Programação I

Membros estáticos, Constantes e Exceções

Samuel da Silva Feitosa

Aula 12

Membros Estáticos

- Há situações em que é desejável compartilhar informações entre as várias instâncias de uma classe.
  - Constante, identificação, contagem ou totalização.
  - Este efeito pode ser obtido com o modificador static.

```
// variável inteira e estática
public static int numero = 0;
```

- Métodos também podem ser estáticos.

```
// método estático com retorno inteiro
public static int getNumero() {
    return numero;
}
```

Exemplo - Membros estáticos

```
public class Dobro {
    // Armazena número de instâncias da classe
    private static int instancias = 0;
    // Armazena o último valor usado pelo método 'dobro'
    public int ultimoValor;
    // Construtor que atualiza número de instâncias
    public Dobro() { instancias++; }
    // Retorna número de instâncias
    public static int getInstancias() { return instancias; }
    // Calcula o dobro
    public int dobro(int valor) {
        ultimoValor = valor;
        return 2 * valor;
    }
}
```

Uso de membros estáticos

- Membros não estáticos são acessíveis apenas pelas instâncias das classes.
- Membros estáticos podem ser usados por meio de suas classes ou de instâncias.

```
// Uso direto de método estático
int n1 = Dobro.getInstancias();
// Uso de método estático a partir de instância
Dobro dobro = new Dobro();
int n2 = dobro.getInstancias();
```

- Em ambos os casos, os modificadores de acesso são respeitados (public/private).
- Membros não estáticos só podem ser usados através de instâncias.

Constantes

- É possível declarar **constantes** pela combinação do modificador **static** com o modificador **final**.
  - **static** define a criação de membros estáticos.
  - **final** indica que o membro não pode ser modificado.

```
public final static int MAX = 10;
public final static double PHI = 0.667;
```

- A inicialização de constantes deve ocorrer na declaração e não pode ser modificada depois.

Finalizadores e Coleta de Lixo

- Criação de objetos em Java é realizada por meio de construtores.
- Destruição dos objetos é automática através do Garbage Collector.
  - Ele procura por objetos não mais referenciados, os quais provavelmente não serão mais usados.

```
// for define um novo bloco
for (int i = 0; i < 100; i++) {
    // com criação de novo objeto,
    // a referência anterior é perdida
    Object obj = new Object();
    // uso do objeto
}
// todos os objetos criados no bloco serão eliminados
```

Finalizadores e Coleta de Lixo

- É possível forçar a remoção de objetos pelo Garbage Collector.

```
// aciona coletor de lixo para tentar
// a remoção de objetos não usados
System.gc();
```

- Quando o objeto precisa devolver recursos alocados ou encerrar conexões.
  - É possível implementar o método **finalize**.

```
protected void finalize() {
    // devolução de recursos ou
    // encerramento de conexões
}
```

Exemplo - Finalizadores

```
public class Objeto {
    public static int instancias = 0;
    private int id;
    public Objeto() {
        id = instancias++;
        System.out.println("Objeto.Objeto() [id=" + id + "]");
    }
    public static int getInstancias() { return instancias; }
    public int getId() { return id; }
    @Override
    protected void finalize() {
        instancias--;
        System.out.println("Objeto.finaliza() [id=" + id + "]");
    }
}
```

Exemplo - Garbage Collector

```
public static void main(String[] args) {
    System.out.println("Instâncias: " + Objeto.getInstancias());
    for (int i = 0; i < 10; i++) {
        Objeto obj = new Objeto();
    }
    System.out.println("Instâncias: " + Objeto.getInstancias());
    System.gc(); // aciona o garbage collector
    System.out.println("Instâncias: " + Objeto.getInstancias());
}
```

Estruturas de Controle de Erros

- Java oferece os construtores **try**, **catch** e **finally**.
  - Propósito de separar código que executa as tarefas desejadas das rotinas de tratamento de erros.
  - Evita que o programador tenha de fazer testes de verificação antes da realização de operações.
- Trecho de código monitorado automaticamente.
- Erros no Java são sinalizados através de exceções (*exceptions*).
  - Objetos especiais que carregam informações sobre o erro detectado.
  - É possível criar suas próprias exceções.

Tratamento de Exceções - Sintaxe

- Cada bloco **try** precisa ter pelo menos um bloco **catch** ou **finally**.

```
try [(recurso = inicialização)] {
   diretiva_normal;
}
catch( <exception1> ) {
   diretiva_de_tratamento_de_erro1;
}
[ catch( <exception2> ) {
   diretiva_de_tratamento_de_erro2;
} ]
[ finally {
   diretiva_de_execução_garantida;
} ]
```

Exemplo (1)

- Esta aplicação apresenta uma contagem regressiva baseada no valor fornecido como primeiro argumento.
  - Se o argumento não for passado, ocorrerá uma exceção *java.lang.ArrayIndexOutOfBoundsException*.

```
public static void main(String[] args) {
    int j = Integer.parseInt(args[0]);
    while (j >= 0) {
        System.out.println(j);
        j--;
    }
}
```

Exemplo (2)

- Este tipo de informação não é útil para leigos.
- Podemos utilizar testes convencionais.
  - Solucionamos a quantidade de argumentos.
  - Porém, se não for passado um inteiro, receberemos outra exceção *java.lang.NumberFormatException*.

```
public static void main(String[] args) {
    if (args.length > 0) { // testa presença de argumento
        int j = Integer.parseInt(args[0]);
        while (j >= 0) {
            System.out.println(j);
            j--;
        }
    } else { // Sinaliza erro
        System.out.println("Falta um argumento inteiro");
    }
}
```

Uso do *try catch*

- O uso do **try catch** separa o código e a rotina de tratamento de erros.
  - Com o uso da exceção geral *Exception*, qualquer erro que ocorrer no **try** é tratado no **catch**.

```
public static void main(String[] args) {
    try { // monitora eventuais problemas
        int j = Integer.parseInt(args[0]);
        while (j >= 0) {
            System.out.println(j);
            j--;
        }
    } catch (Exception e) {
        System.out.println("Argumento inválido!");
    }
}
```

Uso do *try catch* - múltiplas cláusulas

- É possível usar várias cláusulas **catch**, uma para cada erro controlado.

```
public static void main(String[] args) {
    try { // monitora eventuais problemas
        int j = Integer.parseInt(args[0]);
        while (j >= 0) {
            System.out.println(j);
            j--;
        }
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Argumento não informado");
    }
    catch (NumberFormatException e) {
        System.out.println("Argumento com tipo inválido!");
    }
}
```

Cláusula *multi-catch*

- É possível capturar múltiplas exceções em uma única cláusula.
  - Útil quando duas ou mais exceções recebem o mesmo tratamento.

```
public static void main(String[] args) {
    try { // monitora eventuais problemas
        int j = Integer.parseInt(args[0]);
        while (j >= 0) {
            System.out.println(j);
            j--;
        }
    } catch (ArrayIndexOutOfBoundsException | NumberFormatException e) {
        System.out.println("Argumento não fornecido ou inválido!");
    }
}
```

Cláusula *finally*

- A cláusula **finally** é sempre executada.
  - Ocorrendo ou não erros no trecho de código delimitado pelo **try**, sempre são executadas as diretivas contidas na cláusula **finally**.
- No exemplo a seguir, a contagem regressiva é executada:
  - Quando o valor informado do argumento está correto, ou utilizando um valor *default*.
- Ao executar o programa, a contagem regressiva é sempre exibida.

Exemplo - Cláusula finally

```
public static void main(String[] args) {
    int j = 5; // valor default
    try { // monitora eventuais problemas
        j = Integer.parseInt(args[0]);
    } catch (ArrayIndexOutOfBoundsException | NumberFormatException e) {
        System.out.println("Argumento inválido ou ausente. Usando default");
    } finally {
        while (j >= 0) {
            System.out.println(j);
            j--;
        }
    }
}
```

Lançamento de Exceções

- Como vimos, exceções são objetos especiais criados para indicar ocorrência de erros.
- A ocorrência de erros deve ser sinalizada por meio das exceções.
  - Possibilita a separação clara entre o código normal e o código de tratamento de erros.
  - Permite desviar o fluxo de processamento para o contexto superior (método que originou a execução), remetendo o erro a pontos mais adequados.

Throwable, Exception e Error

- Throwable
  - Representa a superclasse de todos os elementos sinalizadores de exceções.
  - Apenas objetos dessa classe podem ser lançados pela JVM com o uso da diretiva **throw**.
- Exception (extends Throwable)
  - Dá origem a um conjunto de subclasses que indica condições anormais em uma aplicação.
- Error (extends Throwable)
  - Dá origem a um conjunto restrito de subclasses que indicam condições de erros severas, em geral associadas a JVM ou ao sistema em si.

Tipos de Exceções

- Não monitoradas
  - Não exigem o tratamento com o uso de diretivas **try/catch/finally**, passando implicitamente para o contexto superior.
  - São subclasses de java.lang.RuntimeException.
- Monitoradas
  - Exigem tratamento obrigatório com **try/catch/finally** ou declaração explícita de seu lançamento para contexto superior por meio da cláusula **throws**.
  - É o compilador que verifica e exige o tratamento ou a declaração explícita do lançamento de exceções.

Exemplo - Não Monitoradas

```
public class Retangulo {
    private double largura, altura;

    public Retangulo(double largura, double altura) {
        this.setTamanho(largura, altura);
    }

    public void setTamanho(double largura, double altura) {
        if (largura >= 0 && altura >= 0) {
            this.largura = largura;
            this.altura = altura;
        } else {
            // Lança exceção não monitorada
            throw new RuntimeException("Dimensões inválidas!");
        }
    }
}
```

Exemplo - Monitoradas

```
public class Retangulo {
    private double largura, altura;

    public Retangulo(double largura, double altura) throws Exception {
        this.setTamanho(largura, altura);
    }

    public void setTamanho(double largura, double altura) throws Exception {
        if (largura >= 0 && altura >= 0) {
            this.largura = largura;
            this.altura = altura;
        } else {
            // Lança exceção não monitorada
            throw new Exception("Dimensões inválidas!");
        }
    }
}
```

Testando as Exceções

```
public static void main(String[] args) {
    try {
        double larg = Double.parseDouble(args[0]);
        double alt = Double.parseDouble(args[1]);

        Retangulo ret = new Retangulo(larg, alt);
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Número insuficiente de argumentos.");
    }
    catch (NumberFormatException e) {
        System.out.println("Argumento(s) inválido(s).");
    }
    catch (Exception e) {
        System.out.println("Dimensões informadas são inválidas!");
    }
}
```

Considerações Finais

- Utilização de membros estáticos.
  - Permite criar variáveis de classe e acessar métodos sem instanciar o objeto.
- Constantes utilizam o modificador **final**.
- Funcionamento do Garbage Collector.
  - Uso do método *gc* e implementação do método *finalize*.
- Lançamento e Tratamento de Exceções
  - Utilização das palavras-chave **throw** e **throws**.
  - Utilização de **try/catch/finally**.

Exercício - OOP2

- Para implementar jogos com cartas são necessárias classes que representam uma carta individual e também um baralho. Implemente essas classes, considerando que as Cartas são Ás, 2 a 10, valete, dama e rei, e os naipes são copas, espadas, ouros e paus; e Baralho é um conjunto de 52 cartas (13 cartas para cada naipe), em ordem ou embaralhado.
  - Faça os devidos tratamentos de exceções para as classes Cartas e Baralho.
  - Implemente um método para embaralhar as cartas de um baralho.
- A entrega desta atividade será via Moodle.
  - Enviar um arquivo zip apenas os códigos Java desenvolvidos no exercício.