

# Linguagem de programação 001

Encapsulamento

# Relembrando

Atributos

Métodos

This

Referência A objetos

Vantagens ao usar métodos? Classes?

# Equals

método equals() é uma forma de comparar se um objeto é igual a outro. Todos as classes fornecidas possuí um método equals.

E nós podemos criar a nossa método equals também

# Equals

```
boolean equals (Livro l){  
    if (l.isbn ==this.isbn ){  
        return true;  
    }else{  
        return false;  
    }  
}
```

- \* escolhemos os atributos que queremos utilizar para comparar os objetos

# Equals

```
boolean equals (Livro l){  
    if ((l.isbn ==this.isbn ) && (l.nome.equals(this.nome))){  
        return true;  
    }else{  
        return false;  
    }  
}
```

Com mais de um atributo

# Encapsulamento

Encapsulamento serve para controlar o acesso aos atributos e métodos de uma classe. É uma forma eficiente de proteger os dados manipulados dentro da classe, além de determinar onde esta classe poderá ser manipulada.

# Encapsulamento

Imagine que na nossa livraria, uma nova regra de negócio é criada e que um livro pode ter no máximo 30% de desconto.

Ou seja não é possível dar descontos maiores que 30% e devemos garantir no código isso!

Vamos evoluir nosso método

# Encapsulamento

```
public boolean aplicaDescontoDe(double porcentagem) {  
    if (porcentagem > 0.3) {  
        return false;  
    }  
    this.valor -= this.valor * porcentagem;  
    return true;  
}
```

```
if (!livro.aplicaDescontoDe(0.1)){  
    System.out.println("Desconto não pode ser maior do que 30%");  
}
```

# Encapsulamento

Nossos métodos devem ser o mais genéricos e flexíveis possível para que podemos utilizá-los nas mais diversas possibilidades

Lembre-se sempre de evitar deixar informações tão específicas em seus moldes, além de não sobrecarregá-los com comportamentos que não deveriam ser de sua responsabilidade.

# Encapsulamento

Nosso código está encapsulado? nossa regra de negocio  
será sempre implementada?

```
li.mostraDetalhes();
if (!li.aplicaDescontoDe(0.4)){
    System.out.println("desconto superior ao permitido");
}
li.mostraDetalhes();
li.valor -= li.valor*0.4;
li.mostraDetalhes();
```

# Modificadores de acesso

**public** - Uma declaração com o modificador public pode ser acessada de qualquer lugar e por qualquer entidade que possa visualizar a classe a que ela pertence.

**private** - Os membros da classe definidos como não podem ser acessados ou usados por nenhuma outra classe. Esse modificador não se aplica às classes, somente para seus métodos e atributos. Esses atributos e métodos também não podem ser visualizados pelas classes herdadas.

**protected** - O modificador protected torna o membro acessível às classes do mesmo pacote ou através de herança, seus membros herdados não são acessíveis a outras classes fora do pacote em que foram declarados.

**default (padrão)**: A classe e/ou seus membros são acessíveis somente por classes do mesmo pacote, na sua declaração não é definido nenhum tipo de modificador, sendo este identificado pelo compilador

# Modificadores de acesso

Ao alterar a visibilidade do atributo ou método valor para private todos os lugares que acessavam esse atributo passaram a não compilar, já que apenas a própria classe Livro pode fazer isso. Por exemplo, observe como estávamos passando o valor do livro:  
`livro.valor = 100;`

# Métodos Get/Set

Os métodos get e set servem para setar valores em atributos (set) e os get para Retornar o valor do atributo

# Quando usar private?

quando não quero que ninguém acesse esse atributo diretamente ?

Mas a questão é: quando eu quero que alguém acesse algum atributo de forma direta?

padrão usarmos atributos sempre como private!

# Métodos get/set

```
public String getNome() {  
    return nome;  
}  
  
public void setNome(String nome) {  
    this.nome = nome;  
}  
  
public String getDescricao() {  
    return descricao;  
}  
  
public void setDescricao(String descricao) {  
    this.descricao = descricao;  
}  
  
public double getValor() {  
    return valor;  
}  
  
public void setValor(double valor) {  
    this.valor = valor;  
}
```

# Código encapsulado

Um bom termômetro para descobrir se o seu código está bem encapsulado é fazendo as duas perguntas:

- O que esse código faz?
- Como esse código faz?

Veja por exemplo quando aplicamos o desconto acessando o atributo diretamente:

`livro.valor -= livro.valor * 0.4;` X

`livro.aplicaDescontoDe(0.1);` C

# Prática

Altere as classe criadas na aula passada (ContaCorrente e Cliente)para que elas fiquem encapsuladas. Adicione na conta Corrente o Atributo limite.

Aplique as seguintes regras de negócio

- 1- uma conta corrente só pode sacar um valor se o mesmo tiver saldo. O limite faz parte do saldo da conta.
- 2- Toda conta corrente quando é criada recebe como saldo 200 reais. Se o cliente é maior que 60 anos seu saldo é de 300 reais.
- 3 - Não é possível criar contas ou clientes iguais. Para considerar um cliente igual ao outro utilize o atributo cpf. Para conta corrente considere o numero da conta e o titular da conta
- 4 - Crie um método para mostrar o saldo;
- 5 - O nome do cliente deve ser salvo em letras maiúsculas
- 6 - Crie uma classe para testar os métodos nessa classe criando 2 objetos conta.