Universidade de São Paulo
Brasil

Aula 03 – Encapsulamento e Abstgração de Dados

MAC0321 - Laboratório de Programação Orientada a Objetos

Professor: Marcelo Finger **(mfinger@ime.usp.br)**

Departamento de Ciência da Computação
Instituto de Matemática e Estatística

Como garantir a integridade dos programas?

Tópicos

1. Classes, objetos, atributos e métodos
2. O modelo de memória Java
    - Tipos primitivos e objetos na memória
3. Referências para objetos
4. Controle de acesso a atributos e métodos
5. Variáveis e métodos de classe (*static*)
6. Sobrecarga de métodos
7. Construtores e inicialização
8. Finalização e coleta de lixo

Classes

Uma classe é um “molde” que usamos para criar objetos “semelhantes”

- Esses objetos são as “instâncias da classe”
- Todo objeto é uma instância de alguma classe

A classe define os atributos (campos) dos objetos

Ela define também as operações (métodos) dos objetos

Exemplo de Classe Só Com Campos

```
class ContaCorrente {
    public int numConta;
    public String titular;
    public double saldo;
}
```

Cada objeto ContaCorrente tem três campos: **`numConta`**, **`titular`** e **`saldo`**

Neste exemplo não temos encapsulamento!

- Como os três campos de uma **`ContaCorrente`** são públicos, eles podem ser acessados livremente
  - Nossa classe parece uma **`struct`** C ou um **`record`** Pascal
- Daqui a pouco vamos melhorar isto

Referências para Objetos

Para lidar com objetos temos que usar referências:

```
// Uma referência para uma ContaCorrente:
ContaCorrente cc;    // Ainda não se refere a uma conta

// Cria uma nova conta e inicializa a referência cc:
cc = new ContaCorrente();

// Agora que cc se refere à nova conta,
// podemos escrever nos seus campos públicos:
cc.numConta = 123;              // Inicializa o número
cc.titular = "Fulano de Tal";   // Inicializa o titular
cc.saldo = 5000.00;             // Inicializa o saldo
```

Podemos declarar e inicializar a referência ao mesmo tempo:

```
ContaCorrente cc = new ContaCorrente(); // recomendável
```

Construtores

```
class ContaCorrente {
    public int numConta;
    public String titular;
    public double saldo = 0;

    // Construtor:
    public ContaCorrente(int num, String tit) {
        numConta = num;
        titular = tit;
    }
}
```

Agora os campos **`numConta`** e **`titular`** são inicializados quando o objeto é criado:

```
ContaCorrente cc =
    new ContaCorrente(123, "Fulano de Tal");
```

Construtores (cont.)

```
class ContaCorrente {

    ... // tudo como no slide anterior

    public ContaCorrente(int num, String tit) {
        numConta = num;
        titular = tit;
    }
}
```

Com este construtor definido não podemos mais dizer

```
cc = new ContaCorrente(); // Erro de compilação
```

Somos obrigados a passar valores iniciais para os campos **`numConta`** e **`titular`** (isto é bom!):

```
cc = new ContaCorrente(1313, "Pato Donald");
```

O Construtor Default

Se você não definir nenhum construtor, Java gerará um construtor default, sem argumentos, que não faz nada

Ou seja, as duas classes abaixo são equivalentes:

```
class ContaCorrente {
    public int numConta;
    public String titular;
    public double saldo;
}
```

```
class ContaCorrente {
    public int numConta;
    public String titular;
    public double saldo;
    public ContaCorrente() {
        // não faz nada
    }
}
```

É por isso que podemos dizer

```
cc = new ContaCorrente();
```

quando não definimos construtor nenhum!

Sobrecarga de Construtores

Podemos ter mais de um construtor numa classe

```
class ContaCorrente {
    public int numConta;
    public String titular;
    public double saldo;

    public ContaCorrente(int num, String tit) {
        numConta = num;
        titular = tit;
        saldo = 0;
    }

    public ContaCorrente(int num, String tit, double sal) {
        numConta = num;
        titular = tit;
        saldo = sal;
    }
}
```

Sobrecarga de Construtores (cont.)

A lista de argumentos diferencia um construtor do outro:

```
// Cria conta com saldo zero
// (chama o primeiro construtor)
ContaCorrente cc1 = new ContaCorrente(1313, "Pato Donald");

// Cria conta com saldo 1000000000.00
// (chama o segundo construtor)
ContaCorrente cc2 = new ContaCorrente(6565,
                                      "William Gates",
                                      1000000000.00);
```

Uso de `this`

Para um construtor chamar outro construtor da mesma classe:

```
class ContaCorrente {
    public int numConta;
    public String titular;
    public double saldo;

    public ContaCorrente(int num, String tit, double sal) {
        numConta = num;
        titular = tit;
        saldo = sal;
    }

    public ContaCorrente(int num, String tit) {
        this(num, tit, 0); // Chama o outro construtor (preferido)
    }
}
```

O `this(...)` precisa ser o primeiro comando executável do construtor!

Outro uso de `this`

Para referenciar campos “escondidos” por argumentos com o mesmo nome:

```
class ContaCorrente {
    public int numConta;
    public String titular;
    public double saldo;

    public ContaCorrente(int numConta, String titular, double saldo) {
        this.numConta = numConta;
        this.titular = titular;
        this.saldo = saldo;
    }
```

Campos Estáticos (Variáveis de Classe)

Vamos fazer o número da conta ser gerado automaticamente pelo construtor:

```
class ContaCorrente {
    public int numConta;
    public String titular;
    public double saldo = 0;

   private static int proximoNumero = 0;

    public ContaCorrente(String titular) {
        numConta = proximoNumero++;
        this.titular = titular;
    }
}
```

Toda instância desta classe tem seus próprios campos **`numConta`**, **`titular`** e **`saldo`**

Mas o campo estático **`proximoNumero`** é **compartilhado** por todas as instâncias da classe

Como o **`proximoNumero`** só é usado dentro da classe ContaCorrente, ele foi declarado **`private`**

Os campos estáticos de Java parecem as variáveis globais de C

Métodos

Exemplo de classe com campos privados e métodos públicos que acessam esses campos:

```
class ContaCorrente {
    private int numConta;
    private String titular;
    private double saldo = 0;
    private static int proximoNumero = 0;
    public ContaCorrente(String titular) {
        numConta = proximoNumero++;
        this.titular = titular;
    }

    // Métodos:
    public int pegaNumero() {
        return numConta;
    }
    public String pegaTitular() {
        return titular;
    }
```

Métodos (cont.)

```
    public double pegaSaldo() {
        return saldo;
    }
    public void deposito(double valor) {
        saldo = saldo + valor;
    }
    public void saque(double valor) {
        if (saldo >= valor)
            saldo = saldo - valor;
        else
            // Isto é provisório (pode ser melhorado)
            throw new RuntimeException("saldo insuficiente");
    }
    public void imprime() {
        System.out.println("numero: " + numConta);
        System.out.println("titular: " + titular);
        System.out.println("saldo: " + saldo);
    }
}
```

Chamadas a Métodos

Com nossa nova classe ContaCorrente podemos dizer

```
ContaCorrente cc = new ContaCorrente("Fulano");
cc.deposito(5000.00);
cc.imprime();
cc.saque(2550.50);
System.out.println("Novo saldo: " + cc.pegaSaldo());
```

Agora temos encapsulamento

Como os campos da conta são privados, eles só podem ser acessados por métodos da classe **`ContaCorrente`**

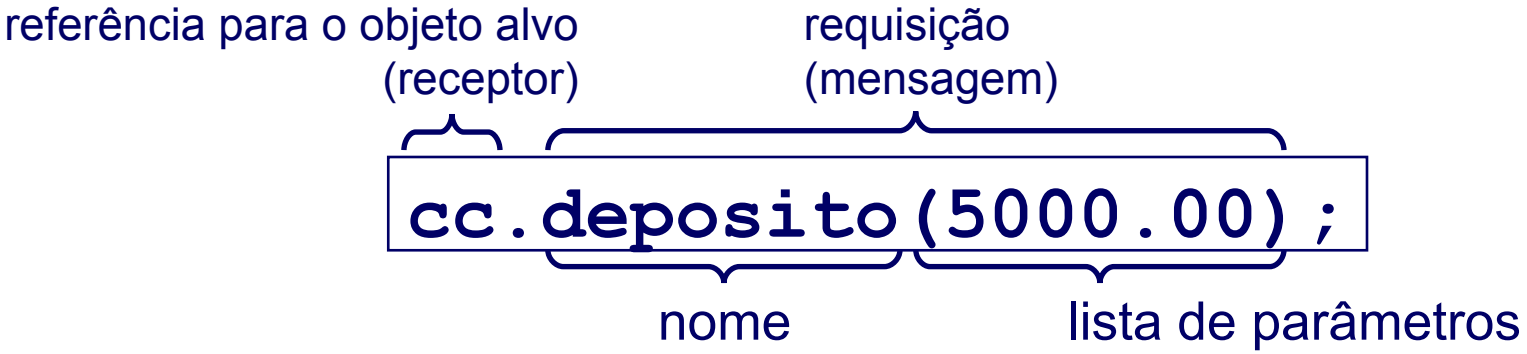
```
// Fora da classe ContaCorrente:
cc.saldo = 1000000;                // Erro de compilação
System.out.println(cc.titular); // Erro de compilação
```

Chamadas a Métodos (cont.)

Cada chamada a método é feita sobre um objeto

Esse objeto é o objeto alvo da chamada

Chamadas a Métodos (cont.)



Dentro do método, o identificador **`this`** é uma referência para o objeto alvo da chamada

```
void deposito(int valor) {
    this.saldo = this.saldo + valor;
}
```

O **`this`** pode ser omitido (geralmente é)

- Obrigatório quando algum campo é "escondido" por um argumento com o mesmo nome

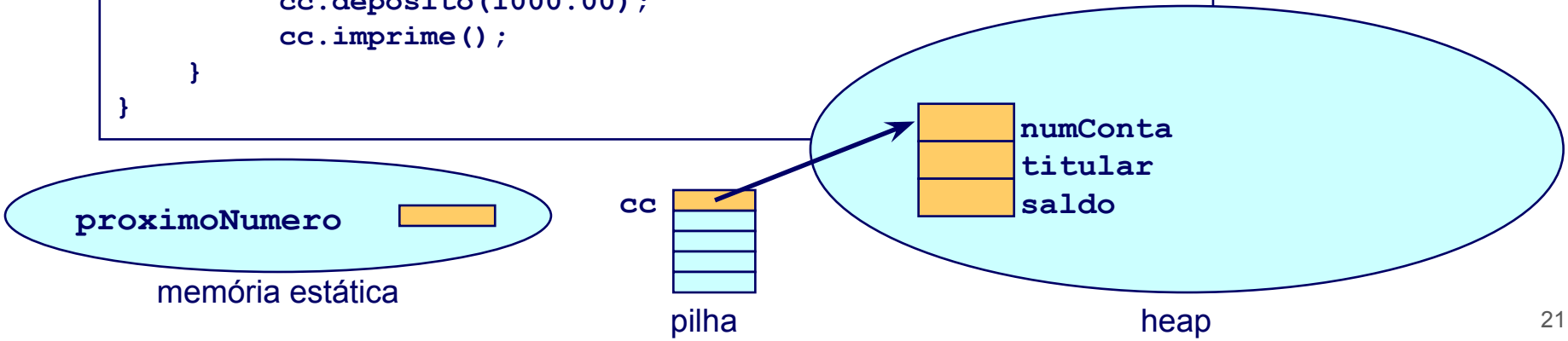
Onde Ficam os Dados e os Objetos?

- Na pilha: variáveis locais (declaradas dentro de métodos)
  - Tempo de vida: o tempo da execução do método
  - A variável é alocada na entrada do método e desalocada quando o método retorna
- Em memória estática: campos estáticos
  - Tempo de vida: o tempo da execução do programa
  - A memória é alocada quando o programa começa a rodar e fica reservada enquanto ele não acabar
- No heap: todos os objetos Java
  - Tempo de vida: controlado pelo programa
  - A memória é alocada quando o programa cria um objeto (com **`new`**) fica reservada enquanto houver alguma referência para o objeto
  - Objetos não referenciados podem ser destruídos pelo coletor de lixo a qualquer momento

A Referência Não É o Objeto

- A referência pode ser uma variável local (na pilha), um campo estático (em memória estática) ou um campo de outro objeto (no heap)
- O objeto está sempre no heap

```
class Teste {
    public static void main(String args[]) {
        ContaCorrente cc;
        cc = new ContaCorrente("Fulano");
        cc.deposito(1000.00);
        cc.imprime();
    }
}
```



O Modelo de Memória Java

Tipos primitivos e referências para objetos podem estar

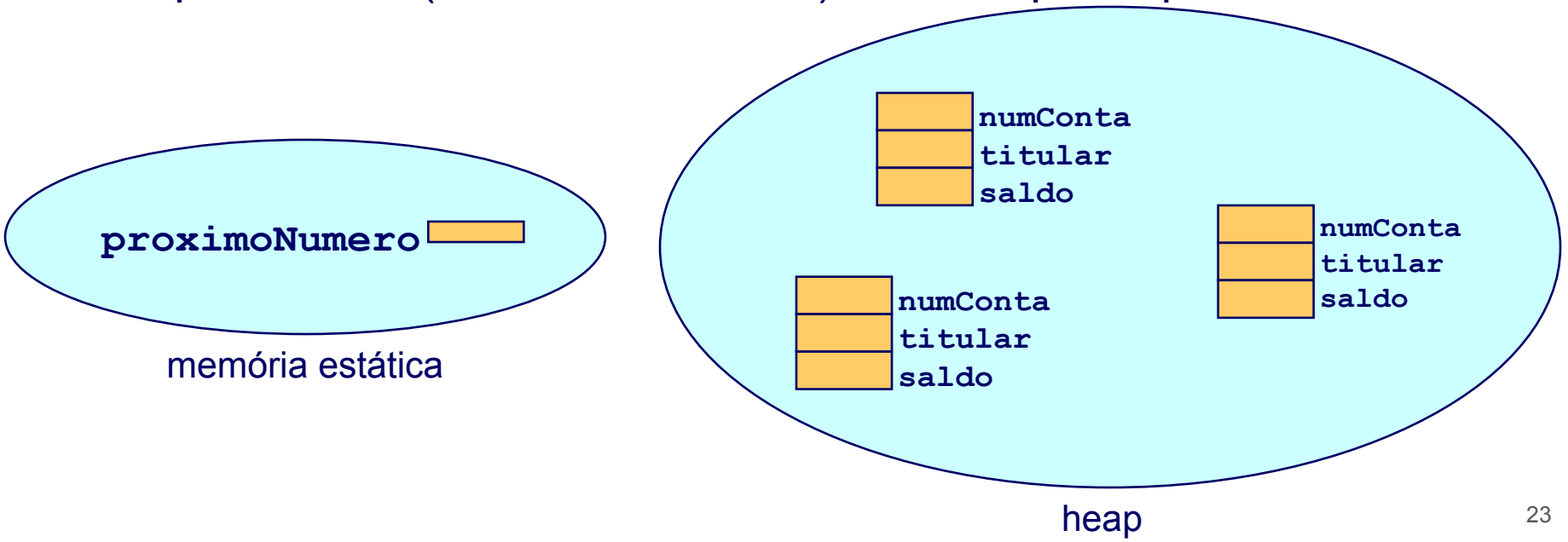
- Na pilha (se forem variáveis locais)
- Na memória estática (se forem campos estáticos)
- No heap (se forem campos "normais" de objetos)

Objetos só podem estar no heap!

Caso contrário o coletor de lixo não irá encontrá-los

Campos "Normais" x Campos Estáticos

- Campo "normal" (variável de instância): cada objeto (instância da sua classe) tem seu próprio campo
- Campo estático (variável de classe): um campo só para toda a classe



Acesso a Campos Públicos

- Campo "normal" (não estático)
  - Se **`saldo`** fosse público, de fora da classe **`ContaCorrente`** poderíamos acessá-lo assim:

```
ContaCorrente cc1 = new ContaCorrente("Fulano");
ContaCorrente cc2 = new ContaCorrente("Sicrano");
cc1.saldo = 5000.00                // Só se saldo fosse público
System.out.println(cc2.saldo);     // Idem
```

Antes do ponto aparece uma referência para um objeto

- Campo estático
  - Se **`proximoNumero`** fosse público, de fora da classe **`ContaCorrente`** poderíamos acessá-lo assim:

```
System.out.println(ContaCorrente.proximoNumero);
```

Antes do ponto podemos usar o nome da Classe

- Campos públicos estragam o encapsulamento!
  - Mesmo assim temos que saber como acessá-los…

Métodos Estáticos (Métodos de Classe)

- Não são chamados sobre instâncias da classe, mas sobre a própria classe
- Em outras palavras: a chamada ao método não tem um objeto alvo
  - O método estático não pode usar o identificador **`this`** (nem mesmo implicitamente)
- Um método estático não pode acessar campos não estáticos
  - Porque **`campo`** é o mesmo que **`this.campo`**
- Ele só pode acessar campos estáticos
- A execução de uma aplicação Java começa sempre por um método estático chamado **`main`**

Método Estático: Definição e Chamada

- Exemplo de definição

```
class ContaCorrente {
    ...
    private static int proximoNumero = 0;
    ...
    public static int pegaProximoNumero() {
        return proximoNumero;
    }
    ...
}
```

O método estático só acessa campos estáticos

- Exemplo de chamada

```
System.out.println(ContaCorrente.pegaProximoNumero());
```

Antes do ponto podemos usar o **nome da classe**

Classes Com Tudo Estático

- Todos os campos e métodos são estáticos
- Essa classe não é um “molde” para criar objetos “semelhantes”
  - Para que criar instâncias dessa classe, se as instâncias não tem campos!
- Uma classe com tudo estático é como um módulo em C
  - Os métodos estáticos são as funções do módulo
  - Os campos estáticos são as variáveis globais
- Ela é útil quando não precisamos de objetos

Sobrecarga de Métodos

- Uma classe pode ter mais de um método com o mesmo nome
- A lista de argumentos diferencia um do outro
- Exemplo (da classe **`java.lang.String`**):

```
/** Primeira posição com ch */
public int indexOf(char ch) { ... }

/** Primeira posição com ch a partir de start */
public int indexOf(char ch, int start) { ... }

/** Primeira posição com str */
public int indexOf(String str) { ... }

/** Primeira posição com str a partir de start */
public int indexOf(String str, int start) { ... }
```

Sobrecarga de Métodos (cont.)

- Exemplos de chamadas a métodos sobrecarregados:

```
String s = "ABCDEFGHIAB";
String s1 = "DEF";
System.out.println(s.indexOf('A'));      // imprime 0
System.out.println(s.indexOf('A', 4));   // imprime 9
System.out.println(s.indexOf(s1));       // imprime 3
System.out.println(s.indexOf(s1, 4));    // imprime -1
System.out.println(s.indexOf("AB", 4)); // imprime 9
```

- Métodos de instância (não estáticos) e de classe (estáticos) podem ser sobrecarregados

Inicialização de Campos Não-Estáticos

- Junto com a declaração do campo:

```
class ContaCorrente {
    ...
    private double saldo = 0;
    ...
}
```

- Num construtor:

```
class ContaCorrente {
    ...
    private String titular;
    ...
    ContaCorrente(String titular) {
        this.titular = titular;
        ...
    }
    ...
}
```

Campos (estáticos ou não) que não foram inicializados explicitamente ficam com valores default:

- `boolean: false`
- `char` : `'\u0000'`
- números: `0` ou `0.0`
- referências para objetos: `null`

Inicializa Campos Estáticos

Junto com a declaração do campo:

```
class ContaCorrente {
    ...
    private static int proximoNumero = 0;
    ...
}
```

A inicialização destes campos só ocorre quando a classe for utilizada pela 1a vez!

Num bloco de inicialização estático:

```
class Primes {
    protected static int[] knownPrimes = new int[10000];

    static {
        knownPrimes[0] = 2;
        for (int i = 1; i < knownPrimes.length; i++)
            knownPrimes[i] = nextPrime();
    }
    private static int nextPrime() { /* ... */ }
}
```

Inicialização de Variáveis Locais

- Variáveis locais podem ser declaradas
  - dentro de métodos (estáticos ou não)
  - dentro de construtores
  - dentro de blocos de inicialização estáticos
- Elas precisam ser inicializadas explicitamente
- Se você deixar de inicializar uma variável local, ela não vai ficar com um valor default (padrão)
  - Campos tem valores default, mas variáveis locais não!
- Percebendo que alguma variável local não foi inicializada, o compilador reclama!

Resumo

- Construtores
  - podem existir vários, conforme os parâmetros
  - existe um padrão se nenhum for definido
- keyword public (private)
  - métodos e variáveis acessíveis a todos (apenas a classe)
- keyword static
  - variável compartilhada entre todos os objetos de uma classe
  - método sem objeto associado
  - trecho de código executado na primeira referência a classe
- keyword this
  - referência ao objeto em questão

O Método finalize()

- Você pode definir um método **`finalize()`** numa classe qualquer:

```
class Exemplo {
    ...
    protected void finalize() throws Throwable {
        ...
        super.finalize(); // esta linha é importante!
    }
}
```

- O **`finalize()`** será chamado antes que o coletor de lixo destrua um objeto não referenciado
- Na prática você não pode depender dele para liberar recursos escassos
  - Não se sabe quando o coletor de lixo vai destruir o objeto!
- Se for o caso, crie um outro método, que os clientes do objeto devem chamar para liberar recursos escassos
  - Nomes populares para esse método: **`dispose`**, **`release`**, **`close`**

Exemplo de Uso de finalize()

```
class ContaCorrenteLogada {
    ...
    private Stream logFile; // Para logar a movimentação da conta
    ...
    public ContaCorrenteLogada(String titular) {
        ...
        logFile = new FileOutputStream("log" + numConta);
    }
    public void dispose() { // Deve ser chamado pelos clientes!
        if (logFile != null) {
            logFile.close();
            logFile = null;
        }
    }
    protected void finalize() throws Throwable {
        dispose();
        super.finalize();
    }
}
```

Lista de exercícios

No computador com o Eclipse

Entrega até o final do dia

MAC321
Lab POO

- Professor: Marcelo Finger

  E-mail: mfinger@ime.usp.br