

# Objetos e Classes

Roland Teodorowitsch

Fundamentos de Programação - Escola Politécnica - PUCRS

26 de outubro de 2022

# 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
- Interface Pública de uma Classe
- Projetando a Representação de Dados
- Implementando Métodos de Instância
- Construtores
- Testando uma Classe
- Solução de Problemas: Depurando Objetos
- Solução de Problemas: Padrões para Dados de Objetos
- Referências a Objetos
- Variáveis e Métodos Estáticos

# 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 greeting = "Hello World";  
int len = greeting.length();  
char c1 = greeting.charAt(0);
```

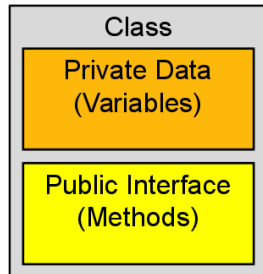
# Diagrama de Classes

- 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

- 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



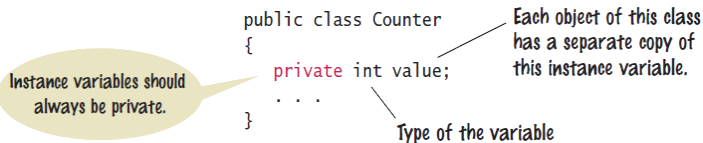
- O que deve ser feito?

- Incrementar o dispositivo
- Obter o valor atual



# Classe Counter

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



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

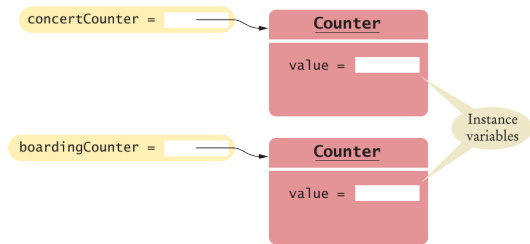
# Instanciando Objetos

- 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 `Counter`, usa-se:

```
// NomeClasse nomeObjeto = new NomeClasse();  
Counter concertCounter = new Counter();  
Counter boardingCounter = new Counter();
```



# Métodos da Classe Counter

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

```
concertCounter.count();  
boardingCounter.count();
```

```
public class Counter {  
    private int value;  
  
    public void count() {  
        value = value + 1;  
    }  
  
    public int getValue() {  
        return value;  
    }  
}
```

# 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>addItem(double)</code>	<code>void</code>
Obtém o total devido	<code>getTotal()</code>	<code>double</code>
Obtém o número de itens comprados	<code>getCount()</code>	<code>int</code>
Limpa o registro da caixa registradora para uma nova venda	<code>clear()</code>	<code>void</code>

# Escrevendo a Interface Pública de uma Classe

- Comentários no estilo Javadoc podem 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

```
/**  
 * A simulated cash register that tracks the item count and the total amount due.  
 */  
public class CashRegister {  
    /**  
     * Adds an item to this cash register.  
     * @param price: the price of this item  
     */  
    public void addItem(double price) {  
        // Method body  
    }  
    /**  
     * Gets the price of all items in the current sale.  
     * @return the total price  
     */  
    public double getTotal() ...  
}
```

## Método Não-Estático Significa...

- Até aqui os métodos das classes foram declarados com o modificador `static`
- Para métodos (de instância) não-estáticos, é preciso instanciar um objeto da respectiva classe antes que o método possa ser invocado
- Então, somente depois de criar um objeto, é possível invocar os seus métodos não-estáticos

```
public static void main(String[] args) {  
    // Construct a CashRegister object  
    CashRegister register1 = new CashRegister();  
    // Invoke a non-static method of the object  
    register1.addItem(1.95);  
}
```

# Métodos *Mutator* e *Accessor*

Muitos métodos podem ser de um dos seguintes tipos:

## 1 Métodos *Accessor* (métodos *get*)

- Solicitam uma informação ao objeto sem alterá-lo
- Normalmente retornam algum valor

```
public double getTotal() { ... }  
public int getCount() { ... }
```

## 2 Métodos *Mutator* (métodos *set*)

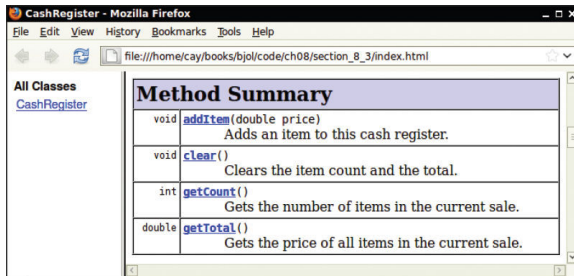
- Alteram valores no objeto
- Geralmente recebem um parâmetro que será usado para alterar uma variável de instância
- Normalmente retornam `void`

```
public void addItem(double price) { ... }  
public void clear() { ... }
```

# Tópico Especial: Javadoc

- O utilitário `javadoc` gera um conjunto de arquivos HTML a partir dos comentários no estilo Javadoc no código-fonte (por exemplo, métodos documentam parâmetros e retornos com `@param` e `@return`)
- Exemplo:

```
javadoc MyClass.java
```





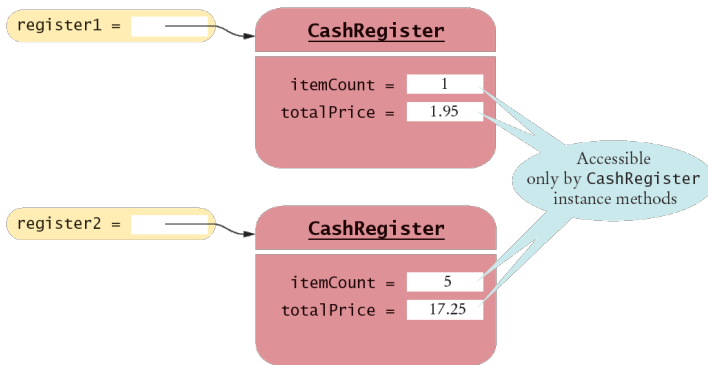
# Projetando a Representação de Dados

- Um objeto armazena dados em variáveis de instância
  - Variáveis de instância são declaradas dentro da classe
  - 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	<code>addItem()</code>	<code>total</code> , <code>count</code>
Obtém o <b>total</b> devido	<code>getTotal()</code>	<code>total</code>
Obtém o <b>número de itens</b> comprados	<code>getCount()</code>	<code>count</code>
Limpa o registro da caixa registradora para uma nova venda	<code>clear()</code>	<code>total</code> , <code>count</code>

# 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) {  
    // ...  
    System.out.println(register1.itemCount); // 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) {  
    // ...  
    System.out.println( register1.getCount() ); // OK  
    // ...  
}
```

# Implementando Métodos de Instância

- Métodos de instância acessam variáveis de instância privadas

```
public void addItem(double price) {  
    itemCount++;  
    totalPrice = totalPrice + price;  
}
```

- Métodos de instância são declarados dentro da classe
  - 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 são listados na declaração do método

# Sintaxe de Métodos de Instância

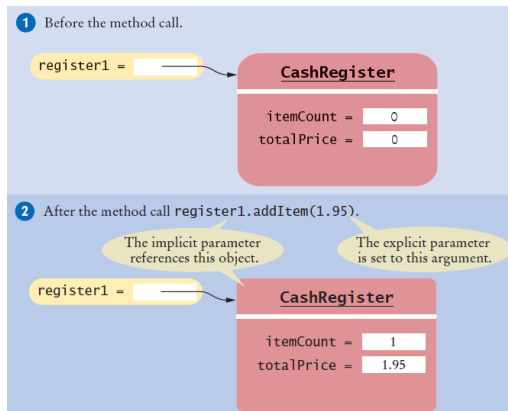
```
public class CashRegister
{
    . . .
    public void addItem(double price)
    {
        itemCount++;
        totalPrice = totalPrice + price;
    }
    . . .
}
```

Explicit parameter

Instance variables of  
the implicit parameter

# Parâmetros Implícitos e Explícitos

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



# 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

```
public class CashRegister {  
    private int itemCount;  
    private double totalPrice;  
    /**  
     * Constroi um objeto CashRegister com itemCount e totalPrice zerados.  
     */  
    public CashRegister() { // Construtor  
        itemCount = 0;  
        totalPrice = 0;  
    }  
}
```

# Múltiplos Construtores

- Uma classe pode ter mais de um construtor
  - Cada um deve ter um conjunto único de parâmetros

```
public class ContaCorrente {  
    ...  
    /**  
     * Constroi uma conta corrente com saldo zero.  
     */  
    public ContaCorrente( ) { ... }  
    /**  
     * Constroi uma conta corrente com um saldo especifico.  
     * @param saldoInicial o valor do saldo inicial  
     */  
    public ContaCorrente(double saldoInicial) { ... }  
}
```

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

```
ContaCorrente contaJoao = new ContaCorrente();  
ContaCorrente contaMaria = new ContaCorrente(499.95);
```



# Sintaxe de Construtores

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

```
public class BankAccount
{
    private double balance;

    public BankAccount()
    {
        balance = 0;
    }

    public BankAccount(double initialBalance)
    {
        balance = initialBalance;
    }
    . . .
}
```

A constructor has no return type, not even void.

A constructor has the same name as the class.

This constructor is picked for the expression `new BankAccount(499.95).`

# 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`

```
public class CashRegister {  
    private int itemCount;  
    private double totalPrice;  
    /**  
     * Faz exatamente o que um construtor gerado pelo compilador faria.  
     */  
    public CashRegister() {  
        itemCount = 0;  
        totalPrice = 0;  
    }  
}
```

# CashRegister.java (HORSTMANN, 2013, p. 377)

```
/**
 * A simulated cash register that tracks the item count and
 * the total amount due.
 */
public class CashRegister {

    private int itemCount;
    private double totalPrice;

    /**
     * Constructs a cash register with cleared item count and total.
     */
    public CashRegister() {
        itemCount = 0;
        totalPrice = 0;
    }

    /**
     * Adds an item to this cash register.
     * @param price the price of this item
     */
    public void addItem(double price) {
        itemCount++;
        totalPrice = totalPrice + price;
    }
}
```

# CashRegister.java (HORSTMANN, 2013, p. 377)

```
/**
 * Gets the price of all items in the current sale.
 * @return the total amount
 */
public double getTotal() {
    return totalPrice;
}

/**
 * Gets the number of items in the current sale.
 * @return the item count
 */
public int getCount() {
    return itemCount;
}

/**
 * Clears the item count and the total.
 */
public void clear() {
    itemCount = 0;
    totalPrice = 0;
}
}
```

# 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 detecta variáveis locais não inicializadas e avisa o programador

```
public class BankAccount {  
    private String name; // default constructor will set to null  
  
    public void showStrings() {  
        String localName;  
        // Runtime Error: java.lang.NullPointerException  
        System.out.println(name.length());  
        // Compiler Error: variable localName might not have been initialized  
        System.out.println(localName.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`

```
CashRegister register1 = new CashRegister();
```

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

```
register1.CashRegister(); // Error
```

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

```
CashRegister register1 = new CashRegister();  
register1.newItem(1.95);  
register1 = new CashRegister();
```

## 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 BankAccount {  
    /**  
        Intended to be a constructor.  
    */  
    public void BankAccount () {  
        ...  
    }  
}
```

## Tópico Especial: 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 print(CashRegister register)      { ... }  
void print(BankAccount account)       { ... }  
void print(int value)                  { ... }  
Void print(double value)               { ... }
```



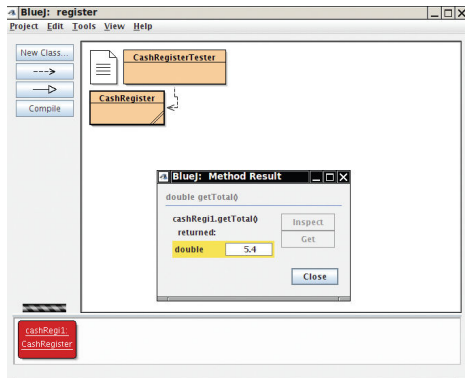
# Testando uma Classe

- 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 CashRegisterTester {  
    public static void main(String[] args) {  
        CashRegister cl = new CashRegister();  
        ...  
    }  
}
```

# 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, mostrando resultados esperados e resultados obtidos para que se possa compará-los
- `CashRegisterTester.java` (HORSTMANN, 2013, p. 377):

```
/**  
 * This program tests the CashRegister class.  
 */  
public class CashRegisterTester {  
    public static void main(String[] args) {  
        CashRegister register1 = new CashRegister();  
        register1.addItem(1.95);  
        register1.addItem(0.95);  
        register1.addItem(2.50);  
        System.out.println(register1.getCount());  
        System.out.println("Expected: 3");  
        System.out.printf("%.2f\n", register1.getTotal());  
        System.out.println("Expected: 5.40");  
    }  
}
```

## Resultado:

```
3  
Expected:5.40  
Expected:3  
5.40
```

# Passos para Implementar uma Classe

- 1 Crie uma lista informal de tarefas para os objetos: exibir menu, obter a entrada, etc.
- 2 Especifique a interface pública

```
public Menu();  
public void addOption(String option);  
public int getInput();
```

- 3 Documente a interface pública (use comentários do Javadoc)

```
/**  
    Adiciona uma opcao no final deste menu.  
    @param option a opcao a ser adicionada  
*/
```

- 4 Determine as variáveis de instância

```
private String[] options;                private int numOptions;
```

- 5 Implemente os construtores e métodos

```
public void addOption(String option) {  
    if (numOptions < options.length) {    options[numOptions++] = option; } else { ... }  
}
```

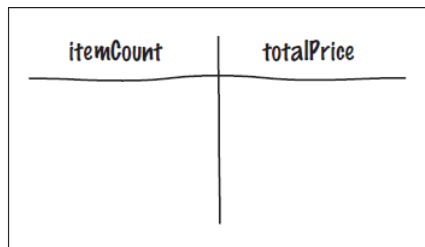
- 6 Teste a classe

# Solução de Problemas: 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*

## Solução de Problemas: 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
<del>0</del> 1	<del>0</del> 19.95

# Solução de Problemas: Padrões para Dados de Objetos

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

# Padrão: Manter um Total

- Exemplos

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

- Variáveis necessárias

- Total: `totalPrice`

- Métodos necessários

- Adição: `addItem()`
- Inicialização: `clear()`
- Acesso: `getTotal()`

```
public class CashRegister {  
    private double totalPrice;  
  
    public void addItem(double price) {  
        totalPrice += price;  
    }  
  
    public void clear() {  
        totalPrice = 0;  
    }  
  
    public double getTotal() {  
        return totalPrice;  
    }  
}
```



# Padrão: Contar Eventos

- Exemplos

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

- Variáveis necessárias

- Contagem: `itemCount`

- Métodos necessários

- Incrementar: `addItem()`
- Inicialização: `clear()`
- Acesso: `getCount()`

```
public class CashRegister {  
    private double totalPrice;  
    private int itemCount;  
  
    public void addItem(double price) {  
        totalPrice += price;  
        itemCount++;  
    }  
  
    public void clear() {  
        totalPrice = 0;  
        itemCount = 0;  
    }  
  
    public double getCount() {  
        return itemCount;  
    }  
}
```

# Padrão: Colectar Valores

- Exemplos
  - Questões de múltipla escolha
  - Carrinho de compras
  - Placar de pontos
- Valores armazenados
  - *Array* ou *arraylist*
- Construtor
  - Inicializa ou cria a coleção vazia
- Métodos necessários
  - Adição: `addItem()`

```
public class Cart {  
    private String[] items;  
    private int itemCount;  
  
    public Cart() { // Constructor  
        items = new String[50];  
        itemCount = 0;  
    }  
  
    public void addItem(String name) {  
        if(itemCount < 50) {  
            items[itemCount] = name;  
            itemCount++;  
        }  
    }  
}
```

# Padrão: Gerenciar Propriedades de Objetos

- Uma propriedade de um objeto pode ser definida e recuperada
- Exemplos
  - Estudante: nome, identificador
- Construtor
  - Define um valor único
- Métodos necessários
  - Definição: `set()`
  - Obtenção: `get()`

```
public class Student {  
    private String name;  
    private int ID;  
  
    public Student(int anID) {  
        ID = anID;  
    }  
  
    public void setName(String newname) {  
        if (newName.length() > 0)  
            name = newName;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

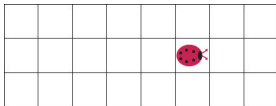
# Padrão: Modelar Objetos com Diferentes Estados

- 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
- Os métodos alteram o estado do objeto
  - `eat()`
  - `move()`

```
public class Fish {  
    private int hungry;  
    public static final int NOT_HUNGRY = 0;  
    public static final int SOMEWHAT_HUNGRY = 1;  
    public static final int VERY_HUNGRY = 2;  
  
    public void eat() {  
        hungry = NOT_HUNGRY;  
    }  
  
    public void move() {  
        if (hungry < VERY_HUNGRY)  
            hungry++;  
    }  
}
```

# Padrão: Descrever a Posição de um Objeto

- Exemplos
  - Objetos de um jogo
  - Inseto em uma grade
  - Bala de canhão
- Valores armazenados
  - Linha, coluna, direção, velocidade, etc.
- Métodos necessários
  - `move()`
  - `turn()`

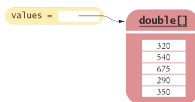


```
public class Bug {  
    private int row;  
    private int column;  
    private int direction;  
    // 0 = N, 1 = E, 2 = S, 3 = W  
  
    public void moveOneUnit() {  
        switch(direction) {  
            case 0: row--; break;  
            case 1: column++; break;  
            // ...  
        }  
    }  
}
```

# Referências a Objetos

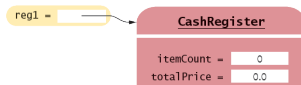
- Uma referência a um objeto especifica a localização de memória do objeto
- Objetos são parecidos com *arrays* porque eles também são acessados por variáveis de referência
  - Referência a *array*

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



- Referência a objeto

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



# Referências Compartilhadas

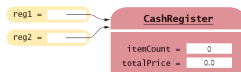
- Múltiplas variáveis do tipo objeto podem conter referências para o mesmo objeto.
  - Referência simples

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



- Referências compartilhando o mesmo objeto

```
CashRegister reg2 = reg1;
```

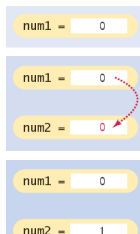


- Os valores internos podem ser alterados através de qualquer uma das referências

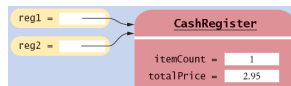
# Cópia de Tipos Primitivos *versus* de Referências

- 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

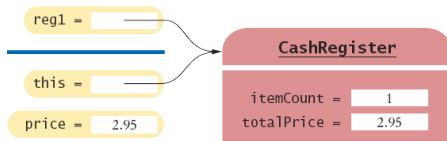
```
CashRegister reg = null;  
System.out.println(reg.getTotal());    // Runtime Error!
```

- 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 addItem(double price) {  
    this.itemCount++;  
    this.totalPrice = this.totalPrice + price;  
}
```

# 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 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

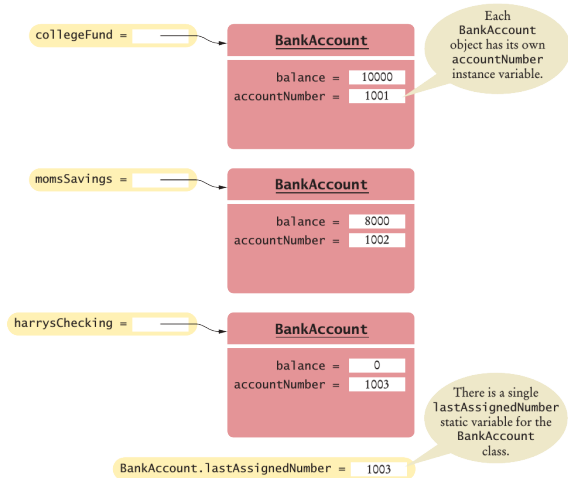
```
public class BankAccount {  
    private double balance;  
    private int accountNumber;  
    private static int lastAssignedNumber = 1000;  
  
    public BankAccount() {  
        lastAssignedNumber++;  
        accountNumber = lastAssignedNumber;  
    }  
    // ...  
}
```

- 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: 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

# Referências

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