Aula 2 - Desafio (médio)

Docupedia Export

Author:Silio Leonardo (CtP/ETS)
Date:29-Apr-2024 15:58

Table of Contents

1 Operadores: Uma visão em binário 4
2 Exercícios Propostos 7
3 Estruturas Básicas 8
4 Funções 11
5 Desafio 12

- Operadores: Uma visão em binário
- Exercícios Propostos
- Estruturas Básicas
- Funções
- Desafio

1 Operadores: Uma visão em binário

A compreensão de números binários é importante para compreender algumas operações, bem como permitir que certas coisas sejam feitas mais eficientemente.
Desta forma, vamos ver algumas das operações mostradas no final da Aula 1 olhando para números binários.
Primariamente, precisamos entender bem o que são números binários.
Um número binário é uma sequência de bits, que são nada mais, nada menos, que 0's e 1's. Vamos usar bastante a noção de byte daqui para frente, que é o conjunto de 8 bits, indo de 0 (00000000) a 255 (11111111) bem como o tipo definido em C#.
Assim como em decimal, que tem os digitos de 0 a 9, ao olhar o próximo número apenas somamos 1 ao digito mais a direita:
34 -> 35
O sucessor de 34 é 35, pois 4 + 1 é 5. Note que quando passamos do último digito (9) voltamos para o primeiro (0) e somamos 1 a casa da esquerda:
49 -> 50
O sucessor de 49 é 50, pois 9 + 1 = 10, assim o 9 torna-se 0 e somamos 1 ao 4 que torna-se 5. Da mesma forma:
9199 -> 9200
Em binário, os mesmos mecânismos acontecem. Porém, só temos 2 digitos, assim:
000 -> 001 001 -> 010 010 -> 011 100 -> 101 101 -> 110 110 -> 111
Pela tabela podemos ver como isso ocorre:

Binário	Decimal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Note que existem formas mais fáceis de realizar conversões binário-decimal após compreender seu funcionamento. Então vejamos elas: Vamos tentar converter 10100101 de binário para decimal. Basta ignorar os 0's e para cada 1 você soma uma potência de 2. Essa potência deve ser correspondente a posição de cada 1 no binário. Assim a contribuição dos 0's é zero e a dos 1's depende da sua posição no binário. Observe:

digito	**7°**	**6º**	**5°**	**4º**	**3°**	**2º**	**1°**	**0º**	**resultado**
binário	1	0	1	0	0	1	0	1	
potência	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
resultado	128	64	32	16	8	4	2	1	
contribuição	128	0	32	0	0	4	0	1	165

Da mesma forma, podemos converter de um decimal para um binário apenas buscando os bits onde a conversão de correta. Interessantemente, só existe uma combinação de 0's e 1's para cada decimal. Um truque para realizar essa operação é ler da esquerda para direita, assim se a soma do bit não extrapolar o decimal, consideramos o mesmo como 1. Observe a conversão de 202 para binário:

passo	**binário**	**conversão**	**é maior que 202**	**ação**
1	1_____	128	não	próximo bit
2	11_____	192	não	próximo bit
3	111_____	224	sim	reverte
4	110_____	192	não	próximo bit
5	1101____	208	sim	reverte
6	1100____	192	não	próximo bit
7	11001___	200	não	próximo bit

passo	binário	conversão	é maior que 202	ação
8	110011__	204	sim	reverte
9	110010__	200	não	próximo bit
10 R:	1100101_ 11001010	202 202	é igual fim	termina, próximos bits iguala a 0

2 Exercícios Propostos

a. Converta os seguintes valores:
- a) 17 para binário
- b) 102 para binário
- c) 254 para binário
- d) 1111_1100 para decimal
- e) 1000_0000 para decimal
- e) 1100_0001 para decimal
- g) 2641 para binário
- h) 1100_1010_1100 para decimal

Podemos observar agora algumas das operações do C# com um pouco de distinção e pensar nos números binários a partir disto. Vamos começar com o Shift ou deslocamento binário:
00000001 << 1 = 00000010
Basicamente, temos o deslocamento uma casa a esquerda. Outro exemplo:
01010101 << 1 = 10101010
Isso ocorre a todos os bits. Note que um deslocamento maior que 1 pode ocorrer:
00000001 << 7 = 10000000
Outra noção importante é que se um número sai para fora dos bits (depende se for um byte, short, int ou long) ele simplesmente desaparece:
11110000 << 2 = 11000000
O Shift também pode ser aplicado a direita:
11110000 >> 2 = 00111100
Ele também segue as mesmas regras que o outro shift:
11111111 >> 3 = 00011111
Você ainda pode escrever em código C#:

```
byte b = 4;
b <<= 2;
```

Da mesma forma como se usa +=, -=, *=, /= em C# e outras linguagens.
Você também pode usar operações lógicas fazendo operações bit-a-bit:
11100000 | 00000111 = 11100111 (Ou, aplicado bit-a-bit)
11111000 & 00011111 = 00011000 (E, aplicado bit-a-bit)
11110000 ^ 11001100 = 00111100 (Ou exclusivo, aplicado bit-a-bit)
~00001111 = 11110000 (Não, aplicado bit-a-bit)
É importante salientar que os números com sinal são representados na forma complemento de 2. Isso significa, entre outras coisas, que o primeiro bit representa se o número é negativo ou não. Logo, caso alguma operação que você realizar jogue um 1 no primeiro bit o número ficará negativo.
Outro ponto rápido: existe uma diferença entre && e || de & e |. A repetição do símbolo denota circuito-curto, o que significa que a depender do primeiro valor, o segundo não é considerado. Assim true || false não considera a segunda parte da expressão enquanto o circuito-completo true | false considera a expressão como um todo. Com isso &&/|| não pode ser usado em tipos binários, mas pode evitar cálculos desnecessários e é bom ser usado em condicionais.

3 Estruturas Básicas

Como você deve ter aprendido em outras linguagens, para programar é necessário de algumas estruturas para controlar o fluxo de execução. Agora, vamos ver como produzi-las em C#:

- if

```java
int idade = scanner.nextInt();
if (idade > 17)
{
    System.out.println("É maior de idade.");
}
else if (idade > 15)
{
    System.out.println("Não é maior de idade. Mas em alguns países já pode dirigir");
}
else
{
    System.out.println("Menor de idade.");
}
```

Então, caso a idade digitada seja 18 para cima a condição no primeiro escopo é executada. Note que a segunda condição também seria verdadeira, porém, apenas uma cláusula é executada por vez. Caso nem o 'if', nem o 'else if' seja executado, executamos o 'else'.

- switch

```java
int idade = scanner.nextInt();
switch (idade)
{
    case 18:
        System.out.println("É maior de idade.");
    break;

    case 19:
        System.out.println("Está ficando velho!");
    break;

    case 16:
    case 17:
        System.out.println("Não é maior de idade. Mas em alguns países já pode dirigir.");
    break;
```

```
16
17      default:
18          System.out.println("Menor de idade.");
19      break;
20  }
```

O switch usa uma forma diferente de execução, tornando-o mais rápido a depender da situação. Se existem poucos intervalos e muitas opções distintas, o switch é uma boa opção. Você precisa fechar todos os casos com break, return ou goto. O mais comum é se utilizar break, mas como pode observar, você pode usar goto para que 2 casos diferentes executem. Você também pode usar 2 cases seguidos para indicar o mesmo comportamento para diferentes valores da variável.

- while

```
1  System.out.println("Fatorial de qual valor você irá querer?");
2  int n = scanner.nextInt();
3
4  int fatorial = 1;
5  while (n > 1)
6  {
7      fatorial *= n;
8      n--;
9  }
```

- do while

```
1  int numeroSecreto = 540;
2  int numero = 0
3
4  do
5  {
6      System.out.println("Tente adivinhar o número secreto...");
7      numero = scanner.nextInt();
8
9      if (numero > numeroSecreto)
10         System.out.println("O numero secreto é menor");
11     else if (numero < numeroSecreto)
12         System.out.println("O numero secreto é maior");
13
14 } while (numero != numeroSecreto);
```

No While e Do...While temos as mais básicas estruturas de repetição. O bloco é executado apenas enquanto a condição seja verdadeira. O que muda é apenas o momento em que a condição é considerada.

- for

```
int[] vetor = new int[50];

System.out.println("Digite 50 valores");
for (int i = 0; i < vetor.Lenth; i++)
{
    vetor[i] = scanner.nextInt();
}
```

Também temos o For que é ótimo para acessar vetores pois o mesmo tem inicialização, condição e incremento no final de cada loop. É bom lembrar que também temos coisas como break, que interrompe um loop e continue, que pula para o próximo loop.

4 Funções

A partir de agora, vamos parar de fazer tantas operações fora de funções. Funções são uma das estruturas mais básicas e importantes dentro da programação e por isso vamos aprender a utilizá-las em Java. A estrutura básica é:
retorno nome(parâmetros) { implementação }
Em C# você não precisa declarar a função antes de usá-la (ela pode estar depois no código), mas você precisa seguir algumas regras:

- Se a função pede algum retorno, você deve retorná-lo, caso contrário você terá um erro de compilação.
- Você deve garantir que todos os caminhos retornam algo.
- Cada parâmetro é como uma variável que deve ser enviado na chamada, a não ser que seja opcional.
- Se você declarar uma parâmetro com mesmo nome de outra variável no programa, ao usar a variável estará usando a declarada dentro da função.

A seguir exemplos:

```
1  int value = modulo(-3);
2
3  static int modulo(int i)
4  {
5      if (i < 0)
6          return -i;
7      return i;
8  }
```

```
1  int n = 14;
2  System.out.println("O resultado é: " + fatorial(n));
3
4  static int fatorial(int n)
5  {
6      if (n < 2)
7          return 1;
8
9      return n * fatorial(n - 1);
10 }
```

5 Desafio

Agora que já aprendemos o básico do Java e já podemos escrever programas simples vamos ao primeiro desafio: Um compressor de dados com perdas.
A ideia é simples: Em arquivos como imagens e vídeos a perda de informação durante a compressão é aceitável e até desejado. Isso advém do fato de que é pouco perceptível aos nossos olhos.
A ideia é eliminar parte dos dados e reduzir o espaço gasto por eles. Ao ver uma cor RGB (255, 255, 255) que é branco podemos perceber que em binário temos:
11111111, 11111111, 11111111
Ao jogarmos fora os 4 bits menos significativos (da direita), poderíamos guardar apenas o 4 dígitos que mais impactam no cálculo do binário:
1111, 1111, 1111
Assim, criamos um compressor de 50%. Ao tentar voltar para 8 bits para poder rever a cor podemos preencher os bits que perdemos com valores como 0:
11110000, 11110000, 11110000
Que é o RGB (240, 240, 240). Procure no goolge por 'rgb(240, 240, 240)' e veja a pequena diferença para o branco 'rgb(255, 255, 255)'. E perceba que este é o pior caso possível, onde mais se tem perda de informação. Se o RGB já fosse (240, 240, 240), por exemplo, não teríamos perda alguma.
Seu trabalho é fazer uma função que receba um vetor de byte que será compactada desta forma. Note que precisará retornar um vetor com a metade do tamanho e pior: A cada 2 bytes você deve colocar 4 bits de cada um e um único bit. Exemplo:
Original: 11111111, 00000000, 10101111, 11111010 Compactado: 1111, 0000, 1010, 1010
Resultado: 11110000, 1010101

```java
public class Main
{
    public static void main(String[] args)
    {
        int[] array = new int[] {
            244, 232, 241, 123,
            232, 064, 111, 140
        };

        int[] result = compact(array);
        for (int i = 0; i < result.length; i++)
            System.out.printf("%d ", result[i]);
    }

    static int[] compact(int[] originalData)
    {
        // Implemente aqui
        return originalData;
    }

    static int[] decompact(int[] originalData)
```

```
22      {
23          // Implemente aqui
24          return originalData;
25      }
26  }
```

Desafio Opcional: Faça uma compactação apenas para 3 bytes, ao invés de 4.
Depois implemente a função de descompactação que retorna os dados aos dados originais, preenchendo os bits que foram perdidos com 0.