An Oddity
by Byte Vector

An odd number can be defined as an integer that is divisible by 2 with a remainder of 1. The expression `n % 2` can be used in many programming languages (C, C++, Java, Python, etc.) to compute the remainder when `n` is divided by 2. So, it seems that this method could be used to determine if a given number, `n` is odd:

```
function isOdd(n):
   if n % 2 = 1 then
      return true
   else
      return false
```

Unfortunately, when `isOdd()` is coded in some programming languages, it returns the wrong answer one quarter of the time.

Why one quarter? Because half of all integer values are negative, and the `isOdd()` function fails for all negative odd values. For example, in Java the function:

```
public static Boolean isOdd(int n) {
   return n % 2 == 1;
} // isOdd()
```

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 [1]. This identity makes perfect sense, but in combination with Java's truncating integer division operator (`/`), it implies that **when the remainder operations returns a nonzero result, it has the same sign as its left operand**. The behavior is similar in C and Go.

This instance of `isOdd()` 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 kind of division, Java's remainder operation is perfectly matched to its integer division operation, which discards the fractional part of its result.

When `n` is a negative odd number, `n % 2` is equal to `-1` rather than `1`, so the `isOdd()` function incorrectly returns `false`. To prevent this sort of surprise, **test that your functions behave properly when passed negative, zero, and positive values for each numerical parameter**. Surprisingly, The `%` operator in Python yields integers with the same sign as the divisor. Thus

```
-7 % 3 is 2
```

Since the divisor (`3`) is positive. But

```
7 % -3 is -2
```

The problem is easy to fix for most implementations (including Python!). Simply do the comparison of `n % 2` to `0` rather than `1`, and reverse the sense of comparison:

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

If you are using the `isOdd()` function 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 n) {
   return (n & 1) != 0;
} // isOdd()
```

The second version may run much faster than the first, depending on what computer you are using, and is unlikely to run any slower. As a general rule, the divide and remainder operations are slow compared to other arithmetic and logical operations. **It is a bad idea to optimize prematurely**, but in this case, the faster version is almost 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 the remainder operator is obvious when its operands are nonnegative, but it isn't so obvious when one or both operands are negative.

References

[1] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman. *The Java Language Specification, Java SE 17 Edition.* Page 652. Oracle America, Inc. Available as https://docs.oracle.com/javase/specs/jls/se17/jls17.pdf As seen on January 21, 2022.