

一、创建和销毁对象

第1条 考虑使用静态工厂方法代替构造器

对于类，让客户端获取它自身的实例的方法：提供公有的构造器，提供一个静态工厂方法返回类的实例的静态方法。

静态工厂方法相对于构造器的优势：

1. 有名称，使产生的客户端代码更易于阅读，避免多个构造器造成的混淆。
2. 不必在每次调它们的时候都创建一个对象，静态工厂方法能够为重复的调用返回相同的对象。
3. 可以返回原返回类型的任何子类型的对象。
4. 在创建参数化类型实例的时候，可以使代码变得更简洁。

如：

```
3*import java.util.HashMap;
6
7 public class FunctionTest {
8     Map<String, List<String>> m=new HashMap<String, List<String>>();
9
10    public static <K,V> HashMap<K,V> newInstance(){
11        return new HashMap<K,V>();
12    }
13
14    Map<String, List<String>> m1=FunctionTest.newInstance();
15
16 }
17
```

静态工厂方法的缺点：

1. 类如果不含有公有的或者受保护的构造器，就不能被子类化。（`private`修饰的构造器不能被子类实例化）
2. 它们与其他的静态方法实际上没有任何区别，因此要想查明如何实例化一个类比较困难。

所以一般统一静态工厂方法的一些惯用名称：

valueOf	返回的实例与参数具有相同的数值
of	valueOf的简写形式
getInstance	getInstance根据传入参数获得实例
newInstance	像getInstance一样，返回一个实例，可以保证返回不同的实例
getType	像getInstance一样，同返回对象类型
newType	像newInstance一样，返回对象类型

第2条 遇到多个构造参数时要考虑用构建器

当一个类中的可选域很多时，程序员一般考虑采用重叠构造器模式，但是这将导致客户端代码很难写，并且难以阅读，也有可能导致错误。当遇到许多构造器参数时，可以考虑以下方法：

1. 调用一个无参构造器来创建对象，然后调用setter方法来设置每个必要的参数，以及每个相关的参数。但是该模式可能会导致数据不一致的情况。
2. 使用构建器模式，这种模式不直接生成想要的对象，而是让客户端利用所有必要的参数调用构造器（或静态工厂方法），得到一个Builder对象。让后客户端在builder对象上调用setter方法来设置参数。

如下所示：

```
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 {
        // Required parameters
        private final int servingSize;
        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) { //必要参数
            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();
    }
}

```

总而言之，如果类的构造器或者静态工厂中具有多个参数，设计这种类时，**Builder**模式就是种不错的选择。

第3条 用私有构造器或者枚举类型强化Singleton属性

一般我们通过如下两种方式来实现**Singleton**，该方法要把构造器保持为私有的，并导出公有的静态成员，以便客户端能够访问该类的唯一实例。

```

public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { }

    public void leaveTheBuilding() {
        System.out.println("Whoa baby, I'm outta here!");
    }

    // This code would normally appear outside the class!
    public static void main(String[] args) {
        Elvis elvis = Elvis.INSTANCE;
        elvis.leaveTheBuilding();
    }
}

```

但是上面的单例模式的实现还存在一些问题，客户端可以通过**AccessibleObject.setObject**方法，通过反射机制调用私有构造器。

下面我们通过静态工厂模式来实现单例模式。对于下面的**Elvis1.getInstance()**方法对于所有的调用都只会返回一个对象引用。

```

public class Elvis1 {
    private static final Elvis1 INSTANCE = new Elvis1();
    private Elvis1() { }
    public static Elvis1 getInstance() { return INSTANCE; }
}

```

```

public void leaveTheBuilding() {
    System.out.println("Whoa baby, I'm outta here!");
}

// This code would normally appear outside the class!
public static void main(String[] args) {
    Elvis1 elvis = Elvis1.getInstance();
    elvis.leaveTheBuilding();
}
}

```

但是以上两种生成单例模式的方法在序列化时也可能导致新的问题，此时为了维护并保证Singleton，必须把所有实例域都transient，并提供一个readResolve方法。否则每次反序列化时都会创建一个新的实例。

```

public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() {}

    public void leaveTheBuilding() {
        System.out.println("Whoa baby, I'm outta here!");
    }

    private Object readResolve() {
        // Return the one true Elvis and let the garbage collector
        // take care of the Elvis impersonator.
        return INSTANCE;
    }

    // This code would normally appear outside the class!
    public static void main(String[] args) {
        Elvis elvis = Elvis.INSTANCE;
        elvis.leaveTheBuilding();
    }
}

```

此外从Java 1.5开始还实现了第三种实现Singleton的方法，就是编写一个包含单个元素的枚举类型。该方法在功能上与公有域方法相近，但是更简洁，并且也提供序列化而且防止多次实例化，即便在面对复杂的序列化或者反射攻击时依然可用。

```

public enum Elvis3 {
    INSTANCE;

    public void leaveTheBuilding() {
        System.out.println("Whoa baby, I'm outta here!");
    }

    // This code would normally appear outside the class!
    public static void main(String[] args) {
        Elvis3 elvis = Elvis3.INSTANCE;
        elvis.leaveTheBuilding();
    }
}

```

第4条 通过私有构造器强化不可实例化的能力

通过企图将类做成抽象类来强制该类不可实例化的方法是行不通，因为该类可以被子类实例化，正确的做法是让该类包含一个私有的构造器，它就不能把实例化了。如下所示：

```

// Noninstantiable utility class
public class UtilityClass {
    // Suppress default constructor for noninstantiability
    private UtilityClass() {
        throw new AssertionError();//保证该类在类的内部也不可调用构造器
    }

    public static void main(String[] args) {
        UtilityClass utilityClass=new UtilityClass();
    }
}

```

```
}  
}  
  
<terminated> UtilityClass [Java Application] C:\Program Files\Java\jre1.8.0_65\bin\javaw.exe (2016年11月11日 14:11:11)  
Exception in thread "main" java.lang.AssertionError  
    at chapter2.item4.UtilityClass.<init>(UtilityClass.java:7)  
    at chapter2.item4.UtilityClass.main(UtilityClass.java:11)
```

第5条 避免创建不必要的对象

使用String s="helloworld";来代替String s=new String("helloworld");

对于同时提供静态工厂方法和构造器的不可变类，通常使用静态工厂方法而不是构造器以免来创建不必要的对象。

除了重用不可变的对象之外，也可以重用那些已知不会被修改的对象，如下所示：

下面的代码问题在于，每次调用isBabyBoomer()方法的时候都需要创建gmtCal对象。

```
public class Person {  
    private final Date birthDate;  
  
    public Person(Date birthDate) {  
        // Defensive copy - see Item 39  
        this.birthDate = new Date(birthDate.getTime());  
    }  
  
    // Other fields, methods 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;  
    }  
}
```

使用静态代码块，在类被加载的时候只执行一次来创建对象，当以后需要使用到这些对象的时候直接调用而不需要再次创建这些对象。

```
class Person {  
    private final Date birthDate;  
  
    public Person(Date birthDate) {  
        // Defensive copy - see Item 39  
        this.birthDate = new Date(birthDate.getTime());  
    }  
  
    // Other fields, methods  
  
    /**  
     * 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();  
        BOOM_END = 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;
    }
}

```

要优先使用基本类型而不是装箱基本类型。如下面的代码所示：

在执行这段代码时，sum被申明成Long型，因此在执行代码的时候程序会构造2³¹个多余的Long实例，而将Long改成long会大大提高程序的执行效率。

```

public class Sum {
    // Hideously slow program! Can you spot the object creation?
    public static void main(String[] args) {
        Long sum = 0L;
        for (long i = 0; i < Integer.MAX_VALUE; i++) {
            sum += i;
        }
        System.out.println(sum);
    }
}

```

第6条 消除过期对象的引用

具有垃圾回收机制的语言一定要注意内存泄漏的问题，以下是容易造成内存泄漏的几个原因：

1. 过期引用，如下面的例子所示，从栈中被弹出的对象不会被当做垃圾回收，因为栈内部还对这些对象保持着过期引用（指永远不会再被解除的引用）

```

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

    public Stack() {
        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];
    }

    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
     */
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}

```

通过修改上面的红体字代码来解除过期引用，使得对象被弹出栈后可以通知垃圾回收器进行回收。

```

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

```

```
}
```

2. 缓存，可以使用WeakHashMap来代表缓存，当缓存项过期之后，他就会被自动删除。
3. 监听器和其他回调。我们可以使用WeakHashMap来保存他们的弱引用来解决该问题。

二、对所有的对象都通用的方法

第8条 覆盖equals时请遵守通用约定

equals要实现的是逻辑上的等。站在数学的角度来看，两个事物相等的条件，有如下几个：

- 1.自反性：对于任何非null的引用值x，x.equals(x)必须返回true。
- 2.对称性：对于非空的引用值x,y，当且仅当x.equals(y)返回true时，y.equals(x)必须返回true。

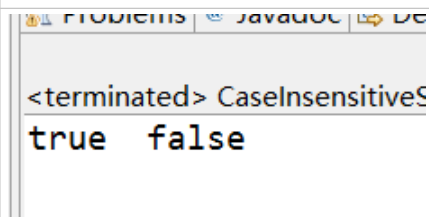
如下面的代码违反了对称性：

```
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;
    }

    public static void main(String[] args) {
        CaseInsensitiveString cis = new
        CaseInsensitiveString("Polish");
        String s = "polish";
        System.out.println(cis.equals(s) + " " + s.equals(cis));
    }
}
```



```
<terminated> CaseInsensitiveS
true false
```

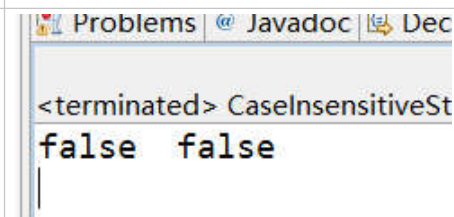
所以上面代码中重写的equals方法违反了对称性约定，编程时应该尽量避免。

```
public final class CaseInsensitiveString {
    private final String s;

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

    //This version is correct.
    @Override
    public boolean equals(Object o) {
        return o instanceof CaseInsensitiveString &&
            ((CaseInsensitiveString)o).s.equalsIgnoreCase(s);
    }

    public static void main(String[] args) {
        CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
        String s = "polish";
        System.out.println(cis.equals(s) + " " + s.equals(cis));
    }
}
```



```
<terminated> CaseInsensitiveSt
false false
```

- 3.传递性：对于任何非null的引用值x,y,z，如果x.equals(y)=true，y.equals(z)=true，那么x.equals(z)也必须返回true。

传递性的判断是x = y，y = z，那么就可以判断x = z了。现在假设我们有一个类Point和一个Point的子类ColorPoint分别如下：

```
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;
        }
    }
}
```

```
public class ColorPoint extends Point {
    private final Color color;

    public ColorPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }
}
```

```

    }

    Point p = (Point) o;

    return p.x == x && p.y == y;
}
}

```

可以看到ColorPoint继承于Point，不过比Point类多一个颜色属性。当我们把ColorPoint与Point和Point与ColorPoint进行比较，如下：

```

public class Main {
    public static void main(String[] args) {
        Point p1 = new Point(1, 2);
        ColorPoint cp1 = new ColorPoint(1, 2, Color.BLUE);

        System.out.println("p1 equal cp1 ? " + p1.equals(cp1));
        System.out.println("cp1 equal p1 ? " + cp1.equals(p1));
    }
}

```

```

<terminated> Main (18) [Java Applica
p1 equal cp1 ? true
cp1 equal p1 ? true

```

为什么两个都true呢？明明两个不同类型啊，如果真的要考虑父类与子类的关系，也应该是一个true一个false啊。因为这里我们的ColorPoint本身没有重写Point的equals，它使用的是Point的equals，这时无论哪一次的比较中，都是去比较x和y，与color无关。这样就会导致一个问题，如果我的两个比较对象都是ColorPoint呢？这样一来如果我的两个ColorPoint的x和y全都一样，只是color不同，那么无论怎么比较，其结果值都会是true。这里不会去检查color。那你可能就会说，那我们就重写ColorPoint的equals啊。但是在这种情况下，我们推荐复合优于继承。

4.一致性：对于任何非null的引用值x,y，只要equals的比较操作在对象中所用的信息没有被修改，多次调用x.equals(y)就会一致地返回true，或一致地返回false。

5.对于非null的引用值x，x.equals(null)必须返回false。

第9条 覆盖equals时总要覆盖hashCode

如果我们在覆盖equals方法时，没有覆盖hashCode方法，将会违反Object.hashCode的通用约定，从而会导致该类无法结合所有基于散列的集合一起正常运行。

没有覆盖hashCode会为违反如下准则：相等的对象必须具有相等的散列码。示例：

```

public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    public PhoneNumber(int areaCode, int prefix, int lineNumber) {
        rangeCheck(areaCode, 999, "area code");
        rangeCheck(prefix, 999, "prefix");
        rangeCheck(lineNumber, 9999, "line number");
        this.areaCode = (short) areaCode;
        this.prefix = (short) prefix;
        this.lineNumber = (short) lineNumber;
    }

    private static void rangeCheck(int arg, int max, String name) {
        if (arg < 0 || arg > max)
            throw new IllegalArgumentException(name + ": " + arg);
    }

    @Override
    public boolean equals(Object o) {
        if (o == this)

```

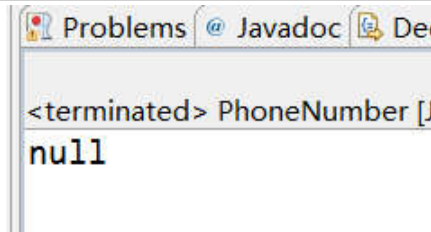


```

        return true;
    if (!(o instanceof PhoneNumber))
        return false;
    PhoneNumber pn = (PhoneNumber)o;
    return pn.lineNumber == lineNumber
        && pn.prefix == prefix
        && pn.areaCode == areaCode;
    }

    public static void main(String[] args) {
        Map<PhoneNumber, String> m = new HashMap<PhoneNumber, String>();
        m.put(new PhoneNumber(707, 867, 5309), "Jenny");
        System.out.println(m.get(new PhoneNumber(707, 867, 5309)));
    }
}

```



这里涉及到了两个PhoneNumber实例，第一个被用于插入到HashMap中，第二个实例和第一个实例相等，被用于在hashmap中获取值。但是由于没有覆盖hashCode方法，所以两次生成的对象的hash码并不相同。

我们可以这样来理解上面的map.put()。如果我们不去覆盖hashCode，那么当我们使用map.put时，我们是把这些PhoneNumber对象放在一个hash bucket中，而我们去map.get()的时候，却去另外一个hash bucket中寻找（当然，如果map.get()和map.put()中的对象是同一个的话，当然可以找到）。

而如果我们覆盖了hashCode方法，这时，如果通过hashCode计算出来的值是相等的，就会放在hash bucket里。这样，只要我们对对象中保存的值是完全一致的，就会找到这个key所对应的value。不知道你发现没有，这个hashCode有点类似于分类，这样在数据量比较大的情况下就会大大提高效率。

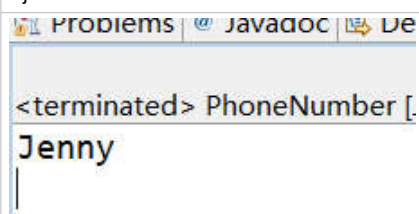
示例：

加入如下方法：

```

@Override
public int hashCode() {
    int result = 17;
    result = 31 * result + areaCode;
    result = 31 * result + prefix;
    result = 31 * result + lineNumber;
    return result;
}

```



第10条 始终要覆盖toString方法

就拿PhoneNumber类来说，如果我们不去覆盖类的toString()方法，后果就是当我们需要去打印这个类的对象时，会有一些并非是我们想要的那种。有时我们不希望打印出这样的对象，那我们就要去覆盖它们的toString方法了。在这个方法里，我们可以按照我们自己的意愿来给类添加toString方法。

如下所示：

```

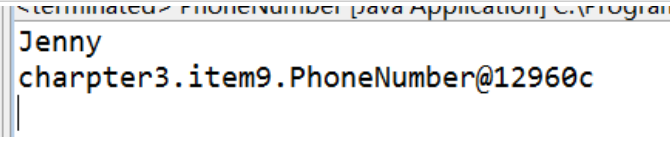
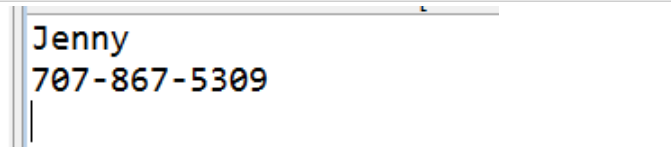
public static void main(String[] args) {
    Map<PhoneNumber, String> m = new
    HashMap<PhoneNumber, String>();
    m.put(new PhoneNumber(707, 867, 5309), "Jenny");
}

```

```

@Override
public String toString() {
    String result = "";
    result += (areaCode + "-");
}

```


<pre> System.out.println(m.get(new PhoneNumber(707, 867, 5309))); System.out.println(new PhoneNumber(707, 867, 5309).toString()); } </pre>	<pre> result += (prefix + "-"); result += (lineNumber); return result; } public static void main(String[] args) { Map<PhoneNumber, String> m = new HashMap<PhoneNumber, String>(); m.put(new PhoneNumber(707, 867, 5309), "Jenny"); System.out.println(m.get(new PhoneNumber(707, 867, 5309))); System.out.println(new PhoneNumber(707, 867, 5309).toString()); } </pre>
	

三、类和接口

第13条：使类和成员的可访问性最小化

第14条：在公有类中使用访问方法而非公有域

第15条：使可变性最小化

第16条：复合优先于继承

第18条：接口优于抽象类

第19条：接口只用于定义类型

第22条：有效考虑静态成员类

四、泛型

第24条：消除非受检警告

用泛型编程时，会遇到许多编译器警告：

1. 非受检强制转化警告（unchecked cast warnings）
2. 非受检方法调用警告
3. 非受检普通数组创建警告
4. 非受检转换警告（unchecked conversion warnings）

当你越来越熟悉泛型之后，遇到的警告也会越来越少，但是不要期待从一开始用泛型编写代码就可以正确地进行编译。

许多非受检警告很容易消除。有些警告比较难以消除。记住！**要尽可能地消除每一个非受检警告**。如果消除了所有警告，就可以确保代码是类型安全的。

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，（只有在这种情况下才可以用一个@SupperessWarnings("unchecked")注解来禁止这条警告）。SupperessWarnings注解可以用在任何粒度级别中，应该始终在**尽可能小的范围内使用SupperessWarnings注解**。永远不要在整个类上使用SupperessWarnings，这么做可能会掩盖重要的警告。

总而言之，非受检警告很重要，不要忽略它们。每一条警告都表示可能在运行时抛出 ClassCastException异常。要尽最大的努力消除这些警告。如果无法消除非受检警告，同时可以证明引起警告的代码是类型安全的，就可以在尽可能小的范围中，用@SupperessWarnings("unchecked")注解禁止该警告。并且要用注解**把禁止该警告的原因记录下来**。

第25条：列表优先于数组

数组是协变的(convariant)，如果Sub是Super的子类型，那么数组类型Sub[]就是Super[]的子类型。泛型确实不可变的，List<Sub>**不是**List<Super>的子类型。

数组是具体化的(reified)，因此数组在运行时才知道并检查它们的元素类型约束。

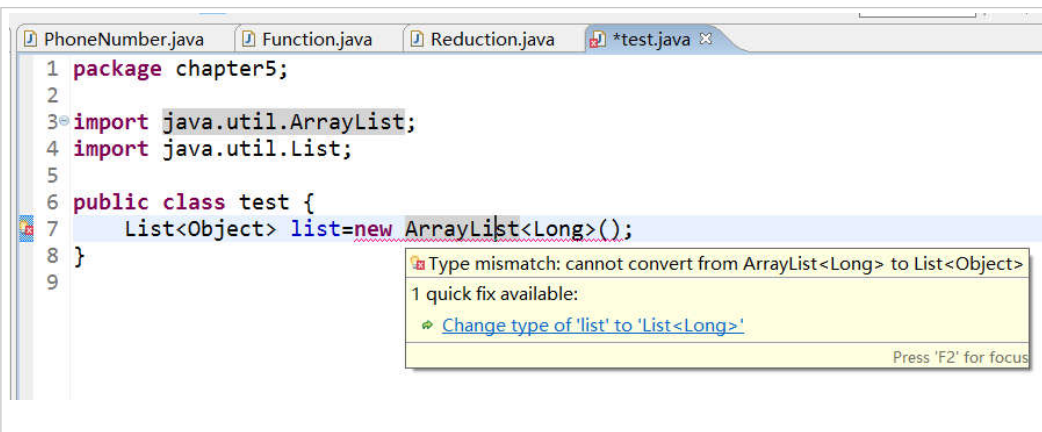
```
public class test {
```

```
public static void main(String[] args) {
    Object[] objectArray=new Long[1];
    objectArray[0]="I do not fit!";
}
}
```

```
<terminated> test [Java Application] C:\Program Files\Java\jre1.8.0_65\bin\javaw.exe (2016年11月27日 下午8:44:54)
Exception in thread "main" java.lang.ArrayStoreException: java.lang.String
    at chapter5.test.main(test.java:6)
```

编译时不会报错，运行时才会报错。

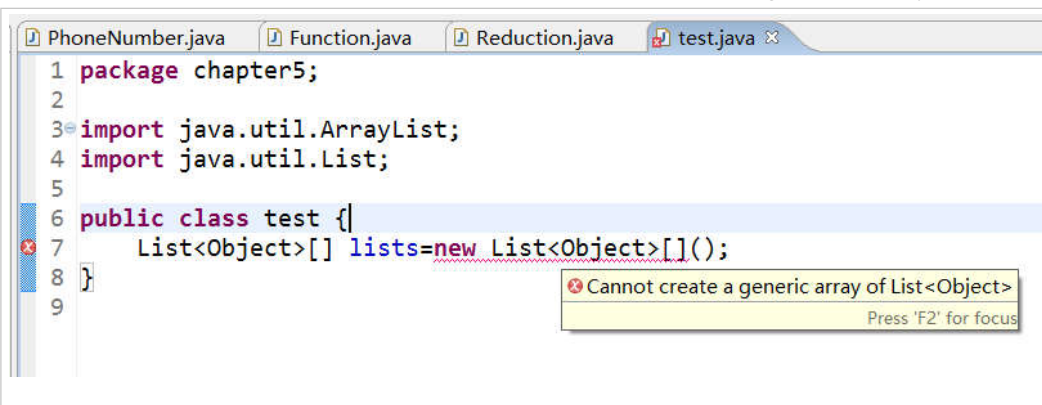
泛型则是通过擦除(erasure)来实现，因此泛型只在编译时强化它们的类型信息，并在运行时丢弃（或者擦除）它们的元素类型约束。擦除就是使泛型可以与没有使用泛型的代码随意进行互用(见第23条)。



```
1 package chapter5;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class test {
7     List<Object> list=new ArrayList<Long>();
8 }
9
```

Type mismatch: cannot convert from ArrayList<Long> to List<Object>
1 quick fix available:
Change type of 'list' to 'List<Long>'

我们不能创建泛型数组，因为在编译时会导致在编译时都会导致一个generic array creation(泛型数组创建)错误。



```
1 package chapter5;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class test {
7     List<Object>[] lists=new List<Object>[]();
8 }
9
```

Cannot create a generic array of List<Object>

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。

数组和泛型有两个重要的不同点。从技术的角度来说，像E、List和List这样的类型应称作不可具体化的(nonreifiable)类型(不可具体化类型是指其运行时表示法包含的信息比它编译时表示法包含的信息更少的类型)。唯一可具体化(reifiable)的参数化类型是无限制的通配符类型，如List<?>, Map<?, ?>。虽然不常用，但是创建无限制通配类型的数组是合法的。如：List<?>[] stringLists=new List<?>[1];//是合法的

一般来说，数组和泛型不能很好地混合使用。如果你发现自己将它们混合起来使用，并且得到了编译时错误或者警告，你的第一反应就应该就是用列表代替数组。

五、枚举和注解

第36条：坚持使用Override注解

@Override注解只能用在方法申明中，它表示被注解的方法申明覆盖了超类中的一个申明。如果使用这个注解，可以防止一大类的非法错误。

如下示例，你只想向集合中加入26个双字母组合：

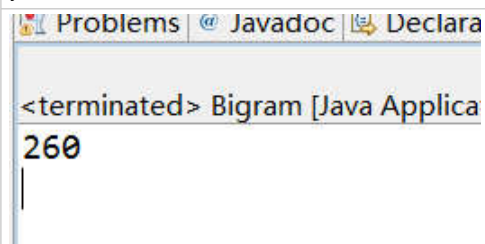
```
public class Bigram {
    private final char first;
    private final char second;
```

```

public Bigram(char first, char second) {
    this.first = first;
    this.second = second;
}
public boolean equals(Bigram b) {
    return b.first == first && b.second == second;
}
public int hashCode() {
    return 31 * first + second;
}

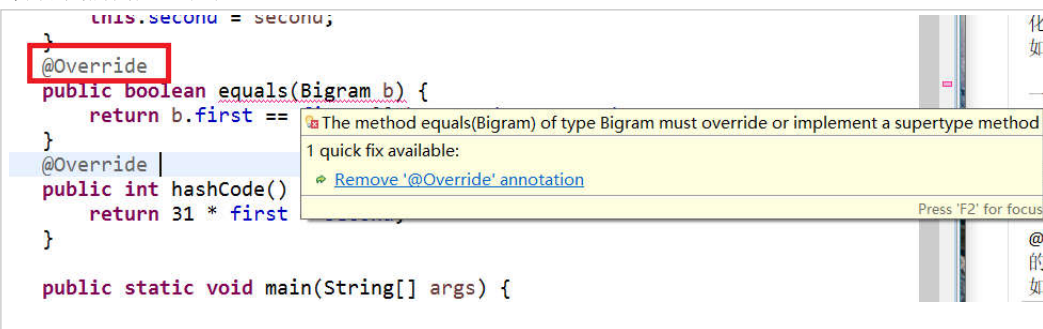
public static void main(String[] args) {
    Set<Bigram> s = new HashSet<Bigram>();
    for (int i = 0; i < 10; i++)
        for (char ch = 'a'; ch <= 'z'; ch++)
            s.add(new Bigram(ch, ch));
    System.out.println(s.size());
}
}

```



但结果却出现了260个，说明equals和hashCode方法并没有被覆盖。

为了表明Bigram类从Object继承了并覆盖了equals和hashCode方法，我们在方法上面加上@Override注解，加上该注解编译器会告诉我们问题出在哪里。



此时，我们修改上面的代码，如下所示：

```

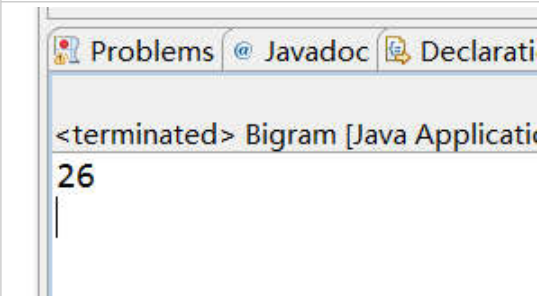
public class Bigram {
    private final char first;
    private final char second;
    public Bigram(char first, char second) {
        this.first = first;
        this.second = second;
    }
    // @Override
    // public boolean equals(Bigram b) {
    //     return b.first == first && b.second == second;
    // }
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Bigram))
            return false;
        Bigram b = (Bigram) obj;
        return b.first == first && b.second == second;
    }
    @Override
    public int hashCode() {
        return 31 * first + second;
    }
}

```

```

public static void main(String[] args) {
    Set<Bigram> s = new HashSet<Bigram>();
    for (int i = 0; i < 10; i++)
        for (char ch = 'a'; ch <= 'z'; ch++)
            s.add(new Bigram(ch, ch));
    System.out.println(s.size());
}
}

```



打印出了正确的结果。

六、方法

第41条：慎用重载

先看如下示例代码1：

```

public class CollectionClassifier {
    public static String classify(Set<?> s) {
        return "Set";
    }

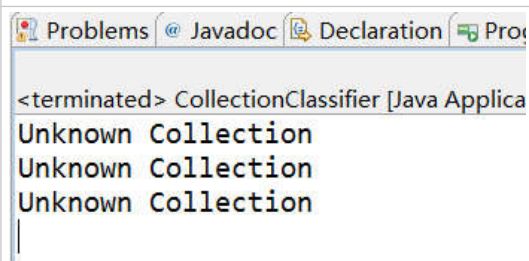
    public static String classify(List<?> lst) {
        return "List";
    }

    public static String classify(Collection<?> c) {
        return "Unknown Collection";
    }

    public static void main(String[] args) {
        Collection<?>[] collections = {
            new HashSet<String>(),
            new ArrayList<BigInteger>(),
            new HashMap<String, String>().values()
        };

        for (Collection<?> c : collections)
            System.out.println(classify(c));
    }
}

```



对于上面的for循环，参数的编译时类型都是相同的`Collection<?>`，但是每次迭代的类型不同，程序期望打印出三个不同的结果，但是三次循环都只调用了`String classify(Collection<?> c)`方法。重载方法的选择是静态的，**因为要调用哪一个重载（overload）方法是在编译时就做出的决定。**

再看下面的覆盖（Override）的示例2：

```

class Wine {
    String name() { return "wine"; }
}

```

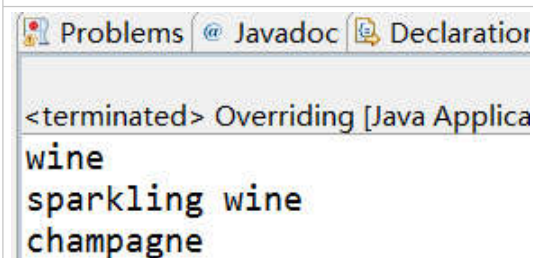
```

class SparklingWine extends Wine {
    @Override
    String name() { return "sparkling wine"; }
}

class Champagne extends SparklingWine {
    @Override
    String name() { return "champagne"; }
}

public class Overriding {
    public static void main(String[] args) {
        Wine[] wines = {
            new Wine(), new SparklingWine(), new Champagne()
        };
        for (Wine wine : wines)
            System.out.println(wine.name());
    }
}

```



```

<terminated> Overriding [Java Applica
wine
sparkling wine
champagne

```

被覆盖（override）的方法的选择时动态的，是在运行时确定的，选择的依据是被调用的方法所在对象的运行时类型。

为了避免示例1中出现的问题，我们应该避免胡乱使用重载：

1. 永远不要导出两个具有相同参数数目的重载方法。
2. 如果方法使用了可变参数，就不要重载他们。

注意下面的示例：

```

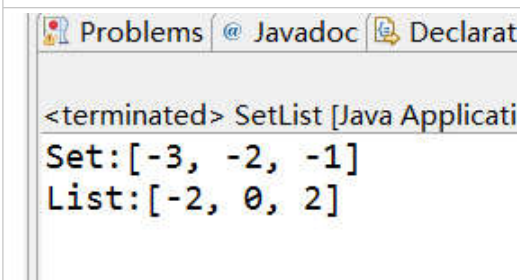
public class SetList {
    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<Integer>();
        List<Integer> list = new ArrayList<Integer>();

        for (int i = -3; i < 3; i++) {
            set.add(i);
            list.add(i);
        }

        for (int i = 0; i < 3; i++) {
            set.remove(i);
            list.remove(i);
        }

        System.out.println("Set:" + set);
        System.out.println("List:" + list);
    }
}

```



```

<terminated> SetList [Java Applicati
Set: [-3, -2, -1]
List: [-2, 0, 2]

```

```

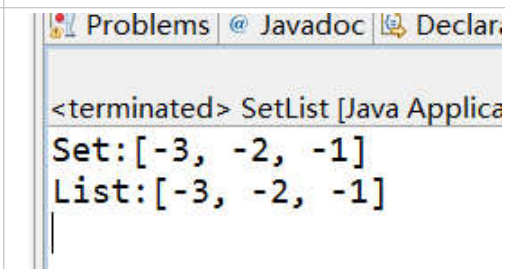
public class SetList {
    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<Integer>();
        List<Integer> list = new ArrayList<Integer>();

        for (int i = -3; i < 3; i++) {
            set.add(i);
            list.add(i);
        }

        for (int i = 0; i < 3; i++) {
            set.remove(i);
            list.remove((Integer)i);
        }

        System.out.println("Set:" + set);
        System.out.println("List:" + list);
    }
}

```



```

<terminated> SetList [Java Applica
Set: [-3, -2, -1]
List: [-3, -2, -1]

```


我们直接看两个方法的JDK源码：

<div><div>Set.class</div><div><div>Set<E></div><div><div>add(E) : boolean</div><div>addAll(Collection<? extends E>) : boolean</div><div>clear() : void</div><div>contains(Object) : boolean</div><div>containsAll(Collection<?>) : boolean</div><div>equals(Object) : boolean</div><div>hashCode() : int</div><div>isEmpty() : boolean</div><div>iterator() : Iterator<E></div><div>remove(Object) : boolean</div><div>removeAll(Collection<?>) : boolean</div><div>retainAll(Collection<?>) : boolean</div><div>size() : int</div><div>spliterator() : Spliterator<E></div><div>toArray() : Object[]</div><div>toArray(T[]) <T> : T[]</div></div></div></div>	<div><div>List.class</div><div><div>List<E></div><div><div>add(E) : boolean</div><div>add(int, E) : void</div><div>addAll(int, Collection<? extends E>) : boolean</div><div>addAll(Collection<? extends E>) : boolean</div><div>clear() : void</div><div>contains(Object) : boolean</div><div>containsAll(Collection<?>) : boolean</div><div>equals(Object) : boolean</div><div>get(int) : E</div><div>hashCode() : int</div><div>indexOf(Object) : int</div><div>isEmpty() : boolean</div><div>iterator() : Iterator<E></div><div>lastIndexOf(Object) : int</div><div>listIterator() : ListIterator<E></div><div>listIterator(int) : ListIterator<E></div><div>remove(int) : E</div><div>remove(Object) : boolean</div><div>removeAll(Collection<?>) : boolean</div><div>replaceAll(UnaryOperator<E>) : void</div><div>retainAll(Collection<?>) : boolean</div></div></div></div>
<pre>public boolean remove(Object o) { return m.remove(o)==PRESENT; }</pre>	<pre>public E remove(int index) { rangeCheck(index); modCount++; E oldValue = elementData(index); int numMoved = size - index - 1; if (numMoved > 0) System.arraycopy(elementData, index+1, elementData, index, numMoved); elementData[--size] = null; // clear to let GC do its work return oldValue; } public boolean remove(Object o) { if (o == null) { for (int index = 0; index < size; index++) if (elementData[index] == null) { fastRemove(index); return true; } } else { for (int index = 0; index < size; index++) if (o.equals(elementData[index])) { fastRemove(index); return true; } } return false; }</pre>

第43条：返回零长度的数组或者集合，而不是null

当程序放回null时，我们需要对null值做很多额外的处理，因此返回类型为数组或者集合时，我们可以返回0长度的数组或集合。

七、通用程序设计

第45条：将局部变量的作用域最小化

第46条：for-each循环优先于for循环

```
public class test {
    public static void main(String[] args) {
        List<String> list=new ArrayList<>();
        list.add("Hello");
        list.add(",");
        list.add("Java");
        list.add("World");
        list.add("!");

        for(int i=0;i<list.size();i++){
            System.out.print(list.get(i)+" ");
        }
        System.out.println();
        for(String s:list){
            System.out.print(s+" ");
        }
    }
}
```

for-each循环和传统的for循环相比的优点：

1. 简洁性
2. 预防bug的优势
3. 无性能损失

因此能够用for-each的尽量用for-each，不要用for循环。collections和arrays都能使用for-each。只要实现了iterable接口的类都能使用for-each。

但是有些情况下不能使用for-each:

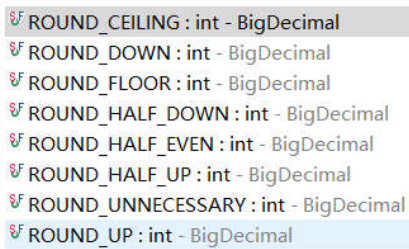
1. 过滤，要删除指定的元素时，不能用for-each
2. 转换，但需要遍历list或者array修改部分或者全部数值的时候，需要记录起始位置和结束为止的迭代器或索引变量
3. 并行迭代，当需要并行迭代不同的集合时，需要迭代器变量或者索引时候，不能用for-each

第48条：如果需要精确的答案，避免使用float和double

float和double类型主要是为了科学计算和工程计算设计的，它们执行二进制浮点运算，这是为了在广泛的数值范围上提供较为精确的快速近似计算位精心设计的。然而，它们并没有提供完全精确的结果，所以不应该被用于需要精确结果的场合。float和double类型尤其不适合用于货币计算，因为要让一个float或者double精确的表示0.1(或者10的任何其它次方值)是不可能的。

在涉及到货币计算时，最好是使用BigDecimal、int以及long（整型需要自己处理十进制小数）。同时BigDecimal还提供自定义四舍五入：

```
double num1=test.setScale(2, BigDecimal.ROUND_HALF_UP).doubleValue();
System.out.println(num1);
```



ROUND_CEILING : int - BigDecimal
ROUND_DOWN : int - BigDecimal
ROUND_FLOOR : int - BigDecimal
ROUND_HALF_DOWN : int - BigDecimal
ROUND_HALF_EVEN : int - BigDecimal
ROUND_HALF_UP : int - BigDecimal
ROUND_UNNECESSARY : int - BigDecimal
ROUND_UP : int - BigDecimal

```
public class test {
    private static MathContext mc;

    public static void main(String[] args) {
        double num=3.1415926;
```



```

        BigDecimal test=new BigDecimal(num);

        double num1=test.setScale(2, BigDecimal.ROUND_HALF_UP).doubleValue();
        System.out.println(num1);//3.14
    }
}

```

示例:

```

public class Arithmetic {
    // Broken - uses floating point for monetary calculation!
    public static void main(String[] args) {
        System.out.println(1.03 - .42);
        System.out.println();

        System.out.println(1.00 - 9 * .10);
        System.out.println();

        howManyCandies1();
        System.out.println();

        howManyCandies2();
        System.out.println();

        howManyCandies3();
    }

    public static void howManyCandies1() {
        double funds = 1.00;
        int itemsBought = 0;
        for (double price = .10; funds >= price; price += .10) {
            funds -= price;
            itemsBought++;
        }
        System.out.println(itemsBought + " items bought.");
        System.out.println("Change: $" + funds);
    }

    public static void howManyCandies2() {
        final BigDecimal TEN_CENTS = new BigDecimal( ".10");

        int itemsBought = 0;
        BigDecimal funds = new BigDecimal("1.00");
        for (BigDecimal price = TEN_CENTS;
            funds.compareTo(price) >= 0;
            price = price.add(TEN_CENTS)) {
            itemsBought++;
            funds = funds.subtract(price);
        }
        System.out.println(itemsBought + " items bought.");
        System.out.println("Money left over: $" + funds);
    }

    public static void howManyCandies3() {
        int itemsBought = 0;
        int funds = 100;
        for (int price = 10; funds >= price; price += 10) {
            itemsBought++;
            funds -= price;
        }
        System.out.println(itemsBought + " items bought.");
        System.out.println("Money left over: " + funds + " cents");
    }
}

```

```

<terminated> Arithmetic [Java Application] C:\Progr
0.6100000000000001
0.09999999999999998
3 items bought.
Change: $0.3999999999999999
4 items bought.
Money left over: $0.00
4 items bought.
Money left over: 0 cents

```

八、并发

第66条：同步访问共享的可变数据

第67条：避免过度同步