创建和销毁对象

1、考虑用静态工厂方法代替构造器

例如

```
public static Boolean valueOf(boolean b){
    return b ? Boolean.TRUE : Boolean.FALSE;
}
```

静态工厂方法相比构造器的优势：

(1)．他们有名称，代码易于阅读。

(2)．不必再每次调用它们的时候多创建一个新对象。可以避免创建不必要的重复对象。

(3)．它们可以返回原返回类型的任何子类型的对象，当我们在选择返回类型的类时具有更大的灵活性。

(4)．在创建参数化类型实例的时候，它们使代码更加简洁，例如

```
Map<String,List<String>> m = new HashMap<String,List<String>>(); //构造器方式
public static    HashMap<K,V> newInstance(){    //静态工厂方式
    return new HashMap<K,V>();
}
Map<String,List<String>> m    = HashMap.newInstance();
```

静态工厂的缺点在于：

(1)．类如果不含共有的或者受保护的构造器，就不能被子类化。

(2)．它们与其他的静态方法实际上没有任何区别。

2、遇到多个构造器参数时要考虑用构建器

```
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;
    public static class Builder {//类的静态成员类
        // Required parameters
        private final int servingSize;
        private final int servings;
        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;
        fat          = builder.fat;
        sodium       = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }

    public static void main(String[] args) {
        NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8).
            calories(100).sodium(35).carbohydrate(27).build();
    }
}
```

3、用私有构造器或者枚举类型强化 Singleton 属性

Jdk 1.5 版本之前，我们通常实现单例模式有两种方式：

(1). 通过公共的静态变量获取

```
public class Elivs{
    // 私有化构造器
    private Elivs(){}
    // 通过静态私有变量保存对象
    public static final Elivs INSTANCE = new Elivs();
  }
```

其主要优势在于组成类的成员的声明很清楚地表面了这个类是一个 Singleton:公有的静态域是 final 的，所以该域将总是包含相同对象引用

(2). 通过静态方法获取

```
public class Elivs{
    // 私有化构造器
    private Elivs(){}
    // 通过静态私有变量保存对象
    private static final Elivs INSTANCE = new Elivs();
    // 提供静态公共接口获取对象
    public static Elivs getInstance(){
        return INSTANCE;
    }
}
```

其优势在于，他提供了灵活性，可以在方法中扩展更多的操作，也可以在不改变 API 的前提下，改变类是否以 Singleton 进行创建。此外，如果是通过工厂方法获取对象的话，我们可以定义泛型来进行创建对象时的扩展。

(3). 在 java1.5 之后，提供了第三种 Singleton 的模式，就是 Enum 枚举类型

```
public enum Elvis{
    INSTANCE;
    public void leaveTheBuilding(){....}
}
```

这种方法在功能上与公有域方法相近，但是代码更加简洁，无偿地提供了序列化机制，绝对防止多次实例化，即使在面对复杂的序列化或者反射攻击的时候。

4、通过私有构造器强化不可实例化的能力

很多工具类，成员都是静态的，你写这个类的原因是想拿来直接用，而不需要实例化的。但是在缺少显示构造函数的时候，编译器会给你默认生成一个构造函数，这样的话，这个类就有可能实例化。

企图将类做成抽象类来强制该类不被实例化，这是行不通的（因为子类可以实例化，而且你写这个类不又不是用来继承的）

将构造器设置为 private 来解决问题：

```
public class UtilityClass {
    // Suppress default constructor for noninstantiability
    //错误是为了防止有人调用构造函数
    private UtilityClass() {
        throw new AssertionError();
    }
}
```

5、避免创建不必要的对象

Java 中 Sting 很特别，有如下两种初始化方式：

(1).String s1 = “This isstring1”;

(2).String s2 = new String(“Thisis string2”);

第一种字符串初始化方式，当有多于一个字符串的内容相同情况，字符串内容会放在字符串缓冲池中，即字符串内容在内存中只有一份。

第二种字符串初始化方式，不论有没有字符串值相同，每次都会在内存堆中存储字符串的值。

如果一个方法中字符串的值都相同，调用 100 万次情况下第一种字符串初始化方式的内存占用率很低，性能非常高，而第二种方式的字符串初始化则会占用大量的内存.

看下面一个例子，直观感受创建不必要对象的性能危害：

```
public class Person{
   private Date birthDate;
   //判断是否是婴儿潮出生的人
   public boolean isBabyBoomer(){
         Calendar cal = Calendar.getInstance(TimeZone.getTimeZone(“GMT”));
         //婴儿潮开始时间
         cal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
         Date boomStart = cal.getTime();
         //婴儿潮结束时间
         cal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
         Date boomEnd = cal.getTime();
         return birthDate.compareTo(boomStart) >= 0 && birthDate.compareTo(boomEnd) < 0;
   }
}
```

每次调用 isBabyBoomer()方式时，都需要创建 Calendar，TimeZone，boomStart 和 boomEnd 四个对象

可以看到，Calendar，TimeZone，boomStart 和 boomEnd 四个对象是所有调用者共用的对象，只需创建一份即可，改进之后代码如下

```
public class Person{
      private Date birthDate;
      private static final Date BOOM_START;
      private static final Date BOOM_END;
      static{
           Calendar cal = Calendar.getInstance(TimeZone.getTimeZone(“GMT”));
           //婴儿潮开始时间
           cal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
           BOOM_START = cal.getTime();
           //婴儿潮结束时间
           cal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
           BOOM_END = cal.getTime();
      }
      //判断是否是婴儿潮出生的人
      public boolean isBabyBoomer(){
           return birthDate.compareTo(BOOM_START) >= 0 && birthDate.compareTo(BOOM_END) < 0;
      }
}
```

6、消除过期的对象引用

```
public class Stock {
   private Object[] elements;
   private int size = 0;
   private static final int DEFAULT_INITIAL_CAPACITY = 16;

   public Stock() {
      elements = new Object[DEFAULT_INITIAL_CAPACITY];
   }

   public void push(Object e) {
      enSureCapacity();
      elements[size++] = e;
   }

   public Object pop() {
      if (size == 0)
         throw new EmptyStackException();
      return elements[--size];
   }

   private void enSureCapacity() {
      if (size == elements.length)
         elements = Arrays.copyOf(elements, 2 * size + 1);
   }
}
```

这段程序有一个“内存泄漏”问题，如果一个栈先是增长，然后再收缩，那么，从栈中弹出来的对象不会被当做垃圾回收，即使使用栈的程序不再引用这些对象，它们也不会被回收。这是因为，栈内部维护这对这些对象的过期使用（obsolete reference），过期引用指永远也不会被解除的引用。

修复的方法很简单：一旦对象引用已经过期，只需要清空这些引用即可。对于上述例子中的 Stack 类而言，只要一个单元弹出栈，指向它的引用就过期了，就可以将它清空。

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

Stock 为什么会有内存泄漏问题呢？问题在于，Stock 类自己管理内存。存储池中包含了 elements 数组（对象引用单元，而不是对象本身）的元素。数组活动区域的元素是已分配的，而数组其余部分的元素是自由的。但是垃圾回收器并不知道这一点，就需要手动清空这些数组元素。

一般而言，只要类是自己管理内存，就应该警惕内存泄漏问题。一旦元素被释放掉，则该元素中包含的任何对象引用都应该被清空。

7、避免使用终结方法（finallze）

通常只有在用于调用 native 方法的析构函数的时候才会使用，要避免使用这届方法

终结方法的缺点

(1)不能保证会被及时地执行。所以注重时间（time-critical）的任务不应该由终结方法来完成。例如：用终结方法来关闭已经打开的文件，这是严重错误，因为打开文件的描述符是一种很有限的资源。

(2) java 语言规范不仅不保证终结方法会被及时地执行，而且根本就不保证它们会被执行，所以，不应该依赖终结方法来更新重要的持久状态。例如：依赖终结方法来释放共享资源（比如数据库）上的永久锁，很容易让整个分布式系统垮掉。

```
public class ExitWithoutFinalizer {
    public void doSomething() {
        System.out.println("do something");
    }
    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("finalize " + this);
    }
    public static void main(String[] args) {
        ExitWithoutFinalizer ewf = new ExitWithoutFinalizer();
        ewf.doSomething();
        ewf = null;
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

(3)不要被 System.gc 和 System.runFinalization 这两个方法所诱惑，他们确实增加了终结方法的执行的机会，但是它们并不保证终结方法一定会被执行。

(4)使用终结方法有一个非常严重的性能损失。