Introdução à Programação
Orientada a Objetos com Java

Encapsulamento, Classes Paramétricas, Métodos e Atributos Estáticos

Centro de Informática
Universidade Federal de Pernambuco

Encapsulamento (Information Hiding)

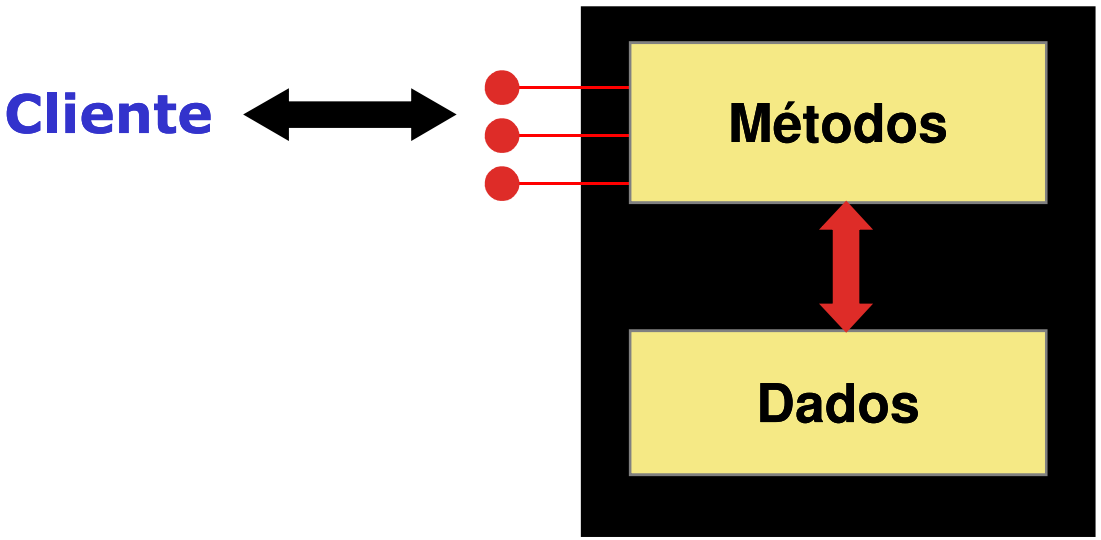
- Um objeto pode ser visto de duas formas diferentes:
  - Internamente
    - Detalhes de variáveis e métodos da classe que o define
  - Externamente
    - Serviços que um objeto fornece e como este objeto interage com o resto do sistema (a interface do objeto)
- A visão externa de um objeto **encapsula** o modo como são fornecidos os serviços
- Isto é, esconde os detalhes de implementação do objeto (**information hiding**)

Encapsulamento (Information Hiding)

- Um objeto (chamado neste caso de cliente) pode usar os serviços fornecidos por um outro objeto
  - Contudo, um cliente não deve precisar saber os detalhes de implementação destes métodos
- Mudanças no estado (atributos) de um objeto devem ser feitas pelos métodos do objeto
- Para permitir uma maior independência entre os objetos,o acesso direto aos atributos de um objeto por um outro objeto deve ser restrito ou quase impossível

Encapsulamento (Information Hiding)

- Um objeto pode ser visto como uma "caixa preta", onde os detalhes internos são escondidos dos clientes
- Clientes acessam o estado do objeto, através dos métodos oferecidos



Modificadores de Acesso

- Em Java, o encapsulamento é possível através do uso apropriado de **modificadores de acesso**
  - Modificadores são palavras reservadas que especificam características particulares de um método ou atributo
- Modificadores podem ser:
  - `public`
  - `protected`
  - `private`

Modificadores de Acesso

- Membros da classe que recebem o modificador **`public`**, podem ser acessados por qualquer outra classe
  - Devem ser utilizados para métodos que definem a interface da classe
  - Não deve ser utilizado para os atributos, excetuando-se o caso onde queremos declarar uma constante
- Membros que recebem o modificador **`private`**, só podem ser acessados dentro da classe
  - Devem ser utilizados para atributos e métodos auxiliares

Modificadores de Acesso

	`public`	`private`
Atributos	Violam encapsulamento	Preservam encapsulamento
Métodos	Fornecem serviços para os clientes	Auxiliam outros métodos da classe

Modificadores de Acesso

- Membros de uma superclasse que recebem o modificador `private`, não podem ser acessados nem pelas subclasses
  - Se colocar `public` e o membro for um atributo, o princípio do encapsulamento é violado
- Membros com o modificador **`protected`** podem ser acessados pelas subclasses
- Um membro de uma classe com o modificador `protected` é visível nas subclasses e qualquer outra classe do mesmo pacote

Modificadores de Acesso

- Membros de uma classe, que não recebem modificador de acesso, tem visibilidade *default*, ou seja só podem ser acessados por classes do mesmo pacote
- Na redefinição de métodos herdados, o modificador de acesso não pode ser trocado por um mais restrito
  - No entanto, podem ser trocados por modificadores menos restritos

Além dos métodos e atributos vistos até agora...

Em Java também temos métodos e atributos **estáticos**

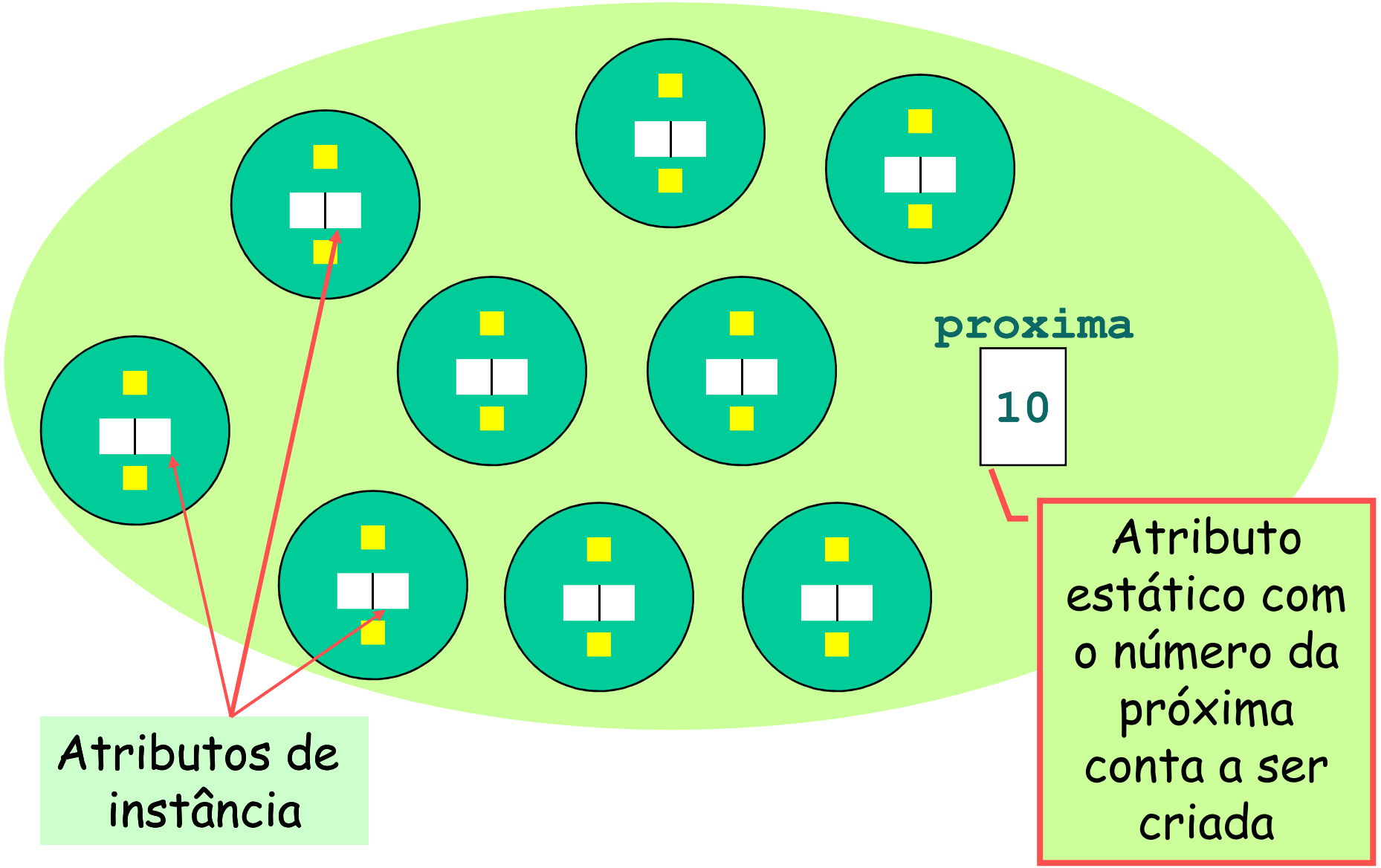
Atributos de instância

- São declarados dentro de uma classe, como visto até agora
- São inicializados quando o objeto é criado
- Cada objeto **(instância)** da classe tem espaço **individual** na memória para armazenar os valores destes atributos

Atributos estáticos

- São declarados dentro de uma classe, usando o **modificador `static`**
- São inicializados quando a classe é carregada para o ambiente de execução
- A classe, não cada um dos objetos, tem espaço para armazenar os valores destes atributos
- São **compartilhados** por todas as instâncias (objetos) de uma classe

Atributo estático de `Conta`



Usando atributos estáticos

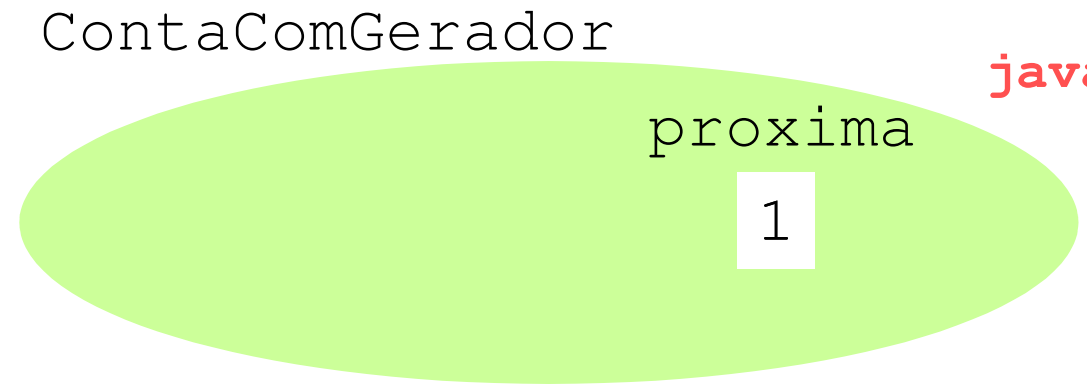
```
class ContaComGerador {
  private int numero;
  private double saldo;
  private static int proxima = 0;

  ContaComGerador(){
    numero = proxima;
    saldo = 0;
    proxima = proxima + 1;
  }
  ...
}
```

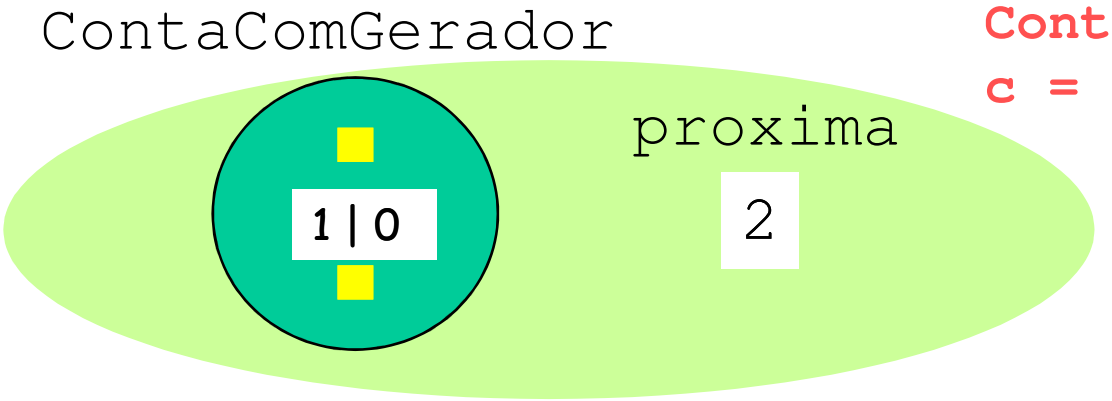
Inicialização no momento que a classe é carregada

Executado quando um objeto é criado

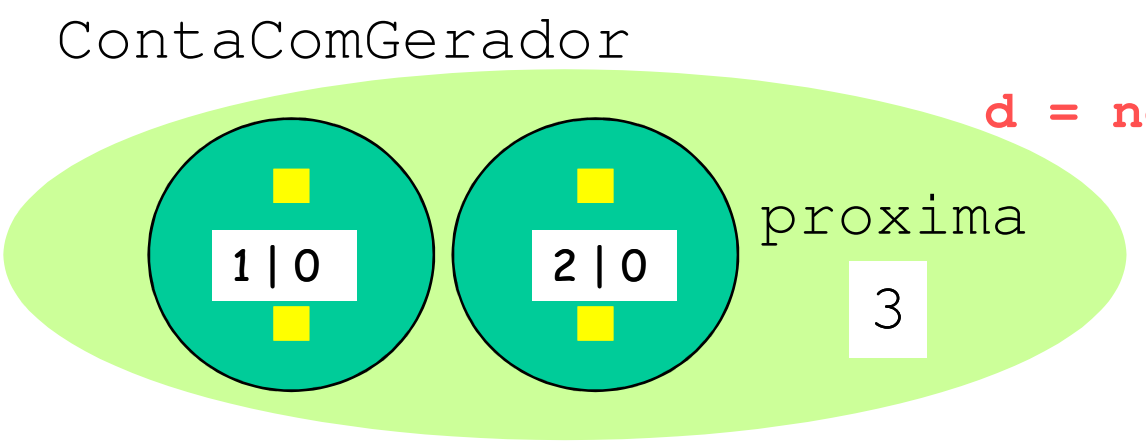
Na prática, não deve ser feito assim!




```
java Teste
```




```
ContaComGerador c, d;
c = new ContaComGerador();
```




```
d = new ContaComGerador();
```

Usando Atributos de Instância

```
class ContaComGerador {
  private int numero;
  private double saldo;
  private int proxima = 1;

  ContaComGerador(){
    numero = proxima;
    saldo = 0;
    proxima = proxima + 1;
  }
  ...
}
```

Inicialização no momento que um objeto é criado

Como **proxima** não foi declarado como estático, toda conta terá `numero` **1**

Métodos Estáticos

- Como os métodos de instância, são declarados dentro de uma classe, mas usando o **modificador `static`**
- Não podem ler nem modificar os valores dos atributos de instância
- Só podem ler ou modificar os valores dos atributos estáticos

Contrastando, um método de instância pode ler e modificar os valores dos atributos estáticos e de instância

Usando Métodos Estáticos

```
class ContaComGerador {
  private int numero;
  private double saldo;
  private static int proxima = 1;
  ContaComGerador(){
    numero = proxima;
    saldo = 0;
    proxima = proxima + 1;
  }
  static void setProxima(int n)  {
    proxima = n;
  }
  ...
}
```

Se o método só acessa atributos estáticos é um sinal que ele deve ser estático

Chamando Métodos Estáticos

```
ContaComGerador c;
c = new ContaComGerador();
c.setProxima(6);...
```

Via uma referência, como os métodos de instância

```
ContaComGerador.setProxima(6);
```

Usando o nome da classe, ao invés de uma referência para um objeto da classe

Herança e `static`

- Métodos estáticos são herdados, mas não os atributos
- Métodos estáticos não podem ser redefinidos
- Métodos estáticos podem ser escondidos (quase redefinidos…)
  - Definição da subclasse esconde a da superclasse
- Quase o mesmo que redefinição, mas temos static binding ao invés de dynamic binding
- Hiding, não overriding
  -

Herança e static

```
public class Conta{
   ...
   public static String getMensagem(){
      return "Eu sou uma conta";
   }
```

Assinaturas idênticas

```
public class Poupanca extends Conta{
   ...
   public static String getMensagem(){
      return "Eu sou uma poupança";
   }
```

```
...
Conta c = new Poupanca("210-3");
console.println(c.getMensagem());
Poupanca p = new Poupanca("211-3");
console.println(p.getMensagem());
...
```

O que será impresso nestas 2 linhas de comando?

Eu sou uma conta

Eu sou uma poupança

Quando Usar Métodos Estáticos?

- Quando comportamento do método não depender do estado de uma instância da classe
- Para construir classes utilitárias, onde não faz sentido ter que instanciar um objeto para usar os métodos
  - Ex: java.lang.Math, java.util.Collections, java.lang.System

Classes Paramétricas

- Não raras vezes, necessitamos implementar classes que possam armazenar uma grande quantidade de um mesmo tipo de informação
  - Ex: Coleções
- Coleções são estruturas que têm uma interface bem definida
  - inserir, remover, procurar, existe
- Coleção é uma estrutura genérica, mas sua implementação depende do tipo de informação armazenada
  - Para cada tipo de informação, uma nova implementação é requerida

Classes Paramétricas

- Muitas vezes a seqüência de comandos dentro dos métodos são similares entre uma implementação e outra, só mudando o tipo de informação manipulado
- O ideal é se pudéssemos ter uma só implementação e passar como parâmetro o tipo de informação armazenado, ou seja uma **classe paramétrica**
- Java dá suporte a classes paramétricas através de *generics*

Uso de Classes Paramétricas com *Generics*

- Maior facilidade de desenvolvimento
- Maior robustez
  - Cria classes type-safe
  - Evita o uso extensivo de type casts e instanceof

O mundo sem Generics...

```
public class Pilha {
    private Vector elements = new Vector();
    public Object pop() {
        Object result = null;
        result = this.elements.removeElementAt(0);
        return result;
    }
    public void push(Object obj) {
        this.elements.insertElementAt(obj, 0);
    }
```

Usando o tipo Pilha

```
public class PilhaString {
  private Pilha pilha = new Pilha();
  public String pop() {
    result = (String)this.pilha.pop();
    return result;
  }
  public void push(String p) {
    this.pilha.push(p);
  }
```

replicação de código

Uso do tipo

Uso obrigatório de casts, mesmo sabendo que só existem objetos do tipo String na pilha

E com generics...

```
public class Pilha<T> {
  private Vector<T> elements = new Vector<T>();
  public T pop() {
        return this.elements.removeElementAt(0);
  }
  public void push(T obj) {
    this.elements.insertElementAt(obj, 0);
  }
```

```
Pilha<String> pilhaString = new Pilha<String>();
```

Definindo tipos genéricos

- Para definir uma classe genérica, basta alterar o identificador da classe, para incluir o elemento de tipagem

```
modificadores class NomeDaClasse<especificadorDeTipo>
[extends SuperClasse ] implements [lista de interfaces]
{
    corpoDaClasse
}
```

Aqui fica o especificador de tipo

Genéricos e subtipos

- Veja este trecho de código:

```
List<String>  listString = new ArrayList<String>();
List<Object> listObject = listString;
```

Este código aparentemente está ok, mas com genéricos, isso não é verdade!

Atenção: Dizer que A estende B, não significa que List<A> extends List<B> !!!

Definindo curingas

- Imagine que você quer definir um método que aceite uma coleção de qualquer tipo de objetos

Java < 1.5

```
void printCollection(Collection c){
    for (int i = 0; i< c.size(); i++){
        Sys
    }
}
```

```
void printCollection(Collection<Object> c){
 for (int i = 0; i< c.size(); i++){
        System.out.println(c.get(i));
}
```

Java 1.5

Atenção: Collection<Object> não é o pai de todas as coleções!!!

Curingas

- A solução neste caso, seria colocar um "?" Indicando que você não conhece, a priori, o tipo da coleção.

```
void printCollection(Collection<?> c){
 for (int i = 0; i< c.size(); i++){
        System.out.println(c.get(i));
}
```

Potenciais problemas

```
class classGenerica<T> {


    void metodoGenerico(T obj){
         obj.iterator();
    }
}
```

E se o tipo T não definir um método iterator?

Pode-se definir restrições sobre os tipos!

```
class classGenerica<T extends Collection> {

    void metodoGenerico(T obj){
        obj.iterator();
    }
}
```

Agora, o tipo T tem de ser do tipo Collection