This is brilliant. How about you identify an incorrect roman numeral. It can also be nasty and see behaviour of my code

ChatGPT said:

Let's do it 😈

I'll pick this invalid numeral:

MIM

A lot of naive converters accidentally treat this as **1999**, but it is **not** a valid Roman numeral, because:

- I can only be subtractive in **IV (4)** or **IX (9)**
- It may *never* subtract from C (100) or M (1000)
- So the pattern IM is illegal, and therefore MIM is illegal too

Now: **what does *your* code do with MIM?**
Let's walk through it in terms of your variables and rules.

---

1. Main: building posValues

Input: MIM

Your main builds:

index: 0 1 2

char: M I M

value: 1000 1 1000

So posValues becomes:

posValues[0] = {0, 1000} // M

posValues[1] = {6, 1} // I

posValues[2] = {0, 1000} // M

(0,6 are just the indices in acceptedNumerals).

processedConversion is initialised to all **0** (nothing marked as "already processed" yet).

---

2. First phase: subtractive pairs (IV, IX, XL, XC, CD, CM)

Your subtractive list:

String[] subtractiveNumerals = {"IV","IX","XL","XC","CD","CM"};

For MIM:

- indexOf("IV") → -1
- indexOf("IX") → -1
- indexOf("XL") → -1
- indexOf("XC") → -1
- indexOf("CD") → -1
- indexOf("CM") → -1

So in this whole big block:

- twoSubtractiveNotations stays **false**
- singleSubtractiveNotations stays **false**
- runningTotal stays **0**
- rule4Single stays **0**
- processedConversion stays **[0, 0, 0]**
- illegalSubtractive still **false** at this stage

So the *subtractive phase* doesn't see anything special in MIM.

---

3. Second phase: scanning left-to-right with rules 2, 5, 6, 7

This is your big loop:

for (int m=1; m<posValues.length; m++)

Step m = 1 (looking at index 1, the middle I)

We are comparing value at index 1 (1) with index 0 (1000):

- posValues[1][1] == posValues[0][1]? → 1 == 1000 → **no**, so no "same value in a row" logic.
- posValues[1][1] < posValues[0][1]? → 1 < 1000 → **yes**
  → This hits your **Rule 5-style** "smaller after larger" case:

 if (processedConversion[m-1] != 1) {

  rule5total = posValues[m][1] + posValues[m-1][1]; // 1 + 1000 = 1001

  rule5 = rule5total + rule5; // rule5 = 1001

```
  rule5State = true;
}
```

So after m=1:

- rule5 = 1001
- rule7 not touched yet
- zeroIndexCount and temp1 will be incremented later in this iteration

 Now the **“look ahead / subtractive” block** with zeroIndexCount:

At this point:

zeroIndexCount = 0;

You check:

```
if (posValues[zeroIndexCount][1] < posValues[zeroIndexCount+1][1])
```

That is:

 1000 < 1 ? → false

So **no** subtractive check is triggered yet, and no IplacedIncorrect is set here.

 You also check Rule 7 (smaller between two larger ones):

```
if (m!=posValues.length && posValues.length>2) {
  if (processedConversion[m]!=1) {
    if (m!=posValues.length-1) {
      if (posValues[m][1]<posValues[m-1][1] && posValues[m][1]<posValues[m+1][1]) {
        rule7total = posValues[m+1][1] - posValues[m][1];
      }
    }
  }
}
```

For m = 1:

- m != 3 → true
- processedConversion[1] != 1 → true (still 0)
- m != 2 → true

- Check: posValues[1][1] < posValues[0][1] && posValues[1][1] < posValues[2][1] → 1 < 1000 && 1 < 1000 → **true**

So:

- rule7total = 1000 - 1 = 999;
- rule7 = 999;
- That nicely encodes “I is sandwiched between two M’s”.
- End of m=1:
    - rule5 = 1001
    - rule7 = 999
    - IplacedIncorrect = false
    - illegalSubtractive = false
    - zeroIndexCount is now 1.

Step m = 2 (final M at index 2)

Now we’re inside the loop with m=2:

- processedConversion[2] != 1 → still **true**
- posValues[2][1] == posValues[1][1]? → 1000 == 1 → **no**
- posValues[2][1] < posValues[1][1]? → 1000 < 1 → **no**

Now comes the **really important part**:

```
if (posValues[zeroIndexCount][1]<posValues[zeroIndexCount+1][1]) {
   // ...
}
```

Right now zeroIndexCount = 1, so we’re comparing index 1 (I) and index 2 (M):

posValues[1][1] = 1

posValues[2][1] = 1000

1 < 1000 ? → true

So we enter this block.

Inside it:

3.1 Illegal subtractive: I before M

```
if (posValues[zeroIndexCount][1]==1 && posValues[zeroIndexCount+1][1]>10) {
   System.out.println("Roman Numeral I(1) ... can not precede: " + convertNumber(...));
   IplacedIncorrect=true;
}
```

For MIM:

- posValues[1][1] == 1 → true
- posValues[2][1] > 10 → 1000 > 10 → true

So your code will print something like:

Roman Numeral I(1) (numeral index: 1) can not precede: M(1000)

and sets:

IplacedIncorrect = true;

This is exactly enforcing:

I can only subtract from V or X, not from M.

3.2 The generic subtractive-or-illegal branch

Next:

```
if (posValues[zeroIndexCount][1]!=5 && posValues[zeroIndexCount][1]!=50
   && posValues[zeroIndexCount][1]!=500 && !IplacedIncorrect)
{
   // treat as legal subtractive pair, update rule6
}
else
{
   System.out.println("Illegal subtractive notation found with V, D or L");
   rule5=0;
   illegalSubtractive=true;
}
```

In our case:

- posValues[1][1] is 1 (so it's not 5, 50, or 500)
- **but** !IplacedIncorrect is **false** (we just set it to true)

So the if fails and we go into the else:

Illegal subtractive notation found with V, D or L

...and we get:

rule5 = 0; // your earlier 1001 is wiped

illegalSubtractive = true;

The message text mentions V, D or L, but in this case it's really about I used illegally – it's just reusing the same error branch.

So at this point we have:

- IplacedIncorrect = true
- illegalSubtractive = true
- rule5 = 0
- rule7 = 999 (from that "sandwiched I" earlier)
- rule6 = 0
- threeInRow = 0
- processedConversion still [0,0,0] (no valid subtractive pairs were recorded)
- unTotal = 1000 + 1 + 1000 = 2001 (sum of all unmarked numerals)

---

4. Validity gate: does it ever claim MIM is valid?

Your "new logic" only treats a numeral as valid if **all** of these are false:

```
if (!illegalSubtractive && !IplacedIncorrect && !rule3Fail && !illegalFourInARowState)
{
  ...
  if (validNumber) {
    System.out.println("... is a VALID roman numeral");
    ...
```

```
    }
}
```

For MIM:

- illegalSubtractive == true
- IplacedIncorrect == true

So that entire block is **skipped**.

Crucially:

- validNumber stays at its default false
- No “VALID roman numeral” message is printed
- The “runningTotal + unTotal” result is **never** reported as a legitimate decimal value

---

5. Final outcome

At the end you have:

```
if (!validNumber) {
    System.out.println("\n********************* GRAND TOTAL*************");
    System.out.println(numeralsToString + " is an INVALID roman numeral");
    System.out.println("Address the issues outputted for a valid Roman numeral");
    System.out.println("***********************************");
}
```

So for MIM, your program’s behaviour is:

1. Spot that I is in an illegal subtractive position before M
2. Set both IplacedIncorrect **and** illegalSubtractive
3. Skip the “valid” reporting path entirely
4. Print that **MIM is an INVALID roman numeral**, with explanatory rule messages beforehand
   (notably the “I(1) cannot precede M(1000)” line)

Even though internally you have numbers like:

- unTotal = 2001

- rule7 = 999
- rule5 = 0 (after being reset)

none of that is treated as a valid decimal conversion, because the flags correctly mark the numeral as illegal.
