静态工厂方法

- **优点**
  - 与 `构造器` 相比，有 `名称`；
  - 与 `构造器` 相比，不必每次都创建新对象；
  - 与 `构造器` 相比，可以返回原返回类型的任何子类型；（如： `java集合框架` 的 `Collections` 类）

    发行版本1.5中引入的类java.util.EnumSet（见第32条）没有公有构造器，只有静态工厂方法。它们返回两种实现类之一，具体则取决于底层枚举类型的大小：如果它的元素有64个或者更少，就像大多数枚举类型一样，静态工厂方法就会返回一个RegalarEumSet实例，用单个long进行支持；如果枚举类型有65个或者更多元素，工厂就返回JumboEnumSet实例，用long数组进行支持。

    这两个实现类的存在对于客户端来说是不可见的。如果RegularEnumSet不能再给小的枚举类型提供性能优势，就可能从未来的发行版本中将它删除，不会造成不良的影响。同样地，如果事实证明对性能有好处，也可能在未来的发行版本中添加第三甚至第四个EnumSet实现。客户端永远不知道也不关心他们从工厂方法中得到的对象的类；他们只关心它是EnumSet的某个子类即可。

  - 与构造器相比，创建参数化类型实例时，代码更加简洁——这一点好像在jdk8中，并没有区别；
    书中**举例**：使用 `Map<String, List<String>> map = HashMap.newInstance()` 代替
    `Map<String, List<String>> map = new HashMap<String, List<String>>()`，理由是静态工厂方法可以进行类型推导（值得推荐）；

- **缺点**
  - 类如果不包含公有的或受保护的构造器，就不能被子类化；
  - 在查找方法的时候，名字不好找，不像构造器那样被特别标注；一般可以遵守默认命名规则： `valueOf/of`（一般用于类型转换）、 `getInstance`、 `getType/newType`（当静态工厂方法在不同类中时）等；

多参数时，考虑用构建器（ `builder` ）

- `重叠构造器` 模式：参数少时，比较好；参数太多后，果断放弃；
- `JavaBeans` 模式：缺点-使保证bean一致性变得困难(因为构造器过程被分到了几个调用中，在构造中 `JAVABean` 可能处于不一致的状态)；

```
//JAVABean模式
public class Temp2 {
  private int p1;//必要参数
  private int p2;
  private int p3;
  private int p4;

  public Temp2(int p1){this.p1 = p1; }
  public void setP1(int p1){ this.p1 = p1; }
  public void setP2(int p2){ this.p2 = p2; }
  public void setP3(int p3){ this.p3 = p3; }
  public void setP4(int p4){ this.p4 = p4; }
}

Temp2 t1 = new Temp2(1);
```

```
t1.setP2(2);
t1.setP4(4);
```

- `Builder` 模式：builder是它构建类的静态内部类，可先通过builder的 `setter` 方法设置属性，然后调用 `builder.build()` 方法，构建对象；

枚举实现单例模式

- 抵御通过 `反射机制` 生成第二实例的方法：构建第二实例的时候抛出异常;
- 单例模式中，如果类是可序列化的（实现 `Serializable` 接口），必须重写 `readResolve` 方法，不然，每次反序列化都会产生一个实例；
- `单元素的枚举类` 是最好的实现单例模式的方法——既可防止反射攻击，也可防止反序列化产生多实例;

使用私有构造器强化不可实例化能力

避免创建不必要的对象

- `String a = new String("string");` 此句创建了2次实例：参数 `string` 就是一个实例；
- 优先使用 `基本类型` ，而不是 `封装类型` ；
- 有时候 `重用对象` 会导致代码很乱，逻辑糟糕，比重建对象的代价更大：

> 不要错误地认为本条目所介绍的内容暗示着“创建对象的代价非常昂贵，我们应该要尽可能地避免创建对象”。相反，由于小对象的构造器只做很少量的显式工作，所以，小对象的创建和回收动作是非常廉价的，特别是在现代的JVM实现上更是如此。通过创建附加的对象，提升程序的清晰性、简洁性和功能性，这通常是件好事。

及时清除过期引用

- `缓存` 、 `监听器` /其他 `回调` 都比较容易发生内存泄漏

避免使用终结方法

- 从一个对象变得 `不可到达` 开始，到它的 `终结方法` 执行，所花费的时间是任意长的；所以，类似在终结方法中关闭文件的做法的错误的；
- 何时执行终结方法也是 `垃圾回收算法` 的一个功能，而垃圾回收算法在不同的 `jvm实现` 中会大相径庭，如果依赖 `finallizer` ，那么不同 `jvm` 中实现会截然不同；
- 有时候 `finallizer` 是否执行都不能保证：程序终止，而 `finallizer` 方法却没执行；
- 不要被 `System.runFinalizersOnExit()` 和 `Runtime.runFinalizersOnExit()` 诱惑，它们都有致命缺陷（多线程情况）；
- `Finallizer` 中的异常不会被打印，容易被忽略；
- `Finallizer` 增加性能损耗；
- 建议使用 `try...finally` ；
- 子类如果重写了终结方法（ `finallizer` ），则必须再调用超类的终结方法；终结方法守卫者可以防止粗心大意而没有执行 `super.finallizer` :

```
public class Parent {
  public static void main(final String[] args) throws Exception {
    doSth();
```

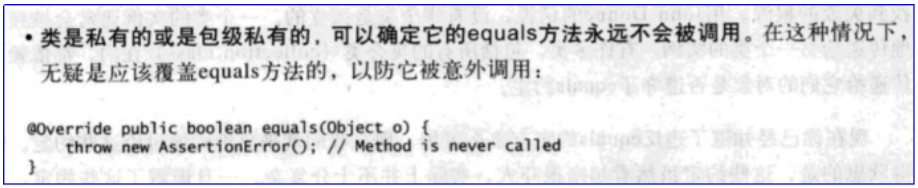
```
    System.gc();
    Thread.sleep(2000);
  }

  private static void doSth() {
    Child c = new Child();
    System.out.println(c);
  }

  @SuppressWarnings("unused")
  private final Object guardian = new Object() {
    @Override
    protected void finalize() {
      System.out.println("父类中匿名内部类--终结方法守卫者 重写的finalize()执行了");
      // 在这里调用Parent重写的finalize即可在清理子类时调用父类自己的清理方法
      parentlFinalize();
      // 注
      // Parent.this.finalize(); 这样写不对，会执行Child重写的finalize()方法
    }
  };

  private void parentlFinalize() {
    System.out.println("父类自身的终结方法执行了");
    // 一些逻辑..
  }

  @Override
  protected void finalize() {
    parentlFinalize();
  }
}
class Child extends Parent {
  @Override
  protected void finalize() {
    System.out.println("子类finalize方法执行了，注意，子类并没有调用super.finalize()");
    // 由于子类（忘记或者其他原因）没有调用super.finalize()
    // 使用终结方法守卫者可以保证子类执行finalize()时(没有调用super.finalize())，父类的清理方法仍旧调用
    // "finally中显式调用super.finalize()"没被执行之后的另一种保障对象被及时销毁的措施
  }
}
```

输出：

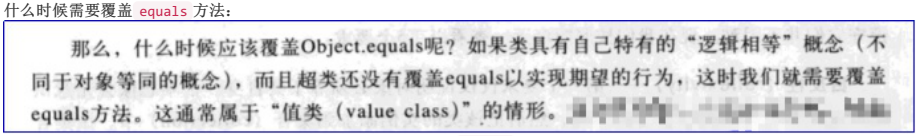
```
Child@131b92e6
子类finalize方法执行了，注意，子类并没有调用super.finalize()
父类中匿名内部类--终结方法守卫者 重写的finalize()执行了
父类自身的终结方法执行了
```

覆盖 `equals` 方法的通用约定

- 类的每个实例实质上都是唯一的；
- 不关心类是否提供 逻辑相等 的测试功能；
- 超类的 `equals` 方法也适合子类；
- 不明白：

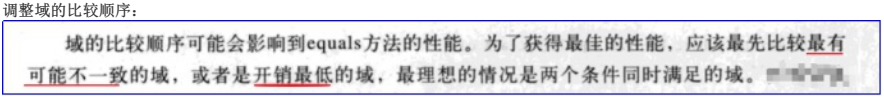


- 什么时候需要覆盖 `equals` 方法：

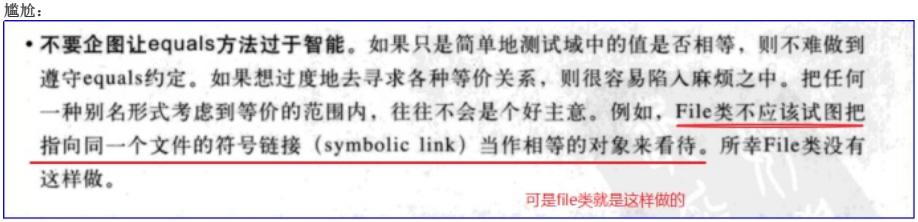


对于枚举类，逻辑相等和对象相等时一个意思，所以没必要覆盖 `equals` ；

- 覆盖 `equals` 需要遵守几个特性： `自反性` 、 `对称性` 、 `传递性` 、 `一致性` 、以及 `null` ；
- `氏替换原则` 简单粗暴的理解：任何基类可以出现的地方，子类一定可以出现；
- `Timestamp` 类（ `Date` 的子类，增加了 `nanoseconds` 域）和 `Date` 类不要混合使用，混合情况下回违反 `equals` 的 `自反性` ；
- `Equals` 优化：
  - 使用 `==` 检查对象引用;
  - 使用 `instanceof` 检查类型；
  - 把参数转化成正确的类型（如： `date` 转成 `long` ）；
  - 调整域的比较顺序：



- 重写 `equals` 的时候也要重写 `hashcode` ；
- 不要将 `equals(Object obj)` 中的 `Object` 替换为其他类型（如： `MyClass` ），这样就不是重写了，而是重载；添加 `@Override` 可以避免；
- 尴尬：



重写 `equals` 的时候也要重写 `hashcode`

- `Hashcode` 也有一致性；
- 如果 `equals` 返回 `true` ，那么 `hashcode` 也要相等，反之，不一定，但是不 `equals` 的对象，返回不同的 `hash` 值，有可能提高 `hash` 性能；
- 如果一个类是不可变的，并且计算 `hash` 码的开销也比较大，应该考虑将 `hash` 值缓存起来；
- 要慎重考虑计算 `hash` 值得时候舍弃某些字段的得与失；

最好子类都重写 `toString` 方法

- 如果 `tostring` 方法用于持久化，那么请确定长久规范；

谨慎重写 `clone`

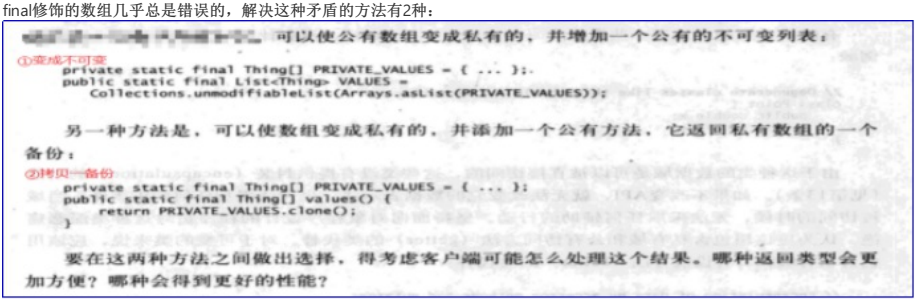
- 数组上调用 `clone` 返回的数组类型和原类型一样；
- `clone` 结构与指向可变对象的 `final` 域的正常用法是不兼容的（文中例子是一个数组 `elemnts` 的 `clone` ），除非原始对象和克隆对象之间可以安全地共享此可变对象。为了能够使一个类能够被克隆，请尽量将某个域的 `final` 修饰符去掉；
- 如果一个专为继承设计的类重写了 `clone` 方法，那么应该效仿 `object.clone` ：声明为 `protected` 、抛出 `CloneNotSupportedException` 异常、不能实现 `Cloneable` ，留给子类选择的空间；
- 线程安全的类，也要保证 `clone` 方法和其他方法一样————— `线程安全` ；

考虑实现Comparable

- 如果创建的类是一个值类，具有明显的内排序，就应该坚定地实现 `Comparable` 接口；
- `compareTo` 和 `equals` 不需要必须等效，比如： `BigDecimal("1.0")` 和 `BigDecimal("1.00");` 但是如果使用 `treeSet` 之类的集合，则只算一个元素；

类和成员可访问性最小

- 最大透明度，称为信息隐藏或封装，软件设计原则之一；
- 好处：解耦（开发、理解、测试、维护都比较容易）；
- 实例的域决不能是公有的；
- Final域应当只包含基本类型的值或不可变对象的引用；
- final修饰的数组几乎总是错误的，解决这种矛盾的方法有2种：



避免直接访问域

- 如果公有类暴露了它的数据域，要想在将来改变其内部表示法是不可能的；

使可变性最小化

- `不可变类` ：实例化后不再改变（如： `string` 、基本类型的包装类、 `BigInteger` 、 `BigDecimal` 等）；
- 不可变类更加容易设计、实现和使用，不易出错，更加安全；
- 设计不可变类原则：
  - 不提供修改状态的方法；
  - 保证类不被扩展（ `final` ）；
  - 所有域设为 `private` ；

    - 所有域设为 `final`；
    - 避免引用可变组件（如其他可变类引用）；

- 对于不可变类，本质上就没有拷贝的必要，所以是实现 `clone` 是不必要的， `String` 就是反面教材（jdk8中String#clone已经被移除）；
- 不可变类会造成性能的浪费（MutableBigInteger就是BigInteger的性能优化版）；
- 另外：

> 有关序列化功能的一条告诫有必要在这里提出来。如果你选择让自己的不可变类实现Serializable接口，并且它包含一个或者多个指向可变对象的域，就必须提供一个显式的readObject或者readResolve方法，或者使用ObjectOutputStream.writeUnshared和ObjectInputStream.readUnshared方法，即使默认的序列化形式是可以接受的，也是如此。否则攻击者可能从不可变的类创建可变的实例。这个话题的详细内容请参见第76条。

复合（ `composition` ）优先于继承

- 继承比较脆弱，如果新版本中添加新的方法很可能对子类造成很大影响，导致不稳定。复合不存在这种问题，复合类似于 `适配器` 模式；

要么为继承而设计，并提供文档说明，要么禁止继承

- 关于文档：好的 `api文档` 应该描述一个给定的方法做了什么，而不是如何做的；
- 构造器决不能调用可被重写的方法：

> 为了允许继承，类还必须遵守其他一些约束。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。如果违反了这条规则，很有可能导致程序失败。超类的构造器在子类的构造器之前运行，所以，子类中覆盖版本的方法将会在子类的构造器运行之前就先被调用。如果该覆盖版本的方法依赖于子类构造器所执行的任何初始化工作，该方法将不会如预期般地执行。

- 为继承而设计的类，应该慎重考虑实现 `Cloneable` 和 `Serializable` 接口；

接口优于抽象类

- 现有类易被更新，以实现新的接口；比如 `jdk` 添加 `Comparable` 接口的时候；
- 接口是定义 `mixin` （ `混合类型` ）的理想选择；
- 接口定义类型，抽象类（一般命名AbstractXXX，如AbstractList）搭建骨架：

> 实现了这个接口的类可以把对于接口方法的调用，转发到一个内部私有类的实例上，这个内部私有类扩展了骨架实现类。这种方法被称作模拟多重继承（**simulated multiple inheritance**）。

- 公有接口的设计一定要谨慎，一旦公开发行，并被广泛实现，再想修改接口，几乎是不可能的（不过 `jdk8` 中，接口可以有默认实现）；
- 接口实现起来比抽象类灵活，但设计了接口，最好定义一个骨架（抽象类）；

接口只用来定义类型

- 常量接口（只有 `final` 的静态域）是对接口的不良使用；

类层次优于标签类

- 标签类过于冗长，易出错，且效率低下：

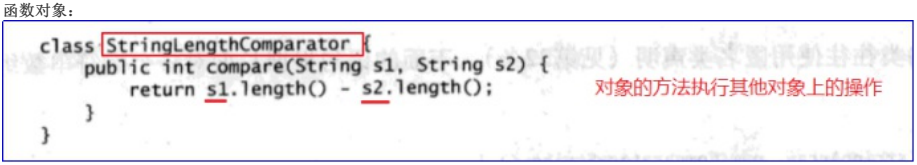
```
// Tagged class - vastly inferior to a class hierarchy
public class Figure1{
  enum Shape {
    RECTANGLE,
    CIRCLE
  }

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

  // These field are use only if shape if RECTANGLE
  double length;
  double width;

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

  // Constructor for circle
  public Figure1(double radius) {
    shape = Shape.CIRCLE;
    this.radius = radius;
  }

  // Constructor for rectangle
  public Figure1(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();
    }
  }
}
```

```
  /**
  * 类层次优于标签类
  * @author weishiyao
  *
  */
  // Class hierarchy replacement for a tagged class
  abstract class Figure2 {
    abstract double area();
  }

  class Circle extends Figure2 {
    final double radius;
```

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

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

class Rectangle extends Figure2 {
  final double length;
  final double width;

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

用函数对象表示策略

- 函数对象：



优先考虑静态成员类

唠唠其他

- 永远不要让客户去做任何类库能够替客户完成的事；