Effective Java 学习笔记

一、第 2 章 创建和销毁对象

1 用静态工厂方法代替构造器

使用 static 工厂方法，而不是构造函数创建对象：仅仅是创建对象的方法，并非 Factory Pattern

优点

命名、接口理解更高效，通过工厂方法的函数名，而不是参数列表来表达其语义

- Instance control，并非每次调用都会创建新对象，可以使用预先创建好的对象，或者对象缓存；便于实现单例；或不可实例化的类；对于 immutable 的对象来说，使得用 == 判等符合语义，且更高效；
- 工厂方法能够返回任何返回类型的子类对象，甚至是私有实现；使得开发模块之间通过接口耦合，降低耦合度；而接口的实现也将更加灵活；接口不能有 static 方法，通常做法是为其再创建一个工厂方法类，如 Collection 与 Collections；

缺点

- 仅有 static 工厂方法，没有 public/protected 构造函数的类将无法被继承；见仁见智，这一方面也迫使开发者倾向于组合而非继承；
- Javadoc 中不能和其他 static 方法区分开，没有构造函数的集中显示优点；但可以通过公约的命名规则来改善；

小结

static 工厂方法和 public 构造函数均有其优缺点，在编码过程中，可以先考虑一下工厂方法是否合适，再进行选择。

2 遇到多个构造器参数时要考虑用构建器

静态工厂和构造器的共同局限性：不能很好地扩展到大量的可选参数。

两种有缺陷的方法：

- 重叠构造器模式，提供多个包含可选参数的构造器。这种的客户端代码很难编写，且难以阅读，调用时不知道各个参数的含义。
- JavaBeans 模式，也就是构造时调用 setter 方法来设置每个参数。这种方式创建实例很容易，代码也容易阅读，但有着严重的缺点：无法保证构造时保证一致性，线程安全。

Builder 模式，既保证重叠构造器的安全性，也保证 JavaBeans 模式的可读性。这种方式不直接生成对象，让客户端利用所有必要参数调用构造器（或静态工厂）得到一个 builder 对象，在 builder 对象上调用 类似 setter 的方法来设置参数。最后，客户端调用无参的 build 方法来生成不可变的对象。

其他优点：

可以进行参数检查；

builder 可以有多个参数，一个参数对应一个 setter 方法，灵活方便；

单个 builder 可以创建多个对象；

结合泛型可以生成抽象工厂 Abstract Factory，避免传统抽象工厂的缺点：newInstance

调用无参数构造函数，不能保证存在；newInstance 方法会抛出所有无参数构造函数中的异常，而且不会被编译期的异常检查机制覆盖；

不足：在注重性能的情况下，创建构造器的开销会影响性能；代码冗长。

总结：在参数很多，特别当参数是可选时，可以使用 builder 模式。

3 用私有构造器或枚举类型强化 Singleton 属性

Singleton 是只能被实例化一次的类，单例模式。

实现 Singleton 两种方法，把构造函数私有化，并且通过静态方法获取一个唯一的实例。

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

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

工厂方法，实现单例模式的形式是一样的。优势是灵活，可以在改变其 API，改变类是否是 Singleton。

```
// 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;
}
```

这两种方法在这个获取的过程中必须保证线程安全、反序列化导致重新生成实例对象等问题。

实现 Singleton 的第三种方法，编写一个包含单个元素的枚举类型。该方法提供了序列号机制，绝对防止多次实例化，而且实现简洁。

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

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

4 通过私有构造器强化不可实例化的能力

对于一些只包含静态方法和静态域的类，不希望类被实例化，可以通过声明一个私有的构造函数来保证类不被实例化，抛出异常保证不被内部方法调用构造函数。其副作用是该类不能被子类化继承。

```
// 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

通过重用对象而不是每次需要时创建一个新对象。

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

导致创建很多不必要的对象，可以这样做：

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

若可不变类同时提供静态工厂方法和构造器，使用静态工厂方法可以避免创建不必要的对象，构造器会创建一个新对象。

还可以重用已知不会被修改的可变对象

```
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();
    }
    将不变的对象设为static，只加载一次，而不是每次调用时都创建新对象
    public boolean isBabyBoomer() {
        return birthDate.compareTo(BOOM_START) >= 0 &&
               birthDate.compareTo(BOOM_END)   <  0;
    }
}
```

- 提高性能：创建对象需要时间、空间，“重量级”对象尤甚；immutable 的对象也应该避免重复创建，例如 String；
- 优先使用基本类型而不是自动装箱基本类型 auto-boxing；
- 为了重用对象而故意不创建必要的对象是错误的，使用 object pool 通常也是没必要的；
- 不建议 lazy initialize，除非使用场景很少且很重量级；
- Map 的 keySet 方法，每次调用返回的是同样的 set 实例。没有必要创建 keySet 视图对象的多个实例；
- 需要 defensive copying 的时候，如果没有实施保护性拷贝创建一个新对象，将导致潜在的错误和安全漏洞；

6 消除过期的对象引用

例如，用数组实现一个栈，在 pop 时如果仅仅是移动下标，没有把 pop 出栈的数组位置引用解除，这些对象不会被回收，将发生内存泄漏。

当一个类管理了一块内存，用于保存其他对象（数据）时，例如用数组实现的栈，底层通过一个数组来管理数据，但是数组的大小不等于有效数据的大小，GC 器却并不知道这件事，需要对其管理的数据对象进行 null 解除引用。

只要类是自己管理内存，就要警惕内存泄漏问题，一旦元素被释放掉，该元素中包含的任何对象引用都应立即被清空。

内存泄漏的另外两个常见来源：

- 缓存
  解决方法，可以用 WeakHashMap 代表缓存，当缓存中的项过期后会被自动删除；通过后台线程来清除没用的项。

- 监听器和其他回调
  解决方法，使用监听器或回调的弱引用来引用他们，保证被垃圾回收。

7 避免使用 finalize 终结方法

终结方法通常不可预测也很危险，使用终结方法会导致行为不稳定、降低性能以及可移植性问题。

Java 在 try-finally 快中完成类似 C++的析构器操作。

- finalize 方法不同于 C++的析构函数，不是用来释放资源的好地方
- finalize 方法的执行并不保证被及时执行，甚至不保证被执行
- 使用 finalize 方法时，性能会严重下降

为了确保资源被终止，可以使用显示的终止方法，其典型例子是 InputStream、OutputStream 和 java.sqo.Connection 的 close（）方法，java.util.Timer 的 cancel 方法。

显示终止方法通常与 try-finally 结合使用，确保及时终止。在 finally 语句中调用显示的终止方法。

```
// 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
}
```

- 终结方法存在的意义
  充当安全网“safety net”的角色，避免对象的使用者忘记调用显式 termination 方法，尽管不能保证 finalize 方法及时执行，但是晚释放资源好过不释放资源。可以输出 log 警告有利于排查 bug；
  用于释放本地对等体 native peer，因为垃圾回收器不会知道它，不会被回收；但是如果 native peer 持有必须要释放的资源时，应该定义显式 termination 方法；

- 子类 finalize 方法并不会自动调用父类 finalize 方法（和构造函数不同），为了避免子类不手动调用父类的 finalize 方法导致父类的资源未被释放，当需要使用 finalize 时，使用 finalizer guardian 比较好：定义一个私有的匿名 Object 子类对象，重写其 finalize 方法，在其中进行父类要做的工作，因为当父类对象被回收时，finalizer guardian 也会被回收，它的 finalize 方法就一定会被触发。

二、第 3 章 所有对象都通用的方法

覆盖 object 类的非 final 方法（equals、hashCode、toString、clone、finalize）时需要遵守一些通用约定，否则该类无法与依赖于这些约定的类如 HashMap、HashSet 一起正常运作。

8 覆盖 equals

如果能不覆盖 equals 就不覆盖，满足其一就可：

- 类的每个实例本质上都是唯一的
- 不关心类是否提供了判断相等的方法，例如 random
- 超类已覆盖 equals，且对子类也适用

需要覆盖 equals 的时候：类具有自己特有的“逻辑相等”概念也就是判断相等的标准，且超类为覆盖 equals 来实现这种期望的行为。

对于枚举这种“值类”，用实例受控确保“每个值至多只存在一个对象”，逻辑相等于对象相同是一回事，因此不需要覆盖 equals。

重写 equals 时需要遵循的约定：

- Reflexive（自反性）: x.equals(x)必须返回 true（x 不为 null）
- Symmetric（对称性）: 当且仅当 x.equals(y)返回 true 时 y.equals(x)返回
- Transitive（传递性）: x.equals(y) && y.equals(z) ==> x.equals(z)
- Consistent（一致性）: 当对象未发生改变时，多次调用应该返回同一结果
- x.equals(null)必须返回 false

传递性

一个类扩展了可实例化的类，同时又增加新的值组件，会违反 equals 传递性约定，如 java.sql.Timestamp。不要混合使用 java.sql.Timestamp 和 java.util.Date，否则会引起不正确的行为。

在抽象类的子类中增加新的值组件，不会违反 equals 传递性约定。因为不会创建抽象类实例。

一致性，不要使 equals 方法依赖于不可靠的资源，否则很难保证一致性。

实现高质量 equals 的诀窍

- 使用==检查是否引用同一对象，提高性能
- 用 instanceof 再检查是否正确类型
- 把参数强制转换为正确的类型
- 再对各个域进行 equals 检查，参数中域与对象中对应的与是否匹配
  域的比较顺序，最先比较最有可能不一致的域
- 确认其语义正确（对称的、传递的、一致的），编写测例
- 覆盖 equals 时，同时也覆盖 hashCode
- 不要将 equals 声明中 Object 对象替换为其他类型！重写 equals 方法，**传入的参数是 Object**

9 覆盖 equals 时总要覆盖 hashCode

如果两个对象 equals 是相等的，它们的 hashCode 值也必须相同。

如果覆盖 equals 而不覆盖 hashCode 方法，会导致该类无法和基于散列的集合类一起正常运作，如集合类 HashMap、HashSet 等。

hashCode 要做到：equals 相等的对象的 hashCode 相同，不 equals 的对象 hashCode 不一样。实现不 equals 对象产生不相等的散列码的方法：

1.取一个素数，例如 17，result = 17

2.对每一个关键的 field（在 equals 中参与判断的 field）f，为该域计算 int 类型散列码 c

- boolean: f ? 1 : 0
- byte/char/short/int: (int) f
- long: (int) (f ^ (f >> 32))
- float: Float.floatToIntBits(f)
- double: Double.doubleToLongBits(f)，再按照 long 类型计算
- Object: f == null ? 0 : f.hashCode()
- array: 先计算每个元素的 hashCode，再把散列值组合起来
- 对每个 field 计算的 c，result = 31 * result + c

3.返回 result

4.编写单元测试

如果一个类是不可变的且 hashCode()开销较大，可以考虑把散列码缓存在对象内部，不必每次都重新计算散列码。

不要试图从散列码计算中排除掉一个对象的关键域来提供性能，这样可能导致散列表速度很慢。

10 覆盖 toString

增加可读性，简洁、可读，提供丰富的信息量

实现 toString 时需决定是否在文档中指定返回值的格式。

11 谨慎覆盖 clone

Cloneable 接口是一个 mixin interface，用于表明一个对象可以被 clone。但缺少 clone 方法，object 的 clone 方法受保护，需借助反射才能调用 clone 方法，而且可能反射调用失败。

Clone 方法的通用约定：

- x.clone() != x 为 true
- x.clone().getClass() == x.getClass()为 true：要求太弱，当一个非 final 类覆盖 clone 方法的时候，创建的对象一定要通过 super.clone()来获得，所有父类都遵循这条规则，如此最终通过 Object.clone()创建出正确类的对象，这种机制不是强制要求的。
- x.clone().equals(x)为 true

对于成员变量都是 primitive type 的类，直接调用 super.clone()，然后 cast 为自己的类型即可（重写时允许返回被重写类返回类型的子类，便于使用方，不必每次 cast）；

成员变量包含对象（包括 primitive type 数组），可以通过递归调用成员的 clone 方法并赋值来实现；

递归调用成员的 clone 方法也会存在性能问题，对 HashTable 递归调用深拷贝也可能导致 StackOverFlow（可以通过遍历添加来避免）；

优雅的方式是通过 super.clone()创建对象，然后为成员变量设置相同的值，而不是简单地递归调用成员的 clone 方法；

和构造函数一样，在 clone 的过程中，不能调用 non final 的方法，如果调用虚函数，那么该函数会优先执行，而此时被 clone 的对象状态还未完成 clone/construct，会导致 corruption。因此上一条中提及的“设置相同的值”所调用的方法，要是 final 或者 private。

重载类的 clone 方法可以省略异常表的定义，如果重写时把可见性改为 public，则应该省略，便于使用；如果设计为应该被继承，则应该重写得和 Object 的一样，且不应该实现 Cloneable 接口；多线程问题也需要考虑；

要实现 clone 方法的类，都应该实现 Cloneable 接口，同时把 clone 方法可见性设为 public，返回类型为自己，应该调用 super.clone()来创建对象，然后手动设置每个域的值

clone 方法太过复杂，如果不实现 Cloneable 接口，也可以通过别的方式实现 copy 功能，或者不提供 copy 功能，immutable 提供 copy 功能是无意义的。

另一个实现对象拷贝的好方法是提供拷贝构造函数，或者拷贝工厂方法，而且此种方法比 Cloneable/clone 方法具有更多优势。

设计用来被继承的类时，如果不实现一个良好的受保护的 clone 方法，那么其子类也不可能实现 Cloneable 接口。

12 考虑实现 Comparable 接口

compareTo 并没有在 Object 中声明，它是 Compareable 接口的唯一方法。实现了 Compareable 接口的对象的排序、搜索、计算极限值以及自动维护很简单。如排序：

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

实现 Compareable 接口可以与依赖于该接口的集合类进行协作。

compareTo 方法的返回值:

小于：负整数

等于：0

大于：正整数

无法比较：抛出 ClassCast Exception 异常

compareTo 方法的约定：

- sgn(x.compareTo(y)) == - sgn(y.compareTo(x))，当类型不对时，应该抛出 ClassCastException，抛出异常的行为应该是一致的
- transitive: x.compareTo(y) > 0 && y.compareTo(z) > 0 ==> x.compareTo(z) > 0
- x.compareTo(y) == 0 则对于任何 z 有 sgn(x.compareTo(z)) == sgn(y.compareTo(z))
- 建议，但非必须：与 equals 保持一致，即 x.compareTo(y) == 0 ==> x.equals(y)，如果不一致，需要在文档中明确指出

无法在用新的值组件扩展可实例化的类，同时保持 compareTo 约定，除非放弃面向对象的抽象优势。为一个实现 compareable 接口的类增加值组件时不要扩展（继承）这个类，而是使用组合，并提供原有对象的访问方法，以保持对约定的遵循。

如果一个类有多个关键域，要从最关键的域开始比较，直到出现不相等的结果或者所有域比较结束。

谨慎使用比较结果直接返回差值的方法，可能会溢出。

三、第 4 章 类和接口

13 使类和成员的可访问性最小化

一个模块设计的好坏在于对于外部模块来说是否隐藏其内部数据及实现细节。这种概念称为封装，软件设计基本原则之一。

信息隐藏/封装重要的原因：有效解耦，降低模块耦合度，使模块间可以独立地开发、测试、使用、理解和修改；加快开发速度；减轻维护负担，因为可以更快地理解模块，调试时不影响其他模块；有效调节性能，看哪些模块影响了系统性能；降低风险，整个系统不可用但模块仍可用。

实体的可访问性由实体声明位置、访问修饰符共同决定。

尽可能使每个类或成员不被外界访问。

- 没必要 public 的类都应该定义为 package private；如果一个类只在另一个类的内部用到，则应该将其定义为私有的内部类；
- 没有使用访问修饰符则该成员是包级私有的，包内的类都可以访问这个成员；protected 是子类可以访问，且是包内的类也可以访问这个成员；
- 对于公有类成员，当访问级别从包级私有变成 protected，会大大增加可访问性。受保护成员应尽量少用。
- 为了便于测试，可以适当放松可见性，但也只应该改为 package private，不能更高
- 成员不能是非 private 的，尤其是可变的对象。一旦外部可访问，将失去对其内容的控制能力，而且会有多线程问题。含有 public 可变域的类不是线程安全的。
- 成员不能是非 private 的，尤其是可变的对象。一旦外部可访问，将失去对其内容的控制能力，而且会有多线程问题。
- 暴露的常量不能是可变的对象，否则 public static final 也将失去其意义，final 成员无法改变其指向，但其指向的对象却是可变的（immutable 的对象除外），长度非 0 的数组同样也是有问题的，可以考虑将公有变成 private 每次访问时返回拷贝，或者使用 Collections.unmodifiableList(Arrays.asList(arr))增加一个公有的不可变列表。

除了 public static final 域的特殊情形外，public 类都不应该有 public 域，且要保证 public static final 域所引用的对象都是不可变的。

14 在公有类中使用访问方法而非公有域

公有域外部可以直接访问，无法改变其数据表示（也就是定义，以后想改名称等无法更改），失去了安全性。用 getter 和 setter 方法来代替。

```
// 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; }
}
```

如果是 package private 或者 private 则可以不必这样。

15 使可变性最小化

使类不可变的规则：

1．不提供可以改变本对象状态的方法

2．保证类不可被扩展（继承）

3．使所有的域都是 final 的

4．使所有的域都成为 private

5．确保对于任何可变组件的互斥访问。在构造函数、accessor 中，对 mutable field 使用保护性拷贝 defensive copy。

不可变对象是线程安全的，不要求同步。

可以共享不可变对象的内部信息。

不可变对象为其他对象提供大量构件。

缺点是每个不同的值都要有应单独的对象，调用改变状态的方法也会创建一个新的对象，对于重量级的对象，开销会很大。

保证 class 无法被继承，除了声明为 final 外，还可以将默认构造函数声明为 private 或 package private，并提供 public static 静态工厂方法来代替公有的构造器。

如果不可变类实现 Serializable 接口，并且它包含一个或多个指向可变对象的域，一定要实现 readObject/readResolve 方法，或者使用 ObjectOutputStream.writeUnshared/ ObjectInputStream.readUnshared。

如果类不能做成不可变的，仍应该尽可能地限制它的可变性。在非必要情况下，使每个域都是 final 的。

16 复合优先于继承

跨包继承或继承不是被设计为应该被继承的实现类，是很危险的。

继承会打破封装性，子类依赖于超类中特定功能的实现细节。超类变化，子类会被破坏。即便代码都没有修改；而设计为应被继承的类，在修改后，是应该有文档说明的，子类开发者既可以得知，也可以知道如何修改。

HashSet 例子，查询从它创建以来添加了多少个元素。

```
// 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;
    }
}
```

结果不正确，addall（）调用子类的 add 方法，使得每个元素被计算了两次。

如果通过不重写 addAll 也只不对的，以后有可能 HashSet 的实现就变了。

好一点的做法，通过重新实现超类的方法，这样做超类的方法可能是自用的，非常耗时，容易出错。

后续版本超类会增加新方法。

为了解决这个问题，演示用复合或者称为组合，将要扩展的类变成新的类的私有的成员变量。新类中的每个实例方法都可以调用现有类实例中对应的方法，而不依赖于现有类的实现细节，这称为转发。

```
// 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)
```

InstrumentedSet 类被称作包装类，每一个 InstrumentedSet 实例都把另一 Set 实例包装起来。

```
// 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;
    }
}
```

包装类缺点：不适合用在回调框架中。

只有当子类真正是超类的子类型时才用继承，就是 is-a 关系，否则应该使用组合。

继承会暴露实现细节，可能会导致语义上的混淆，也可能修改超类。继承除了会继承父类的 API 功能，也会继承父类的设计缺陷，而组合则可以隐藏成员类的设计缺陷。包装类不仅比子类更加健壮，功能也更强大。

17 为继承而设计并提供文档说明，否则就禁止继承

一个类必须在文档中说明，每个可覆盖的方法，在该类的实现中的哪些地方会被调用。调用时机、顺序、结果产生的影响。

在发布类之前先编写子类对类进行测试。

为了允许继承，构造器决不能调用可被覆盖的方法，无论是直接调用还是间接调用。

父类的构造函数比子类的构造函数先执行，而如果父类构造函数中调用了可重写的方法，那么就会导致子类的重写方法比子类的构造函数先执行，会导致 corruption。

对于普通的类，既不是 final 也不是为了子类化而设计和编写文档的，要禁止子类化。两种禁止方法，把这个类声明为 final，另一种是把所有的构造器都变成私有的或者包级私有，并增加一些公有的静态工厂来代替构造器。

如果允许继承，要确保这个类永远不会调用它的任何可覆盖方法，并在文档中说明。也就是消除这个类中可覆盖方法的自用特性。

18 接口由于抽象类

现有的类可以很容易被更新以实现接口。Java 类只允许单继承，接口可以多继承，使用接口定义类型，使得 class hierarchy 类层次更加灵活

接口是定义 mixin（混合类型）的理想选择。Mixin 允许任选的功能可被混合到类型的主要功能中，抽象类不能被用于定义 mixin。需要增加一个功能的类只需要 implement 该接口即可，而如果使用抽象类，则无法增加一个 extends 语句。

接口允许构建非层次的类型框架。例如一个接口代表歌唱家，一个代表作曲家，有些歌

唱家本身也是作曲家，可以同时实现这个两个接口。

使用接口定义类型，可以使得 16 条中提到的 wrapper 模式安全地增强类的功能。

skeletal implementation 抽象骨架实现，把接口和抽象类的优点结合起来。

Simulated multiple inheritance 模拟多重继承：通过实现定义的接口，同时在内部实现一个匿名的 skeletal implementation，将对对该接口的调用转发到内部匿名类中，起到“多继承”的效果，同时避免了相应的缺陷。

simple implementation 简单实现：提供一个非抽象的接口实现类，提供一个最简单、有效的实现，也允许被继承。

使用接口定义类型的缺点：不便于演进，一旦接口发布且被广泛实现，如果想要改变实现增加功能（增加方法），则是不可能的。

总结

通过接口定义类型，可以允许多实现（多继承）

但是演进需求大于灵活性、功能性时，抽象类更合适

提供接口时，提供一个 skeletal implementation，同时谨慎考虑接口设计

19 接口只用于定义类型

仅仅用 interface 去定义一个类型。常量接口模式是对接口的不良使用。

```
// 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;
}
```

- 避免用接口去定义常量。如果常量与类或接口紧密相关，把这些常量添加到类或接口中；若常量最好被看作枚举，应该用枚举类型；否则，应该用不可实例化工具类 utility class 去定义常量。

20 类层次优于标签类

标签类有许多缺点，其中的样板代码破坏了可读性。标签类过于冗长、容易出错，且效率低下。

使用子类型化可以更好地定义能表示多种风格对象的单个数据类型。

将标签类转变成类层次：

首先为标签类的每个方法定义一个包含抽象方法的抽象类，方法行为依赖于标签值。

接着，为每种标签类定义根类的具体子类。每个子类中还包括针对根类中每个抽象方法的相应实现。

```
// 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; }

    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;
    }
    double area() { return length * width; }
}
```

类层次消除了标签类的缺点，使得代码简单清楚，每个类型的实现都有自己的类，不会受到不相关数据域的拖累。还可以用来反映类型之间本质上的层次关系，增强灵活性。

标签类很少适用，若遇到可以考虑将其重构成层次结构。

21 用函数对象表示策略

- 函数指针主要用来实现策略模式。只提供一个功能函数的类实例，没有成员变量，只需一个对象，为其功能定义一个接口，则可以实现策略模式，把具体策略传入相应函数中，使用策略
- 具体的策略实例只被使用一次时，通常使用匿名类定义并实例化这个策略类

22 优先考虑静态成员类

嵌套类是指被定义在另一个类内部的类，为其外围类提供服务。四种：静态成员类、非静态成员类、匿名类和局部类。除了第一种，后三个被称为内部类。

静态成员类

被声明在类内部的普通类，作为公有的辅助类。

嵌套类要独立于外围类的实例之外存在，嵌套类必须是静态成员类。

非静态成员类

将持有外部类实例的强引用，可以直接引用外部类的成员和方法

用处一：定义一个 Adapter，使得外部类的实例被看作是另一个不相关的类的实例，可以作为和外部类语义不同的实例来查看（访问），例如 Collection 的 Iterator。

如果嵌套类不需要引用外部类的成员和方法，则一定要将其定义为 static，避免空间/时间开销，避免内存泄漏。

匿名类

只能在被声明的地方进行实例化
无法进行 instanceof 测试
不能用匿名类实现多个接口
不能用匿名类继承一个类的同时实现接口
匿名类中新添加的方法无法在匿名类外部访问
不能有 static 成员
应该尽量保持简短
用处一：创建函数对象 function object
用处二：创建过程对象 process object，例如：Runnable, Thread, TimberTask
用处三：用于静态工厂方法内部，例如 Collections 类里面的一些工厂方法，很多是返回一个匿名的内部实现。

局部类

用的少
在任何生命局部变量的地方声明
有名字，可以被重复使用
不能有 static 成员
也应尽量保持简短

四、第 5 章 泛型

23 不要在新代码中使用原生态类型

泛型类或者接口，声明中具有一个或多个类型参数的类或者接口。
每个泛型都定义赢原生态类型 raw type，即不带任何实际类型参数的泛型名称。
Java 泛型从 1.5 引入，为了保持兼容性，实现的是伪泛型，类型参数信息在编译完成之后都会被擦除，其在运行时的类型都是 raw type，类型参数保存的都是 Object 类型， List<E> 的 raw type 就是 List。

泛型使得如果集合插入错误的类型的数据会产生编译时错误。

原生态 List 和参数化类型 List<Object>的区别：
前者逃避了泛型检查，不具备类型安全性，后者具备；
List<String> 是 List 的子类，但却不是 List<Object> 的子类；

当不确定类型参数，或者说类型参数不重要时，也不应该使用 raw type，而应该使用 List<?>。

相关术语

表5-1 本章条目中所介绍的术语

术 语	示 例	所在条目
参数化的类型	List<String>	第23条
实际类型参数	String	第23条
泛型	List<E>	第23，26条
形式类型参数	E	第23条
无限制通配符类型	List<?>	第23条
原生态类型	List	第23条
有限制类型参数	<E extends Number>	第26条
递归类型限制	<T extends Comparable<T>>	第27条
有限制通配符类型	List<? extends Number>	第28条
泛型方法	static <E> List<E> asList(E[] a)	第27条
类型令牌	String.class	第29条

24 消除非受检警告

编译器警告：非受检强制转化警告、非受检方法调用警告、非受检普通数组创建警告以及非受检转化警告。

要尽可能消除每一个非受检警告。

- 当出现类型不安全的强制转换时（一般都是涉及泛型，raw type），编译器会给出警告，首先要做的是尽量消除不安全的转换，消除警告
- 实在无法消除/确定不会导致运行时的 **ClassCastException**，可以通过 **@SuppressWarnings("unchecked")** 消除警告，但不要直接忽略该警告
- 使用 **@SuppressWarnings("unchecked")** 时，应该在注视内证明确实不存在运行时的 **ClassCastException**；同时应该尽量减小其作用的范围，通常是应该为一个赋值语句添加注解

```
// 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;
}
```

25 列表优先于数组

- 数组是 covariant(协变): 如果 **Sub** 是 **Super** 的子类，那么 **Sub[]** 也是 **Super[]** 的子类
- 泛型是 invariant(不变): 任意两个不同的类 **Type1** 和 **Type2**，**List<Type1>** 和 **List<Type2>** 之间没有任何继承关系。

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");
```

- 创建泛型数组是非法的；
- E、List<E>、和 List<String>这样的类型称为不可具体化类型，其运行时表示法包含的信息比它的编译时表示法包含的信息更少；
- 无限制的通配符类型如 List<?>和 Map<?,?>是唯一可具体化的；
- 有时为了类型安全，不得不做些妥协，牺牲性能和简洁，使用 List 而不是数组

```
// Naive generic version of reduction - won't compile!
static <E> E reduce(List<E> list, Function<E> f, E initVal) {
    E[] snapshot = list.toArray();  // Locks list
    E result = initVal;
    for (E e : snapshot)
        result = f.apply(result, e);
    return result;
}
```

- 将数组强制转换成不可具体化的类型是很危险的；
- 一般来说，不可将数组和泛型混合使用，使用 list 代替数组

26 优先考虑泛型

- 当需要一个类成员的数据类型具备一般性时，应该用泛型，这也正是泛型的设计场景之一，不应该用 Object 类
- 但使用泛型有时也不得不进行 cast，例如当泛型遇上数组；
- 当编写用数组支持的泛型时会不能创建不可具体化类型的数组，两种解决方法。第一，创建 Object 数组将其转换成泛型数组类型；第二，将 elements 域的数据类型从 E[]改为 Object[]；
- 总的来说把 suppress 数组类型强转的 unchecked warning 比 suppress 一个标量类型强转的 unchecked warning 风险更大，但有时出于代码简洁性考虑，也不得不做出妥协
- Java 原生没有 List，ArrayList 必须基于数组实现，HashMap 也是基于数组实现的
- 泛型比使用需要在客户端代码中进行进行转换的类型更加安全，把类做成泛型的对于新用户具有兼容性。

27 优先考虑泛型方法

静态工具方法适合于泛型化。

泛型用于指定方法的参数类型和返回值的参数类型。

```
// 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;
}
```

泛型方法的一个显著特性是，无需明确指定特定类型参数。

使用泛型静态工厂方法可以消除调用泛型构造器时代码的冗余，如

```
// Parameterized type instance creation with constructor
Map<String, List<String>> anagrams =
    new HashMap<String, List<String>>();
```

```
// Generic static factory method
public static <K,V> HashMap<K,V> newHashMap() {
    return new HashMap<K,V>();
}
```

通过这个泛型静态工厂方法，可以用下面这段简洁的代码来取代上面那个重复的声明：

```
// Parameterized type instance creation with static factory
Map<String, List<String>> anagrams = newHashMap();
```

泛型方法比客户端转换输入参数并返回值的方法更安全、容易。若需要方法不用类型转换就可以使用，可以将其泛型化。

28 利用有限制通配符来提升 API 的灵活性

参数化类型是不可变的。

使用有限制通配符可以解决参数化类型不可变问题，如

```
Stack<Number> numberStack = new Stack<Number>();
Iterable<Integer> integers = ... ;
numberStack.pushAll(integers);
```

Iterable<? Extends E>保证了 pushAll(integers)成功调用。

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

<? extends E> , <? super E> , PECS stands for producer-extends, consumer-super. 如果传入的参数是要输入给该类型数据的，则应该使用 extends，如果是要容纳该类型数据的输出，则应该使用 super。

如果编写将被广泛使用的类库，一定要适当使用通配符类型。记住 producer-extends, consumer-super，所有的 comparable 和 comparator 都是消费者。

29 优先考虑类型安全的异构容器

使用泛型时，类型参数是有限个的，例如 List<T> ， Map<K, V>。有时需要更灵活的方式安全的访问各种类型，例如可以放入数据库中任意类型的列的值。可以将键进行参数化而

不是容器参数化，然后把参数化的键交给容器来插入或获取值。用泛型系统来确保值的类型与它的键相符合。

Favorites的实现小得出奇。它的完整实现如下：

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

Favorites 类有两种局限性。首先，恶意的客户端若以它的原生态形式使用 Class 对象会破坏 Favorites 实例的类型安全；第二，它不能用在不可具体化的类型中

通过将类型参数放在键上而不是容器上可以避免容器只能有固定数目的类型参数的限制。对于类型安全的异构容器，可以用 Class 对象作为键。

五、第 6 章 枚举与注解

30 用 enum 代替 int 常量

Int 枚举模式的不足，类型安全和使用方便性方面存在不足。将一个 int 枚举传到另一个 int 枚举中进行比较，编译器也不会出现警告。若与枚举常量关联的 int 发生变化，客户端必须重新编译。遍历一组 int 枚举、获取枚举组的大小，都没有方便可靠的方法。

String 枚举模式同样存在不足，存在性能问题和将字符串常量硬编码到客户端代码中。

枚举类型可以用来代替，枚举类型是类型安全的枚举模式，枚举类型是 final 的，不能进行继承，只能声明枚举常量。

枚举是编译时类型安全的；

增加和重排枚举类型中的常量，客户端不用重新编译代码；

枚举类型还允许添加方法和成员变量，实现接口；

```
// Enum type with data and behavior
public enum Planet {
    MERCURY(3.302e+23, 2.439e6),
    VENUS  (4.869e+24, 6.052e6),
    EARTH  (5.975e+24, 6.378e6),
    MARS   (6.419e+23, 3.393e6),
    JUPITER(1.899e+27, 7.149e7),
    SATURN (5.685e+26, 6.027e7),
    URANUS (8.683e+25, 2.556e7),
    NEPTUNE(1.024e+26, 2.477e7);
    private final double mass;           // In kilograms
    private final double radius;         // In meters
    private final double surfaceGravity; // In m / s^2

    // Universal gravitational constant in m^3 / kg s^2
    private static final double G = 6.67300E-11;

    // Constructor
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
        surfaceGravity = G * mass / (radius * radius);
    }

    public double mass()           { return mass; }
    public double radius()         { return radius; }
    public double surfaceGravity() { return surfaceGravity; }

    public double surfaceWeight(double mass) {
        return mass * surfaceGravity;  // F = ma
    }
}
```

将不同的行为与每个枚举常量关联起来，可以声明一个抽象的 apply 方法，在常量的类主体中用具体的方法覆盖每个常量的抽象 apply 方法。

```
// Enum type with constant-specific method implementations
public enum Operation {
    PLUS   { double apply(double x, double y){return x + y;} },
    MINUS  { double apply(double x, double y){return x - y;} },
    TIMES  { double apply(double x, double y){return x * y;} },
    DIVIDE { double apply(double x, double y){return x / y;} };

    abstract double apply(double x, double y);
}
```

31 用实例域代替序数

不要根据枚举的序数导出与它关联的值，要将它保存在一个实例域中。

32 用 EnumSet 代替位域

位域实现方式不优雅、容易出错、没有类型安全性。

EnumSet 类可以用来表示从单个枚举类型中提取的多个值的多个集合，用来代替位域。

33 用 EnumMap 代替序数索引

使用 ordinal 方法得到的序数来作为数组的索引，存在许多问题，一般情况下不使用 ordinal 方法，且数组不能与泛型兼容，而且 ordinal 的值本来就不是表达 index 含义的，极易导致隐蔽错误，必须自己保重使用了正确的枚举序数。

可以用 EnumMap 来代替序数索引。

34 用接口模拟可伸缩的枚举

让 API 的用户提供自己的操作，扩展 API 提供的操作集，可以用枚举类型给操作码类型和枚举定义接口，利用接口的扩展性实现对枚举的扩展，允许用户编写自己的枚举来实现接口。

```
// 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;
    }
}
```

第二种方法时使用 Collection<? Extends Operation>作为参数的类型。

35 注解优先于命名模式

命名模式的缺点：

- 拼写错误没有提示，不能及时发现
- 无法保证命名模式只用于相应的程序元素上，例如可能有人以 **test** 开头命名测例类，方法却没有，JUnit 则不会运行测例
- 没有提供将参数值与程序元素关联起来的好方法，没有值/类型信息，编译器无法提

前发现问题

注解可以解决这些问题。

Test 注解只在方法声明中才是合法的，不能运用到类、域或其他程序元素上。

36 坚持使用 Override 注解

对每个覆盖超类的方法使用 Override 注解。

37 用标记接口定义类型

标记接口是没有包含方法声明的接口，只是指明一个类实现了具有某种属性的接口。

标记接口好于标记注解的地方：

定义了一个类型，标记注解没有这样的类型，可以在编译时捕捉后者运行时才能捕捉的错误。可以进行 instanceof 判断，可以声明参数类型；

可以被更加精确地锁定；

标记注解的好处：

可以通过默认方式添加一个或多个注解类型元素；

是注解机制的一部分，在支持注解的框架中具有一致性；

如何选择？

若标记是用到任何程序元素而不是类或接口，那就用注解；

若标记用于类和接口，看方法是否只接受这种标记，如果是则优先用标记接口。

作业、打印“ABC”

```
package loopprint;

import java.util.concurrent.TimeUnit;

public class PrintThread implements Runnable {

    private String name;

    private static String currentThread = "A";

    private Object o = new Object();

    public PrintThread(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        while (true) {
```

```
            synchronized (o) {
                if (currentThread.equals(name)) {
                    System.out.println(name);
                    if (currentThread.equals("A"))
                        currentThread = "B";
                    else if (currentThread.equals("B")) {
                        currentThread = "C";
                    } else if (currentThread.equals("C")) {
                        currentThread = "A";
                    }

                    try {
                        TimeUnit.SECONDS.sleep(1);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            }

        }
    }

    public static void main(String[] args) {

        Thread[] printtask = new Thread[3];
        String[] str = { "A", "B", "C" };

        for (int i = 0; i < 3; i++) {
            printtask[i] = new Thread(new PrintThread(str[i]));
            printtask[i].start();
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```