Effective java notes

# Chapter 2. Creating and Destroying Objects

## ITEM 1: CONSIDER STATIC FACTORY METHODS INSTEAD OF CONSTRUCTORS

**One advantage** of static factory methods is that, unlike constructors, they have names. For example, the constructor BigInteger(int, int, Random), which returns a BigInteger that is probably prime, would have been better expressed as a static factory method named BigInteger.probablePrime.

**A second advantage** of static factory methods is that, unlike constructors, they are not required to create a new object each time they’re invoked. 静态方法的一个好处

**A third advantage** of static factory methods is that, unlike constructors, they can return an object of any subtype of their return type

**A fourth advantage** of static factories is that the class of the returned object can vary from call to call as a function of the input parameters

其实三四点可以归纳到一起，第三点是可以返回当前类的子类，第四点是根据输入的，创建不同的子类

**A fifth advantage** of static factories is that the class of the returned object need not exist when the class containing the method is written.

The **main** **limitation** of providing only static factory methods is that classes without public or protected constructors cannot be subclassed.

A **second shortcoming** of static factory methods is that they are hard for programmers to find.

## ITEM 2: CONSIDER A BUILDER WHEN FACED WITH MANY CONSTRUCTOR PARAMETERS

Disadvantage: the telescoping constructor pattern works, but it is hard to write client code when there are many parameters, and harder still to read it.

当一个类的参数过于多时，有些参数是optional的，难道需要针对每种情况都写一个constructer吗

Solution：

1.setter

public class NutritionFacts {

// Parameters initialized to default values (if any)

private int servingSize = -1; // Required; no default value

private int servings = -1; // Required; no default value

private int calories = 0;

private int fat = 0;

private int sodium = 0;

private int carbohydrate = 0;

public NutritionFacts() { }

// Setters

public void setServingSize(int val) { servingSize = val; }

public void setServings(int val) { servings = val; }

public void setCalories(int val) { calories = val; }

public void setFat(int val) { fat = val; }

public void setSodium(int val) { sodium = val; }

public void setCarbohydrate(int val) { carbohydrate = val; }

}

Unfortunately, the JavaBeans pattern has serious disadvantages of its own. Because construction is split across multiple calls, a JavaBean may be in an inconsistent state partway through its construction. The class does not have the option of enforcing consistency merely by checking the validity of the constructor parameters. Attempting to use an object when it’s in an inconsistent state may cause failures that are far removed from the code containing the bug and hence difficult to debug. A related disadvantage is that the JavaBeans pattern precludes the possibility of making a class immutable (Item 17) and requires added effort on the part of the programmer to ensure thread safety. It is possible to reduce these disadvantages by manually “freezing” the object when its construction is complete and not allowing it to be used until frozen, but this variant is unwieldy and rarely used in practice. Moreover, it can cause errors at runtime because the compiler cannot ensure that the programmer calls the freeze method on an object before using it.

总的来说就是线程不安全

2. Builder

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 sodium = 0;

private int carbohydrate = 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 sodium(int val)

{ sodium = val; return this; }

public Builder carbohydrate(int val)

{ carbohydrate = 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;

}

}

**The Builder pattern simulates named optional parameters.**

**The Builder pattern is well suited to class hierarchies**

**// Builder pattern for class hierarchies**

public abstract class Pizza {

public enum Topping { HAM, MUSHROOM, ONION, PEPPER,

SAUSAGE }

final Set<Topping> toppings;

abstract static class **Builder<T extends Builder<T>>** {

EnumSet<Topping> toppings =

EnumSet.noneOf(Topping.class);

public T addTopping(Topping topping) {

toppings.add(Objects.requireNonNull(topping));

**return self();**

}

abstract Pizza build();

**// Subclasses must override this method to return**

**"this"**

**protected abstract T self();**

}

Pizza(Builder<?> builder) {

toppings = builder.toppings.clone(); // See Item 50

}

}

public class NyPizza extends Pizza {

public enum Size { SMALL, MEDIUM, LARGE }

private final Size size;

public static class Builder extends Pizza.Builder<Builder> {

private final Size size;

public Builder(Size size) {

this.size = Objects.requireNonNull(size);

}

@Override public NyPizza build() {

return new NyPizza(this);

}

@Override protected Builder self() { return this; }

}

private NyPizza(Builder builder) {

super(builder);

size = builder.size;

}

}

public class Calzone extends Pizza {

private final boolean sauceInside;

public static class Builder extends Pizza.Builder<Builder> {

private boolean sauceInside = false; // Default

public Builder sauceInside() {

sauceInside = true;

return this;

}

@Override public Calzone build() {

return new Calzone(this);

}

@Override protected Builder self() { return this; }

}

private Calzone(Builder builder) {

super(builder);

sauceInside = builder.sauceInside;

}

}

NyPizza pizza = new NyPizza.Builder(SMALL).addTopping(SAUSAGE).addTopping(ONION).build();

Calzone calzone = new Calzone.Builder() .addTopping(HAM).sauceInside().build();

The Builder pattern has disadvantages as well. In order to create an object, you must first create its builder. While the cost of creating this builder is unlikely to be noticeable in practice, it could be a problem in performance-critical situations.

## ITEM 3: ENFORCE THE SINGLETON PROPERTY WITH A PRIVATE CONSTRUCTOR OR AN ENUM TYPE

## ITEM 4: ENFORCE NONINSTANTIABILITY WITH A PRIVATE CONSTRUCTOR

包含多个静态方法的工具类constructor要private

## ITEM 5: PREFER DEPENDENCY INJECTION TO HARDWIRING RESOURCES

A simple pattern that satisfies this requirement is to pass the resource into the constructor when creating a new instance.

最简单的依赖注入

public class SpellChecker {

private static final Lexicon dictionary = ...;

private SpellChecker() {} // Noninstantiable

public static boolean isValid(String word) { ... }

public static List<String> suggestions(String typo) { ... }

}

public class SpellChecker {

private final Lexicon dictionary;

**public SpellChecker(Lexicon dictionary)** {

this.dictionary = Objects.requireNonNull(dictionary);

}

public boolean isValid(String word) { ... }

public List<String> suggestions(String typo) { ... }

}

## ITEM 6: AVOID CREATING UNNECESSARY OBJECTS

避免反复创建新的对象

String s = new String("bikini"); **// DON'T DO THIS!**

String s = "bikini";

**// Hideously slow! Can you spot the object creation?**

private static long sum() {

Long sum = 0L;

for (long i = 0; i <= Integer.MAX\_VALUE; i++)

sum += i;

return sum;

}

prefer primitives to boxed primitives, and watch out for unintentional autoboxing.

## ITEM 7: ELIMINATE OBSOLETE OBJECT REFERENCES

private Object[] elements;

public Object pop() {

if (size == 0)

throw new EmptyStackException();

return elements[--size];

}

public Object pop() {

if (size == 0)

throw new EmptyStackException();

Object result = elements[--size];

**elements[size] = null; // Eliminate obsolete reference**

return result;

}

## ITEM 8: AVOID FINALIZERS AND CLEANERS PREFER TRY-WITH-RESOURCES TO TRY-FINALLY

# Chapter 3. Methods Common to All Objects

## ITEM10 OBEY THE GENERAL CONTRACT WHEN OVERRIDING EQUALS

* Each instance of the class is inherently unique
* There is no need for the class to provide a “logical equality” test.
* A superclass has already overridden equals and the superclass behavior is appropriate for this class
* The class is private or package-private and you are certain that the equals method will never be invoked

When you override the equals method, you must adhere to its general contract. Here is the contract, from the specification for Object:

**Reflexivity**—The first requirement says merely that an object must be equal to itself.

**Symmetry**—The second requirement says that any two objects must agree on whether they are equal.

一个不满足的例子：

**public static void** main(String[] args) {  
 CaseInsensitiveString cis = **new** CaseInsensitiveString(**"Polish"**);  
 String s = **"polish"**;  
 System.***out***.println(cis.equals(s) + **" "** + s.equals(cis));  
}  
  
**private static class** CaseInsensitiveString {  
 **private final** String **s**;  
  
 **public** CaseInsensitiveString(String s) {  
 **this**.**s** = s;  
 }  
  
 @Override  
 **public boolean** equals(Object obj) {  
 **if** (obj **instanceof** CaseInsensitiveString) {  
 **return s**.equalsIgnoreCase(((CaseInsensitiveString) obj).**s**);  
 }  
  
 **if** (obj **instanceof** String) {  
 **return s**.equalsIgnoreCase((String) obj);  
 }  
 **return false**;  
 }  
}

**更有可能导致下面后果：**

List<CaseInsensitiveString> list = new ArrayList<>(); list.add(cis);

What does list.contains(s) return at this point? Who knows?

返回的结果和JDK的实现有关

**如何消除这种影响？**

**return** obj **instanceof** CaseInsensitiveString &&  
 ((CaseInsensitiveString) obj).**s**.equalsIgnoreCase(**s**);

**Transitivity**—The third requirement of the equals contract says that if one object is equal to a second and the second object is equal to a third, then the first object must be equal to the third. Again, it’s not hard to imagine violating this requirement unintentionally. Consider the case of a subclass that adds a new *value component* to its superclass. In other words, the subclass adds a piece of information that affects equals comparisons. Let’s start with a simple immutable two-dimensional integer point class:

**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**;  
 Point p = (Point) o;  
 **return** p.**x** == **x** && p.**y** == **y**;  
 }  
}  
  
**public class** ColorPoint **extends** Point {  
 **private final** Color **color**;  
  
 **public** ColorPoint(**int** x, **int** y, Color color) {  
 **super**(x, y);  
 **this**.**color** = color;  
 }  
}

思考一下，这里的ColorPoint的equals方法要怎么写(类不同则一定不同)？

写法1：

*// Broken - violates symmetry!*@Override  
**public boolean** equals(Object o) {  
 **if**(!(o **instanceof** ColorPoint)){  
 **return false**;  
 }  
 **return super**.equals(o) && ((ColorPoint) o).**color** == **color**;  
}

The problem with this method is that you might get different results when comparing a point to a color point and vice versa. The former comparison ignores color, while the latter comparison always returns false because the type of the argument is incorrect. To make this concrete, let’s create one point and one color point:

其实主要是ColorPoint方法里写法有误，怎么修改？

*// Broken - violates transitivity!*@Override   
**public boolean** equals(Object o) {   
 **if** (!(o **instanceof** Point)) {  
 **return false**;  
 }  
 *// If o is a normal Point, do a color-blind comparison* **if** (!(o **instanceof** ColorPoint)){  
 **return** o.equals(**this**);  
 }   
 *// o is a ColorPoint; do a full comparison* **return super**.equals(o) && ((ColorPoint) o).**color** == **color**;   
}

其实这里的写法依然是错误的，违法的传递性，看下面的代码：

ColorPoint **p1** = **new** ColorPoint(1, 2, Color.***RED***);  
Point **p2** = **new** Point(1, 2);  
ColorPoint **p3** = **new** ColorPoint(1, 2, Color.***BLUE***);

**There is no way to extend an instantiable class and add a value component while preserving  
the equals contract**, unless you’re willing to forgo the benefits of object-oriented abstraction.

那么到底有没有方法可以解决这个问题：

You may hear it said that you can extend an instantiable class and add a value component while preserving the equals contract by using a getClass test in place of the instanceof test in  
the equals method:

参见[java中instanceof和getClass()的作用](https://www.cnblogs.com/aoguren/p/4822380.html)

修改Color里的equals方法：

*// Broken - violates Liskov substitution principle (page 43)*@Override   
**public boolean** equals(Object o) {  
 **if** (o == **null** || o.getClass() != getClass()) **return false**;  
 Point p = (Point) o;  
 **return** p.**x** == **x** && p.**y** == **y**;   
}

为什么这里违反了LSP原则？

LSP要求子类能够完全替换父类，但是ColorPoint却不再是Point类了，比如一个HashSet里包含多个Point，HashSet的contains方法用的是equals来判断，所以ColorPoint永远不会等于一个Point。

还有种方法，就是放弃继承：

public class ColorPoint {

private final Point point;

private final Color color;

}

There are some classes in the Java platform libraries that do extend an instantiable class and add a value component. For example, java.sql.Timestamp extends java.util.Date and adds  
a nanoseconds field. The equals implementation for Timestamp does violate symmetry and can cause erratic behavior

if Timestamp and Date objects are used in the same collection or are otherwise intermixed.

Note that you *can* add a value component to a subclass of  
an *abstract* class without violating the equals contract.

**Consistency**—The fourth requirement of the equals contract says that if two objects are equal, they must remain equal for all time unless one (or both) of them is modified.

**Non-nullity—**It says that all objects must be unequal to null.

知道了以上这些，那么到底该如何写equals方法：

1. **Use the == operator to check if the argument is a reference to this object.**

2. **Use the instanceof operator to check if the argument has the correct type.**

Instanceof 会检查null

3. **Cast the argument to the correct type.**

4. **For each “significant” field in the class, check if that field of the argument matches the corresponding field of this object.**

示例：

HashMap中Node的equals方法：

**public final boolean** equals(Object o) {  
 **if** (o == **this**)  
 **return true**;  
 **if** (o **instanceof** Map.Entry) {  
 Map.Entry<?,?> e = (Map.Entry<?,?>)o;  
 **if** (Objects.*equals*(**key**, e.getKey()) &&  
 Objects.*equals*(**value**, e.getValue()))  
 **return true**;  
 }  
 **return false**;  
}

## ITEM 11: ALWAYS OVERRIDE HASHCODE WHEN YOU OVERRIDE EQUALS

**You must override hashCode in every class that overrides equals.**

**Contract:**

• When the hashCode method is invoked on an object repeatedly during an execution of an application, it must consistently return the same value

• If two objects are equal according to the equals(Object) method, then calling hashCode on the two objects must produce the same integer result.

• If two objects are unequal according to the equals(Object) method, it is *not* required that calling hashCode on each of the objects must produce distinct results.

A good hash function tends to produce unequal hash codes for unequal instances.

如何写hashcode：

Equals方法里没有比较的field就不需要写了

其实就是Array.hashcode里的方法，我们可以直接使用Objects.hash()，很方便

**int** prime = 31;  
**int** result = 1;  
result = prime \* result + ((firstName == **null**) ? 0 : firstName.hashCode());  
result = prime \* result + ((lastName == **null**) ? 0 : lastName.hashCode());  
**return** result;

## ITEM 12: ALWAYS OVERRIDE TOSTRING

## ITEM 13: OVERRIDE CLONE JUDICIOUSLY

**in practice, a class implementing Cloneable is expected to provide a properly functioning public clone method.**

## ITEM 14: CONSIDER IMPLEMENTING COMPARABLE

public interface Comparable<T> { int compareTo(T t);

}

The general contract of the compareTo method is similar to that  
of equals:  
Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

**Use of the relational  
operators < and > in compareTo methods is verbose and error-prone and no longer recommended.**

不推荐直接使用< >来进行比较

正确的比较方式：

**public int** compareTo(PhoneNumber pn) {  
 **int** result = Short.*compare*(areaCode, pn.areaCode); **if** (result == 0) {  
 result = Short.*compare*(prefix, pn.prefix);   
 **if** (result == 0)   
 result = Short.*compare*(lineNum, pn.lineNum);   
 }  
 **return** result;   
}

另一种方法

通过构造函数的形式

private static final Comparator<PhoneNumber> COMPARATOR = comparingInt((PhoneNumber pn) -> pn.areaCode)

.thenComparingInt(pn -> pn.prefix)

.thenComparingInt(pn -> pn.lineNum);

public int compareTo(PhoneNumber pn) { return COMPARATOR.compare(this, pn);

}

错误的一种形式：

*// BROKEN difference-based comparator - violates transitivity!*

**static** Comparator<Object> hashCodeOrder = **new** Comparator<>() {  
 **public int** compare(Object o1, Object o2) {   
 **return** o1.hashCode() - o2.hashCode();  
 }   
};

正确的形式：

*// Comparator based on static compare method***static** Comparator<Object> hashCodeOrder = **new** Comparator<>() {  
 **public int** compare(Object o1, Object o2) {  
 **return** Integer.*compare*(o1.hashCode(), o2.hashCode());  
 }  
};  
  
**static** Comparator<Object> hashCodeOrder =  
 Comparator.*comparingInt*(o -> o.hashCode());

In summary, whenever you implement a value class that has a sensible ordering, you should have the class implement  
the Comparable interface so that its instances can be easily sorted, searched, and used in comparison-based collections. When comparing field values in the implementations of

the compareTo methods, avoid the use of the < and > operators. Instead, use the static compare methods in the boxed primitive classes or the comparator construction methods in  
the Comparator interface.

# Chapter 4. Classes and Interfaces

## ITEM 15: MINIMIZE THE

## ACCESSIBILITY OF CLASSES AND

## MEMBERS

好处：

**Information hiding** is important for many reasons, most of which stem from the fact that it *decouples*the components that comprise a system, allowing them to be developed, tested, optimized, used, understood, and modified in isolation. This speeds up system development because components can be developed in parallel. It eases the burden of maintenance because components can be understood more quickly and debugged or replaced with little fear of harming other components. While information hiding does not, in and of itself, cause good performance, it enables effective performance tuning: once a system is complete and profiling has determined which components are causing performance problems (Item 67), those components can be optimized without affecting the correctness of others. Information hiding increases software reuse because components that aren’t tightly coupled often prove useful in other contexts besides the ones for which they were developed. Finally, information hiding decreases the risk in building large systems because individual components may prove successful even if the system does not.

The rule of thumb is simple: **make each class or member as inaccessible as possible.** In other words, use the lowest possible access level consistent with the proper functioning of the software that you are writing.

4种access levels注意package-private 和protected的区别

• **private**—The member is accessible only from the top-level class where it is declared.

• **package-private**—The member is accessible from any class in the package where it is declared. Technically known  
as *default* access, this is the access level you get if no access modifier is specified (except for interface members, which are public by default).

• **protected**—The member is accessible from subclasses of the class where it is declared (subject to a few restrictions [JLS, 6.6.2]) and from any class in the package where it is declared.

• **public**—The member is accessible from anywhere.

**it is wrong for a class to have a public static final array field, or an accessor that returns such a field.**

**// Potential security hole!**

public static final Thing[] VALUES = { ... };

private static final Thing[] PRIVATE\_VALUES = { ... }; public static final List<Thing> VALUES =

Collections.unmodifiableList(Arrays.asList(PRIVATE\_VALUES)) ;

or

private static final Thing[] PRIVATE\_VALUES = { ... }; public static final Thing[] values() {

return PRIVATE\_VALUES.clone(); }

## ITEM 16: IN PUBLIC CLASSES, USE ACCESSOR METHODS, NOT PUBLIC

## FIELDS

## ITEM 17: MINIMIZE MUTABILITY

对于不会改变的类，可以直接声明为final：

Immutable classes are easier to design, implement, and use than mutable classes. They are less prone to error and are more secure. To make a class immutable, follow these five rules:

声明这样的类需要遵循下面几点：

1. **Don’t provide methods that modify the object’s state .**

2. **Ensure that the class can’t be extended.**

3. **Make all fields final.**

4. **Make all fields private.**

5. **Ensure exclusive access to any mutable components.**

**Immutable objects are inherently thread-safe; they require no synchronization.**

**immutable objects can be shared freely.**

**The major disadvantage of immutable classes is that they require a separate object for each distinct value.**

Now that you know how to make an immutable class and you understand the pros and cons of immutability, let’s discuss a few design alternatives. Recall that to guarantee immutability, a class must not permit itself to be subclassed. This can be done by making the class final, but there is another, more flexible alternative. Instead of making an immutable class final, you can make all of its constructors private or package-private and add public static factories in place of the public constructors (Item 1). To make this concrete, here’s how Complex would look if you took this approach:

*// Immutable class with static factories instead of constructors***public class** Complex {  
 **private final double re**;  
 **private final double im**;  
  
 **private** Complex(**double** re, **double** im) {  
 **this**.**re** = re;  
 **this**.**im** = im;  
 }  
  
 **public static** Complex valueOf(**double** re, **double** im) {  
 **return new** Complex(re, im);  
 }  
}

**Constructors should create fully initialized objects with all of their invariants established.** Don’t provide a public initialization method separate from the constructor or static factory unless there is a *compelling* reason to do so.

总之给变量和方法加限定访问符的时候要尽量保守

## ITEM 18: FAVOR COMPOSITION OVER INHERITANCE

**Unlike method invocation, inheritance violates encapsulation**

To make this concrete, let’s suppose we have a program that uses a HashSet. To tune the performance of our program, we need to query the HashSet as to how many elements have been added since it was created (not to be confused with its current size, which goes down when an element is removed). To provide this functionality, we write a HashSet variant that keeps count of the number of attempted element insertions and exports an accessor for this count. The HashSet class contains two methods capable of adding elements, add and addAll, so we override both of these methods:

*// 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**;  
 }  
}

AddAll会调用add方法，导致错误

如果是直接添加新方法也会导致一些错误

Luckily, there is a way to avoid all of the problems described above. Instead of extending an existing class, give your new class a private field that references an instance of the existing class. This design is called *composition* because the existing class becomes a component of the new one. Each instance method in the new class invokes the corresponding method on the contained instance of the existing class and returns the results. This is known as *forwarding*, and the methods in the new class are known as *forwarding methods*.

**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**;  
 }  
}

*// Reusable forwarding class  
// 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) {  
 **return s**.addAll(c);  
 }  
  
 **public boolean** removeAll(Collection<?> c) {  
 **return s**.removeAll(c);  
 }  
  
 **public boolean** retainAll(Collection<?> c) {  
 **return s**.retainAll(c);  
 }  
  
 **public** Object[] toArray() {  
 **return s**.toArray();  
 }  
  
 **public** <T> T[] toArray(T[] a) {  
 **return s**.toArray(a);  
 }  
  
 @Override  
 **public boolean** equals(Object o) {  
 **return s**.equals(o);  
 }  
  
 @Override  
 **public int** hashCode() {  
 **return s**.hashCode();  
 }  
  
 @Override  
 **public** String toString() {  
 **return s**.toString();  
 }  
}

Inheritance is appropriate only in circumstances where the subclass really is a *subtype* of the superclass. In other words, a class *B* should extend a class *A* only if an “is-a” relationship exists between the two classes. If you are tempted to have a  
class *B* extend a class *A*, ask yourself the question: Is every *B* really an *A*? If you cannot truthfully answer yes to this question, *B* should not extend *A*. If the answer is no, it is often the case that *B* should contain a private instance of *A* and expose a different API: *A* is not an essential part of *B*, merely a detail of its implementation.

There are a number of obvious violations of this principle in the Java platform libraries. For example, a stack is not a vector,  
so Stack should not extend Vector. Similarly, a property list is not a hash table, so Properties should not extend Hashtable. In both cases, composition would have been preferable.

## ITEM 19: DESIGN AND DOCUMENT FOR INHERITANCE OR ELSE

## PROHIBIT IT

## ITEM 20: PREFER INTERFACES TO ABSTRACT CLASSES

## ITEM 21: DESIGN INTERFACES FOR POSTERITY

Java8之后，在interface里添加方法可以用default关键字修饰了

## ITEM 22: USE INTERFACES ONLY TO DEFINE TYPES

interface不要用来定义常量

## ITEM 23: PREFER CLASS HIERARCHIES TO TAGGED CLASSES

## ITEM 24: FAVOR STATIC MEMBER CLASSES OVER NONSTATIC

There are four kinds of nested classes: *static member classes*, *nonstatic member classes*, *anonymous classes*, and *local*

*classes*. All but the first kind are known as *inner classes*.

**One common use of a static member class** is as a public helper class, useful only in conjunction with its outer class. For example, consider an enum describing the operations supported by a calculator (Item 34). The Operation enum should be a public static member class of the Calculator class. Clients of Calculator could then refer to operations using names

like Calculator.Operation.PLUS and Calculator.Operation.MINUS.

**One common use of a nonstatic member class is to define  
an *Adapter***[Gamma95] that allows an instance of the outer class to be viewed as an instance of some unrelated class. For example, implementations of the Map interface typically use nonstatic member classes to implement their *collection views*, which are returned by Map’s keySet, entrySet, and values methods. Similarly, implementations of the collection interfaces, such as Set and List, typically use nonstatic member classes to implement their iterators:

**If you declare a member class that does not require access to an enclosing instance, always put  
the static modifier in its declaration**

If you omit this modifier, each instance will have a hidden extraneous reference to its enclosing instance. As previously mentioned, storing this reference takes time and space.

Before lambdas were added to Java (Chapter 6), **anonymous classes** were the preferred means of creating small *function objects* and *process objects* on the fly, but lambdas are now preferred (Item 42). Another common use of anonymous classes is in the implementation of static factory methods

## ITEM 25: LIMIT SOURCE FILES TO A SINGLE TOP-LEVEL CLASS

一个文件定义一个类

## ITEM 26: DON’T USE RAW TYPES

private final **Collection** stamps = ... ;

这样定义是不合适的。

虽然可以通过编译，但是在runtime期间可能会报错，比如插入一个”coin”,如果声明了类型 的话，编译期就会报错

As noted earlier, it is legal to use raw types (generic types without their type parameters), but you should never do it. **If you use raw types, you lose all the safety and expressiveness benefits of generics.**

**那么List和List<Object>有区别吗**

While you shouldn’t use raw types such as List, it is fine to use types that are parameterized to allow insertion of arbitrary objects, such as List<Object>. Just what is the difference between the raw type List and the parameterized type List<Object>? Loosely speaking, the former has opted out of the generic type system, while the latter has explicitly told the compiler that it is capable of holding objects of any type. While you can pass a List<String> to a parameter of type List, you can’t pass it to a parameter of type List<Object>. There are sub-typing rules for generics,  
and List<String>is a subtype of the raw type List, but not of the parameterized type List<Object> (Item 28). As a consequence, **you lose type safety if you use a raw type such as List, but not if you use a parameterized type such as List<Object>.**

如果类型不确定的话请用通配符

You might be tempted to use a raw type for a collection whose element type is unknown and doesn’t matter. For example, suppose you want to write a method that takes two sets and returns the number of elements they have in common.

*// Use of raw type for unknown element type - don't do this!***static int** numElementsInCommon(Set s1, Set s2) {  
 **int** result = 0;  
 **for** (Object o1 : s1)  
 **if** (s2.contains(o1)) result++;  
 **return** result;   
}

*// Uses unbounded wildcard type - typesafe and flexible***static int** numElementsInCommon(Set<?> s1, Set<?> s2) { ... }

**you can’t put any element (other than null) into a Collection<?>.**

**You must use raw types in class literals.**

In other words, List.class, String[].class, and int.class are all legal, but List<String>.class and List<?>.classare not.

**This is the preferred way to use the instanceof operator with generic types:**

*// Legitimate use of raw type - instanceof operator* **if** (o **instanceof** Set) { *// Raw type* Set<?> s = (Set<?>) o; *// Wildcard type*}

## ITEM 27: ELIMINATE UNCHECKED WARNINGS

**If you can’t eliminate a warning, but you can prove that the code that provoked the warning is typesafe, then (and only then) suppress the warning with  
an @SuppressWarnings("unchecked") annotation.**

**Always use the SuppressWarnings annotation on the smallest scope possible.** Never use SuppressWarnings on an entire class. Doing so could mask critical warnings.

**Every time you use a @SuppressWarnings("unchecked") annotation, add a comment saying why it is safe to do so.**

## ITEM 28: PREFER LISTS TO ARRAYS

Arrays differ from generic types in two important ways. First, arrays are *covariant*. This scary-sounding word means simply that if Sub is a subtype of Super, then the array type Sub[] is a subtype of

the array type Super[]. Generics, by contrast, are *invariant*: for any two distinct types Type1 and Type2, List<Type1> is neither a subtype nor a supertype of List<Type2>

*// Fails at runtime!*Object[] **objectArray** = **new** Long[1]; **objectArray**[0] = **"I don't fit in"**; *// Throws ArrayStoreException*but **this** one is not:   
*// Won't compile!*List<Object> **ol** = **new** ArrayList<Long>(); *// Incompatible types***ol**.add(**"I don't fit in"**);

Either way you can’t put a String into a Long container, but with an array you find out that you’ve made a mistake at runtime; with a list, you find out at compile time. Of course, you’d rather find out at compile time.

泛型提供了编译期安全

you can’t create an array of a non-reifiable type, such as E. This problem arises every time you write a generic type that is backed by an array.

## ITEM 29: FAVOR GENERIC TYPES

## ITEM 30: FAVOR GENERIC METHODS

## ITEM 31: USE BOUNDED WILDCARDS TO INCREASE API FLEXIBILITY

# Chapter 8. Methods

## ITEM 49: CHECK PARAMETERS FOR V ALIDITY

## ITEM 50: MAKE DEFENSIVE COPIES WHEN NEEDED

**You must program defensively, with the assumption that clients of your class will do their best to destroy its invariants.**

传递引用的时候，修改引用会导致被引用的也修改。所以最好传递copy

**do not use the clone method to make a defensive copy of a parameter whose type is subclassable by untrusted parties.**

## ITEM 51: DESIGN METHOD SIGNATURES CAREFULLY

**Choose method names carefully.**

## ITEM 52: USE OVERLOADING JUDICIOUSLY

**the choice of which overloading to invoke is made at compile time.**

**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));  
 }  
}

You might expect this program to print Set, followed  
by List and Unknown Collection, but it doesn’t. It prints Unknown Collection three times. Why does this happen? Because  
the classifymethod is *overloaded*, and **the choice of which overloading to invoke is made at compile time.** For all three iterations of the loop, the compile-time type of the parameter is the same: Collection<?>. The runtime type is different in each iteration, but this does not affect the choice of overloading. Because the compile-time type of the parameter is Collection<?>, the only applicable overloading is the third one, classify(Collection<?>), and this overloading is invoked in each iteration of the loop.

The behavior of this program is counterintuitive  
because **selection among overloaded methods is static, while selection among overridden methods is  
dynamic.** The correct version of an *overridden* method is chosen at runtime, based on the runtime type of the object on which the method is invoked. As a reminder, a method is overridden when a subclass contains a method declaration with the same signature as a method declaration in an ancestor. If an instance method is overridden in a subclass and this method is invoked on an instance of the subclass, the subclass’s *overriding method*executes, regardless of the compile-time type of the subclass instance. To make this concrete, consider the following program:

**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) {  
 List<Wine> wineList = List.of(**new** Wine(), **new** SparklingWine(), **new** Champagne());  
 **for** (Wine wine : wineList) System.***out***.println(wine.name());  
 } }

The name method is declared in class Wine and overridden in subclasses SparklingWine and Champagne. As you would expect, this program prints out wine, sparkling wine, and champagne, even though the compile-time type of the instance is Wine in each iteration of the loop. The compile-time type of an object has no effect on which method is executed when an overridden method is invoked; the “most specific” overriding method always gets executed. Compare this to overloading, where the runtime type of an object has no effect on which overloading is executed; the selection is made at compile time, based entirely on the compile-time types of the parameters.

**A safe, conservative policy is never to export two overloadings with the same number of parameters.**

**you can always give methods different names instead of overloading them.**

## ITEM 53: USE VARARGS JUDICIOUSLY

## ITEM 54: RETURN EMPTY COLLECTIONS OR ARRAYS, NOT NULLS

## ITEM 55: RETURN OPTIONALS JUDICIOUSLY

## ITEM 56: WRITE DOC COMMENTS FOR ALL EXPOSED API ELEMENTS

## ITEM 57: MINIMIZE THE SCOPE OF LOCAL VARIABLES

**The most powerful technique for minimizing the scope of a local variable is to declare it where it is first used.**

**Nearly every local variable declaration should contain an initializer.** If you don’t yet have enough information to initialize a variable sensibly, you should postpone the declaration until you do.

## ITEM 58: PREFER FOR-EACH LOOPS TO TRADITIONAL FOR LOOPS

## ITEM 59: KNOW AND USE THE LIBRARIES

## ITEM 60: AVOID FLOAT AND DOUBLE IF EXACT ANSWERS ARE REQUIRED

For example, suppose you have a dollar in your pocket, and you see a shelf with a row of delicious candies priced at 10¢, 20¢, 30¢, and so forth, up to a dollar. You buy one of each candy, starting with the one that costs 10¢, until you can’t afford to buy the next candy on the shelf. How many candies do you buy, and how much change do you get? Here’s a naive program designed to solve this problem:

**public static void** main(String[] args) {  
 **double** funds = 1.00;  
 **int** itemsBought = 0;  
 **for** (**double** price = 0.10; funds >= price; price += 0.10) {  
 funds -= price;  
 itemsBought++;  
 }  
 System.***out***.println(itemsBought + **" items bought."**);  
 System.***out***.println(**"Change: $"** + funds);  
}

output:

3 items bought.

Change: $0.3999999999999999

If you run the program, you’ll find that you can afford three pieces of candy, and you have $0.3999999999999999 left. This is the wrong answer! The right way to solve this problem is to **useBigDecimal, int, or long for monetary calculations**.

**public static void** main(String[] args) {  
 **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)) {  
 funds = funds.subtract(price);  
 itemsBought++;  
 }  
 System.***out***.println(itemsBought + **" items bought."**);  
 System.***out***.println(**"Money left over: $"** + funds);  
}

In summary, don’t use float or double for any calculations that require an exact answer. Use BigDecimal if you want the system to keep track of the decimal point and you don’t mind the inconvenience and cost of not using a primitive type.

## ITEM 61: PREFER PRIMITIVE TYPES TO BOXED PRIMITIVES

There are three major differences between primitives and boxed primitives. **First**, primitives have only their values, whereas boxed

primitives have identities distinct from their values. In other words, two boxed primitive instances can have the same value and different identities.

**Second**, primitive types have only fully functional values, whereas each boxed primitive type has one nonfunctional value, which is null, in addition to all the functional values of the corresponding primitive type.

**Last**, primitives are more time- and space-efficient than boxed primitives.

**// Broken comparator - can you spot the flaw?**

Comparator<Integer> naturalOrder = (i, j) -> (i < j) ? -1 : (i == j ? 0 : 1);

This comparator looks like it ought to work, and it will pass many tests. For example, it can be used with Collections.sort to correctly sort a million-element list, whether or not the list contains duplicate elements. But the comparator is deeply flawed. To convince yourself of this, merely print the value

of naturalOrder.compare(new Integer(42), new Integer(42)).  
Both Integer instances represent the same value (42), so the value of this expression should be 0, but it’s 1, which indicates that the first Integer value is greater than the second!

**Applying  
the == operator to boxed primitives is almost always wrong.**

修改：

Comparator<Integer> naturalOrder = (iBoxed, jBoxed) -> { int i = iBoxed, j = jBoxed; // Auto-unboxing  
return i < j ? -1 : (i == j ? 0 : 1);

};

**public class** Unbelievable {   
 **static** Integer *i*;  
 **public static void** main(String[] args) {   
 **if** (*i* == 42)  
 System.***out***.println(**"Unbelievable"**);   
 }  
}

No, it doesn’t print Unbelievable—but what it does is almost as strange. It throws a NullPointerException when evaluating the expression i==42. The problem is that i is an Integer, not an int, and like all nonconstant object reference fields, its initial value is null. When the program evaluates the expression i==42, it is comparing an Integer to an int. In nearly every case **when you mix primitives and boxed primitives in an operation, the boxed primitive is auto-unboxed.** If a null object reference is auto-unboxed, you get a NullPointerException.

*// 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);  
}

This program is much slower than it should be because it accidentally declares a local variable (sum) to be of the boxed primitive type Long instead of the primitive type long. The program compiles without error or warning, and the variable is repeatedly boxed and unboxed, causing the observed performance degradation.

## ITEM 62: AVOID STRINGS WHERE OTHER TYPES ARE MORE APPROPRIATE

## ITEM 63: BEWARE THE PERFORMANCE OF STRING CONCATENATION

## ITEM 64: REFER TO OBJECTS BY THEIR INTERFACES

**If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types.**

**// Good - uses interface as type Set<**Son> sonSet = new LinkedHashSet<>();

not this:

**Click here to view code image**

**// Bad - uses class as type!  
LinkedHashSet**<Son> sonSet = new LinkedHashSet<>();

ITEM 65: PREFER INTERFACES TO REFLECTION

ITEM 66: USE NATIVE METHODS JUDICIOUSLY

ITEM 67: OPTIMIZE JUDICIOUSLY

ITEM 68: ADHERE TO GENERALLY ACCEPTED NAMING CONVENTIONS

# Chapter 10. Exceptions

ITEM 69: USE EXCEPTIONS ONLY FOR EXCEPTIONAL CONDITIONS

# Chapter 11. Concurrency

ITEM 78: SYNCHRONIZE ACCESS TO SHARED MUTABLE DATA

**Do not use Thread.stop.**

*// Broken! - How long would you expect this program to run?***public class** StopThread {  
 **private static boolean** *stopRequested*;  
 **public static void** main(String[] args) **throws** InterruptedException {  
 Thread backgroundThread = **new** Thread(() -> { **int** i = 0;  
 **while** (!*stopRequested*) i++;  
 }); backgroundThread.start();  
 TimeUnit.SECONDS.sleep(1);  
 *stopRequested* = **true**; }  
}

You might expect this program to run for about a second, after which the main thread sets stopRequested to true, causing the background thread’s loop to terminate. On my machine, however, the program *never* terminates: the background thread loops forever!

The problem is that in the absence of synchronization, there is no guarantee as to when, if ever, the background thread will see the change in the value of stopRequested made by the main thread. In the absence of synchronization, it’s quite acceptable for the virtual machine to transform this code:

**while** (!stopRequested){  
 i++;  
}

into this code:

**if** (!stopRequested) {  
 **while** (**true**)  
 i++;  
}

One way to fix the problem is to synchronize access to the stopRequested field.

*// Properly synchronized cooperative thread termination***public class** StopThread {  
 **private static boolean** *stopRequested*;  
 **private static synchronized void** requestStop() {

*stopRequested* = **true**;  
 }  
 **private static synchronized boolean** stopRequested() {

**return** *stopRequested*;  
 }  
 **public static void** main(String[] args) **throws** InterruptedException {  
 Thread backgroundThread = **new** Thread(() -> {  
 **int** i = 0;  
 **while** (!*stopRequested*())  
 i++; });  
 backgroundThread.start();  
 TimeUnit.SECONDS.sleep(1);  
 *requestStop*();

}  
}

Note that both the write method (requestStop) and the read method (stop-Requested) are synchronized. It is *not* sufficient to synchronize only the write method! **Synchronization is not guaranteed to work unless both read and write operations are synchronized.**

*// Cooperative thread termination with a volatile field***public class** StopThread {  
 **private static volatile boolean** *stopRequested*;  
 **public static void** main(String[] args) **throws** InterruptedException {  
 Thread backgroundThread = **new** Thread(() -> { **int** i = 0;  
 **while** (!*stopRequested*) i++;  
 }); backgroundThread.start();  
 TimeUnit.SECONDS.sleep(1);  
 *stopRequested* = **true**; }  
}

You do have to be careful when using volatile. Consider the following method, which is supposed to generate serial numbers:

*// Broken - requires synchronization!***private static volatile int** *nextSerialNumber* = 0;  
**public static int** generateSerialNumber() {   
 **return** *nextSerialNumber*++;  
}

Still, the method won’t work properly without synchronization.  
The problem is that the increment operator (++) is not atomic. It performs *two* operations on the nextSerialNumber field: first it reads the value, and then it writes back a new value, equal to the old  
value plus one. If a second thread reads the field between the time  
a thread reads the old value and writes back a new one, the second thread will see the same value as the first and return the same  
serial number. This is a *safety failure*: the program computes the wrong results.

One way to fix generateSerialNumber is to add  
the synchronized modifier to its declaration. This ensures that multiple invocations won’t be interleaved and that each invocation of the method will see the effects of all previous invocations. Once you’ve done that, you can and should remove the volatile modifier from nextSerialNumber. To bulletproof the method, use long instead  
of int, or throw an exception if nextSerialNumber is about to wrap. Better still, follow the advice in Item 59 and use the class AtomicLong, which is part of java.util.concurrent.atomic. This package provides primitives for lock-free, thread-safe programming on single variables. While volatile provides only the communication effects of synchronization, this package also provides atomicity.

*// Lock-free synchronization with java.util.concurrent.atomic***private static final** AtomicLong ***nextSerialNum*** = **new** AtomicLong();  
**public static long** generateSerialNumber() {  
 **return *nextSerialNum***.getAndIncrement();  
}

The best way to avoid the problems discussed in this item is not to share mutable data. Either share immutable data (Item 17) or don’t share at all. In other words, **confine mutable data to a single thread.**

**when multiple threads share mutable data, each thread that reads or writes the data must perform synchronization.**

ITEM 79: AVOID EXCESSIVE SYNCHRONIZATION

**To avoid liveness and safety failures, never cede control to the client within a synchronized method or block.**

In other words, inside a synchronized region, do not invoke a method that is designed to be overridden, or one provided by a client in the form of a function object

An alien method might run for an arbitrarily long period. If the alien method were invoked from a synchronized region, other threads would be denied access to the protected resource unnecessarily.  
**As a rule, you should do as little work as possible inside synchronized regions.**

If you are writing a mutable class, you have two options: you can omit all synchronization and allow the client to synchronize externally if concurrent use is desired, or you can synchronize internally, making the class *thread-safe* (Item 82). You should choose the latter option only if you can achieve significantly higher concurrency with internal synchronization than you could by having the client lock the entire object externally. The collections in java.util (with the exception of the obsolete Vectorand Hashtable) take the former approach, while those in java.util.concurrent take the latter (Item 81).

*// Broken - invokes alien method from synchronized block!***public class** ObservableSet<E> **extends** ForwardingSet<E> {  
 **public** ObservableSet(Set<E> set) {  
 **super**(set);  
 }  
  
 **private final** List<SetObserver<E>> **observers** = **new** ArrayList<>();  
  
 **public void** addObserver(SetObserver<E> observer) {  
 **synchronized** (**observers**) {  
 **observers**.add(observer);  
 }  
 }  
  
 **public boolean** removeObserver(SetObserver<E> observer) {  
 **synchronized** (**observers**) {  
 **return observers**.remove(observer);  
 }  
 }  
  
 **private void** notifyElementAdded(E element) {  
 **synchronized** (**observers**) {  
 **for** (SetObserver<E> observer : **observers**)  
 observer.added(**this**, element);  
 }  
 }  
  
 @Override  
 **public boolean** add(E element) {  
 **boolean** added = **super**.add(element);  
 **if** (added)  
 notifyElementAdded(element);  
 **return** added;  
 }  
  
 @Override  
 **public boolean** addAll(Collection<? **extends** E> c) {  
 **boolean** result = **false**;  
 **for** (E element : c)  
  
 result |= add(element); *// Calls notifyElementAdded return result;* **return** result;  
 }  
}

**void** run(){  
 ObservableSet<Integer> set =  
 **new** ObservableSet<>(**new** HashSet<>()); set.addObserver((s, e) -> System.***out***.println(e));  
 **for** (**int** i = 0; i < 100; i++) set.add(i);  
}  
**public static void** main(String[] args) {  
 Main m=**new** Main();  
 m.run();  
}

For example, the following program prints the numbers  
from 0 through 99:

Now let’s try something a bit fancier. Suppose we replace  
the addObserver call with one that passes an observer that prints the Integer value that was added to the set and removes itself if the value is 23:

ObservableSet<Integer> set =  
 **new** ObservableSet<>(**new** HashSet<>());  
set.addObserver(**new** SetObserver<>() {  
 **public void** added(ObservableSet<Integer> s, Integer e) {  
 System.***out***.println(e);  
 **if** (e == 23) s.removeObserver(**this**);  
 } });  
**for** (**int** i = 0; i < 100; i++) set.add(i);

You might expect the program to print the numbers 0 through 23, after which the observer would unsubscribe and the program would terminate silently. In fact, it prints these numbers and then throws a ConcurrentModificationException.

The problem is that notifyElementAdded is in the process of iterating over  
the observers list when it invokes the observer’s added method.  
The added method calls the observable set’s removeObserver method, which in turn calls the method observers.remove. Now we’re in trouble. We are trying to remove an element from a list in the midst of iterating over it, which is illegal. The iteration in  
the notifyElementAdded method is in a synchronized block to prevent concurrent modification, but it doesn’t prevent the iterating thread itself from calling back into the observable set and modifying  
its observers list.

Now let’s try something odd: let’s write an observer that tries to unsubscribe, but instead of calling removeObserver directly, it engages the services of another thread to do the deed. This observer uses an *executor service* (Item 80):

set.addObserver(**new** SetObserver<>() {  
 **public void** added(ObservableSet<Integer> s, Integer e) {  
 System.***out***.println(e); **if** (e == 23) {  
 ExecutorService exec = Executors.*newSingleThreadExecutor*();  
 **try** {  
 exec.submit(() -> s.removeObserver(**this**)).get();  
 } **catch** (ExecutionException | InterruptedException ex) { **throw new** AssertionError(ex);  
 } **finally** { exec.shutdown();  
 } }  
 } });

When we run this program, we don’t get an exception; we get a deadlock.

The background thread calls s.removeObserver, which attempts to lock observers, but it can’t acquire the lock, because the main thread already has the lock. All the while, the main thread is waiting for the background thread to finish removing the observer, which explains the deadlock.

In fact, there’s a better way to move the alien method invocations out of the synchronized block. The libraries provide a *concurrent collection* (Item 81) known as CopyOnWriteArrayList that is tailor-made for this purpose.

*// Thread-safe observable set with CopyOnWriteArrayList***private final** List<SetObserver<E>> **observers** =  
 **new** CopyOnWriteArrayList<>();  
  
**public void** addObserver(SetObserver<E> observer) {  
 **observers**.add(observer);  
}  
  
**public boolean** removeObserver(SetObserver<E> observer) {  
 **return observers**.remove(observer);  
}  
  
**private void** notifyElementAdded(E element) {  
 **for** (SetObserver<E> observer : **observers**)  
 observer.added(**this**, element);  
}