Aula 04

Introdução à Programação Modular

Funções

Programação 1, 2015-2016

v1.0, 11-10-2015

, DETI, Universidade de Aveiro

Conteúdo

1 Introdução à programação modular

Introdução (1)

- Na especificação de um problema chegamos facilmente a um conjunto de tarefas básicas (ex: `ler`, `calcular`, `imprimir`, `sqrt`).
- Por vezes, estas tarefas básicas *repetem-se* no algoritmo de resolução do problema.
- Outras vezes, é possível reduzir o impacto dessas tarefas noutras partes do programa apenas pelo efeito que têm no estado do programa, ou mesmo concentrar esse efeito numa única variável (ex: `sqrt`, `diasDoMes`, etc.).
- Não por acaso, verifica-se frequentemente uma enorme assimetria entre a *implementação* de um algoritmos e o *efeito* que dele queremos obter para o resto do programa.

Implementação *versus* **Utilização**: A implementação dum algoritmo é geralmente bem mais complicada do que o resultado que dele queremos retirar.

- Por exemplo: ler um número num determinado intervalo, calcular a raiz quadrada de um número, calcular o número de dias de um mês (num determinado ano), etc..

2 Funções

Funções (1)

- Para resolver estes problemas as linguagens de programação implementam um mecanismo designado por *função*.

- Uma função separa claramente a sua *implementação* da sua *utilização*.
- A implementação pode ter alguma complexidade, a utilização deve ser sempre tão simples quanto possível.

```
public static void main(String[] args) {
   int nota = lerNumero(0, 20); // utilização simples
   System.out.print("Nota: "+nota);
}

static int lerNumero(int min, int max) {
   // implementação "complicada"
   int res;
   Scanner sc = new Scanner(System.in);
   do {
      System.out.print("Valor: ");
      res = sc.nextInt();
      if (res < min || res > max)
         System.out.println("ERROR: valor fora do intervalo!");
   } while (res < min || res > max);
   return res;
}
```

04.4

Funções (2)

- Uma *função* permite *realizar* um determinado conjunto de operações, se necessário receber informação por *argumentos* e *devolver* um valor.
- As funções desenvolvidas pelo programador são chamadas no programa da mesma forma que as funções criadas por terceiros (por exemplo as funções de leitura ou escrita de dados ou as funções da classe `Math`).
- Para além das funções permitirem eliminar repetição de algoritmos no programa, elas são um mecanismo de *abstracção*.
- A abstracção é a forma como nós, humanos, lidamos com a complexidade: na resolução de problemas não temos de saber detalhadamente tudo (desde o Universo até aos quarks), basta o mínimo necessário para a resolução do problema em mãos.
- Assim, torna-se possível, por exemplo, utilizar carros sem saber como é que eles funcionam, mas tão só apenas como se utilizam.

04.5

Funções (3)

Abstracção Algorítmica: As funções implementam a chamada *abstracção algorítmica*: a sua utilização não requer o conhecimento da sua implementação e é geralmente muito mais simples.

- Estrutura de um programa:

```
inclusão de classes externas
public class Programa
{
   funções desenvolvidas pelo programador

   public static void main(String[] args) {
      ...
   }

   funções desenvolvidas pelo programador
}
```

- As funções são criadas dentro da classe, antes ou depois da definição da função `main`.

04.6

Definição de uma função (1)

```
cabeçalho da função {
   corpo da função
}
```

- No cabeçalho da função indicam-se os qualificadores de acesso (neste momento sempre `static`), o tipo do resultado de saída, o nome da função e dentro de parênteses curvos a lista de argumentos.

```
public static tipo nome(argumentos) {
   declaração de variáveis
   sequências de instruções
}
```

04.7

Definição de uma função (2)

- Uma função é uma unidade auto-contida que recebe dados do exterior através dos argumentos, se necessário, realiza tarefas e eventualmente devolve um resultado.
- O resultado de saída de uma função pode ser de qualquer tipo primitivo (`int`, `double`, `char`, ...), de qualquer tipo referência (iremos ver noutra aula) ou `void` (no caso de uma função não devolver um valor).
- A lista de argumentos (ou parâmetros) é uma lista de pares de identificadores separados por vírgula, onde para cada argumento se indica o seu tipo de dados e o seu nome.
- O corpo da função assemelha-se à estrutura de um programa.
- Se a função devolver um valor utiliza-se a palavra reservada `return` para o devolver.
- O valor devolvido na instrução de `return` deve ser compatível com o tipo de saída da função.

04.8

Outros Exemplos

```
public static void escreveSoma(int x, int y) {
   System.out.printf("%d + %d = %d\n", x, y, x+y);
}

public static int diasDoMes(int mes, int ano) {
   assert mes >= 1 && mes <= 12;
   int res = 31; // correcto em 7/12 dos casos!
   ...
   return res;
}

// máximo divisor comum
public static int mdc(int x, int y) {
   ...
}
```

- Podemos ter *implementações incorrectas* de funções e *utilizações correctas*!
- Nesses casos, bastará rectificar a implementação.

04.9

Exemplo completo

```
public static void main(String[] args) {
   int a, b, r;
   a = lerPositivo(); // utilização das funções definidas pelo programador
   b = lerPositivo(); // da mesma forma que utilizamos todas as outras...
   r = soma(a, b); // o valor de a e b são passados à função soma
   printf("%d + %d = %d\n", a, b, r);
}

public static int lerPositivo() {
```

```
    int res;
    Scanner sc = new Scanner(System.in);
    do {
        System.out.print("Valor (positivo): ");
        res = sc.nextInt();
        if (res <= 0)
            System.out.println("ERROR: invalid number!");
    } while(res <= 0);
    return res;
}

public static int soma(int x, int y) { // neste exemplo x = a e y = b
    return x + y;
}
```

04.10

3 Tipos primitivos como argumentos

Tipos primitivos como argumentos

- Tomando como exemplo o programa da soma de dois positivos, podemos ver que a função `lerPositivo` não recebe argumentos e devolve um valor inteiro.
- Quando chamada duas vezes dentro da função `main`, o seu valor de retorno é armazenado nas variáveis `a` e `b`.
- A função `soma` recebe dois argumentos do tipo inteiro e devolve também um valor inteiro.
- Quanto executada, o conteúdo das variáveis `a` e `b` são passados para "dentro" da função `soma` através dos parâmetros `x` e `y` respetivamente ($x = a$, $y = b$).
- Sempre que uma função é usada, o computador "salta" para dentro da função, executa o seu código e quando termina continua na instrução seguinte à chamada da função.

04.11

4 Visibilidade das variáveis

Visibilidade das variáveis

- Vimos que um programa pode conter várias funções, sendo obrigatoriamente uma delas a função `main`.
- As variáveis locais apenas são visíveis no corpo da função onde são declaradas:
- no exemplo anterior, `a`, `b` e `r` apenas são visíveis na função `main`;
- os argumentos `x` e `y` apenas são visíveis na função `soma`.
- As variáveis declaradas dentro de um bloco (ou seja dentro do conjunto de instruções delimitado por `{ ... }` têm visibilidade limitada ao bloco (por exemplo ciclo `for`).
- Podemos também ter variáveis globais, sendo estas declaradas fora da função `main`, dentro da classe que implementa o programa (precedidas da palavra reservada `static`).

04.12

Exemplo

```
public class visibilidade {
    // variáveis globais:
    static int a = 1; // variável global
    static Scanner sc = new Scanner(System.in);

    public static void main(String[] args) {
        int x, y; // variáveis locais à função main
        ...
    }
    public static int f1(int y) { // variável local à função f1
        int x; // variável local à função f1
        ...
        for (int i = 0 ; i < x ; i++) { // i é apenas visível no ciclo for
            ...
        }
```

```
        ...
    }
}
```