FEATURE STORY

Java Primer

Coding Standards in Java - Do We Need Them?

Six common Java pitfalls and how to prevent them

by Adam Kolawa

Dr. Adam Kolawa is Chairman and CEO of ParaSoft Corp. in Monrovia, CA. He can be contacted at ukola@parasoft.com.

JAVA HAS BEEN heralded as the language that can solve all the problems that C++ has. Java is supposed to be less error-prone and easier to program with; it is also supposed to resolve many of the difficulties of C++, such as memory corruption and memory management.

Presenting Java as the miracle solution to all of our programming difficulties is a dangerous idea. Whereas developers alertly watch for errors in C++, the common perception of Java as a "safe" language has encouraged developers to let their guards down as they write code. At first, Java programmers believed that the new language was strict enough to keep them out of error trouble. However, as time passed and programmers gained more experience with the language, they started to recognize the many opportunities for making mistakes in a language that was originally thought to be "bug-free."

Java developers who relax their standards invariably find themselves in trouble when it comes time to debug their code. Early attempts to warn programmers about Java pitfalls led to the creation of coding guidelines, which in turn have been adopted as internal coding standards by many companies.

I believe that coding standards are every bit as important in Java as they have been in C/C++. However, rather than merely bombarding you with arguments that support my thesis, I will describe six common Java pitfalls and present you with coding standards that prevent them. I will therefore let you decide whether it is necessary to enforce coding standards in Java.

Naming Variables

You are writing code and choosing names for your variables. *How can you make sure that you and other developers will be able to connect your variable names to their real-life functions?*

Code should be as easy to read and understand as possible, because it is merely a language that represents real-life functions and processes. Anyone who looks at the code even months after it's been written should be able to determine its function at a glance.

There is another important reason to produce clearly-written code: Later in the development process, when it is time to check code for errors, you should be able to debug the algorithm and easily determine whether the logic is correct. Therefore, when you choose names for your variables, you should be trying to connect the variables to real life.

The quickest and easiest solution if you are trying to connect the variables to real life might be to make up an abbreviation for each variable. You will save a few keystrokes every time you use a variable during the coding process, and you can assign abbreviations that have a particular meaning to you. You might write your code as follows:

```
public class ba {
    public static String curr = "dollars";

    public void dep (int i) {
        bal += i;
    }
    public void wit (int i) {
        bal -= i;
    }
    public String get () {
        return Integer.toString (bal) + " " + curr;
    }

    private int bal;
}
```

However, this solution is inadequate, because it severely hampers the code's readability and reusability for other developers. Any medium-sized code written this way will be very difficult to understand; at a glance, no one can tell what the code does because it is just code. An abbreviation such as ba does not necessarily tell the reader that the code is dealing with a bank account. Such abbreviations only have meaning to you, and even you may forget what the abbreviations mean over time.

To determine the function of this code, other developers will have to figure out what each abbreviated variable name stands for. Their minds will be forced to make the constant translation of `ba` into BankAccount and `dep` into deposit at the same time that they are supposedly checking the function of the code.

Therefore, you should use standard naming conventions to connect code to its real-life function. By using naming conventions, you free your mind to focus on one problem at a time, which is the key to successful error prevention. The human brain is far more accurate when it compartmentalizes tasks, and accuracy is of utmost importance in preventing errors. With clearly-written code, you can check the logic of the code and focus on real problems. You can rewrite this same segment of code as follows:

```
public class BankAccount  {
    public static String CURRENCY = "dollars";

    public void deposit (int amount) {
        _balance += amount;
    }
    public void withdraw (int amount) {
        _balance -= amount;
    }
    public String getBalance () {
        return Integer.toString (_balance) + " " + CURRENCY;
    }

    private int _balance;
}
```

There is another advantage to using standard naming conventions: They allow you to apply real-life experience and intuition to code checking. Intuition is important in checking code for the same reason that it is important in understanding mathematics. Students who have trouble with algebra in high school often lack the ability to connect equations to real life. To them, algebra is a meaningless jumble of numbers and letters, and the process of solving a problem remains a mystery. However, when students learn to connect the problems to real life, they often show a marked improvement in their ability to reach a correct answer. They are now using intuition and applying life experience to problems. Applying intuition is also essential to testing the logic of an algorithm.

Furthermore, adhering to naming conventions can have lasting benefits for the productivity of your development group. Developers who join the group later will not waste time being confused by abbreviations that don't follow a particular pattern. The few extra seconds you spend using standard naming conventions today can save hours of extra work for you and your development group later in the development cycle.

In the long run, you may even find naming conventions more convenient than abbreviations, because they will help you avoid the confusion of having to assign and remember abbreviations for a long list of variables. You can save yourself the trouble of improvising to find a different abbreviation for each variable. Developers generally know they should follow naming conventions, but they fail to do it because they mistakenly think it is easier to use abbreviations.

Because the developer writing the second code example above used the variables `BankAccount, deposit, withdrawal,` and `balance`, it is clear that this code is meant to add and subtract money from a bank account and display the final balance. With good naming conventions in place, you and other developers can apply life experience as you check code. Nearly everyone has had the experience of depositing to and withdrawing from a bank account; this experience is invaluable in determining whether a program that performs these functions is working correctly.

Rule: "Use clear naming conventions for identifiers."

Optimizing Performance

You are writing Java code and want it to perform as quickly as possible. *How will you optimize the code and make good use of the Java virtual machine (JVM)?*

The JVM presents a new challenge to developers who have made the switch from C++ to Java. Too often, developers force the JVM to do extra work, which adds overhead to any program. The key to understanding how to optimize Java code is to understand what the JVM really does.

For example, developers often try to optimize Java code by using `String`. `String` is a constant object, which means that by default, it can only be created and read but never modified. You can use `String` when you do not think you will want to modify it as you execute the program:

```
public class MakeMessage {
    public String getMessage (String[] words) {
        String message = "";

        for (int i = 0; i < words.length; i++)
             message += " " + words [i];

        return message;
    }
}
```

This segment of code gets a new word every time it passes through the loop. It appears that we keep creating a new message string and that the message is being appended, but these appearances are deceiving. In fact, the old memory under the message is being disregarded and new memory is being allocated. The old information from the message is copied to the new memory and then a new character is added at the end. The new memory created is one word longer than the old memory.

The code itself is deceptive to read in this instance. Because it contains `message +=`, the code looks as if it will increment the message. However, the message is actually being created from scratch each time. We are creating extra work for the VM because the garbage collector must clean up whatever memory is left behind. If we travel through the loop 1,000 times, there will be 1,000 chunks of memory for the garbage collector to identify and delete. The extra chunks of memory create significant overhead for the program.

Java developers need to understand the difference between `String` and `StringBuffer`. `StringBuffer` is a dynamic object. Because it can be modified, `StringBuffer` can truly be appended, rather than merely giving the false appearance of being appended. Developers can use `StringBuffer` in performance-critical sections of code such as loops.

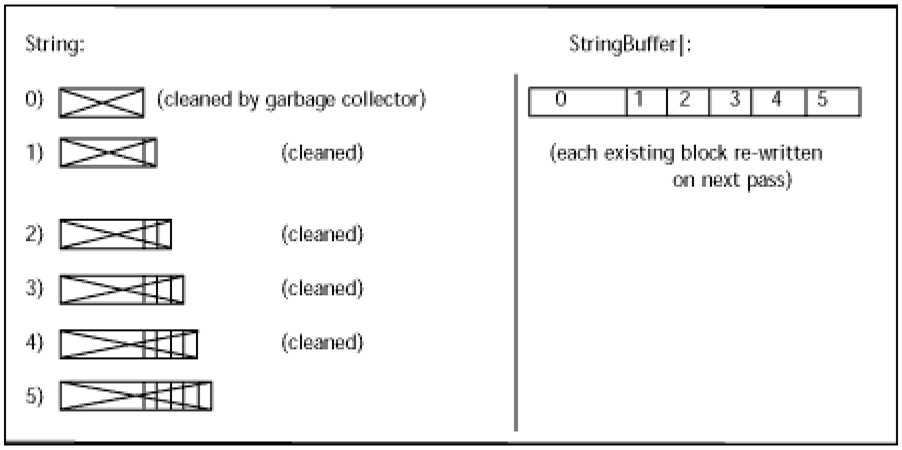
String:
0)
(cleaned by garbage collector)
1)
(cleaned)
2)
(cleaned)
3)
(cleaned)
4)
(cleaned)
5)
StringBuffer|:
0
1
2
3
4
5
(each existing block re-written on next pass)

Figure 1. Memory allotment using String and StringBuffer.

Figure 1 compares memory allotment using `String` and `StringBuffer`, with the numbers representing each pass through the loop.

Therefore, you should define the message as `StringBuffer` rather than `String` in situations where you will want to modify the code. You will then be able to use the method append to modify the memory without creating new memory each time through the loop. You will merely be expanding the existing memory without leaving behind a mess for the garbage collector.

In the following example, we have used `StringBuffer` to modify the string inside the loop:

```
public class MakeMessage {
    public String getMessage (String[] words) {
        StringBuffer message = new StringBuffer ();

        for (int i = 0; i < words.length; i++) {
            message.append (" ");
           message.append (words [i]);
        }

        return message.toString ();
    }
}
```

Here we have used the method append to add to the buffer. Every time we travel through the loop, we are extending the memory as if we are growing the buffer. This way, we do not force the garbage collector to work harder. Performance-critical situations such as loops call for you to use `StringBuffer`.

Rule: "Use `StringBuffer` instead of `String` for nonconstant strings."

Optimizing Memory

You want to optimize memory in your Java applications. *How do you get rid of chunks of memory that are no longer needed?*

As more and more developers use Java, they are starting to worry about the familiar problem of memory leaks. The memory leak is one of the biggest difficulties in C and C++. In theory, memory leaks are not even possible in Java, but Java developers insist they are running into problems with memory leaks.

Before I discuss methods of optimizing memory in Java, I should clarify the issue of memory leaks. Because of the manner in which Java is constructed, traditional memory leaks are not possible. Every chunk of memory that is not in use can be collected by the garbage collector and cleaned out of the system. Developers who are confounded by so-called memory leaks in Java are probably referring to the fact that if they have memory to which references exist, the garbage collector cannot get rid of the memory. If there are reference pointers pointing to a chunk of memory, that memory remains in the program.

This quirk is the closest thing we have to a memory leak in Java. In other languages, this problem is called *outstanding memory*.

Java cannot distinguish between memory leaks and outstanding memory. If you program in C, you can have a chunk of memory that nobody points to, and there will be no garbage collector to clean it. You can simply get rid of the memory yourself. However, no such memory can remain in Java code; the garbage collector cleans memory to which there are no references.

The problem occurs when there are pointers in the program that refer to the chunk of memory. They are essentially forgotten pointers because the memory is no longer needed. The pointers are useless, and the user did not intend to keep them, but they remain in the program because the user neglected to null them.

We have noticed that the tendency for developers to commit memory errors extends across languages. Developers who tend not to call free or delete on their pointers in C and C++ will have a tendency not to null unnecessary pointer references in Java. Developers who used to leak memory in C and C++ now have lots of outstanding memory issues in their Java programs. Their problems are not related to the language and its features-they are related to the developers who use the language.

In the following example, we have the function `big_splash_image`. We assume it calls new and uses a lot of memory. This function can be a display or whatever else we want. We are sending the memory reference for `big_splash_image` to the function that displays it. After the image is displayed, the memory is really not needed in the program:

```
public class GUI {
    public static void main (String[] args) {
        Graphics big_splash_image = makeSplashImage ();
        displayMomentarily (big_splash_image);

        while (moreUserInput ())
            process ();
    }
}
```

However, you will notice in this example that the reference pointer has not been nulled. Therefore, the Java garbage collector cannot get rid of this memory. In C and C++, you can create the memory in make_splash and then call free when you have used it and want to get rid of it. The equivalent in Java is to assign null to `big_splash_image`.

Therefore, developers who want to avoid leaving unwanted references in their Java programs should zero these references as soon as they no longer need them:

```
public class GUI {
    public static void main (String[] args) {
        Graphics big_splash_image = makeSplashImage ();
        displayMomentarily (big_splash_image);
        big_splash_image = null;
while (moreUserInput ())
            process ();
    }
}
```

Memory issues are very common when developers use temporary variables in Java. When you forget to zero your temporary variables, you will end up with what essentially amounts to a memory leak.

Rule: "Nullify temporary references to objects taking large amounts of memory as soon as they are no longer needed."

Optimizing External Resource Usage

You want to access external resources in your Java program. *How do you avoid leaking resources?*

When you program in Java, you need to be aware that there are many layers of code working underneath you. Though you may be dealing mainly with high-level language performing complicated functions for you, the layers of code underneath that language are performing a host of other functions. These layers will behave differently depending on what you do to the code at the upper level.

The lower layers of your code are vital to the proper functioning of your application; they are the behind-the-scenes workers that make high-level functionality possible. Ignore these hidden layers, and they will most likely come back to cause problems in your application.

One lower layer of your code you cannot afford to ignore is your communication with external resources. There is a central rule that should guide all interactions with external resources. Simply put, if you open an external resource, you should close it as soon as you have finished using it. The following coding example deals specifically with file inputs, which are just one type of external resource. However, the lesson from this example applies to interactions with any external resources. You will notice that this code looks as if it makes a clean exit:

```
public class ReadFile {
    public String read (String path)
        throws IOException
    {
        FileInputStream file = new FileInputStream (path);
        String contents = readFile (file);
        file.close ();
        return contents;
    }
}
```

However, this code is deceptive: The `file.close` command does not make a clean exit. Whether you are talking to a database, opening files, opening sockets, or sending instructions to the screen, you need to close any external resources you have opened. The dangerous aspect of dealing with external resources is that when you write a piece of code such as the aforementioned segment, it seems to run well. However, you may encounter an error when you deal with an external resource.

In such a case, Java (or your code) will throw an exception, which indicates a serious problem. Exceptions can transfer control to different parts of the program. In our example, if the method `readFile` throws an exception, control will be transferred out of the method read and the file will not be closed.

In this situation, you may choose to handle the exception and ignore the problem. However, you should stop to consider that the exception may have come from your interactions with external resources. If you merely handle the exception, you will face the strong possibility of a resource leak. In other words, you will run out of resources at some point, which means that you will not be able to open files or sockets and will not have access to the database.

To put it another way, you have a finite amount of resources. Leaking resources puts a strain on these limited capabilities, causing your program to think that your resources are still in use when, in fact they are not.

If your code throws exceptions when you use external resources, you need to write a finally block. The finally block will always be executed, regardless of whether you exit your code through exceptions or through normal execution. When you use the finally block, you are guaranteed that your code will clean up after you by closing all of your external resources.

Therefore, developers should always exit code cleanly by using the finally block:

```
public class ReadFile {
    public String read (String path)
        throws IOException
    {
        try {
            FileInputStream file = null;
            String contents = readFile (file);
            return contents;

        } finally {
            if(file != null)
            file.close ();
        }
    }
}
```

This segment of code is an example of how to clean up in the `finally` block rather than in the original code. We inserted the `finally` block just before the `file.close` command. We now know that we will be able to close the external resources and exit the code. We will also be able to open the external resources next time we need to use them. Using the `finally` block guarantees that you will not leak resources.

Rule: "Write a `finally` block to clean up when you are dealing with external resources."

Error-Prone Constructs

You are writing code in Java. *How can you help make the code clear and readable?*

Code written in any language should be clear and readable. People should be able to read the code and determine its function at a glance.

One way of making code clear to humans is to follow a standard pattern of indentation. The indentation helps people figure out how certain parts of the code correspond to other parts of the code. The segment of code below is an example of well-indented code:

```
public class DanglingElse {
    public int map (int i) {
        if (i > 0)
            if (i > 10)
                return 10;
        else
            return -1;
    return 0;
    }
}
```

However, this code has another clarity problem - it does not make proper use of braces. Braces are the curly scoping marks that give the compiler (and people) some very necessary information about parts of the code that correspond to specific other parts.

The developer who wrote the previous code intended for it to return -1 whenever i is less than or equal to 0. The indentation in the code suggests that the desired outcome will indeed occur. Because the first if is indented at the same level as else, it appears that the code will return -1 in any case where i is not greater than 0.

However, this code will actually return -1 whenever i is smaller than or equal to 10. Braces instruct the compiler to associate the statements differently. When you write a segment of code that contains several if statements, you need to use braces to make sure that the compiler associates statements the way you intended. When you fail to use braces with else statements, you are said to have created a dangling else statement.

Therefore, you should use braces to avoid *dangling* `else` statements:

```
public class DanglingElse {
    public int map (int i) {
        if (i > 0) {
            if (i > 10)
                return 10;
        } else {
            return -1;
        }
    return 0;
    }
}
```

Braces eliminate problems because they make your code explicit. Whenever you have else statements, use brackets to specify the proper order of operations.

Rule: "Avoid dangling `else` statements."

Additional Examples of Error-Prone Constructs

Example 1. Rules that concern proper use of marks such as braces or parentheses may seem trivial, but they actually apply to many important areas of your code. For example, the way you use parentheses affects the order of evaluation for statements. In much the same way that you follow an order of operations when you solve algebraic equations, the Java compiler will execute first any statements enclosed in parentheses. Whenever you are unsure of your order of execution, you should mark with parentheses any statements that you know you want the compiler to execute first. The following example illustrates the dangers of incorrectly-used parentheses:

```
public class PT_DCP {
    public static void main (String args []) {
        System.err.println ("2 + 9 = " + 2 + 9);
    }
}
```

This code was supposed to print "11". Much to the developer's surprise, however, it will instead print "29" as the answer. The outcome of the program will be drastically wrong because the developer neglected to use parentheses properly.

If the developer had enclosed the second 2 + 9 in parentheses, that section of the code would have been interpreted as an integer operation, which was the desired effect. 2 + 9 would have been a simple integer addition. Furthermore, 2 + 9 would have been calculated first before the rest of the computations were performed. Without the parentheses, however, 2 + 9 = is a string. The + that follows the 2 + 9 = is treated as a string concatenation. The 2 and the 9 are interpreted as two separate strings, and they are printed side by side as "29", providing a wildly inaccurate answer to a simple computation.

Rule: "Use parentheses to clear up ambiguities."

Example 2. Many developers do not monitor the contents of their `if` statements. Errors in `if` statements are easy to overlook when you are checking your code. The following example illustrates the possible dangers of `if` statements:

```
public class PT_ASI {
    void method (boolean val) {
        if (val = true) {
            k += 1;
        }
    }

    private int k = 10;
}
```

The developer who wrote this code was trying to compare `val` to `true`, but the code has been written incorrectly. Because there is only one = between `val` and `true`, the statement inside if is an assignment statement. When this code is executed, k will always be added regardless of what the value is.

In checking over the code, the developer failed to notice the missing = between `val` and `true`. To make a comparison, you need to use == rather than =. This error is an easy one to catch automatically, but it causes major problems when it goes undetected.

Rule: "Watch out for errors inside `if` statements."

Example 3. Below is another segment of code that contains an ambiguity:

```
class AClass {
    static void getIt () {
    }
}

public class PT_AUO {
    void method () {
        AClass object = new AClass ();
        object.getIt ();
        AClass.getIt ();
    }
}
```

You will notice that because we are dealing with the static method `getIt` in `AClass`, we should not call it through the object `(object.getIt)`. Calling the method through the object would falsely indicate that we are dealing with a method that belongs to a class that is not really static.

The correct way to call this method is by using `AClass`, which provides us with the type of class rather than just the object that is generated out of the class. Calling with `object.getIt` would be correct if the method was not static, but because the method is static, we should call it with `AClass.getIt`. As it is written now, the code is confusing because when the developer reads it and sees `object.getIt`, he/she will expect to be dealing with a member function that is not static. However, if you read `AClass.getIt`, you will know that you are dealing with a static function. We need to call this static function using `AClass.getIt`.

Rule: "Call static functions through class references."

Example 4. Java developers are usually eager to optimize their code. Optimization is useful in many cases, but failed attempts at optimization can actually be detrimental to the code's performance:

```
public class PT_NEA {
    void method () {
        int i = 2;
        int j = 2;
        short r = 4;
        double d = 2;
        double x = 3;
        int k = 3;

        d = (k = i + j) + r;
        d -= (x = i + j) + r;
        d /= (x /= i + j) + r;
    }
}
```

At the bottom of this segment of code are three lines, each of which contains three statements. The developer wrote the code this way in an attempt to optimize the code's performance. However, the code has merely ended up being messy, confusing, and error-prone.

To enhance the code's readability, the line of code
`d = (k = i + j) + r;`
should be split up in this manner:
```
k = i + j;
d = k + r;
```

The code is less confusing when it is written this way. Ironically enough, writing the code the way it appears in the coding example does not help optimize it. There is a common misconception among developers that the code will perform faster when it is written on the same line, but this technique has no effect in terms of speed. The lesson here is that developers should not try to be too smart by optimizing code that doesn't need to be optimized.

Rule: "Do not optimize code that does not need to be optimized."

Example 5. Developers who are unsure about how to use the `default` label often choose to omit the label from their code rather than wrestle with it. The results for the code can be severe:

```
public class PT_PDS
{
    private int a = 0;
    void method (int i) {
        switch (i) {
        case 1:
            a = 10;
            break;
        case 2:
        case 3:
            a = 20;
            return;
        }  // missing default label
    }
}
```

As you can see, if you present an unexpected value to the code, the value will pass right through the code. It will not matter whether the value is right or wrong.

To avoid passing values through the code without monitoring the results, you must provide a `default` label for every switch statement. If you are unsure of what to use as a `default` label, simply write:

```
default:
throw new IllegalArgumentException ();
```

Rule: "Provide a `default` label for each case statement."

Example 6. The following segment of code presents an example that is similar to the previous case:

```
public class PT_SBC
{
    private int a = 0;
    void method (int i) {
        switch (i) {
            case 1:
                a = 10;
            case 2: case 3:
                a = 20;
            default:
                a = 40;
        }
    }
}
```

The developer in this case has made an omission. There should be a `break;` after each case, as displayed below:

```
case 1:
    a = 10;
    break;
```

Without these `break;` statements, the code will fall right through the case statements. It will not matter which case you are in; you will automatically fall right through to the `default`.

Fixing this problem is as simple as making sure that each case statement is followed by `break;`.

Rule: "Follow case statements with `break`."

Example 7. Below is another example of how a simple spacing error can destroy the functionality of your code:

```
public class PT_TLS {
    static int method (int i) {
        switch (i) {
        case 4:
        case3:
            i++;
            break;
        case 25:
        wronglabel:
            break;
        default:
        }
        return i;
    }

    public static void main (String args[]) {
        int i = method (3);
        System.out.println (i);
    }
}
```

As you can see, the developer intended to write `case 3` but instead wrote `case3`. Because of this simple typographical error, `case3` will now become a new `case`. Meanwhile, when i equals 3, the value will not go to `case3`. Instead, `i = 3` will always go to the `default`.

Rule: "Check your code for proper spacing in labels."

Example 8. One of the most common mistakes for developers to make is to compare strings incorrectly:

```
public class PT_UEI {
    int method (String s1, String s2) {
        if (s1 == s2) {
            return s1.length () + s2.length ();
        }
        return 2 * s1.length ();
    }
}
```

In this example, the developer intended to compare strings, but instead ended up comparing references by using `if (s1 == s2)`. The way the code is written now, it merely says "if `s1` as a reference pointer is equal to `s2` as a reference pointer..."

To avoid it, you need to use the `String.compare to` function.

Rule: "Use the `String.compare to` or `String.equals` function to compare strings."

Object-Oriented Programming

You are using object-oriented (OO) programming in your Java code. How do you avoid the pitfalls of OO programming without sacrificing any of the benefits?

OO programming makes code reusable, easily maintainable and better organized. However, there are a number of pitfalls, for example:

```
public class BankAccount {
    public int _balance;
}
```

The class `BankAccount` is used to represent a bank account, but the variable used to represent the balance has been made public. Even though declaring the variable `public` is legal according to the Java language, it makes the code very difficult to modify and improve. There is a safer way of writing the code and achieving the same effect:

```
public class BankAccount {
    public int getBalance () {
        return _balance;
    }
    public void setBalance (int balance) {
        _balance = balance;
    }
    private int _balance;
}
```

Here, the `_balance` variable has been declared private and public methods have been defined to access it. The code is now very easy to maintain because you can change the `BankAccount` implementation without having to change any of the client code.

For example, you can make the `BankAccount` object thread-safe just by declaring the `getBalance` and `setBalance` methods synchronized. Note that none of the other methods that may be using `BankAccount` objects need to be modified in this case.

Therefore, you should declare your variables private.

Rule: "Avoid public and package instance variables."

Conclusion

If you think that the coding standards we have illustrated are not worth enforcing, you may stop reading now. Otherwise, please continue.

The rules discussed are examples of coding standards. There is a common misconception among developers that using coding standards will help them find errors in their code. In fact, coding standards do not find errors; they are designed to prevent errors, helping developers create better code from the beginning of a project. In general, coding standards are rules that protect programmers from writing dangerous, inefficient, error-prone code.

Therefore, using coding standards is the best alternative to debugging. If you implement coding standards effectively, you will avoid many bugs, produce better code, and save much valuable time.

The key to being successful with coding standards is twofold:

1. Apply coding standards consistently through the development process.
2. Apply coding standards across the development team.

Coding standards have very little benefit when one member of a development team consistently breaks them; other team members are then forced to find and fix the errors that this careless developer introduced to the code.

To enforce coding standards, a team should sit down and decide which rules are important to them, and then start enforcing them across the entire team. After some period of time - approximately a month or two - the team should review the effects of using coding standards, then decide which standards to add to the enforced set and which ones to remove. As you can see, communication and flexibility are critical to effective usage of coding standards.

We have found approximately 150 coding standards for Java. We also have found that the typical development team enforces about half of these standards. How many do you enforce? Do you spend a significant portion of your development cycle chasing down bugs? Maybe it is time to think about this issue!