2019

Programming Safety Tips: Why You Should Use Immutable Objects or How to create programs with bugs that can never be found or fixed.

Charles W. Kann
Gettysburg College

Programming Safety Tips: Why You Should Use Immutable Objects or How to create programs with bugs that can never be found or fixed.

Abstract

Program safety deals with how to make programs as error free as possible. The hardest errors in a program for a programmer to find are often errors in using memory. There are two reasons for this. The first is that errors in accessing memory almost never show problems in the proximate area of the program where the error is made. The error has no apparent impact when it is made, but often causes catastrophic results to occur much later in the program, in areas of the program unrelated to memory error that caused it.

The second reason memory errors are so difficult to find is that the working of memory is often poorly understood by most novice, and many professional, programmers. This makes it difficult for many programmers to even understand why an action causes the error.

This article will show an example of a program error that can easily occur when memory access is poorly understood. This leads to program errors that are very easy to fix when they are found, but extremely difficult to find. The article will then explain how many memory errors can be easily avoided by following the very simple rule, “Make all object immutable unless there is a good reason to make them mutable”, and why immutable objects are an essential tool in good, safe programming practice.

Keywords

java "immutable objects" "program safety" "object oriented programming"

Disciplines

Other Computer Sciences | Programming Languages and Compilers | Software Engineering

Programming Safety Tips: Why You Should Use Immutable Objects:

or

How to create programs with bugs that can never be found or fixed.

Charles W. Kann III
chuck(at)chuckkann.com

Abstract

Program safety deals with how to make programs as error free as possible. The hardest errors in a program for a programmer to find are often errors in using memory. There are two reasons for this. The first is that errors in accessing memory almost never show problems in the proximate area of the program where the error is made. The error has no apparent impact when it is made, but often causes catastrophic results to occur much later in the program, in areas of the program unrelated to memory error that caused it.

The second reason memory errors are so difficult to find is that the working of memory is often poorly understood by most novice, and many professional, programmers. This makes it difficult for many programmers to even understand why an action causes the error.

This article will show an example of a program error that can easily occur when memory access is poorly understood. This leads to program errors that are very easy to fix when they are found, but extremely difficult to find. The article will then explain how many memory errors can be easily avoided by following the very simple rule, “Make all object immutable unless there is a good reason to make them mutable”, and why immutable objects are an essential tool in good, safe programming practice.

Note that the examples in this article are in Java, but the principals outlined in the article are generically applicable to programming and can be applied in many programming languages.

Introduction

Many people believe that some chess players are better than others because they can see more moves into the future. The truth is that the complexity of a position makes it impossible for anyone to evaluate more than a few moves or combinations of moves. What makes a good chess player is that ability to recognize good structures in the position of their pieces, and to maintain good positions while working towards better ones.

I was thinking about how program structure reduces complexity in programs when I first wrote this article about 20 years ago. I was drawn to Object Orient Programming (OOP) because it has many structures that help a programmer avoid complexity and the errors that complexity cause. Early on in my career I realized that it is not the ability to think algorithmically that makes a good programmer. What is important to good programming is understanding the importance of good program structure on the complexity of a program.

This article will explain how structuring a program around immutable objects can be used to reduce the complexity, and thus enhance the safety, of a program. This article will first define and show an immutable object. There are many articles online that explain immutable objects and how to create them in detail, and this article is not intended to add to this already crowded topic. Instead this article will show how using immutable objects make a program safer.

This article will create an example using mutable object and show how mutable objects can be used in a Java HashTable to create a memory error that is non-obvious and nearly impossible to find in a large program. The article will then explain how using immutable objects as keys to a Map removes any possibility of this specific error ever occurring. This example illustrates how immutable objects prevent the inadvertent addition of a bug in programming logic, reinforcing the point that good programming relies not on good algorithmic thinking, but on good structure.

Before I begin describing immutable objects, I would like to describe some of the history of this article. This article has been around for many years. When I first was introduced to OOP, it was using C++ and Ada. While these languages did provide flexibility and reuse, I could not really understand the design safety that Ada tried to provide. As for C++, the language itself provides little more than lip service, if that much, with regards to how a programming language can enhance program safety.

When I was introduced to Small Talk, and later Java, I could suddenly find much of the program safety I was looking for in an OOP language. Interfaces provided polymorphism in a safe context. Runtime type checking made casting of objects not only easy, but casting made sense and worked. Finally, concepts such as immutable types, while not built into the Java language, could be implemented and enforced.

I wrote this article and tried to get it published, but at the time in the late 1990's, you still needed to get articles into magazines or other print media to get distribution. This article did not seem to meet a need at any of the places I tried to publish it, possibly because the concept of program safety was not important in the major languages of interest at that time.

Over time the concept of program safety, and how a language can program safety, has become more central to the idea of programming. I felt the need for this type of article is more relevant to current thinking about programming.

With the advent of the internet, I no longer need a print distribution network because I can put my work out for free and widely distribute it. I do not even need an editor, though I know I could sorely use one. But by avoiding all the trappings of the previous world of publishing, I can make useful material available for free. I know this works, as I have had several books with nearly 100,000 downloads from nearly every country on the planet.

Finally, I have always thought this article is a good illustration of several concepts and have used this article in my classes as an example of why object immutability is an extremely important tool that every programmer should know and use. I also use the concepts in this article to illustrate the hazards of uncontrolled memory. Recently using this article in a class, I decided to update it and post it on the internet.

What is an Immutable Object?

An immutable object is an object where the values referenced by the object cannot be changed after the object has been constructed. To see this, consider the Java primitive type `int`. If an `int` is declared `final`, its value cannot be changed.

```
final int j = 7;
j = 12; // illegal assignment
```

An immutable object is like a final primitive in that it can never be changed. However, objects are generally more complicated than primitive types because they are made up of several other variables, and these variables can be primitives or other objects. Because of the makeup of an object, simply declaring a variable `final` does not mean that the data inside of the object can never change.   Thus, declaring an object variable final does not always mean that the object cannot be changed. To create an object that cannot be changed the object must be made immutable.

This article is about the benefits of immutability, and how structuring your program around concepts like immutable objects will help you avoid some program errors that will result in very horrible bugs. To start, a brief example of a mutable and an immutable object are given.

Listing 1 shows the class definition for a mutable `Name` object. In this definition, the class includes a `setName` method that allows the object to be modified after the object has been constructed.

```
/**
 * A mutable Name class
 */
class Name {
    private String firstName;
    private String lastName;

    public Name(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public void setName(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String toString() {
        return (firstName + " " + lastName);
    }
}
```

Listing 1 –Mutable Name Class

To illustrate the point that the object is mutable, and that the `final` keyword on an object variable does not ensure the variable is immutable, Listing 2 provides a program that changes the `Name` object. The meaning of the final modifier is often misunderstood, even by experienced programmers. It means the object reference cannot be changed, but as is shown here, the object itself is still mutable and can be changed.

```
public class NameTest {
  public static void main(String... args) {
      final Name n = new Name("Chuck", "Kann");
      n.setName("Jessica", "Meng");
  }
}
```

Listing 2 – Using a Mutable Object with a final modifier

Listing 3 creates an immutable object. Note that an object created from this class cannot be changed after it is constructed. The biggest change between the classes in Listing1 and Listing 3 is the presence of the `setName` method in Listing 1. However, removing the `setName` method is just one change that must be considered when creating an immutable object. For the reader who is interested, referring to an article on how to make objects immutable will show that this new object is indeed immutable..

```
/**
 * An immutable Name class
 */
final class Name {
    private final String firstName;
    private final String lastName;

    public Name(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String toString() {
        return (firstName + " " + lastName);
    }
}
```

Listing 3 – Immutable Name Class

Many readers steeped in classes that taught classes are created by including setter and getter methods on all fields in an object are likely at this point questioning why not just put the setName method in the class. It does not seem to cause any problems and could be useful. The rest of this article will illustrate the peril in that by implementing a simple program that uses a `HashMap.` The program will use a mutable Name object as the key and result in a severe and obscure bug. This bug will be easily fixed by making the class immutable.

How a Simple HashMap works

To use an object as key to a `HashMap`, the class that defines that object must include two methods, the `hashCode` and `equals` methods. Shortly we will explain how these two methods work with a `HashMap`, but for now they are added to the mutable implementation of the Name class in Listing 4.

```
/**
 *  A mutable Name class, used as a key to a HashMap
 */
public class Name {
    String firstName;
    String lastName;

    public Name(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public void setName(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public boolean equals(Object name) {
        Name toCheck = (Name) name;
        return (firstName.equals(toCheck.firstName) &&
                lastName.equals(toCheck.lastName));
    }

    public int hashCode() {
        String s = firstName + lastName;
        return s.hashCode();
    }

    public String toString() {
        return (firstName + " " + lastName);
    }
}
```

Listing 4 –Mutable Name class for use in HashMap

An instance of the Name class from Listing 4 is now used in a HashMap in the program in Listing 5. This program shows that the key is added to the HashMap as expected, and appears to confirm that a mutable object is an acceptable key to a Map.

```
/**
 *  A Main program that shows the insertion of the key
 *  “Kanga Roo” into a HashMap.
 */

import java.util.HashMap;
```

```
      import java.util.Iterator;

      public class Listing5 {
          public static void main(String args[]) {
              HashMap<Name, Object> table = new HashMap<Name,
                                                  Object>();

              // Add and retrieve "Kanga" "Ro"
              Name name = new Name("Kanga", "Roo");
              table.put(name, new Object());
              if (table.get(new Name("Kanga", "Roo")) != null)
                  System.out.println("Kanga Roo found");
              else
                  System.out.println("Kanga Roo not found");
          }
}
```

To understand the problem which we will encounter using mutable objects as keys, it is necessary to know how a `HashMap` works. When inserting or looking for data in a `HashMap` there are two steps. The first step uses a *hash function* on the key which reduces the key into an index into an array. In Java this hash function must exist in the class for the object that is to be used for the key, and it must be the method named `hashCode`. In Listing 4 we have implemented the `hashcode` method by concatenating the `firstName` and `lastName` data fields to create a `String` object containing both names, and then applied the `hashCode` method from the class `String` to the concatenated field. This is a simple way to implement a `hashCode` method when two or more strings are used as a concatenated key.

Once the index in the array where the key is to be stored has been determined, the key is placed at that index in the array. However, because of the nature of hashing there are often cases where two or more values will hash to the same index in the array. So, a second step is needed where the `HashMap` keeps a linked list of keys that were mapped to the same value and places the key in this list. When a `HashMap` wants to find a key, it simply hashes the key that it is looking for into an array position, and then follows the linked list of keys at that array position looking for the appropriate key. If the key is not found in the linked list of keys at the array position pointed to by the hash function, it assumes the key is not present in the `HashMap`.

To illustrate this process, figure 1 shows how the program in Listing 5 works. In the `main` method an object with a key that has a first name “Kanga” and a last name “Roo” is added to a `HashMap`. To add the key, first the `hashCode` method for the `Name` class is called to get an array position to place the key “Kanga Roo”, as illustrated in Step 1 of Figure 1. The key is then inserted into the linked list of keys at that array position, as illustrated in Step 2.

Later when the key “Kanga Roo” is needed, the `HashMap` again uses the `hashCode` method on the key to find the position in the array where “Kanga Roo” is stored, and then uses the `equals` method to look in the linked list to find the appropriate key. If no object is found in the chain at the array index, the `HashMap` determines that the key is not contained in the `HashMap`.

Note that the keys to the table must be unique to ensure that each key can retrieve the object stored with a key. This will be important later.

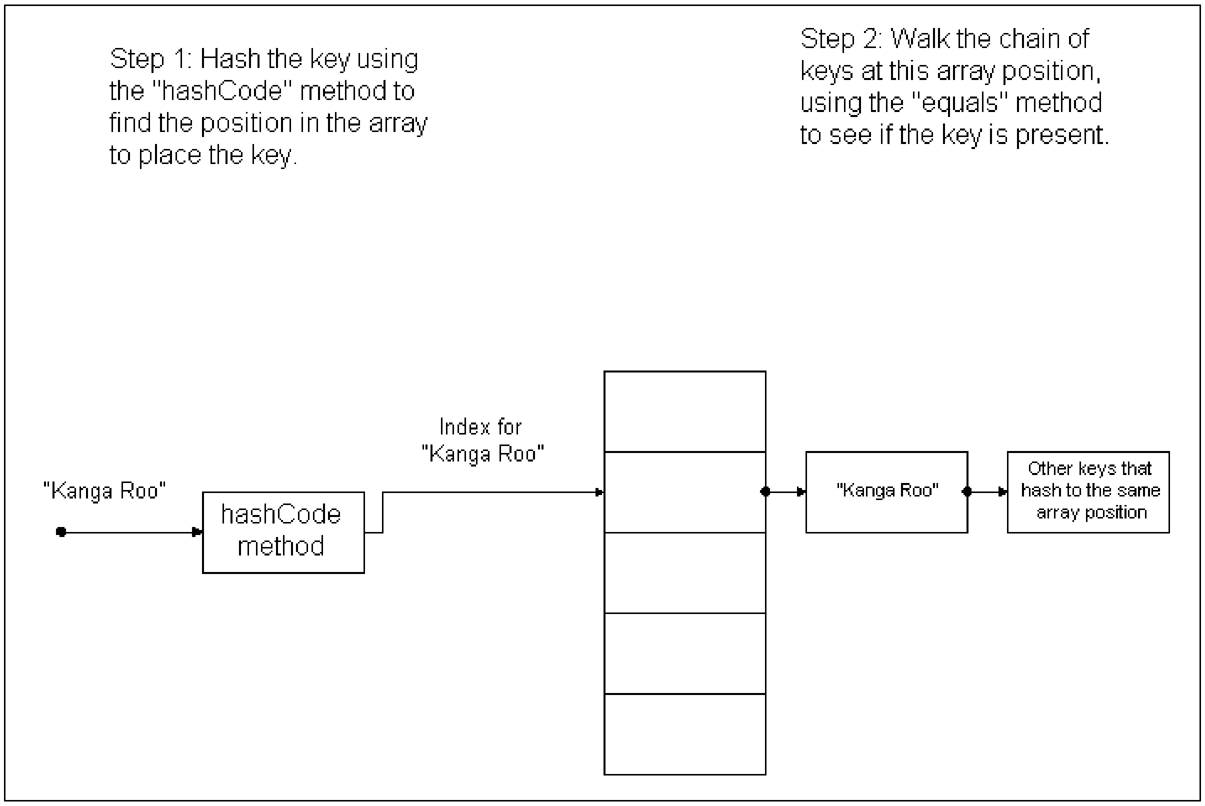
Step 1: Hash the key using the "hashCode" method to find the position in the array to place the key.
Step 2: Walk the chain of keys at this array position, using the "equals" method to see if the key is present.
"Kanga Roo"
hashCode method
Index for "Kanga Roo"
"Kanga Roo"
Other keys that hash to the same array position

Figure 1 – Inserting “Kanga Roo” into a HashMap

So what is the problem?

The program meets the basic functionality of inserting the “Kanga Roo” object into the `HashMap`. However, consider the following situation where a programmer wants to insert another key into the `HashMap` for the key “Koala Bear”. For efficiency the programmer decides to reuse the `name` object that is conveniently available and has a setter that allows them to change the values in the object. This program change is shown in Listing 6.

```
/*
 *  This program shows how if a mutable object
 *  is used as a key in a Hashtable, it is
 *  possible to completely lose the object.
 */

import java.util.HashMap;
import java.util.Iterator;
```

```
public class Listing6 {
    public static void main(String args[]) {
        HashMap<Name, Object> table = new HashMap<Name,
                                          Object>();
        Name name = new Name("Kanga", "Roo");
        table.put(name, new Object());

        // Now change the name object, losing the key.
        name.setName("Koala", "Bear");
        table.put(name, new Object());

        // Now try to find the object, which appears to be gone!
        if (table.get(new Name("Kanga", "Roo")) != null)
            System.out.println("Kanga Roo found");
        else
            System.out.println("Kanga Roo not found");

        // Kanga Roo is not a key, but Koala Bear is in twice!
        System.out.println("\nPrint the keys");
        Iterator i = (table.keySet()).iterator();
        while (i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

Listing 6 –Losing the key "Kanga Roo"

When this program is run, it produces the output in Figure 2. The output shows that once the key "Koala Bear" is inserted into the `HashMap`, the key "Kanga Roo" can no longer be found. Even more strange is the fact that the output shows the key "Koala Bear" is stored twice in the `HashMap`, which violates the rule, enforced by the HashMap, that says a key must be unique. What has happened?

```
Kanga Roo not found

Print the keys
Koala Bear
Koala Bear
```

Figure 2 – Output from Listing 6

To see how this problem came about, consider Figure 3, which represents the `HashMap` after the key "Koala Bear" has been inserted. In this figure, we can see that when the key for "Kanga Roo" was changed to "Koala Bear" the original `name` object was changed. However, the index was not moved to the array position that would correspond to "Koala Bear". Thus, the key for "Kanga Roo" is no longer stored in the `HashMap`, since it is now "Koala Bear". But the first key for "Koala Bear" is stored not in the index position for a "Koala Bear", but for a "Kanga Roo", and is thus not found when looking to insert a new key of "Koala Bear". The key for "Koala Bear" is effectively stored twice in the `HashMap`, once at the correct index, and once at the incorrect index. The `HashMap` is not aware that the key for "Koala Bear" is stored twice because it cannot see the "Koala

Bear" that is stored at the incorrect index. Thus, the key for the object "Kanga Roo" has been lost in the hash map, and the key structure for the `HashMap` has been completely corrupted.

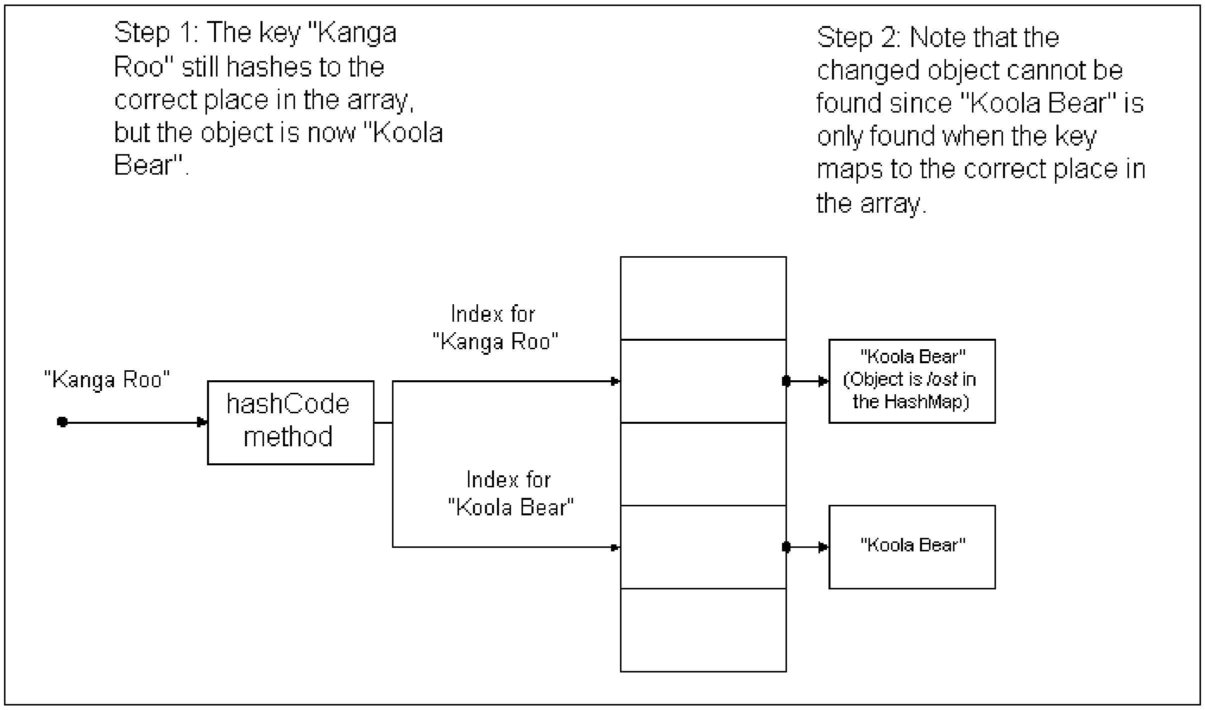
Step 1: The key "Kanga Roo" still hashes to the correct place in the array, but the object is now "Koola Bear".
Step 2: Note that the changed object cannot be found since "Koola Bear" is only found when the key maps to the correct place in the array.
"Kanga Roo"
hashCode method
Index for "Kanga Roo"
Index for "Koola Bear"
"Koola Bear" (Object is *lost* in the HashMap)
"Koola Bear"

Figure 3 - Losing the key "Kanga Roo" from the HashMap

When reading this code, it is important to note that there is no obvious problem with the `Name` class, or the way that the `hashCode` or `equals` methods have been implemented. There is also no bug in the implementation of the `HashMap`. Everything is working exactly as it was intended. This effectively hides the error from most programmers.

When the code for this bug is presented in a simplified fashion as above, the error becomes obvious. However, this type of error is less obvious in a system of thousands (or tens or even hundreds of thousands) of lines of code, and where countless records added or deleted from the `HashMap` from many different places in the program. That data is lost, and the key structure of the table compromised is a serious problem, but the root cause, while relatively simple to fix, would be difficult to find.

How to solve the problem

The solution to the problem illustrated above is as simple as it is obvious. There is no good reason to allow an object used as a key to ever be changed. This can be done by removing the `setName` method from the `Name` class and not allow `name` objects to be reused. More precisely, this means making the `Name` object immutable. If the object is immutable and cannot be changed, there is no possibility of it appearing at the wrong index in the HashMap array. When a programmer now wants to insert a new key into the

`HashMap`, they will always have to instantiate a new object, and thus this problem cannot happen.

Having pointed out the program safety advantage of making objects immutable, I realize that many programmers respond badly to this type of suggestion. When suggesting this type of enhancement to program safety these programmers will say that if they want to change the 3 bit of a variable, for any reason, they know what they are doing and should be allowed to do it without the language enforcing any rules, or requiring them to document why they do it. A good programmer knows what they are doing, and should be allowed to do anything they feel is necessary.

While it might be true that a good programmer will know about memory errors such as the ones illustrated in this article, the first thing that must be pointed out is that few programs will ever be completely written and maintained by "good" programmers. And even if the "good" programmer can assure that no one but themselves will ever change the program, even the best-intentioned programmer can lose sight of potential problems when faced with complex situations, especially when combined with time constraints or other aggravating circumstances.

To me, the bottom line is that an error like this would be very hard to trace and fix. While making an object immutable seems to be taking away some of its flexibility, this flexibility is largely illusionary because it is seldom safe to change the value of an object which is used as a key. And even if there were reasons to reuse the object, the gain in the safety of the program should make the programmer at least seriously consider the use of immutable objects.

Finally, I will point out to the reader who might ask, "If this problem with changing keys to a `HashMap` is so important, why doesn't it occur frequently when using Java Map classes?" The answer is very simple. When programmers use a Java Map class, very often they will use classes from java.lang, such as String or Integer. When developing the classes in java.lang the developers of the Java language realized how important immutable objects were and made the most important classes immutable. Thus, unless you are defining your own key for a Java Map class, this problem is unlikely to occur, which makes the problem more perplexing when it does.

Conclusion

Immutable objects have many uses in Java other than just as keys to Java Maps. For example, Java Event objects (such as the Action Event object for an ActionListener) are normally immutable. Programmer defined Throwable objects (such as Exceptions) are also often immutable. Finally, in concurrent programs, immutable objects are often used since they can be safely used by multiple threads. Immutable object can be used to make the programs which use these objects much safer and more robust.

Immutable objects are just one technical design decision that must be made when implementing classes. However, small decisions like this can have very disproportionate effects on program safety. If decisions are made incorrectly, the programmers maintaining the program can be forced to spend an inordinate amount of time chasing problems that should never have been possible to create. Thus, it is important to understand the type of issues you have control over when structuring a program, and to structure the program to your advantage. The use of immutable objects is just one of these structuring decisions, albeit an important one, at your control.