

Programação Orientada a Objetos (PO24CP)

Aula #04 - Classes e objetos: atributos e métodos

Prof^a Luciene de Oliveira Marin
lucienemarin@utfpr.edu.br

Classe e objetos: atributos e métodos

Modificadores de acesso: **public** e **private** (1/3)

Paradigma da programação orientada a objetos

- **Objetos interagem** com objetos através da **troca mensagens**
- A troca de mensagens ocorre através da **invocação de métodos** de objetos

Encapsulamento

- Emissor da mensagem **não precisa saber como o resultado foi obtido**, para este só importa o resultado
- O emissor precisa **conhecer quais operações o receptor sabe realizar** ou quais informações o receptor pode fornecer

Modificadores de acesso

- Indicam **quais atributos e métodos** de um objeto estarão **visíveis aos demais objetos** do sistema

Modificadores de acesso: `public` e `private` (2/3)

`private`

Os membros de uma classe (atributos e métodos) definidos como privados só poderão ser acessados (invocados) pelos demais **métodos da própria classe.**

`public`

Os membros de uma classe definidos como públicos poderão ser invocados por **métodos de qualquer classe.**

Princípios da POO

- Geralmente **atributos** de uma classe devem ser declarados como **privados**
- **Métodos** geralmente devem ser **públicos**, porém há casos que um método só interessa a própria classe e assim este deve ser **privado**
- **Qual a vantagem?** Isto **garante a integridade do estado do objeto**, pois somente métodos da própria classe poderão alterá-lo (encapsulamento).

Modificadores de acesso: exemplo não ideal

```
public class CarroNaoldeal{  
  
    // atributos  
    public float velocidade;  
  
    // metodos  
    public void definirVelocidade(float v){  
        if (v <= 200){ velocidade = v;}  
        else velocidade = 0;  
    }  
  
    public void acelerar(float v){  
        // o carro so' pode atingir 200km/h  
        if ((velocidade + v) <= 200){ velocidade += v;}  
        else velocidade = 200;  
    }  
}
```

Modificadores de acesso: exemplo não ideal

- Classe aplicativa UsaCarro

```
public class UsaCarro{  
  
    public static void main(String[] args){  
  
        //declarando o objeto fusca da classe CarroNaoldeal  
        CarroNaoldeal fusca = new CarroNaoldeal();  
  
        //alterando a velocidade atraves dos metodos do objeto  
        fusca.definirVelocidade(150); // velocidade = 150  
        fusca.acelerar(400); // velocidade = 200  
  
        //alterando diretamente o valor do atributo  
        fusca.velocidade = 400;  
    }  
}
```

Modificadores de acesso: exemplo ideal

```
public class CarroIdeal{

    // atributos
    private float velocidade;

    // metodos
    public void definirVelocidade(float v){
        if (v <= 200){ velocidade = v; }
        else velocidade = 0;
    }

    public void acelerar(float v){
        // o carro so' pode atingir 200km/h
        if ((velocidade + v) <= 200){ velocidade += v; }
        else{ velocidade = 200; }
    }
}
```


Modificadores de acesso: exemplo ideal

```
public class UsaCarro{  
  
    public static void main(String[] args){  
  
        //declarando o objeto fusca da classe Carroldeal  
        Carroldeal fusca = new Carroldeal();  
  
        // alterando a velocidade atraves dos metodos do objeto  
        fusca.definirVelocidade(150); // velocidade = 150  
        fusca.acelerar(400); // velocidade = 200  
  
        // alterando diretamente o valor do atributo  
        fusca.velocidade = 400; // Erro! nao ira' compilar  
    }  
}
```

Valores iniciais de atributos

```
public class Pessoa{  
    private String nome;  
    private String cpf;  
    private int anoNasc;  
  
    public void imprimirDados(){  
        System.out.println("Nome: " + nome);  
        System.out.println("CPF: " + cpf);  
        System.out.println("Ano: " + anoNasc);  
    }  
} // fim da classe
```

```
public class DemoPessoa  
{  
    public static void main(String[] args)  
    {  
        Pessoa p = new Pessoa();  
        p.imprimirDados();  
    }  
}
```

O que será impresso ao executarmos a classe applicativa?

Nome:
CPF:
Ano: 0

Valores iniciais de atributos

Em Java atributos de um objeto que não forem iniciados na criação deste objeto, receberão valores padrões

- números ficam 0,
- boolean com false e
- referências de objetos com null

Uma boa prática de programação

Sempre iniciar os atributos de forma explícita

```
Pessoa p = new Pessoa();  
  
p.definirNome("Joao");  
p.definirCPF("123.456.789-00");  
p.definirAno(1950);
```

Método construtor

- Método especial cujo objetivo é iniciar com valores os atributos de um objeto.
- Deve possuir o **mesmo nome da classe** e **não possuir tipo de retorno**.

Exemplos:

a) `CarroIdeal fusca = new CarroIdeal();`

b) `Pessoa p = new Pessoa();`

Sobrecarga de métodos

- Consiste em declarar métodos com o mesmo nome, porém com **assinaturas** diferentes.
 - A **assinatura** de um método é dada pelo **tipo de retorno** e pela **lista de parâmetros**.
- ⇒ Uma classe pode conter métodos construtores **sobrecarregados**.
- ⇒ **Ao criar um objeto o desenvolvedor indica qual construtor irá chamar.**

Método construtor sobrecarregado - exemplo

```
public class Pessoa{  
  
    private String nome, cpf;  
    private int anoNasc;  
  
    // metodo construtor padrao  
    public Pessoa(){  
        nome = ""; cpf = ""; anoNasc = 0;  
    }  
  
    // metodo construtor com 1 parametro  
    public Pessoa(String no){  
        nome = no; cpf = ""; anoNasc = 0;  
    }  
  
    // metodo construtor com 3 parametros  
    public Pessoa(String no, String c, int a){  
        nome = no; cpf = c; anoNasc = a;  
    }  
} // fim da classe
```

Invocando métodos construtores

```
Pessoa a = new Pessoa();  
Pessoa b = new Pessoa("Maria");  
Pessoa c = new Pessoa("Maria", "123.456.789-00", 1959);
```

Nota:

Método **construtor padrão** é aquele cuja lista de parâmetros está vazia. **Toda classe Java possui um construtor padrão vazio implícito.**

Mais exemplos...

Exemplo 1 - conta bancária (1/2)

```
public class ContaBancaria
{
    private String nomeCorrentista;
    private double saldo;
    public ContaBancaria(String n, double s)
    {
        nomeCorrentista = n;
        saldo = s;
    }
    public double getSaldo() { return saldo; }
    public String getNome() { return nomeCorrentista; }
    public void deposita(double quantia)
    {
        saldo = saldo + quantia;
    }
    public void retira(double quantia)
    {
        if (quantia <= saldo) saldo = saldo - quantia;
    }
}
```

Exemplo 1 - conta bancária (2/2)

```
public class UsaContaBancaria
{
    public static void main(String[] args)
    {
        ContaBancaria fred = new ContaBancaria("Fred",1000);
        ContaBancaria richard = new ContaBancaria("Richard",2000);
        richard.retira(500);
        fred.deposita(500); // ok
        richard.saldo = 1000000; // Erro de compilação!
    }
}
```

Exemplo 2 - conta bancária melhorada (1/2)

```
public class ContaBancaria2
{
    private String nomeCorrentista;
    private double saldo;
    public ContaBancaria2(String n, double s)
    {
        nomeCorrentista = n;
        saldo = s;
    }
    public double getSaldo() { return saldo; }
    public String getNome() { return nomeCorrentista; }
    public void deposita(double quantia)
    { saldo = saldo + quantia; }
    public void retira(double quantia)
    { if (quantia <= saldo) saldo = saldo - quantia; }
    //Melhorias...
    public void transfereDe(ContaBancaria2 de, double quanto)
    { deposita(quanto); //Má idéia. Por quê?
      de.retira(quanto); }

    public String toString()
    { return "Conta de "+nomeCorrentista+" tem saldo "+saldo; }
}
```

Exemplo - conta bancária melhorada (2/2)

```
class UsaContaBancaria2
{
    public static void main(String[] args)
    {
        ContaBancaria2 fred = new ContaBancaria2("Fred",1000);
        ContaBancaria2 richard = new ContaBancaria2("Richard",2000);
        System.out.println(fred);
        System.out.println(richard);
        fred.transfereDe(richard,400);
        System.out.println(fred);
        System.out.println(richard);
    }
}
```

Saída:

```
Conta de Fred tem saldo 1000.0
Conta de Richard tem saldo 2000.0
Conta de Fred tem saldo 1400.0
Conta de Richard tem saldo 1600.0
```

Exercícios de fixação

Exercícios (1/10)

- 1) Qual é a diferença entre uma classe e um objeto?
- 2) Como uma classe é definida?
- 3) Cada objeto tem sua própria cópia de que?
- 4) Que nome tem um construtor?
- 5) O que **new** faz?
- 6) Um construtor pode ter um ou mais parâmetros?
- 7) Se um método não retornar um valor, qual deve ser seu tipo de retorno?

8) Crie uma classe **Die** com uma variável de instância inteira **sideUp**. Forneça a ela um construtor, um método **getSideUp()** que retorne o valor de **sideUp** e um método **void roll()** que altere o valor de **sideUp** para um valor aleatório de 1 a 6. Em seguida, crie uma classe **DieDemo** com um método principal que gere dois objetos **Die**, jogue-os e exiba a soma dos dois lados superiores.

9) Escreva em Java a classe **NumeroComplexo** que represente um número complexo. A classe deverá ter mais de um método construtor e os seguintes métodos:

- **inicializaNumero**, que recebe dois valores como argumentos para inicializar os campos da classe (parte real e imaginária);
- **imprimeNumero**, que deve imprimir o número complexo encapsulado usando a notação $a + bi$ onde a é a parte real e b a imaginária;
- **eIgual**, que recebe outra instância da classe **NumeroComplexo** e retorna true se os valores dos campos encapsulados forem iguais aos da instância passada como argumento;
- **soma**, que recebe outra instância da classe **NumeroComplexo** e soma este número complexo com o encapsulado usando a fórmula $(a + bi) + (c + di) = (a + c) + (b + d)i$;

9) continuação...

- **subtrai**, que recebe outra instância da classe `NumeroComplexo` e subtrai o argumento do número complexo encapsulado usando a fórmula
$$(a + bi) - (c + di) = (a - c) + (b - d)i;$$
- **multiplica**, que recebe outra instância da classe `NumeroComplexo` e multiplica este número complexo com o encapsulado usando a fórmula
$$(a + bi) * (c + di) = (ac - bd) + (ad + bc)i;$$
- **divide**, que recebe outra instância da classe `NumeroComplexo` e divide o número encapsulado pelo passado como argumento usando a fórmula
$$\frac{(a + bi)}{(c + di)} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i;$$

10) Escreva uma aplicação (classe aplicativa) que demonstre o uso de instâncias da classe **NumeroComplexo** criada no item anterior e demonstre o uso de todas as operações.

11) Crie uma classe chamada **IntegerSet**. Cada objeto **IntegerSet** pode armazenar inteiros no intervalo de 0 a 100. O conjunto é representado por um array de booleans. O elemento de array $a[i]$ é `true` se o inteiro i estiver no conjunto. O elemento do array $a[j]$ é `false` se o inteiro j não estiver no conjunto. O construtor sem argumento inicializa o array Java como “conjunto vazio” (isto é, um conjunto cuja representação de array contém todos os valores `false`). Forneça os seguintes métodos:

12) continuação...

- O método **union** cria um terceiro conjunto que é a união teórica de dois conjuntos existentes (isto é, um elemento do terceiro array do conjunto é configurado como `true` se esse elemento for `true` em qualquer um dos conjuntos existentes ou em ambos; caso contrário, o elemento do terceiro conjunto é configurado como `false`).
- O método **intersection** cria um terceiro conjunto que é a interseção teórica de dois conjuntos existentes (isto é, um elemento do array do terceiro conjunto é configurado como `false` se esse elemento for `false` em qualquer um ou em ambos os conjuntos existentes - caso contrário, o elemento do terceiro conjunto é configurado como `true`).

12) continuação...

- O método `insertElement` insere um novo inteiro k em um conjunto (configurando $a[k]$ como `true`).
- O método `deleteElement` exclui o inteiro m (configurando $a[m]$ como `false`).
- O método `toSetString` retorna uma string contendo um conjunto como uma lista de números separados por espaços. Inclua somente os elementos que estão presentes no conjunto. Utilize - - para representar um conjunto vazio.
- O método `isEqualTo` determina se dois conjuntos são iguais.

13) Escreva uma classe aplicativa para testar a classe **IntegerSet**. Teste se todos os seus métodos funcionam adequadamente.

14) Crie uma classe chamada **Rational** para realizar aritmética com frações. Utilize variáveis do tipo inteiro para representar as variáveis de instância `private` da classe - o `numerator` e o `denominator`. Forneça um construtor que permita que um objeto dessa classe seja inicializado quando ele for declarado. O construtor deve armazenar a fração em uma forma reduzida. A fração $\frac{2}{4}$ é equivalente a $\frac{1}{2}$ e seria armazenada no objeto como 1 no `numerator` e 2 no `denominator`. Forneça um construtor sem argumento com valores-padrão caso nenhum inicializador seja fornecido. Forneça métodos `public` que realizam cada uma das operações a seguir:

14) continuação..

- **Somar** dois números **Rational**: O resultado da adição deve ser armazenado na forma reduzida.
- **Subtrair** dois números **Rational**: O resultado da subtração deve ser armazenado na forma reduzida.
- **Multiplicar** dois números **Rational**: O resultado da multiplicação deve ser armazenado na forma reduzida.
- **Dividir** dois números **Rational**: O resultado da divisão deve ser armazenado na forma reduzida
- **Imprimir** números **Rational** na forma a/b , onde a é o numerator e b é o denominator.
- **Imprimir** os números **Rational** no formato de ponto flutuante. (Considere a possibilidade de fornecer capacidades de formatação que permitam que o usuário da classe especifique o número de dígitos de precisão à direita do ponto de fração decimal.)

15) Escreva uma classe aplicativa para testar as funcionalidade da classe **Rational**.