UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Programação Orientada a Objetos

BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

PROF. LUCIO AGOSTINHO ROCHA

AULA 10:
INTERFACES E CLASSES ABSTRATAS

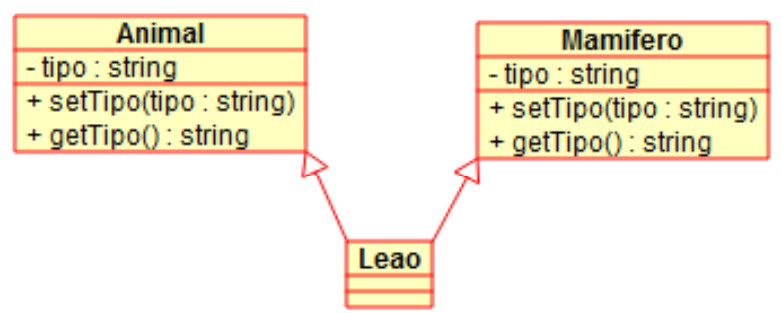
2º.SEMESTRE 2022

Interfaces

Interfaces

- Interfaces:
  - Java não suporta herança múltipla, mas admite múltiplas Interfaces.
  - Classes implementam Interfaces
    - A Interface garante que as classes implementem os métodos.
    - Métodos na interface devem ser declarados 'public abstract'
  - Interfaces permitem que métodos sejam implementados em Interfaces diferentes, e não todos em uma única classe.
  - Ao implementar uma interface a classe explicitamente deve definir qual método será implementado.

Interfaces



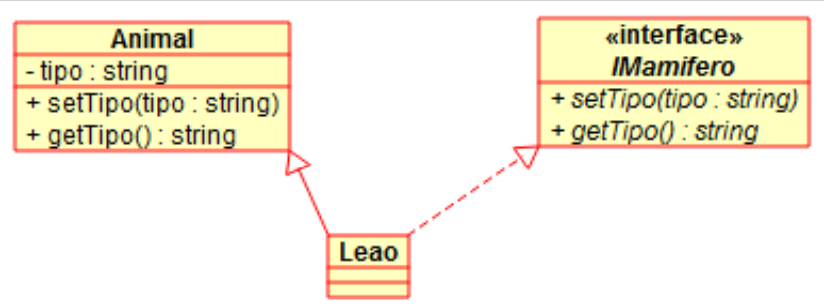
- Figura: Diagrama de Classes com Herança Múltipla.

Interfaces


```
 1 #include <iostream>
 2 #include <string>
 3
 4 #include "Leao.h"
 5
 6 using namespace std;
 7
 8 int main() {
 9
10     Leao leao;
11     cout << leao.Animal::getTipo() << endl;
12     cout << leao.Mamifero::getTipo() << endl;
13
14
15     return 0;
16 }
17
```

- C++ explicitamente informa a superclasse na chamada do método.

Interfaces



- Figura: Diagrama de Classes com Interface.

Interfaces


```
1
2  public interface IMamifero {
3
4      public final String tipo="mamifero";
5
6      public void setTipo(String tipo);
7
8      public String getTipo();
9  }
10
```

- Java: 1) Declaração dos métodos da interface.

Interfaces


```
1
2  public class Leao extends Animal implements IMamifero{
3
4      public Leao(){
5
6      }
7
8      public String toString(){
9          return this.getTipo();
10     }
11
12 }
13
```

- Java: 2) métodos da interface devem ser implementados ou sobrecarregados.

Interfaces


```
1
2  public class Principal {
3
4      public static void main(String[] args) {
5
6          Leao leao = new Leao();
7
8          System.out.println(leao);       //Animal
9          System.out.println(leao.tipo); //Mamifero
10     }
11
12 }
13
```

- Java: 3) Declaração e Instanciação do objeto.

Classes Abstratas

Classes Abstratas

- Classe Abstrata:
  - Permitir que todas as classes herdem umas das outras é um risco de segurança.
  - Classe abstrata: fornece uma superclasse para a qual outras classes podem herdar.
  - Não instanciam objetos.
  - Subclasses devem implementar todos os métodos abstratos. Se não, a subclasse se torna abstrata.
- Classe Concreta:
  - Classes que permitem instanciar objetos.
  - Fornece modelo para instanciar objetos específicos.
    - Ex.: Circulo, Quadrado, Triangulo, Rosa, Margarida, Samambaia.

Classes Abstratas

Estudo de Caso com Classe Abstrata:

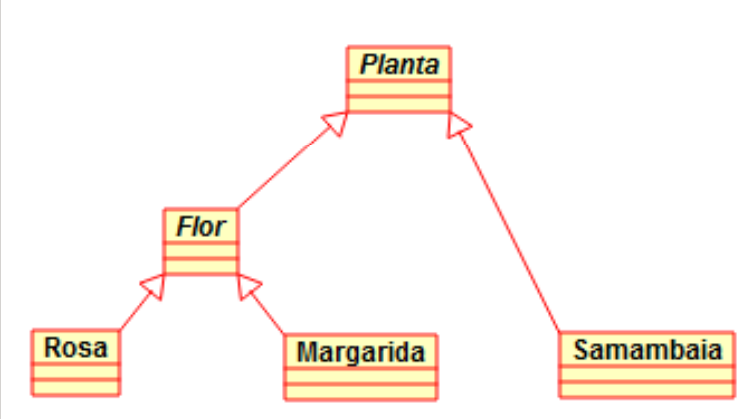


Figura: Classe Planta e Classe Flor são classes Abstract.
Subclasses folha não deveriam ser herdadas (final).

Classes Abstratas

Estudo de Caso com Classe Abstrata:

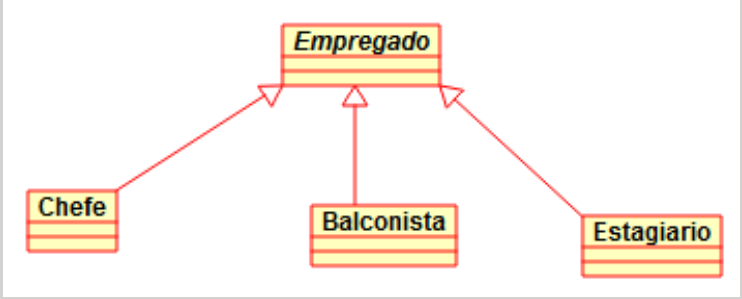


Figura: Classe Empregado é classe Abstract.
Subclasses folha não deveriam ser herdadas (final).

Classes Abstratas

- Modificador de acesso ‘final’:
  - Classes ‘final’ não podem ser herdadas.
  - Métodos ‘final’ não podem ser sobrecarregados.
  - Variáveis de instância ‘final’ são herdadas, mas não podem ser modificadas.

Revisão

Revisão

- Interfaces
- Classes Abstratas

Exercícios

<Ver conteúdo na plataforma de ensino>

Referências

- Referências bibliográficas da disciplina.

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Programação Orientada a Objetos

BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

PROF. LUCIO AGOSTINHO ROCHA

AULA 10:
INTERFACES E CLASSES ABSTRATAS

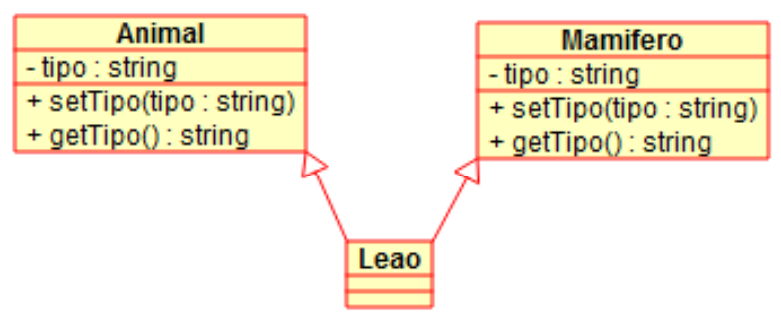
2º.SEMESTRE 2022

Interfaces

Interfaces

- Interfaces:
  - Java não suporta herança múltipla, mas admite múltiplas Interfaces.
  - Classes implementam Interfaces
    - A Interface garante que as classes implementem os métodos.
    - Métodos na interface devem ser declarados 'public abstract'
  - Interfaces permitem que métodos sejam implementados em Interfaces diferentes, e não todos em uma única classe.
  - Ao implementar uma interface a classe explicitamente deve definir qual método será implementado.

Interfaces



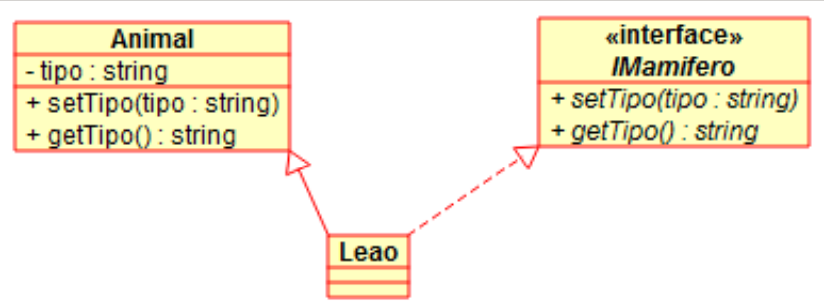
- Figura: Diagrama de Classes com Herança Múltipla.

Interfaces


```
 1 #include <iostream>
 2 #include <string>
 3
 4 #include "Leao.h"
 5
 6 using namespace std;
 7
 8 int main() {
 9
10     Leao leao;
11     cout << leao.Animal::getTipo() << endl;
12     cout << leao.Mamifero::getTipo() << endl;
13
14
15     return 0;
16 }
17
```

- C++ explicitamente informa a superclasse na chamada do método.

Interfaces



- Figura: Diagrama de Classes com Interface.

Interfaces


```
1
2  public interface IMamifero {
3
4      public final String tipo="mamifero";
5
6      public void setTipo(String tipo);
7
8      public String getTipo();
9  }
10
```

- Java: 1) Declaração dos métodos da interface.

Interfaces


```
1
2  public class Leao extends Animal implements IMamifero{
3
4      public Leao(){
5
6      }
7
8      public String toString(){
9          return this.getTipo();
10     }
11
12 }
13
```

- Java: 2) métodos da interface devem ser implementados ou sobrecarregados.

Interfaces


```
1
2  public class Principal {
3
4      public static void main(String[] args) {
5
6          Leao leao = new Leao();
7
8          System.out.println(leao);       //Animal
9          System.out.println(leao.tipo); //Mamifero
10     }
11
12 }
13
```

- Java: 3) Declaração e Instanciação do objeto.

Classes Abstratas

Classes Abstratas

- Classe Abstrata:
  - Permitir que todas as classes herdem umas das outras é um risco de segurança.
  - Classe abstrata: fornece uma superclasse para a qual outras classes podem herdar.
  - Não instanciam objetos.
  - Subclasses devem implementar todos os métodos abstratos. Se não, a subclasse se torna abstrata.
- Classe Concreta:
  - Classes que permitem instanciar objetos.
  - Fornece modelo para instanciar objetos específicos.
    - Ex.: Circulo, Quadrado, Triangulo, Rosa, Margarida, Samambaia.

Classes Abstratas

Estudo de Caso com Classe Abstrata:

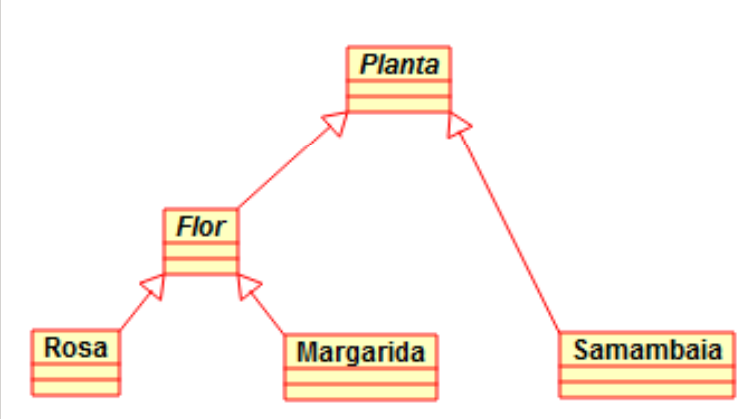


Figura: Classe Planta e Classe Flor são classes Abstract.
Subclasses folha não deveriam ser herdadas (final).

Classes Abstratas

Estudo de Caso com Classe Abstrata:

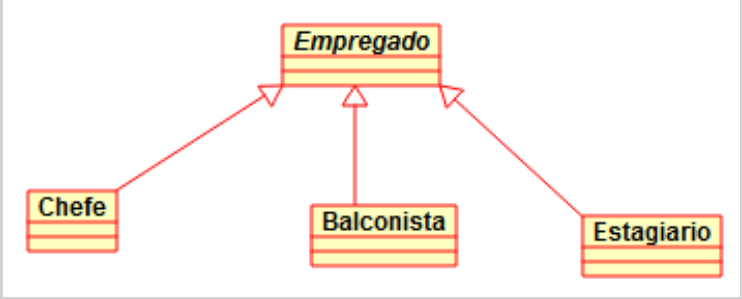


Figura: Classe Empregado é classe Abstract.
Subclasses folha não deveriam ser herdadas (final).

Classes Abstratas

- Modificador de acesso ‘final’:
  - Classes ‘final’ não podem ser herdadas.
  - Métodos ‘final’ não podem ser sobrecarregados.
  - Variáveis de instância ‘final’ são herdadas, mas não podem ser modificadas.

Revisão

Revisão

- Interfaces
- Classes Abstratas

Exercícios

<Ver conteúdo na plataforma de ensino>

Referências

- Referências bibliográficas da disciplina.

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Programação Orientada a Objetos

BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

PROF. LUCIO AGOSTINHO ROCHA

AULA 13: CLASSE INTERNA E CLASSE ANÔNIMA

2º.SEMESTRE 2022

Classe Interna

Classe Interna

- Classe Interna:
  - É uma definição de Classe dentro de uma Classe.
  - Propósito:
    - Agregar várias Classes, sem necessidade de criar novos arquivos.
    - Proteger a visibilidade de uma Classe de outras Classes.
  - Comporta-se como uma parte da Classe.
  - Possui todos os membros da Classe convencional:
    Variáveis de instância e métodos.
  - Classes Internas possuem acesso aos membros privados da Classe Externa.
  - Nota: Classe Interna comum não pode definir membros 'static'.
    Para isso, a Classe Interna deve ser 'static'.

Classe Interna

- Classe Interna:
  - Deve ser instanciada com uma referência ao objeto da ClasseExterna.

```
public  class ClasseExterna {  //1)
  private int var;

  public void executar(){
    ClasseInterna classeInterna = new ClasseInterna();
  }
  private class ClasseInterna {
        //TEM acesso aos membros da ClasseExterna
        var = 111;
  }
}//fim classe Externa
  ...
  ClasseExterna classeExterna = new ClasseExterna();
  classeExterna.executar();
```


Classe Interna

- Classe Interna:
  - Deve ser instanciada com uma referência ao objeto da ClasseExterna.

```
public  class ClasseExterna {  //2)
  private int var;

  ...
  private class ClasseInterna {
        //TEM acesso aos membros da ClasseExterna
        var = 111;
  }
  ...
  ClasseExterna.ClasseInterna classeInterna =
              new ClasseExterna().new ClasseInterna();

}
```

Classe Interna

- Classe Interna ‘static’:
  - Motivação: Classe Interna com uso exclusivo dentro da Classe.
  - Comporta-se como uma Classe isolada dentro da Classe Externa.
  - É uma Classe interna, mas sem uma referência para a Classe Externa.
  - Não possui acesso imediato aos membros da Classe Externa.
  - Outras Classes não têm acesso à Classe Interna static (deveria ser declarada ‘private’).

```
public  class ClasseExterna{
  private int var;
  private static class ClasseInterna {
        //NÃO TEM acesso aos
         //  membros da ClasseExterna
        //var = 111;
  }
  ...
  ClasseExterna.ClasseInterna classeInterna =
              new ClasseExterna().new ClasseInterna();
}
```


Resumo

- Classe Interna:
  - Gera um arquivo .class separado
  - Modificadores de acesso permitidos:
    - public, protected, private, ou acesso de package
  - Classe externa é responsável por criar objetos da Classe Interna
  - Classe interna também pode ser ‘static’

Classe Anônima

Classe Anônima

- Classe Anônima:
  - Uma classe anônima é uma subclasse sem nome de uma superclasse OU
  - É uma classe que implementa uma interface.
  - Em ambos os casos, É uma Classe que não tem um nome.
    - Não há a palavra reservada ‘class’, mas não é só isso: a classe é criada, mas não se tem a referência para criar uma instância da classe anônima.
  - Não tem Construtor:
    - Utiliza o construtor da superclasse
  - Classe interna anônima pode acessar os membros da sua Classe de primeiro nível.

```
public class GUI3 {
  private int var;

  botao.addActionListener(
        new ActionListener(){ //chamada da
           //classe interna anônima para
           //implementar a interface ActionListener

        //Classe interna anonima TEM acesso aos
        //membros da classe superior de primeiro nivel
        public void actionPerformed(ActionEvent e){
              var=1;
             System.out.println(var);
         }
   });//fim classe interna

}//fim classe
```


Revisão

Revisão

- Classe Interna
- Classe Anônima

Exercícios

<Ver conteúdo na plataforma de ensino>

Referências

- Referências bibliográficas da disciplina.

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Programação Orientada a Objetos

BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

PROF. LUCIO AGOSTINHO ROCHA

AULA 15: POLIMORFISMO

2º.SEMESTRE 2022

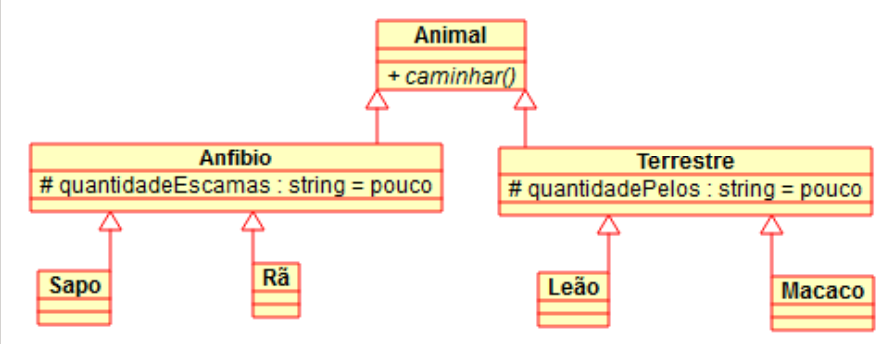
Polimorfismo

Polimorfismo

- Polimorfismo é um recurso de programação onde objetos que compartilham a mesma superclasse são tratados como objetos dessa superclasse.
- Polimorfismo: é um recurso que permite programar “no geral”, ao invés de programar “no específico”.
- Exemplo:
  - Suponha um programa que implemente a ação de caminhar dos seguintes animais: sapo, rã, leão e macaco.
  - Todos os animais caminham.
  - Todos os animais herdam da Classe Animal

Polimorfismo

- Polimorfismo:



- O que estas subclasses têm em comum?

Sapo	Rã	Leão	Macaco

Polimorfismo

- Cada classe derivada herda o método ‘caminhar()’ da superclasse Animal.
- Na Classe Principal é mantida uma lista de objetos das subclasses.
- O mesmo método “genérico” é enviado para cada objeto da classe derivada.
- Esse método é sobrecarregado por cada objeto, que o define da sua própria maneira.

Polimorfismo

- Polimorfismo:
  - Auxilia a construir programas extensíveis.
  - Programas processam genericamente objetos como superclasses
    - Novas classes podem ser facilmente inseridas no sistema
      - Novas classes devem fazer parte da hierarquia.
  - Exemplo:
    - Classe Coelho como subclasse da Classe Terrestre.
    - Classe Coelho herda o método genérico caminhar() e o implementa da sua própria maneira.

Polimorfismo

- Sem Polimorfismo:

```
public class Principal {
...
public static void main(String [ ] args){
   Sapo sapo = new Sapo();
   Ra ra = new Ra();
   Leao leao = new Leao();
   Macaco macaco = new Macaco();

   sapo.caminhar();
   ra.caminhar();
   leao.caminhar();
   macaco.caminhar();
}
```

Polimorfismo

- Com Polimorfismo:

```
public class Principal {
...
public static void main(String [ ] args){
   ArrayList<Animal> lista = new ArrayList<>();
   lista.add( new Sapo() );
   lista.add( new Ra() );
   lista.add( new Leao() );
   lista.add( new Macaco() );

   for( Animal  animal : lista )
       animal.caminhar();
}
```


Polimorfismo

- Polimorfismo:
  - Um objeto da classe base (superclasse) sempre pode receber um objeto da classe derivada (subclasse).
  - Ex.: Sapo é um Animal.

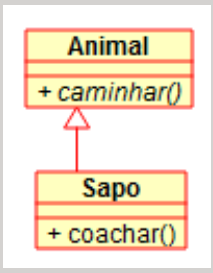
```
Animal  animal = new Sapo();
animal.caminhar();              //Método da classe derivada
```

    - O inverso não é permitido.

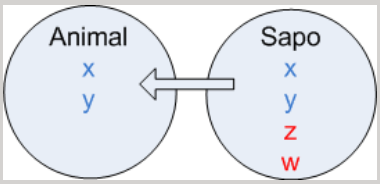
  - O objeto 'animal' continua sendo da Classe Animal, porém, adquiriu mais membros da classe derivada. Logo, caso se queira acessar membros da subclasse específica, é necessário explicitar a subclasse:

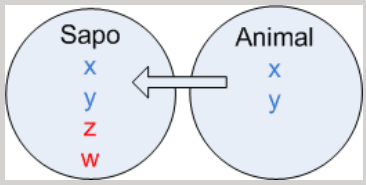
```
((Sapo) animal).coachar();    //Método específico da subclasse
```

Polimorfismo



- Sapo é um Animal





- Faltam campos

Polimorfismo

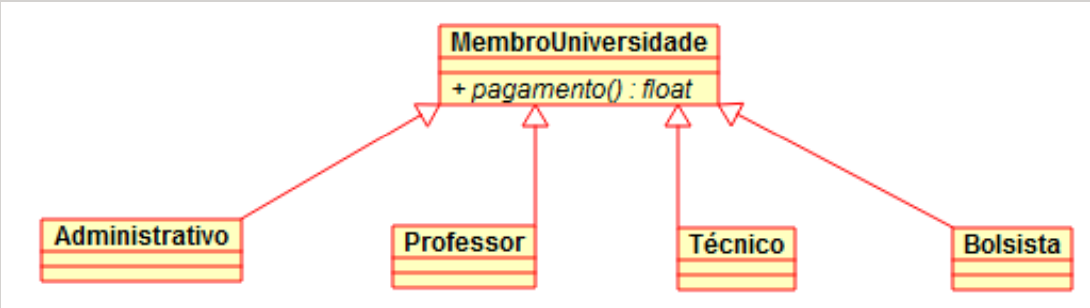

```
public class Principal {

public static void main(String [] args){
   Animal animal = new Sapo();
  animal.caminhar();                    //metodo generico

  animal = new Leao();                  //mudou de forma
  animal.caminhar();                    //metodo generico
}

}
```

Polimorfismo

- Outro exemplo de Polimorfismo:
  - Um programa de planilha de pagamentos dos membros de uma universidade.
  - Os membros da universidade são:
    - Administrativo, Professor, Técnicos, Bolsistas.
  - Todos os membros herdam da Classe MembroUniversidade.
  - Todos os membros recebem um pagamento.

Polimorfismo

- Polimorfismo:



- O que estas subclasses têm em comum?

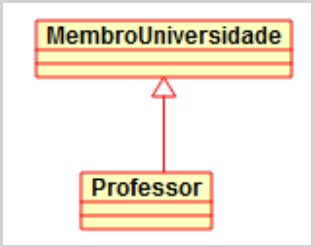
Administrativo	Professor	Técnico	Bolsista

Polimorfismo

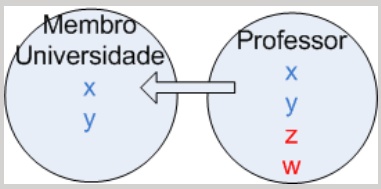
- Com Polimorfismo:

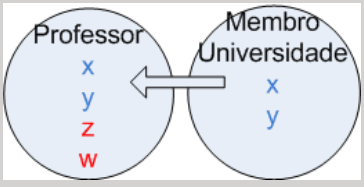
```
public class Principal {
...
public static void main(String [ ] args){
   ArrayList<MembroUniversidade> lista = new ArrayList<>();
   lista.add( new Administrativo() );
   lista.add( new Professor() );
   lista.add( new Tecnico() );
   lista.add( new Bolsista() );

   for( MembroUniversidade  membro : lista )
       membro.pagamento();
}
```


Polimorfismo



- Professor é um MembroUniversidade





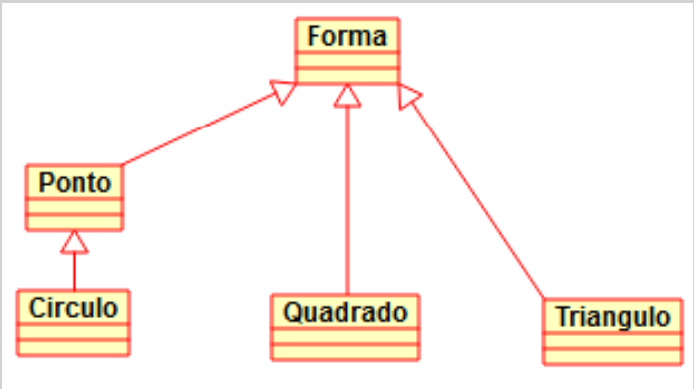
- Faltam campos

Polimorfismo


```
public class Principal {

public static void main(String [] args){
   MembroUniversidade membro = new Professor();
   membro.pagamento();                    //metodo generico

  membro = new Bolsista();               //mudou de forma
  membro.pagamento();                    //metodo generico
}

}
```


Polimorfismo

- Outro exemplo de Polimorfismo:

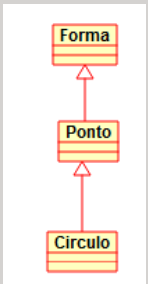


Polimorfismo

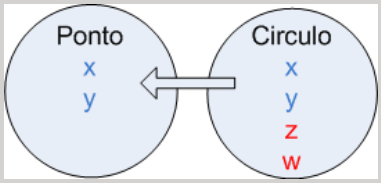
- Com Polimorfismo:

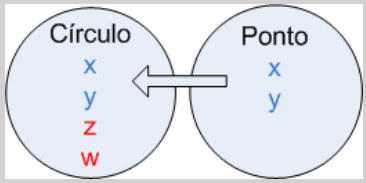
```
public class Principal {
...
public static void main(String [ ] args){
   ArrayList<Forma> lista = new ArrayList<>();
   lista.add( new Circulo() );
   lista.add( new Triangulo() );
   lista.add( new Quadrado() );
   lista.add( new Ponto() );

   for( Forma forma : lista )
       forma.imprimir();
}
```


Polimorfismo



- Círculo é um Ponto.
- Círculo é uma Forma.





- Faltam campos

Polimorfismo


```
public class Principal {

public static void main(String [] args){
   Forma forma = new Circulo();
   forma.imprimir();                    //metodo generico

  forma = new Triangulo();          //mudou de forma
  forma.imprimir();                    //metodo generico
}

}
```


Revisão

Revisão

- Polimorfismo
- Interface
- Classe Abstrata

Revisão

- Ligação dinâmica:
  - Implementa processamento polimórfico de objetos.
  - Utiliza a referência da Superclasse para o objeto da Subclasse.
  - Os métodos são os mesmos, mas cada objeto de uma Subclasse implementa o seu (sobrecarga).

Revisão

- Alternativas ao polimorfismo: switch
  - Tratar cada novo objeto em uma estrutura switch
  - Problemas:
    - Deixar a critério exclusivo do programador a validação dos Membros da Classe
    - Adicionar e remover novas classes.

Exercícios

<Ver conteúdo na plataforma de ensino>

Referências

- Referências bibliográficas da disciplina.

Programação Orientada a Objetos

BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

PROF. LUCIO AGOSTINHO ROCHA

TRATAMENTO DE EXCEÇÕES

2º.SEMESTRE 2022

Tratamento de Exceções

Tratamento de Exceções

- Exceção:
  - É a ocorrência de um erro infrequente durante a execução do programa:
    - Ex.: conversão de tipos, erros aritméticos (divisão por zero), tentativa de acesso fora dos limites de um vetor, acesso a classes inexistentes, e outras exceções.
  - Tratamento de exceções:
    - Evita que o programa seja interrompido abruptamente em um estado inválido.
    - O programa trata a exceção e continua a executar como se nada tivesse acontecido, ou encerra o programa.

Tratamento de Exceções

- Quando utilizar o Tratamento de Exceções:
  - Em regiões do código onde a exceção poderá existir com maior frequência;
  - Em muitos casos, a própria linguagem Java exige a inserção do tratamento de exceção (Ex.: arquivos).
  - Padronização

Tratamento de Exceções

- O método que encontra um erro durante a execução dispara (throw) uma exceção.
  - O tratador da exceção processa especificamente esse erro.
- try:
  - Bloco que inclui o trecho de código que potencialmente gerará um erro.
- catch:
  - Bloco que captura o erro e faz o tratamento da exceção.
- finally:
  - Bloco que é sempre executado, com ou sem o erro.

Tratamento de Exceções

- Estrutura do bloco try-catch:

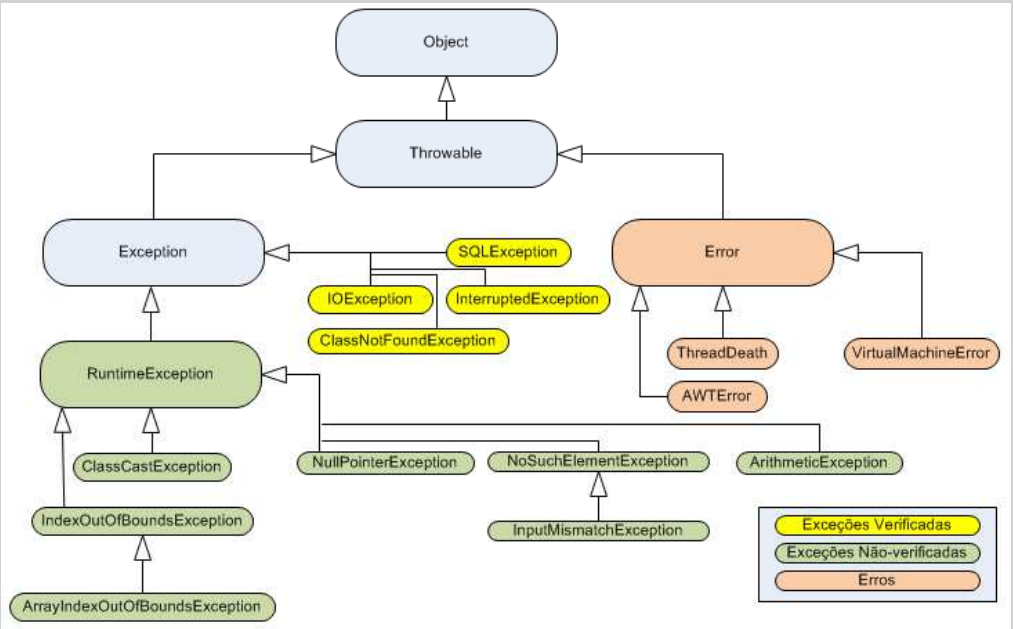
```
try {

} catch ( TipoExceção1 exceção1 ) {

} catch ( TipoExceção2 exceção2 ) {

} finally {

}
```

Tratamento de Exceções

- throws:
  - É uma declaração utilizada na declaração do método para indicar que a execução do método poderá disparar uma exceção.
  - Atua em qualquer classe derivada de 'Throwable'
    - Throwable possui as subclasses:
      - Exception: erros que podem ser capturados e tratados.
      - Error: erros que não deveriam ser capturados.
- Programa termina se há um erro durante a execução e não houver um tratamento da exceção.
- throw:
  - Indica que o método dispara uma exceção.
- Um único bloco 'catch' pode capturar múltiplas exceções.
- Apenas um catch é utilizado caso uma exceção seja capturada.

Tratamento de Exceções

- Hierarquia da Classe Throwable (algumas das principais Classes):



Tratamento de Exceções

- Exceções Verificadas:
  - Todas as subclasses derivadas diretamente da Classe Exception são verificadas.
  - Tratáveis na compilação.
  - É exigido o tratamento na escrita do código pelo compilador.
  - Exceção deve ser capturada (catch) ou disparada (throw)
  - Exemplos: exceções personalizadas, abertura de arquivos, threads, acesso a base de dados, classes não encontradas, sockets, etc.
- Exceções Não-verificadas:
  - Subclasses derivadas da Classe RuntimeException.
  - Tratáveis em tempo de execução.
  - Não é exigido o tratamento na escrita do código pelo compilador.

Tratamento de Exceções

- Exceções personalizadas:
  - Estendem a Classe Exception ou uma Classe derivada dela.
  - Tornam o detalhamento do erro mais consistente para o usuário.
- 4 (quatro) construtores:
  - Construtor sem argumentos: mensagem de erro padrão.
  - Construtor com uma String: mensagem enviada para a superclasse.
  - Construtor com uma String e uma Throwable (para encadear exceções)
  - Construtor com uma Throwable: enviada para a superclasse.

Revisão

Revisão

- Tratamento de Exceções

Exercícios

<Ver conteúdo na plataforma de ensino>

Referências

- Referências bibliográficas da disciplina.