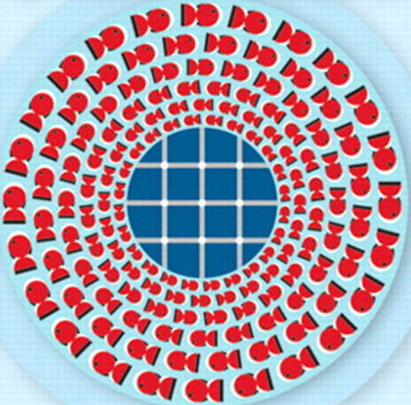
Traps, Pitfalls, and Corner Cases

Java™ Puzzlers



Joshua Bloch Neal Gafter

Java™ Puzzlers

This page intentionally left blank

Java™ Puzzlers

Traps, Pitfalls, and Corner Cases

Joshua Bloch
Neal Gafter

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data
Bloch, Joshua.
Java puzzlers : traps, pitfalls, and corner cases.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-33678-X (pbk. : alk. paper)
1. Java (Computer program language) I. Gafter, Neal. II. Title.

QA76.73.J38B58 2005
005.13'3—dc22
2005015278

ISBN 0-321-33678-X
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
Sixth printing, January 2009

To the memory of our fathers:
Fritz W. Bloch (May 2, 1911–May 24, 2003)
Benjamin Abraham Gafter (June 15, 1923–December 14, 2003)

This page intentionally left blank

Contents

Preface

Like many books, this one had a long gestation period. We've collected Java puzzles for as long as we've worked with the platform: since mid-1996, in case you're curious. In early 2001, we came up with the idea of doing a talk consisting entirely of Java puzzles. We pitched the idea to Larry Jacobs, then at Oracle, and he bought it hook, line, and sinker.

We gave the first "Java Puzzlers" talk at the Oracle Open World conference in San Francisco in November 2001. To add a bit of pizazz, we introduced ourselves as "Click and Hack, the Type-it Brothers" and stole a bunch of jokes from Tom and Ray Magliozzi of *Car Talk* fame. The presentation was voted best-in-show, and probably would have been even if we hadn't voted for ourselves. We knew we were on to something.

Dressed in spiffy blue mechanic's overalls emblazoned with the "cup and steam" Java logo, we recycled the Oracle talk at JavaOne 2002 to rave reviews—at least from our friends. In the years that followed, we came up with three more "Java Puzzlers" talks and presented them at countless conferences, corporations, and colleges in cities around the globe, from Oslo to Tokyo. The talks were almost universally well liked, and we got very little fruit thrown at us. In the March 2003 issue of *Linux Magazine*, we published an article consisting entirely of Java puzzles and received almost no hate mail. This book contains nearly all the puzzles from our talks and articles and many, many more.

Although this book draws attention to the traps and pitfalls of the Java platform, we do not mean to denigrate it in any way. It is because we love the Java platform that we've devoted nearly a decade of our professional lives to it. Every platform with enough power to do real work has some problems, and Java has far fewer than most. The better you understand the problems, the less likely you are to get hurt by them, and that's where this book comes in.

Most of the puzzles in the book focus on short programs that appear to do one thing but actually do something else. That's why we've chosen to decorate the book with optical illusions—drawings that appear to be one thing but are actually another. Also, you can stare at them while you're trying to figure out what in the world the programs do.

Above all, we wanted this book to be fun. We sincerely hope that you enjoy solving the puzzles as much as we enjoyed writing them and that you learn as much from them as we did.

And by all means, send us your puzzlers! If you have a puzzle that you think belongs in a future edition of this book, write it on the back of a $20 bill and send it to us, or e-mail it to puzzlers@javapuzzlers.com. If we use your puzzle, we'll give you credit.

Last but not least, don't code like my brother.

ACKNOWLEDGMENTS

We thank the whole team at Addison-Wesley for their kindness and professionalism. Early in the life of this project, Ann Sellers was our editor. Her infectious enthusiasm helped get the project off to a good start. When Ann moved on, Greg Doench, executive editor, took over. Greg is a wonderful editor and a perfect gentleman. He accommodated the many demands of this project without batting an eyelash. Greg's editorial assistant is Noreen Regina. Our project editor is Tyrrell Albaugh and our marketing manager is Stephane Nakib. Our cover designer is Chuti Prasertsith and our copy editor is Evelyn Pyle. They all did great work under a tight schedule.

We thank our management at Google for their support. Our director, Prabha Krishna, was unfailingly encouraging. We thank Sergey Brin, Larry Page, and Eric Schmidt for creating the best engineering environment on the planet.

We thank the many Java programmers who submitted bug reports to Sun over the years, especially those who submitted bug reports that turned out not to describe real bugs. Such bug reports were perhaps the richest source of puzzler material: If correct behavior misled programmers into thinking that they had dis-

covered a bug, it probably represents a trap or pitfall. In a similar vein, we thank Sun for having the courage and wisdom to put the entire Java bug database on the Web in 1996 [Bug]. This action was unheard of at the time; even today it is rare.

“Send us your puzzlers,” we said at the end of each talk, and send them you did—from all over the world. Special thanks are due Ron Gabor and Mike “madbot” McCloskey for the sheer magnitude of their contributions. Ron contributed Puzzles 28, 29, 30, and 31 and Mike contributed Puzzles 18, 23, 40, 56, and 67. We thank Martin Buchholz for contributing Puzzle 81; Armand Dijamco for contributing Puzzle 14; Prof. Dr. Dominik Gruntz for contributing Puzzles 68 and 69; Kai Huang for contributing Puzzle 77; Jim Hugunin for contributing Puzzle 45; Tim Huske for contributing Puzzle 41; Peter Kessler for contributing Puzzle 35; Michael Klobe for contributing Puzzle 59; Magnus Lundgren for contributing Puzzle 84; Scott Seligman for contributing Puzzle 22; Peter Stout for contributing Puzzle 39; Michael Tennes for contributing Puzzle 70; and Martin Traverso for contributing Puzzle 54.

We thank the dedicated band of reviewers who read the chapters of this book in raw form: Peter von der Ahé, Pablo Bellver, Tracy Bialik, Cindy Bloch, Dan Bloch, Beth Bottos, Joe Bowbeer, Joe Darcy, Bob Evans, Brian Goetz, Tim Halloran, Barry Hayes, Tim Huske, Akiyoshi Kitaoka, Chris Lopez, Mike “madbot” McCloskey, Michael Nielsen, Tim Peierls, Peter Rathmann, Russ Rufer, Steve Schirripa, Yoshiki Shibata, Marshall Spight, Guy Steele, Dean Sutherland, Mark Taylor, Darlene Wallach, and Frank Yellin. They found flaws, suggested improvements, offered encouragement, and hurled invective. Any flaws that remain are the fault of my coauthor.

We thank the queen of the bloggers, Mary Smaragdis, for providing a home for us on her celebrated blog [MaryBlog]. She graciously let us try out the material that became Puzzles 43, 53, 73, 87, and 94 on her readers. We judged the solutions and Mary gave out the prizes. For the record, the winners were Tom Hawtin, Tom Hawtin (again), Bob “Crazybob” Lee, Chris Nokleberg, and the mysterious AT of Odessa, Ukraine. The discussions on Mary’s blog contributed greatly to these puzzles.

We thank our many supporters who responded enthusiastically to Java Puzzlers over the years. The members of SDForum Java SIG served as guinea pigs for each talk in its preliminary form. The JavaOne program committee provided a home for the talks. Yuka Kamiya and Masayoshi Okutsu made the “Java Puzzlers” talks a success in Japan, where they took the form of real game shows with onstage contestants. Remarkably, the same person won every single contest: The undisputed Java Puzzler champion of Japan is Yasuhiro Endoh.

We thank James Gosling and the many fine engineers who created the Java platform and improved it over the years. A book like this makes sense only for a platform that is rock solid; without Java, there could be no "Java Puzzlers."

Numerous colleagues at Google, Sun, and elsewhere participated in technical discussions that improved the quality of this book. Among others, Peter von der Ahé, Dan Bloch, and Gilad Bracha contributed useful insights. We give special thanks to Doug Lea, who served as a sounding board for many of the ideas in the book. Once again, Doug was unfailingly generous with his time and knowledge.

We thank Professor Akiyoshi Kitaoka of the Department of Psychology at Ritsumeikan University in Kyoto, Japan, for permission to use some of his optical illusions to decorate this work. Professor Kitaoka's illusions are, quite simply, astonishing. Words cannot do them justice, so you owe it to yourself to take a look. He has two volumes available in Japanese [Kitaoka02, Kitaoka03]. An English translation encompassing both volumes is available [Kitaoka05]. While you wait for your copy to arrive in the mail, pay a visit to Kitaoka's Web site: http://www.ritsumei.ac.jp/~akitaoka/index-e.html. You won't be disappointed.

We thank Tom and Ray Magliozzi of *Car Talk* for providing jokes for us to steal, and we thank their legal counsel of Dewey, Cheetham, and Howe for not suing us.

We thank Josh's wife, Cindy, for helping us with FrameMaker, writing the index, helping us edit the book, and designing the decorative stripe at the beginning of each chapter. Last but not least, we thank our families—Cindy, Tim, and Matt Bloch, and Ricki Lee, Sarah, and Hannah Gafter—for encouraging us to write and for putting up with us while we wrote.

Josh Bloch
Neal Gafter
San Jose, California
May 2005

1 Introduction

This book is filled with brainteasers about the Java programming language and its core libraries. Anyone with a working knowledge of Java can understand these puzzles, but many of them are tough enough to challenge even the most experienced programmer. Don't feel bad if you can't solve them. They are grouped loosely according to the features they use, but don't assume that the trick to a puzzle is related to its chapter heading; we reserve the right to mislead you.

Most of the puzzles exploit counterintuitive or obscure behaviors that can lead to bugs. These behaviors are known as *traps, pitfalls,* and *corner cases*. Every platform has them, but Java has far fewer than other platforms of comparable power. The goal of the book is to entertain you with puzzles while teaching you to avoid the underlying traps and pitfalls. By working through the puzzles, you will become less likely to fall prey to these dangers in your code and more likely to spot them in code that you are reviewing or revising.

This book is meant to be read with a computer at your side. You'll need a Java development environment, such as Sun's JDK [JDK-5.0]. It should support release 5.0, as some of the puzzles rely on features introduced in this release. You can download the source code for the puzzles from www.javapuzzlers.com. Unless you're a glutton for punishment, we recommend that you do this before solving the puzzles. It's a heck of a lot easier than typing them in yourself.

Most of the puzzles take the form of a short program that appears to do one thing but actually does something else. It's your job to figure out what the program does. To get the most out of these puzzles, we recommend that you take this approach:

1. Study the program and try to predict its behavior without using a computer. If you don't see a trick, keep looking.
2. Once you think you know what the program does, run it. Did it do what you thought it would? If not, can you come up with an explanation for the behavior you observed?
3. Think about how you might fix the program, assuming it is broken.
4. Then and only then, read the solution.

Some of the puzzles require you to write a small amount of code. To get the most out of these puzzles, we recommend that you try—at least briefly—to solve them without using a computer, and then test your solution on a computer. If your code doesn't work, play around with it and see whether you can make it work before reading the solution.

Unlike most puzzle books, this one alternates between puzzles and their solutions. This allows you to read the book without flipping back and forth between puzzles and solutions. The book is laid out so that you must turn the page to get from a puzzle to its solution, so you needn't fear reading a solution accidentally while you're still trying to solve a puzzle.

We encourage you to read each solution, even if you succeed in solving the puzzle. The solutions contain analysis that goes well beyond a simple explanation of the program's behavior. They discuss the relevant traps and pitfalls, and provide lessons on how to avoid falling prey to these hazards. Like most best-practice guidelines, these lessons are not hard-and-fast rules, but you should violate them only rarely and with good reason.

Most solutions contain references to relevant sections of *The Java™ Language Specification, Third Edition* [JLS]. These references aren't essential to understanding the puzzles, but they are useful if you want to delve deeper into the language rules underlying the puzzles. Similarly, many solutions contain references to relevant items in *Effective Java™ Programming Language Guide* [EJ]. These references are useful if you want to delve deeper into best practices.

Some solutions contain discussions of the language or API design decisions that led to the danger illustrated by the puzzle. These "lessons for language designers" are meant only as food for thought and, like other food, should be taken with a grain of salt. Language design decisions cannot be made in isolation. Every language embodies thousands of design decisions that interact in subtle ways. A design decision that is right for one language may be wrong for another.

Many of the traps and pitfalls in these puzzles are amenable to automatic detection by *static analysis*: analyzing programs without running them. Some excellent tools are available for detecting bugs by static analysis, such as Bill Pugh and David Hovemeyer's *FindBugs* [Hovemeyer04]. Some compilers and IDEs, such as Jikes and Eclipse, perform bug detection as well [Jikes, Eclipse]. If you are using one of these compilers, it is especially important that you not compile a puzzle until you've tried to solve it: The compiler's warning messages may give away the solution.

Appendix A of this book is a catalog of the traps and pitfalls in the Java platform. It provides a concise taxonomy of the anomalies exploited by the puzzles, with references back to the puzzles and to other relevant resources. Do not look at the appendix until you're done solving the puzzles. Reading the appendix first would take all the fun out of the puzzles. After you've finished the puzzles, though, this is the place you'll turn to for reference.

Appendix B describes the optical illusions that decorate the book. This appendix explains the nature of each illusion and identifies the inventor, if known. It provides many bibliographic references. This appendix is the place to go if you aren't sure why a given drawing constitutes an illusion, or if you simply want to know more about one of the illusions.

2

Expressive Puzzlers

The puzzles in this chapter are simple. They involve only expression evaluation. But remember, just because they're simple doesn't make them easy.

Puzzle 1: Oddity

The following method purports to determine whether its sole argument is an odd number. Does the method work?

```
public static boolean isOdd(int i) {
    return i % 2 == 1;
}
```

Solution 1: Oddity

An odd number can be defined as an integer that is divisible by 2 with a remainder of 1. The expression `i % 2` computes the remainder when `i` is divided by 2, so it would seem that this program ought to work. Unfortunately, it doesn't; it returns the wrong answer one quarter of the time.

Why one quarter? Because half of all `int` values are negative, and the `isOdd` method fails for all negative odd values. It returns `false` when invoked on any negative value, whether even or odd.

This is a consequence of the definition of Java's remainder operator (`%`). It is defined to satisfy the following identity for all `int` values `a` and all nonzero `int` values `b`:

```
(a / b) * b + (a % b) == a
```

In other words, if you divide `a` by `b`, multiply the result by `b`, and add the remainder, you are back where you started [JLS 15.17.3]. This identity makes perfect sense, but in combination with Java's truncating integer division operator [JLS 15.17.2], it implies that **when the remainder operation returns a nonzero result, it has the same sign as its left operand.**

The `isOdd` method and the definition of the term *odd* on which it was based both assume that all remainders are positive. Although this assumption makes sense for some kinds of division [Boute92], Java's remainder operation is perfectly matched to its integer division operation, which discards the fractional part of its result.

When `i` is a negative odd number, `i % 2` is equal to `-1` rather than `1`, so the `isOdd` method incorrectly returns `false`. To prevent this sort of surprise, **test that your methods behave properly when passed negative, zero, and positive values for each numerical parameter.**

The problem is easy to fix. Simply compare `i % 2` to `0` rather than to `1`, and reverse the sense of the comparison:

```
public static boolean isOdd(int i) {
    return i % 2 != 0;
}
```

If you are using the `isOdd` method in a performance-critical setting, you would be better off using the bitwise AND operator (&) in place of the remainder operator:

```
public static boolean isOdd(int i) {
    return (i & 1) != 0;
}
```

The second version may run much faster than the first, depending on what platform and virtual machine you are using, and is unlikely to run slower. As a general rule, the divide and remainder operations are slow compared to other arithmetic and logical operations. **It's a bad idea to optimize prematurely**, but in this case, the faster version is as clear as the original, so there is no reason to prefer the original.

In summary, think about the signs of the operands and of the result whenever you use the remainder operator. The behavior of this operator is obvious when its operands are nonnegative, but it isn't so obvious when one or both operands are negative.

Puzzle 2: Time for a Change

Consider the following word problem:

> Tom goes to the auto parts store to buy a spark plug that costs $1.10, but all he has in his wallet are two-dollar bills. How much change should he get if he pays for the spark plug with a two-dollar bill?

Here is a program that attempts to solve the word problem. What does it print?

```
public class Change {
    public static void main(String args[]) {
        System.out.println(2.00 - 1.10);
    }
}
```

Solution 2: Time for a Change

Naively, you might expect the program to print `0.90`, but how could it know that you wanted two digits after the decimal point? If you know something about the rules for converting `double` values to strings, which are specified by the documentation for `Double.toString` [Java-API], you know that the program prints the shortest decimal fraction sufficient to distinguish the `double` value from its nearest neighbor, with at least one digit before and after the decimal point. It seems reasonable, then, that the program should print `0.9`. Reasonable, perhaps, but not correct. If you ran the program, you found that it prints `0.8999999999999999`.

The problem is that the number 1.1 can't be represented exactly as a `double`, so it is represented by the closest `double` value. The program subtracts this value from 2. Unfortunately, the result of this calculation is not the closest `double` value to 0.9. The shortest representation of the resulting `double` value is the hideous number that you see printed.

More generally, the problem is that **not all decimals can be represented exactly using binary floating-point.** If you are using release 5.0 or a later release, you might be tempted to fix the program by using the `printf` facility to set the precision of the output:

```
// Poor solution - still uses binary floating-point!
System.out.printf("%.2f%n", 2.00 - 1.10);
```

This prints the right answer but does not represent a general solution to the underlying problem: It still uses `double` arithmetic, which is binary floating-point. Floating-point arithmetic provides good approximations over a wide range of values but does not generally yield exact results. **Binary floating-point is particularly ill-suited to monetary calculations**, as it is impossible to represent 0.1—or any other negative power of 10—exactly as a finite-length binary fraction [EJ Item 31].

One way to solve the problem is to use an integral type, such as `int` or `long`, and to perform the computation in cents. If you go this route, make sure the integral type is large enough to represent all the values you will use in your program. For this puzzle, `int` is ample. Here is how the `println` looks if we rewrite it using `int` values to represent monetary values in cents. This version prints `90 cents`, which is the right answer:

```
System.out.println((200 - 110) + " cents");
```

Another way to solve the problem is to use `BigDecimal`, which performs exact decimal arithmetic. It also interoperates with the SQL `DECIMAL` type via JDBC. There is one caveat: **Always use the `BigDecimal(String)` constructor, never `BigDecimal(double)`.** The latter constructor creates an instance with the *exact* value of its argument: `new BigDecimal(.1)` returns a `BigDecimal` representing 0.1000000000000000055511151231257827021181583404541015625. Using `BigDecimal` correctly, the program prints the expected result of `0.90`:

```
import java.math.BigDecimal;
public class Change {
    public static void main(String args[]) {
        System.out.println(new BigDecimal("2.00").
                           subtract(new BigDecimal("1.10")));
    }
}
```

This version is not terribly pretty, as Java provides no linguistic support for `BigDecimal`. Calculations with `BigDecimal` are also likely to be slower than those with any primitive type, which might be an issue for some programs that make heavy use of decimal calculations. It is of no consequence for most programs.

In summary, **avoid `float` and `double` where exact answers are required; for monetary calculations, use `int`, `long`, or `BigDecimal`.** For language designers, consider providing linguistic support for decimal arithmetic. One approach is to offer limited support for operator overloading, so that arithmetic operators can be made to work with numerical reference types, such as `BigDecimal`. Another approach is to provide a primitive decimal type, as did COBOL and PL/I.

Puzzle 3: Long Division

This puzzle is called Long Division because it concerns a program that divides two `long` values. The dividend represents the number of microseconds in a day; the divisor, the number of milliseconds in a day. What does the program print?

```
public class LongDivision {
    public static void main(String[] args) {
        final long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
        final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
    }
}
```

Solution 3: Long Division

This puzzle seems reasonably straightforward. The number of milliseconds per day and the number of microseconds per day are constants. For clarity, they are expressed as products. The number of microseconds per day is (24 hours/day · 60 minutes/hour · 60 seconds/minute · 1,000 milliseconds/second · 1,000 microseconds/millisecond). The number of milliseconds per day differs only in that it is missing the final factor of 1,000. When you divide the number of microseconds per day by the number of milliseconds per day, all the factors in the divisor cancel out, and you are left with 1,000, which is the number of microseconds per millisecond. Both the divisor and the dividend are of type `long`, which is easily large enough to hold either product without overflow. It seems, then, that the program must print `1000`. Unfortunately, it prints `5`. What exactly is going on here?

The problem is that the computation of the constant `MICROS_PER_DAY` *does* overflow. Although the result of the computation fits in a `long` with room to spare, it doesn't fit in an `int`. The computation is performed entirely in `int` arithmetic, and only after the computation completes is the result promoted to a `long`. By then, it's too late: The computation has already overflowed, returning a value that is too low by a factor of 200. The promotion from `int` to `long` is a *widening primitive conversion* [JLS 5.1.2], which preserves the (incorrect) numerical value. This value is then divided by `MILLIS_PER_DAY`, which was computed correctly because it does fit in an `int`. The result of this division is 5.

So why is the computation performed in `int` arithmetic? Because all the factors that are multiplied together are `int` values. When you multiply two `int` values, you get another `int` value. Java does not have *target typing*, a language feature wherein the type of the variable in which a result is to be stored influences the type of the computation.

It's easy to fix the program by using a `long` literal in place of an `int` as the first factor in each product. This forces all subsequent computations in the expression to be done with `long` arithmetic. Although it is necessary to do this only in the expression for `MICROS_PER_DAY`, it is good form to do it in both products. Similarly, it isn't always necessary to use a `long` as the *first* value in a product, but it is good form to do so. Beginning both computations with `long` values makes it clear that they won't overflow. This program prints `1000` as expected:

```
public class LongDivision {
    public static void main(String[] args) {
        final long MICROS_PER_DAY = 24L * 60 * 60 * 1000 * 1000;
        final long MILLIS_PER_DAY = 24L * 60 * 60 * 1000;
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
    }
}
```

The lesson is simple: **When working with large numbers, watch out for overflow—it's a silent killer.** Just because a variable is large enough to hold a result doesn't mean that the computation leading to the result is of the correct type. When in doubt, perform the entire computation using `long` arithmetic.

The lesson for language designers is that it may be worth reducing the likelihood of silent overflow. This could be done by providing support for arithmetic that does not overflow silently. Programs could throw an exception instead of overflowing, as does Ada, or they could switch to a larger internal representation automatically as required to avoid overflow, as does Lisp. Both of these approaches may have performance penalties associated with them. Another way to reduce the likelihood of silent overflow is to support target typing, but this adds significant complexity to the type system [Modula-3 1.4.8].

Puzzle 4: It's Elementary

OK, so the last puzzle was a bit tricky, but it was about division. Everyone knows that division is tough. This program involves only addition. What does it print?

```
public class Elementary {
    public static void main(String[] args) {
        System.out.println(12345 + 5432l);
    }
}
```

Solution 4: It's Elementary

On the face of it, this looks like an easy puzzle—so easy that you can solve it without pencil or paper. The digits of the left operand of the plus operator ascend from 1 to 5, and the digits of the right operand descend. Therefore, the sums of corresponding digits remain constant, and the program must surely print `66666`. There is only one problem with this analysis: When you run the program, it prints `17777`. Could it be that Java has an aversion to printing such a beastly number? Somehow this doesn't seem like a plausible explanation.

Things are seldom what they seem. Take this program, for instance. It doesn't say what you think it does. Take a careful look at the two operands of the + operator. We are adding the `int` value `12345` to the `long` value `5432l`. Note the subtle difference in shape between the digit 1 at the beginning of the left operand and the lowercase letter *el* at the end of the right operand. The digit 1 has an acute angle between the horizontal stroke, or *arm*, and the vertical stroke, or *stem*. The lowercase letter *el*, by contrast, has a right angle between the arm and the stem.

Before you cry "foul," note that this issue has caused real confusion. Also note that the puzzle's title contained a hint: It's El-ementary; get it? Finally, note that there is a real lesson here. **Always use a capital *el* (`L`) in `long` literals, never a lowercase *el* (`l`).** This completely eliminates the source of confusion on which the puzzle relies:

```
System.out.println(12345 + 5432L);
```

Similarly, **avoid using a lone *el* (`l`) as a variable name.** It is difficult to tell by looking at this code snippet whether it prints the list `l` or the number `1`:

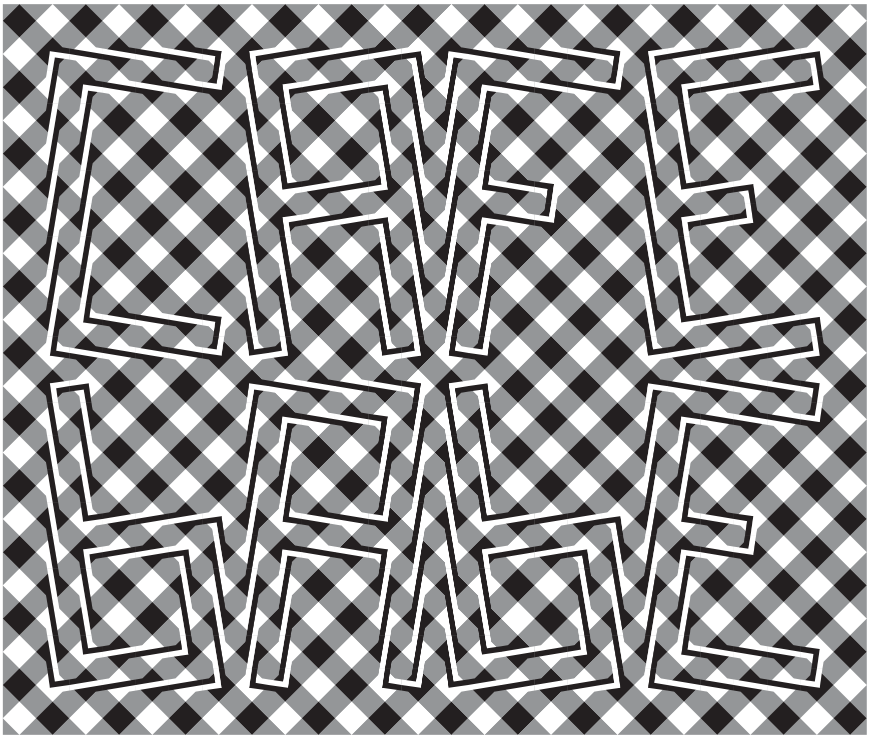
```
// Bad code - uses el (l) as a variable name
List<String> l = new ArrayList<String>();
l.add("Foo");
System.out.println(1);
```

In summary, the lowercase letter *el* and the digit 1 are nearly identical in most typewriter fonts. To avoid confusing the readers of your program, never use a lowercase *el* to terminate a `long` literal or as a variable name. Java inherited much from the C programming language, including its syntax for `long` literals. It was probably a mistake to allow `long` literals to be written with a lowercase *el*.

Puzzle 5: The Joy of Hex

The following program adds two hexadecimal, or “hex,” literals and prints the result in hex. What does the program print?

```
public class JoyOfHex {
    public static void main(String[] args) {
        System.out.println(
            Long.toHexString(0x100000000L + 0xcafebabe));
    }
}
```



Solution 5: The Joy of Hex

It seems obvious that the program should print `1cafebabe`. After all, that is the sum of the hex numbers 100000000_{16} and cafebabe_{16}. The program uses `long` arithmetic, which permits 16 hex digits, so arithmetic overflow is not an issue. Yet, if you ran the program, you found that it prints `cafebabe`, with no leading `1` digit. This output represents the low-order 32 bits of the correct sum, but somehow the thirty-third bit gets lost. It is as if the program were doing `int` arithmetic instead of `long`, or forgetting to add the first operand. What's going on here?

Decimal literals have a nice property that is not shared by hexadecimal or octal literals: Decimal literals are all positive [JLS 3.10.1]. To write a negative decimal constant, you use the unary negation operator (`-`) in combination with a decimal literal. In this way, you can write any `int` or `long` value, whether positive or negative, in decimal form, and **negative decimal constants are clearly identifiable by the presence of a minus sign.** Not so for hexadecimal and octal literals. They can take on both positive and negative values. **Hex and octal literals are negative if their high-order bit is set.** In this program, the number `0xcafebabe` is an `int` constant with its high-order bit set, so it is negative. It is equivalent to the decimal value `-889275714`.

The addition performed by the program is a *mixed-type computation*: The left operand is of type `long`, and the right operand is of type `int`. To perform the computation, Java promotes the `int` value to a `long` with a *widening primitive conversion* [JLS 5.1.2] and adds the two `long` values. Because `int` is a signed integral type, the conversion performs *sign extension*: It promotes the negative `int` value to a numerically equal `long` value.

The right operand of the addition, `0xcafebabe`, is promoted to the `long` value `0xffffffffcafebabeL`. This value is then added to the left operand, which is `0x100000000L`. When viewed as an `int`, the high-order 32 bits of the sign-extended right operand are `-1`, and the high-order 32 bits of the left operand are `1`. Add these two values together and you get `0`, which explains the absence of the leading `1` digit in the program's output. Here is how the addition looks when done in longhand. (The digits at the top of the addition are carries.)

```
   1111111
  0xffffffffcafebabeL
+ 0x0000000100000000L
  -------------------
  0x00000000cafebabeL
```

Fixing the problem is as simple as using a `long` hex literal to represent the right operand. This avoids the damaging sign extension, and the program prints the expected result of `1cafebabe`:

```
public class JoyOfHex {
    public static void main(String[] args) {
        System.out.println(
            Long.toHexString(0x100000000L + 0xcafebabeL));
    }
}
```

The lesson of this puzzle is that mixed-type computations can be confusing, more so given that hex and octal literals can take on negative values without an explicit minus sign. To avoid this sort of difficulty, **it is generally best to avoid mixed-type computations.** For language designers, it is worth considering support for unsigned integral types, which eliminate the possibility of sign extension. One might argue that negative hex and octal literals should be prohibited, but this would likely frustrate programmers, who often use hex literals to represent values whose sign is of no significance.

Puzzle 6: Multicast

Casts are used to convert a value from one type to another. This program uses three casts in succession. What does it print?

```
public class Multicast {
    public static void main(String[] args) {
        System.out.println((int) (char) (byte) -1);
    }
}
```