UNIRIO

Enumerações, exceções (parte 1)

Aula 6

BSI — 2018.2

Jefferson Elbert Simões

CCET/DIA

6 de setembro de 2018

Exercício

1. Implemente uma função `nomeDoDia` que receba por parâmetro um número inteiro entre 1 e 7 e retorne o nome do dia da semana correspondente.

Exercício

1. Implemente uma função `nomeDoDia` que receba por parâmetro um número inteiro entre 1 e 7 e retorne o nome do dia da semana correspondente.

Solução:

```
public static String nomeDoDia( int num ) {
    if( num == 1 ) return "domingo";
    else if( num == 2 ) return "segunda-feira";
    ...
}
```

Exercício

1. Implemente uma função `nomeDoDia` que receba por parâmetro um número inteiro entre 1 e 7 e retorne o nome do dia da semana correspondente.

Solução:

```
public static String nomeDoDia( int num ) {
    switch( num ) {
        case 1 :
            return "domingo";
        case 2 :
            return "segunda-feira";
        ...
    }
}
```

Enumerações

- Tipos cujas variáveis devem pertencer a um certo conjunto de constantes pré-definidas
- Exemplos:
  - Dias da semana
  - Pontos cardeais
  - Opções em um menu

Enumerações

- Em Java, enumerações são classes declaradas com uma sintaxe especial
  - Possuem métodos especiais (automaticamente gerados)

```
public enum DiaSemana {
    DOMINGO, SEGUNDA, TERCA, QUARTA,
    QUINTA, SEXTA, SABADO;
}

// em outro ponto do código...
for( DiaSemana dia : DiaSemana.values() ) {
    System.out.println( dia.name() );
}
```

Enumerações

Enumerações podem ter atributos, métodos, inclusive métodos estáticos, e construtores

```
public enum Planeta {
    MERCURIO (3.303e+23, 2.4397e6),  VENUS   (4.869e+24, 6.0518e6),
    TERRA    (5.976e+24, 6.37814e6), MARTE   (6.421e+23, 3.3972e6),
    JUPITER  (1.9e+27,   7.1492e7),  SATURNO (5.688e+26, 6.0268e7),
    URANO    (8.686e+25, 2.5559e7),  NETUNO  (1.024e+26, 2.4746e7);
    private double massa; // em quilogramas
    private double raio;  // em metros
    Planeta (double massa, double raio) {
        this.massa = massa;
        this.raio = raio;
    }
    private static double G = 6.673E-11; // grav.universal (m3 kg-1 s-2)
    public double gravidadeSuperficie () {
        return G * massa / (raio * raio); }
    public double pesoSuperficie (double massaCorpo) {
        return massaCorpo * gravidadeSuperficie(); }
}
```

Enumerações

- Vantagens:
  - Substitui números/strings por objetos nomeados — aumenta a legibilidade do código
  - Evita o uso de `switch/case`'s para obter valores pré-fixados
  - Evita o uso de nomes incorretos — erros são detectados em tempo de compilação
  - Garante que não serão instanciados objetos "copiados"
    - Podemos comparar referências!
    - Podemos testar por objetos específicos em `switch/case`
- ATENÇÃO: todos os objetos existentes devem ser declarados na enumeração e são conhecidos em tempo de compilação

Exercício

1. Implemente uma função `nomeDoDia` que receba por parâmetro um número inteiro entre 1 e 7 e retorne o nome do dia da semana correspondente.

Exercício

1. Implemente uma função `nomeDoDia` que receba por parâmetro um número inteiro entre 1 e 7 e retorne o nome do dia da semana correspondente.

Solução:

```
public enum DiaSemana {
    DOMINGO(1, "domingo"),
    SEGUNDA(2, "segunda-feira"),
    ...
    private int valor;
    private String nome;
    DiaSemana( int valor, String nome ) {
        this.valor = valor;
        this.nome = nome;
    }
    public int getValor() {return valor;}
    public String getNome() {return nome;}
}
```

Exercício

1. Implemente uma função `nomeDoDia` que receba por parâmetro um número inteiro entre 1 e 7 e retorne o nome do dia da semana correspondente.

Solução:

```
public static String nomeDoDia( int num ) {
    for( DiaSemana dia : DiaSemana.values() )
        if( dia.getValor() == num )
            return dia.getNome();
}
```

Este método deve pertencer à classe `DiaSemana`! (por quê?)

Classes aninhadas

- Enumerações frequentemente são implementadas como classes aninhadas
  - Classes "dentro de" outras classes
  - Classe interna é membro da classe externa, da mesma forma que atributos e métodos
- Objetivos:

  Modularização: Permite agrupar classes com contexto semelhante

  Encapsulamento: Classe interna fica oculta do restante do código
- Classe interna pode ser `public` ou `private`
  - Se classe interna for pública, outras classes além da externa podem acessá-la

Classes aninhadas

Classe interna pode ser `static` ou não

`static`: existe uma única classe interna, comum a todos os objetos da classe externa

não `static`: cada objeto da classe externa possui sua "cópia" da classe interna

Cuidado para não confundir classes `static` com atributos `static`!

- Sintaxe pode ficar complexa e confusa...

Classes aninhadas

Quando você deve/irá usar classes aninhadas?

Classes aninhadas

Quando você deve/irá usar classes aninhadas?

- Quase nunca! Sintaxe é confusa e geralmente o custo/benefício é baixo!
  - (você pode trocar por classe não-aninhada privada, por exemplo)

Classes aninhadas

Quando você deve/irá usar classes aninhadas?

- Quase nunca! Sintaxe é confusa e geralmente o custo/benefício é baixo!
  - (você pode trocar por classe não-aninhada privada, por exemplo)

No entanto, existem alguns tipos especiais (avançados) de classes aninhadas que aparecem com frequência:

- Classes locais
  - Declaradas dentro de um bloco (e.g.: método, laço)
- Classes anônimas
  - Implementações de interfaces que declaram a classe e instanciam um objeto ao mesmo tempo

Exceções

- Exceções são uma forma mais sistemática e menos intrusiva de lidar com comportamentos imprevistos do código
- Nativo em várias linguagens: diversos bugs são percebidos através de exceções que eles causam

  `Null pointer exception:` tentar acessar um objeto através de uma referência nula

  `Index out of bounds exception:` tentar acessar uma posição inexistente em um array

Exceções

- Uma exceção é uma representação de um problema que ocorreu durante a execução de um comando
- Ideia: exceções acontecem raramente (=P)
  - Por que complicar o fluxo normal do código para tratar esses casos?
  - Muita perda de legibilidade, "pouco" ganho de bom comportamento

Exceções

Como funciona o mecanismo de exceções?

- Programas possuem uma pilha de execução
- Quando ocorre um comportamento imprevisto em uma função, ela não deixa o problema acontecer e deve, em vez disso, disparar uma exceção
- O fluxo do código é interrompido, e o controle do programa passa para um tratador de exceção
  - Bloco de código de uma função projetado para lidar com esta exceção somente se ela acontecer
- Caso a função no topo da pilha não tenha um tratador de exceção, ela é removida da pilha e o processo se repete para a função seguinte
  - Apenas se nenhuma função tratar a exceção, o programa é interrompido

Exceções

Em Java:

- `try`: bloco de código que pode gerar uma exceção
- `catch`: bloco de código que trata exceções de um certo tipo em um bloco `try`
  - Cada bloco `try` pode ser seguido de um ou mais blocos `catch`
- `finally`: bloco de código após o bloco `try` que sempre é executado, independente de qual exceção ocorreu (ou se ocorreu alguma)
  - Em geral, utilizado para liberar recursos (e.g.: fechar arquivos ou conexões de rede)
- `throw`: comando que dispara uma exceção

Exceções

Exemplo:

```
public class Aluno {
    private ArrayList<Float> historico;
    Aluno() {
        historico = new ArrayList<>();
    }
    public void adicionarNota( float nota ) {
        historico.add( nota );
    }
    public float calculaCR() {...}
}
```

Exceções

Exemplo:

```
public class Aluno {
    private ArrayList<Float> historico;
    Aluno() {
        historico = new ArrayList<>();
    }
    public void adicionarNota( float nota ) {
        historico.add( nota ); // e se nota < 0, ou nota > 10 ?
    }
    public float calculaCR() {...}
}
```

Exceções

Exemplo:

```
public class Aluno {
    private ArrayList<Float> historico;
    Aluno() {
        historico = new ArrayList<>();
    }
    public void adicionarNota( float nota ) {
        if( (nota < 0) || (nota > 10) )
            throw new IllegalArgumentException( "Nota inválida" );
        historico.add( nota );
    }
    public float calculaCR() {...}
}
```

Exceções

Exemplo:

```
public class Aluno {
    private ArrayList<Float> historico;
    Aluno() {...}
    public void adicionarNota( float nota ) {...}
    public float calculaCR() {
        int qtdNotas = 0; float soma = 0.0f, CR = 0.0f;
        for( float nota :  historico ) {
            soma += nota;
            qtdNotas++;
        }
        CR = soma/qtdNotas;
        return CR;
    }
}
```

Exceções

Exemplo:

```
public class Aluno {
    private ArrayList<Float> historico;
    Aluno() {...}
    public void adicionarNota( float nota ) {...}
    public float calculaCR() {
        int qtdNotas = 0; float soma = 0.0f, CR = 0.0f;
        for( float nota :  historico ) {
            soma += nota;
            qtdNotas++;
        }
        CR = soma/qtdNotas; // e se qtdNotas == 0 ?
        return CR;
    }
}
```

Exceções

Exemplo:

```
public class Aluno {
    private ArrayList<Float> historico;
    Aluno() {...}
    public void adicionarNota( float nota ) {...}
    public float calculaCR() {
        int qtdNotas = 0; float soma = 0.0f, CR = 0.0f;
        for( float nota :  historico ) {
            soma += nota;
            qtdNotas++;
        }
        try {
             CR = soma/qtdNotas;
        } catch( ArithmeticException e ) {
            CR = 0.0f;
        }
        return CR;
    }
}
```

Exceções

Algumas exceções úteis:

`ArrayIndexOutOfBoundsException`

`IllegalArgumentException`: parâmetro com valor inesperado (e.g.: sacar quantia negativa)

`IllegalStateException`: objeto em estado inesperado (e.g.: tentar abrir um arquivo duas vezes)

`NotImplementedException`: método não implementado

`NullPointerException`

`NumberFormatException`: tentar converter uma String mal formatada para um número

`ParseException`: falha ao realizar parsing de uma String mal formatada

Exceções

Em Java existem dois tipos de exceções:

não-verificada Utilizadas em geral devido a erros de programação; podem acontecer em quase qualquer lugar

- Inclui todos os exemplos do slide anterior

verificada Ressaltam comportamentos pontuais imprevistos

- Estudaremos daqui a algumas aulas

Exceções

Vantagens do uso de exceções:

- Separa a detecção do problema do seu tratamento
  - Podem acontecer em funções diferentes
- Evita o uso de código de erro
  - Padrão de construção de métodos: retorno da função indica o erro que aconteceu (geralmente do tipo `bool` ou algum `Enum` criado para isto)
  - Somente pode ser usado se a função já não iria retornar nada

Exercícios

Crie enumerações para representar os seguintes conceitos:

1. Meses de um ano — inclua o nome do mês como um atributo;
2. Os estados do Brasil — inclua atributos como nome, sigla, nome da capital e ano de federação;
3. As cartas de um baralho — inclua como atributos seu valor de face e seu naipe[1];
4. Os 151 Pokémon da primeira geração — escolha você os atributos mais interessantes ;)

[1] Como representar um valor de face ou um naipe? Um `int`? Uma `String`? Ou tem alguma forma melhor?

Exercícios

5. No exercício 3, pedimos para você representar cartas de um baralho como objetos de uma enumeração. Por que isto, na verdade, não é adequado?

Exercícios

6. Escreva uma classe que represente números racionais e crie um método estático para dividir dois números racionais. Utilize o mecanismo de exceções na implementação desta classe para lidar com divisões por zero (lembre-se que zero é um número racional válido).

Créditos

- Alguns exemplos retirados de documentação oficial do Java e traduzidos