Repetição de Comandos

Material didáctico elaborado pelas diferentes equipas de
Introdução à Programação

Luís Caires (Responsável), Armanda Rodrigues, António Ravara, Carla Ferreira, Fernanda Barbosa, Fernando Birra, Jácome Cunha, João Araújo, Miguel Goulão, Miguel Pessoa Monteiro, e Sofia Cavaco.

Mestrado Integrado em Engenharia Informática FCT UNL

A nossa biblioteca `Mathematics`

- De modo a praticar os ciclos vamos fazer uma biblioteca com operações matemáticas diversas.
- Para isso vamos criar uma classe `Mathematics`, onde todos os métodos são `static` (estes métodos denominam-se “métodos de classe”).
- Note que esta classe não tem construtores, pois nunca vamos criar objectos desta classe. É como a classe `Math` do Java.
- De igual modo, esta classe também não tem variáveis de instância. Repare, uma variável de instância é uma variável que pertence a uma instância, ou seja, a um objecto. Se nunca vamos criar um objecto desta classe, não faria sentido criar variáveis de instância.





Números Primos

Números Primos

- Um número inteiro positivo é primo se tiver apenas dois divisores distintos
- Por exemplo, 3, 11 e 17 são números primos:
  - 3 é divisível apenas por 1 e 3
  - 11 é divisível apenas por 1 e 11
  - 17 é divisível apenas por 1 e 17

Números Primos

- Crie um método na classe “Mathematics” que indica se um dado número inteiro positivo é primo

```
public static boolean isPrime(int n)
```

Indica se o número n é primo

pre: n >= 1

- Teste o método implementado, num programa principal e verifique que o método `isPrime` se comporta como o esperado. Por exemplo, experimente fazer um programa principal em que o utilizador vai dando valores inteiros positivos, que vão sendo testados. Para terminar, o utilizador dá o número `0`.

Números Primos

- O método implica a divisão inteira do número *n* >=1 por todos os números inteiros menores que *n* e maiores que um e a verificação do resto da operação

  (desde que n > 1; se n = 1, não é primo)

- Note que a operação de teste de divisibilidade deve ser repetida um certo número de vezes
  - O número de repetições depende do valor *n*
  - É necessário utilizar um comando de repetição ou iteração de comandos

Números Primos

```
n % 2     != 0
n % 3     != 0
...
n % n-2   != 0
n % n-1   != 0
```




Números primos

Intervalo de variação de v:
1 < v < =n

```
n % v != 0
n % v != 0
...
n % v != 0
n % v != 0
```

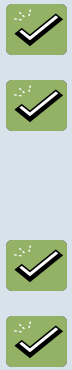



Em cada passo

```
v++;
```

Instrução `while`

```
n % v  != 0
n % v  != 0
...
n % v  != 0
n % v  != 0
```



Intervalo de variação de v:
1 < v <= n

```
while (condicao)
    instrucao
```

Em cada passo

```
v++;
```

Esta condição é suficiente?

```
while ( n % v != 0 )
        v++;
```

Instrução `while`

```
n % v  != 0
n % v  != 0
...
n % v  != 0
n % v  != 0
```

Em cada passo

```
v++;
```

Intervalo de variação de v:
1 < v <= n

```
while (condicao)
    instrucao
```

Esta condição é suficiente?

Sim, se `1<v<=n`

```
while ( n % v != 0 )
        v++;
```

Instrução `while`

```
n % v  != 0
n % v  != 0
...
n % v  != 0
n % v  != 0
```

Intervalo de variação de v:
1 < v < n

```
while (condicao)
    instrucao
```

É preciso guarantir que se avalia o ciclo só quando n > 1

Em cada passo

```
v++;
```

```
if (n>1)
    while ((n % v) != 0)
        v++;
```

Números primos

```
/* Pre: n >= 1  */
public static boolean isPrime(int n){
    int v = 2;
    if (n>1)
        while ((n % v) != 0)
                v++;
    return (v==n);
}
```

Números primos

```
public static boolean isPrime(int n){
    int v = n-1;
    if (n>1)
            while ((v > 1) && ( (n % v) != 0))
                v--;
    return (v==1);
}
```

Também podíamos ter começado pelo limite superior do intervalo de variação. Mas nesse caso o algoritmo seria mais lento a encontrar divisores que tornem a condição do ciclo `false`.

Análise de casos

- `n <= 0` – não faz sentido pois o método só deve ser aplicado a números inteiros positivos (estes casos são excluídos pela pré-condição definida)
- `n = 1 → v = 2` – O ciclo não é executado pois `v` não se encontra dentro dos valores a serem testados como divisores
  - Desta forma, porque `v = 2 != 1`, o método devolve `false` → 1 não é primo
- `n = 2 → v = 2` – O ciclo não é executado pois `v%n = 0`
  - `v == 2`, devolve `true` → 2 é primo
- `n > 2`
  - Se o número for primo, o ciclo é executado até `v == n`, o método devolve `true` → o número é primo
  - Se o número não for primo, o ciclo termina quando `v` igual ao 1º divisor de `n`, assim `v!=n,` o método devolve `false` → o número não é primo

Números primos

Para saber se n é primo vale a pena testar todos os naturais entre 2 e n-1?

Não vale a pena testar números maiores que n/2 !
Com excepção do próprio n, nenhum é divisor de n.

Números primos

```
// pre n>=1
public static boolean isPrime(int n){
    int v = 2;
    while ((v <= Math.sqrt(n)) && ( (n % v) != 0)) {
            v++;
    }
    return (v> Math.sqrt(n) && n > 1);
}
```

Pensando ainda melhor, nem sequer vale a pena testar números maiores que a raiz quadrada de n. Se n for divisível por um número p, então n = p x q Se q for menor que p, n já terá sido detectado como sendo divisível por q, ou por um divisor de q. Podemos fazer um raciocínio semelhante sobre p. Assim, no máximo, p = q = raiz quadrada de n. Além disso, n tem de ser > 1.

Class Main

```
import java.util.Scanner;

public class Main {

  public static void main(String[] args) {
     Scanner in = new Scanner(System.in);
     int number;
     System.out.print("Introduza um numero para verificar se e primo: ");
     number = in.nextInt();
     in.nextLine();
     if (Mathematics.isPrime(number))
          System.out.println("E primo.");
     else
          System.out.println("Nao e primo.");
     System.out.println("Adeus!!!");
     in.close();
  }
}
```

Class `Main`

```
import java.util.Scanner;

public class Main {

  public static void main(String[] args) {
     Scanner in = new Scanner(System.in);
     int number;
     System.out.print("Introduza um numero para verificar se e primo: ");
     number = in.nextInt();
     in.nextLine();
     if (Mathematics.isPrime(number))
          System.out.println("E primo.");
     else
          System.out.println("Nao e primo.");
     System.out.println("Adeus!!!");
     in.close();
  }
}
```

Então e se quisermos testar vários números?

Class Main

```
import java.util.Scanner;

public class Main {

  public static void main(String[] args) {
     Scanner in = new Scanner(System.in);
     int number;
     System.out.print("Introduza um numero para verificar se e primo: ");
     number = in.nextInt();
     in.nextLine();
     if (Mathematics.isPrime(number))
          System.out.println("E primo.");
     else
          System.out.println("Nao e primo.");
     System.out.println("Adeus!!!");
     in.close();
  }
}
```

Vamos utilizar o 0 (que não é positivo) como um valor sentinela que determina o fim do input

Class Main

```
import java.util.Scanner;


public class Main {
  public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      int number;
      System.out.print("Introduza um numero positivo para verificar se e primo (0 para terminar):");
      number = in.nextInt();
      in.nextLine();
      while (number > 0){
            if (Mathematics.isPrime(number))
                  System.out.println("E primo.");
            else
                  System.out.println("Nao e primo.");
      }
      System.out.println("Adeus!!!");
      in.close();
  }
}
```

Estas três linhas têm de ser repetidas em cada passo para:

-Informar o utilizador sobre o input;

-Guardar o valor introduzido.

Class Main

```
import java.util.Scanner;


public class Main {
  public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      int number;
      System.out.print("Introduza um número positivo para verificar se e primo (0 para terminar):)";
      number = in.nextInt();
      in.nextLine();
      while (number > 0){
            if (Mathematics.isPrime(number))
                        System.out.println("E primo.");
                  else
                        System.out.println("Nao e primo.");
          System.out.print("Introduza um numero positivo para verificar se e primo (0 para terminar):)";
          number = in.nextInt();
          in.nextLine();
      }
      System.out.println("Adeus!!!");
      in.close();
   }
}
```

Estas três linhas têm de ser repetidas em cada passo para:

-Informar o utilizador sobre o input;

-Guardar o valor introduzido.

Class Main
Como eliminar esta repetição?

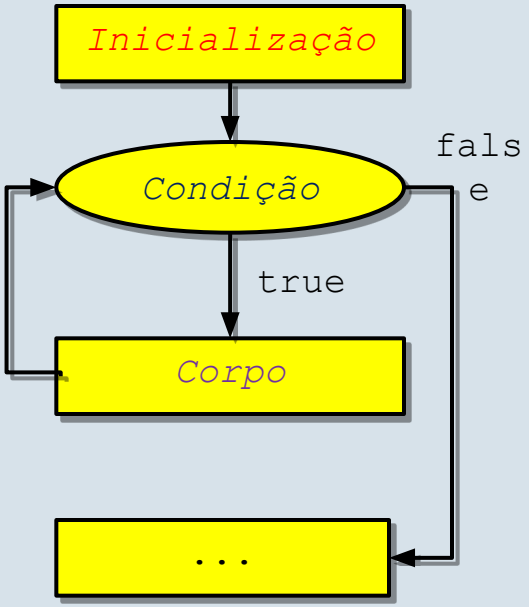
```
import java.util.Scanner;


public class Main {
  public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      int number;
      System.out.print("Introduza um número positivo para verificar se e primo (0 para terminar):");
      number = in.nextInt();
      in.nextLine();
      while (number > 0){
            if (Mathematics.isPrime(number))
                        System.out.println("E primo.");
                  else
                        System.out.println("Nao e primo.");
            System.out.print("Introduza um numero positivo para verificar se e primo (0 para terminar):");
            number = in.nextInt();
            in.nextLine();
      }
      System.out.println("Adeus!!!");
      in.close();
   }
}
```

O ciclo while...do executa-se 0 ou mais vezes. Neste caso, queremos que estes testes aos números primos se executem 1 ou mais vezes. O java inclui a instrução composta do...while, precisamente com uma semântica de execução de 1 ou mais iterações!

Ciclo `while...do` *vs.* `do...while`

`while... do`

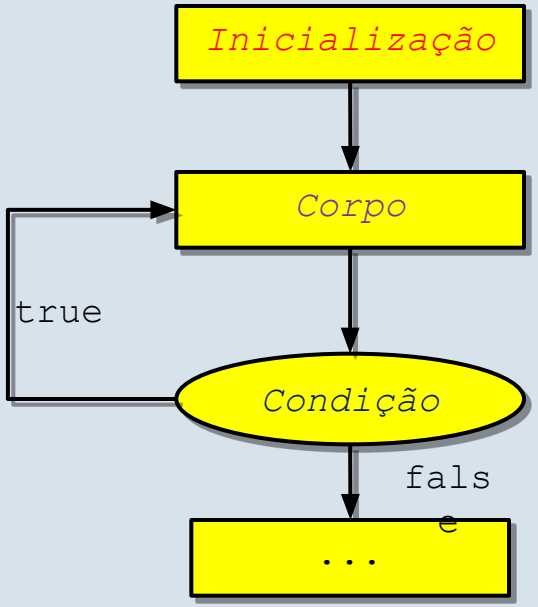
Executa 0 ou mais vezes




```
Inicialização ;
while ( Condição ) {
  Corpo;
}
```

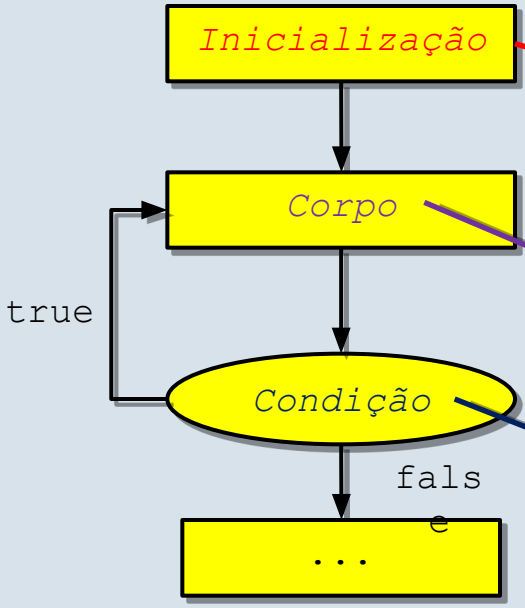
`do...while`

Executa 1 ou mais vezes




```
Inicialização ;
do {
  Corpo;
} while ( Condição )
```

O ciclo do...while




```
Inicialização ;
do {
   Corpo;
} while ( Condição )
```

```
public static void main(String[] args) {
    Scanner in=new Scanner(System.in);


    int number;

    do {
        System.out.print(...);
        number=in.nextInt()
        in.nextLine();
        if (number > 0)
        if (Mathematics.isPrime(number))
            System.out.println("E primo.");
        else
            System.out.println("Nao e primo.");
    } while ( number > 0 );
    System.out.println("Adeus!!!");
    in.close();
}
```

Class `Main` com um ciclo `do... while`

```
import java.util.Scanner;


public class Main {


   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      int number;
      do {
            System.out.print("Introduza um numero positivo para verificar se e primo (0 para terminar):");
            number = in.nextInt();
            if (number > 0)
                  if (Mathematics.isPrime(pr))
                        System.out.println("E primo.");
                  else
                        System.out.println("Nao e primo.");
      } while (number > 0);
      System.out.println("Adeus!!!");
      in.close();
   }
}
```

Evitámos a repetição da pergunta e da leitura de dados mas temos que testar a pré-condição

Testando vários métodos

- No resto deste bloco pretendemos testar vários métodos:
  - Uma hipótese seria criar um programa para testar cada um deles, como no slide anterior (mais repetitiva)
  - Outra hipótese é criar vários métodos auxiliares de teste, um para cada operação a testar:
    - Cada método auxiliar pode ser declarado privado
    - Pode comentar os métodos que não deseja testar
    - O corpo do método auxiliar é semelhante ao que antes colocaria no método main

```
public class Main {
  public static void main(String[] args) {
    Main.testePrimos();
  }

  private static void testePrimos() { ... }

} // Fim da classe Main
```

Aqui ficaria um corpo semelhante ao do `main` no slide anterior

Números Perfeitos

Números Perfeitos

- Um número perfeito é um inteiro positivo cuja soma dos seus divisores próprios é o próprio número
- Os divisores próprios de um número são os divisores que são diferentes do próprio número
- Por exemplo, 6 é um número perfeito:

  1, 2 e 3 são os divisores próprios de 6

  6 = 1 + 2 + 3

Números Perfeitos

- Crie um método na classe `Mathematics` que determina se um número `n` é um número perfeito

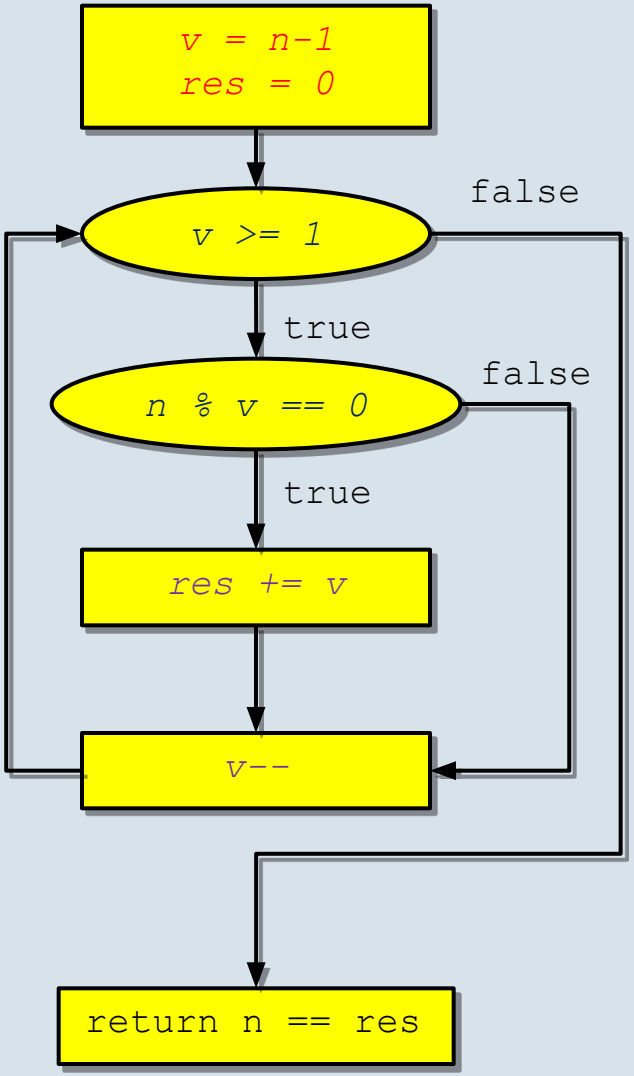
```
public static boolean isPerfectNumber(int n)
```

Indica se o número n é um número perfeito.

pre: n >= 0

- Crie depois um método auxiliar no seu programa principal que imprima todos os números perfeitos menores que 10000.

Números perfeitos



Objectivo: testar se a soma dos divisores próprios de dado natural n não nulo é igual a esse natural

Há dois casos especiais - 0 e 1 - que não têm divisores próprios (logo a soma dos seus divisores é 0).

Algoritmo (para n>1):

usando uma variável auxiliar v a variar de n-1 a 1, acumular o valor da variável no resultado se esse valor for divisor de n. Finalmente, comparar o valor acumulado com n.

Inicialização: v = n-1; res = 0

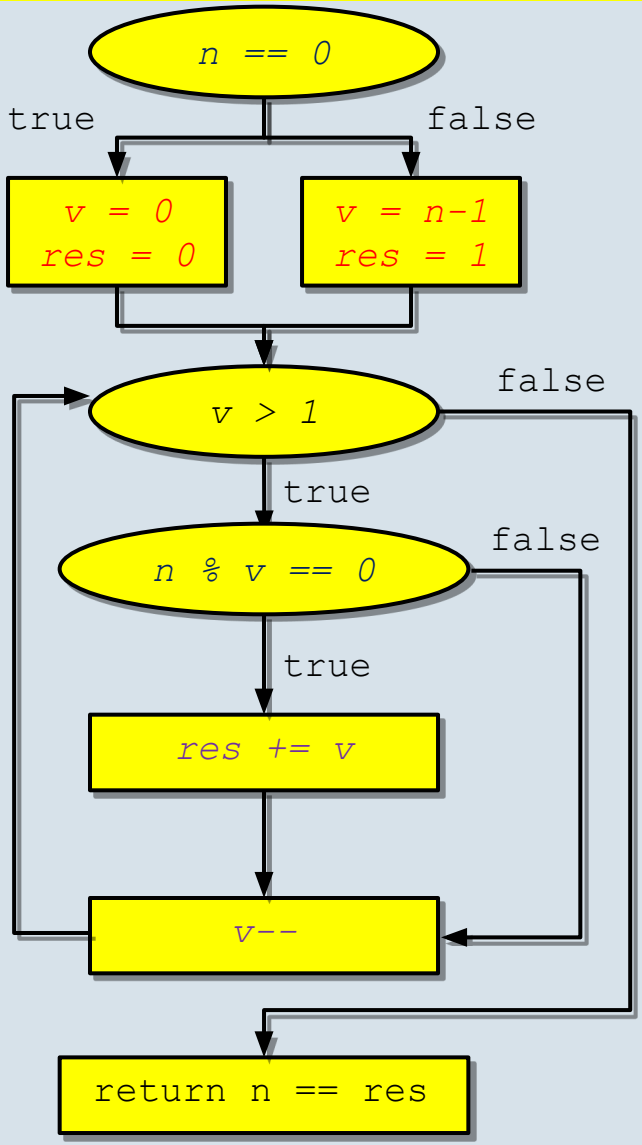
Condição: v >= 1

Acção: if (n%v == 0) res += v

Progresso: v--

Resultado: n == res

Números perfeitos



Optimização: como 1 é divisor de todos os naturais, podemos optimizar a operação, inicializando o resultado a 1. Isso implica que o ciclo já não executa quando n = 1 . Nesse caso, temos de especificar que 0 é perfeito explicitamente.

Inicialização:

if (n==0) { v=0; res=0; }

else { v = n-1; res = 1; }

Condição: v > 1

Corpo:

Acção: if (n%v == 0) res += v

Progresso: v--

Resultado: n == res

Factorial

Definição de Factorial

- Definição recursiva:

$$n! = \begin{cases} 1 & , se\ n = 0 \\ n \times (n-1)! & , se\ n > 0 \end{cases}$$

- Definição iterativa:

$$n! = \prod_{k=1}^{n} k$$

- Em ambos os casos, 0! = 1
  - Repare que quando o conjunto indexante é vazio o produtório – tal como acontece com o somatório – tem como resultado o elemento neutro da operação (neste caso, o elemento neutro é 1).
- Podemos resolver o mesmo problema quer de forma iterativa, quer de forma recursiva! Ambas têm vantagens e inconvenientes.

Acrescentemos dois métodos à classe `Mathematics`

```
/**
 * Calcula o factorial de n de forma recursiva
 * @param n - valor sobre o qual se quer calcular o factorial
 * @return n! (factorial de n)
 * @pre n >= 0
 */
public static int factorial(int n) { ... }

/**
 * Calcula o factorial de n de forma iterativa.
 * @param n - valor sobre o qual se quer calcular o factorial
 * @return n! (factorial de n)
 * @pre n >= 0
 */
public static int factorialIt(int n) { ... }
```

Factorial (versão recursiva)

$$n! = \begin{cases} 1 & , se\ n = 0, \\ n \times (n-1)! & , se\ n > 0. \end{cases}$$

```
Exemplo: Cálculo de 5!
5!
(5 x  4!)
(5 x (4 x  3!))
(5 x (4 x (3 x  2!)))
(5 x (4 x (3 x (2 x  1!))))
(5 x (4 x (3 x (2 x (1 x  0!)))))
(5 x (4 x (3 x (2 x (1 x (1))))))
(5 x (4 x (3 x (2 x (1 x 1)))))
(5 x (4 x (3 x (2 x 1))))
(5 x (4 x (3 x 2)))
(5 x (4 x 6))
(5 x 24)
120
```

A `expressão sublinhada` é a próxima a calcular.
Para calcular factorial de 5, temos de multiplicar 5 por factorial de 4. Antes dessa multiplicação, temos de calcular factorial de 4.
E assim sucessivamente.
No final, podemos finalmente realizar as multiplicações que fomos adiando ao longo do processo.

Factorial (versão recursiva)

$$n! = \begin{cases} 1 & , se\ n = 0, \\ n \times (n-1)! & , se\ n > 0. \end{cases}$$

```
/**
 * Calcula o factorial de n de forma recursiva
 * @param n - valor sobre o qual se quer calcular o factorial
 * @return n! (factorial de n)
 * @pre n >= 0
 */
public static int factorial(int n) {
  int result;
  if (n == 0)
    result = 1;
  else
    result = n * factorial(n-1);
  return result;
}
```

Prós:
Definição intuitiva da operação.

Contras:
Pilha de chamadas aninhadas penaliza a eficiência do algoritmo.

Factorial (versão iterativa)

$$n! = \prod_{k=1}^{n} k$$

```
/**
 * Calcula o factorial de n de forma iterativa.
 * Ideia: em vez de adiar o cálculo, guarda-se na variável auxiliar
 * <code>result</code>.
 * @param n - valor sobre o qual se quer calcular o factorial
 * @return n! (factorial de n)
 * @pre n >= 0
 */
public static int factorialIt(int n){
  int result = 1;
  int counter = 1;
  while (counter <= n) {
    result *= counter;
    counter++;
  }
  return result;
}
```

Prós:
Algoritmo bastante mais eficiente que o anterior.

Contras:
Ligeiramente menos intuitivo que a versão recursiva.

Factorial (outra versão iterativa)

$$n! = \prod_{k=1}^{n} k$$

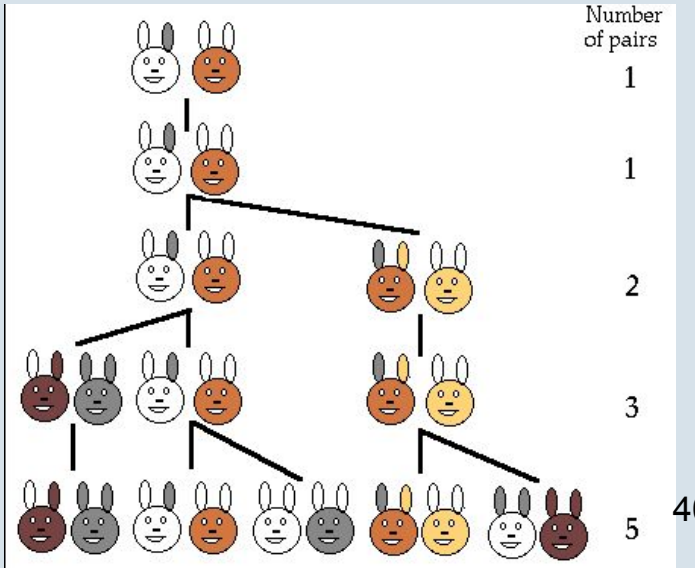
```
/**
 * Calcula o factorial de n de forma iterativa.
 * Ideia: em vez de adiar o cálculo, guarda-se na variável auxiliar
 * <code>result</code>.
 * @param n - valor sobre o qual se quer calcular o factorial
 * @return n! (factorial de n)
 * @pre n >= 0
 */
public static int factorialIt(int n){
  int result = 1;
  int counter = n;
  while (counter > 1) {
    result *= counter;
    counter--;
  }
  return result;
}
```

Semelhante à anterior, mas neste caso iteramos do valor mais alto para o mais baixo.

Sequência de Fibonacci

Sequência de Fibonacci

- Em 1225, Leonardo Fibonacci participou numa competição matemática em Pisa onde surgiu a seguinte questão:
- Começando apenas com um par de coelhos, se em cada mês cada par produtivo de coelhos produzir um novo par de coelhos (o qual se torna produtivo apenas um mês de depois de nascer), quantos pares de coelhos existirão ao fim de n meses?



Sequência de Fibonacci

- A resposta é a chamada sequência de Fibonacci:

  fib(0)=0, fib(1)=1,fib(2)= 1,fib(3)= 2,

  fib(4)=3, fib(5)=5,fib(6)= 8, ...

- Esta sequência é muito fácil de obter: cada número da sequência é a soma dos dois números anteriores!

$$fibonacci(n) = \begin{cases} 0 & , \quad n = 0 \\ 1 & , \quad n = 1 \\ fibonacci(n-1) + fibonacci(n-2) & , \quad n > 1 \end{cases}$$

- Ou seja...

$$fibonacci(n) = \begin{cases} n & , n <= 1 \\ fibonnacci(n-1) + fibonacci(n-2) & , n > 1 \end{cases}$$

Sequência de Fibonacci

- Crie um método na classe `Mathematics` que calcule o valor de sequência de Fibonacci para um determinado número `n` (ou seja, quantos coelhos existem passados `n` meses).

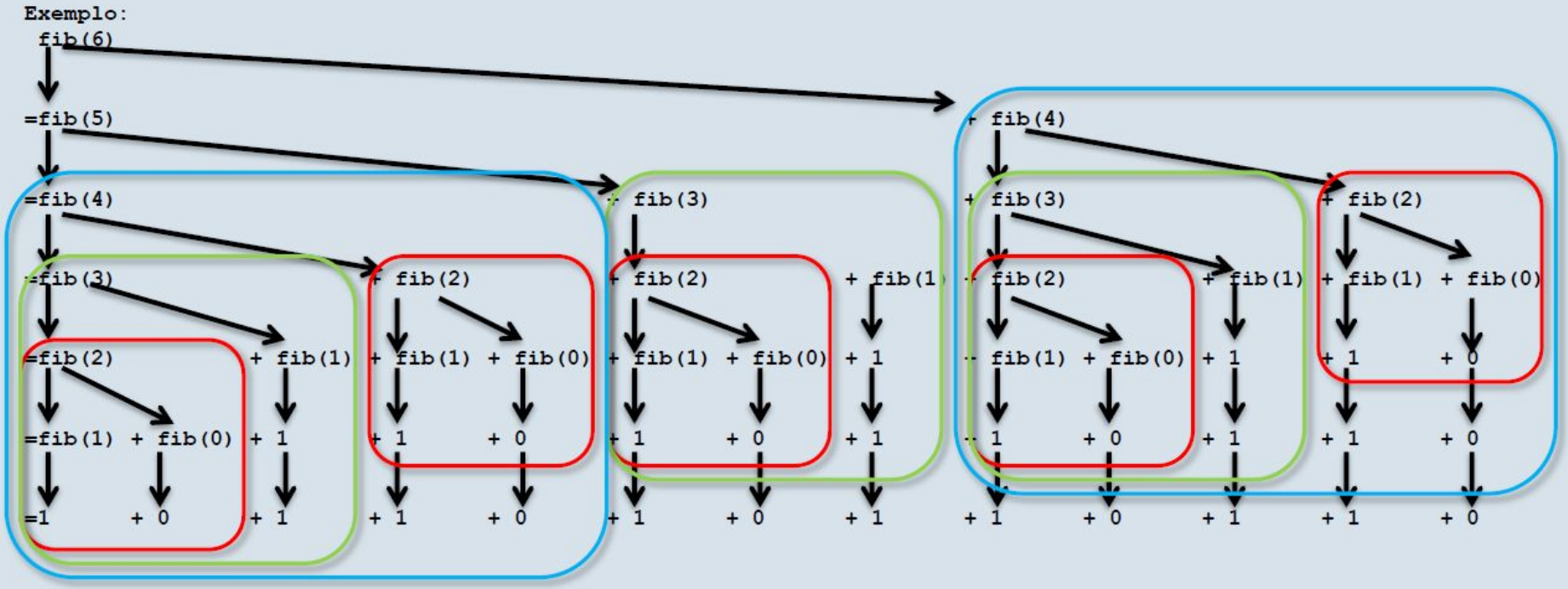
```
public static int fibonacci(int n)
```

Calcula o valor da sequência de Fibonacci na posição n.

Pre: n >= 0

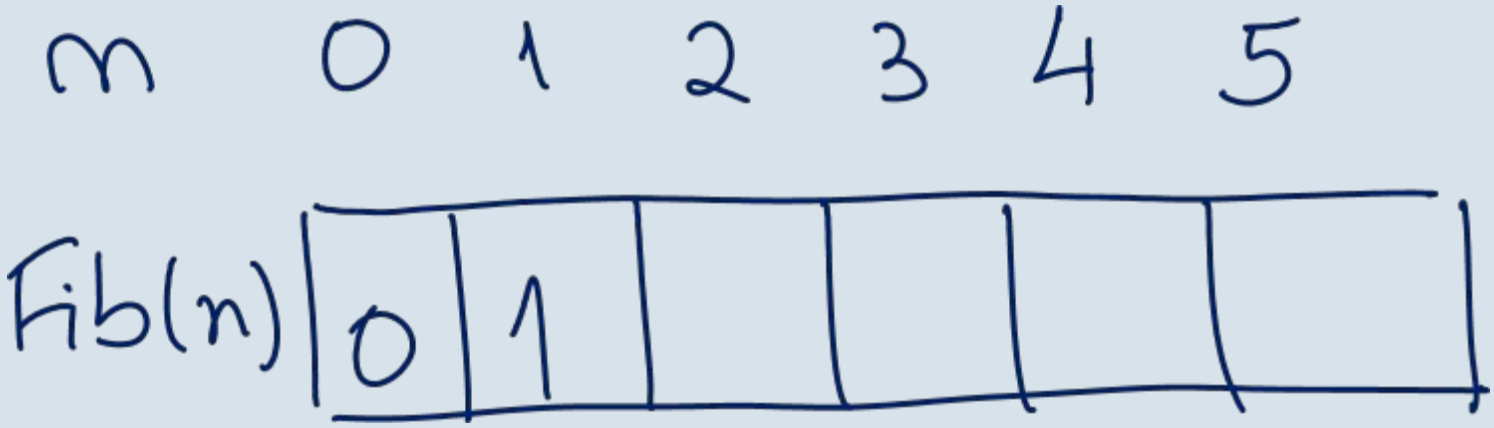
- Crie depois um método de teste no seu programa principal que imprima os primeiros 10 valores da sequência de Fibonacci.

Sequência de Fibonacci (versão recursiva)

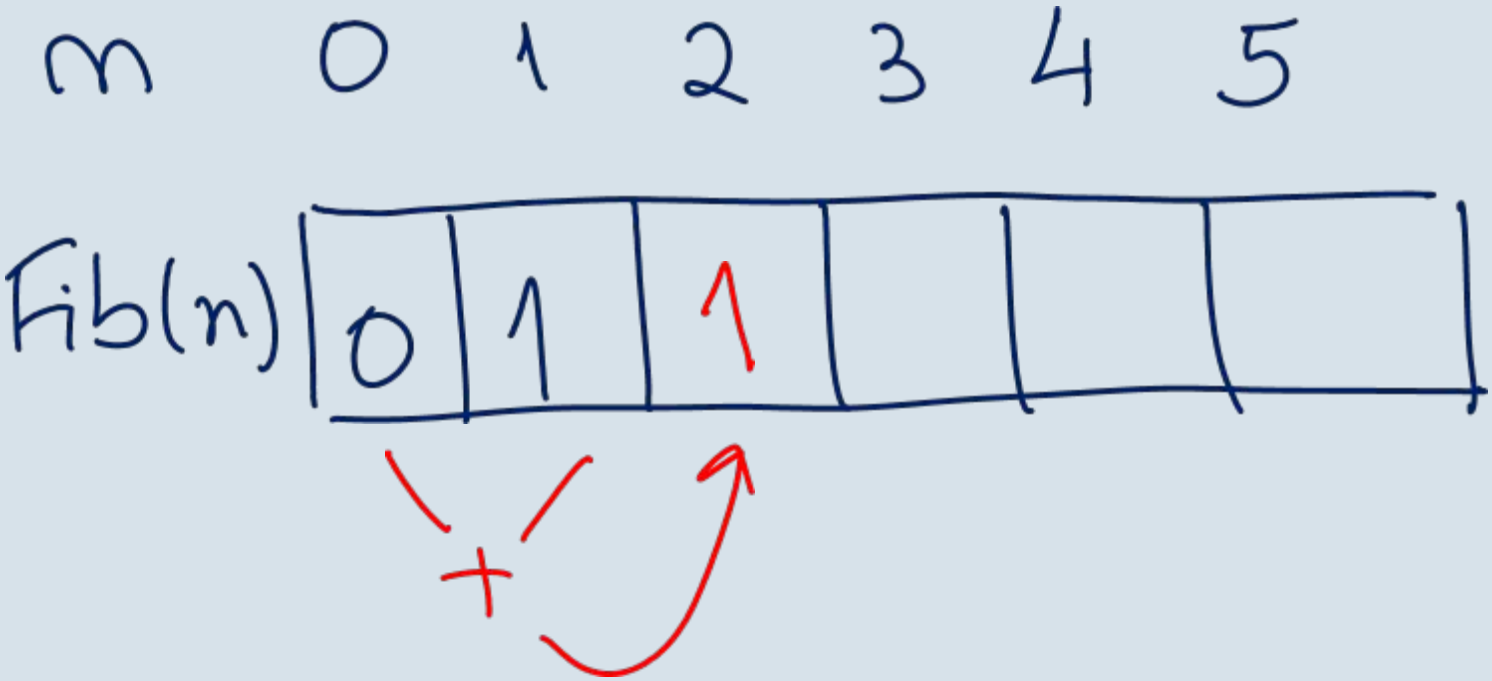


$$fibonacci(n) = \begin{cases} n & ,n <= 1 \\ fibonnacci(n-1) + fibonacci(n-2) & ,n > 1 \end{cases}$$

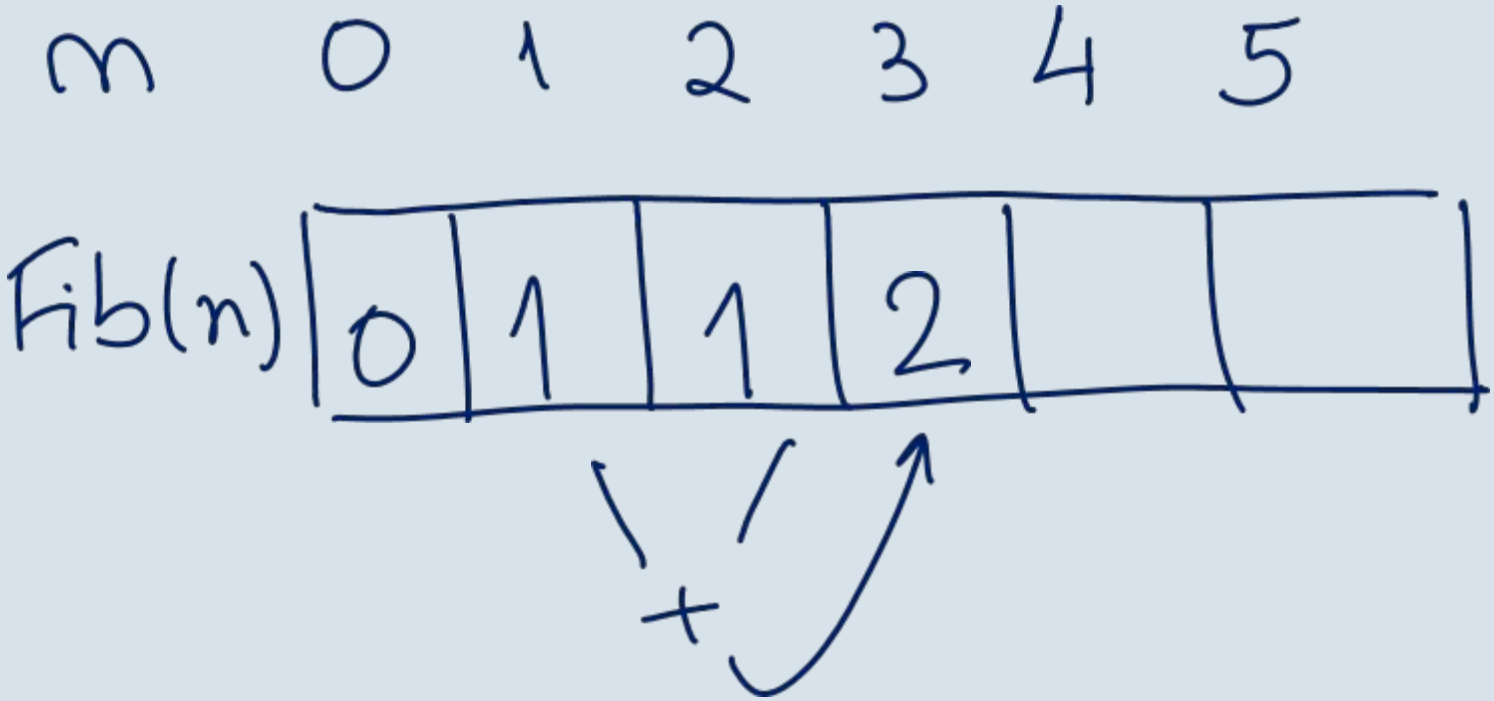
Fibonacci (versão iterativa)



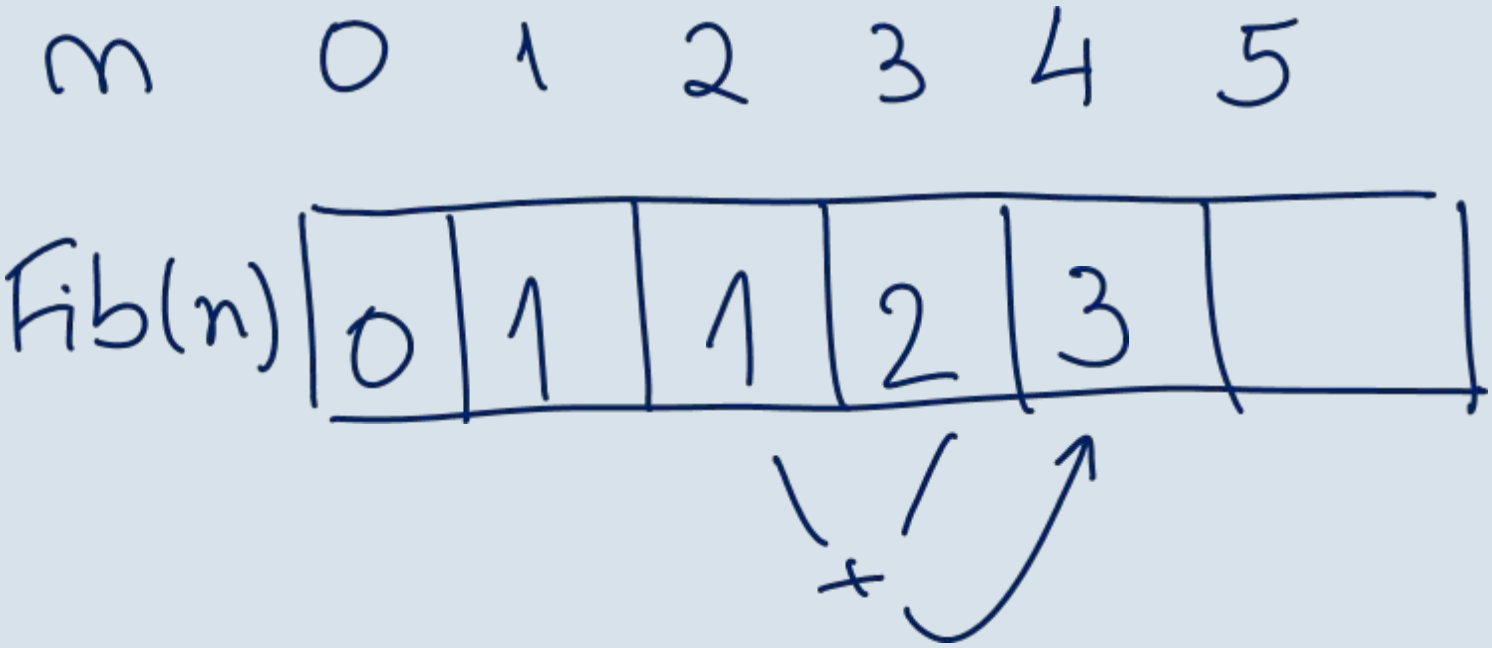
Fibonacci (versão iterativa)



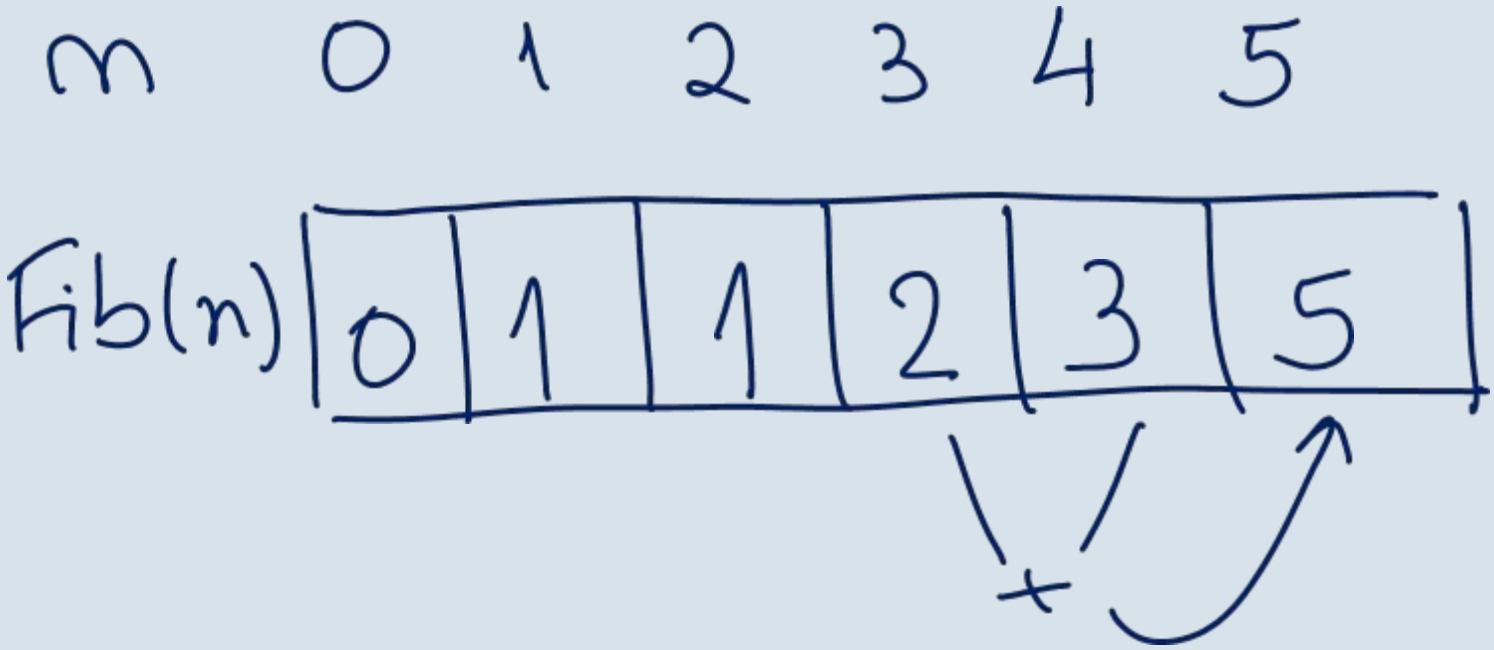
Fibonacci (versão iterativa)



Fibonacci (versão iterativa)

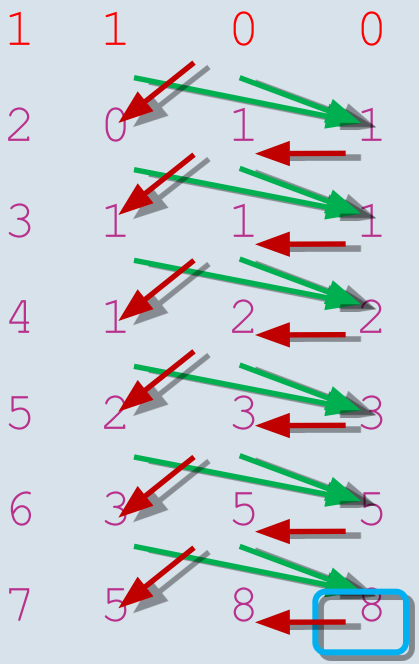


Fibonacci (versão iterativa)



Fibonacci (versão iterativa)

```
Exemplo:
Fibonacci(6)
k   pen ult res
```



- Como pode ver na tabela de emulação do Fibonacci(n), a cada passo temos o valor do último `ult` e do penúltimo `pen`. Em `res` guardamos o valor da soma de `ult` com `pen` no passo anterior. Depois, temos de actualizar ult e pen.
- Como programar o algoritmo com um ciclo? recorde a forma genérica de um "while":

```
inic;
while (condição){
   corpo
}
resultado
```

Sejam:

```
ult = 1
pen = 0
Se n < 2
   res = n
else
   k = 2
```

um contador auxiliar

A condição é: `k<=n`

O corpo é:

```
res = ult + pen;
pen = ult;
ult = res;
k++;
```

O resultado é: `return res;`

$$fibonacci(n) = \begin{cases} n & ,n <= 1 \\ fibonnacci(n-1) + fibonacci(n-2) & ,n > 1 \end{cases}$$

Máximo Divisor Comum

Máximo Divisor Comum

- O Algoritmo de Euclides para cálculo do Máximo Divisor Comum (mdc) é o seguinte:
- Dados dois números naturais a e b, em que pelo menos um deles é diferente de zero
  - Ver se b é zero
    - Se sim, a é o mdc;
    - Caso contrário, repetir o processo usando respectivamente b e a % b;

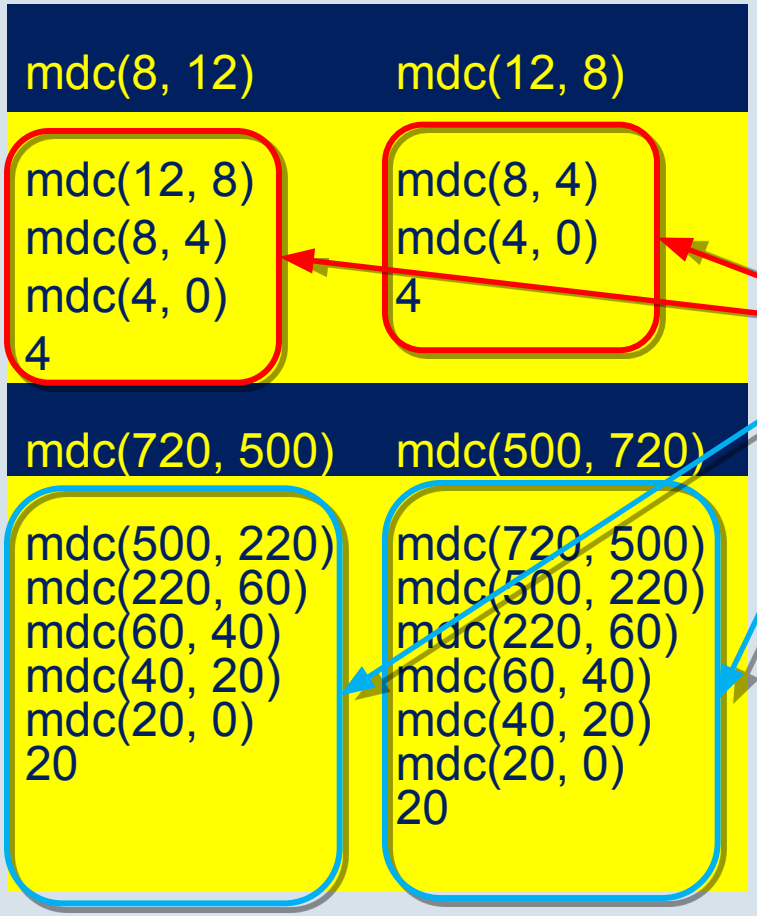
$$mdc(a,b) = \begin{cases} a & , se\ b = 0 \\ mdc(b, a\%b) & , se\ b \neq 0 \end{cases}$$

- Por exemplo MDC(8,12) = 4

Máximo Divisor Comum

Exemplificando o algoritmo:

$$mdc(a,b) = \begin{cases} a & ,se\ b = 0 \\ mdc(b, a\%b) & ,se\ b \neq 0 \end{cases}$$



Após a primeira iteração nas chamadas a mdc com o primeiro argumento menor que o segundo, a execução é semelhante à obtida quando o mdc é calculado com o primeiro argumento maior que o segundo.

Máximo Divisor Comum

- Crie um método na classe `Mathematics` que calcula o MDC entre dois números naturais `a` e `b`, utilizando o algoritmo de Euclides

```
public static int mdc(int a, int b)
```

Devolve o mdc dos números a e b.

Pre: a >= 0 && b >= 0

- Teste o método num programa principal que vá pedindo ao utilizador pares de números naturais e devolvendo, para cada par, o seu MDC. O programa deve terminar quando o utilizador tentar indicar números menores ou iguais a 0.

Máximo Divisor Comum (versão recursiva)

- O Algoritmo de Euclides para cálculo do Máximo Divisor Comum (mdc) é o seguinte:
- Dados dois números naturais a e b, em que pelo menos um deles é diferente de zero
  - Ver se b é zero
    - Se sim, a é o mdc;
    - Caso contrário, repetir o processo usando respectivamente b e a % b;

$$mdc(a,b) = \begin{cases} a & ,se\ b = 0 \\ mdc(b, a\%b) & ,se\ b \neq 0 \end{cases}$$

Máximo divisor comum (versão iterativa)

mdc(8, 12)	mdc(12, 8)
mdc(12, 8) mdc(8, 4) mdc(4, 0) 4	mdc(8, 4) mdc(4, 0) 4

mdc(720, 500)	mdc(500, 720)
mdc(500, 220) mdc(220, 60) mdc(60, 40) mdc(40, 20) mdc(20, 0) 20	mdc(720, 500) mdc(500, 220) mdc(220, 60) mdc(60, 40) mdc(40, 20) mdc(20, 0) 20

- Como pode ver nas tabelas de emulação do mdc(a,b), as chamadas consecutivas da função colocam no primeiro parâmetro o segundo e no segundo o resto da divisão dos dois parâmetros
- Podemos fazer isto com variáveis auxiliares!
- Mas como trocar valores?
  - Usar uma variavel extra para guardar o valor original de uma delas e depois, proceder à troca
- Como programar o algoritmo com um ciclo? recorde a forma genérica de um "while":

```
inic;
while (condição){
   corpo
}
resultado
```

Seja `t` a variável auxiliar.
A condição é: `b>0`
O corpo é:

```
t=b;
b=a%b;
a=t;
```

O resultado é: `return a;`

Algumas ideias a reter

- Em vários destes problemas, existia uma definição recursiva bastante intuitiva (que nos era dada)
- Tipicamente, essa solução era elegante, mas pouco eficiente
- Para resolver esse problema, transformámos a solução recursiva numa solução iterativa
  - Na solução recursiva, havia um cálculo que ia sendo adiado, até se chegar ao caso base
  - Na solução iterativa, recorremos a variáveis auxiliares para realizar os cálculos intermédios que na solução recursiva iam sendo adiados

- Note bem: um estudo mais aprofundado da recursão e das técnicas para a construção de algoritmos recursivos é deixado para outras cadeiras.
- Em IP, dado um algoritmo sob a sua forma recursiva, como aconteceu com os algoritmos do factorial, Fibonacci, e máximo divisor comum de Euclides, deverá ser capaz de o implementar em Java.

A nossa biblioteca

- Praticámos a utilização de ciclos na construção de uma biblioteca com operações matemáticas diversas.
- Ficámos a conhecer alguns algoritmos clássicos da programação.
  - Vimos, para vários desses algoritmos clássicos, duas versões: uma recursiva, tipicamente intuitiva, mas menos eficiente, outra iterativa, menos intuitiva mas mais eficiente
- Criámos uma classe `Mathematics`, onde todos os métodos são `static` (estes métodos denominam-se “métodos de classe”).
  - Esta classe não tem construtores, pois nunca vamos criar objectos desta classe. É como a classe `Math` do Java.
- De igual modo, esta classe também não tem variáveis de instância. Repare, uma variável de instância é uma variável que pertence a uma instância, ou seja, a um objecto. Se nunca vamos criar um objecto desta classe, não faria sentido criar variáveis de instância.