Introdução à Programação

Recursão e Geração de Exceções

Recursão

decoração
Substantivo feminino.
1.Ato ou efeito de **decorar**

decorar
Verbo transitivo direto.
1.Guarnecer com adorno(s); dispor formas e cores em; ornamentar, embelezar;
2.Realçar, avivar, adornar;

Fonte: Dicionário Aurélio

Tópicos da Aula

- Hoje, aprenderemos a usar uma técnica de programação chamada de recursão
  - Conceito de Recursão
  - Definições Recursivas
  - Casos Base e Geral
  - Programando Recursivamente
  - Recursão X Laço
- Finalmente, aprenderemos como Gerar Exceções
  - Criando exceções e lançando com cláusula *throw*

Recursão

- Uma definição **recursiva** de um conceito consiste em utilizar o próprio conceito na definição
  - Ex: fat(n) = n * fat(n-1)
- Definições recursivas em linguagem natural não são, geralmente, muito úteis
- Contudo, em outras situações, uma definição recursiva pode ser a mais apropriada para explicar um conceito
- **Recursão** é uma técnica de programação que pode fornecer soluções elegantes para determinados problemas

Mas, antes, deve-se aprender a pensar recursivamente!

Definições Recursivas: *Lista*

- Considere a seguinte lista de números

24, 88, 40, 37

- Podemos definir uma lista de números como:

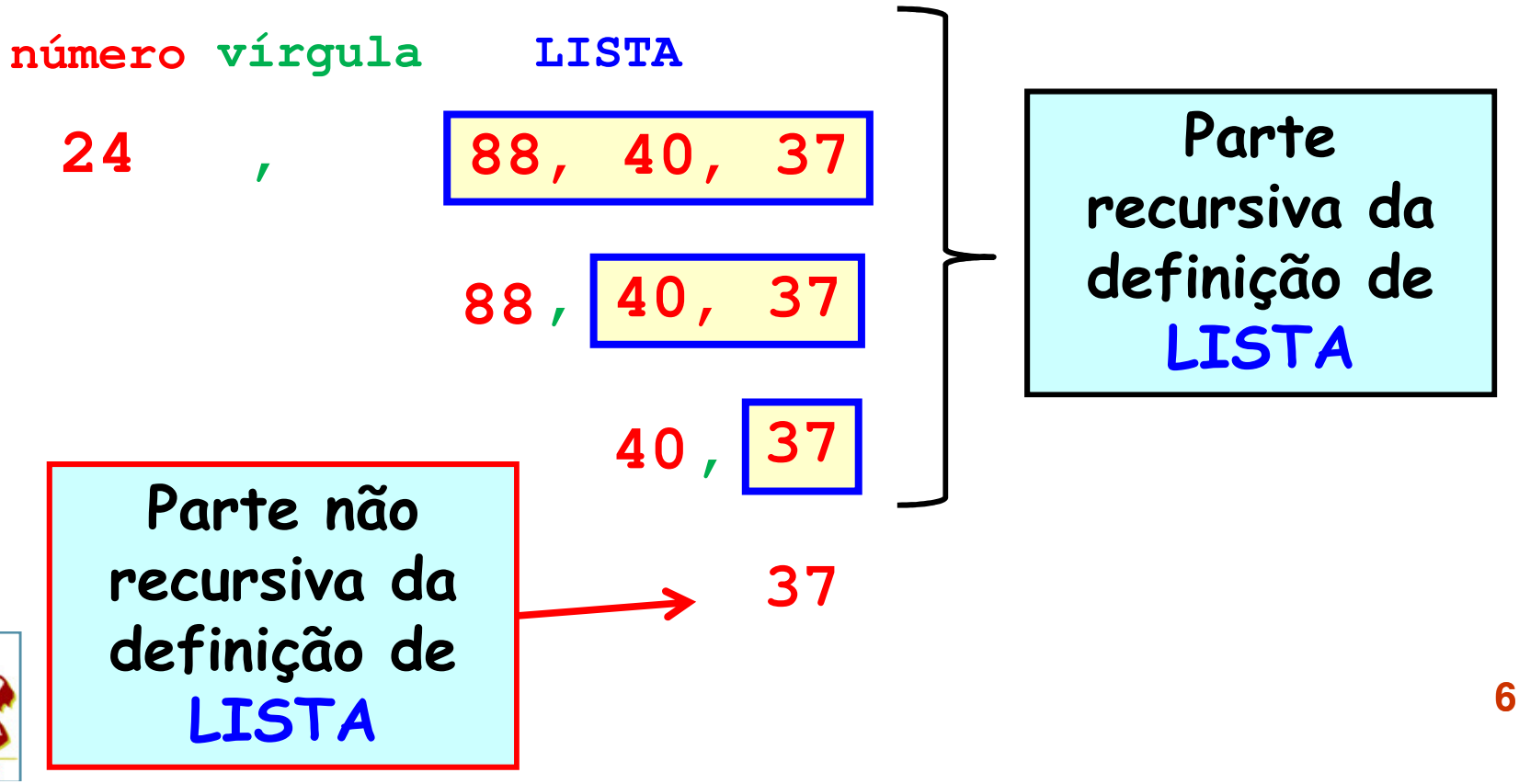
Uma **LISTA** é um: **número**
ou um: **número** **vírgula** **LISTA**

Resumindo: uma LISTA é definida ou como um único número, ou como um número seguido de uma vírgula seguida de uma LISTA

Utilizamos LISTA para sua própria definição

Definições Recursivas: *Lista*

A parte recursiva da definição de **LISTA** é utilizada diversas vezes, até que se possa utilizar a parte não recursiva



Definições Recursivas: *Fatorial*

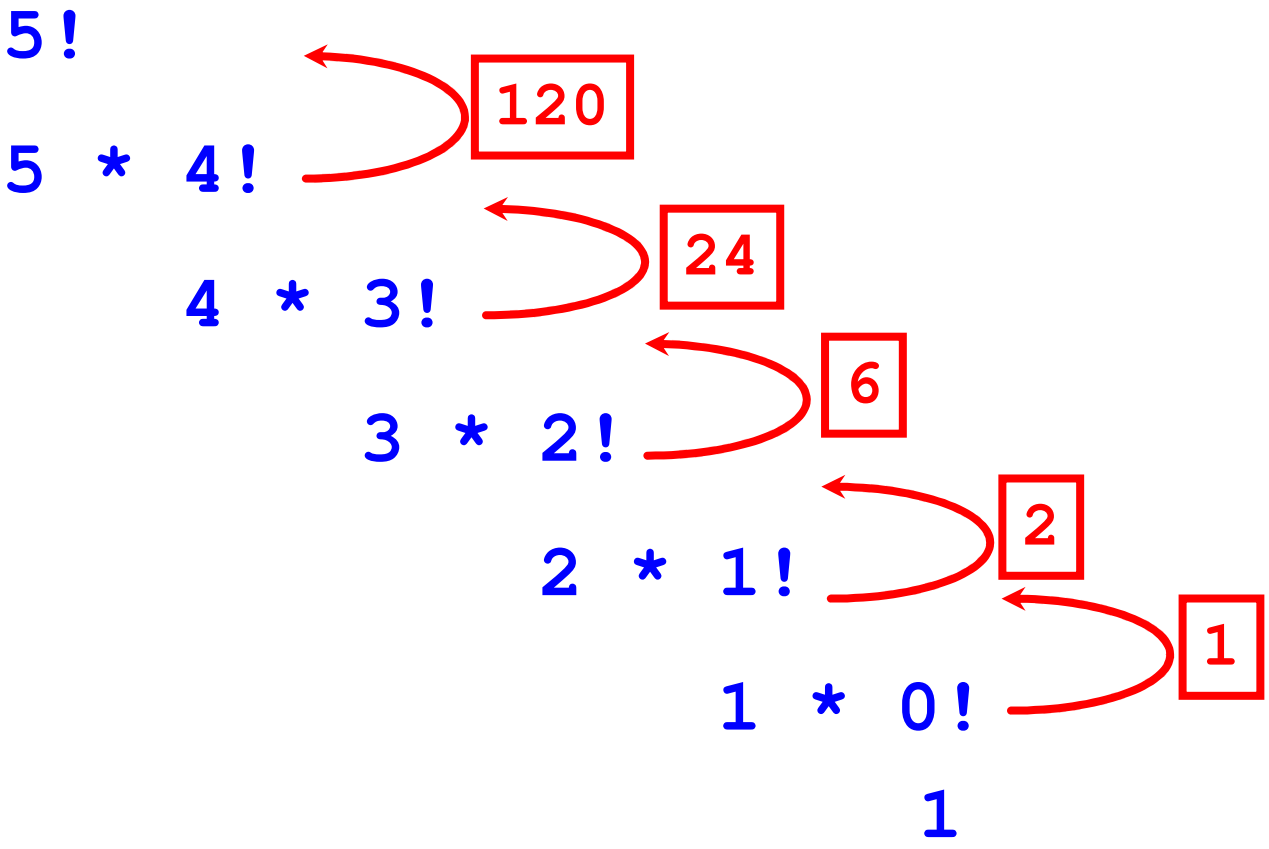
- **Fatorial** de um número N (N!), que é inteiro e positivo, é definido como a multiplicação de todos os inteiros compreendidos entre 1 e N (inclusive)

- Podemos definir fatorial recursivamente:

```
0! = 1
N! = N x (N-1)!
```

Mais uma vez, utilizamos fatorial para sua própria definição

Definições Recursivas: *Fatorial*



Caso Base e Caso Geral

- Toda definição recursiva deve ter uma parte recursiva e outra não recursiva
- Se não houver a parte não recursiva, o caminho recursivo nunca termina
  - Gera uma recursão infinita
  - Similar a um laço infinito
- Denomina-se de **caso base**, a parte não recursiva da definição, enquanto o **caso geral** é a parte recursiva

Programação Recursiva

- Um método em Java pode chamar ele mesmo
  - Método recursivo
- O código de um método recursivo **deve** tratar o caso base e o caso geral
- Cada chamada do método cria novos parâmetros e variáveis locais
- Como em qualquer chamada de método, assim que o método termina sua execução, o controle retorna para o método que o chamou (que neste caso, é ele mesmo)

Exemplo de Método Recursivo: *Fatorial*

```
int fatorial(int n) {
     int resultado;

     if (n == 0)
          resultado = 1;

     else
          resultado = n * fatorial(n - 1);

     return resultado;
}
```

Caso Base

Caso Geral

Recursão x Laço

- Não é porque existe uma solução recursiva, que sempre devemos usá-la
- Soluções iterativas (com laço) são geralmente mais eficientes
  - Soluções recursivas geralmente demandam mais poder de processamento e memória
- Porém, soluções utilizando recursão podem ser mais elegantes e mais compreensíveis que soluções iterativas equivalentes
- Programador deve analisar caso a caso qual é a melhor técnica a aplicar para a resolução do problema

Revendo Exceções

```
class Conta {
   private String numero;
   private double saldo;
   /* ... */
   void debitar(double valor) {
        saldo = saldo - valor;
   }
}
```

Como evitar débitos maiores que o saldo da conta?

Desconsiderar Operação

```
class Conta {
   private String numero;
   private double saldo;
   /* ... */
   void debitar(double valor) {
      if (valor <= saldo)
         saldo = saldo - valor;
   }
}
```

Não faz nada se `saldo` for menor que `valor`

Desconsiderar Operação

- Problemas:
  - quem solicita a operação não tem como saber se ela foi realizada ou não
  - nenhuma informação é dada ao usuário do sistema

Mostrar Mensagem de Erro

```
class Conta {
    private String numero;
    private double saldo;
    /* ... */
    void debitar(double valor) {
        if (valor <= saldo)
            saldo = saldo - valor;
        else {
            MiniJavaSystem s;
            s = new MiniJavaSystem();
            s.print("Saldo Insuficiente");
        }
    }
}
```

Imprime no console mensagem de erro

Mostrar Mensagem de Erro

- Problemas:
  - Informação é dada ao usuário do sistema, mas nenhuma sinalização de erro é fornecida para métodos que invocam `debitar`
  - Há uma forte dependência entre a classe `Conta` e sua interface com o usuário

Retornar Código de Erro

```
class Conta {
  private String numero;
  private double saldo;
  /* ... */
  boolean debitar(double valor) {
    boolean r = false;
    if (valor <= saldo) {
        saldo = saldo - valor;
        r = true;
    }else
        r = false;
    return r;
  }
}
```

Muda-se a assinatura do método

Retornam `true` ou `false` dependendo do sucesso da operação

Retornar Código de Erro

- Problemas:
  - Dificulta a definição e o uso do método
  - Métodos que invocam `debitar` têm que testar o resultado retornado para decidir o que deve ser feito
  - A dificuldade é maior para métodos que retornam valores:
    - Se `debitar` já retornasse um outro valor qualquer? O que teria que ser feito?

Código de Erro: Problemas

```
class Conta {
  private String numero;
  private double saldo;
  /* Método retorna o saldo depois do
débito*/
  double debitar(double valor) {
    if (valor <= saldo) {
        saldo = saldo - valor;
        return saldo;
    }else
        return -9999;
    }
}
```

Uma hora retorna o saldo

Outra hora retorna código de erro

Definindo Exceções em Java

- Em vez de códigos, teremos **exceções**...
- São objetos comuns, portanto têm que ter uma classe associada
- Classes representando exceções herdam e são subclasses de `Exception` (pré-definida)
- Define-se subclasses de `Exception` para
  - oferecer informações extras sobre a falha, ou
  - distinguir os vários tipos de falhas

Definindo Exceções

```
public class SIException extends Exception {
   private double saldo;
   private String    numero;
   public SIException (double s, String n) {
      saldo = s;
      numero = n;
   }
}
```

```
public class NumNegException extends Exception{

}
```

Definindo Métodos com Exceções

```
class Conta {
  /* ... */
  void debitar(double v) throws SIException {
    if (v <= saldo)
        saldo = saldo - v;
    else {
      SIException e;
      e = new SIException(saldo,numero);
      throw e;
    }
}
```

Declara na assinatura que pode lançar exceção

Cria objeto da Exceção

Lança Exceção

Definindo Métodos com Exceções

```
class Conta {
  /* ... */
  void debitar(double v)throws
                SIException, NumNegativoException {
    if (v < 0){
        throw new NumNegativoException();
     } else {
          if (v <= saldo)
               saldo = saldo - v;
          else {
               SIException e;
               e = new SIException(saldo,numero);
               throw e;
          }
    }
  }
```

Declara mais de uma exceção

Cria e lança objeto da Exceção

Definindo e Levantando Exceções

- Um método pode levantar mais de uma exceção
- Todos os tipos de exceções levantadas no corpo de um método devem ser declaradas na sua assinatura, separadas por vírgula
- Levantando exceções: `throw` *obj-exceção*
- Fluxo de controle e exceção são passados para o código que chamou o método que lançou a exceção

Resumindo ...

- Recursão
  - Conceito
  - Exemplos de definições recursivas
  - Casos Base e Geral
  - Programando Recursivamente
  - Recursão X Laço
- Geração de Exceções
  - Criando exceções
  - Cláusula throw