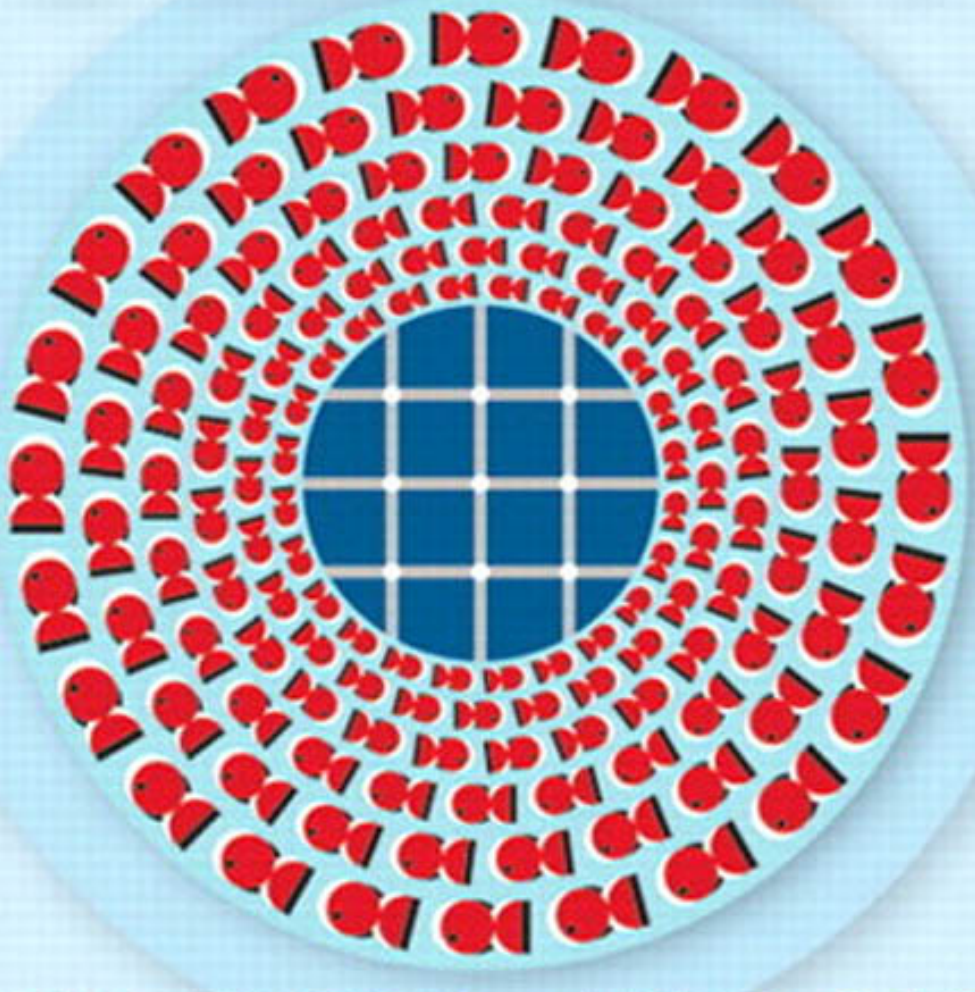
TRAPS, PITFALLS, AND CORNER CASES

JAVA™ PUZZLERS



JOSHUA BLOCH NEAL GAFTER

Java™ Puzzlers

bits, so adding 50 to 2,000,000,000 doesn't influence any bit higher than the sixth from the right. In particular, the seventh and eighth bits from the right are still 0. Promoting this 31-bit `int` to a `float` with 24 bits of precision rounds between the seventh and eighth bits, which simply discards the rightmost seven bits. The rightmost six bits are the only ones on which 2,000,000,000 and 2,000,000,050 differ, so their `float` representations are identical.

The moral of this puzzle is simple: **Do not use floating-point loop indices,** because it can lead to unpredictable behavior. If you need a floating-point value in the body of a loop, take the `int` or `long` loop index and convert it to a `float` or `double`. **You may lose precision when converting an `int` or `long` to a `float` or a `long` to a `double`**, but at least it will not affect the loop itself. **When you use floating-point, use `double` rather than `float`** unless you are certain that `float` provides enough precision *and* you have a compelling performance need to use `float`. The times when it's appropriate to use `float` rather than `double` are few and far between.

The lesson for language designers, yet again, is that silent loss of precision can be very confusing to programmers. See Puzzle 31 for further discussion.

Puzzle 35: Minute by Minute

The following program simulates a simple clock. Its loop variable represents a millisecond counter that goes from 0 to the number of milliseconds in an hour. The body of the loop increments a minute counter at regular intervals. Finally, the program prints the minute counter. What does it print?

```
public class Clock {
    public static void main(String[] args) {
        int minutes = 0;
        for (int ms = 0; ms < 60*60*1000; ms++)
            if (ms % 60*1000 == 0)
                minutes++;
        System.out.println(minutes);
    }
}
```

Solution 35: Minute by Minute

The loop in this program is the standard idiomatic `for` loop. It steps the millisecond counter (`ms`) from 0 to the number of milliseconds in an hour, or 3,600,000, including the former but not the latter. The body of the loop appears to increment the minute counter (`minutes`) each time the millisecond counter is a multiple of 60,000, which is the number of milliseconds in a minute. This happens 3,600,000 / 60,000, or 60 times in the lifetime of the loop, so you might expect the program to print `60`, which is, after all, the number of minutes in an hour. Running the program, however, tells a different story: It prints `60000`. Why does it increment `minutes` so often?

The problem lies in the `boolean` expression `(ms % 60*1000 == 0)`. You might think that this expression is equivalent to `(ms % 60000 == 0)`, but it isn't. The remainder and multiplication operators have the same precedence [JLS 15.17], so the expression `ms % 60*1000` is equivalent to `(ms % 60) * 1000`. This expression is equal to 0 if `(ms % 60)` is 0, so the loop increments `minutes` every 60 iterations. This accounts for the final result being off by a factor of a thousand.

The easiest way to fix the program is to insert a pair of parentheses into the `boolean` expression to force the correct order of evaluation:

```
if (ms % (60 * 1000) == 0)
    minutes++;
```

There is, however, a much better way to fix the program. **Replace all magic numbers with appropriately named constants**:

```
public class Clock {
    private static final int MS_PER_HOUR   = 60 * 60 * 1000;
    private static final int MS_PER_MINUTE = 60 * 1000;
    public static void main(String[] args) {
        int minutes = 0;
        for (int ms = 0; ms < MS_PER_HOUR; ms++)
            if (ms % MS_PER_MINUTE == 0)
                minutes++;
        System.out.println(minutes);
    }
}
```

The expression `ms % 60*1000` in the original program was laid out to fool you into thinking that multiplication has higher precedence than remainder. The compiler, however, ignores this white space, so **never use spacing to express grouping; use parentheses**. Spacing can be deceptive, but parentheses never lie.

5

Exceptional Puzzlers

The puzzles in this chapter concern exceptions and the closely related `try-finally` statement. A word of caution: Puzzle 44 is exceptionally difficult.

Puzzle 36: Indecision

This poor little program can't quite make up its mind. The `decision` method returns `true`. But it also returns `false`. What does it print? Is it even legal?

```
public class Indecisive {
    public static void main(String[] args) {
        System.out.println(decision());
    }

    static boolean decision() {
        try {
            return true;
        } finally {
            return false;
        }
    }
}
```

Solution 36: Indecision

You might think that this program is illegal. After all, the `decision` method can't return both `true` and `false`. If you tried it, you found that it compiles without error and prints `false`. Why?

The reason is that **in a `try-finally` statement, the `finally` block is always executed when control leaves the `try` block** [JLS 14.20.2]. This is true whether the `try` block completes normally or *abruptly.* Abrupt completion of a statement or block occurs when it throws an exception, executes a `break` or `continue` to an enclosing statement, or executes a `return` from the method as in this program. These are called abrupt completions because they prevent the program from executing the next statement in sequence.

When both the `try` block and the `finally` block complete abruptly, the reason for the abrupt completion in the `try` block is discarded, and the whole `try-finally` statement completes abruptly for the same reason as the `finally` block. In this program, the abrupt completion caused by the `return` statement in the `try` block is discarded, and the `try-finally` statement completes abruptly because of the `return` statement in the `finally` block. Simply put, the program `tries` to `return true` but `finally` it `returns false`.

Discarding the reason for abrupt completion is almost never what you want, because the original reason for abrupt completion might be important to the behavior of a program. It is especially difficult to understand the behavior of a program that executes a `break`, `continue`, or `return` statement in a `try` block only to have the statement's behavior vetoed by a `finally` block.

In summary, every `finally` block should complete normally, barring an unchecked exception. **Never exit a `finally` block with a `return`, `break`, `continue`, or `throw`, and never allow a checked exception to propagate out of a `finally` block.**

For language designers, `finally` blocks should perhaps be required to complete normally in the absence of unchecked exceptions. Toward this end, a `try-finally` construct would require that the `finally` block *can complete normally* [JLS 14.21]. A `return`, `break`, or `continue` statement that transfers control out of a `finally` block would be disallowed, as would any statement that could cause a checked exception to propagate out of the `finally` block.

Puzzle 37: Exceptionally Arcane

This puzzle tests your knowledge of the rules for declaring exceptions thrown by methods and caught by `catch` blocks. What does each of the following three programs do? Don't assume that all of them compile:

```
import java.io.IOException;
public class Arcane1 {
    public static void main(String[] args) {
        try {
            System.out.println("Hello world");
        } catch (IOException e) {
            System.out.println("I've never seen println fail!");
        }
    }
}


public class Arcane2 {
    public static void main(String[] args) {
        try {
            // If you have nothing nice to say, say nothing
        } catch (Exception e) {
            System.out.println("This can't happen");
        }
    }
}


interface Type1 {
    void f() throws CloneNotSupportedException;
}
interface Type2 {
    void f() throws InterruptedException;
}
interface Type3 extends Type1, Type2 {
}
public class Arcane3 implements Type3 {
    public void f() {
        System.out.println("Hello world");
    }
    public static void main(String[] args) {
        Type3 t3 = new Arcane3();
        t3.f();
    }
}
```

Solution 37: Exceptionally Arcane

The first program, `Arcane1`, illustrates a basic principle of checked exceptions. It may look as though it should compile: The `try` clause does I/O, and the `catch` clause catches `IOException`. But the program does not compile because the `println` method isn't declared to throw any checked exceptions, and `IOException` is a checked exception. The language specification says that **it is a compile-time error for a `catch` clause to catch a checked exception type *E* if the corresponding `try` clause can't throw an exception of some subtype of *E*** [JLS 11.2.3].

By the same token, the second program, `Arcane2`, may look as though it shouldn't compile, but it does. It compiles because its sole `catch` clause checks for `Exception`. Although the JLS is not terribly clear on this point, **`catch` clauses that catch `Exception` or `Throwable` are legal regardless of the contents of the corresponding `try` clause**. Although `Arcane2` is a legal program, the contents of its `catch` clause will never be executed; the program prints nothing.

The third program, `Arcane3`, also looks as though it shouldn't compile. Method `f` is declared to throw checked exception `CloneNotSupportedException` in interface `Type1` and to throw checked exception `InterruptedException` in interface `Type2`. Interface `Type3` inherits from `Type1` and `Type2`, so it would seem that invoking `f` on an object whose static type is `Type3` could potentially throw either of these exceptions. A method must either catch each checked exception its body can throw, or declare that it throws the exception. The `main` method in `Arcane3` invokes `f` on an object whose static type is `Type3` but does neither of these things for `CloneNotSupportedException` or `InterruptedException`. Why does the program compile?

The flaw in this analysis is the assumption that `Type3.f` can throw either the exception declared on `Type1.f` or the one declared on `Type2.f`. This simply isn't true. Each interface limits the set of checked exceptions that method `f` can throw. **The set of checked exceptions that a method can throw is the intersection of the sets of checked exceptions that it is declared to throw in all applicable types**, not the union [JLS 15.12.2.5]. As a result, the `f` method on an object whose static type is `Type3` can't throw any checked exceptions at all. Therefore, `Arcane3` compiles without error and prints `Hello world`.

In summary, the first program illustrates the basic requirement that `catch` clauses for checked exceptions are permitted only when the corresponding `try` clause can throw the exception in question. The second program illustrates a cor-