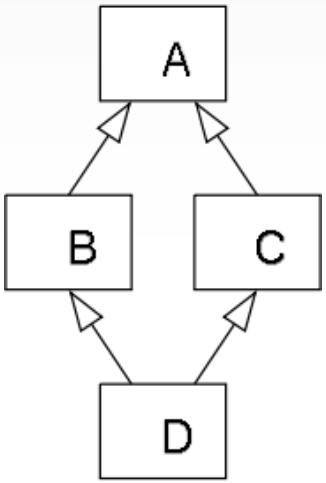
Java
Orientação a Objetos

Wagner G. Al-Alam
wgalam@gmail.com

Herança

- Herança
  - Forma de reutilização de software:
  - Classes são criadas a partir de classes já existentes;
  - Absorve-se atributos e comportamentos da classe pai;
  - Pode-se adicionar ou redefinir comportamentos;
  - Java não permite herança múltipla:
    - Não apresentando o “Diamond problem”



Herança
Definições

- Superclasse:
  - Classe que tem seus atributos e métodos herdados por outra classe.
  - Pode ser:
    - Direta:

      Herda diretamente através da palavra chave **extends**.
    - Indireta

      É herdada de dois ou mais níveis acima na hierarquia de classe.
- Subclasse:
  - Classe que herda atributos e métodos de uma Superclasse.
  - É candidata a ser superclasse de outra subclasse.

Herança
Tipos

- Herança simples:
  - É uma classe que deriva de uma superclasse.
- Herança múltipla:
  - Java não suporta herança múltipla.
  - Suporta interfaces:
    - Ajudam a alcançar muitas vantagens da herança múltipla sem os problemas associados.

Visibilidade

- Private
  - Não é acessível pelas subclasses.
- Public
  - É acessível por subclasses e demais classes do software.
- Protected
  - É acessível pelas subclasses.

Referência This

- Quando um método de uma classe referencia outro membro dessa classe para um objeto específico dessa classe, como Java assegura que o objeto adequado é referenciado?
- É que cada objeto tem acesso a uma referência a ele próprio chamada de referência this.
- A referência **this** é implicitamente utilizada para referenciar variáveis de instâncias e métodos de um objeto.
- Outra utilização da referência **this** está em permitir chamadas de métodos em cascata.
- Usar uma referência **this** em um método **static** é um erro de sintaxe.

Getters e Setters

- Utilizado para atributos private;
- Permite o acesso aos atributos de uma maneira controlada.
- Comumente criando dois métodos, um que retorna o valor e outro que altera o valor;
- O padrão para esses métodos é de colocar a palavra get ou set antes do nome do atributo.
- Ex:
  - private String name;
  - public String getName(){
    return name;
    }
  - public void setName(String name){
    this.name = name;
    }

Construtores e Finalizadores

- Quando um objeto de uma subclasse é instanciado, o construtor da superclasse deve ser chamado para fazer qualquer inicialização necessária das variáveis de instância da superclasse do objeto de subclasse.
- Uma chamada explicita ao construtor da superclasse (através da referência **super** pode ser fornecida como primeira instrução no construtor de superclasse.
- Caso contrário, o construtor de subclasse chamará o construtor *default* da superclasse implicitamente.
- Se as classes em sua hieraquia de classes definem métodos **finalize**, o método **finalize** da subclasse deveria invocar o método **finalize** da superclasse para assegurar que todas as partes de um objeto são finalizados adequadamente se o coletor de lixo reinvidicar a memória para o objeto.

Operador de Teste de Herança

- Operador instanceof:
  - Permite determinar se o objeto é de determinada classe ou extensão da mesma.
  - Representa o teste “é um”.

Herança
Exemplo 1 - Escola

- public class FiguraGeometrica{
- }
- public class circulo extends FiguraGeometrica{
- }
- public class Quadrado extends FiguraGeometrica{
- }

Exemplo 1
Pessoa.java

```
public class Pessoa {
   private int idade;
   protected String nome;
public Pessoa(int idade, String nome){
   this.nome = nome;
   this.idade = idade;
}
public int getIdade(){
   return idade;
   }
}
```

Exemplo 1
Aluno.java

```
public class Aluno extends Pessoa{
   protected String matricula;
public Aluno(int idade, String nome, String matricula) {
   super(idade, nome);
   this.matricula = matricula;
   }
}
```

Exemplo 1
AlunoPos.java

```
public class AlunoPos extends Aluno{
   String departamento;
   String orientador;
public AlunoPos(int idade, String nome, String matricula, String
   departamento, String orientador) {
   super(idade, nome, matricula);
   this.departamento = departamento;
   this.orientador = orientador;
}
public String toString(){
   return "Aluno: "+super.nome+", idade: "+super.getIdade()+", matricula:
   "+super.matricula+ ", departamento: " +departamento + ", orientador: "
   + orientador;
   }
}
```

Exemplo 1
AlunoGraduacao.java

```
public class AlunoGraduacao extends Aluno{
   String curso;
public AlunoGraduacao(int idade, String nome, String matricula, String
   curso) {
   super(idade, nome, matricula);
   this.curso = curso;
}
public String toString(){
   return "Aluno: "+super.nome+", idade: "+super.getIdade()+", matricula:
   "+super.matricula+", curso: " +curso;
}
}
```

Exemplo 1
Main.java

```
public class Main {


public static void main(String[] args) {
   AlunoGraduacao aluno1 = new AlunoGraduacao(20, "Joao", "123456",
   "Computação");
   AlunoPos aluno2 = new AlunoPos(30, "Jose", "654123", "Computação",
   "Nome do orientador");
   if((Object)aluno1 instanceof Aluno){
            System.out.println("aluno1 é um Aluno");
   }else{
            System.out.println("aluno1 não é um Aluno");
            }
```

Exemplo 1
Main.java (continuação)

```
    if((Object)aluno1 instanceof AlunoGraduacao){
            System.out.println("aluno1 é um AlunoGraduação");
    }else{
            System.out.println("aluno1 não é um AlunoGraduação");
            }
    if((Object)aluno1 instanceof AlunoPos){
            System.out.println("aluno1 é um AlunoPos");
    }else{
            System.out.println("aluno1 não é um AlunoPos");
            }
    }
}
```

Sobreposição de Métodos

- Quando um método da subclasse possui um nome igual a um método da superclasse, diz-se que o método da subclasse sobrepõe (override) o método da superclasse.
- O polimorfismo é *essencial* em POO por uma razão:
  - Permite que uma classe geral especifique métodos que serão comuns a todos os seus descendentes, permitindo que as subclasses definam uma implementação específicas de alguns ou todos os métodos da superclasse.

Campos de Tipo e Instruções Switch

- Pode se utilizar uma instrução switch para se testar a que classe o objeto se refere e chamar o método adequado, por exemplo para calcular a área de uma figura geométrica.
- Podemos ter uma superclasse FiguraGeometrica e diversas subclasses (Circulo, Quadrado, Triangulo) e em cada subclasse uma implementação do método area().
- Através do switch encontraríamos o método adequado a ser chamado.
- Porém caso o programador esqueça de implementar um case teríamos problemas, devido a isto temos o polimorfismo que se apresenta como uma maneira mais eficiente para resolver este problema.

Polimorfismo

- Permite projetar e implementar sistemas que são mais facilmente extensíveis.
- Os programas podem ser escritos para processar genericamente, como objetos de superclasse, objetos de todas as classes existentes em uma hierarquia.
- As classes que nao existem durante o desenvolvimento do programa podem ser adicionadas com pouca ou nenhuma modificação da parte genérica do programa.
- Só se faz necessária alteração no código naquelas partes que exigem conhecimento direto da classe particular que é adicionada à hierarquia.

Polimorfismo
Exemplo (Adaptação em Main.java)

```
AlunoGraduacao aluno1 = new AlunoGraduacao(20, "Joao", "123456",
  "Computação");
AlunoPos aluno2 = new AlunoPos(30, "Jose", "654123", "Computação", "Nome
  do orientador");
Pessoa pessoa;
pessoa = aluno1;
System.out.println("Imprimindo pessoa 1 vez:");
System.out.println(pessoa.toString());
pessoa = aluno2;
System.out.println("Imprimindo pessoa 2 vez:");
System.out.println(pessoa.toString());
```

```
Imprimindo pessoa 1 vez:
Aluno: Joao, idade: 20, matricula: 123456, curso: Computação
Imprimindo pessoa 2 vez:
Aluno: Jose, idade: 30, matricula: 654123, departamento: Computação, orientador: Nome do orientador
```

Vinculação Dinâmica de Método

- Por Exemplo,
  - Superclasse Shape;
  - Subclasses Circle, Rectangle e Square;
  - Cada classe desenha ela mesma de acordo com o tipo de classe:
    - Shape tem o método draw;
    - Cada classe sobreescre o métododraw;
    - Chamar o métododraw da superclasse Shape.
      - O programa determina dinamicamente de qual subclasse o método draw será invocado.

Métodos e Classes final

- Variáveis declaradas como **final** não poderão ser modificadas depois que são declaradas e devem ser inicializadas quando declaradas.
- Métodos declarados como **final** não podem ser sobrescritos em uma subclasse.
- Métodos declarados como **static** ou **private** são implicitamente **final**.
  - Permite otimização do código pelo compilador, através de uma técnica chamada de inclusão de código inline, ou seja, as chamadas aos métodos final são substituídas pelo código expandido.
- Uma classe declarada **final** não pode ser superclasse.
  - Todos métodos de uma classe **final** são implicitamente **final**.

Superclasses Abstratas e Classes Concretas

- Classes Abstratas
  - São as classes as quais o programador não pode instanciar nenhum objeto;
  - Essas classes são utilizadas como superclasses em situações de herança (***superclasses abstratas***);
  - **Propósito**: é fornecer uma superclasse apropriada da qual as outras classes possam herdar ***interfaces*** e/ou ***implementação***.
- Classes Concretas
  - São as classes da qual os objetos podem ser instanciados.
- Exemplo
  - Classe nativa AbstractList não pode ser instanciada.
  - Mas a classe concreta arrayList ou Vector, podem ser instanciadas.

Interfaces

- É um tipo de dado especial que especifica o que uma classe deve fazer, mas não especifica como fazer.
- As interfaces são muito semelhantes às classes, contudo, são desprovidas de variáveis de instância (se existir devem ser estáticas e finais) e, seus métodos são declarados sem implementação.
- Uma vez definida uma interface, qualquer classe pode implementá-la e uma classe pode implementar quantas interfaces quiser, permitindo resultados semelhantes aos obtidos com a **herança múltipla**.
- As interfaces são utilizadas para que classes não relacionadas possam implementar métodos com a mesma interface.

Interfaces
Sintaxe

acesso **interface** NomeDaInterface**{**

//declaração de atributos final static

ooo

//declaração de métodos (sem implementação)

ooo

}

- **É possível criarmos uma interface estendendo outra, isto é, através da herança simples de outra interface, mas não de uma classe simples e vice-versa, sendo a sintaxe:**

acesso **interface** NomeDaInterface **extends** InterfaceBase**{**

//declaração de atributos final static

ooo

//declaração de métodos (sem implementação)

ooo

}

Interface
Exemplo

```
//definição da interface Forma
public interface Forma{
public double area();
   public double volume();
   public String getNome();
}  //definição da classe Ponto
public class Ponto implements Forma{
   //corpo da classe
   public double area(){
            //corpo do metodo
   }public double volume(){
            //corpo do metodo
   }public String getNome(){
            //corpo do metodo
   }
}
```

Classes Internas

```
class ClasseExterna{
        private String nome = "variável private da classe externa.";
        class ClasseInterna{
                public void acesso(){
                        System.out.println("Acesso pegou a " + nome);
                }
        }
  }

public class Teste{

public static void main(String [] args){
        ClasseExterna.ClasseInterna ci = new ClasseExterna().new
        ClasseInterna();
        ci.acesso();
        }
  }
```

Classe Interna
Declaradas dentro de métodos

```
public class ClasseExterna {
  public void fazInterna(){
        final int X = 45;
        class ClasseInterna {
              public void acessaClasseExterna(){
                    System.out.println("X = " + X);
              }

        }
  ClasseInterna ci = new ClasseInterna();
  ci.acessaClasseExterna ();
  }

public static void main(String [] args){
  ClasseExterna ce = new ClasseExterna();
  ce.fazInterna();
  }

}
```

Classe Interna
Classe Interna Anônima

```
public class Teste{
    public void delta(){
                System.out.println("Método Teste.delta();");
    }
public void alfa(){
    System.out.println("Método Teste.alfa();");
    }
public static void main(String args[]){
    Teste t = new Teste();
    t.delta();
    ExemploClasseAnonima eci = new ExemploClasseAnonima();
    eci.teste.delta();
    eci.teste.alfa();
    }
}
class ExemploClasseAnonima{
    Teste teste = new Teste() { //aqui temos uma sintaxe estranha
    public void delta(){
                System.out.println("Método delta() sobrescrito.");
                }
    };
}
```

Referências

- [1] Java Como Programar. Deitel 4ª edição.
- [2] Programação Java. André Rauber Du Bois. Universidade Católica de Pelotas.
- [3] http://pt.wikipedia.org/wiki/Java_(linguagem_de_programa%C3%A7%C3%A3o)
- [4] http://www.scribd.com/doc/28029957/Programacao-Orientada-a-Objeto-Com-Java#
- [5]http://imasters.com.br/artigo/1921/java/classes_internas_e_classes_internas_anonimas/