

Objetos e Classes

Roland Teodorowitsch

Fundamentos de Programação - Escola Politécnica - PUCRS

8 de junho de 2023

Introdução

Objetivos

- Entender os conceitos de classes, objetos e encapsulamento
- Implementar variáveis, métodos e construtores de instância
- Ser capaz de projetar, implementar e testar classes
- Entender o compartimento de referências a objetos, variáveis estáticas e métodos estáticos

Conteúdos

- Programação Orientada a Objetos
- Implementando uma Classe Simples
- Construtores
- Exemplos
- Passos para Implementar uma Classe
- Testando uma Classe
- Padrões para Dados de Objetos
- Referências a Objetos
- Variáveis e Métodos Estáticos
- Sumário
- Tópicos Complementares

Programação Orientada a Objetos

Programação Orientada a Objetos

- Até agora foram apresentadas técnicas de programação estruturada
 - Quebrar tarefas em subtarefas
 - Escrever métodos reusáveis para tratar tarefas
- A partir de agora serão estudados objetos e classes
 - Para construir programas maiores e mais complexos
 - Para modelar objetos que são usados no mundo real

Classes e Objetos

Uma classe descreve objetos com um comportamento comum. Por exemplo, a classe Carro descreve todos os veículos de passageiros que tem determinada capacidade e formato.

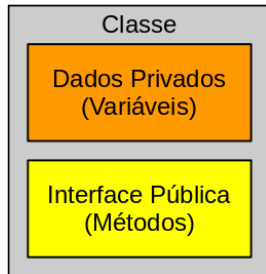
Objetos e Programas

- Programas Java são feitos por objetos que interagem uns com os outros
 - Cada objeto é baseado em uma classe
 - Uma classe descreve um conjunto de objetos o mesmo comportamento
- Cada classe define um conjunto específico de métodos para ser usado com os seus objetos
 - Por exemplo, a classe `String` provê métodos tais como `length()` e `charAt()`
 - Estes métodos foram definidos na classe `String` e podem ser usados por qualquer objeto desta classe

```
String boasVindas = "Sejam bem-vindos!";  
int tamanho = boasVindas.length();  
char caract1 = boasVindas.charAt(0);
```

Diagrama de Classes

- Dados Privados
 - Cada objeto tem seus próprios dados privados que outros objetos não podem acessar diretamente
 - Métodos da interface pública provêm acesso a dados privados, enquanto escondem detalhes de implementação
 - Isto é chamado de **encapsulamento**
- Interface Pública
 - Cada objeto tem um conjunto de métodos disponível para ser usado por outros objetos



Tipos Abstratos de Dados

- **Abstração** é uma visão ou representação de uma entidade que inclui apenas os seus atributos mais importantes segundo determinado ponto de vista
 - Em Computação, usa-se a abstração para atenuar a complexidade de problemas
- Um **Tipo Abstrato de Dados** (TAD) é uma estrutura sintática que define um tipo para determinada entidade, de forma que quem o usa não necessite conhecer os detalhes da sua implementação (armazenamento interno de dados ou implementação de operações suportadas)
 - TADs são importantes para garantir encapsulamento
- **Encapsulamento** é uma técnica que agrupa elementos relacionados entre si (tipos, variáveis, métodos, etc.) em um módulo, escondendo do usuário seus detalhes internos, o que garante abstração
 - O encapsulamento define quais partes de um objeto serão visíveis (públicas) e quais partes permanecerão ocultas (privadas)
- Em Java, classes são usadas para a criação de Tipos Abstratos de Dados

Implementando uma Classe Simples

Implementando uma Classe Simples

- Exemplo: contador

Uma classe que modela um dispositivo mecânico que é usado para realizar contagens

- Por exemplo, para contar quantas pessoas estão assistindo a um concerto ou quantas pessoas embarcaram em um ônibus



- O que deve ser feito?

- Inicializar o contador (Java já faz isso automaticamente...)
- Incrementar o dispositivo
- Obter o valor atual

Classe Contador

- Especifica-se variáveis de instância na declaração da classe:

Variáveis de instância
sempre deveriam ser
privadas!

```
public class Contador {  
    private int valor;  
    // ...  
}
```

Cada objeto desta classe
tem uma cópia distinta
desta variável de instância.

Tipo da variável de instância.

- Cada objeto instanciado a partir desta classe terá seu próprio conjunto de variáveis de instância
 - Cada objeto da classe `Contador` terá sua própria variável `valor`
- Especificadores de acesso
 - Classes (e métodos de interface) são públicos (`public`)
 - Variáveis de instância são privadas (`private`)

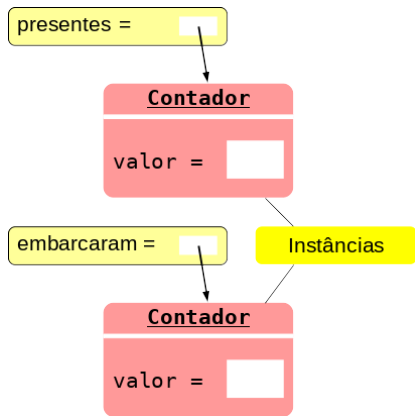
Instanciando Objetos

- Objetos são criados a partir de classes
 - Usa-se o operador `new` para construir objetos
 - Cada objeto recebe um nome único (da mesma forma que uma variável)
- O operador `new` já apareceu em exemplos anteriores

```
Scanner in = new Scanner(System.in);
```

- Para criar duas instâncias de objetos da classe `Contador`, usa-se:

```
// NomeClasse nomeObjeto = new NomeClasse();  
Contador presentes = new Contador();  
Contador embarcaram = new Contador();
```



Métodos da Classe Contador

- Dois métodos serão usados para acessar as variáveis de instância dos objetos da classe Contador
 - `incrementaValor()`: incrementa o valor da variável de instância `valor`
 - `obtemValor()`: retorna o valor da variável de instância `valor`
- Para usar estes métodos, é preciso especificar sobre qual objeto eles deverão ser aplicados

```
presentes.incrementaValor();  
embarcaram.incrementaValor();
```

```
/** Classe para contagem de eventos,  
    simulando um dispositivo de contagem.  
    @version 0.0 */  
public class Contador {  
    private int valor;  
  
    public int obtemValor() {  
        return valor;  
    }  
  
    public void incrementaValor() {  
        valor++;  
    }  
}
```

Tipos de Métodos

1 Métodos de Acesso (*Accessors* ou *getters*)

- Solicitam uma informação ao objeto sem alterá-lo
- Normalmente retornam algum valor
- Em inglês costumam iniciar com o prefixo `get`; em Português, obtem

```
public int obtemValor() { return valor }
```

2 Métodos de Alteração (*Mutators* ou *setters*)

- Alteram valores no objeto
- Geralmente recebem um parâmetro que será usado para alterar uma variável de instância
- Normalmente o tipo de retorno é `void`
- Em inglês costumam iniciar com o prefixo `set`; em Português, define

```
public void incrementaValor() { ++valor; }  
public void defineValor(int v) { valor = v; }
```

Métodos Estáticos x Não-Estáticos

- Quando um método (ou membro) é declarado como `static`, ele existe e pode ser acessado mesmo se nenhum objeto da classe for criado (lembre-se da classe `Math`)
- Para métodos de instância (não-estáticos), é preciso instanciar um objeto da classe antes que o método possa ser invocado (lembre-se da classe `Scanner`)
- Somente depois de criar um objeto, é possível invocar os seus métodos não-estáticos
- Métodos estáticos **SOMENTE** podem invocar métodos estáticos
- Métodos de instância podem acessar métodos estáticos

```
Contador presentes = new Contador(); // Cria o objeto
presentes.incrementaValor(); // Invoca um de seus metodos
```


Construtores

Construtores

- Um construtor é um método que inicializa as variáveis de instância de um objeto
 - Ele é automaticamente chamado quando um objeto é criado
 - Ele tem exatamente o mesmo nome da classe
- Construtores nunca retornam valores, mas **não se usa void na sua declaração**

```
/** Classe para contagem de eventos, simulando um dispositivo de contagem.
    @version 1.0 */
public class Contador {
    private int valor;

    /** Construtor que inicializa o valor com 0. */
    public Contador() {           // Faz exatamente o que o construtor padrão
        valor = 0;                // gerado pelo compilador faria.
    }

    public int obtenValor() { return valor; }
    public void incrementaValor() { valor++; }
}
```

Múltiplos Construtores (Sobrecarga)

- Uma classe pode ter mais de um construtor, mas cada um tem que ter um conjunto único de parâmetros

```
/** Classe para contagem de eventos, simulando um dispositivo de contagem.
    @version 2.0 */
public class Contador {
    private int valor;

    /** Construtor que inicializa o valor com 0. */
    public Contador() { valor = 0; }

    /** Construtor que inicializa o valor com um valor específico.
        @param v Valor inicial do contador que será criado. */
    public Contador(int v) { valor = v; }

    public int obterValor() { return valor; }
    public void incrementaValor() { valor++; }
}
```

- O compilador seleciona o construtor que corresponde aos parâmetros especificados na construção

```
Contador presentes = new Contador(10);
Contador embarcaram = new Contador();
```

Sintaxe de Construtores

- Um construtor é invocado quando um objeto é criado com a palavra-reservada `new`

Um construtor NÃO tem tipo de retorno, nem mesmo `void`.

```
public class Contador {  
    private int valor;
```

```
    public Contador() {  
        valor = 0;  
    }
```

Um construtor tem o mesmo nome da classe.

```
    public Contador(int v) {  
        valor = v;  
    }
```

Este construtor será utilizado quando for executado `new Contador(10)`

```
    // ...
```

```
}
```

O Construtor Padrão

- Se nenhum construtor for declarado, o compilador criará um construtor padrão automaticamente
 - Ele não receberá nenhum parâmetro
 - Ele inicializará todas as variáveis de instância
 - Números são inicializados com 0, booleanos com `false` e objetos com `null`

```
/** Classe para contagem de eventos, simulando um dispositivo de contagem.
    @version 1.0 */
public class Contador {
    private int valor;

    /** Construtor que inicializa o valor com 0. */
    public Contador() {           // Faz exatamente o que o construtor padrão
        valor = 0;                // gerado pelo compilador faria.
    }

    public int obterValor() { return valor; }
    public void incrementaValor() { valor++; }
}
```

Exemplos

Classe com main(): Contador.java

```
/** Classe para contagem de eventos, simulando um dispositivo de contagem. */  
public class Contador {  
    private int valor;  
    public Contador() { valor = 0; }  
    public Contador(int v) { valor = v; }  
    public int obterValor() { return valor; }  
    public void definevalor(int v) { valor = v; }  
    public void zeraValor() { valor = 0; }  
    public void incrementaValor() { valor++; }  
    public static String info() { return "Contador - Versão 1.0"; }  
  
    /** Metodo inicial.  
     * @param args Argumentos da linha de comandos (NÃO utilizado). */  
    public static void main(String[] args) {  
        System.out.println( Contador.info() );  
        Contador c1 = new Contador();  
        System.out.println( "c1: " + c1.obterValor() );  
        c1.incrementaValor();  
        System.out.println( "c1: " + c1.obterValor() );  
        Contador c2 = new Contador(100);  
        for (int i=0; i<11; ++i) c2.incrementaValor();  
        System.out.println( "c2: " + c2.obterValor() );  
        Contador c3 = c2;  
        System.out.println( "c3: " + c3.obterValor() );  
    }  
}
```

Duas Classes no mesmo Arquivo: TestaContador.java

```
/** Classe para contagem de eventos, simulando um dispositivo de contagem. */
class Contador {
    private int valor;
    public Contador() { valor = 0; }
    public Contador(int v) { valor = v; }
    public int obtemValor() { return valor; }
    public void definevalor(int v) { valor = v; }
    public void zeraValor() { valor = 0; }
    public void incrementaValor() { valor++; }
    public static String info() { return "Contador - Versão 1.0"; }
}

/** Classe para testar a classe Contador. */
public class TestaContador {
    /** Metodo inicial.
     * @param args Argumentos da linha de comandos (NÃO utilizado). */
    public static void main(String[] args) {
        System.out.println( Contador.info() );
        Contador c1 = new Contador(); System.out.println( "c1: " + c1.obtemValor() );
        c1.incrementaValor(); System.out.println( "c1: " + c1.obtemValor() );
        Contador c2 = new Contador(100);
        for (int i=0; i<11; ++i) c2.incrementaValor();
        System.out.println( "c2: " + c2.obtemValor() );
        Contador c3 = c2;
        System.out.println( "c3: " + c3.obtemValor() );
    }
}
```


Classes em Arquivos Separados: Contador.java

```
/** Classe para contagem de eventos, simulando um dispositivo de contagem. */  
public class Contador {  
    private int valor;  
    public Contador() { valor = 0; }  
    public Contador(int v) { valor = v; }  
    public int obterValor() { return valor; }  
    public void definevalor(int v) { valor = v; }  
    public void zeraValor() { valor = 0; }  
    public void incrementaValor() { valor++; }  
    public static String info() { return "Contador - Versão 1.0"; }  
}
```

Classes em Arquivos Separados: TestaContador.java

```
/** Classe para testar a classe Contador. */  
public class TestaContador {  
    /** Metodo inicial.  
        @param args Argumentos da linha de comandos (NÃO utilizado). */  
    public static void main(String[] args) {  
        System.out.println( Contador.info() );  
        Contador c1 = new Contador();  
        System.out.println( "c1: " + c1.obtemValor() );  
        c1.incrementaValor();  
        System.out.println( "c1: " + c1.obtemValor() );  
        Contador c2 = new Contador(100);  
        for (int i=0; i<11; ++i) c2.incrementaValor();  
        System.out.println( "c2: " + c2.obtemValor() );  
        Contador c3 = c2;  
        System.out.println( "c3: " + c3.obtemValor() );  
    }  
}
```

Passos para Implementar uma Classe

Passos para Implementar uma Classe

- 1 Crie uma lista informal de tarefas para os objetos: adicionar, obter, limpar, etc.
- 2 Especifique a interface pública (por exemplo, para uma caixa registradora)

```
void adicionaItem(double preco);    int obtemNumItems();    double obtemTotal();    void limpa();
```

- 3 Documente a interface pública com comentários Javadoc

```
/** Adiciona um item na caixa registradora.  
 * @param preco Preço do item a ser registrado. */
```

- 4 Determine as variáveis de instância

```
private int numItens;                private double total;
```

- 5 Implemente os construtores e métodos

```
public void adicionaItem(double preco) { numItens++; total = total + preco; }
```

- 6 Teste a classe

Interface Pública de uma Classe

- Quando se projeta uma classe, um dos primeiros passos é especificar a sua **interface pública**
- Por exemplo: uma classe para uma caixa registradora
 - Quais tarefas esta classe deverá executar?
 - Que métodos serão necessários?
 - Que parâmetros cada método receberá?
 - O que os métodos retornarão?

Tarefa	Método	Retorno
Adiciona o preço de um item	<code>adicionaItem(double)</code>	<code>void</code>
Obtém o total devido	<code>obtemTotal()</code>	<code>double</code>
Obtém o número de itens comprados	<code>obtemNumItens()</code>	<code>int</code>
Limpa o registro da caixa registradora para uma nova venda	<code>limpa()</code>	<code>void</code>

Escrevendo a Interface Pública de uma Classe

- É importante usar comentários no estilo Javadoc para documentar a classe e o funcionamento de cada método
- As declarações de métodos correspondem à interface pública da classe
- Os dados e o corpo dos métodos correspondem à implementação privada da classe

```
/** Simula uma caixa registradora, com número de itens e valor total dos itens. */  
public class CaixaRegistradora {  
  
    /** Adiciona um item na caixa registradora.  
     * @param preco Preço do item a ser registrado. */  
    public void adicionaItem(double preco) {  
        numItens++;  
        total = total + preco;  
    }  
  
    /** Obtém o valor total de todos os itens registrados.  
     * @return Valor total de todos os itens registrados. */  
    public double obtemTotal() {  
        return total;  
    }  
  
    // ...  
}
```

Javadoc

- O utilitário javadoc gera um conjunto de arquivos HTML a partir dos comentários no estilo Javadoc
- Parâmetros e retornos de métodos devem ser descritos com as anotações `@param` e `@return`

```
javadoc CaixaRegistradora.java
```

Constructors

Constructor	Description
<code>CaixaRegistradora()</code>	Constrói um objeto sem qualquer item registrado.

Method Summary

All Methods			Static Methods	Instance Methods	Concrete Methods
Modifier and Type	Method	Description			
void	<code>adicionarItem(double preco)</code>	Adiciona um item na caixa registradora.			
static String	<code>info()</code>	Método estático que retorna informações sobre a classe.			
void	<code>limpa()</code>	Limpa a caixa registradora para iniciar uma nova venda.			
int	<code>obtemNumItens()</code>	Obtém o número de itens registrados.			
double	<code>obtemTotal()</code>	Obtém o valor total de todos os itens registrados.			

Projetando a Representação de Dados

- Um objeto armazena dados em variáveis de instância
 - Variáveis de instância são declaradas dentro da classe e devem ser privadas

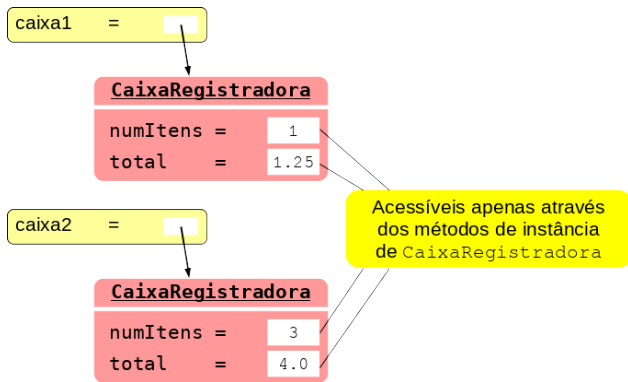
```
/** Simula uma caixa registradora, com número de itens e valor total dos itens. */
public class CaixaRegistradora {
    private int numItems;
    private double total;
    // ...
}
```

- Todos os métodos (não estáticos) dentro da classe têm acesso a elas, podendo modificar os seus valores
- Quais dados os métodos da classe da caixa registradora necessitam?

Tarefa	Método	Dado(s) necessário(s)
Adiciona o preço de um item	adicionaItem()	total, numItems
Obtém o total devido	obtemTotal()	total
Obtém o número de itens comprados	obtemNumItems()	numItems
Limpa o registro da caixa registradora para uma nova venda	limpa()	total, numItems

Variáveis de Instância de Objetos

- Cada objeto de uma classe tem um conjunto exclusivo de variáveis de instância
- Os valores armazenados nas variáveis de instância constituem o estado do objeto



Acessando Variáveis de Instância

- Variáveis de instância privadas (`private`) não podem ser acessadas de fora da classe: o compilador não permite esta violação de privacidade

```
public static void main(String[] args) {  
    CaixaRegistradora caixa1 = new CaixaRegistradora();  
    System.out.println( caixa1.numItens ); // ERRO  
}
```

- Em vez disto, usam-se métodos para acessar os dados da classe: o encapsulamento provê uma interface pública e esconde os detalhes de implementação

```
public static void main(String[] args) {  
    CaixaRegistradora caixa1 = new CaixaRegistradora();  
    System.out.println( caixa1.obtemNumItens() ); // OK  
}
```

Implementando Métodos de Instância

- Métodos de instância acessam variáveis de instância privadas

```
public void adicionaItem(double preco) {  
    numItens++;  
    total = total + preco;  
}
```

- Métodos de instância
 - São declarados dentro da classe como `public`
 - Não há necessidade de especificar o nome do objeto (parâmetro implícito) quando se usa alguma variável de instância dentro de uma classe
 - Os parâmetros explícitos (variáveis paramétricas) são listados na declaração do método

Sintaxe de Métodos de Instância

```
public class CaixaRegistradora {  
    private int numItens;  
    private double total;
```

Parâmetro EXPLÍCITO

Variáveis de
instância do
parâmetro
IMPLÍCITO

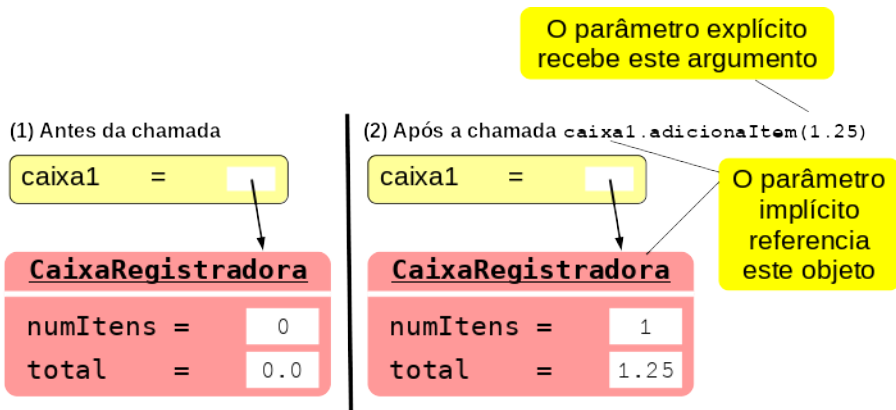
```
    public void adicionaItem(double preco) {  
        numItens++;  
        total = total + preco;  
    }
```

```
    // ...
```

```
}
```

Parâmetros Implícitos e Explícitos

- Quando um item é adicionado, isto afeta as variáveis de instância do objeto sobre o qual o método é invocado



Erros Comuns (1)

- Não inicializar referências a objetos na construção
 - Referências são inicializadas por padrão com `null`
 - Chamar um método de uma referência que contém `null` resulta em um erro de execução: `NullPointerException`
 - O compilador consegue apenas detectar variáveis locais não inicializadas, gerando um erro de compilação

```
public class ErrosComuns {  
    private String nome; // O construtor default inicializará nome com null  
  
    public void mostraNomes() {  
        String nomeLocal;  
        // Erro de execução: java.lang.NullPointerException  
        System.out.println( nome.length() );  
        // Erro de compilação: a variável nomeLocal pode NÃO ter sido inicializada  
        System.out.println( nomeLocal.length() );  
    }  
}
```

Erros Comuns (2)

- Tentar chamar um construtor

- Não se pode chamar um construtor como é feito com outros métodos
- Ele é “invocado” automaticamente pela palavra reservada `new`

```
CaixaRegistradora caixa = new CaixaRegistradora();
```

- Não se pode invocar um construtor para um objeto que já existe

```
caixa.CaixaRegistradora(); // ERRO!
```

- Mas pode-se criar um novo objeto usando uma referência existente

```
CaixaRegistradora caixa = new CaixaRegistradora();  
caixa.adicionaItem(1.25);  
caixa = new CaixaRegistradora();
```

Erros Comuns (3)

- Declarar um construtor como `void`
 - Construtores não tem tipo de retorno
 - Isto cria um método com um tipo de retorno `void` que NÃO é um construtor!
 - O compilador Java não considera isto um erro...

```
public class CaixaRegistradora {  
    // ...  
  
    /** Pretendia-se criar um construtor... */  
    public void CaixaRegistradora() {  
        // ...  
    }  
}
```


Sobrecarga (*Overloading*)

- Pode-se criar múltiplos construtores para uma classe
- Cada um deles tem o mesmo nome, mas possui uma lista de parâmetros diferente
- Isto se chama **sobrecarga** e pode ser aplicado a qualquer método em Java
 - Sobrecarga = mesmo nome de método com parâmetros diferentes

```
void imprima(CaixaRegistradora caixa) { ... }  
void imprima(ContaBancaria conta)      { ... }  
void imprima(int valor)                 { ... }  
void imprima(double valor)              { ... }
```

CaixaRegistradora.java

```

/** Simula uma caixa registradora, com número de itens e valor total dos itens. Adaptado de HORSTMANN (2013, p. 377).
    @version 2.0 */
public class CaixaRegistradora {
    private int numItens;          private double total;

    /** Constrói um objeto sem qualquer item registrado. */
    public CaixaRegistradora() { numItens = 0; total = 0.0; }

    /** Adiciona um item na caixa registradora.
        @param preco Preço do item a ser registrado. */
    public void adicionaItem(double preco) { numItens++; total = total + preco; }

    /** Obtém o valor total de todos os itens registrados.
        @return Valor total de todos os itens registrados. */
    public double obtemTotal() { return total; }

    /** Obtém o número de itens registrados.
        @return Número de itens registrados. */
    public int obtemNumItens() { return numItens; }

    /** Limpa a caixa registradora para iniciar uma nova venda. */
    public void limpa() { numItens = 0; total = 0.0; }

    /** Método estático que retorna informações sobre a classe.
        @return Cadeia de caracteres com o nome da classe e a sua versão. */
    public static String info() { return "CaixaRegistradora - Versão 1.0"; }
}

```

Fruteira.java

```
import java.util.Scanner;

/** Programa que realiza o serviço de caixa registradora para uma fruteira. */
public class Fruteira {
    /** Metodo inicial.
     * @param args Argumentos da linha de comandos (NÃO utilizado). */
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Fruteira - " + CaixaRegistradora.info() );
        CaixaRegistradora caixa = new CaixaRegistradora();
        System.out.println("Digite os valores dos itens e FIM para encerrar: ");
        while ( in.hasNextDouble() ) {
            double p = in.nextDouble();
            caixa.adicionaItem( p );
        }
        System.out.println("TOTAL = " + caixa.obtemTotal() );
        System.out.println("ITENS = " + caixa.obtemNumItens() );
    }
}
```

Exercício

- 1 Implemente uma classe chamada `ContaBancaria` que gerencie os dados de uma conta bancária, considerando as seguintes informações: número da conta (valor inteiro), nome do titular da conta (cadeia de caracteres) e saldo (valor real). Para gerenciar os objetos dessa classe implemente métodos para: construir objetos (considere um construtor que recebe todos os dados e outro que recebe o número da conta e o titular), realizar depósito, realizar saque (considere que o valor do saldo NÃO poderá ser negativo), obter os dados da conta, modificar os dados da conta e obter uma cadeia de caracteres com todos os dados da conta (chame este método de `toString()`).
- 2 Implemente uma classe com método `main()` para exemplificar o uso da classe `ContaBancaria`.

Solução: ContaBancaria.java

```
/** Classe que gerencia uma conta bancária. */  
public class ContaBancaria {  
    int numero;  
    String titular;  
    double saldo;  
  
    public ContaBancaria(int n, String t, double s) {  
        numero = n;  titular = t;  saldo = s;  
    }  
    public ContaBancaria(int n, String t) {  
        numero = n;  titular = t;  saldo = 0.0;  
    }  
    public int obtenNumero() { return numero; }  
    public String obtenTitular() { return titular; }  
    public double obtenSaldo() { return saldo; }  
    public void defineNumero(int n) { numero = n; }  
    public void defineTitular(String t) { titular = t; }  
    public void defineSaldo(double s) { saldo = s; }  
    public void deposita(double v) { saldo += v; }  
    public void saca(double v) {  
        if (v < saldo) { saldo -= v; }  
        else { saldo = 0.0; }  
    }  
    public String toString() { return numero + " - " + titular + ": R$" + saldo; }  
    public static String info() { return "ContaBancaria - Versão 1.0"; }  
}
```

Solução: GerenciaContaBancaria.java

```
import java.util.Scanner;

/** Este programa testa a classe ContaBancaria. */
public class GerenciaContaBancaria {
    /** Metodo inicial.
     * @param args Argumentos da linha de comandos (NÃO utilizado). */
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        ContaBancaria conta = new ContaBancaria(1234, "Fulano de Tal");
        System.out.println("Use: depositar <valor> ou sacar <valor> ou fim.");
        while ( true ) {
            System.out.printf( "[%d] %s => R$%.2f\n", conta.obtemNumero(), conta.obtemTitular(), conta.obtemSaldo() );
            String comando = in.next().trim().toLowerCase();
            if ( comando.equals("fim") ) break;
            else if ( comando.equals("depositar") ) {
                if ( in.hasNextDouble() ) conta.deposita( in.nextDouble() );
                else System.out.printf("\nERRO> Comando 'depositar' usado com valor inválido ('%s')!\n\n", in.next());
            }
            else if ( comando.equals("sacar") ) {
                if ( in.hasNextDouble() ) conta.saca( in.nextDouble() );
                else System.out.printf("\nERRO> Comando 'sacar' usado com valor inválido ('%s')!\n\n", in.next());
            }
            else System.out.printf("\nERRO> Comando '%s' inválido!\n\n", comando);
        }
    }
}
```

Testando uma Classe

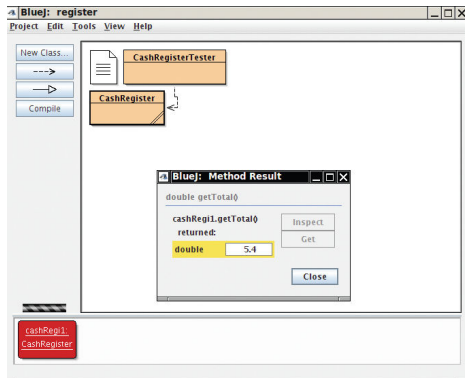
Testando uma Classe

- A maioria das classes que os programadores criam não possui método `main()`, pois elas são criadas para fazer parte de um programa maior
- Para testar uma classe será preciso criar um **teste unitário**
- Para testar uma nova classe pode-se usar:
 - Ferramentas de programação que criam objetos interativamente:
 - DrJava: <http://www.drjava.org>
 - BlueJ: <http://www.bluej.org>
 - Escrever uma classe de teste com um método `main()`:

```
public class TestaContaBancaria {  
    public static void main(String[] args) {  
        ContaBancaria c1 = new ContaBancaria();  
        ...  
    }  
}
```


Usando BlueJ para Teste

- BlueJ pode instanciar objetos de uma classe interativamente, o que permite que seus métodos sejam invocados
- Isto é excelente para realizar testes!



Criando uma Unidade de Teste

- Uma unidade de teste verifica se uma classe funciona corretamente de forma isolada (fora do programa completo)
- Ela deve testar todos os métodos, identificando quando alguma inconsistência for identificada

```
/** Este programa testa a classe CaixaRegistradora. */
public class TestaCaixaRegistradora {
    /** Metodo inicial.
        @param args Argumentos da linha de comandos (NÃO utilizado). */
    public static void main(String[] args) {
        CaixaRegistradora caixa = new CaixaRegistradora();
        if ( caixa.obtemNumItens() != 0 || caixa.obtemTotal() != 0.0 ) { System.err.println("ERRO"); System.exit(1); }
        caixa.adicionaItem(1.25);
        if ( caixa.obtemNumItens() != 1 || caixa.obtemTotal() != 1.25 ) { System.err.println("ERRO"); System.exit(1); }
        caixa.adicionaItem(0.65);
        if ( caixa.obtemNumItens() != 2 || caixa.obtemTotal() != 1.9 ) { System.err.println("ERRO"); System.exit(1); }
        caixa.adicionaItem(2.10);
        if ( caixa.obtemNumItens() != 3 || caixa.obtemTotal() != 4.0 ) { System.err.println("ERRO"); System.exit(1); }
        caixa.limpa();
        if ( caixa.obtemNumItens() != 0 || caixa.obtemTotal() != 0.0 ) { System.err.println("ERRO"); System.exit(1); }
        System.out.println( CaixaRegistradora.info() + " [OK] ");
    }
}
```

Padrões para Dados de Objetos

Padrões para Dados de Objetos

- Existem alguns padrões comuns quando variáveis de instância são projetadas
 - Manter um total
 - Contar eventos
 - Coletar valores
 - Gerenciar propriedades de objetos
 - Modelar objetos com diferentes estados
 - Descrever a posição de um objeto

Padrão: Manter um Total

- Exemplos

- Total de caixas registradoras
- Saldo de contas bancárias
- Nível do tanque de gasolina de um carro

- Variáveis necessárias

- Total: `total`

- Métodos necessários

- Adição: `adicionaItem()`
- Inicialização: `limpa()`
- Acesso: `obtemTotal()`

```
/** Simula uma caixa registradora simples.  
    @version 0.0 */  
public class CaixaRegistradora {  
    private double total;  
  
    public void adicionaItem(double preco) {  
        total += preco;  
    }  
  
    public void limpa() {  
        total = 0;  
    }  
  
    public double obtemTotal() {  
        return total;  
    }  
}
```

Padrão: Contar Eventos

- Exemplos

- Número de itens de uma caixa registradora
- Custo de transações bancárias

- Variáveis necessárias

- Contagem: `numItens`

- Métodos necessários

- Incrementar: `adicionaItem()`
- Inicialização: `limpa()`
- Acesso: `obtemNumItens()`

```
/** Simula uma caixa registradora.
    @version 1.0 */
public class CaixaRegistradora {
    private int numItens;
    private double total;

    public void adicionaItem(double preco) {
        numItens++;
        total = total + preco;
    }

    public double obtemTotal() {
        return total;
    }

    public int obtemNumItens() {
        return numItens;
    }

    public void limpa() {
        numItens = 0;
        total = 0.0;
    }
}
```

Padrão: Colectar Valores

Exemplos

- Carrinho de compras
- Questões de múltipla escolha
- Placar de pontos
- Opções de um menu

Valores armazenados

- *Array* parcialmente preenchido ou *Arraylist*

Construtor

- Inicializa ou cria a coleção vazia

Métodos necessários

- **Adição:** `adicionaItem()`, `obtemNumItens()`, `obtemItem()`

```
/** Gerencia um cacrrinho de compras em uma loja virtual,
    com até 50 itens, usando um vetor parcial. */
public class CarrinhoDeCompras {
    private String[] itens;
    private int numItens;

    public CarrinhoDeCompras() { // Constructor
        itens = new String[50];
        numItens = 0;
    }

    public void adicionaItem(String nome) {
        if (numItens < itens.length) {
            itens[numItens] = nome;
            numItens++;
        }
    }

    public int obtemNumItens() { return numItens; }

    public String obtemItem(int i) {
        if (i < 0 || i >= numItens) return null;
        return itens[i];
    }
}
```

Padrão: Gerenciar Propriedades de Objetos

- Uma propriedade de um objeto pode ser definida e recuperada
- Exemplos
 - Estudante: matrícula e nome
- Construtor
 - Inicializa as variáveis de instância
- Métodos necessários
 - Obtenção: `obtemMatricula()`, `obtemNome()`
 - Definição: `defineMatricula()`, `defineNome()`

```
/** Classe para gerenciar informações de um estudante. */
public class Estudante {
    private int matricula;
    private String nome;

    public Estudante(int m, String n) {
        matricula = m;
        nome = n;
    }

    public int obtenMatricula() { return matricula; }

    public String obtenNome() { return nome; }

    public void defineMatricula(int m) {
        matricula = m;
    }

    public void defineNome(String n) {
        nome = n;
    }
}
```


Padrão: Modelar Objetos com Diferentes Estados

- Alguns objetos podem estar em um estado de um conjunto de estados possíveis
- Exemplos
 - Um peixe que pode estar nos seguintes estados: sem fome, com alguma fome, com muita fome
- O peixe inicia sem fome (construtor)
- Os métodos alteram o estado do objeto
 - `come()`
 - `nada()`

```
/** Simula o comportamento de um peixe. */  
public class Peixe {  
    public static final int SEM_FOME = 0;  
    public static final int COM_ALGUMA_FOME = 1;  
    public static final int COM_MUITA_FOME = 2;  
    private int fome;  
  
    public Peixe() {  
        fome = SEM_FOME;  
    }  
  
    public void come() {  
        fome = SEM_FOME;  
    }  
  
    public void nada() {  
        if ( fome < COM_MUITA_FOME )  
            fome++;  
    }  
}
```

Padrão: Descrever a Posição de um Objeto

Exemplos

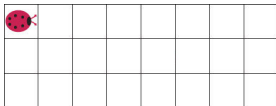
- Inseto em uma grade
- Objetos de um jogo
- Bala de canhão

Valores armazenados

- Linha, coluna, direção, etc.

Métodos necessários

- construtor
- anda()
- gira()



```
/** Gerencia a movimentação de um inseto em uma grade */
public class Inseto {
    public static final int LESTE = 0, SUL = 1,
                          OESTE = 2, NORTE = 3;
    public static final int LINHAS = 3, COLUNAS = 8;
    private int linha, coluna, direcao;

    public Inseto() {
        linha = coluna = direcao = 0;
    }

    public void anda() {
        switch (direcao) {
            case LESTE: if (coluna < COLUNAS-1) ++coluna; break;
            case OESTE: if (coluna > 0) --coluna; break;
            case SUL: if (linha < LINHAS-1) ++linha; break;
            case NORTE: if (linha > 0) --linha; break;
        }
    }

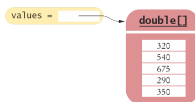
    public void gira() {
        if (direcao == NORTE) direcao = LESTE;
        else ++direcao;
    }
}
```

Referências a Objetos

Referências a Objetos

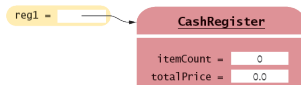
- Uma referência a um objeto especifica a localização de memória do objeto
- Objetos são parecidos com *arrays* porque eles também são acessados por variáveis de referência
 - Referência a *array*

```
double[] values = new double[5];
```



- Referência a objeto

```
CashRegister reg1 = new CashRegister();
```



Referências Compartilhadas

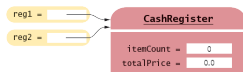
- Múltiplas variáveis do tipo objeto podem conter referências para o mesmo objeto.
 - Referência simples

```
CashRegister reg1 = new CashRegister();
```



- Referências compartilhando o mesmo objeto

```
CashRegister reg2 = reg1;
```

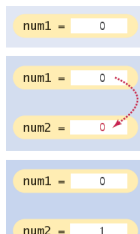


- Os valores internos podem ser alterados através de qualquer uma das referências

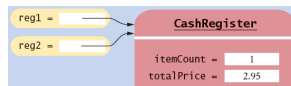
Cópia de Tipos Primitivos *versus* de Referências

- Variáveis de tipos primitivos podem ser copiadas, mas funcionam de forma diferente do que referências de objetos
- Cópia de dados primitivos: 2 localizações
- Cópia de referências: 1 localização para as 2 referências

```
int num1 = 0;  
int num2 = num1;  
num2++;
```



```
CashRegister reg1 = new CashRegister();  
CashRegister reg2 = reg1;  
reg2.addItem(2.95);
```



- Objetos podem ocupar muito mais espaço, por isso Java realiza a cópia apenas da referência

A Referência `null`

- Uma referência pode apontar para “nenhum” objeto `null`
 - Não se pode invocar métodos de um objeto através de uma referência `null`, pois isto causará uma exceção

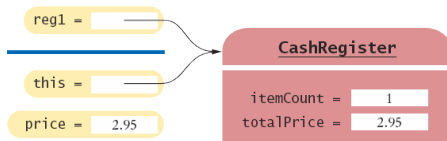
```
CashRegister reg = null;  
System.out.println(reg.getTotal());    // Runtime Error!
```

- Para testar se uma referência é `null` antes de acessá-la, usa-se:

```
String middleInitial = null; // No middle initial  
  
if (middleInitial == null)  
    System.out.println(firstName + " " + lastName);  
else  
    System.out.println(firstName + " " + middleInitial + "." + lastName);
```

A Referência `this`

- Métodos recebem um “parâmetro implícito” em uma variável de referência chamada `this`
 - Trata-se de uma referência ao objeto sobre o qual o método foi invocado:



- Assim pode-se deixar mais claro quando será usada uma variável de instância

```
void adicionaItem(double preco) {  
    this.numItens++;  
    this.total = this.total + preco;  
}
```


Referências a `this` em Construtores

- A referência `this` é muito usada em construtores
 - Ela torna mais claro que se está definindo variáveis de instância:

```
public class Student {  
    private int id;  
    private String name;  
    public Student(int id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
}
```

Variáveis Estáticas e Métodos

Variáveis e Métodos Estáticos

- Variáveis podem ser declaradas como `static` na declaração da classe
 - Haverá apenas uma cópia da variável `static` que será compartilhada entre todos os objetos da classe

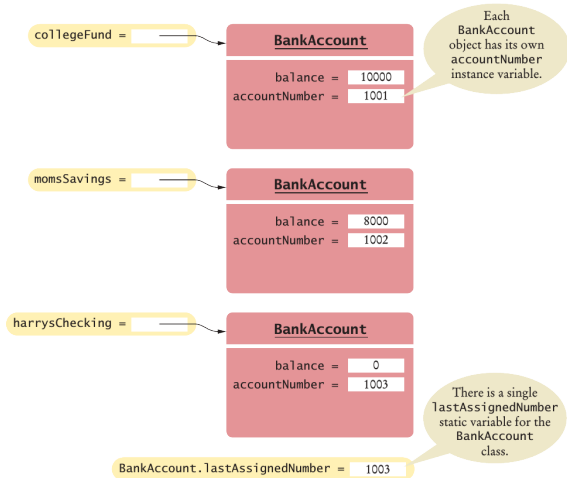
```
public class ContaBancaria {  
    private int numero;  
    private String titular;  
    private double saldo;  
    private static int proximoNumeroDeConta = 1000;  
  
    public ContaBancaria(String t) {  
        numero = proximoNumeroDeConta++;  
        titular = t;  
        saldo = 0.0;  
    }  
    // ...  
}
```

- Métodos de qualquer objeto da classe podem usar ou alterar o valor de uma variável `static`

Usando Variáveis Estáticas

- Exemplo:

- Cada vez que uma nova conta for criada, a variável `lastAssignedNumber` será incrementada pelo construtor
- Acessa-se a variável usando `NomeDaCasse.nomeDaVariavel`



Usando Métodos Estáticos

- A API de Java tem muitas classes que provêm métodos que podem ser usados sem que se necessite instanciar um objeto
 - A classe `Math` é um exemplo que já apareceu em exemplos anteriores
 - `Math.sqrt(valor)` é um método estático que retorna a raiz quadrada de um valor
 - Não é necessário instanciar um objeto da classe `Math` antes de usá-lo
- Acessa-se métodos `static` usando:

```
NomeDaClasse.nomeDoMetodo();
```

Escrevendo Seus Próprios Métodos Estáticos

- Você pode definir seus próprios métodos estáticos

```
public class Financial {  
    /**  
     * Computes a percentage of an amount.  
     * @param percentage the percentage to apply  
     * @param amount the amount to which the percentage is applied  
     * @return the requested percentage of the amount  
     */  
    public static double percentOf(double percentage, double amount) {  
        return (percentage / 100) * amount;  
    }  
}
```

- Invoca-se o método estático sobre a classe, e não sobre um objeto

```
double tax = Financial.percentOf(taxRate, total);
```

- Métodos estáticos geralmente retornam um valor. Eles apenas podem acessar variáveis estáticas e métodos estáticos

Sumário

Sumário: Classes e Objetos

Uma classe descreve um conjunto de objetos com o mesmo comportamento

- Cada classe tem uma interface pública: uma coleção de métodos através dos quais os objetos da classe podem ser manipulados
- Encapsulamento consiste em prover uma interface pública e esconder os detalhes de implementação
- Encapsulamento habilita alterações na implementação sem afetar os usuários da classe

Sumário: Variáveis e Métodos

- Variáveis de Instância de objetos armazenam dados que são usados pelos seus métodos
- Cada objeto de uma classe tem seu próprio conjunto de variáveis de instância
- Um método de instância pode acessar variáveis de instância do objeto sobre o qual ele atua
- Uma variável de instância privada pode ser acessada apenas por métodos de sua própria classe
- Variáveis declaradas como estáticas em uma classe possuem uma única cópia compartilhada entre todos os objetos criados a partir desta classe

Sumário: Cabeçalhos de Métodos, Dados

- Cabeçalhos de Métodos

- Pode-se usar cabeçalhos de métodos e comentários de métodos para especificar a interface pública de uma classe
- Um método *mutator* altera o objeto sobre o qual ele opera
- Um método *accessor* não altera o objeto sobre o qual ele atua

- Declaração de Dados

- Para cada método *accessor*, um objeto deve ou armazenar ou calcular o resultado
- Frequentemente há mais de uma forma de representar os dados de um objeto, e deve-se fazer uma escolha
- Deve-se ter certeza de que a representação de dados suporta chamadas de métodos em qualquer ordem

Sumário: Parâmetros, Construtores

- Parâmetros de Métodos

- O objeto sobre o qual um método é aplicado é o parâmetro implícito
- Parâmetros explícitos de um método são listados na declaração do método

- Construtores

- Um construtor inicializa as variáveis de instância do objeto
- Um construtor é invocado quando um objeto é criado com o operador `new`
- O nome de um construtor é sempre o mesmo que o nome da classe
- Uma classe pode ter múltiplos construtores
- O compilador seleciona o construtor compatível com os argumentos especificados na criação do objeto

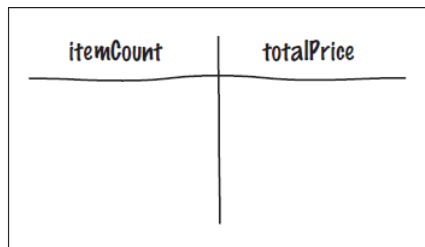
Tópicos Complementares

Depurando Objetos

- Uma sugestão para depurar programas que usam objetos é criar um cartão para cada objeto:
 - na parte frontal se apresentam os métodos da interface pública
 - no verso se controla os dados encapsulados



front



back

Depurando Objetos (2)

- Quando o construtor for chamado, as variáveis de instância são inicializadas

itemCount	totalPrice
0	0

- Quando um método *mutator* for chamado, será necessário atualizar variáveis de instância

itemCount	totalPrice
0 1	0 19.95

Referências

Referências

HORSTMANN, C. **Java for Everyone – Late Objects**. 2. ed. Hoboken: Wiley, 2013. xxxiv, 589 p.