Introdução à Programação

Leak de Memória e Utilização de Estado para Comunicação entre Objetos

Leak de Memória

Objetos que ele nunca mais vai utilizar

Leak de Memória

Ele não consegue guardar a bola que ele usa toda a semana

Tópicos da Aula

- Hoje vamos aprender como evitar *leaks* de memória
  - Conceito de *Leak* de Memória
  - Causas
  - Revendo o *Garbage Collector* de Java
  - Exemplos em MiniJava para evitar o *leak*
- Depois veremos como podemos utilizar estado para criar uma ligação entre objetos diferentes
  - Exemplos em MiniJava de janelas comunicantes

Conceito de *Leak* de Memória

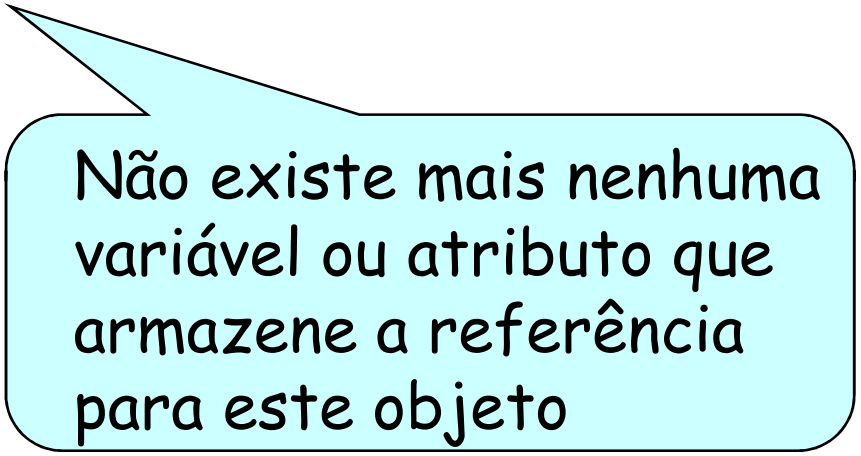
- Dizemos que um ***leak* ou vazamento** de memória ocorreu quando uma porção de memória, alocada para uma determinada operação, não é liberada quando não é mais necessária
- Pode ocorrer um alto consumo de memória levando à degradação do sistema ou mesmo causando uma falha no sistema
- Geralmente, estão relacionadas a erros de programação

Causas

- As causas mais comuns são:
  - Blocos de memória estão alocados e disponíveis para o programa, só que as referências para estes blocos não são guardados, fazendo com que estes blocos fiquem inacessíveis
  - Blocos de memória possuem dados que são inacessíveis, mas ainda são referenciados no programa, mesmo que este não use mais estes dados (veremos mais detalhes em arrays e coleções)
    - Bloco de memória para estes dados não pode ser liberado
- Java Virtual Machine minimiza a ocorrência da primeira causa, através do *Garbage Collection*

Revendo o *Garbage Collection*

- Não existe mecanismo de remoção explícita de objetos da memória em Java
- O Garbage Collector de Java elimina um objeto da memória quando este não é mais referenciado



- O JVM dispara o *garbage collection* de tempos em tempos
  - Indeterminismo na remoção de objetos

O Método `finalize`

- É possível liberar recursos ou fazer alguma ação quando o objeto está na iminência de ser destruído
- O método **`finalize`** deve ser redefinido
  - Cautela no uso deste método, pois este método pode nunca vir a ser chamado

```
class Conta {...
   void finalize() {
      ...
   }
}
```

O último pedido do objeto...

O método finalize é chamado antes do objeto ser removido

Leak de Memória e Java

- O *Garbage Collector* de Java minimiza leak de memória, pois remove objetos inacessíveis
- Contudo, como esta remoção é realizada esporadicamente, pode ainda haver um leak de memória antes que o *garbage collector* atue
- Erros de programação podem deixar objetos que não são utilizados ainda ativos, e assim estes objetos não são removidos automaticamente

Exemplo de *Leak*

```
public class MinhaJanela extends Window  {

Button botao;
public MinhaJanela() {

 botao = new Button();
  ...
}
  ...
}
```

Cria-se um objeto do tipo `Button`, mas este nunca é utilizado

Garbage Collector não remove objeto porque ainda tem um atributo armazenando a referência para este objeto

Exemplo de *Leak*

```
public class MinhaJanela extends Window {

  MinhaJanelaErro janelaErro;
  public MinhaJanela() {...}
  void acaoBotaoCalcular ( ) {
    int entrada = 0;
    try {
        entrada = campoEntrada.getIntExc();
        resultado = 2 * entrada;
        campoResultado.setInt(resultado);
    }catch (InvalidConversionException e) {
        janelaErro = new MinhaJanelaErro();

        janelaErro.setVisible(true);
    }
   ...
  }
```

Atributo que guarda uma janela de erro

Toda vez que há um erro, um novo objeto `MinhaJanelaErro` é criado

Evitando o *Leak*

```
public class MinhaJanela extends Window {
MinhaJanelaErro janelaErro;
public MinhaJanela () {

   janelaErro = new MinhaJanelaErro();
    ...
}
void acaoBotaoCalcular ( ) {
   int entrada = 0;
   try {
        entrada = campoEntrada.getIntExc();
        resultado = 2 * entrada;
        campoResultado.setInt(resultado);
   }catch (InvalidConversionException e) {
        janelaErro.setVisible(true);
   }
  ...
}
```

Cria-se o objeto no construtor

Quando há um erro, o objeto da classe `MinhaJanelaErro` fica visivel

Utilizando Estado para Armazenar Informações Passadas

- Podemos utilizar atributos para armazenar o estado passado do objeto
  - Com este tipo de informação podemos realizar certas ações dependendo do estado anterior

```
public class MinhaJanela extends Window  {
 Button botaoCalcular;
 int entradaAnterior;
 TextField campoEntrada, campoResultado;
  ...
 public void acaoBotaoCalcular() {
  int entrada = 0;
  try {
        entrada = campoEntrada.getIntExc();
        if (entrada == entradaAnterior) {
           campoResultado.setInt(3 * entrada);
        } else {
           campoResultado.setInt(2 * entrada);
         }
  } catch(InvalidConversionException ie) {...}
  entradaAnterior = entrada;
  campoEntrada.setText("");
 }
```

Atributo que guarda o que foi digitado no campoEntrada quando o botaoCalcular é clicado

Dependendo do estado anterior, a ação é diferente

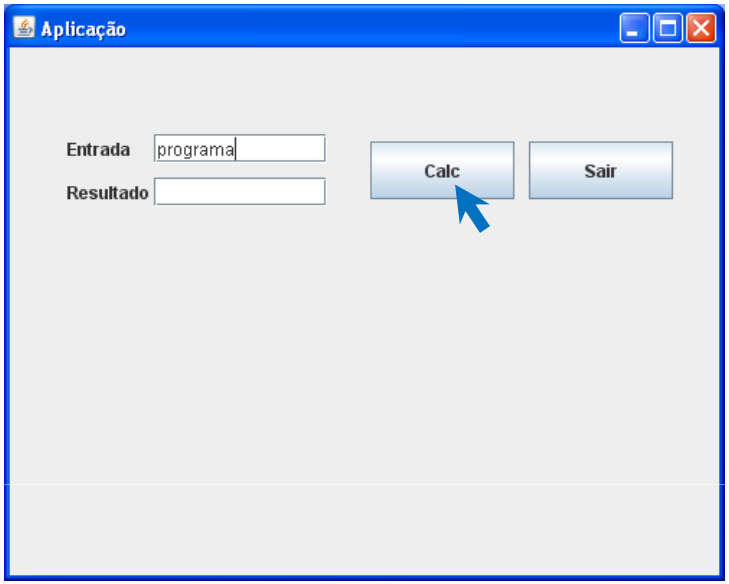
Utilizando Estado para a Comunicação

- Podemos utilizar atributos de um objeto para armazenar outra instância do mesmo (super)tipo de objeto
- Isto permite a comunicação de instâncias diferentes de objetos da mesma (super)classe

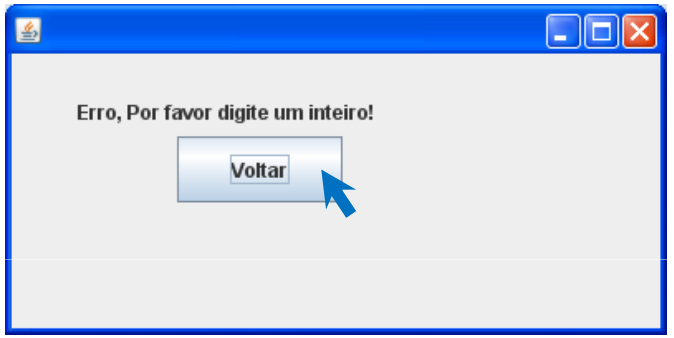
```
public class MinhaJanela extends Window  {

 MinhaJanela outraJanela;
 Button botaoCriarJanela;
 public MinhaJanela() {
   ...
 }
 public void clickEvent(Component componente) {
    if (componente == botaoCriarJanela){
       outraJanela = new MinhaJanela();
       outraJanela.setVisible(true);
    }...
 }...
```

Instâncias diferentes de `MinhaJanela` se comunicam

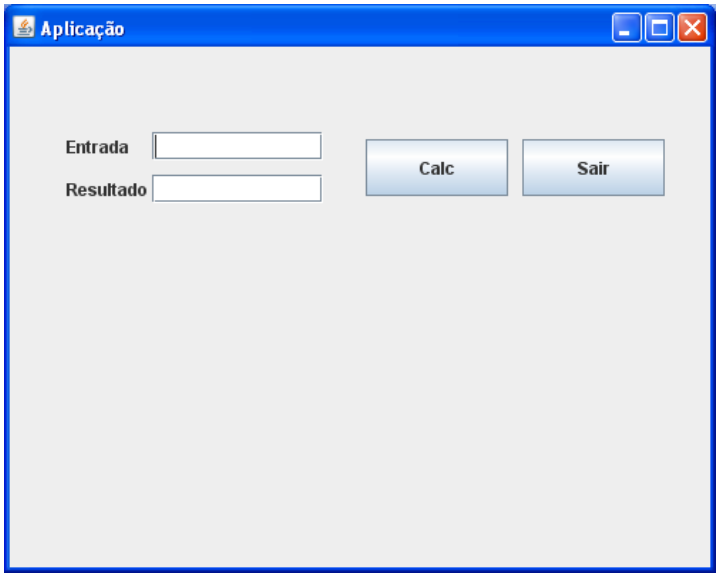
Exemplo de Janelas Comunicantes



Chama a janela de erro e desaparece



Chama a janela principal e desaparece



Janela Principal com Ligação com a Janela de Erro

```
public class MinhaJanela extends Window {
MinhaJanelaErro janelaErro;
public MinhaJanela () {
   janelaErro = new MinhaJanelaErro(this);
    ...
}
void acaoBotaoCalcular ( ) {
  int entrada = 0;
  try {
       entrada = campoEntrada.getIntExc();
       if (entrada == entradaAnterior) {
          campoResultado.setInt(3 * entrada);
       } else {
          campoResultado.setInt(2 * entrada);
        }
  } catch(InvalidConversionException ie) {
       janelaErro.setMensagem("Por favor, digite número");
       this.setVisible(false);
       janelaErro.setVisible(true);
  }...
```

Cria-se o objeto no construtor

Faz a janela de erro aparecer e janela principal desaparecer

Janela de Erro com Ligação com a Janela Principal

```
public class MinhaJanelaErro extends Window {
Label mensagemErro;
Button botaoVoltar;
Window janelaAnterior;
public MinhaJanelaErro (Window janelaOrigem){
mensagemErro = new Label();

botaoVoltar = new Button("Voltar");

janelaAnterior = janelaOrigem;
}
public void clickEvent(Component c) {
   if (c == botaoVoltar) {
     this.setVisible(false);
      janelaAnterior.setVisible(true);
   }
}
public void setMensagem(String s){
   mensagemErro.setText(s);
}
```

Estabelece ligação com uma janela que chama janela de erro

Faz a janela de erro desaparecer e a janela anterior aparecer

Resumindo ...

- *Leaks* de memória
  - Conceito de *Leak* de Memória
  - Causas
  - Java e Leak de Memória
  - Exemplos em MiniJava para evitar o *leak*
- Utilização de estado
  - Para armazenar informações passadas
  - Para criar ligação entre objetos diferentes
  - Exemplos em MiniJava de janelas comunicantes