Aula 03
Programação Modular

Programação II, 2016-2017

v2.9, 26-02-2017

DETI, Universidade de Aveiro

Objectivos:
- Medidas de qualidade de programas;
- Modularidade;

Conteúdo

1 Métricas de Qualidade

Avaliando a qualidade do software

- *Correcção*: o software efectua as suas funções exactamente como definido nas suas especificações;
- *Robustez*: o software "funciona" em situações fora das suas especificações;

- *Fiabilidade*: o software é correcto e robusto;
- *Extensibilidade*: o software é fácil de adaptar a mudanças de especificações;
- *Reutilização*: o software pode ser utilizado total, ou parcialmente, para novas aplicações;
- *Eficiência*: o software utiliza o mínimo de recursos de hardware necessários (directamente relacionada com a complexidade algorítmica):
  - CPU: tempo de execução;
  - RAM: memória gasta.

03.3

- Outros factores de qualidade importantes são:
  - *legibilidade* - facilidade de leitura e compreensão de software;
  - *verificabilidade* - facilidade com que o mesmo pode ser testado;
- Muito embora todos estes factores sejam relevantes, eles não têm todos a mesma importância:
  - O mais importante de todos é indubitavelmente a *correcção*. Se um programa/função/módulo não faz o que é suposto fazer é irrelevante que o mesmo seja eficiente (por exemplo).
  - O segundo mais importante é a *robustez*;
  - Na construção de programas devem usar-se técnicas que maximizem estes dois factores de qualidade.

03.4

2 Modularidade

Modularidade: introdução

Os programas vão crescendo em tamanho e complexidade levantando novas questões:

- Torna-se cada vez mais importante não só o seu funcionamento externo mas também a forma como é construído;
- Pode existir a necessidade de ter várias pessoas a trabalhar simultaneamente no programa, pelo que uma comunicação fácil entre programadores através do próprio código passa a ser cada vez mais determinante;
- O número potencial de erros tende a aumentar obrigando a técnicas que facilitem a sua detecção e atempada correcção (e que evitem um crescimento exponencial da complexidade no seu tratamento);
- É necessário facilitar a manutenção e eventuais futuras evoluções do programa (extensibilidade).

03.5

Modularidade: conceito e critérios

Módulos: Excertos de programas (blocos) independentes com os quais se podem construir novos programas.

Dizemos que um bloco de um programa é modular se for:

1. Facilmente *separável* de outros blocos;
2. Facilmente *combinável* com outros blocos;
3. Fácil de ser *compreendido* isoladamente;
4. *Continuidade*: pequenas modificações num módulo apenas o afectam a ele ou eventualmente aos seus clientes directos;
5. *Auto-protegido*: dados internos protegidos contra utilizações abusivas.

03.6

Modularidade: vantagens

- Cada módulo pode ser desenvolvido, analisado e testado de forma *independente*:
  - Pode ser da responsabilidade de entidades (pessoas) distintas.
- Mais fácil de maximizar a *correcção* e a *robustez*;
- *Reduz a complexidade* do programa global:
  - Implementação de mecanismos de abstracção para facilitar tarefas.
- Facilita a *reutilização* de código:
  - Ao desenvolvermos um módulo especializado numa tarefa/funcionalidade, podemos facilmente reutilizá-lo noutro programa com as mesmas necessidades.

03.7

3 Evolução histórica: o caminho até à modularidade

1. Programa monolítico
2. Abstracção algorítmica
3. Registos

} Programação I

4. Ficheiros como unidades de suporte à modularidade
   (*a*) Prog. Principal + Módulo Tipo Dados + Módulo Funções
   (*b*) Prog. Principal + Módulo Tipo Dados e Operações Associadas
5. Contexto de Existência de Objecto
6. Encapsulamento: Objectos e Tipos de Dados Abstractos

03.8

3.1 Programa monolítico

Programação modular – passo 1

PROGRAMA $\Leftrightarrow$ FUNÇÃO `main`

Programa monolítico!

O único "módulo" é o próprio programa

03.9

3.2 Abstracção algorítmica: funções

Programação modular – passo 2

PROGRAMA $\Leftrightarrow$ `main` + funções (no mesmo ficheiro)

ABSTRACÇÃO ALGORÍTMICA!

Funções podem ser utilizadas como módulos internos ao programa

03.10

Abstracção algorítmica!

Criação de Funções (métodos):

- Encapsulamento de uma sequência de instruções dentro de um módulo funcional;
- A compreensão desse módulo reduz-se à compreensão da sua utilização (e não da sua implementação);
- Tal como os programas, as funções podem ter parâmetros de entrada e de saída;
- Permite a sua fácil reutilização dentro do programa sem a necessidade de replicar a sua implementação;
- Podemos associar-lhes especificações formais de correcção através de pré-condições e pós-condições.

Problemas:

- O facto de serem unidades de código que são compiladas conjuntamente com o programa principal limita a sua reutilização (só utilizando "*copy & paste*");
- A representação interna dos dados manipulados pelas funções está propagada em todo o lado.

03.11

3.3 Abstracção de dados: registos

Programação modular – passo 3

REGISTOS
PROGRAMA ⇔ `main` + funções + registos
1ª ABSTRACÇÃO DE DADOS
Representação de dados dos módulos deixa de estar directamente exposta aos clientes

03.12

Registos

Criação de novos tipos de dados (registos):

- Primeira abstracção de dados;
- Encapsulamento de um conjunto de tipos de dados dentro de um novo tipo de dados;
- O registo pode ser compreendido pelo seu todo (e não somente pelos tipos de dados sobre os quais é construído);
- Podemos lhes associar uma especificação formal de correcção através de um invariante (em `Java` não há suporte em tempo de execução para este formalismo).

Problemas:

- Dependem fortemente das funções que os manipulam;
- O facto de serem unidades de código que são compilados conjuntamente com o programa principal limita a sua reutilização (só utilizando "*copy & paste*").

03.13

3.4 Módulos de registos e módulos de funções

Programação modular – passo 4(a)

MÓDULOS REUTILIZÁVEIS DE FUNÇÕES
MÓDULOS REUTILIZÁVEIS DE REGISTOS
PROGRAMA ⇔ `main` + módulos de funções + módulos de registos
Módulos separados em diferentes ficheiros
Funções podem ser (re)utilizadas noutros programas
Representação de dados dos módulos directamente exposta aos clientes

03.14

Criação de novos módulos

- Unidades autónomas (ficheiros) contendo:
  - Tipo de Dados;
  - Funções.
- Podem ser invocados do exterior;
- Podem ser compilados isoladamente e "ligados" (*linked*) a outros programas que deles necessitem facilitando assim a reutilização de código;
- Programação Modular orientada a funções; (e.g.: biblioteca `Math`).

Questão:

- Se as operações estão associadas a um tipo de dados, faz sentido manter DADOS e OPERAÇÕES em ficheiros diferentes?

03.15

3.5 Módulos com registos e funções associadas

Programação modular – passo 4(b)

JUNÇÃO REGISTOS COM MÓDULOS DE FUNÇÕES
PROTECÇÃO E ABSTRACÇÃO DE DADOS
PROGRAMA $\Leftrightarrow$ `main` + módulos de registos/funções
Registo é passado como argumento das funções do próprio módulo

03.16

Programação modular – passo 4(b)

- Já vimos que a definição de novos tipos de dados de pouco serve se não forem definidas operações sobre eles:
  - Manipulação de dados;
  - Comparação, Atribuição, ...
  - Operações algébricas;
  - etc.
- *Então*: "Qual o interesse de termos os Dados e Funções Associadas em módulos diferentes?"
- Surgiu então um *novo conceito*:
  Módulo: Novo Tipo de Dados + Conjunto de Operações Associadas

03.17

3.6 Objectos

Programação modular – passo 5

Contexto de Existência de Objecto!
PROGRAMA $\Leftrightarrow$ `main` + objectos
As funções do módulo (métodos) e a respectiva estrutura de dados são indissociáveis.
Os métodos estáticos que recebiam o registo como argumento passam a métodos de objecto funcionando no contexto do objecto.

03.18

OBJECTOS!

PROGRAMA ⇔ `main` + objectos

Nenhum atributo deverá ser tornado público (ao contrário dos registos)

03.19

Objectos

Módulos absolutamente independentes:

- Cada módulo só deve aceder a dados locais;
- A interacção com o exterior deve ser efectuada através de funções do próprio módulo (interface);
- Obriga a utilizar mecanismos de protecção para "esconder" os dados do mundo exterior:
  `public / private / protected`

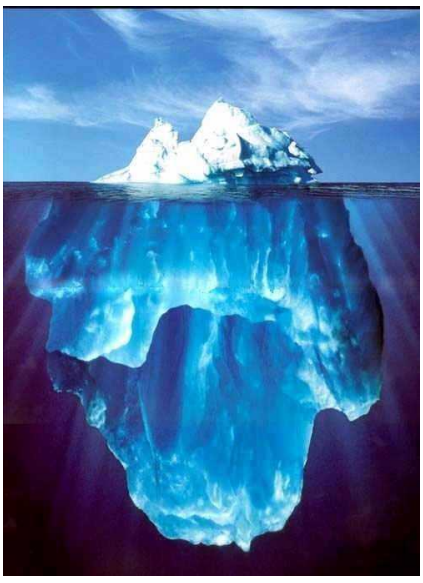
Um novo conceito:

- Encapsulamento (*Information Hiding*)!

03.20

Information hiding

A informação contida num módulo está inacessível a outros módulos que não têm necessidade dessa informação.



03.21

4 Evolução histórica: exemplo

Nesta secção mostramos através de um exemplo como se pode aproveitar os mecanismos de abstracção da linguagem de programação (funções, classes) para escrever um programa com boa modularidade, com as vantagens daí decorrentes.

O exemplo é uma calculadora de fracções: um programa que faz aritmética com números racionais como $\frac{2}{5} + \frac{1}{6} = \frac{17}{30}$. O programa recebe três parâmetros: duas fracções e uma operação ($+$, $-$, $*$, $/$) e apresenta o resultado correspondente (uma fracção também). Por exemplo, o comando:

```
java v1.FractionCalculator 2/5 + 1/6
```

deve produzir o resultado:

```
(2/5) + (1/6) = (17/30).
```

Começamos com uma versão monolítica do programa: o programa está todo numa única classe (num ficheiro só), numa função apenas. Depois vamos reorganizando progressivamente o código de forma a melhorar a sua modularidade, com ganhos sucessivos de legibilidade, reutilizabilidade, extensibilidade, etc..

O código fonte deste exemplo é disponibilizado no ficheiro `FractionsExample.zip`, que contém um pacote `Java` para cada versão.

4.1 Calculadora de Fracções, versão 1: programa monolítico (passo 1)

Nesta versão, o programa está implementado numa única função no ficheiro `v1.FractionCalculator`. A decomposição de nível 1 tem os passos seguintes:

1. Verificar e armazenar os argumentos e definir as variáveis fundamentais para as duas fracções dadas e para a fracção resultado. Cada fracção é armazenada na forma de dois inteiros: um numerador e um denominador.

```
public class FractionCalculator {
  public static void main(String[] args) {
    if (args.length != 3) {
      err.println("Uso: java v1.FractionCalculator <frac1> <op> <frac2>");
      exit(1);
    }
    String frac1 = args[0];
    String op = args[1];
    String frac2 = args[2];

    int x_num, x_den;  // numerador e denominador do primeiro operando
    int y_num, y_den;  // numerador e denominador do segundo operando
    int z_num=0, z_den=1;  // numerador e denominador do resultado
```

2. Interpretar os argumentos para extrair os numeradores e denominadores respectivos.

```
    p = frac1.indexOf('/');  // descobre o sinal de fracção
    if (p>=0) { // se encontrar, extrai numerador e denominador
      x_num = Integer.parseInt(frac1.substring(0,p));
      x_den = Integer.parseInt(frac1.substring(p+1));
    } else {    // senão, extrai numerador e deixa o denominador a 1
      x_num = Integer.parseInt(frac1);
      x_den = 1;
    }
    assert x_den != 0;
    // fazer o mesmo para frac2...
```

3. Executar a operação. Consoante o operador, selecciona as instruções certas para calcular o resultado.

```
switch (op) {
  case "+": // adicionar é trabalhoso
    z_num = x_num*y_den + x_den*y_num;
    z_den = x_den*y_den;
    break;
  case "*": // multiplicar é fácil
    z_num = x_num*y_num;
    z_den = x_den*y_den;
    break;
  // Deixam-se as outras operações como exercício...
```

4. Imprimir a operação solicitada e o resultado, ou seja: imprimir a primeira fracção, depois o operador, a segunda fracção, um sinal de igual e finalmente, o resultado.

```
// converter a primeira fracção numa string do tipo "(1/2)"
str = "(";
if (x_den>0) { // se denominador positivo, é fácil
  str += (x_num) + "/" + (x_den);
} else { // se negativo, troca sinais para evitar (3/-4) por exemplo
  str += (-x_num) + "/" + (-x_den);
}
str += ")";
out.print(str);             // imprime primeira fracção
out.printf(" %s ", op);     // imprime operador
// converter a segunda fracção da mesma maneira...
out.print(str + " = ");     // imprime segunda fracção e igual
// converter a fracção resultado da mesma maneira...
out.println(str);           // imprime resultado
```

Apesar de correcta e funcional, esta versão tem vários pontos fracos:

1. A função é longa e difícil de compreender porque inclui código referente a múltiplos níveis de decomposição algorítmica.
2. Há claramente redundância no código. Isso é particularmente evidente no código de leitura e escrita de fracções. A redundância dificulta a manutenção do código e potencia a introdução de erros.
3. Não há uma relação forte entre as duas variáveis que representam cada fracção; apenas os nomes bem escolhidos revelam essa relação.

A próxima versão resolve esses problemas.

4.2 Calculadora de Fracções, versão 2: programa estruturado com funções e um novo tipo de dados (passos 2 e 3)

Nesta versão definimos um novo tipo de dados para representar fracções: um registo com dois inteiros (numerador e denominador).

```
class Fraction {
  int num = 0;
  int den = 1;
  // Quando criada, uma fracção é inicializada por defeito a zero (0/1)
}
```

Também definimos uma série de funções que operam sobre fracções:

- multiplicar fracções;

```
static Fraction fractionMultiply(Fraction a, Fraction b) {
  Fraction r = new Fraction();  // fracção para o resultado
  r.num = a.num*b.num;
  r.den = a.den*b.den;
  return r;
}
```

- adicionar fracções;

```
static Fraction fractionAdd(Fraction a, Fraction b) {
  Fraction r = new Fraction();  // fracção para o resultado
  r.num = a.num*b.den + a.den*b.num;
  r.den = a.den*b.den;
  return r;
}
```

- converter fracção em String;

```
static String fractionToString(Fraction f) {
  String s = "(";
  if (f.den>0) {
    s += (f.num) + "/" + (f.den);
  } else {
    s += (-f.num) + "/" + (-f.den);
  }
  s += ")";
  return s;
}
```

- converter String em fracção.

```
static Fraction fractionFromString(String str) {
  Fraction f = new Fraction();
  int p = str.indexOf('/');
  if (p>=0) {
    f.num = Integer.parseInt(str.substring(0,p));
    f.den = Integer.parseInt(str.substring(p+1));
  } else {
    f.num = Integer.parseInt(str);
    f.den = 1;
  }
  assert f.den != 0;
  return f;
}
```

Com isto conseguimos escrever a função principal de uma forma bastante concisa e elegante, abstraindo-nos dos detalhes de implementação.

```
public static void main(String[] args) {
  // Tratar argumentos e definir variáveis
  // ...
  Fraction x;
  Fraction y;
  Fraction z = new Fraction();

  // 1) definir fracções (a partir dos argumentos);
  x = fractionFromString( frac1 );
  y = fractionFromString( frac2 );

  // 2) executar a operação;
  switch (op) {
    case "+":
      z = fractionAdd(x,y);  break;
    case "*":
      z = fractionMultiply(x,y);  break;
    // ...
  }

  // 3) imprimir a operação solicitada e o resultado;
  out.printf("%s %s %s = %s\n",
      fractionToString(x), op, fractionToString(y), fractionToString(z));
}
```

É um bom exemplo de *abstacção algorítmica*.

Neste nível de abstracção, as fracções são tratadas como entidades unas, de pleno direito: são criadas, manipuladas e escritas sem nunca se vislumbrarem as suas partes constituintes (não aparecem no `main` quaisquer referências aos atributos `num` ou `den`). É um bom exemplo de *abstracção de dados*.

As restantes funções também são bastante elegantes:

- são curtas;
- são fáceis de ler e compreender;
- têm propósitos bem definidos e distintos entre si, permitindo combiná-las para resolver inúmeros problemas.

Este último ponto é particularmente importante porque promove a reutilizabilidade das funções dentro do programa: Poderíamos ter escrito uma função que adicionasse duas fracções e imprimisse o resultado em vez de o devolver, mas não a poderíamos reutilizar para calcular $x * (y + x)$, por exemplo.

Apesar disto, se quisermos reutilizar estas funções noutro programa que lide com fracções, a única forma seria fazer *copy & paste* para outro ficheiro. Isto implica redundância: código repetido, neste caso não no mesmo programa, mas em múltiplos programas. Essa situação não é nada aconselhável porque qualquer correcção ou extensão que façamos no futuro a estas funções deverá implicar a alteração do código de todos os programas em que estiverem copiadas. Uma tarefa ingrata e sujeita a falhas.

4.3 Calculadora de Fracções, versão 3: Programa + módulo tipo de dados + módulo de funções (passo 4_a)

Nesta versão, o programa foi dividido em 3 ficheiros, cada qual com a sua classe. A classe `Fraction`, anteriormente incluída no único ficheiro do programa, ganhou independência no seu próprio ficheiro. Deste modo a classe pôde ser qualificada como `public`, permitindo assim que qualquer programa cliente possa declarar variáveis e criar objectos do tipo `Fraction`.

```
public class Fraction {
  int num = 0;
  int den = 1;
}
```

As funções que operam sobre fracções foram também extraídas para uma nova classe `FractionOps` de forma a poderem ser reutilizadas facilmente. A classe foi também declarada `public` para ser acessível aos programas clientes e a outras classes externas (noutros ficheiros e possivelmente noutros directórios e pacotes).

Entretanto, aproveitámos para eliminar o prefixo `fraction-` dos nomes dos métodos. Esta é uma alteração relativamente superficial, mas simplifica um bocado o código e não introduz qualquer ambiguidade. De facto, como nos programas clientes a invocação dos métodos será sempre precedida do nome da classe (`FractionOps.something()`), não poderá haver qualquer confusão com outros métodos `something()` de outras classes.

```
public class FractionOps {
  static Fraction fromString(String str) {
    // (Corpo idêntico ao de fractionFromString na versão 2)...
  }

  static String toString(Fraction f) {
    ...
  }

  static Fraction multiply(Fraction a, Fraction b) {
    ...
  }

  static Fraction add(Fraction a, Fraction b) {
    ...
  }
}
```

Finalmente, na classe `FractionCalculator` ficou apenas a função principal do programa, quase inalterada. Bastou alterar a invocação das funções sobre fracções, recorrendo às novas designações qualificadas com o nome da nova classe a que agora pertencem. Por exemplo: `FractionOps.add(x, y)` em vez de: `fractionAdd(x, y)`.

```
public class FractionCalculator {
  public static void main(String[] args) {
    ...
    // 1) definir fracções (a partir dos argumentos);
    x = FractionOps.fromString( frac1 );
    y = FractionOps.fromString( frac2 );

    // 2) executar a operação;
    switch (op) {
      case "+":
        z = FractionOps.add(x,y); break;
      case "*":
        z = FractionOps.multiply(x,y); break;
    ...

    // 3) imprimir a operação solicitada e o resultado;
    out.printf("%s %s %s = %s\n",
        FractionOps.toString(x), op, FractionOps.toString(y), FractionOps.toString(z));
```

As classes `Fraction` e `FractionOps` são agora *módulos*: unidades independentes do programa principal, compiláveis separadamente e facilmente combináveis para produzir novos programas. Os módulos possibilitam assim a reutilização de código noutros programas sem introduzir redundância.

Isto é certamente um grande progresso, mas esta versão ainda tem alguns defeitos. Repare-se que todos os programas que usem operações de `FractionOps` precisam necessariamente de usar também o tipo de dados `Fraction`. Por outro lado, será raro um programa usar `Fraction` sem necessitar simultaneamente das operações definidas em `FractionOps`. Por outras palavras: estes dois módulos têm uma grande interdependência. Uma regra de boa programação diz que em geral um bom módulo deve ter poucas dependências externas e fortes dependências internas. A próxima versão resolve facilmente esse problema.

4.4 Calculadora de Fracções, versão 4: Programa + módulo tipo de dados e respectivas operações (passo 4_b)

Uma vez que `Fraction` e `FractionOps` são tão interdependentes, porque não juntá-los? No fim de contas uma classe não está limitada a ter apenas atributos ou apenas métodos; pode ter ambos. É isso que fazemos nesta versão: movemos as operações sobre fracções para a classe `Fraction`.

```
public class Fraction {
  int num = 0;
  int den = 1;

  // Métodos simplesmente copiados de FractionOps:
  static Fraction fromString(String str) { ... }
  static String toString(Fraction f) { ... }
  static Fraction multiply(Fraction a, Fraction b) { ... }
  static Fraction add(Fraction a, Fraction b) { ... }
```

O módulo `Fraction` é agora uma entidade coesa internamente e desacoplada do programa principal; características que indicam uma boa modularidade da solução.

Naturalmente que a classe `FractionOps` deixou de ter interesse e foi simplesmente suprimida. No programa principal só temos que substituir as invocações de métodos `FractionOps.something()` por `Fraction.something()`.

```
public class FractionCalculator {
  public static void main(String[] args) {
    ...
    x = Fraction.fromString( frac1 );
    y = Fraction.fromString( frac2 );

    switch (op) {
      case "+":
            z = Fraction.add(x,y); break;
    ...

    out.printf("%s %s %s = %s\n",
        Fraction.toString(x), op, Fraction.toString(y), Fraction.toString(z));
```

A alteração introduzida nesta versão, aparentemente cosmética, tem na verdade fortes implicações na forma de construir e organizar os programas.

Até agora tínhamos classes que serviam de meros invólucros para os métodos (funções) que compunham os nossos programas; ou então tínhamos classes que continham apenas atributos (campos) e que serviam para definir novos tipos de dados estruturados (designados registos). Pela primeira vez temos uma classe que contém tanto atributos (dados) como métodos (operações), e que nos serve para definir um novo tipo de dados, mas com um "comportamento" e um "significado" que lhe são conferidos pelas operações e propriedades associadas.

A associação das operações ao tipo de dados faz todo o sentido. Na verdade, um tipo de dados não é definido meramente pelos atributos que o compõem: uma fracção não é apenas um par de inteiros. Um ponto num gráfico também pode ser representado por um par de inteiros, mas não é do mesmo tipo. O que nos permite distinguir a noção de fracção da de ponto são as diferentes operações que se podem aplicar sobre esses dois tipos de dados (podem multiplicar-se fracções, mas o que significa multiplicar pontos?); bem como as propriedades distintas que essas operações e dados respeitam (a ordenada de um ponto pode ser zero, mas o denominador de uma fracção não; os pontos $(2,3)$ e $(4,6)$ não são iguais, mas as fracções $\frac{2}{3}$ e $\frac{4}{6}$ são).

4.5 Calculadora de Fracções, versão 5: Programa + Objectos (passos 5 e 6)

Nesta versão introduzimos métodos de objecto (não `static`) e encapsulamento de dados (qualificadores `public` e `private`). Também são introduzidos alguns métodos novos e construtores.

4.5.1 Métodos de objecto

Na versão anterior verificámos que quase todos os métodos da classe `v4.Fraction` tinham pelo menos um parâmetro do próprio tipo `Fraction`. Esse é afinal um sintoma da forte coesão entre dados e

operações que justificou a sua associação num mesmo módulo. Esse parâmetro comum (`f`) é invariavelmente a fracção da qual se vai extrair alguma informação (`Fraction.toString(f)`) ou sobre a qual se vai aplicar alguma operação (`Fraction.add(f,b)`). A linguagem `Java` permite-nos realçar esta associação das operações ao objecto a que se aplicam usando métodos de objecto (não `static`). Assim, nesta versão a maioria dos métodos deixou de ser `static`.

Que implicações tem isso para a classe?

- Os métodos perdem um parâmetro do tipo `Fraction`, que é substituído pelo objecto ao qual se aplica implicitamente a operação. Ver, por exemplo, o método `toString()`.

```
// ANTES: public static String toString(Fraction f)  ...
public String toString() {
  ...
```

- No corpo desses métodos, o acesso aos atributos desse objecto implícito passa a fazer-se referindo directamente os nomes desses atributos, (`num` e `den`, em vez de `f.num` e `f.den`).

```
public String toString() {
  String s = "(";
  if (den>0) {    // ANTES tinha:  if (f.den>0)
    s += (num) + "/" + (den);
  } else {
    s += (-num) + "/" + (-den);
  }
  s += ")";
  return s;
}
```

- Se quisermos, podemos referir-nos ao objecto implícito como `this` e aos seus atributos como `this.num` e `this.den`. Nalguns casos pode mesmo ser necessário fazê-lo dessa forma para evitar ambiguidade com parâmetros ou variáveis locais com o mesmo nome.

```
public void set(int num, int den) {
  assert den != 0;
  this.num = num;    // num é o parâmetro, this.num é o atributo
  this.den = den;
}
```

Podemos optar por manter o método `fromString` como `static`, ou então fazer com que seja uma inicialização do objecto (tal como o método `set`) a partir de uma `String`.

Quais as consequências no programa cliente? Os métodos não estáticos passam a aplicar-se aos objectos do tipo Fraction, por exemplo: `x.toString()`; em vez de serem "aplicados" à classe, passando o objecto como parâmetro: `Fraction.toString(x)`.

Que vantagens tem isso?

- O objecto a que se aplica a operação fica claramente identificado, em vez de ser apenas um parâmetro.
- Consultas como `y.toString()` ficam sintacticamente semelhantes a meros acessos a atributos.
- Podemos considerar os novos métodos como uma extensão do conceito de "campo" de um registo: passámos a ter campos calculados em vez de armazenados.

Algum código beneficia bastante com a mudança.

Como decidir se um método deve ser de classe (`static`) ou de objecto? Primeiro que tudo deve ter-se em conta as restrições que cada um desses tipos de métodos implica (ver tabela 1). Mesmo assim, por vezes poderá haver várias alternativas para implementar uma certa funcionalidade, ora através de métodos de objecto, ora através de métodos de classe (`static`). A escolha deve ter em conta a clareza, a facilidade de utilização, a extensibilidade e a coerência do conjunto.

	Método de objecto (não `static`)	**Método de classe** (`static`)
Definição	`public class Fraction {` `  void f() {...}`	`public class Fraction {` `  static void g() {...}`
Implementação	Tem acesso a tudo na classe: métodos e atributos `static` ou não. Pode usar o parâmetro implícito `this`.	Só tem acesso directo a métodos e atributos `static`. Não pode usar `this`.

Invocação na mesma classe	`f();  // ou`
	`this.f();`
		`g();  // ou`
		`Fraction.g();`

Invocação noutra classe	`Fraction x = new Fraction();`
	`x.f();`
		`Fraction.g(); // ou`
		`x.g();`

Tabela 1: Restrições na implementação e utilização de métodos de objecto e de classe (`static`).

4.5.2 Encapsulamento dos dados

A segunda alteração introduzida neste módulo foi a qualificação dos atributos como privados (`private`).

```
public class Fraction {
  private int num = 0;
  private int den = 1;
  ...
```

Isso implica que só os métodos desta classe lhes podem aceder. Qualquer classe externa (como o programa principal) não os pode modificar nem observar directamente.

Os restantes membros da classe foram declarados `public` de forma a poderem ser usados por qualquer outra classe, mesmo de fora do pacote.

Também aproveitámos para definir novos métodos, `num()` e `den()`, necessários para permitir a consulta dos atributos entretanto protegidos; e `set(int,int)` para permitir a atribuição de um novo valor à fracção.

```
  public int num() { return num; }
  public int den() { return den; }
  public void set(int num, int den) {
    assert den != 0; // importante, como se verá na próxima aula!
    this.num = num;
    this.den = den;
  }
```

Estes métodos permitem que os programas clientes possam continuar a manipular fracções com detalhe, mas agora apenas de forma segura: note-se a pré-condição exigida pelo método `set()`.

Temos agora um módulo muito mais robusto, que garante a coerência dos seus dados, agora protegidos pelo qualificador `private`.

Claro que continua a ser possível ter atributos `public`, mas o programador deve ponderar bem as consequências. Também há lugar a métodos `private`, se só fizerem sentido para uso interno da classe. No entanto, o mais usual é que todos os atributos sejam `private` e os métodos sejam `public`, eventualmente com algumas excepções.

Que efeitos tem esta alteração num programa cliente (como `FractionCalculator`)? A consequência importante é que passou a ser impossível alterar directamente os atributos de uma fracção. Por exemplo, deixou de ser possível fazer:

```
  x.num = 1; x.den = n;  // NÃO PODE fazer assim!
```

que corria o risco de deixar em `x` uma representação inválida (se `n==0`), o que se iria repercutir em resultados incorrectos, mas que poderiam passar despercebidos durante muito tempo, sabe-se lá com que consequências. Agora se quisermos modificar uma fracção teremos de usar um método criado especificamente com esse propósito:

```
x.set(1, n);
```

o que tem a vantagem de interromper a execução caso falhe a pré-condição `n!=0`, evitando propagar o erro ao resto do programa.

Além disso, para podermos observar o numerador e o denominador de uma dada fracção também é necessário agora recorrer a métodos de consulta criados com esse propósito: `num()` e `den()`. É um pequeno preço a pagar pela segurança acrescida que ganhámos.

Temos agora a possibilidade de garantir a coerência das entidades do tipo `Fraction` protegendo devidamente os acessos aos seus métodos com as asserções apropriadas.

4.5.3 Construtores

Finalmente, aproveitámos para definir alguns construtores úteis:

```
public Fraction(int num, int den) {
  assert den != 0;
  this.set(num, den);
}
```

que permite criar uma fracção dados os seus numerador e denominador;

```
public Fraction(String str) {
  int p = str.indexOf('/');
  if (p>=0) {
    num = Integer.parseInt(str.substring(0,p));
    den = Integer.parseInt(str.substring(p+1));
  } else {
    num = Integer.parseInt(str);
    den = 1;
  }
  assert den != 0;
}
```

para criar um fracção a partir de uma representação em forma de texto; e ainda o construtor sem parâmetros

```
public Fraction() {}
```

que cria simplesmente uma fracção com o valor $\frac{0}{1}$.

Repare-se que se evitou introduzir qualquer redundância: o primeiro destes construtores recorre à função `set()` e, por outro lado, o segundo construtor ficou com o código do método `fromString()`, que agora foi reimplementado para invocar esse construtor.

```
static Fraction fromString(String str) {
  return new Fraction(str);
}
```

Esta é a última versão deste programa e do módulo `Fraction` (para já). Ao longo das várias versões, mostrámos:

1. Um programa monolítico. (v1)
2. Estruturação por abstracção algorítmica (funções) e de dados (registos). (v2)

3. Separação em módulos para potenciar a reutilizabilidade. (v3)
4. Agregação de dados e operações para dar mais coesão ao módulo. (v4)
5. Introdução de métodos de objecto e protecção dos seus dados. (v5)

Certifique-se que percebeu bem estes conceitos e os aspectos técnicos associados.

Daqui em diante produziremos programas e módulos análogos aos que temos na versão 5. Não faz sentido percorrer de novo todos os passos que demos aqui, a não ser pelo interesse pedagógico, claro.