Effective Java

第一章 引言

第二章 创建/销毁对象

1. 用静态工厂方法代替构造器

2. 类提供一个共有的静态工厂方法，返回类的实例。
3. 注意：与工厂模式不同，并不直接对应。
4. 静态工厂方法比起构造器的优势
   a) **它们有名称：**
      i. 如果构造器的参数不能正确描述正被返回的对象，具有适当名称的静态方法更容易使用。
      ii. 当一个类需要多个相同签名的构造器时，就可以用静态工厂代替构造器
      例子：BigInterger.probablePrime()
   b) **不必每次调用它们的时候都创建新的对象**：单件模式或者享元模式。
      i. 静态工厂方法能为重复的调用返回相同的类，有助于类控制在哪个时间段存在哪些实例。被称为**实例受控的类**。
   c) 他们可以返回原返回类型的的任何子类型的实例。
      i. API 可以返回对象，又不会使对象的类变成共有的，适合基于接口的框架。
      ii. 被返回的对象由相关的借口精确指定。
      iii. 共有的静态工厂方法所返回的类不仅可以是非公有的，还可以随着每次调用发生变化，这取决于工厂方法参数值。
      iv. 静态工厂方法返回的对象所属的类，在编写该静态方法时可以不必存在（留给开发者实现）
   d) 在创建参数化实例的时候，它们使代码更为简单
      i. 例子：
      Map<String, List<String>> m = new Map<String, List<String>>();
      Map<String, List<String>> m = HashMap.newInstance(); //减少一次参数
5. 静态工厂方法的缺点
   a) 类如果不含公有或者受保护的构造器，就不能被子类化。
      i. 可能鼓励程序员使用复合，而非继承
   b) 它们与其他静态方法没有任何区别。
      i. 要想查明如何实例化一个类是非常困难的
      ii. 静态工厂方法的惯用名称
         1. ValueOf: 类型转换方法
         2. getInstance: 返回唯一的实例
         3. newInstance：保证每个返回的实例都与其他不同

4. getType/newType：返回工厂对象的类

2. 遇到多个构造器参数时要考虑构建器（建造者模式）

1. 重叠构造器模式：第一个构造器只有一个必要参数，第二个有一个可选参数，第三个有两个，以此类推，最后一个构造器有全部的可选参数。**创建实例的时候，选择最短的列表参数的构造器**

   结论：当有许多参数的时候，客户端代码会很难编写

```
private final int calories;      //                optional
private final int fat;           // (g)            optional
private final int sodium;        // (mg)           optional
private final int carbohydrate;  // (g)            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;
}
}
```

2. JavaBeans 模式：调用一个无参数构造器来创建对象，然后用 setter 方法设置每个必要的参数

   评价：**有严重的缺点，构造过程被分配到几个过程中，JavaBean 可能处于不一致的状态。需要程序员付出额外的努力来确保它的线程安全。**

3. 建造者方法：既能保证安全性，又能保证可读性。**最好一开始就使用 Builder 模式.**

```
        private final int servings;

        // Optional parameters - initialized to default values
        private int calories      = 0;
        private int fat           = 0;
        private int carbohydrate  = 0;
        private int sodium        = 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 carbohydrate(int val)
            { carbohydrate = val;  return this; }
        public Builder sodium(int val)
            { sodium = val;        return this; }

        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }

    private NutritionFacts(Builder builder) {
        servingSize  = builder.servingSize;
        servings     = builder.servings;
        calories     = builder.calories;
```

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

先调用类的 builder 方法创建一个 builder，再用 setter 设置各个参数(**注意使用 return this;可以构造参数链**)，最后调用 builder 返回一个类

a) Builder 可以进行域的检查，是否违反约束条件
b) Builder 可以有多个可变参数
c) Builder 方法可以自动填充域，也可以返回不同的对象
d) 使用泛型的 builder

```
// A builder for objects of type T
public interface Builder<T> {
    public T build();
}
```

e) Java 传统的抽象工厂实现是 Class 对象，用 newInstance()来 build。
   评价：newInstance()会主动调用无参数的构造函数，而且没有编译时错误，只能在 runtime 抛出异常。**这破坏了编译时的异常检查。**
f) Builder 模式的不足之处：必须先构建 Builder 对象。**可能有性能问题，必须在有很多参数时才适合使用。**

3. 用私有构造器或者枚举类型强化 Singleton 属性

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

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

a) 注意客户端可以使用**反射机制**来调用私有的构造方法：应该在类被要求创造第二个实例的时候抛出异常 AccessibleObject.setAccessble()

b) 公有属性的好处：组成类的成员的声明很清楚地表明了这个类是一个 Singleton

c) 工厂方法的好处：它提供了灵活性。在不改变 API 的前提下可以改变该类是否为 Singleton 的想法，**或者修改成每一个调用的线程都返回一个唯一的实例。**

d) 序列化：仅仅加上 implements Serializable 是不够的，需要所有域都是瞬时的，**而且提供一个 readResolve()方法防止假冒对象**。

```
// 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;
}
```

e) 编写单个元素的枚举类型：**更加简洁，无偿提供序列化机制，而且绝对防止多次实例化，**是目前最好的方法。

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

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

4. 通过私有构造器强化不可被实例化的类的特性

a) 让工具类包含私有构造器，应该加上注释

b) 副作用：**使该类不能被子类化**

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

5. 避免创建不必要的对象

a) 如果对象不可变，就始终可以重用。

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

错误的代码：会创建不必要的 String 实例

```
String s = "stringette";
```

b) 对于同时提供了静态工厂和构造器的不可变类，通常**使用静态工厂而不是构造器，以避免创建不必要的对象**

例子：Boolean.valueOf(String)总是优先于 Boolean(String)

c) 重用一直不会被改变的可变对象

```
public class Person {
    private final Date birthDate;

    // Other fields, methods, and constructor omitted
    // DON'T DO THIS!
    public boolean isBabyBoomer() {
        // Unnecessary allocation of expensive object
        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomStart = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomEnd = gmtCal.getTime();
        return birthDate.compareTo(boomStart) >= 0 &&
               birthDate.compareTo(boomEnd)   <  0;
    }
}
```

错误的代码：创建了不必要的 Date 对象

```
class Person {
    private final Date birthDate;
    // Other fields, methods, and constructor omitted

    /**
     * The starting and ending dates of the baby boom.
     */
    private static final Date BOOM_START;
    private static final Date BOOM_END;

    static {
        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_START = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_END = gmtCal.getTime();
    }

    public boolean isBabyBoomer() {
        return birthDate.compareTo(BOOM_START) >= 0 &&
               birthDate.compareTo(BOOM_END)   <  0;
    }
}
```

使用 Static 语句块能避免这个问题

d) 适配器对象：将功能委托给后备对象，为后备对象提供接口。
例如 Map 接口的 keySet()函数返回时不创建新的实例

e) 自动装箱：**优先使用基本类型而非装箱基本类型**

f) 通过创建对象来提升程序的清晰性，简洁性和功能性，通常是好事。

g) 除非**对象池里的对象非常重量级（数据库连接池）**，不然不要通过使用对象池来避免创建对象

6. 消除过期对象的引用

a) 过期引用
   i. 栈内部维护着对过期对象的过期引用，永远也不会被解除。导致内存泄露

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

   ii. 清空过期对象的好处：错误引用时立即抛出异常
   iii. **清空对象应该是一种例外而非规范。最好的方法是让包含此引用的变量结束其生命周期。**
   iv. 只要类自己管理内存，程序员就**应该警惕内存泄露**。

b) 缓存
   i. 可以考虑使用 WeakHashMap 来代表缓存
   ii. 使用后台线程来定期清理
   iii. 对于更复杂的缓存，使用 java.lang.ref

c) 监听器和回调：只保留弱引用

7. 避免使用终结方法 finalize()

a) **终结方法不可预测，很危险**
   i. 不能保证会及时被执行，甚至不能保证执行：time-critical 的任务不应该由终结方法执行。例如关闭已经打开的文件（文件描述符是很有限的资源）
   ii. 不能保证在所有的 JVM 上都得到同样的体验
   iii. 可能随意延迟其实例的回收过程
   iv. **不应该依赖终结方法来更新重要的持久状态，例如分布式系统的永久锁**
   v. 不要使用 System.gc()等函数

b) 如果在终结方法执行时抛出未捕获异常，则**此异常会被忽略，终结方法也将终止**

c) **非常严重的性能损失**

d) 应该使用显式的终止方法，并要求客户端在每个实例不再有用时调用这个方法。该实例必须记录自己是否被终止，终止后的调用要抛出异常。
例子：InputStream 的 close 函数， Timer 的 cancel 函数等

应该在 try-catch 块的 finally 部分调用这个显式终止方法

```
// try-finally block guarantees execution of termination method
Foo foo = new Foo(...);
try {
    // Do what must be done with foo
    ...
} finally {
    foo.terminate();  // Explicit termination method
}
```

e) 终结方法的正确使用
   i. 在对象的所有者忘记调用显式终止方法时，**充当安全网函数**。
   ii. 对象的本地对等体：终结方法释放本地对等体的重要资源

f) 子类覆盖终结方法时，需要显式调用父类的终结方法

```
// Manual finalizer chaining
@Override protected void finalize() throws Throwable {
    try {
        ... // Finalize subclass state
    } finally {
        super.finalize();
    }
}
```

g) 终结方法守卫者：匿名内部类，终结它的外围实例。用于子类忘记调用父类的终结方法的一种补救措施

```
// Finalizer Guardian idiom
public class Foo {
  // Sole purpose of this object is to finalize outer Foo object
  private final Object finalizerGuardian = new Object() {
    @Override protected void finalize() throws Throwable {
      ... // Finalize outer Foo object
    }
  };
  ...  // Remainder omitted
}
```

对于每一个带有自定义终结方法的非 final 公有类，都应该考虑使用这个方法。

第三章 对于所有对象都通用的方法

8. 覆盖 equals 时请遵守通用约定

a) 确定应该/不应该覆盖 equals 方法的情况
   i. 类的每个实例本质上是唯一的
   ii. 不关心类是否提供了“逻辑相等“的测试功能。
   iii. 超类已经覆盖了 equals，并且从超类继承过来的子类也是合适的
   例如：从 AbstractSet 继承 equals
   iv. 类是私有的或者包级私有的：**此时应该覆盖并抛出异常。**

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

b) 可以考虑覆盖 equals 方法：“值类”，程序员只关注是否逻辑相等

c) Equals 的通用约定

i. **自反性 reflexive, x.equals(x)必须返回 true**

违背此条会导致集合不能包含刚加入的方法

ii. **对称性 symmetric 对于任意非 null 的 x 和 y，x.equals(y)和 y.equals(x)必须返回相同的结果**

```
// Broken - violates symmetry!
public final class CaseInsensitiveString {
    private final String s;

    public CaseInsensitiveString(String s) {
        if (s == null)
            throw new NullPointerException();
        this.s = 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
}
```

这里违反了对称性，String.equals()方法不知道区分大小写的字符串，因此会返回 false。正确做法是去掉 String 的比较

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

iii. **传递性 transitivity，对于任何非 null 的对象 x,y 和 z，如果 x.equals(y)和 y.equals(z)都返回 true，那么 x.equals(z)也应该返回 true**

一个长例子：

1. 一个普通的 point 类

```
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
}
```

2. 拓展这个类，加入颜色点概念

```
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
}
```

3. 提供盲色对比

```
// Broken - violates transitivity!
@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;
}
```

4. 牺牲了传递性：两个颜色点等于一个普通点，但是颜色点之间不同

结论：我们无法在扩展可实例化的类的同时，既增加新的组件，又能保留 equals 约定。

一种权宜之计：**使用复合而非继承。**不再让 ColorPoint 拓展 Point，而是在 ColorPoint 里加入一个私有的 Point 域以及一个公有的视图

另外的解决方法：**在抽象类的子类里加入新组件不会违反 equals 约定，因为不能创建超类的实例。**

```
// 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) {
      if (color == null)
         throw new NullPointerException();
      point = new Point(x, y);
      this.color = 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
}
```

例子：TimeStamp 和 Date 类， 不能混合使用

iv. 一致性 consistency：如果两个对象相等，那么他们必须始终保持相等，直到其中有一个被修改过。

1. 不可变的类应该满足： **相等的始终相等，不等的始终不等**
2. 不能使 equals 依赖不可靠的资源，例如 URL 的 equals 可能需要访问网络

Equals()方法应该始终对驻留在内存里的对象执行确定性操作

v. 非空性 null-nullity：需要类型检查，不需要额外代码

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

d) 高质量 equals 方法的诀窍

i. **使用 == 操作符检查“参数是否为自身的引用”：性能优化**

ii. **使用 instanceof 操作符检查“参数是否为正确的类型”：类或者接口**

iii. **把参数转化为正确的类型：上一步确保成功**

iv. **对于每个关键域，检查是否匹配**

1. 对象引用域：递归调用 equals
2. Float/Double：调用 Double/Float.compare
3. Null 域的比较

```
(field == null ? o.field == null : field.equals(o.field))
```

v. 对于不可变的类使用范式

vi. **先从最有可能不一致、开销最低的域比较。不比较属于对象逻辑状态的域，例如 Lock。可以适当使用冗余域，能节省时间。**

vii. **使用单元测试来检查性质**

e) 其他的告诫

i. **必须覆盖 hashCode 方法**

ii. **不要企图让 equals 过于智能**

iii. **不要把 equals 里声明的 Object 替换成其他类型，否则不是 override 而是重载**

9. 覆盖 equals 时总要覆盖 hashCode

a) 约定

i. 对于任意一个对象，只要对象中涉及 equals 的关键域没有改变，所有的 hashCode 必须返回同样的整数结果

ii. 如果两个对象的 equals 返回相等，那么他们的 hashCode 也必须相等
违反此条会导致 HashMap 类无法使用

iii. 如果两个对象的 equals 返回不等，**最好返回截然不同的整数**
使散列表有较高的效率

b) 一个简单的解决办法

1. 把某个非零的常数值，比如说17，保存在一个名为result的int类型的变量中。

2. 对于对象中每个关键域f（指equals方法中涉及的每个域），完成以下步骤：

a. 为该域计算int类型的散列码c：

i. 如果该域是boolean类型，则计算(f ? 1 : 0)。

ii. 如果该域是byte、char、short或者int类型，则计算(int)f。

iii. 如果该域是long类型，则计算(int)(f ^ (f >>> 32))。

iv. 如果该域是float类型，则计算Float.floatToIntBits(f)。

v. 如果该域是double类型，则计算Double.doubleToLongBits(f)，然后按照步骤2.a.iii，为得到的long类型值计算散列值。

vi. 如果该域是一个对象引用，并且该类的equals方法通过递归地调用equals的方式来比较这个域，则同样为这个域递归地调用hashCode。如果需要更复杂的比较，则为这个域计算一个“范式（canonical representation）”，然后针对这个范式调用hashCode。如果这个域的值为null，则返回0（或者其他某个常数，但通常是0）。

vii. 如果该域是一个数组，则要把每一个元素当做单独的域来处理。也就是说，递归地应用上述规则，对每个重要的元素计算一个散列码，然后根据步骤2.b中的做法把这些散列值组合起来。如果数组域中的每个元素都很重要，可以利用发行版本1.5中增加的其中一个Arrays.hashCode方法。

b. 按照下面的公式，把步骤2.a中计算得到的散列码c合并到result中：

```
result = 31 * result + c;
```

3. 返回result。

i. 把冗余码排除在外：任何可以通过其他域计算的域都不应该包括在内

ii. 使用 31 是因为 31 有更好的性能：**31 * I = I << 5 – 1**

iii. 如果一个类是**不可变**的，而且计算散列码的开销也比较大，就应该**缓存散列码，要么在创建对象时，要么在第一次调用 hashCode 时**

iv. **不要试图从散列码计算中排除关键域作为提高性能的方法：可能导致散列函数效率变低**

v. **不要让散列码成为一个确切函数，任何客户端都不应该依赖散列码的准确返回值**

10. 始终要覆盖 toString()

a) **所有的子类都应该覆盖 toString()：让类使用起来更舒适**

b) **toString 应该返回对象所包含的所有值得关注的信息**

c) 对象太大或者包含的状态无法用字符串描述时，应该返回摘要

d) 是否应该在文档里指定返回值的格式

   a) 值类：应该，**如果指定了格式，就必须坚持，不能随意更改**

   b) toString 包含的信息应该有 API 提供访问

   c) 应该在文档里表现你的意图，如

```
/**
 * Returns the string representation of this phone number.
 * The string consists of fourteen 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".
 *
 * Note that there is a single space separating the closing
 * parenthesis after the area code from the first digit of the
 * prefix.
 */
@Override public String toString() {
    return String.format("(%03d) %03d-%04d",
                         areaCode, prefix, lineNumber);
}
```

如果你决定不指定格式，那么文档注释部分也应该有如下所

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

11. 谨慎地覆盖 clone()

a) Object 的 clone 方法是受保护的，所以实现 cloneable 接口并不能调用 clone，除非使用反射机制。

b) Cloneable 接口的作用：**决定了 Object 中受保护的 clone 方法的实现行为**
   i. **如果实现接口。Clone()会返回这个类的逐级拷贝**
   ii. **如果不实现：抛出 CloneNotSupportedException**
   iii. 实现了一种语言之外的机制：**无需调用构造器就可以创建对象**

c) 约定（比较弱，不是绝对的要求）
   i. x.clone(x) != x
   ii. x.clone().getClass == x.getClass()
   iii. x.clone().equals(x) == true

d) 不调用构造器的规定太强硬：可以使用构造器/类是 final 的时候可能直接返回构造器创建的对象

e) **如果覆盖了非 final 类的 clone 方法，就应该返回一个通过调用 super.clone()而得到的对象**

f) 对于实现了 Cloneable 的类，我们希望它提供公有方法

g) 例子：

```
@Override public PhoneNumber clone() {
    try {
        return (PhoneNumber) super.clone();
    } catch(CloneNotSupportedException e) {
        throw new AssertionError();  // Can't happen
    }
}
```

   i. **注意 clone 返回的是 PhoneNumber 类：永远不要让客户做任何类库可以为他们做到的事情**

h) 实际上，clone 方法是另一个**构造器**
   i. 不能伤害到原始对象
   ii. 正确创建被克隆对象中的约束条件
   iii. 例如：Stack 类的 clone 方法应该在栈里递归调用 clone
   iv. 如果某个域是 final 的，clone 方法禁止给 elements 域赋新值：**可能需要去掉 final 修饰符**

i) 最后一种办法：**先调用 super.clone，然后把所有的域置成空白状态，然后调用高层方法重现产生对象的状态：**性能不高

j) Clone 不应该在构造的过程中调用新对象的任何非 final 方法：可能导致不一致

k) 公有的 clone 方法
   i. 不应该抛出异常
   ii. 调用 super.clone
   iii. 修正任何需要修正的域

l) 如果要为了继承而覆盖 clone 方法应该
   i. 声明为 protected，抛出异常
   ii. 不能实现 cloneable 接口

m) Object 的 clone 没有同步，应该实现同步
n) 最好提供某些其他的途径来代替对象拷贝，或者不提供类似方法
   i. 拷贝构造器：public Yum(Yum yum)
   ii. 拷贝工厂：public static Yum newInstance(Yum yum)
   iii. 可以带一些其他参数
o) 作为一个专门为继承而设计的类，如果不能提供良好的 protected 方法，子类就不能实现 cloneable 接口

12. 考虑实现 Comparable 接口

a) 实现 Comparable 接口表明它的实例具有内在的排序关系。一旦类实现了 Comparable 接口，就可以和很多泛型算法进行协作。**如果你正在编写一个值类，并且具有非常明显的顺序关系，那么就应该坚决实现这个接口**
b) **compareTo 方法**约定
   i. 将当前对象与参数比较，当对象小于，等于或者大于参数时，返回一个负整数，0 和正整数。**无法比较或者类不同时则抛出 ClassCastException**。
   ii. 实现者必须满足自反性，传递性等性质
   与 equals 相同，**我们无法在扩展可实例化的类的同时，既增加新的组件，又能保留 compareTo 约定。**权且之计也可以在这里使用
   iii. compareTo 施加的顺序关系应该同 equals 保持一致
   iv. 如果一个域没有实现 compareTo 或者需要一个非标准的排序，可以使用显式的 Comparator 来代替

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

   注意这里实现的接口确保了比较时不能跨类型
c) 比较的顺序
   i. 从最关键的域开始，如果出现非 0 结果，则比较结束
   ii. 返回减法结果是确认相关的域不能为负数

第四章 类和接口

13. 使类和成员的可访问性最小化

a) 封装：模块之间只通过 API 通信，内部实现细节被隐藏。
   i. 解耦合，使模块并行开发
   ii. 能够快速调节性能
   iii. 降低了构建大型系统的风险
b) 第一规则：**尽可能使每个类或者成员不被外界访问**
   i. 对于顶级的非嵌套类和接口只有两种访问级别：**package-private** 和 **public**
   ii. 通过把类做成包级私有的，类就不是 API 的一部分，因此可以进行删改，否则就必须永远支持以确保兼容性
   iii. 如果某个包级私有的类只在一个类内部被用到，就应该考虑使它成为私有嵌套类（24 条）
   iv. 只有同一个包里的其他类真正需要访问一个成员的时候，才应该删除 private 修饰符，把它变成包级私有
   v. **受保护的成员也是 API 的一部分，需要永远得到支持**
   vi. 如果一个类实现了一个接口，所有相关的类方法也必须实现接口的公有函数
   vii. **不能为了测试，而将类或者成员变成包的 API 的一部分。应该把测试程序当做包的一部分运行**
   viii. **实例域绝不能公开**
      1. 等于放弃对该值进行限制的能力，也不是线程安全的
      2. 同样的逻辑也适用于静态域。可以加上 final 修饰符之后暴露常量。**常量应该大写并用下划线分开（54 条）**
      3. 具有公开的静态 final 数组，或者返回这种域的方法，**几乎都是错误的。**
         a) 把公有数组变成私有的，并增加公有的不可变列表

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

b) 访问方法返回私有数组的拷贝

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

14. 在公有类中使用访问方法而不是公有域

a) 如果类可以在它所在的包的外部进行访问，就**提供访问方法。**以保留将来改变类的内部表示法的灵活性
b) 如果类是包级私有或者嵌套私有的，暴露数据域并没有本质的错误

c) 公有类的域如果是不可变的，危害会小些

```
// 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
}
```

15. 使可变性最小化

a) 不可变类比可变类更容易设计，实现和使用。

b) 五条原则

i. **不要提供任何会修改对象状态的方法**

ii. **保证类不会被扩展：一般做法是让这个类成为 final**

iii. **使所有的域都是 final 的**

iv. **使所有的域都成为私有的**

v. **确保对于任何可变组件的互斥访问**：如果类具有指向可变变量的域，则必须确保该类的客户端无法获得指向这些对象的引用。并且永远不要用客户端提供的对象引用来初始化这样的域，**也不要提供任何访问方法返回该对象引用**

例子：Complex 类，这个不可变类的运算**返回新的实例而不是改变内部状态**

c) 不可变类的优势

i. 比较简单，确保构造器建立了这个类的约束关系，那么可确保这些约束关系在**整个生命周期都不会变化。**可靠地使用一个可变类是非常困难的

ii. 不可变对象本质上是线程安全的，他们不要求同步。**它们可以被自由的共享**

1. **不可变的类可以提供静态工厂，把频繁请求的实例缓存起来，当现有实例符合请求的时候就不必创建新的实例**
2. **永远不用进行保护性拷贝**
3. **可以共享这些类的内部信息**

iii. 不可变对象为其他的对象提供了大量的“构件”

iv. 不可变类的真正缺点是：**对于每一个不同的类都需要一个单独的对象**

例如：BigInteger 类，几百万位只有一位不同

1. 猜测一下经常会用到哪些多步骤的操作，然后将它们作为基本类型提供。不可变的类在内部可以更灵活

2. 最好的办法是提供一个**公有的可变配套类：String 的配套类是 StringBuilder**
3. 类绝对不允许自身被子类化：final，或者所有构造器包级私有
    a) 第二种方法比较灵活，适合之后的改良

d) 真正的要求：**没有一个方法能够对对象产生外部可见的改变**
   i. 许多对象拥有一个或者多个非 final 类，存储一些开销昂贵的计算结果

e) 告诫
   i. 如果选择让不可变类实现 Serializable 接口，并且它包含一个或者多个指向可变对象的域，就必须提供显式的 readResolve 或者 readObject 方法
   ii. 坚决不要为 get 方法添加 setter：**除非有很好的理由让类成为可变的，否则就应该是不可变**
   iii. 如果类不能做成不可变的，依然应该尽可能限制它的可变性。**除非有令人信服的原因把域做成非 final 的，否则每个域都必须是 final 的**
   iv. 构造器应该创建完全初始化的对象，建立其所有的约束关系，**不要提供额外的公有初始化方法，也不要提供重新初始化方法**

16. 复合优先于实现继承

a) **继承打破了封装性。**超类变化的时候会影响子类，除非是专门为继承设计的超类，有良好的拓展性和文档
   例子：HashMap 类拓展导致重复计数

b) 超类的自用性是实现细节，不保证在所有的发行版本中都保持一致

c) **超类在后续的发行版本获得新的方法，会影响子类**。例如：子类有一个签名相同但是返回值不同的函数，这样会导致子类无法通过编译。如果返回值相同，那么等同于 override。也不能保证子类的方法能够满足超类的约定

d) **复合与转发方法**
   i. 灵活，健壮，InstrumentedSet 类实现了 Set 接口并且拥有私有 Set 成员

```
// Wrapper class - uses composition in place of inheritance
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;
    }
}

// Reusable forwarding class
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(); }
}
```

ii. 这样的类叫做**包装类 wrapper class，或者说装饰者模式 decorator pattern**

iii. 包装类的缺点：不适合在回调框架下工作。**在回调框架，对象把自身的引用传递给其他的对象。**

e) **只有在子类真的是超类的子类型时，才应该是用继承**。否则，B 应该包含 A 的一个私有实例，并暴露一个较小的 API。

f) 如果在适合复合的时候使用了继承，那么会不必要的暴露细节。这样得到的 API 会限制在原始实现并限定了性能。**客户端甚至可以直接访问内部细节**。

g) **继承机制会把超类所有的缺陷传播到子类**，但是复合可以掩盖这些缺陷

17. 要么为继承而设计并提供文档，要么禁止继承

a) 该类的文档必须精确的描述覆盖每个方法所带来的影响：**自用性说明。**
b) 类必须说明在哪些情况下会调用可覆盖的方法
c) 按照惯例，**如果方法调用了可覆盖的方法，在它的文档末尾应该包含关于这些调用的描述信息，以"This implementation …"开头。这段描述关注该方法的内部实现细节。**这违反了封装性，因为文档里不应该关注实现细节

```
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 (o==null ? e==null: o.equals(e)), if the collection contains one or more such elements. Returns true if the collection contained the specified element (or equivalently, if the collection changed as a result of the call).

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.

d) 类可以通过钩子 hook 以便能够进入方法的实现内部。这样的方法可以是 protected 方法
e) 最佳实现：
   i. **在发布之前编写子类进行大量测试**
   ii. **对于并非为了安全的进行子类化而设计和编写文档的类，要禁止子类化**
      1. **声明 final**
      2. **构造器包级私有并使用静态工厂（更灵活）**
   iii. **如果一定要允许继承，一种合理的方法是确保这个类永远不会调用自身任何可覆盖方法，并在文档中说明。**使用 helper method，把可覆盖方法的代码体移到私有的方法并调用这个方法
f) 其他约束
   i. 构造器决不能调用可覆盖的方法，否则会导致意外
   ii. 谨慎的使用 Cloneable 接口和 Serializable 接口：clone 和 readObject 方法同样不能调用可覆盖的方法
   iii. 如果在一个为继承而设计的类实现 Serializable 接口，**就必须让 readResolve 方法成为受保护的方法：为了继承而暴露实现细节**

18. 接口与抽象类

a) 接口的优势
   i. **现有的类可以很容易被更新，已实现新的接口。**
   扩展抽象类就必须把抽象类放到类层次的高处。间接伤害类层次

ii. **接口是定义 mixin 的理想选择。**

Mixin：类除了它的基本类型，还可以实现这个 mixin 类型，以表明他提供了某些可供选择的行为，例如 comparable 接口。**抽象类不能成为 mixin，因为他们不能被更新到现有的类里**

iii. **接口允许我们构造非层次的类框架。**

一个人可以同时是 singer 和 songwriter，而且可以针对这种组合实现特殊的第三个接口。**这样避免了类的组合爆炸**。通过修饰者模式，接口可以安全地增强类的功能。

```
public interface Singer {
    AudioClip sing(Song s);
}
public interface Songwriter {
    Song compose(boolean hit);
}


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

iv. **对每一个接口都最好提供一个抽象的骨架实现类 skeleton implementation，把接口和抽象类的优势结合起来**

按照惯例，骨架实现被称为 AbstractInterface. 如果设计得当，骨架实现可以使程序员很容易提供他们自己的接口实现

```
// Concrete implementation built atop skeletal implementation
static List<Integer> intArrayAsList(final int[] a) {
    if (a == null)
        throw new NullPointerException();

    return new AbstractList<Integer>() {
        public Integer get(int i) {
            return a[i];  // Autoboxing (Item 5)
        }

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

例子里提供了一个匿名类，被隐藏在静态工厂的内部。

b) 骨架实现类

i. **它们为抽象类提供了实现上帮助，但又不强加“抽象类被用作类型定义”时所特有的严格限制**

ii. 如果预置的类无法拓展骨架实现类，这个类始终可以实现对应的接口。**实现了这个接口的类可以把接口方法的调用转发到一个内部私有类的实例上，这个实例扩展了骨架实现类。**这种方法叫做模拟多重继承。

iii. 编写骨架类

1. 研究接口，确定哪些方法是最为基本的。基本方法将成为抽象类的抽象方法，其他的方法需要抽象类提供具体的最简单的有效实现。

```
// Skeletal Implementation
public abstract class AbstractMapEntry<K,V>
        implements Map.Entry<K,V> {
    // Primitive operations
    public abstract K getKey();
    public abstract V getValue();

    // Entries in modifiable maps must override this method
    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<?,?> arg = (Map.Entry) o;
        return equals(getKey(),   arg.getKey()) &&
               equals(getValue(), arg.getValue());
    }
    private static boolean equals(Object o1, Object o2) {
        return o1 == null ? o2 == null : o1.equals(o2);
    }

    // Implements the general contract of Map.Entry.hashCode
    @Override public int hashCode() {
        return hashCode(getKey()) ^ hashCode(getValue());
    }
    private static int hashCode(Object obj) {
        return obj == null ? 0 : obj.hashCode();
    }
}
```

c) 抽象类的优势

i. **抽象类的演变要比接口的演变更加容易**

如果想在后续的发行版本里加入新的方法，抽象类始终可以实现具体方法，并且所有实现都能使用新方法。**接口一旦被公开发行并实现，就不能更改**

ii. 发行接口之前必须尽可能测试接口

19. 接口只用于定义类型

a) **常量接口是对接口的不良使用，不要这么做**

```
// Constant interface antipattern - do not use!
public interface PhysicalConstants {
    // Avogadro's number (1/mol)
    static final double AVOGADROS_NUMBER   = 6.02214199e23;

    // Boltzmann constant (J/K)
    static final double BOLTZMANN_CONSTANT = 1.3806503e-23;

    // Mass of the electron (kg)
    static final double ELECTRON_MASS      = 9.10938188e-31;
}
```

常量接口代表了一种承诺：在将来的发行版本里，如果这个类被修改不需要使用常量了，它依然必须实现这个接口。如果非 final 类实现了常量接口，它的子类也会被污染

b) 导出常量的其他方法

i. **如果常量与某个现有的类密切相关，就应该把这些常量添加到这个类或接口中。例如 Interger.MIN_VALUE。**可以考虑使用枚举类型或者不可实例化的工具类

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

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

  public static final double AVOGADROS_NUMBER   = 6.02214199e23;
  public static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
  public static final double ELECTRON_MASS      = 9.10938188e-31;
}
```

ii. 如果不想使用类名修饰常量名，应该利用 Java 静态导入机制

```
// 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
}
```

20. 类层次优于标签类

a) 标签类的缺点

i. 标签类破坏了**可读性。**

ii. 标签类占用了**额外内存**。

iii. 域不能做成 final 的

iv. 无法给标签类添加新的风格而不修改源代码

v. 实例没有提供任何关于其风格的线索

b) **标签类实际是类层次的简单效仿**

c) 把标签类转化成类层次

i. **为标签类的每个依赖标签值方法、数据域都定义一个包含抽象方法的抽象类**

ii. **不依赖标签值的方法和数据域放到父类中**

iii. **子类的所有域都应该是 final 的**

d) 类层次的好处

i. 简单清楚，可读性好

ii. 可以反应类型之前本质上的层次关系，有助于增强灵活性，并进行更好的编译时检查

e) **标签类几乎没有使用的时候，使用标签类就要考虑是否重构成类层次**

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

21. 用函数对象表示策略

a) **类似 C 语言的函数指针（策略模式），在 Java 里有函数对象，这种实例的方法是执行在其他对象上的某种操作。或者说是某种操作的具体策略。**

```
class StringLengthComparator {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

b) 策略接口：策略类应该 extend 的接口，需要使用泛型实现

```
// Strategy interface
public interface Comparator<T> {
    public int compare(T t1, T t2);
}
```

c) 策略类的具体实现

i. 大部分时候使用**匿名类**

ii. 如果策略类需要经常调用，就应该将这个类保存在一个私有静态 final 域中。

d) 宿主类：避免把具体策略导成公共类，**具体策略是宿主类的私有嵌套类**

```
// Exporting a concrete strategy
class Host {
    private static class StrLenCmp
            implements Comparator<String>, Serializable {

        public int compare(String s1, String s2) {
            return s1.length() - s2.length();
        }
    }

    // Returned comparator is serializable
    public static final Comparator<String>
        STRING_LENGTH_COMPARATOR = new StrLenCmp();

    ... // Bulk of class omitted
}
```

22. 优先考虑静态成员类

a) 嵌套类：被定义在另一个类的内部的类，存在目的是为它的外围类提供服务

b) 四种嵌套类

i. **静态成员类**

1. **最好看作普通的类，只是碰巧声明在另一个类之中，可访问外围类的所有域，遵循同样的访问限制**
2. 常见用法：
   a) **公有类的辅助类。**例如：AlertDialog.Builder 或者 Calculation.Operation.MINUS
   b) **代表外围实例的组件**。例子：Map 的内部 Entry 对象。

ii. **非静态成员类**

1. 每个非静态成员类的每个实例都与**一个外围实例**相关联。非静态成员类的实例可以调用方法或者 OuterClass.this 来获得外围实例的引用
2. **不能在没有外围实例的情况下创建非静态成员类的实例。**
3. 常见用法：**定义一个 Adapter**。例如类似 Set 或者 List 的类会使用非静态成员类上来实现它们的 iterator
4. **如果成员类不需要访问外围实例，就始终使用 static 修饰符**

iii. **匿名类**

1. 当且仅当匿名类出现在非静态环境中，它才有外围实例。但即使如此，它也不会有任何静态成员。
2. 匿名类不能实例化，不能使用 instanceof，不能拓展接口，不能调用任何非超类的成员。**必须保持简短**。
3. **动态创建函数对象（策略模式）。**
4. **过程对象：例如 Runnable**

iv. **局部类**

1. 用在任何可以创建局部变量的地方，**有名字，可以重复使用。**

第五章 泛型

23. 不要在新代码中使用原生态类型

a) 泛型的原生态类型：List<E>对应的是不带任何实际参数类型的 List

b) **泛型的优点：**

i. 插入元素时自带类型检查

ii. 删除元素是不需要进行手工转换

iii. **可以使用 for-each 循环，两种方法**

```
// for-each loop over a parameterized collection - typesafe
for (Stamp s : stamps) { // No cast
    ... // Do something with the stamp
}
```

或者无论是否使用传统的for循环也一样：

```
// for loop with parameterized iterator declaration - typesafe
for (Iterator<Stamp> i = stamps.iterator(); i.hasNext(); ) {
    Stamp s = i.next(); // No cast necessary
    ... // Do something with the stamp
}
```

iv. 使用原生态类型，会失掉泛型在安全性和表述性方面所有的优势

v. **List 与 List<Object>的区别：前者可以接受 List<String>，但是后者不行**

vi. 不确定类型的时候，应该使用**无限制的通配符类型**，而不是原生态类型，例如 Set<?>。 Collection<?>不能将任何元素放入其中，因为不能确定类型。

c) 例外

i. **类文字中必须使用原生态类型，比如 List.class**

ii. **泛型信息会在运行时被擦除，所以在 instanceof 后面可以使用 Set**

24. 消除非受检警告：unchecked warning

a) 如果无法消除警告，**同时可以证明引起警告的代码是类型安全的。**那么可以使用**@SupressWarnings(“unchecked”)**来禁止警告。**必须在每次使用这条注解的后面加注释，说明为什么要这么做**

b) **应该始终在尽可能小的范围内使用 SupressWarnings 注解，**永远不要在整个类上使用这条注解

c) Return 不能使用这条注解。**应该声明一个局部变量并注解**

```
@SuppressWarnings("unchecked")
public static <T> T cast(Object obj) {
    return (T) obj;
}
```

25. 列表优先于数组

a) **数组是协变的：如果 Sub 是 Super 的子类型，那么 Sub[]也是 Super[]的子类型**

下面的代码片段是合法的：

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

但下面这段代码则不合法：

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

b) 数组是具体化的，**知道运行时才知道并检查他们的元素类型约束。**而泛型在编译时强化类型约束，在运行时擦除类型信息，**不可具体化，指的是运行时包含的信息比编译时更少。不可具体化的类型的数组转换只能在特殊情况下使用。**

c) **创建泛型数组是非法的，并非类型安全，会抛出 ClassCastException 异常。**但是创建 List<?>或者 Map<?,?>的数组是合法的

List<String>是 Object 的子类，因此 List<String>也是 Object[]的子类。

解决方法：优先使用集合类型 List<E>

```
List<String>[] stringLists = new List<String>[1];  // (1)
List<Integer> intList = Arrays.asList(42);         // (2)
Object[] objects = stringLists;                    // (3)
objects[0] = intList;                              // (4)
String s = stringLists[0].get(0);                  // (5)
```

d) 一个使用泛型转化数组的标准方式

```
// List-based generic reduction
static <E> E reduce(List<E> list, Function<E> f, E initVal) {
    List<E> snapshot;
    synchronized(list) {
        snapshot = new ArrayList<E>(list);
    }
    E result = initVal;
    for (E e : snapshot)
        result = f.apply(result, e);
    return result;
}
```

26. 优先考虑泛型

a) 泛型 Stack 类的应用

通常，你将至少得到一个错误或警告，这个类也不例外。幸运的是，这个类只产生一个错误，如下：

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

如第25条中所述，你不能创建不可具体化的（non-reifiable）类型的数组，如E。每当编写用数组支持的泛型时，都会出现这个问题。解决这个问题有两种方法。第一种，直接绕过创建泛型数组的禁令：创建一个Object的数组，并将它转换成泛型数组类型。现在错误是消除了，但是编译器会产生一条警告。这种用法是合法的，但（整体上而言）不是类型安全的：

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

```
// Little program to exercise our generic Stack
public static void main(String[] args) {
    Stack<String> stack = new Stack<String>();
    for (String arg : args)

        stack.push(arg);
    while (!stack.isEmpty())
        System.out.println(stack.pop().toUpperCase());
}
```

b) **Java 并不是天生就支持泛型**。很多泛型如 ArrayList 必须在数组上实现；**为了提升性能**，很多其他泛型也在数组上实现，例如 HashMap 类

c) **不能创建基本类型的泛型**，可以通过装箱基本类型来实现。

d) E 必须是 Delayed 的子集，这被称为**有限制的类型参数**

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

27. 优先考虑泛型方法

a) 一个典型的泛型方法

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

b) 泛型方法的特性：**无需明确指定类型参数的值，不像调用泛型构造器一样。**这被称为**类型推导 type inference**

c) **泛型单例工厂**创建许多不可变但适合于多个不同的对象，需要编写一个静态方法，重复的给每个必要的类型参数分发对象。
下文例子中的未受检异常是可以接受的，因为所有的代码都在我们的控制之下。

```
// Generic singleton factory pattern
private static UnaryFunction<Object> IDENTITY_FUNCTION =
    new UnaryFunction<Object>() {
        public Object apply(Object arg) { return arg; }
    };

// IDENTITY_FUNCTION is stateless and its type parameter is
// unbounded so it's safe to share one instance across all types.
@SuppressWarnings("unchecked")
public static <T> UnaryFunction<T> identityFunction() {
    return (UnaryFunction<T>) IDENTITY_FUNCTION;
}
```

d) **递归类型限制：通过某个包含该类型本身的表达式来限制类型参数。**例子：通过 Comparable 接口拓展的 Max 方法

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

```
// Using a recursive type bound to express mutual comparability
public static <T extends Comparable<T>> T max(List<T> list) {...}
```

这个方法可以读作：**针对每个可以和自身比较的类型 T**

完整方法

```
// Returns the maximum value in a list - uses recursive type bound
public static <T extends Comparable<T>> T max(List<T> list) {
    Iterator<T> i = list.iterator();
    T result = i.next();
    while (i.hasNext()) {
        T t = i.next();
        if (t.compareTo(result) > 0)
            result = t;
    }
    return result;
}
```

28. 利用有限制通配符来提升 API 的灵活性

a) 假设我们有一个泛型 Number 的 Stack，我们想把 Integer 加入这个 Stack 中，尽管直观上是可以的，并且 push(intVal)也是可以的，但是下面这个方法是错误的

```
// pushAll method without wildcard type - deficient!
public void pushAll(Iterable<E> src) {
    for (E e : src)
        push(e);
}
```

因为泛型是不可具体化的类，Iterable<Integer>并不是 Iterable<Number>的子类

使用**有限制的通配符：<? extends E>**

```
// Wildcard type for parameter that serves as an E producer
public void pushAll(Iterable<? extends E> src) {
    for (E e : src)
        push(e);
}
```

b) 相对应的，如果想把元素 pop 到一个给定的集合，那么下面这个方法是错误的

```
// popAll method without wildcard type - deficient!
public void popAll(Collection<E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

原因相同。此时应该使用**有限制的通配符：<? super E>**

```
// Wildcard type for parameter that serves as an E consumer
public void popAll(Collection<? super E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

c) **原则：PECS， producer extends；consumer super。【函数调用者】生产数据给方法所在的实例：extends；【函数调用者】从实例里获得数据：super**

d) 因此，之前的合并集合操作，其签名应该是

```
public static <E> Set<E> union(Set<? extends E> s1,
                               Set<? extends E> s2)
```

e) 下面这个方法是错误的，类型推导无法得出正确的结果

```
Set<Integer> integers = ... ;
Set<Double> doubles = ... ;
Set<Number> numbers = union(integers, doubles);
```

这个时候我们需要显式声明类型

```
Set<Number> numbers = Union.<Number>union(integers, doubles);
```

f) 之前的 max 操作，其签名应该声明为

```
public static <T extends Comparable<? super T>> T max(
        List<? extends T> list)
```

为了从初始声明中得到修改后的版本，要应用PECS转换两次。最直接的是运用到参数list。它产生T实例，因此将类型从List<T>改成List<? extends T>。更灵活的是运用到类型参数T。这是我们第一次见到将通配符运用到类型参数。最初T被指定用来扩展Comparable<T>，但是T的comparable消费T实例（并产生表示顺序关系的整值）。因此，参数化类型Comparable<T>被有限制通配符类型Comparable<? super T>取代。comparable始终是消费者，因此使用时始终应该是**Comparable<? super T>**优先于**Comparable<T>**。对于comparator也一样，因此使

g) 无限制的通配符与无限制的类型参数

```
// 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);
```

i. **如果类型参数只出现一次，那就应该使用通配符**

ii. 无限制的类型参数**阻止调用者将任何非 null 的元素加入已有集合。**也即 List<?>

不能放入任何参数

iii. 解决方法：**使用辅助方法。**把泛型声明暴露给用户，在内部使用私有的实现

```
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)));
}
```

29. 优先考虑类型安全的异构容器

a) 问题：使用类型安全的方式访问数据库的所有列，避免使用原生态

解决：**将 Key 参数化，而不是将 container 参数化**

使用 Class<T>, T 这样的键值对，这被称为类型安全的异构容器

```
// Typesafe heterogeneous container pattern - implementation
public class Favorites {
    private Map<Class<?>, Object> favorites =
        new HashMap<Class<?>, Object>();

    public <T> void putFavorite(Class<T> type, T instance) {
        if (type == null)
            throw new NullPointerException("Type is null");
        favorites.put(type, instance);
    }

    public <T> T getFavorite(Class<T> type) {
        return type.cast(favorites.get(type));
    }
}
```

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

注意：

i. **Map 的值是 Object，这样不保证键值一样，但是我们的代码确保了这一点**

ii. **getFavorite 方法做了类型转换**

b) 该容器的局限性

i. 放入时有可能破坏键值约定

解决方法：放入时进行转换，同时节省了检查指针为空

```
// Achieving runtime type safety with a dynamic cast
public <T> void putFavorite(Class<T> type, T instance) {
    favorites.put(type, type.cast(instance));
}
```

ii. 不能用在不可具体化的类上，例如 List<String>.class 是不存在的

c) 使用注解 API（什么鬼，没看懂啊。。。。）

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

欢迎发邮件到 wc012@ie.cuhk.edu.hk 进行交流~