2005

Teaching Java to IS Students: Top Ten Most Heinous Programming Errors

Mark Pendergast
Florida Gulf Coast University, mpenderg@fgcu.edu

Teaching Java to IS Students: Top Ten Most Heinous Programming Errors

Mark Pendergast
Florida Gulf Coast University
Fort Myers, FL, USA
mpenderg@fgcu.edu

ABSTRACT

Learning to write computer programs is a difficult process. This short experience paper details ten common, yet difficult to spot errors made by students in my introductory Java programming classes. The errors detailed in this article are considered "heinous" because they either don't occur every time, are hard to spot when desk checking code, are due to an inconsistency in the Java language itself, or all of the above. Along with the errors this article describes teaching techniques to help students avoid them and suggests potential language modifications to preclude them.

Keywords
Java, Information Systems, Education, Introductory Programming Course.

INTRODUCTION

Picture, if you dare, being stuck in an office for hours each week with a line of dazed and desperate Java students waiting outside your door. Each desires to get "just a couple of minutes" of your time to help them past their current problem in the often-futile attempt to complete their assignment on time. If you are a recent college grad you may remember standing in such a line, or if you are the resident office Java guru, you may have such a line outside your door right now. You may also remember certain problems you spent all night staring at only to realize it was a simple matter of a misplaced semi-colon or an inconsistency in the way standard Java objects work. The importance of a properly constructed first course in programming cannot be overstated. A well taught course will leave students with good programming habits, the ability to learn on their own, and a favorable opinion of programming as a profession. A poor experience may result in a "just get by" attitude, bad programming habits, and could lead to a change in majors.

During CS/IS boom in the 1990's the loss of students to other disciplines was not a major concern. These students were just written off as not having an "aptitude" for programming. This philosophy will change due to shrinking enrollments in CS/IS fields (Zweben, 2004). (Hensel, 1998) foretold of shrinking enrollment, predicting that enrollment would drop due to a decline in college age students, poor preparation for technical classes (leading to failures and reluctance to enroll), and changing interests in students. This added to fears of outsourcing, the dot com bust, and the hiring slow down after Y2K conversions have kept students away from CS/IS. CS/IS faculty members have little control of national birthrates and the hiring practices of corporations, but they can look for ways to attract more students and retain the ones they now have.

This article starts with a little background on Information Systems programs then details ten such conceptual problems routinely made by my students and occasionally made myself. Along with the problems I propose coping mechanism and potential compiler enhancements.

EVOLUTION OF INFORMATION SYSTEMS CURRICULUM

Teaching programming courses to Information Systems students in the College of Business has much in common with teaching Computer Science students but there are some differences in student background and prerequisite courses. Unlike CS students, IS students are generally only required to have a semester of statistics and a semester of "business calculus" to graduate. More often than not, these courses are put off to their senior years. Therefore the mathematical background of IS students consists of what the may or may not have learned in high school and what they may or may not remember of it. In addition, a typical IS major only takes 20-30 credits of computer related courses whereas the CS major is required by ABET

standards to have 42 hours (or more). This leaves IS students with only 2 or 3 courses whose primary purpose is to teach programming fundamentals.

As late as the 1990, COBOL was the programming language of choice for IS students due to its popularity in business applications. In the early 1990s IS programs began using PASCAL (in addition to COBOL) so that their students would have experience with a procedural language. This trend was reinforced by the introduction of personal computers and Borland's Turbo Pascal on campuses. The combination of Microsoft's introduction of Visual Basic and the advent of the world-wide-web caused many IS programs to abandon Pascal in favor of Visual Basic or Java. C and C++ are also used by many IS programs, especially those that depend on the Computer Science department to provide programming courses. (Culwin, 1999) (Noll, Wilkins, 2002) noted the increased need for developing programming skills for web-based languages (Perl, JavaSript, Java, SQL) over traditional application programming languages (COBOL, C). See (AIS, 2002) for more information on IS curriculum standards.

On our campus we have proponents of all three languages, VB, Java, and C++. We chose to teach Java because of its dominance in web applications, it is less complex and easier to debug than C++, and it has a syntax based on C/C++, making it easier for students to transition to those languages once they graduate. (Tyma, 1998) cited these reasons along with platform independence, higher programmer productivity, and better systems stability as reasons for moving to Java. More recently, NASA chose Java as its programming language for the Mars rover missions (Sun 2004) due to programmer productivity. Java has long been denigrated for its slow performance due to the use of a virtual machine to execute programs however the perceived performance difference may not be as great as some people claim. (Mangione, 1998) found little or no difference for most tests, and (Lewis, 2003) found Java to be as fast or faster than C++.

JAVA'S STRENGTHS AND WEAKNESSES

Teaching object-oriented programming and object-oriented languages have come a long way since (Decker, Hirshfield 1994) enumerated ten reasons why CS/IS programs were avoiding teaching OO programming in their introductory programming language. Chief among these were a resistance to change and the idea OO programming was "too hard." At that time OO programming languages were evolving and coming onto favor and going out of favor at a high rate. C++, SmallTalk, Lisp, EIFFEL, SATHER were all vying for the hearts and minds of programmers. (Kolling, etal 1995) concluded that none of these languages could be considered as appropriate for use by beginning programmers. He went on to propose ten requirements for an object-oriented language that is to be used in a first year programming course. See table 1.

Kolling Requirement	**Java Support**
1. The language should support clean, simple, and well-defined concepts. This applies especially to the type system, which will have a major influence on the structure of the language. The basic concepts of object-oriented programming, such as information hiding, inheritance, type parameterization and dynamic dispatch, should be supported in a consistent and easily understandable manner.	No, since Java inherited much of its OO nature from C++, it inherited many of its pitfalls. Java did replace multiple inheritance with single inheritance augmented with a new construct called "interfaces".
2. The language should exhibit "pure" object-orientation in the sense that object-oriented constructs are not an additional option amongst other possible structures, but are the basic abstraction used in programming.	Yes, unlike C++, all code within a Java program must reside in an object. GUI and file IO are controlled by library object. However, this doesn't preclude someone from creating a poorly structured program.
3. It should avoid concepts that are likely to result in erroneous programs. In particular it should have a safe, statically checked (as far as possible) type system, no	Yes, unlike scripting languages, Java has a tightly typed system and the compiler automatically detects uninitialized variables. Java has no explicit pointers or use of indirection.

explicit pointers and no undetectable uninitialized variables.	It does have reference variables.
4. The language should not include constructs that concern machine internals and have no semantic value. This includes, most importantly, dynamic storage allocation. As a result the system must provide automatic garbage collection.	Yes, Java is platform independent and does not contain constructs that concern machine internals (e.g. register variables). Java does provide automatic garbage collection, though this has proven to hurt performance.
5. It should have a well-defined, easily understandable execution model.	No, in the case of over-loaded and overridden methods it is not always clear to beginning students which method will be invoked. Also the use of static methods and variables complicates understanding where data resides and how it can be accessed.
6. The language should have an easily readable, consistent syntax. Consistency and understandability are enhanced by ensuring that the same syntax is used for semantically similar constructs and that different syntax is used for other constructs.	No, again, due to its parentage with C and C++, there are multiple constructs for performing simple tasks, e.g. there are at least three ways to increment a variable by one: i = i+1; i++; and i+= 1;
7. The language itself should be small, clear and avoid redundancy in language constructs.	No, not only does Java inherit a large number of constructs from C++, it adds a large library of system objects that must be learned as well.
8. It should, as far as possible, ease the transition to other widely used languages, such as C.	Yes, Java shares many language constructs with both C and C++.
9. It should provide support for correctness assurance, such as assertions, debug instructions, and pre and post conditions.	Yes, to some degree. Java does provide an exception handling mechanism to assure proper execution and provides automatic testing for array bounds, null pointers, and arithmetic exceptions. However, consistency is a problem, in the case of arithmetic exceptions a divide by 0 in floating point operations results in a value of infinity, while in integer arithmetic the result is a divide by zero exception.
10. Finally, the language should have an easy-to-use development environment, including a debugger, so that the students can concentrate on learning programming concepts rather than the environment itself.	Yes, many mature IDEs are available, including Borland's JBuilder, Sun Microsystems Sun One Studio, Eclipse, and NetBeans, and other open source IDE. The main drawback to these environments is their cryptic compiler error messages and clumsy form editors.

Table 1 – Java's Compliance with Kolling's 10 Criteria

(Kolling, 1995) observations from 1995 are as relevant today as there were when he made them. As evidence of this the next section presents what I consider to be the most difficult errors for students to detect, correct, and comprehend. These errors are included because they either don't occur every time, are hard to spot when desk checking code, are due to an inconsistency in the Java language itself, or all of the above.

HEINOUS ERRORS

Extraneous Semicolons

Beginning students tend to take anything a professor or a textbook says literally. When their textbook (Liang, 2003) states "Every statement in Java ends with a semicolon (;)", then that's what they do. IDE's such as Netbeans that identify syntax errors as the students type the code can contribute to the programs. For example, when a student types in the first line of a

for statement, Netbeans will mark it as an error until the body of the for loop is added. The following code snippet includes 5 such cases:

```
public class SemiClass ; // 1
{
  public SemiClass(int parameter); //2
 {
    int i = 0, j = 0, x = 0;
    if(i < 9); //3
       j = 1;
    else; //4
       j = 2;
   for(x = 0; x < 100; x++); //5
   {
     i = i +x;
     return;
    }
   }
}
```

This code generates 5 syntax errors when compiled by the J2SDK, but only one of the erroneous semicolons is flagged. Extra semicolons on the end of if, while, and for statements will not generate compilation errors. Instead the code will compile, but not execute as intended since the extra ";" ends the statement. As a result, a student's for loop might execute a do-nothing statement 100 times, and their intended body of the loop only once!

Helping overcome this problem can be achieved by having textbooks more precisely define what a statement is, using lots of examples during class presentation, and by having a compiler that supports warning messages for control structures with do-nothing bodies.

Integer Arithmetic Errors

These occur most frequently when an equation that yields a double result has an intermediate calculation that contains one integer being divided by another. Not only is this type of error hard to spot while desk checking it is also difficult to explain the rationale behind the standard. Students can readily understand that results must be truncated when they are stored into integer variables, but have a harder time with the concept of truncating numbers that are to be stored into double or floats. Example:

```
int x = 1, y = 2;
double z = x/y + 44.0;
System.out.println("z = "+z);
```

In this case z is given a value of 44.0 instead of 44.5. The best advice to give to students is to double-check every division for divide by zero errors AND integer truncation. Changing the operation of a compiler to use double precision math for all intermediate operations would simplify the teaching of the language, but may generate unintended errors in existing programs. A compiler generated warning message is more appropriate.

Zero Relative and One Relative Inconsistencies

Java, like C and C++, uses zero relative indexing for arrays. Most Java library classes also use zero relative indexes for retrieving items (e.g. Strings, Vectors, JComboBox, JList, JTable). Exceptions to this rule are for accessing fields in Java SQL objects. In particular the PreparedStatement and ResultSet objects are one-relative. Students programs that use one-relative indexing for arrays and container objects generally result in IndexOutOfBoundExceptions being thrown. SQLExceptions are thrown for using zero-relative indexing with PreparedStatements and ResultSet object produce SQL Exceptions. These exceptions are thrown only if the programs try to access every element, if not, then the error can go unnoticed.

Dealing with this inconsistency requires diligence by the students while coding and debugging. Having a consistent standard for the creation of standard language objects would help.

Not Recompiling all Dependent Classes When static final Variables Change

In Java, constants are declared with the *final* modifier. Constants that are universally useful, e.g. PI, are declared in a class with the *static* and *public* modifiers as well. E.g.

```
 public static final double PI = 3.141592653589793;
```

This makes the constants accessible by methods in other classes without having to use accessor methods. The problem arises when constants defined this way are changed and their class files are recompiled but classes that use them are not recompiled. This is because the values for final variables are treated by the Java compiler as literals and are set in the class code at compile time. Example, if class Other is initially defined and compiled as:

```
public class Other {
  public static final int C1 = 1;
  public static int c2  = 22;
}
and class Main is defined and compiled as:

public class Main {
 public static void main(String args[])
 {
System.out.println(Other.C1+","+Other.c2);
 }
}
```

The output is “1,22” as one would expect. However, if class Other is later modified and compiled as the following:

```
public class Other {
  public static final int C1 = 3;
  public static int c2  = 33;
}
```

When class Main is run without recompiling, then the output is “1,33”, not “3, 33” as you would expect. This illustrates that the value for the final C1 variable was accessed at compile time and the non final variable c2 at run time. Many IDEs

don't automatically check for dependency changes of this type (some don't check for changes at all), and the compiler does not generate errors or warnings. The only sure solution to give to students to resolve this issue is to tell them to either avoid using the "final" keyword, or to use the "build-all" function of their IDE to force recompilation of all classes. A build-all is also necessary if the students update their Java class libraries as some constants may have changed since the last release.

Operator Precedence Problems

One exam question I recently gave to my System Design class required them to calculate the lines of code for a system using some of the classical empirical estimation models (Cocomo, Bailey-Basili, Boehm (Pressman, 2001)). Completing the exam question only required the ability to plug values into an equation and generate the result. I was surprised to discover that over half of the class could not do this correctly for an equation as simple as:

$$E = 5.5 + .73K^{1.16}$$

Not only did they not know that the value of K should be raised to the 1.16 power before multiply by .73, many did not know that the addition should be done last. They assumed everything should be done in a strict left to right sequence! While this particular class is not typical of all students at my university or all IS students, the problems they experienced pose a further teaching challenge. Correctly converting mathematical equations to Java code not only requires the understanding of Java precedence rules, but precedence rules used in the equations themselves. Business programming requires very accurate calculations of accrued interest, present values, future values, depreciation, etc. Accurate answers cannot be stressed enough.

Most, if not all authors of Java programming books include tables showing operator precedence order. Getting the students to learn how to correctly interpret the tables is a first step. A second step is to require extensive testing with problem sets that have known answers.

Variable Scope Errors

In Java, as in C++, variable may be declared in the class definition, as parameters in methods, and as local variables in methods. Java allows local variables to be given the same name as those defined in the class. Furthermore, a method in Java may declare multiple variables with the same name as long as they occur in different code blocks. Java code resolves ambiguous references by using the variable declared in the block of code.

Another confusing aspect of variable scope is with variables defined in do-while blocks and in the header of for loops. In the example below, even though the while part of the do-while is considered part of the statement, the condition in the while cannot access variables defined in the while block. However, variables defined in the header of a for-loop can be accessed by code in the block.

```
do{
    int x = 2;
    cost = cost + x;
    x = x-1;
  }while(x > 0);  // compile error here

  for(int i = 0; i < 3; i++)
  {
   int x = x+i;
```

```
}
```

Variable scope problems can be overcome to some extent by using proper Java variable naming conventions. Under these conventions, local variables are defined with all lower case letters (e.g. productname) and class variables are declared with mixed case using lower case for the first word in the name and upper case for the first character of subsequent words (e.g productName). This is just a partial solution because it differentiates only variable names that are more than one word long, and because the naming convention is not enforced by the compiler (and would be somewhat difficult to do so). A better solution would be to not allow local variable and method parameters to be given the same name as class variables.

For now, I have adopted the strategy of telling the students to never name a local variable the same as a class variable, and to declare all their variables at the beginning of the methods. This helps them spot naming conflicts, avoids ambiguous definitions, and makes the code easier to read.

Casting and Type Checking

This is as much a conceptual problem as it is a programming problem. Many of my students have learned web-scripting languages, such as PHP, that do not require the declaration of variables and that do no type checking. They see this as a faster, easier, and therefore better approach to programming. Those of us with experience managing and maintaining systems have learned "the hard way" the value of languages that have tight type checking. Tight type checking reduces coding errors in both the creation and maintenance stages of a program's lifecycle. Students without scripting experience do not have these preconceived notions. In either case, student must learn to understand type checking and casting in order to create Java programs.

Java's type checking makes perfect sense, once you understand it. Basically, Java will allow you to store values in variables without complaint as long as there is no loss of precision. Integers can be stored in longs, floats in doubles, integers into doubles, etc. However, loss of precision compiler errors will result if doubles are stored into floats or integers. When this is explained my students nod their heads indicating understanding, but still write code like the below:

```
double d = 3.0;
int x = d+1;
```

They assume that since d does not have a decimal part, it should be able to be used with an integer. Since most of the functions in the Java Math object return doubles, the need to store doubles in integers comes up a lot. One solution that often appears in books is to cast the result as an integer.

```
int x = (int)Math.sqrt(143);
```

This has the effect of converting (and truncating) the result into an integer. So long as students realize they are truncating the decimal part of the answer and not rounding it, then their program will produce the intended results. However, often times they forget, or they assume that it won't make a difference in their final result. Therefore I also teach them the use the Math.round function. Also the students need to learn that casting can be used to convert real number to integers, but casting will not convert Strings or characters to numbers. For example:

```
char c = '5';
int x = (int)c;
```

x will be set to the ascii code for the character 5 (53), not 5.

Casting Objects

Beyond numeric type conversions, casting is a necessity when working with object streams, Java data structure, and container objects such as Vectors, JTables, JComboBoxes, JLists. These objects maintain lists of other objects. An object of any type can be inserted into them without compiler complaint since the "add" methods accept objects of type "Object" (Object is the root type of all Java objects). Java allows objects of a subtype to be stored in a supertype without casting. The reverse is not true. Supertype objects cannot be stored into subtype variables without casting. Since the "get" methods associated with container objects return a value of type Object, casting is mandatory. The conceptual problem that students have with casting objects is they assume they are converting data as they did when casting primitive data types. Therefore they make mistakes like the following:

```
JTextField nameInput = new JTextField();
…
String s = (String)nameInput;
```

When they define the object nameInput they are creating a Java interface object that allows the input of string data. However, simply casting the object to a string does not retrieve its data. The example above will generate a compilation error since String is not a subtype of JTextField. This error can be explained and fixed. Casting of objects as they are retrieved from Vectors or other container/collection objects can produce run time errors that are less than obvious. In the following code a String object and an Employee object are stored into a Vector, they are both retrieved into Strings. The code compiles correctly but will generate a runtime error (ClassCastException) on the second call to "get".

```
Vector v = new Vector();
String a = "A String";
Employee e = new Employee();

v.add(a);
v.add(e);

String s1=(String)v.get(0);
String s2=(String)v.get(1);//runtime error!
```

Avoiding errors of this sort is a matter of keeping track of the order and type of objects added to the Vector, and/or, by properly using the *instanceof* operator when retrieving items. The best advice to give to beginning students is to tell them to only add objects of one type to collections and containers.

Handling NaN and infinity Values

Next to grading, explaining language inconsistencies is my least favorite teaching task. This is particularly true of Java's handling of arithmetic operations errors. Java will generate an ArithmeticException when a divide by zero occurs for integer division, but not for double division. Instead, Java assigns the value of infinity to the double result. Taking a square root of a negative number does not produce an exception, instead, it assigns a value of NaN (not a number) to the result. The following example illustrates both cases.

```
double y = Math.sqrt(-1);
double x = 5.0/0.0;
System.out.println("x = "+x+"  y = "+y);
```

```
int z = 1/0;
System.out.println("z = "+z);
```

produces:

```
x = Infinity  y = NaN
Exception in thread "main" java.lang.ArithmeticException: / by zero  at Main.main(Main.java:12)
```

To overcome these errors students must be instructed to carefully examine every equation, and to not rely on catching ArithmeticExceptions. Also, code must be added to test for results that produce infinity or NaN. The handling of divide by zero errors by the Java engine should be consistent, either always producing an exception or always result in infinity.

Accidental Octal Literals

I've yet to encounter a good use for octal numbers in business applications but they are useful for some computer science applications and therefore they are supported most by programming languages, including Java. Java uses a leading 0 to distinguish an octal literal from a decimal literal, 0123 is an octal number (decimal 83), 123 decimal. This is fine until you start storing such things as product id numbers, phone numbers, and other values that have (any may require) leading 0s. Most of the time these values are read in from databases or screens and the leading 0 is just a matter of formatting for user display on reports, literals are not necessary. Other times the issue can be avoided completely by storing such numbers as Strings instead of ints and longs. However, the rare circumstance does come up where students use literals to enter such numbers into their programs. E.g.

```
int internationalDialing = 011;
long productID = 01112222;
```

These mistakes are difficult to spot. It would make life easier if the Java designers had chosen a prefix for octal numbers that make them stand out. For example, since 0x is used to denote hexadecimal number, 0c could have been used for octal numbers.

CONCLUSION

The previous section detailed ten of what I consider to be the hardest conceptual and programming errors that my students make. Some are due to the complexity of modern object-oriented programming languages, others due to inconsistencies in the language itself, and others arise because of the backgrounds of the students (and instructor) themselves. These problems illustrate that seemingly simple concepts can become very difficult to program if students do not have the proper level of comprehension of the syntax, structure, and sometimes-arbitrary/inconsistent rules of a programming language. An updated version of the Java language, know to some as Java 1.5 and to others as simply Java 5, makes certain constructs easier to handle, for example "boxing" primitive objects, enumerations, and generic typing. While this does make some code more readable it does little to alleviate the overall complexity and consistency of the language.

(White 2002) stipulates that both procedural and object-oriented programming require cognitive skills at the "formal operations" level. Students must possess the capability to deal with abstractions, solve problems systematically, and engage in mental manipulations. Once their level of cognition is exceeded then "burnout" insures (or as my students would say, their brains are fried). Students experiencing burnout are less likely to continue in a CS/IS program. Therefore it is imperative to create strategies to present information in a manner that will allow average students to succeed without experience a level of frustration resulting in "burnout." My intent is to bring about thought and discussion of education and learning techniques, supply suggestions for textbook authors, and to detail possible changes to the Java SDK.

REFERENCES

1 Association for Information Systems, IS 2002 Standards, http://www.aisnet.org/Curriculum/IS2002-12-31.pdf, 2002.

2 Culwin, F. (1999), "Object Imperatives!", ACM SIGCSE Bulletin , The Proceedings Of The Thirtieth SIGCSE Technical Symposium On Computer Science Education, 31, 1, 31-36.

3 Decker, R., Hirshfield, S. (1994), "The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught in CS 1", ACM SIGCSE Bulletin , Selected Papers Of The Twenty-Fifth Annual SIGCSE Symposium On Computer Science Education, 26, 1, 51-55.

4 Hensel, M. (1998) "The Approaching Crisis in Computing Education: Enrollment, Technology, Curricula, Standards, and their Impact on Educational Institutions", Journal of Information Systems Education, 9,4, 24-27.

5 Kölling, M., Koch, B., Rosenberg, J. (1995) "Requirements for a first year object-oriented teaching language", ACM SIGCSE Bulletin, Papers of the 26th SIGCSE technical symposium on Computer science education, 27, 1.

6 Lewis, J.P., Neuman, U.,(2003), "Performance of Java versus C++", from http://www.idiom.com/~zilla/Computer/javaCbenchmark.html.

7 Liang, Y.D., *Introduction to Java Programming with Sun One Studio 4*, Prentice Hall, 2003.

8 Mangione, C., (1998) "Performance tests show Java as fast as C++", JavaWorld, from http://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf_p.html.

9 Noll, C. L., Wilkins, M.,(2002), "Critical Skills of IS Professionals: A Model for Curriculum Development", Journal of Information Technology, 1, 3, 143-154.

10 Pressman, R.S. *Software Engineering: A Practitioner's Approach*, 5th Edition, MCGraw-Hill, 2001.

11 Sun MicroSystems, (2004), "The Mars Mission Continues", http://www.sun.com/aboutsun/media/features/mars.html, January, 2004.

12 Tyma, P. (1998), "Why We are Using Java, Again?", Communications of the ACM, 41, 6, 38-42.

13 White, G., Sivitanides, M., (2002), "A Theory of the Relationships between Cognitive Requirements of Computer Programming Languages and Programmers' Cognitive Characteristics", Journal of Information Systems Education, 13, 1, 60-66.

14 Zweben, S. , Aspray. W.,(2004) "Undergraduate Enrollments Drop; Department Growth Expectations Moderate", 2002-2003 Taulbee Survey, Computing Research Association. Retrieved from http://www.cra.org/CRN/articles/may04/taulbee.html.