A *Java Puzzlers* Sampler

This sampler contains one puzzle from each chapter of *Java Puzzlers* by Joshua Bloch and Neal Gafter (Addison Wesley, 2005). The 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.

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.

The book is meant to be read with a computer at your side. You'll need a Java development environment, such as Sun's JDK. It should support release 5.0, as some puzzles rely on features introduced in this release.

Most of the puzzles take the form of a short program that appears to do one thing but actually does 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. It's your job to figure out each 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.

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.

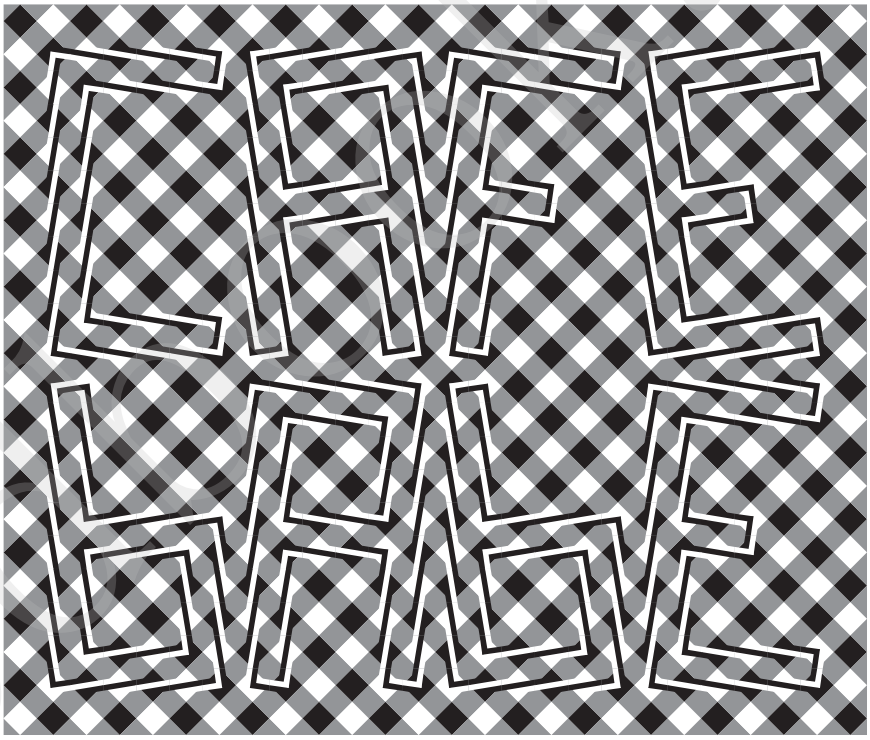
Some solutions contain discussions of the language or API design decisions that led to the danger illustrated by the puzzle. These "lessons for language designers" are meant only as food for thought and, like other food, should be taken with a grain of salt. Language design decisions cannot be made in 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.

The book contains two appendices. Appendix A 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. Appendix B is describes the optical illusions contained in the book. This sampler includes the relevant parts of Appendix B.

Puzzle 1: 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 1: 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 2: Line Printer

The *line separator* is the name given to the character or characters used to separate lines of text, and varies from platform to platform. On Windows, it is the CR character (carriage return) followed by the LF character (linefeed). On UNIX, it is the LF character alone, often referred to as the newline character. The following program passes this character to `println`. What does it print? Is its behavior platform dependent?

```
public class LinePrinter {
  public static void main(String[] args) {
    // Note: \u000A is Unicode representation of linefeed (LF)
    char c = 0x000A;
    System.out.println(c);
  }
}
```

Solution 2: Line Printer

The behavior of this program is platform independent: It won't compile on any platform. If you tried to compile it, you got an error message that looks something like this:

```
LinePrinter.java:3: ';' expected
    // Note: \u000A is Unicode representation of linefeed (LF)
                                              ^
1 error
```

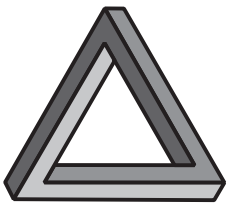
If you are like most people, this message did not help to clarify matters.

The key to this puzzle is the comment on the third line of the program. Like the best of comments, this one is true. Unfortunately, this one is a bit too true. The compiler not only translates Unicode escapes into the characters they represent before it parses a program into tokens, but it does so before discarding comments and white space [JLS 3.2].

This program contains a single Unicode escape (`\u000A`), located in its sole comment. As the comment tells you, this escape represents the linefeed character, and the compiler duly translates it before discarding the comment. Unfortunately, this linefeed character is the first *line terminator* after the two slash characters that begin the comment (`//`) and so terminates the comment [JLS 3.4]. The words following the escape (`is Unicode representation of linefeed (LF)`) are therefore not part of the comment; nor are they syntactically valid.

To make this more concrete, here is what the program looks like after the Unicode escape has been translated into the character it represents:

```
public class LinePrinter {
    public static void main(String[] args) {
        // Note:
 is Unicode representation of linefeed (LF)
        char c = 0x000A;
        System.out.println(c);
    }
}
```



The easiest way to fix the program is to remove the Unicode escape from the comment, but a better way is to initialize `c` with an escape sequence instead of a hex integer literal, obviating the need for the comment:

```
public class LinePrinter {
    public static void main(String[] args) {
        char c = '\n';
        System.out.println(c);
    }
}
```

Once this has been done, the program will compile and run, but it's still a questionable program. It is platform dependent for exactly the reason suggested in the puzzle. On certain platforms, such as UNIX, it will print two complete line separators; on others, such as Windows, it won't. Although the output may look the same to the naked eye, it could easily cause problems if it were saved in a file or piped to another program for subsequent processing.

If you want to print two blank lines, you should invoke `println` twice. As of release 5.0, you can use `printf` instead of `println`, with the format string `"%n%n"`. Each occurrence of the characters `%n` will cause `printf` to print the appropriate platform-specific line separator.

Hopefully, this puzzle convinced you that Unicode escapes can be thoroughly confusing. The lesson is simple: **Avoid Unicode escapes except where they are truly necessary.** They are rarely necessary.

Puzzle 3: A Big Delight in Every Byte

This program loops through the `byte` values, looking for a certain value. What does the program print?

```
public class BigDelight {
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++) {
            if (b == 0x90)
                System.out.print("Joy!");
        }
    }
}
```

Solution 3: A Big Delight in Every Byte

The loop iterates over all the `byte` values except `Byte.MAX_VALUE`, looking for `0x90`. This value fits in a byte and is not equal to `Byte.MAX_VALUE`, so you might think that the loop would hit it once and print `Joy!` on that iteration. Looks can be deceiving. If you ran the program, you found that it prints nothing. What happened?

Simply put, `0x90` is an `int` constant that is outside the range of `byte` values. This is counterintuitive because `0x90` is a two-digit hexadecimal literal. Each hex digit takes up 4 bits, so the entire value takes up 8 bits, or 1 byte. The problem is that `byte` is a signed type. The constant `0x90` is a positive `int` value of 8 bits with the highest bit set. Legal `byte` values range from −128 to +127, but the `int` constant `0x90` is equal to +144.

The comparison of a `byte` to an `int` is a *mixed-type comparison*. If you think of `byte` values as apples and `int` values as oranges, the program is comparing apples to oranges. Consider the expression `((byte)0x90 == 0x90)`. Appearances notwithstanding, it evaluates to `false`. To compare the `byte` value `(byte)0x90` to the `int` value `0x90`, Java promotes the `byte` to an `int` with a widening primitive conversion [JLS 5.1.2] and compares the two `int` values. Because `byte` is a signed type, the conversion performs *sign extension,* promoting negative `byte` values to numerically equal `int` values. In this case, the conversion promotes `(byte)0x90` to the `int` value `-112`, which is unequal to the `int` value `0x90`, or `+144`.

Mixed-type comparisons are always confusing because the system is forced to promote one operand to match the type of the other. The conversion is invisible and may not yield the results that you expect. There are several ways to avoid mixed-type comparisons. To pursue our fruit metaphor, you can choose to compare apples to apples or oranges to oranges. You can cast the `int` to a `byte`, after which you will be comparing one `byte` value to another:

```
if (b == (byte)0x90)
    System.out.println("Joy!");
```

Alternatively, you can convert the `byte` to an `int`, suppressing sign extension with a mask, after which you will be comparing one `int` value to another:

```
if ((b & 0xff) == 0x90)
    System.out.println("Joy!");
```

Either of these solutions works, but the best way to avoid this kind of problem is to move the constant value outside the loop and into a constant declaration.

Here is a first attempt:

```
public class BigDelight {
    private static final byte TARGET = 0x90;   // Broken!
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++)
            if (b == TARGET)
                System.out.print("Joy!");
    }
}
```

Unfortunately, it doesn't compile. The constant declaration is broken, and the compiler will tell you the problem: `0x90` is not a valid value for the type `byte`. If you fix the declaration as follows, the program will work fine:

```
private static final byte TARGET = (byte)0x90;
```

To summarize: **Avoid mixed-type comparisons, because they are inherently confusing**. To help achieve this goal, **use declared constants in place of "magic numbers."** You already knew that this was a good idea; it documents the meanings of constants, centralizes their definitions, and eliminates duplicate definitions. Now you know that it also forces you to give each constant a type appropriate for its use, eliminating one source of mixed-type comparisons.

The lesson for language designers is that sign extension of `byte` values is a common source of bugs and confusion. The masking that is required in order to suppress sign extension clutters programs, making them less readable. Therefore, the `byte` type should be unsigned. Also, consider providing literals for all primitive types, reducing the need for error-prone type conversions.

Puzzle 4: Hello, Goodbye

This program adds an unusual twist to the usual `Hello world` program. What does it print?

```
public class HelloGoodbye {
    public static void main(String[] args) {
        try {
            System.out.println("Hello world");
            System.exit(0);
        } finally {
            System.out.println("Goodbye world");
        }
    }
}
```

Solution 4: Hello, Goodbye

The program contains two `println` statements: one in a `try` block and the other in the corresponding `finally` block. The `try` block executes its `println` and finishes execution prematurely by calling `System.exit`. At this point, you might expect control to transfer to the `finally` block. If you tried the program, though, you found that it never can say goodbye: It prints only `Hello world`.

It is true that a `finally` block is executed when a `try` block completes execution whether normally or abruptly. In this program, however, the `try` block does not complete execution at all. **The `System.exit` method halts the execution of the current thread and all others dead in their tracks.** The presence of a `finally` clause does not give a thread special permission to continue executing.

When `System.exit` is called, the virtual machine performs two cleanup tasks before shutting down. First, it executes all *shutdown hooks* that have been registered with `Runtime.addShutdownHook`. This is useful to release resources external to the VM. **Use shutdown hooks for behavior that must occur before the VM exits.** The following version of the program demonstrates this technique, printing both `Hello world` and `Goodbye world`, as expected:

```
public class HelloGoodbye {
    public static void main(String[] args) {
        System.out.println("Hello world");
        Runtime.getRuntime().addShutdownHook(
            new Thread() {
                public void run() {
                    System.out.println("Goodbye world");
                }
            });
        System.exit(0);
    }
}
```

The second cleanup task performed by the VM when `System.exit` is called concerns finalizers. If either `System.runFinalizersOnExit` or its evil twin `Runtime.runFinalizersOnExit` has been called, the VM runs the finalizers on all objects that have not yet been finalized. These methods were deprecated a long time ago and with good reason. **Never call `System.runFinalizersOnExit` or `Runtime.runFinalizersOnExit` for any reason: They are among the most dangerous methods in the Java libraries**. Calling these methods can result in

finalizers being run on live objects while other threads are concurrently manipulating them, resulting in erratic behavior or deadlock.

In summary, `System.exit` stops all program threads immediately; it does not cause `finally` blocks to execute, but it does run shutdown hooks before halting the VM. Use shutdown hooks to terminate external resources when the VM shuts down. It is possible to halt the VM without executing shutdown hooks by calling `System.halt`, but this method is rarely used.

Puzzle 5: Larger Than Life

Lest you think that this book is going entirely to the dogs, this puzzle concerns royalty. If the tabloids are to be believed, the King of Rock 'n' Roll is still alive. Not one of his many impersonators but the one true Elvis. This program estimates his current belt size by projecting the trend observed during his public performances. The program uses the idiom `Calendar.getInstance().get(Calendar.YEAR)`, which returns the current calendar year. What does the program print?

```
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private final int beltSize;
    private static final int CURRENT_YEAR =
        Calendar.getInstance().get(Calendar.YEAR);

    private Elvis() {
        beltSize = CURRENT_YEAR - 1930;
    }

    public int beltSize() {
        return beltSize;
    }

    public static void main(String[] args) {
        System.out.println("Elvis wears a size " +
                           INSTANCE.beltSize() + " belt.");
    }
}
```

Solution 5: Larger Than Life

At first glance, this program appears to compute the current year minus 1930. If that were correct, in the year 2006, the program would print `Elvis wears a size 76 belt.` If you tried running the program, you learned that the tabloids were wrong, proving that you can't believe everything you read in the papers. It prints `Elvis wears a size -1930 belt.` Perhaps the King has gone on to inhabit an anti-matter universe?

This program suffers from a circularity in the order of *class initialization* [JLS 12.4]. Let's follow it in detail. Initialization of the class `Elvis` is triggered by the VM's call to its `main` method. First, static fields are set to their default values [JLS 4.12.5]. The field `INSTANCE` is set to `null`, and `CURRENT_YEAR` is set to `0`. Next, static field initializers are executed in order of appearance. The first static field is `INSTANCE`. Its value is computed by invoking the `Elvis()` constructor.

The constructor initializes `beltSize` to an expression involving the static field `CURRENT_YEAR`. Normally, reading a static field is one of the things that causes a class to be initialized, but we are already initializing the class `Elvis`. Recursive initialization attempts are simply ignored [JLS 12.4.2, step 3]. Consequently, the value of `CURRENT_YEAR` still has its default value of 0. That is why Elvis's belt size turns out to be `-1930`.

Finally, returning from the constructor to complete the class initialization of `Elvis`, we initialize the static field `CURRENT_YEAR` to 2006, assuming you're running the program in 2006. Unfortunately, it is too late for the now correct value of this field to affect the computation of `Elvis.INSTANCE.beltSize`, which already has the value `-1930`. This is the value that will be returned by all subsequent calls to `Elvis.INSTANCE.beltSize`().

This program shows that **it is possible to observe a final static field before it is initialized,** when it still contains the default value for its type. That is counterintuitive, because we usually think of final fields as constants. Final fields are constants only if the initializing expression is a *constant expression* [JLS 15.28].

Problems arising from cycles in class initialization are difficult to diagnose but once diagnosed are usually easy to fix. **To fix a class initialization cycle, reorder the static field initializers so that each initializer appears before any initializers that depend on it.** In this program, the declaration for `CURRENT_YEAR` belongs before the declaration for `INSTANCE`, because the creation of an Elvis instance requires that `CURRENT_YEAR` be initialized. Once the declaration for `CURRENT_YEAR` has been moved, Elvis will indeed be larger than life.

Some common design patterns are naturally subject to initialization cycles, notably the Singleton [Gamma95], which is illustrated in this puzzle, and the Service Provider Framework [EJ Item 1]. The Typesafe Enum pattern [EJ Item 21] also causes class initialization cycles. Release 5.0 adds linguistic support for this pattern with enum types. To reduce the likelihood of problems, there are some restrictions on static initializers in enum types [JLS 16.5, 8.9].

In summary, **be careful of class initialization cycles.** The simplest ones involve only a single class, but they can also involve multiple classes. It isn't always wrong to have class initialization cycles, but they may result in constructor invocation before static fields are initialized. Static fields, even final static fields, may be observed with their default value before they are initialized.

Puzzle 6: What's in a Name?

This program consists of a simple immutable class that represents a name, with a `main` method that puts a name into a set and checks whether the set contains the name. What does the program print?

```
import java.util.*;

public class Name {
    private final String first, last;

    public Name(String first, String last) {
        this.first = first;
        this.last = last;
    }

    public boolean equals(Object o) {
        if (!(o instanceof Name))
            return false;
        Name n = (Name)o;
        return n.first.equals(first) && n.last.equals(last);
    }

    public static void main(String[] args) {
        Set<Name> s = new HashSet<Name>();
        s.add(new Name("Mickey", "Mouse"));
        System.out.println(
            s.contains(new Name("Mickey", "Mouse")));
    }
}
```

Solution 6: What's in a Name?

A `Name` instance consists of a first name and a last name. Two `Name` instances are equal, as computed by the `equals` method, if their first names are equal and their last names are equal. First names and last names are compared using the `equals` method defined in `String`. Two strings are equal if they consist of the same characters in the same order. Therefore, two `Name` instances are equal if they represent the same name. For example, the following method invocation returns `true`:

```
new Name("Mickey", "Mouse").equals(new Name("Mickey", "Mouse"))
```

The `main` method of the program creates two `Name` instances, both representing Mickey Mouse. The program puts the first instance into a hash set and then checks whether the set contains the second. The two `Name` instances are equal, so it might seem that the program should print `true`. If you ran it, it almost certainly printed `false`. What is wrong with the program?

The bug is that `Name` violates the `hashCode` contract. This might seem strange, as `Name` doesn't even have a `hashCode` method, but that is precisely the problem. The `Name` class overrides the `equals` method, and the `hashCode` contract demands that equal objects have equal hash codes. To fulfill this contract, **you must override `hashCode` whenever you override `equals`** [EJ Item 8].

Because it fails to override `hashCode`, the `Name` class inherits its `hashCode` implementation from `Object`. This implementation returns an *identity-based* hash code. In other words, distinct objects are likely to have unequal hash values, even if they are equal. `Name` does not fulfill the `hashCode` contract, so the behavior of a hash set containing `Name` elements is unspecified.

When the program puts the first `Name` instance into the hash set, the set puts an entry for this instance into a hash bucket. The set chooses the hash bucket based on the hash value of the instance, as computed by its `hashCode` method. When it checks whether the second `Name` instance is contained in the hash set, the program chooses which bucket to search based on the hash value of the second instance. Because the second instance is distinct from the first, it is likely to have a different hash value. If the two hash values map to different buckets, the `contains` method will return `false`: The beloved rodent is in the hash set, but the set can't find him.

Suppose that the two `Name` instances map to the same bucket. Then what? All `HashSet` implementations that we know of have an optimization in which each entry stores the hash value of its element in addition to the element itself. When

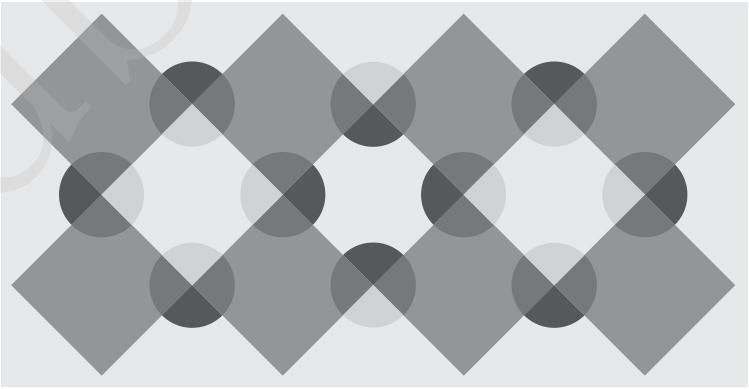
searching for an element, the implementation selects the appropriate hash bucket and traverses its entries, comparing the hash value stored in each entry with the hash value of the desired element. Only if the two hash values are equal does the implementation check the elements for equality. This optimization makes sense because it is usually much cheaper to compare hash codes than elements.

Because of this optimization, it is not enough for the hash set to search in the right bucket; the two `Name` instances must have equal hash values in order for the hash set to recognize them as equal. The odds that the program prints `true` are therefore the odds that two consecutively created objects have the same identity hash code. A quick experiment showed these odds to be about one in 25,000,000. Results may vary depending on which Java implementation is used, but you are highly unlikely to see the program print `true` on any JRE we know of.

To fix the problem, simply add an appropriate `hashCode` method to the `Name` class. Although any method whose return value is determined solely by the first and last name will satisfy the contract, a high-quality hash function should attempt to return different hash values for different names. The following method will do nicely [EJ Item 8]. Once this method is added, the program will print `true` as expected:

```
public int hashCode() {
    return 37 * first.hashCode() + last.hashCode();
}
```

In summary, always override `hashCode` when you override `equals`. More generally, obey the general contract when you override a method that has one. This is an issue for most of the non-final methods declared in `Object` [EJ Chapter 3]. Failure to follow this advice can result in arbitrary, unspecified behavior.



Puzzle 7: All Strung Out

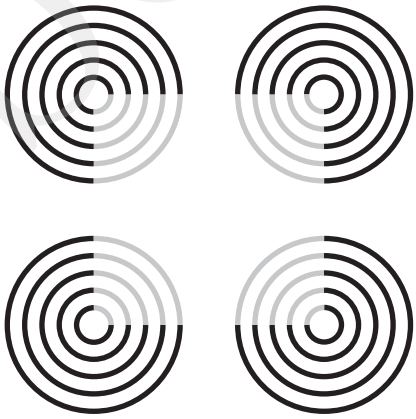
One name can be used to refer to multiple classes in different packages. This program explores what happens when you reuse a platform class name. What do you think it does? Although this is the kind of program you'd normally be embarrassed to be seen with, go ahead and lock the doors, close the shades, and give it a try:

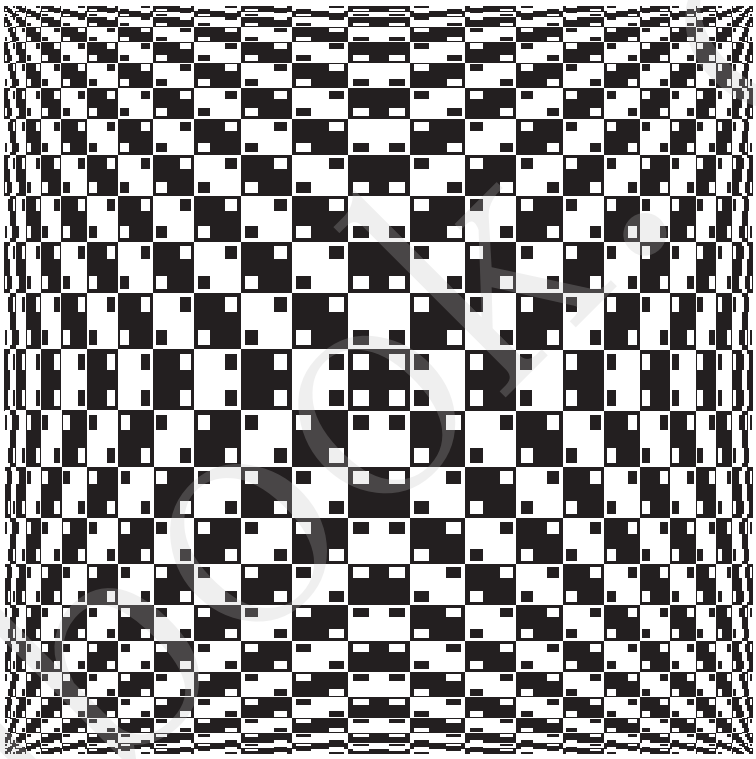
```
public class StrungOut {
    public static void main(String[] args) {
        String s = new String("Hello world");
        System.out.println(s);
    }
}

class String {
    private final java.lang.String s;

    public String(java.lang.String s) {
        this.s = s;
    }

    public java.lang.String toString() {
        return s;
    }
}
```





Solution 7: All Strung Out

This program looks simple enough, if a bit repulsive. The class `String` in the unnamed package is simply a wrapper for a `java.lang.String` instance. It seems the program should print `Hello world`. If you tried to run the program, though, you found that you could not. The VM emits an error message something like this:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

But surely there *is* a `main` method: It's right there in black and white. Why can't the VM find it?

The VM can't find the `main` method because it isn't there. Although `StrungOut` has a method *named* `main`, it has the wrong signature. A `main` method must accept a single argument that is an array of strings [JVMS 5.2]. What the VM is struggling to tell us is that `StrungOut.main` accepts an array of *our* `String` class, which has nothing whatsoever to do with `java.lang.String`.

If you really must write your own string class, for heaven's sake, don't call it `String`. **Avoid reusing the names of platform classes, and *never* reuse class names from `java.lang`,** because these names are automatically imported everywhere. Programmers are used to seeing these names in their unqualified form and naturally assume that these names refer to the familiar classes from `java.lang`. If you reuse one of these names, the unqualified name will refer to the new definition any time it is used inside its own package.

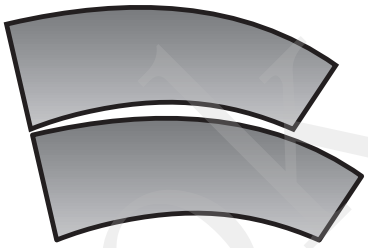
To fix the program, simply pick a reasonable name for the nonstandard string class. The following version of the program is clearly correct and much easier to understand than the original. It prints `Hello world` just as you'd expect:

```
public class StrungOut {
    public static void main(String[] args) {
        MyString s = new MyString("Hello world");
        System.out.println(s);
    }
}
class MyString {
    private final String s;

    public MyString(String s) { this.s = s; }

    public String toString()  { return s; }
}
```

Broadly speaking, the lesson of this puzzle is to avoid the reuse of class names, especially Java platform class names. Never reuse class names from the package `java.lang`. The same lesson applies to library designers. The Java platform designers slipped up a few times. Notable examples include `java.sql.Date`, which conflicts with `java.util.Date`, and `org.omg.CORBA.Object`. This lesson is a specific case of the principle that you should avoid name reuse, with the exception of overriding. For platform implementers, the lesson is that diagnostics should make clear the reason for a failure. The VM could easily have distinguished the case where there is no `main` method with the correct signature from the case where there is no `main` method at all.



Puzzle 8: Reflection Infection

This puzzle illustrates a simple application of reflection. What does this program print?

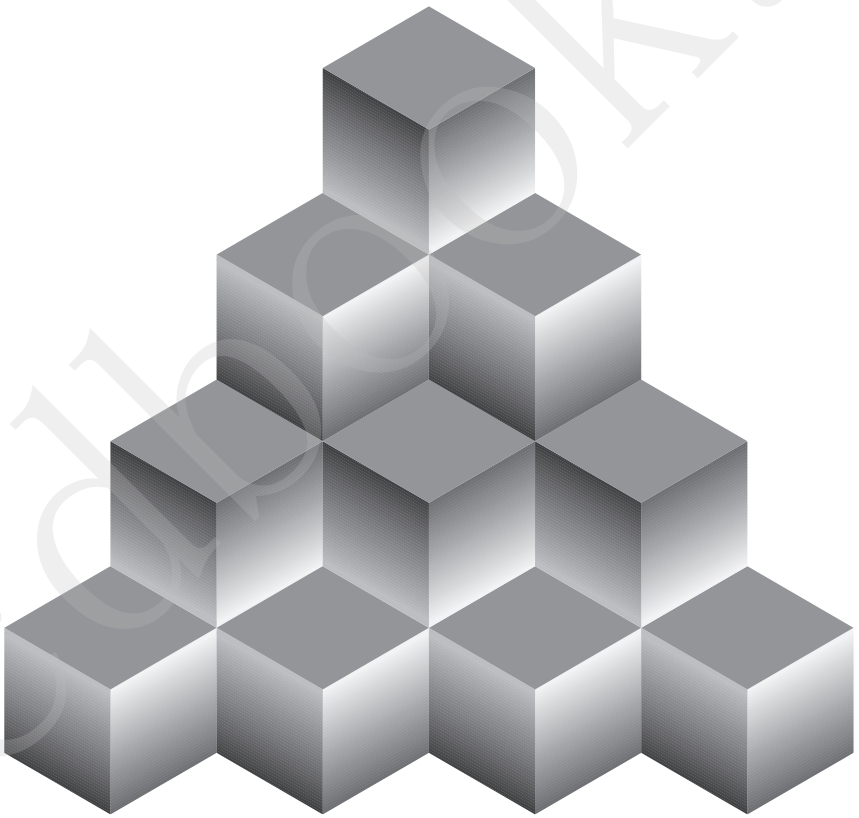
```
import java.util.*;
import java.lang.reflect.*;

public class Reflector {
    public static void main(String[] args) throws Exception {
        Set<String> s = new HashSet<String>();
        s.add("foo");
        Iterator it = s.iterator();
        Method m = it.getClass().getMethod("hasNext");
        System.out.println(m.invoke(it));
    }
}
```

Solution 8: Reflection Infection

The program creates a set with a single element in it, gets an iterator over the set, invokes the iterator's `hasNext` method reflectively, and prints the result of the method invocation. As the iterator hasn't yet returned the set's sole element, `hasNext` should return `true`. Running the program, however, tells a different story:

```
Exception in thread "main" IllegalAccessException:
  Class Reflector can not access a member of class HashMap
  $HashIterator with modifiers "public"
    at Reflection.ensureMemberAccess(Reflection.java:65)
    at Method.invoke(Method.java:578)
    at Reflector.main(Reflector.java:11)
```



How can this be? Of course the `hasNext` method is public, just as the exception tells us, and so can be accessed from anywhere. So why should the reflective method invocation be illegal?

The problem isn't the access level of the method; it's the access level of the type from which the method is selected. This type plays the same role as the *qualifying type* in an ordinary method invocation [JLS 13.1]. In this program, the method is selected from the class represented by the `Class` object that is returned by `it.getClass`. This is the dynamic type of the iterator, which happens to be the private nested class `java.util.HashMap.KeyIterator`. The reason for the `IllegalAccessException` is that this class is not public and comes from another package: **You cannot legally access a member of a nonpublic type from another package** [JLS 6.6.1].

This prohibition applies whether the access is normal or reflective. Here is a program that runs afoul of this rule without resorting to reflection:

```
package library;
public class Api {
    static class PackagePrivate {}
    public static PackagePrivate member = new PackagePrivate();
}

package client;
import library.Api;
class Client {
    public static void main(String[] args) {
        System.out.println(Api.member.hashCode());
    }
}
```

Attempting to compile the program results in this error:

```
Client.java:5: Object.hashCode() isn't defined in a public
class or interface; can't be accessed from outside package
        System.out.println(Api.member.hashCode());
                                      ^
```

This diagnostic makes about as much sense as the runtime error generated by the original reflective program. The class `Object` and the method `hashCode` are both public. The problem is that the `hashCode` method is invoked with a qualifying type that is inaccessible to the client. The qualifying type of the method invocation is `library.Api.PackagePrivate`, which is a nonpublic class in a different package.

This does not imply that `Client` can't invoke `hashCode` on `Api.member`. To do this, it has merely to use an accessible qualifying type, which it can do by casting `Api.member` to `Object`. With this change, `Client` compiles and runs successfully:

```
System.out.println(((Object)Api.member).hashCode());
```

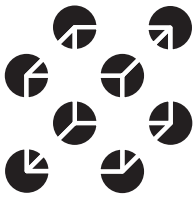
As a practical matter, this problem doesn't arise in ordinary nonreflective access, because API writers use only public types in their public APIs. Even if the problem were to occur, it would manifest itself as a compile-time error, so it would be fixed quickly and easily. Reflective access is another matter. **Although common, the idiom `object.getClass().getMethod("methodName")` is broken and should not be used.** It can easily result in an `IllegalAccessException` at run time, as we saw in the original program.

When accessing a type reflectively, use a `Class` object that represents an accessible type. Going back to our original program, the `hasNext` method is declared in the public type `java.util.Iterator`, so its class object should be used for reflective access. With this change, the `Reflector` program prints `true` as expected:

```
Method m = Iterator.class.getMethod("hasNext");
```

You can avoid this whole category of problem if you use reflection only for instantiation and use interfaces to invoke methods [EJ Item 35]. This use of reflection isolates the class that invokes methods from the class that implements them and provides a high degree of type-safety. It is commonly used in Service Provider Frameworks. This pattern does not solve every problem that demands reflective access, but if it solves your problem, by all means use it.

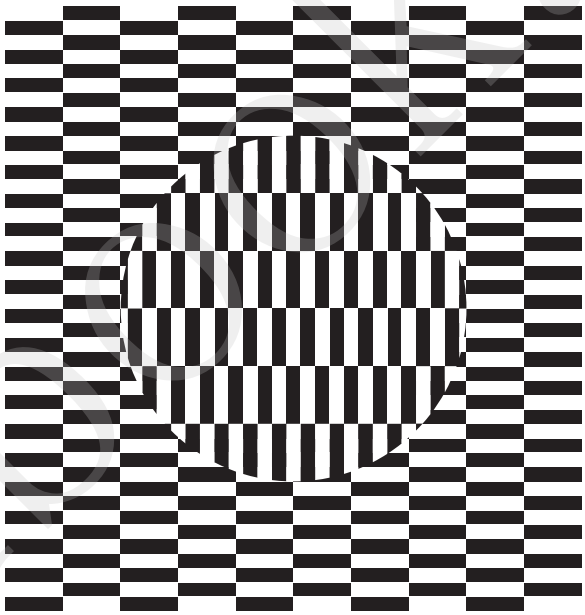
In summary, it is illegal to access a member of a nonpublic type in a different package, even if the member is also declared `public` in a public type. This is true whether the member is accessed normally or reflectively. The problem is likely to manifest itself only in reflective access. For platform designers, the lesson is to make diagnostics as clear as possible. Both the runtime exception and the compiler diagnostic leave something to be desired.



Puzzle 9: It's Absurd, It's a Pain, It's Superclass!

The following program doesn't actually do anything. Worse, it won't compile. Why not? How can you fix it?

```
public class Outer {
    class Inner1 extends Outer {}
    class Inner2 extends Inner1 {}
}
```



Solution 9: It's Absurd, It's a Pain, It's Superclass

This program looks too simple to have anything wrong with it, but if you try to compile it, you get this helpful error message:

```
Outer.java:3: cannot reference this before
              supertype constructor has been called
    class Inner2 extends Inner1 {}
    ^
```

OK, maybe it's not so helpful, but we'll work on that. The problem is that the compiler-generated default constructor for `Inner2` cannot find an appropriate enclosing instance for its `super` invocation. Let's look at the program with the default constructors included explicitly:

```
public class Outer {
    public Outer() {}

    class Inner1 extends Outer {
        public Inner1() {
            super();  // invokes Object() constructor
        }
    }

    class Inner2 extends Inner1 {
        public Inner2() {
            super();  // invokes Inner1() constructor
        }
    }
}
```

Now the error message gives a bit more information:

```
Outer.java:12: cannot reference this before
               supertype constructor has been called
    super(); // invokes Inner1() constructor
    ^
```

Because the superclass of `Inner2` is itself an inner class, an obscure language rule comes into play. As you know, the instantiation of an inner class, such as `Inner1`, requires an enclosing instance to be supplied to the constructor. Normally, it is supplied implicitly, but it can also be supplied explicitly with a *superclass constructor invocation* of the form `expression.super(args)` [JLS 8.8.7].

If the enclosing instance is supplied implicitly, the compiler generates the expression: It uses the `this` reference for the innermost enclosing class of which the superclass is a member. This is, admittedly, quite a mouthful, but it is what the compiler does. In this case, the superclass is `Inner1`. Because the current class, `Inner2`, extends `Outer` indirectly, it has `Inner1` as an inherited member. Therefore, the qualifying expression for the superclass constructor is simply `this`. The compiler supplies an enclosing instance, rewriting `super` to `this.super`. Had we done this ourselves, the compilation error would have made even more sense:

```
Outer.java:12: cannot reference this before
               supertype constructor has been called
    this.super();
    ^
```

Now the problem is clear: The default `Inner2` constructor attempts to reference `this` before the superclass constructor has been called, which is illegal [JLS 8.8.7.1]. The brute-force way to fix this problem is to provide the reasonable enclosing instance explicitly:

```
public class Outer {
    class Inner1 extends Outer { }

    class Inner2 extends Inner1 {
        public Inner2() {
            Outer.this.super();
        }
    }
}
```

This compiles, but it is mind-numbingly complex. There is a better solution: **Whenever you write a member class, ask yourself, Does this class really need an enclosing instance? If the answer is no, make it static.** Inner classes are sometimes useful, but they can easily introduce complications that make a program difficult to understand. They have complex interactions with generics, reflection, and inheritance. If you declare `Inner1` to be `static`, the problem goes away. If you also declare `Inner2` to be `static`, you can actually understand what the program does: a nice bonus indeed.

In summary, it is rarely appropriate for one class to be both an inner class and a subclass of another. More generally, **it is rarely appropriate to extend an inner class; if you must, think long and hard about the enclosing instance.** Also, prefer static nested classes to nonstatic [EJ Item 18]. Most member classes can and should be declared `static`.

Notes on the Illusions

This section contains brief descriptions of the illusions that appear throughout the sampler. The descriptions are grouped loosely by category. Within each category, the order is roughly chronological.

Ambiguous Figures

An *ambiguous figure* is a drawing that can be seen in two or more ways, though not at the same time. One kind of ambiguous figure is the *figure-ground illusion*, which is a drawing that can be seen in two ways, depending on what you perceive as the figure and what you perceive as the background. The drawing on pages 5 can be seen as black arrows pointing outward against a white background, or as white arrows pointing inward against a black background.

Impossible Figures

An *impossible figure* is a two-dimensional perspective drawing of a figure that cannot exist in three dimensions. The *Penrose Triangle* (page 6) is was created by physicist Roger Penrose in 1954.

Geometrical Illusions: Size

The *Jastrow illusion* (page 19) was described by psychologist Joseph Jastrow in 1891. The two shaded areas are identical in size and shape, but most people perceive the top one to be smaller.

Geometrical Illusions: Direction

The *Twisted Cord illusion* (page 3), also known as the *Fraser figure*, was described by psychologist James Fraser in 1908. The letters "CAFE babe" are set straight and true; the perceived tilt is illusory.

The *Cushion illusion* (page 17) was devised by vision scientist and artist Akiyoshi Kitaoka in 1998. This drawing consists solely of rectangles and squares, set straight and true; the curvature is all in your mind. If you find this hard to believe, you can confirm it with a straightedge.

Subjective Contours

A *subjective contour*, also known as an *illusory contour*, is a perceived edge that does not exist. The classic example is found in the *Kanizsa triangle*, devised by Gestalt psychologist Gaetano Kanizsa in 1955. The variant on page 11 is based on a figure devised by Takeo Watanabe and Patrick Cavanagh [Watanabe92]. A white square appears to float above a black plus sign, but the lines that form the white square don't exist. Your mind constructs them from the contours implied by the four irregular pentagons.

Anomalous Motion Illusions

Anomalous motion illusions are drawings whose components appear to move. The *Ouchi illusion* (page 23) was devised by artist Hajime Ouchi in 1973 [Ouchi77]. The inset appears to be on a different plane from the main figure, and to vibrate.

Illusions of Lightness

An illusion of lightness is an image in which we misperceive the lightness (or *luminance*) of some portion of the image. *Logvinenko's illusion* (page 15) was devised by vision scientist Alexander Logvinenko [Logvinenko99]. Strange as it may seem, the horizontal cube faces are all the same shade of gray. If you think the horizontal faces in the first and third rows are lighter than the ones in the second and fourth, cover the vertical faces with a mask and prepare to be surprised.

The drawing on page 15 is based on *Adelson's Illusion of Haze* [Adelson99]. The central diamond, which appears clear, is exactly the same shade of gray as the two flanking diamonds, which appear hazy.

Compound Illusions

Compound illusions combine two or more illusory effects. The *Subjective Necker Cube* (page 22) was devised by Bradley and Petry [Bradley77]. It combines subjective contours with an ambiguous figure. You can see two cubes, one at a time, but the figure contains none. It merely suggests the edges comprising the cubes.

The drawing on pages 16 is based on the *Neon Square illusion* of Marc Albert and Donald Hoffman [Albert99]. This illusion combines subjective contours with the *Neon Spreading effect* [van Tuijl75]. Not only do you perceive a square where none exist, but the square has illusory color: It appears hazy.