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；

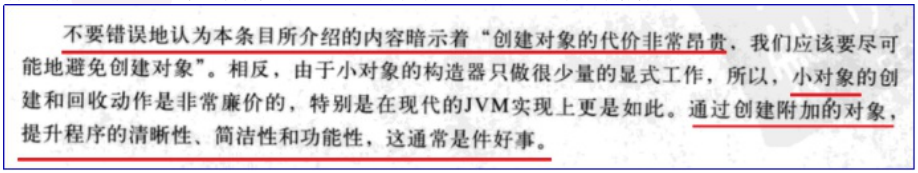
枚举实现单例模式

- 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：

> • **类是私有的或是包级私有的，可以确定它的equals方法永远不会被调用。**在这种情况下，无疑是应该覆盖equals方法的，以防它被意外调用：
>
> ```
> @Override public boolean equals(Object o) {
>     throw new AssertionError(); // Method is never called
> }
> ```

- 什么时候需要覆盖 `equals` 方法：

> 那么，什么时候应该覆盖Object.equals呢？如果类具有自己特有的“逻辑相等”概念（不同于对象等同的概念），而且超类还没有覆盖equals以实现期望的行为，这时我们就需要覆盖equals方法。这通常属于“值类（value class）”的情形。

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：

> 域的比较顺序可能会影响到equals方法的性能。为了获得最佳的性能，应该最先比较最有可能不一致的域，或者是开销最低的域，最理想的情况是两个条件同时满足的域。

- 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：

> • **不要企图让equals方法过于智能。**如果只是简单地测试域中的值是否相等，则不难做到遵守equals约定。如果想过度地去寻求各种等价关系，则很容易陷入麻烦之中。把任何一种别名形式考虑到等价的范围内，往往不会是个好主意。例如，File类不应该试图把指向同一个文件的符号链接（symbolic link）当作相等的对象来看待。所幸File类没有这样做。
>
> 可是file类就是这样做的

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`

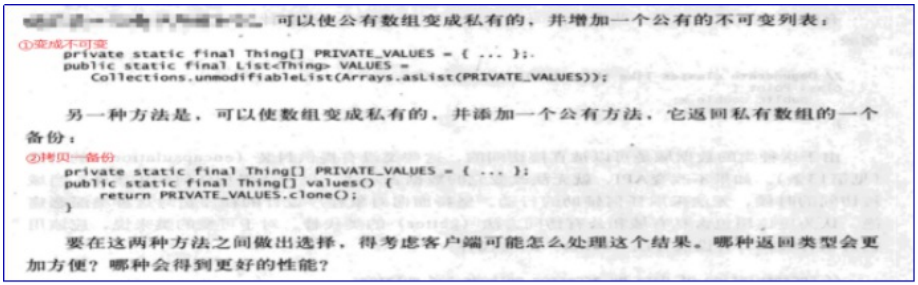
- 数组上调用 `clone` 返回的数组类型和原类型一样；
- `clone` 结构与指向可变对象的 `final` 域的正常用法是不兼容的（文中例子是一个数组 `elemnts` 的 `clone`），除非原始对象和克隆对象之间可以安全地共享此可变对象。为了能够使一个类能够被克隆，请尽量将某个域的 `final` 修饰符去掉；
- 如果一个专为继承设计的类重写了 `clone` 方法，那么应该效仿 `object.clone`：声明为 `protected`、抛出 `CloneNotSupportedException` 异常、不能实现 `Cloneable`，留给子类选择的空间；
- 线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



可以使公有数组变成私有的，并增加一个公有的不可变列表：

①变成不可变
```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final List<Thing> VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

另一种方法是，可以使数组变成私有的，并添加一个公有方法，它返回私有数组的一个备份：

②拷贝一备份
```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final Thing[] values() {
    return PRIVATE_VALUES.clone();
}
```

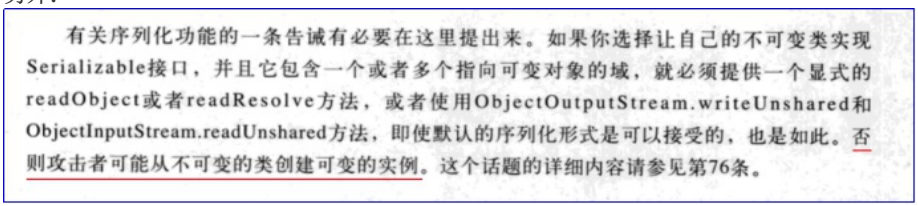
要在这两种方法之间做出选择，得考虑客户端可能怎么处理这个结果。哪种返回类型会更加方便？哪种会得到更好的性能？

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：



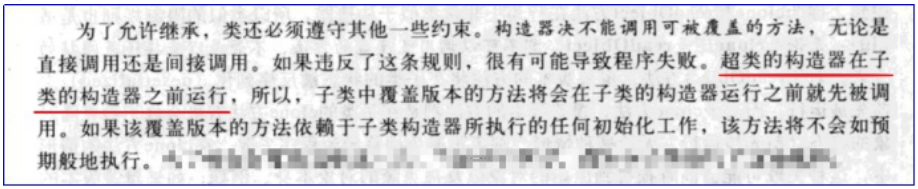
有关序列化功能的一条告诫有必要在这里提出来。如果你选择让自己的不可变类实现Serializable接口，并且它包含一个或者多个指向可变对象的域，就必须提供一个显式的readObject或者readResolve方法，或者使用ObjectOutputStream.writeUnshared和ObjectInputStream.readUnshared方法，即使默认的序列化形式是可以接受的，也是如此。否则攻击者可能从不可变的类创建可变的实例。这个话题的详细内容请参见第76条。

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：

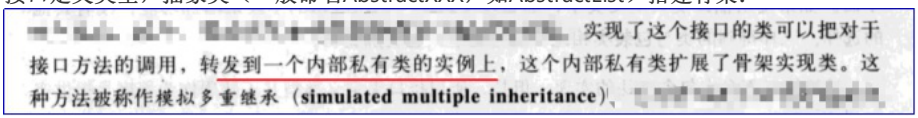


为了允许继承，类还必须遵守其他一些约束。构造器决不能调用可被覆盖的方法，无论是直接调用还是间接调用。如果违反了这条规则，很有可能导致程序失败。超类的构造器在子类的构造器之前运行，所以，子类中覆盖版本的方法将会在子类的构造器运行之前就先被调用。如果该覆盖版本的方法依赖于子类构造器所执行的任何初始化工作，该方法将不会如预期般地执行。

- 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,
```

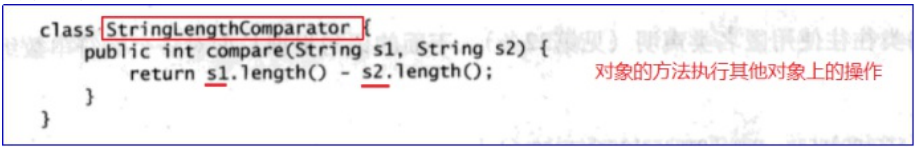
```
    CIRCLE
  }

  // 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

- **名词解释**

  - 定义在代码块、方法体内的类叫 `局部内部类`；
  - `函数对象` 做了这么一件事，我们可以定义一个只有方法而没有数据的类，然后把这个类的对象传递给别的方法，这时传递的这个对象就是一个函数对象。jdk8的

- **访问控制修饰符**
  - `default` (即缺省，什么也不写)：在同一包内可见，不使用任何修饰符。使用对象：类、接口、变量、方法；
  - `private` ：在同一类内可见。使用对象：变量、方法。 注意：`不能修饰类（外部类）` ；
  - `public` ：对所有类可见。使用对象：类、接口、变量、方法；
  - `protected` ：对同一包内的类和所有子类可见。使用对象：变量、方法。 注意：`不能修饰类（外部类）` ；

- 一个静态内部类的使用例子：

  静态成员类的一种常见用法是作为公有的辅助类，仅当与它的外部类一起使用时才有意义。例如，考虑一个枚举，它描述了计算器支持的各种操作（见第30条）。Operation枚举应该是Calculator类的公有静态成员类，然后，Calculator类的客户端就可以用诸如Calculator.Operation.PLUS和Calculator.Operation.MINUS这样的名称来引用这些操作。

  非静态成员类的一种常见用法是定义一个Adapter[Gamma95, p.139]，它允许外部类的实例被看作是另一个不相关的类的实例。例如，Map接口的实现往往使用非静态成员类来实现它们的集合视图（**collection view**），这些集合视图是由Map的keySet、entrySet和Values方法返回的。同样地，诸如Set和List这种集合接口的实现往往也使用非静态成员类来实现它们的迭代器（iterator）：

- 如果成员类不要求访问外围实例，就要始终添加 `static` 修饰符，因为非静态内部类总会保存一个外围实例的引用，保存这份引用会额外消耗时间和空间，并可能导致外围实例符合垃圾回收时却任然被保留。
- 当且仅当**匿名内部类**出现在 `非静态环境` 中时才包含外围实例的引用；（ `局部类` 也是如此）
- 即使**匿名内部类**在 `静态环境` 中，也不可能拥有任何静态成员；（ `局部类` 也是如此）
- 匿名内部类使用场景：
  - 函数对象；
  - 过程对象，eg： `Runable` 、 `Thread` 、 `TimerTask` 等；
  - 静态工厂方法的内部；

请不要在新代码中使用原生态类型

- 比如： `List<E>` 对应的原生态类型是 `List` ；
- 泛型有子类型化规则： `List<String>` 是 `List` 的子类，但不是 `List<Object>` 的子类；

泛型 术语	示例
参数化的类型	List<String>
实际类型参数	String
泛型	List<E>
形式类型参数	E
无限制通配符类型	List<?>
原生态类型	List
有限制类型参数	<E extends Number>
递归类型限制	<T extends Comparable<T>>
有限制通配符类型	List<? extends Number>
泛型方法	static <E> List<E> asList(E[] a)
类型令牌	String.class

消除非受检警告

- `SuppressWarning`可以用在任何力度的级别，应该始终在尽可能小的范围内使用`SuppressWarning`；

列表优先于数组

- 数组是`协变的`（`convariant`）
  - 如果Sub是Super的子类，那么Sub[]也是Super[]的子类；
- 泛型是`不可变`的（`invariant`）
  - 对于不同类型Type1和Type2，`List<Type1>`既不是`List<Type2>`的子类，也不是的父类；
- 相较于列表（list），数组（array）是有缺陷的：

```
Object[] objcetArray = new Long[1];
objcetArray[0] = "I don't fit in"; // 抛出ArrayStoreException
```

- 为什么创建泛型数组是非法的？(如：new ArrayList[10])

为什么创建泛型数组是非法的？因为它不是类型安全的。要是它合法，编译器在其他正确的程序中发生的转换就会在运行时失败，并出现一个**ClassCastException**异常。这就违背了泛型系统提供的基本保证。

但是无限制通配符类型和数组可以同用，比如：List、Map；

优先考虑泛型

- 下面是一个泛型

```
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]; // 这样写的话会报错

  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
}
```

上面的例子中`elements = new E[DEFAULT_INITIAL_CAPACITY];`会报错误或警告，是因为不能创建泛型数组（见"列表优先于数组"）；
解决`方案一`：

```
@SuppressWarnings("unchecked")  // 此处确定是安全的，可以抑制掉非受检警告
elements = (E[])new Object[DEFAULT_INITIAL_CAPACITY];
```

解决`方案二`：

```
// 定义elements为Object数组
```

```
private Object[] elements;

//修改pop方法
public E pop() {
    if (size == 0)
      throw new EmptyStackException();
    @SuppressWarnings("unchecked") // 此处确定是安全的，可以抑制掉非受检警告
    E result = (E)elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
  }
```

优先考虑泛型化

- 没有类型推导的jdk6中：

```
static <T> T pick(T a1, T a2) { return a2; }
// 比较特别的语法 this.<Serializable>
Serializable s = this.<Serializable>pick("d", new ArrayList<String>());
```

利用有限通配符来提升API的灵活性

- 如下代码：假如SubE是E的子类，则 `pushAll(Iterable<SubE>)` 在代码一的情况下，就会出错，因为 `参数化类型` 是 `不可变的` （ `List<SubE>` 既不是 `List<E>` 的子类，也不是 `List<E>` 的超类）；代码二就刚好解决这问题

```
// 代码一
public void pushAll(Iterable<E> src) {
  for (E e : src)
    push(e);
}

// 代码二
public void pushAll(Iterable<? extends E> src) {
  for (E e : src)
    push(e);
}
```

优先使用类型安全的异构容器

- 什么是类型安全的异构容器，我也没明白，读者自己体会，下面列出书中代码示例：

```
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) {
    // java.lang.Class<T>#cast
    return type.cast(favorites.get(type));
  }
}
```

书中提到 `Favorites` 有2个 局限性：

- 类型安全容易被破坏

Favorites类有两种局限性值得注意。首先，恶意的客户端可以很轻松地破坏Favorites实例的类型安全，只要以它的原生态形式（raw form）使用Class对象。但会造成客户端代码在编译时产生未受检的警告。这与一般的集合实现，如HashSet和HashMap并没有什么区别。你可以很容易地利用原生态类型HashSet（见第23条）将String放进HashSet<Integer>中。也就是说，如果愿意付出一点点代价，就可以拥有运行时的类型安全。确保Favorites永远不违背它的类型约束条件的方式是，让putFavorite方法检验instance是否真的是type所表示的类型的实例。我们已经知道这要如何进行了，只要使用一个动态的转换：

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

解决方案：添加类型转换检查

- 局限二没有很好的解决方案

Favorites类的第二种局限性在于它不能用在不可具体化的（non-reifiable）类型中（见第25条）。换句话说，你可以保存最喜爱的String或者String[]，但不能保存最喜爱的List<String>。如果试图保存最喜爱的List<String>，程序就不能进行编译。原因在于你无法为List<String>获得一个Class对象：List<String>.Class是个语法错误，这也是件好事。List<String>和List<Integer>共用一个Class对象，即List.class。如果从“字面（type literal）”上来看，List<String>.class和List<Integer>.class是合法的，并返回了相同的对象引用，就会破坏Favorites对象的内部结构。

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；