

Repetição de Comandos

Mestrado Integrado em Engenharia Informática FCT UNL

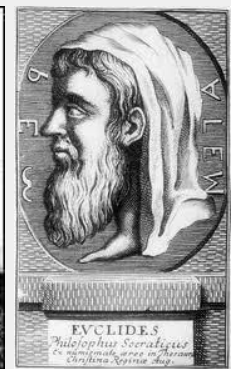
<http://ctp.di.fct.unl.pt/miei/ip/>

Corpo Docente 2020/2021

António Ravara, Artur Miguel Dias, Bernardo Toninho,
Ema Vieira, Inês Fernandes, Margarida Mamede,
Miguel Monteiro, Rui Nóbrega

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 (o 1 e ele próprio).
- 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 $n \geq 1$ 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

$n \% 2$

$\neq 0$



$n \% 3$

$\neq 0$



...

$n \% n-2$

$\neq 0$



$n \% n-1$

$\neq 0$



Números primos

Intervalo de variação de v:
 $1 < v \leq n$

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`:
 $1 < v \leq n$

```
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`:
 $1 < v \leq n$

```
while (condicao)
    instrucao
```

Esta condição é suficiente?

Sim, se $1 < v \leq n$

→

```
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 garantir 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

```
/** pre: n >= 1 */  
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

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

```
    }
```

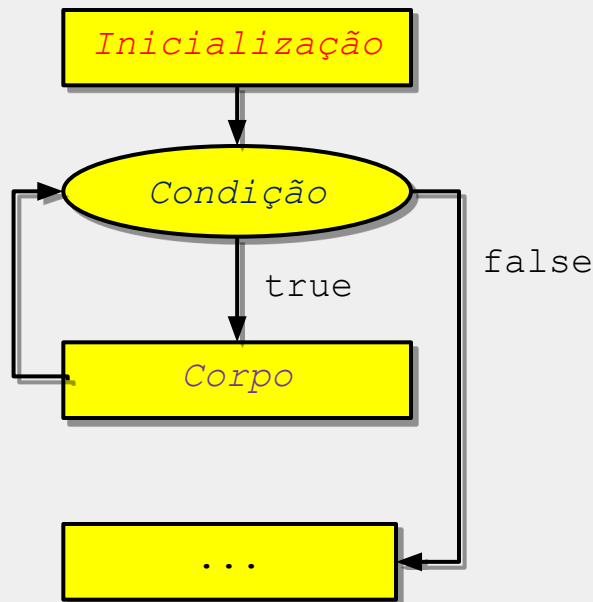
```
}
```

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`

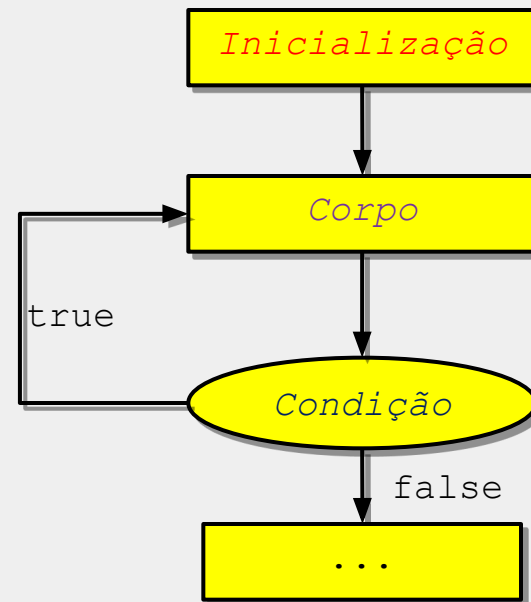
Executa 0 ou mais vezes



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

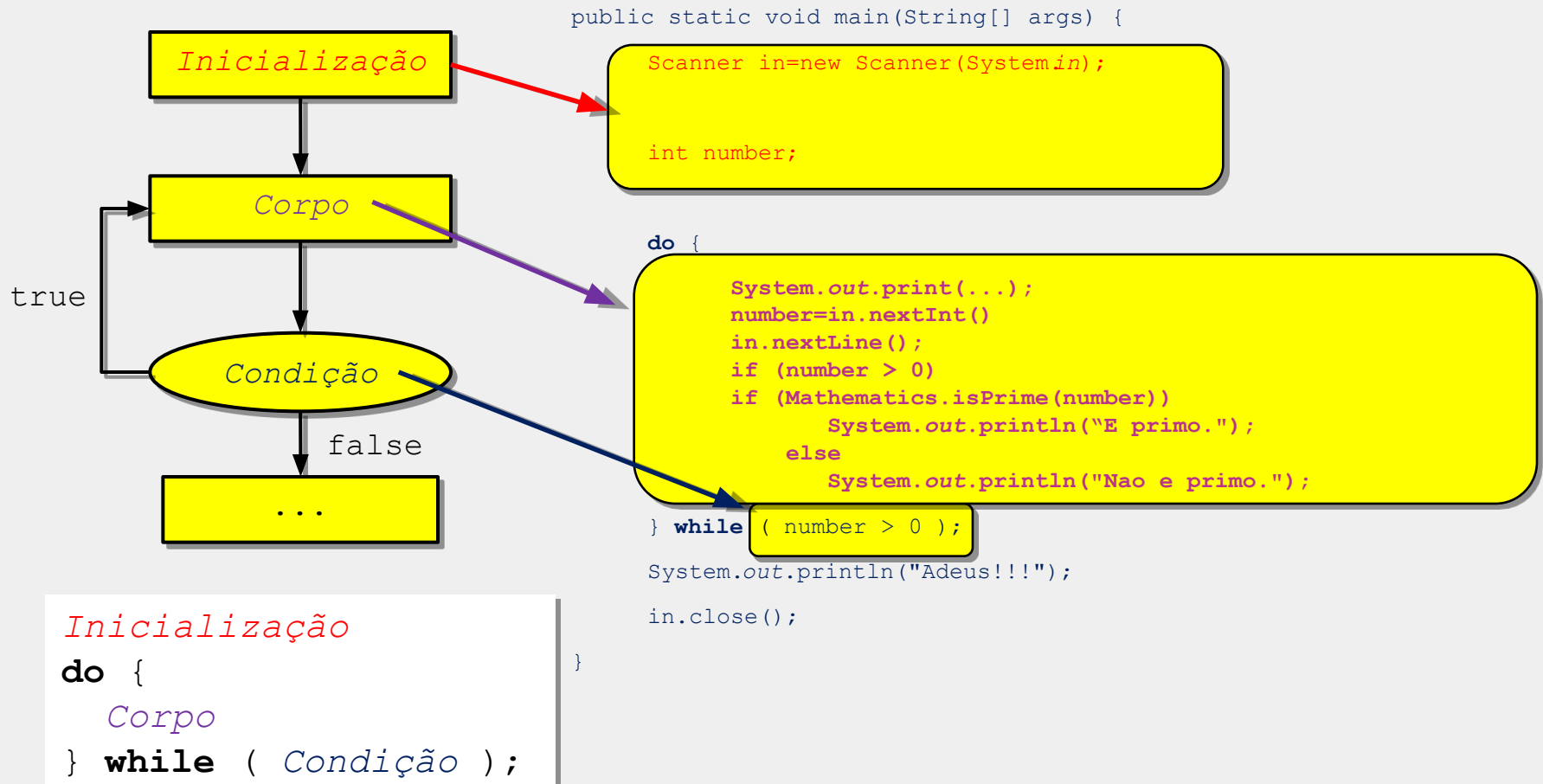
`do...while`

Executa 1 ou mais vezes



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

O ciclo do...while



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();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();
    }
}
```

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

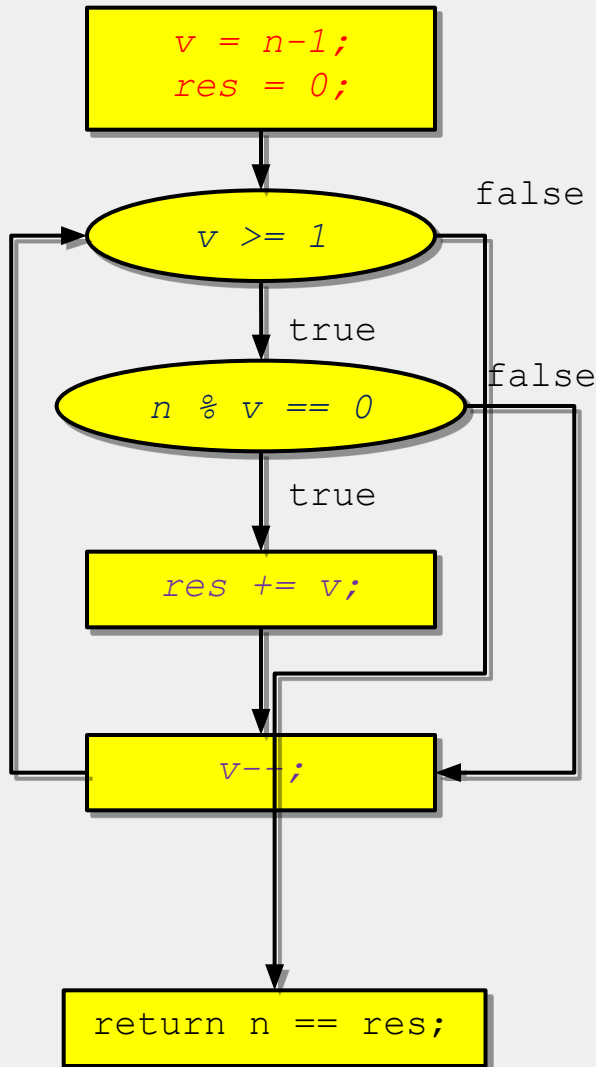
```
public static boolean isPerfectNumber(int n)
```

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: $v = n-1$; $res = 0$;

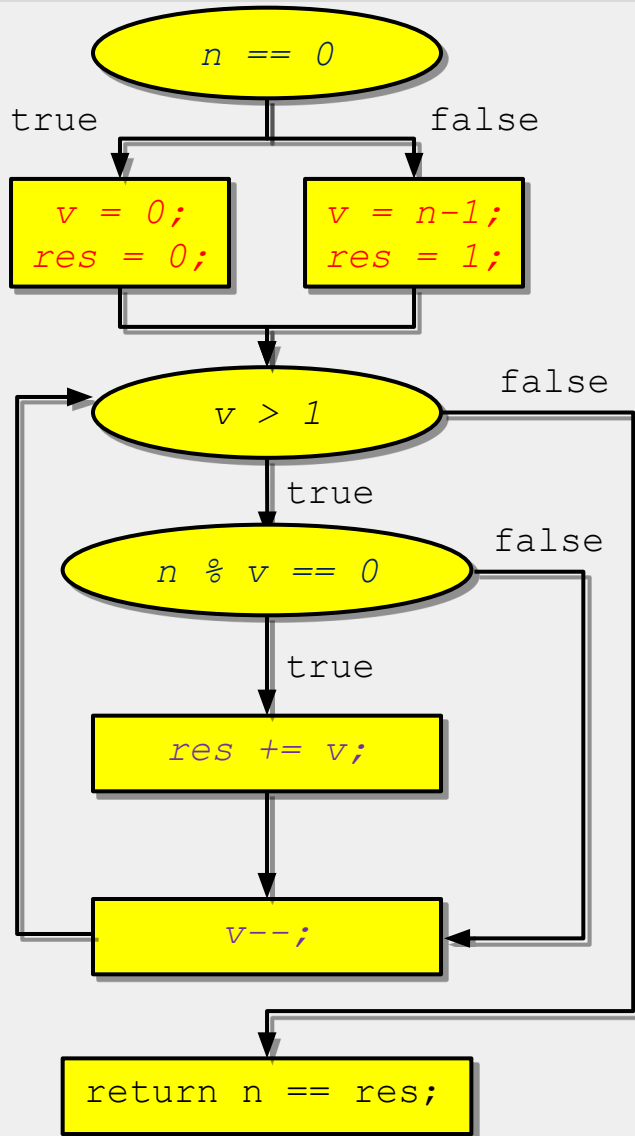
Condição: $v \geq 1$

Acção: `if (n%v == 0) res += v;`

Progresso: $v--$;

Resultado: $n == res$

Números perfeitos



Optimização: como 1 é divisor de todos os naturais, podemos otimizar 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: $\text{if } (n \% v == 0) \text{ res} += v;$

Progresso: $v--;$

Resultado: $n == \text{res}$

Factorial



Definição de Factorial

- Definição recursiva (caso **base** e **passo** - chamada 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. O conjunto indexante é vazio se $n < 1$.
- 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) {
    if (n == 0) // caso base
        return 1;
    else // chamada recursiva (passo)
        return n * factorial(n - 1);
}
```

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) {
    if (n == 0) // caso base
        return 1;
    else // chamada recursiva (passo)
        return n * factorial(n - 1);
}
```

Prós:

Definição intuitiva da operação (é quase uma tradução!).

Contras:

Pilha de chamadas aninhadas penaliza a eficiência do algoritmo.

Factorial (versão iterativa)

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

Estratégia para a implementação:

- sabemos o número de iterações: n (último menos primeiro mais 1)
- queremos multiplicar todos os naturais de 1 a n
- é natural “ir multiplicando”, do menor para o maior
- precisamos de um acumulador para guardar os resultados parciais

Exemplo: n = 4

```
result = 1;  
result = result*1; // result tem (1)*1  
result = result*2; // result tem (1*1)*2  
result = result*3; // result tem (1*1*2)*3  
result = result*4; // result tem (1*1*2*3)*4
```

Factorial (versão iterativa)

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

Exemplo: n = 4

```
result = 1;  
result = result*1; // result tem (1)*1  
result = result*2; // result tem (1*1)*2  
result = result*3; // result tem (1*1*2)*3  
result = result*4; // result tem (1*1*2*3)*4
```

Estratégia:

- começar com o k a 1 e ir multiplicando até k ser n ;
- temos de ir *acumulando* as multiplicações até obter o resultado final.

Ou seja, para k de 1 até n , multiplica-se k ao acumulador.

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  
 */
```

Inicialização: `result = 1; e k = 1;`
Condição: `k <= n`
Acção: `result = result*k;`
Progresso: `k = k+1;`

Factorial (versão iterativa)

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

Inicialização: `result = 1; e k = 1;`

Condição: `k <= n`

Acção: `result = result*k;`

Progresso: `k = k+1;`

Quando sabemos à priori o número de iterações, o ciclo mais adequado não é o **while** mas sim o **for**:

Forma geral:

for (*inicialização*; *condição*; *progresso*) *acção*

inicialização: da variável de iteração

condição: enquanto se verificar repete-se a *acção*

progresso: incremento da variável de iteração

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;  
    for (int counter = 1; counter <= n; counter++;)  
        result *= counter;  
    return result;  
}
```

Máximo Divisor Comum

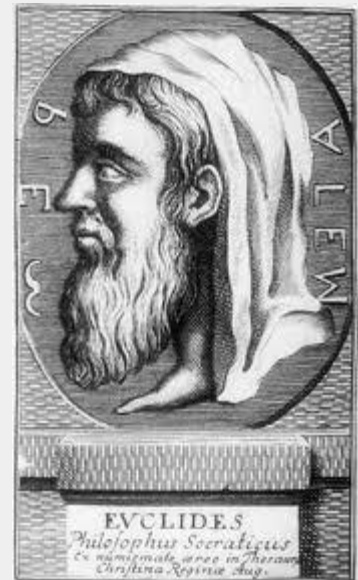


Máximo Divisor Comum

- 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 (caso base)
 - Se sim, a é o mdc;
 - Caso contrário (passo), *repetir* o processo usando respectivamente b e $a \% b$;

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

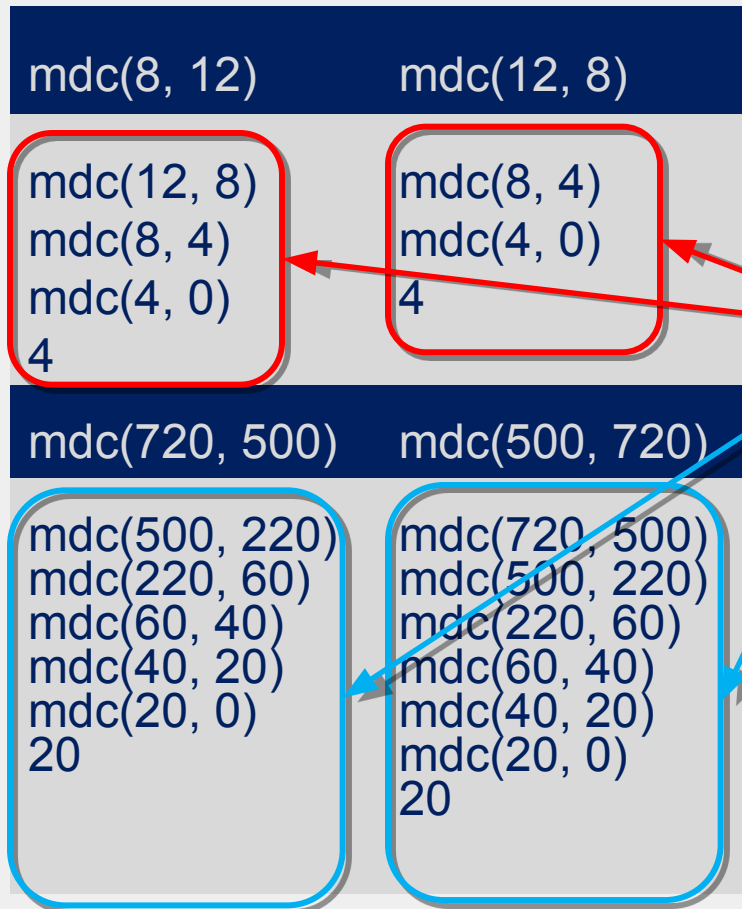
- Por exemplo $MDC(8,12) = 4$



Máximo Divisor Comum

Exemplificando o algoritmo:

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



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)

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

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

```
public static int mdc(int a, int b){  
    if (b==0)  
        return a;  
    else  
        return mdc(b, a%b);  
}
```

Algumas ideias a reter

- Em vários destes problemas, existia uma definição recursiva bastante intuitiva (que nos era dada)
- Vimos igualmente uma 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.

