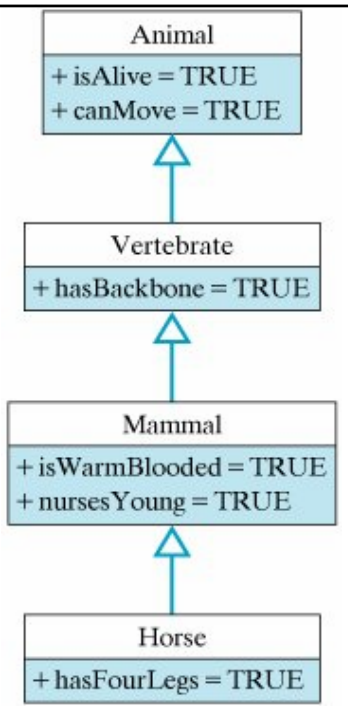
Programação Orientada a Objetos

Alexandre César Muniz de Oliveira

Herança e polimorfismo

Parte VII

Herança



Herança

- Subclasse herda
  - Métodos públicos e protegidos
  - Variáveis de instâncias
- Exemplo:
  - Classe *Object*
  - Método toString()

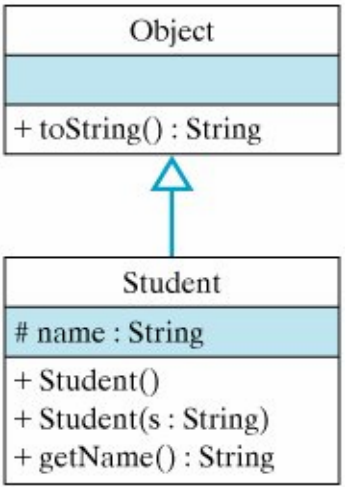
Herança

```
public class Student {
    protected String name;
    public Student(String s) {
        name = s;
    }
    public String getName() {
        return name;
    }
}
```

Herança

Student stu = new Student("Stu");
System.out.println(stu.toString());

1. Procura na própria classe
2. Não encontrando sobe a hierarquia de classes
3. até achar uma definição *public* ou *protected* definition
   - **SAÍDA: Student@cde100**



Sobreposição de métodos

- Sobreposição de métodos (*Overriding*)
  - Redefenir um método permite configurar um dado método herdado às necessidades de determinada classe.

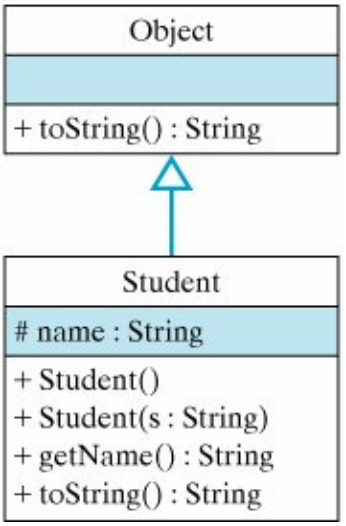
Sobreposição de métodos

- Sobrescrever *toString()* para a classe Student :

```
public String toString() {
  return "My name is " + name + " and I am a
    Student.";
}
```

- Ambos *Object* e *Student* contêm implementações de *toString()*.
- >>> stu.toString() :
  - ***My name is Stu and I am a Student.***

Sobreposição de métodos



Sobreposição de métodos

- Herança
  - Métodos e variáveis *public* e *protected* podem são herdados por suas subclasses.
- Sobreposição:
  - Sobrescreve um método herdado, adequando-o à subclasse

Polimorfismo

- Ligação dinâmica
  - Polimorfismo permite múltiplas formas de comportamento
  - Implementação correta de um método é definida em tempo de execução
  - Diferente da ligação estática que é resolvida em tempo de compilação
  - Todos os métodos são ligados dinamicamente, exceto:
    - Métodos privados
    - Métodos *finais*

Polimorfismo

```
Object obj;
obj = new Student("Stu");
System.out.println(obj.toString());
obj = new OneRowNim(11);
System.out.println(obj.toString());
```

Polimorfismo

- Método polimórfico

```
public void polyMethod(Object obj) {
  System.out.println(obj.toString()); // Polymorphic
}
...
        Student stu = new Student("Stu");
        polyMethod(stu);
        OneRowNim nim = new OneRowNim();
        polyMethod(nim);
...
```

Polimorfismo

- Métodos sobrecarregados

```
print(char c);        println(char c);
print(int i);         println(int i);
print(double d);      println(double d);
print(float f);       println(float f);
print(String s);      println(String s);
print(Object o);      println(Object o);
```

Polimorfismo

- Flexibilidade e extensibilidade:
  - The print() println() podem imprimir objetos que nem existiam quando a biblioteca foi escrita

```
public void print(Object o) {
    System.out.print(o.toString());
}

public void println(Object o) {
    System.out.println(o.toString());
}
```

Polimorfismo

- Super
  - Chama métodos da superclasse
  - Exemplo:
    super.toString() + toString()

Polimorfismo

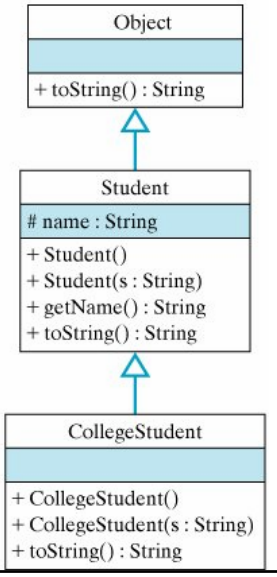
```
public class Abelha {
    public void voa() { System.out.println(“Vôo da Abêia"); }
}
public class Besouro extends Abelha {
    public void voa() { System.out.println(“Vôo do Bisôro"); }
} // …
Abelha a = new Abelha ();
a.voa();
a = new Besouro();
a.voa();
Besouro b = new Besouro();
b.voa();
```

Polimorfismo

```
public class Besouro extends Abelha {
    super();
     public void voa() { System.out.println(“Vôo do
    Bisôro"); }
} // …
Abelha a = new Besouro();
a = new Abelha();
Besouro b = new Abelha();
b = new Besouro();
```

Polimorfismo

- Construtores



Polimorfismo

- *Construtores não são herdados*

```
public class CollegeStudent extends Student {
    public CollegeStudent() { }
    public CollegeStudent(String s) {
        super(s);
    }
    public String toString() {
        return "Meu nome é " + name +
                " e eu sou um Colegial";
    }
}
```

Polimorfismo

- Encadeamento de construtores
  - Construtor default da super classe é automaticamente invocado, caso nenhum outro o seja explicitamente
  - Os construtores são invocados subindo através de toda a hierarquia da classe.
  - Exemplo:
    - Student() -> Object()
  - Falta de construtor default (implícito ou explícito) causa erro

Polimorfismo

- Pode-se chamar explicitamente um construtor de super classe através de super()
- A passagem de parâmetros em super() chama o construtor apropriado

```
public B() {
    A();   // Call the superconstructor
           // Now continue with this constructor's code
}
```

- O construtor da super classe é executado antes

Polimorfismo

```
public class A {
    public A() { System.out.println("A"); }
}
public class B extends A {
    public B() { System.out.println("B"); }
}
public class C extends B {
    public C() { System.out.println("C"); }
} // ...
A a = new A();
B b = new B();
C c = new C();
```

Polimorfismo

- Três tipos de polimorfismo
  - Sobreposição de método herdado
  - Implementação de método abstrato
  - Implementação de interface
- As formas de polimorfismo são baseadas na ligação dinâmica

Métodos abstratos

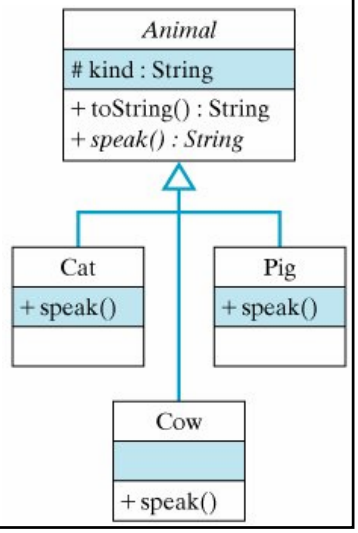
- Métodos que foram **abstratamente** definidos na super classe
  - Métodos sem corpo
  - Métodos com corpo formado por outros métodos abstratos
- Postergam os detalhes de implementação para as diferentes subclasses

Classes abstratas

- Segunda forma de polimorfismo
- Apresentam um ou mais métodos abstratos
- Correspondem a conceitos que não serão instanciados em nenhum momento

Classes abstratas

```
public abstract class Animal {
    protected String kind;
    public Animal()  { }
    public String toString() {
    return kind + " deve " + speak();
    }
    public abstract String speak();
}
// ..
Animal a = new Animal(); // Erro
```



Classes abstratas

- Regras para classes abstratas
  - Classe contendo métodos abstratos deve ser *abstrata*
  - Não pode ser instanciada
  - Deve ser especializada (subclassed)
  - A subclasse pode ser instanciada se foram implementados todos os médodos abstatos, senão continua sendo abstrata
  - Uma classe pode ser abstrata mesmo sem métodos abstratos
    - Contendo variáveis de instâncias comuns às subclasses

Classes abstratas

```
public class Cat extends Animal
    {
    public Cat() {
        kind = “gato";
    }
    public String speak() {
        return “miar";
    }}

public class Cow extends
     Animal {
    public Cow() {
        kind = “boi";
    }
    public String speak() {
        return “mugir";
    }}
```

```
Animal animal = new Cow();
System.out.println(animal.toString());

animal = new Cat();
System.out.println(animal.toString());
```

Classes abstratas

- Extensibilidade
  - Pode-se criar novas subclasses sem a necessidade de redefinir e recompilar as demais classes da hierarquia
  - Contra-exemplo (recompilação em caso de um novo):

```
public String talk(Animal a) {
  if (a instanceof Cow)
     return kind + " deve " +  a.moo();
  else if (a instanceof Cat)
     return kind + " deve " + a.meow();
  else
     return “Sem noção do bicho"; // ou … defina um novo
}
```

Classes abstratas

```
public abstract class Mensagem {
    private String remetente;
    private int status;
    public abstract void play();
    public abstract void testaItens();
    public abstract void aplicaItens();
    public abstract boolean noPadrao();
    public String getRemetente() {
                return this.remetente;
    }
    public String Formata() {
                this.testaItens();
                if (this.noPadrao())
                        this.aplicaItens();
    }}
```

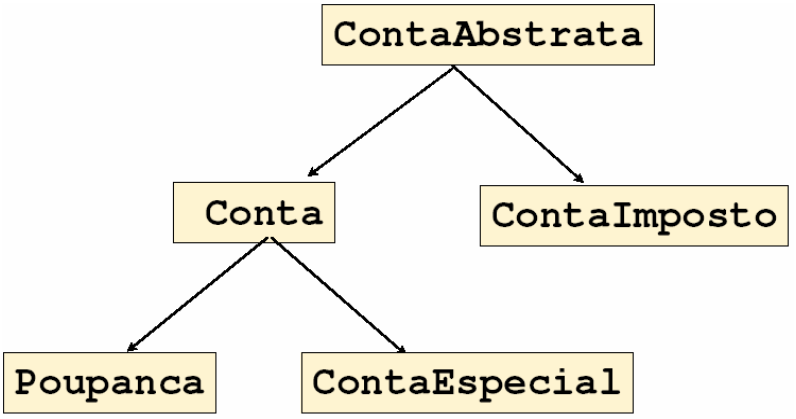
Exemplos

```
public abstract class Empregado {
   private String nome, sobrenome;
   public Empregado(String n, String s) {
                nome= n; sobrenome=s;
   }
   public String getNome() {
                return nome;
   }
   public abstract double calcSalario();}
```

Exemplos

```
public final class Gerente extends Empregado {
    private double salario;
    public Gerente(String n, String s) {
                super(n, s);
                this.salario = 1000.0;
    }
    public double calcSalario() {
                return salario;
    }}
```

Exemplos



Exemplos

```
public abstract class ContaAbstrata {
    private String numero;
    private double saldo;
    public ContaAbstrata(String numero) {
                this.numero = numero;
                saldo = 0.0;
    }
    public void depositar(double valor) {
                saldo += valor;
    }
    public abstract void sacar(double valor);
    // ...
```

Exemplos

```
protected void Saldo(double saldo) {
                this.saldo = saldo;
    }
    public double Saldo() {
                return saldo;
    }
} // ...
```

Exemplos

```
public class Conta extends ContaAbstrata {
   public Conta(String numero) {
            super (numero);
   }
   public void sacar(double valor) {
            this.Saldo(Saldo() - valor);
   }
}
```

Exemplos

```
public class Poupanca extends Conta {
   public Poupanca(String numero) {
            super (numero);
   }
   public void renderJuros(double taxa) {
            depositar( Saldo()*taxa);
   }
}
```

Exemplos

```
public class ContaEspecial extends Conta {
    public static final double TAXA = 0.01;
    private double bonus;
    public ContaEspecial (String numero) {
                super (numero);
    }
    public void depositar(double valor) {
                bonus = bonus + (valor * TAXA);
                super.depositar(valor);
    }
}
```

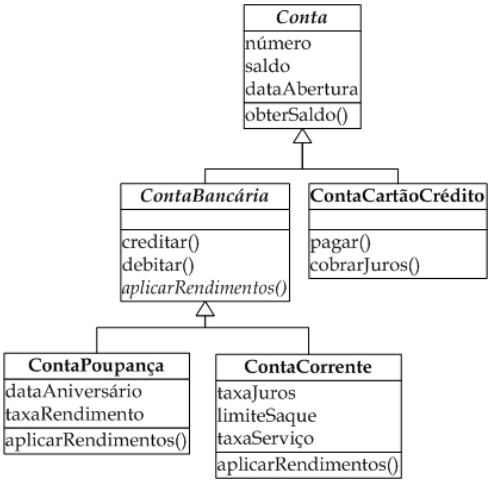
Exemplos

```
public class ContaImposto extends ContaAbstrata {
    public static final double TAXA = 0.001;
    public ContaImposto (String numero) {
                super (numero);
    }
    public void sacar(double valor) {
                double imposto = valor * TAXA;
                double total = valor + imposto;
                this.Saldo( Saldo() – total);
    }
}
```

Exemplos

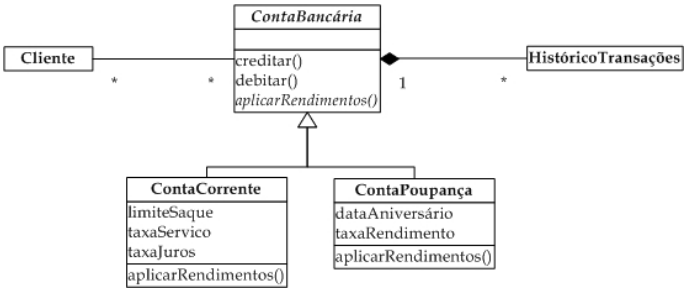
```
public class Programa {
   public static void main(String [ ] args) {
                ContaAbstrata conta1, conta2;
                conta1 = new ContaEspecial("21.342-7");
                conta2 = new ContaImposto("21.987-8");
                conta1.sacar(500);
                conta2.sacar(500);
                System.out.println(conta1. Saldo());
                System.out.println(conta2. Saldo());
   }
}
```

Exemplos



Exemplos

- Na UML, uma classe abstrata é representada com o seu nome em *itálico*.
- No exemplo a seguir, ContaBancária é uma classe abstrata.



Interfaces

- Terceira forma de polimorfismo
- Similar a classes,
  - mas contém apenas definição de métodos abstratos
  - e variáveis finais
  - Nenhuma variável de instância

Interfaces

- Também conhecidas como classes abstratas “puras”
- Não têm construtores
- Não contém atributos de dados (a não ser constantes estáticas – public, static, final)
- Todos os métodos são implicitamente abstratos
- Provêm uma interface de serviços e comportamentos

Interfaces

- Forte separação entre funcionalidade e implementação.
  - Embora parâmetros e tipos de retorno sejam obrigatórios.
- Clientes interagem independentemente da implementação.
  - Mas os clientes podem escolher implementações alternativas.

Interfaces

- Especifica que métodos devem ser implementados pelas classes que a implementarem
- Alternativa a um modelo usando classes abstratas
  - Em vez de definir speak() dentro da superclasse,
  - ele deve ser um método abstrato de uma interface "Speakable"

Interfaces

```
public interface Speakable {
    public String speak();
}
public class Animal {
    protected String kind; public Animal() { }
    public String toString() {
        return kind + " deve " + ((Speakable)this).speak();
 } }
```

- A classe Animal não é mais abstrata.

Interfaces

```
public interface Speakable {
    public String speak();
}
```

```
public class Animal {
    protected String kind;
    public Animal() { }
    public String toString() {
        return kind + " deve " + ((Speakable)this).speak();
} }
```

- Método *speak()*
  - Definido e implementado nas subclasses
  - Chamado indiretamente na classe abstrata
  - *This* habilita sua ligação dinâmica com o *speak()* do objeto implementando "*Speakable*" (sintaticamente, uma superclasse)

Interfaces

- *Speak()* pode ser formalmente implementado nas subclasses, usando "*implements*"

```
public class Cat extends Animal implements Speakable {
    public Cat() { kind = "gato"; }
    public String speak() { return "miar"; }
}
public class Cow extends Animal implements Speakable {
    public Cow() { kind = "boi"; }
    public String speak() { return "mugir"; }
}
```

Interfaces

```
public class Cat extends Animal {
    public Cat() { kind = “gato"; }
    public String speak() { return "miar"; } // na superclasse??
}
public static void main(String[] args) {
        Animal animal = new Cat();
        System.out.println(animal.toString());  // A cat goes meow
    }
```

Exception in thread "main" java.lang.ClassCastException: Cat
at Animal.toString(Animal.java:5)
at Bicharada.main(Bicharada.java:12)

Abstração de classes

- Herança múltiplas

  ((Speakable)this).speak(); // molda este objeto em um objeto Speakable

  - Um Animal não necessariamente tem um método *speak()*
  - Para invocar *speak()* de um objeto de uma das subclasses de Animal o objeto deve ser “Speakable”
  - O gato é “Animal” e “Speakable”

Abstração de classes

- Herança múltiplas
  - A interface faz parte da lista de tipos das classes que a implementam
  - A implementação de Interface é uma forma de herança
  - Uma classe java pode ser diretamente subclasse de uma única classe, mas pode implementar várias interfaces

Abstração de classes

- Herança múltiplas
  - Herança múltipla pode causar conflitos de métodos iguais com implementações diferentes - Java não suporta
  - Uma classe pode “implementar” vários serviços (múltiplas interfaces) causando um efeito similar à herança múltipla

Abstração de classes

- Classe abstrata ou Interface
  - Métodos definidos em uma classe abstrata
    - Acessam variáveis comuns a todas as subclasses
    - Realizam ações inerentes à toda hierarquia de classes
    - Contribuem fundamentalmente para a definição básica de objetos

Abstração de classes

- Interface ou Classe abstrata
  - Métodos definidos em uma interface
    - Independem de uma hierarquia particular de classes
    - Podem ser agregados a qualquer classe
    - São flexíveis, simulando herança múltipla
    - Definem uma relação mais fraca com as demais classes
    - Definem regras, funcionalidades, serviços

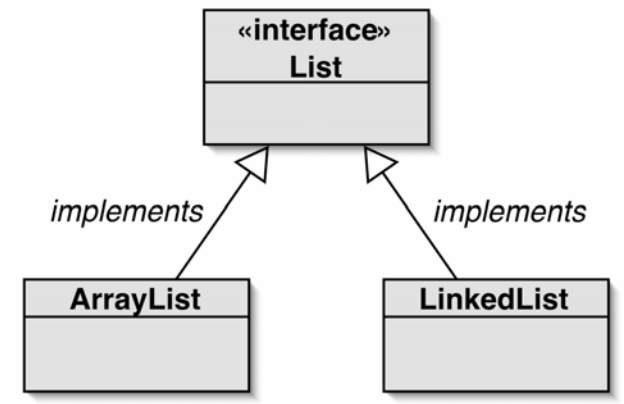
Exemplos

- Exemplo um veículo pode acelerar, frear e virar para uma direção

```
public interface Veiculo {
        public freiar();
        public acelerar();
        public virar(direcao);
}
```

- As classes caminhão, trator, carroça, ..., que implementam a interface veículo é que dizem como efetuar estas operações em cada classe
- Pode-se projetar sistemas e serviços utilizando interfaces sem a preocupação com sua implementação

Exemplos



Exemplos

```
interface Leitor {
    String lendo();
}
interface Programador {
    void pensando(char[] ideias);
    String digitando();
}
```

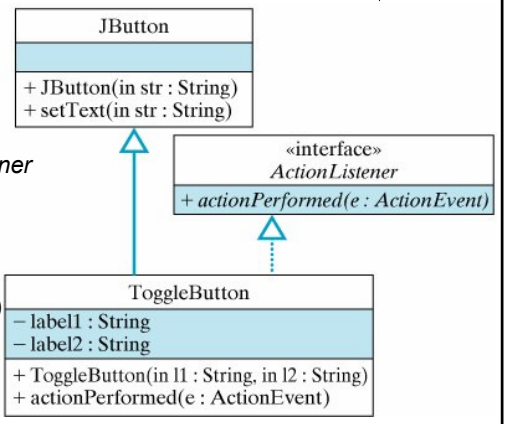
```
class ParticipanteForum implements Leitor, Programador
{
        String pensamento;
        public String lendo() { return "Forum"; }
        public void pensando(char[] ideias) {
                pensamento = new String(ideias);
        }
        public String digitando() {
                return pensamento;
        }
        // método exclusivo desta classe
        private String aprendendo() {
                return "Java";
        }
}
```

Exemplos

- Um *ToggleButton* é um *JButton* com dois "labels" que implementa *ActionListener*.

Exemplos

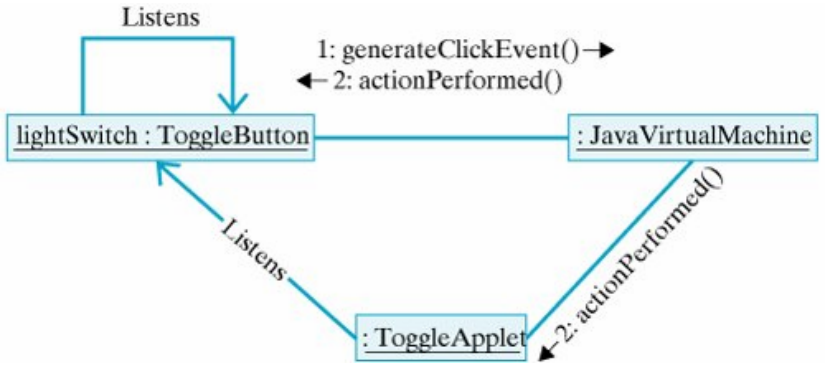
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ToggleButton
extends JButton implements ActionListener
{
  private String label1;
  private String label2;

  public ToggleButton(String l1, String l2)
 {
    super(l1);
    label1 = l1;
    label2 = l2;
    addActionListener(this);
  }
// ...
```



Exemplos

```
public void actionPerformed(ActionEvent e) {
    String tempS = label1;
    label1 = label2;
    label2 = tempS;
    setText(label1);
  } // actionPerformed()
} // ToggleButton class
```

Exemplos



Exemplos


```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ToggleApplet extends JApplet implements ActionListener {
  private ToggleButton lightSwitch;

  public void init() {
    lightSwitch = new ToggleButton ("off","on");
    getContentPane().add(lightSwitch);
    lightSwitch.addActionListener(this);
  } // init()

  public void actionPerformed(ActionEvent e) {
    showStatus("The light is " + lightSwitch.getText());
  } // actionPerformed()
} // ToggleApplet class
```

Exemplos

- Princípios de POO em uso
  - Herança para extender funcionalidade de um predefinido *JButton*
  - Encapsula o comportamento *ToggleButton* em uma classe.
  - Oculta o mecanismo pelo qual os rótulos são gerenciados
  - Um *ToggleButton* faz tudo o que um *JButton* faz e mais um comportamento específico

Exemplos

```
public interface Funcionario {
    public double calcularSalario();
    public int bonus();
}

class Gerente implements Funcionario {
    private static final int SALARIO = 40000;
    private static final int BONUS = 0;
    public double calcularSalario() {
                return SALARIO;
    }
    public int getBonus( ) { return BONUS; }
}
```

Exemplos

```
class Programador implements Funcionario {
    private static final double SALARIO = 50000;
    private static final int BONUS = 10000;
    public double calcularSalario() {
            return SALARIO;
    }
    public int getBonus() { return BONUS; }
}
```

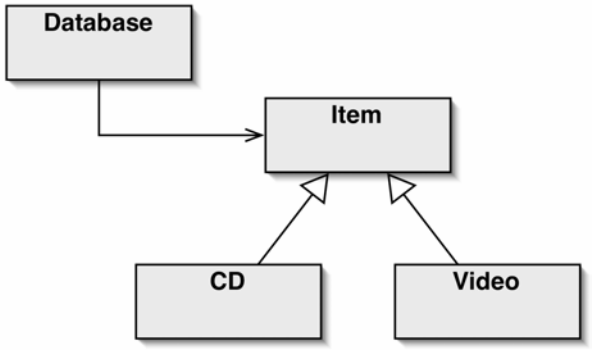
Exemplos

```
public class FolhaPagamento {
    public double calcularFolhaPagamento(Funcionario emp) {
        return emp.calcularSalario() + emp.bonus();
    }
    public static void main(String arg[]) {
        FolhaPagamento fp = new FolhaPagamento();
        Programador prg = new Programador();
        Gerente mgr = new Gerente();
        System.out.println("Salário do Programador " +
                fp.calcularFolhaPagamento(prg));
        System.out.println("Salário do Gerente " +
                fp.calcularFolhaPagamento(mgr));
    } }
```

Exemplos

```
public interface Transacao {
    public void depositar (double valor);
    public boolean sacar (double valor);
}
public class Conta implements Transacao{
private double saldo;
private String numero;
public void depositar (double valor){ saldo += valor; }
public boolean sacar (double valor){
    if (saldo >= valor){
            saldo -= valor;
            return true;
    }
    return false;
}
public String toString (){ return "" + numero + "-" + saldo; } }
```

Exemplos



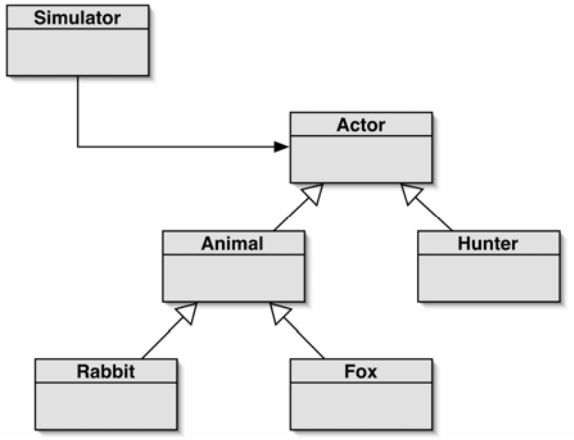
Exemplos

```
public class Database
{
 public void addItem(Item theItem)
    {
        ...
    }
}
```

```
Video video = new Video(...);
CD cd = new CD(...);

database.addItem(video);
database.addItem(cd);
```

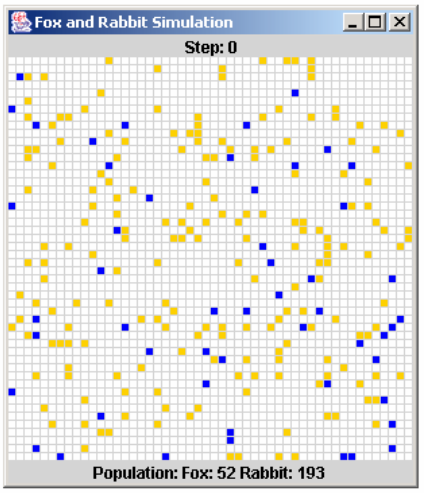
Exemplos

- Simulador



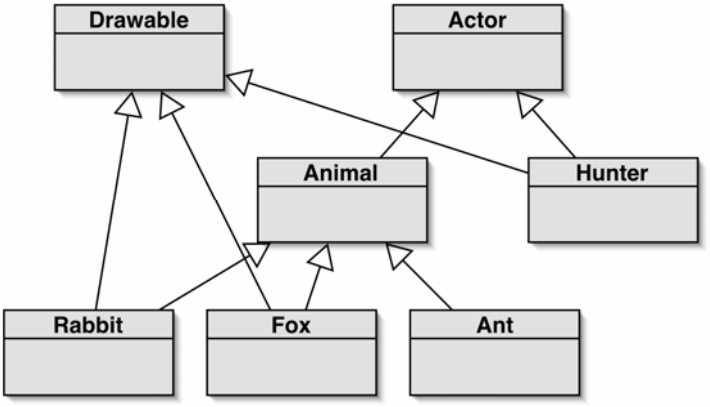
Exemplos

- Simulador



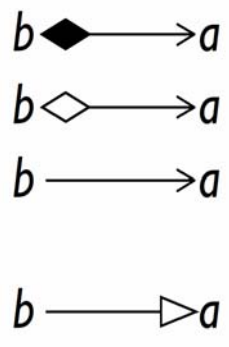
Exemplos

- Simulador com visualização separada da atuação



Relacionamento entre classes

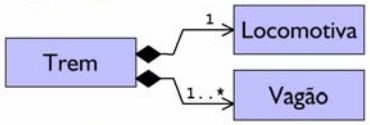
- Tipos básicos:
  - Composição: a “é parte essencial de” b
  - Agregação: a “é parte de” b
  - Associação: a “é usado” por b
  - Herança: b “é” a
  - Herança: b “é um tipo de” a
- UML:



  - The Unified Modeling Language (UML)

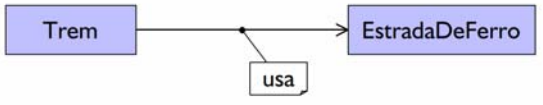
Relacionamento entre classes

- ***Composição***: *um trem é formado por locomotiva e vagões*



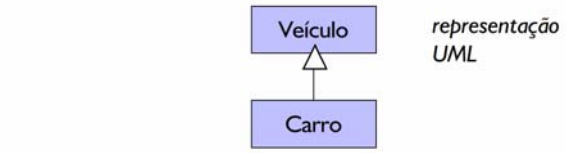
- ***Agregação***: *uma locomotiva tem um farol (mas não vai deixar de ser uma locomotiva se não o tiver)*

- ***Associação***: *um trem usa uma estrada de ferro (não faz parte do trem, mas ele depende dela)*

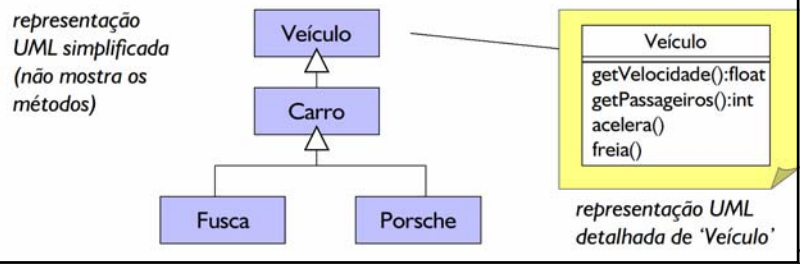


Relacionamento entre classes

- *Um carro é um veículo*

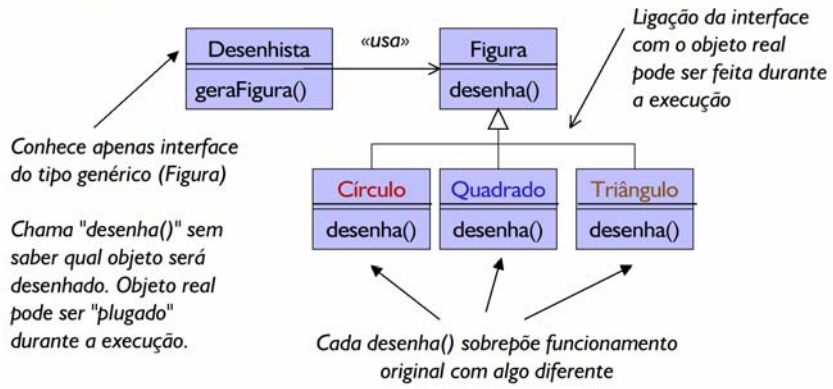


- *Fuscas e Porsches são carros (e também veículos)*



Relacionamento entre classes

- *Uso de um objeto no lugar de outro*
  - *pode-se escrever código que não dependa da existência prévia de tipos específicos*



Relacionamento entre classes

- Associação
  - Complementa a informação que se tem sobre dois objetos/classes em um determinado instante,
  - Referencia informação associativa nova
  - Pode ser fruto de uma operação/transação ocorrida que envolva duas ou mais classes
  - Uma associação complementa a informação necessária para que dois ou mais conceitos façam sentido

Relacionamento entre classes

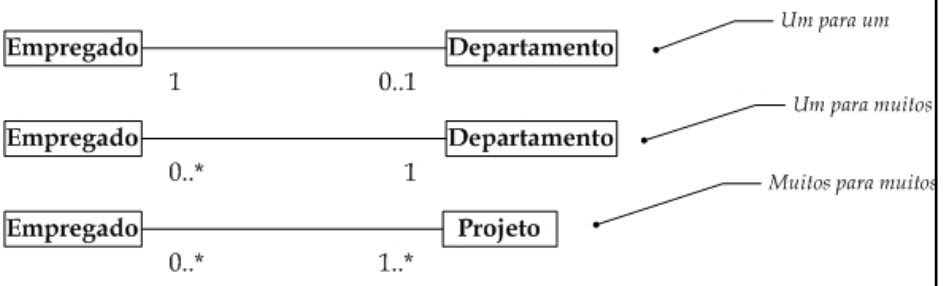
- Classe associativa
  - É uma classe que está ligada a uma associação, em vez de estar ligada a outras classes.
  - É normalmente necessária quando duas ou mais classes estão associadas, e é necessário manter informações sobre esta associação.
  - Uma classe associativa pode estar ligada a associações de qualquer tipo de conectividade.

Relacionamento entre classes

- Multiplicidade
  - Pode haver um cliente que esteja associado a vários pedidos.
  - Pode haver um cliente que não esteja associado a pedido algum.
  - Um pedido está associado a um, e somente um, cliente.

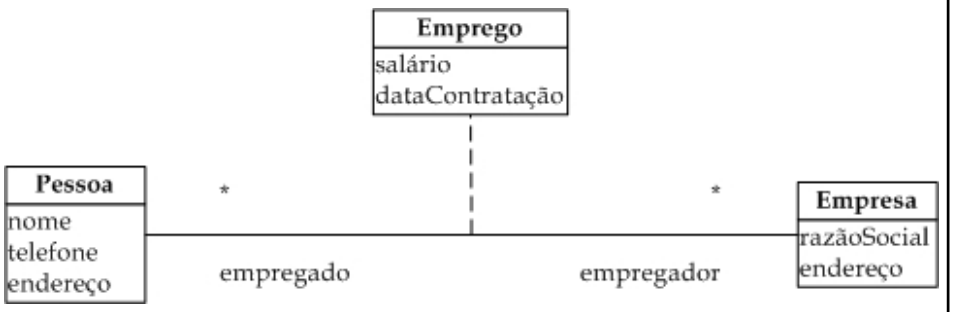
Relacionamento entre classes

- Conectividade



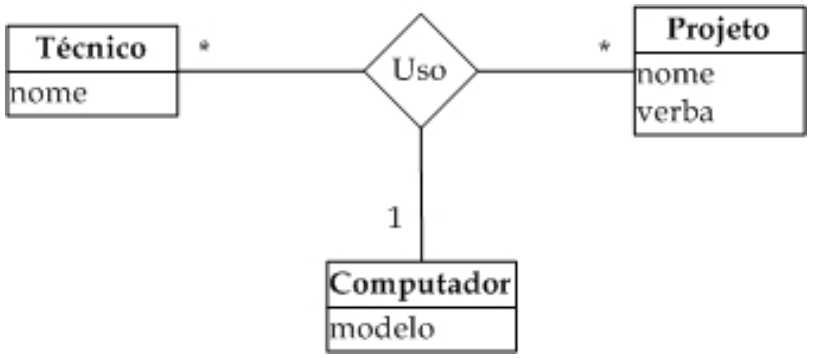
Relacionamento entre classes

- A classe é ligada a uma associação por uma linha tracejada.

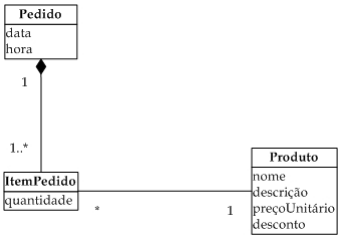


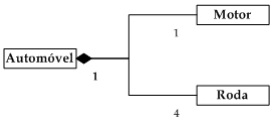
Relacionamento entre classes

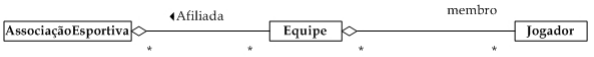
- Associação ternária
  - Na notação da UML, as linhas de uma associação n-ária se interceptam em um losango nomeado.

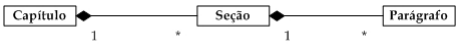


Exemplos

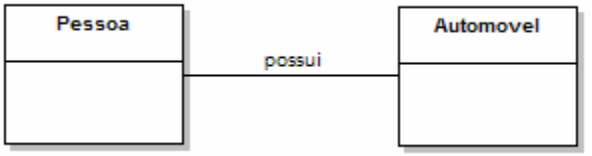


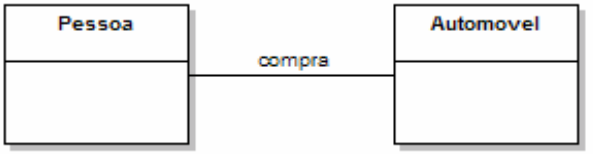


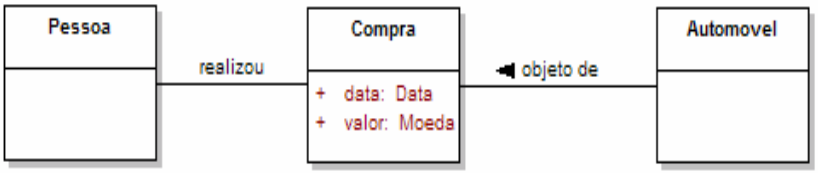




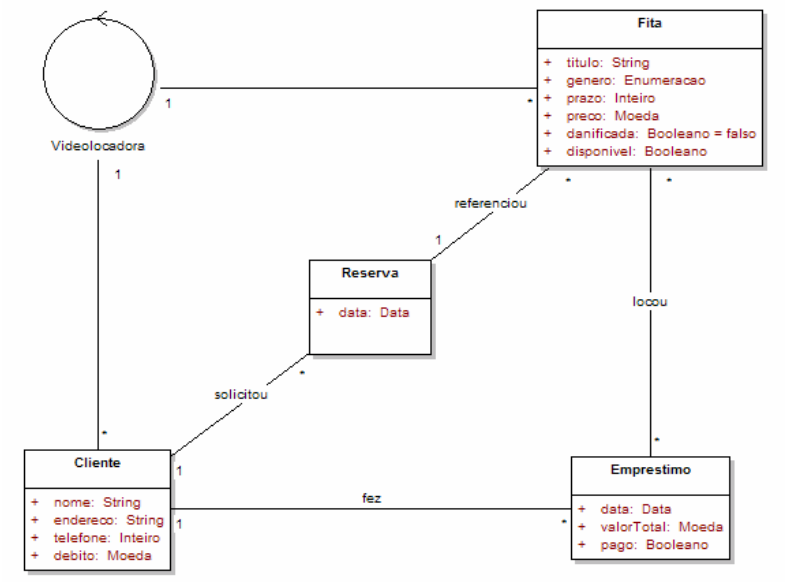
Exemplos







Exemplos



Exercícios

- Um hotel de uma rede possui clientes que costumam ficar hospedados nele.
- Em geral, eles reservam um ou mais apartamentos de determinados tipos, considerando suas necessidades, para um período específico, se houver disponibilidade.
- Quando chegam, os hóspedes fazem o checkin, independente de terem feito reservas.
- Todo o consumo no hotel, como diárias, refeições e ligações, é consolidado no checkout, ficando um histórico de estadias e serviços consumidos para consulta posterior.

Projeto Livraria Virtual LeiaBem

...