，在没有想要安全地子类化的设计和文档说明的类中禁止子类化。 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，那么你可能会通过禁止继承来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。

总之，设计一个继承类是一件很辛苦的事情。 你必须文档说明所有的自用模式，一旦你文档说明了它们，必须承诺为他们的整个生命周期。 如果你不这样做，子类可能会依赖于父类的实现细节，并且如果父类的实现发生改变，子类可能会损坏。 为了允许其他人编写高效的子类，可能还需要导出一个或多个受保护的方法。 除非你知道有一个真正的子类需要，否则你可能最好是通过声明你的类为 `final` 禁止继承，或者确保没有可访问的构造方法。

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（条目 18）。 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（条目 24）。

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（条目 20）。

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，**但是非常恶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（条目 34）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（条目 4）。 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. 优先使用类层次而不是标签类

有时你可能会碰到一个类，它的实例有两个或更多的风格，并且包含一个标签属性（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。 他们杂乱无章的样板代码，包括枚举声明，标签属性和 `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。 属性不能成为 final，除非构造方法初始化不相关的属性，导致更多的样板代码。 构造方法在编译器的帮助下，必须设置标签属性并初始化正确的数据属性：如果初始化错误的属性，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属性，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属性。

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，半径属性是属于圆的，长度和宽度属性都是矩形的。 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属性所占用。 所有的属性是 `final` 的。 编译器确保每个类的构造方法初始化其数据属性，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，上述层中的属性是直接访问的，而不是访问方法。 这里是为了简洁起见，如果类层次是公开的（条目 16），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明显标签属性的类，请考虑标签属性是否可以被删除，而类是否被类层次替换。 当遇到一个带有标签属性的现有类时，可以考虑将其重构为一个类层次中。

24. 优先考虑静态成员类

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（条目 34）。 `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（条目 7）。 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，所以它们必须保持短——约十行或更少——否则可读性将受损。

在将 lambda 表达式添加到 Java（第 6 章）之前，匿名类是创建小方法对象和处理对象的首选方法，但 lambda 表达式现在是首选（条目 42）。 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（条目 24）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（条目 15）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（条目 28）。

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。 你不仅可以将任何元素（除 `null` 以外）放入一个 `Collection<?>` 中，但是不能保证你所得到的对象的类型。 如果这些限制是不可接受的，可以使用泛型方法（条目 30）或有限制配符类型（条目 31）。

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` 不是合法的。

规则的第二个例外涉及 `instanceof` 操作符。 因为泛型类型信息在运行时被删除，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以任何方式影响 `instanceof` 运算符的行为。 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`。 这是一个强制转换，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。 它们仅用于与泛型引入之前的传统代码的兼容性和互操作性。 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
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（条目 26），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<?, ?>`（条目 26）。 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（条目 53）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（条目 32）。

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，可以使用选择器作为游戏模具，魔术 8 球或数据源进行蒙特卡罗模拟。 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 `Object` 的 `choose` 方法的返回值转换为所需的类型，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（条目 27）。

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，反之亦然。 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（条目 68）。

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）（条目 32）：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（条目 61）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（条目 29 和 68）：

```
// 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和编译时没有错误或警告：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和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）（条目 31），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（条目 28），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。 这种称为泛型单例工厂（generic singleton factory）的模式用于方法对象（ function objects）（条目 42），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（条目 59），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（条目 14）。 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](http://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>`（条目 55）。

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（条目 31）和模拟的自我类型用法（条目 2），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（条目 26）。

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 `Type`，`List<Type1>` 既不是 `List<Type2>` 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。 所以你也不能这样做吗?

```
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 C `ollection <? 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（条目 53）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。 这里是 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，应该使用cn，而在 C 中使用\n。%n 生成适用的特定于平台的行分隔符，该分隔符在很多但不是所有平台上都是\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（条目 28）。 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> .class` 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（条目 30）或限定的通配符（条目 31）来放置可以表示的类型的边界。

注解 API（条目 39）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（条目 27）。 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套装。 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（条目 3）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 Compa此处输入代码rable（条目 14）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 = 6.6730OE-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（条目 17）。 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（条目 16）。 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` 程序只需要打印一行少一行的表格。 什么是客户端程序引用删除的元素（在这种情况下，`Planet.Pluto`）？ 如果重新编译客户端程序，编译将会失败并在引用前一个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，允许包含枚举的类或包在与常量一起呈现时作出适当的反应。 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（条目 15）。

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（条目 24）。 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（条目 34）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（条目 55）。

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，如果没有给所有这些 `int` 值添加常量，也不能为某个 `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，一般来说使用 int 枚举模式（条目 34），下面将 2 的不同倍数赋值给每个常量：

```
// 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属性（bit field）的集合中：

```
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```

位属性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。 但是位属性具有 `int` 枚举常量等的所有缺点。 当打印为数字时，解释位属性比简单的 `int` 枚举常量更难理解。 没有简单的方法遍历所有由位属性表示的元素。 最后，必须预测在编写 API 时需要的最大位数，并相应地为位属性（通常为 `int` 或 `long`）选择一种类型。 一旦你选择了一个类型，你就不能超过它的宽度（32 或 64 位）而不改变 API。

一些程序员使用枚举优于 `int` 常量，当他们需要传递常量集合时仍然使用位属性。 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` 都表示为一个位矢量（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，所以它的性能与位属性的性能相当。 批量操作（如 `removeAll` 和 `retainAll`）是使用按位算术实现的，就像你为位属性手动操作一样。 但是完全避免了手动位混乱的丑陋和错误倾向：`EnumSet` 为你做了很大的努力。

下面是前一个使用枚举和枚举集合替代位属性的示例。 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（条目 64）。 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，所以没有理由用位属性来表示它。 `EnumSet` 类将位属性的简洁性和性能与条目 34 中所述的枚举类型的所有优点相结合。 `EnumSet` 的一个真正缺点是，它不像 Java 9 那样创建一个不可变的 `EnumSet`，但是在即将发布的版本中可能会得到补救。 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（条目 35）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（条目 28），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` （条目 45）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`（条目 35），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（条目 33）。`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（条目 31），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`（条目 36）和 `EnumMap`（条目 37）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（条目 20）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（条目 62）。 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（条目 33）。 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**（条目 40，27）。 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（条目 10），甚至记得重写 `hashCode`（条目 11）。 不幸的是，我们倒霉的程序员没有重写 `equals`，而是重载它（条目 52）。 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。 每个祖母组合的 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（条目 10）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），不包含方法声明，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“序列化”）。

你可能会听说过标记注解（条目 39）标记一个接口是废弃过时的。 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，**标记接口定义了一个由标记类实例实现的类型；标记注解则不会。** 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.write` 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，它可以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较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，如果标记适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，请花点时间确定它是否应该是注释类型，是不是标记接口是否更合适。**

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。

---

github: https://github.com/sjsdfg/effctive-java-3rd-chinese