Linguagem de Programação Orientada a Objetos

Caros alunos, as vídeo aulas desta disciplina encontram-se no AVA (Ambiente Virtual de Aprendizagem).

UniFil

Unidade 1

Introdução da Unidade

Nossa primeira unidade fará introdução de conceitos fundamentais para a programação. Muito importante acompanhar cada conceito e exercitar sempre que possível, utilizando os projetos indicados no texto. Assim, passo a passo, você irá desenvolver as habilidades para se tornar um programador de linguagens Orientadas à Objetos, como o Java.

Objetivos

- Aprender sobre os conceitos fundamentais da programação Orientada à Objetos.
- Descrever classes e suas responsabilidades.
- Criar e manipular objetos a partir de classes bem definidas.

Conteúdo programático

Aula 01 – Introdução à Orientação Objetos utilizando Java

Aula 02 – Definindo classes

Referências

BARNES, D. J.; KLLING, M. **Objects First with Java**: A Practical Introduction Using BlueJ. 6th edition. USA: Prentice Hall Press, 2017.

BlueJ. A free Java Development Environment designed for beginners. Disponível em: <https://bluej.org>. Acesso em: 18 maio. 2020.

Repositório de Projetos. Disponível em: <https://www.bluej.org/objects-first/resources/projects.zip>. Acesso em: 18 maio. 2020.

The BlueJ Tutorial (Portuguese). PDF. Translated by João Luiz Silva Barbosa. Disponível em: <https://www.bluej.org/tutorial/tutorial-portuguese.pdf>. Acesso em: 18 maio. 2020.

Videoaula Apresentação da Disciplina

Utilize o QR Code para assistir!

Assista!

Videoaula Minicurrículo

Utilize o QR Code para assistir!

Assista!

Aula 01 - Introdução à Orientação Objetos

Nossas aulas serão apresentadas em capítulos e seções, com conceitos fundamentais na programação orientada à objetos. Nossa linguagem de escolha será o Java, linguagem de grande importância para a comunidade de desenvolvedores, de fácil aprendizagem e com muitos recursos úteis. Além disso faremos uso de uma ferramenta de auxílio à programação, chamada BlueJ (<www.bluej.org>). O BlueJ é uma ferramenta de apoio à aprendizagem e de uso da linguagem de programação Java.

Introdução

É hora de começar nossa discussão sobre programação orientada a objetos. Aprender a programar exige uma mistura de alguma teoria e muita prática. Neste material, apresentaremos os dois, para que os dois se reforcem.

No centro da orientação a objetos estão dois conceitos que precisamos entender primeiro: objetos e classes. Esses conceitos formam a base de toda à programação em linguagens orientadas a objetos. Então, comecemos com uma breve discussão dessas duas fundamentações.

1.1 Objetos e classes

Se você escrever um programa de computador em uma linguagem orientada a objetos, está criando, em seu computador, um modelo de alguma parte do mundo. As partes das quais o modelo é construído são os objetos que aparecem no domínio do problema. Esses objetos devem ser representados no modelo de computador que está sendo criado. Os objetos do domínio do problema variam de acordo com o programa que você está escrevendo. Podem ser palavras e parágrafos se você estiver programando um processador de texto, usuários e mensagens se estiver trabalhando em um sistema de rede social ou “monstros” se estiver escrevendo um jogo de computador.

Objetos podem ser categorizados. Uma classe descreve, de maneira abstrata, todos os objetos de um tipo específico. Podemos tornar essas noções abstratas mais claras, observando um exemplo. Suponha que você queira modelar uma simulação de tráfego. Um tipo de entidade com a qual você precisa lidar é com carros. O que é um carro em nosso contexto? É uma classe ou um objeto? Algumas perguntas podem nos ajudar a tomar uma decisão.

Que cor é um carro? Quão rápido ele pode ir? Onde está agora?

Você notará que não podemos responder a essas perguntas até falarmos sobre um carro específico. A razão é que a palavra "carro", neste contexto, refere-se à classe carro; estamos falando de carros em geral, não de um carro em particular.

Se eu falar do “meu carro antigo que está estacionado em casa na minha garagem”, podemos responder às perguntas acima. Esse carro é vermelho, não vai muito rápido e está na minha garagem. Agora estou falando de um objeto - de um exemplo particular de carro.

Geralmente nos referimos a um objeto específico como uma instância. Usaremos o termo "instância" regularmente a partir de agora. "Instância" é aproximadamente sinônimo de "objeto"; nos referimos a objetos como instâncias quando queremos enfatizar que eles são de uma classe específica (como "este objeto é uma instância do carro de classe").

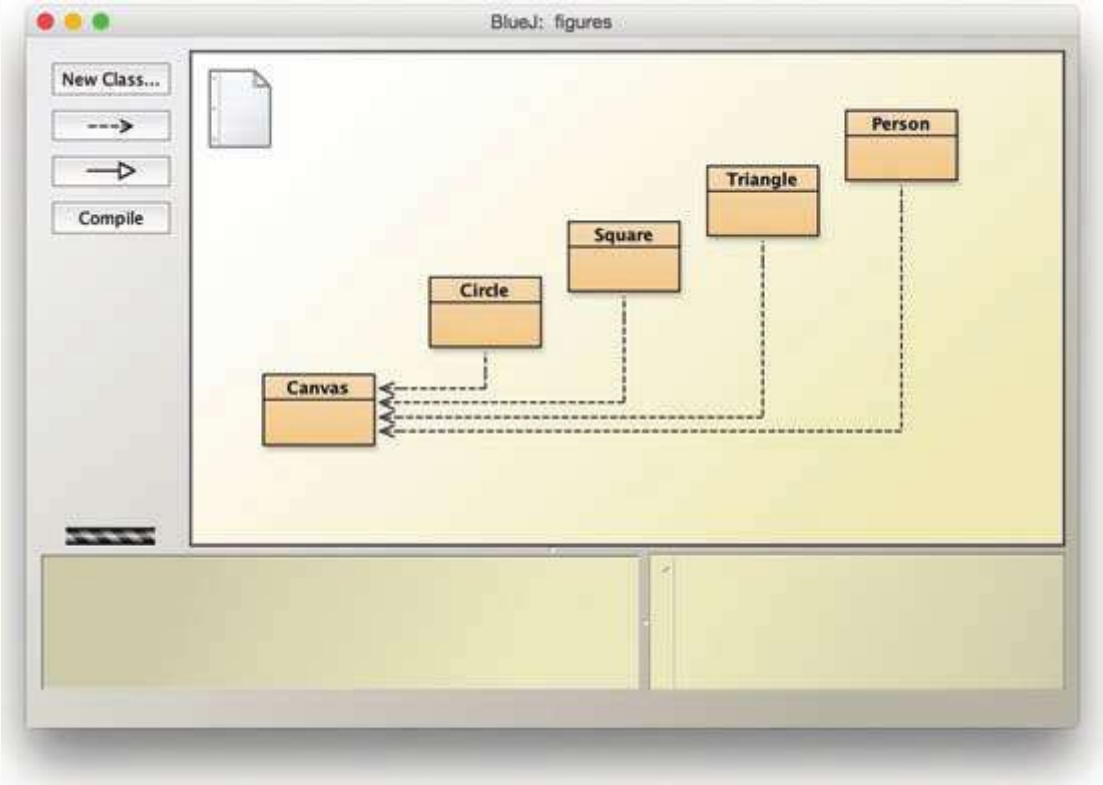
Antes de continuarmos essa discussão teórica, vejamos um exemplo.

1.2 Criando objetos

Inicie o BlueJ e abra o exemplo chamado figures. Você deve ver uma janela semelhante à mostrada na Figura 1.1.

Nesta janela, um diagrama deve se tornar visível. Cada um dos retângulos coloridos no diagrama representa uma classe em nosso projeto. Neste projeto, temos classes denominadas Círculo, Quadrado, Triângulo, Pessoa e Tela. Clique com o botão direito na classe *Circle* e escolha *new Circle().*

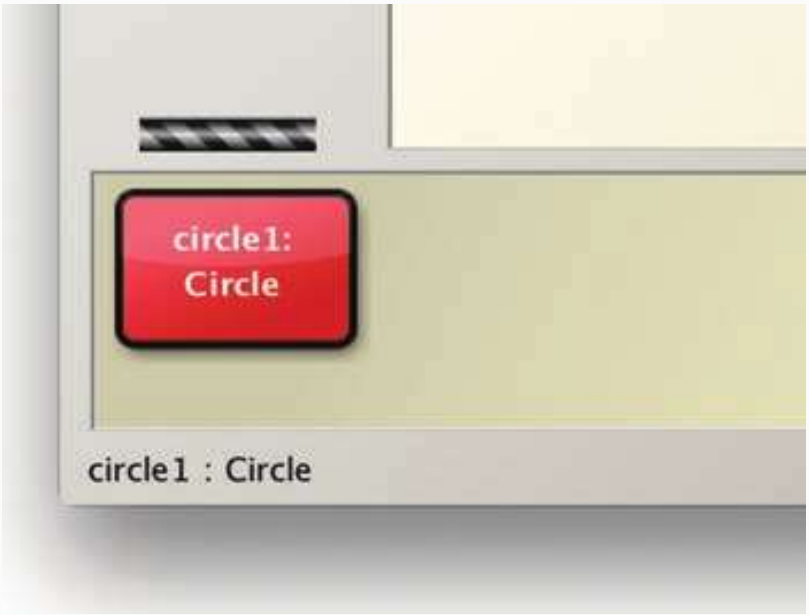
Figura 1.1 - *Figures - BlueJ*



Fonte: BlueJ.

No menu *pop-up.* O sistema solicita um "nome da instância"; clique em OK (o nome padrão fornecido é bom o suficiente por enquanto). Você verá um retângulo vermelho na parte inferior da tela rotulado como *circle1*(Figura 1.2).

Figura 1.2 - *Circle1*



Fonte: BlueJ.

Iniciamos os nomes de classes com letras maiúsculas (como Círculo) e nomes de objetos com letras minúsculas (como círculo1). Isso ajuda a distinguir do que estamos falando.

Você acabou de criar seu primeiro objeto! "Círculo", o ícone retangular na Figura 1.1, representa a classe *Circle*; *circle1* é um objeto criado a partir desta classe. A área na parte inferior da tela, onde o objeto é mostrado, é chamada de banco de objetos.

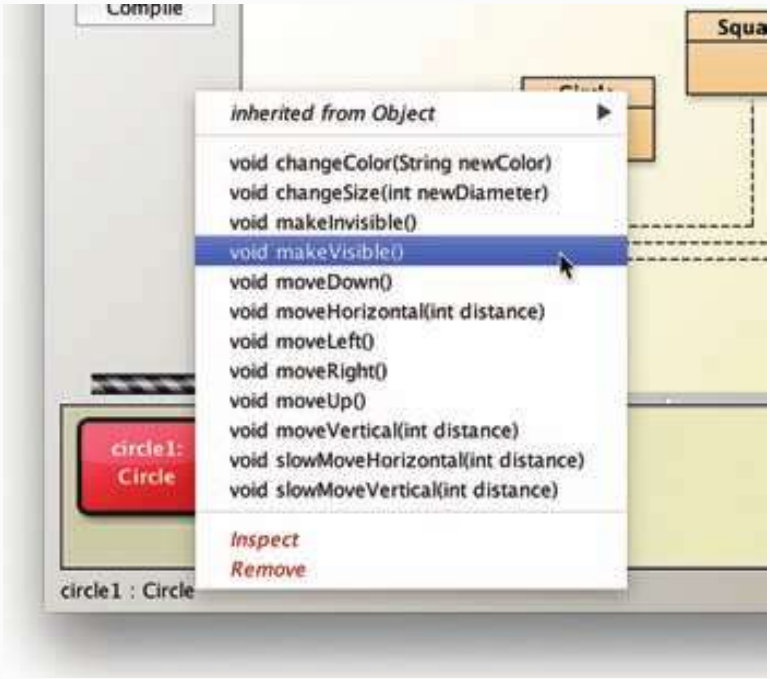
1.3 Chamando métodos

Clique com o botão direito do *mouse* em um dos objetos do círculo (não na classe!) E você verá um menu *pop-up* com várias operações (Figura 1.3). Escolha *makeVisible* no menu; isso irá desenhar uma representação desse círculo em uma janela separada (Figura 1.4)

Você notará várias outras operações no menu do círculo. Tente chamar *moveRight* e *moveDown* algumas vezes para mover o círculo para mais perto do canto da tela. Você também pode utilizar o *makeInvisible* e *makeVisible* para ocultar e mostrar o círculo.

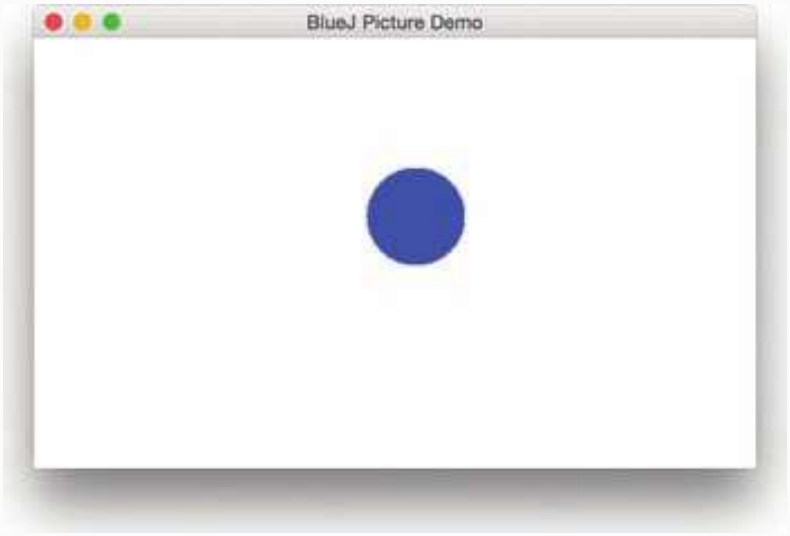
As entradas no menu do círculo representam operações que você pode usar para manipular o círculo. Estes são chamados métodos em Java. Usando terminologia comum, dizemos que esses métodos são chamados ou invocados. Usaremos essa terminologia apropriada a partir de agora. Podemos pedir que você "invoque o método *moveRight* do círculo1".

Figura 1.3 – Menu *pop-up*



Fonte: BlueJ.

Figura 1.4 – Representação do círculo



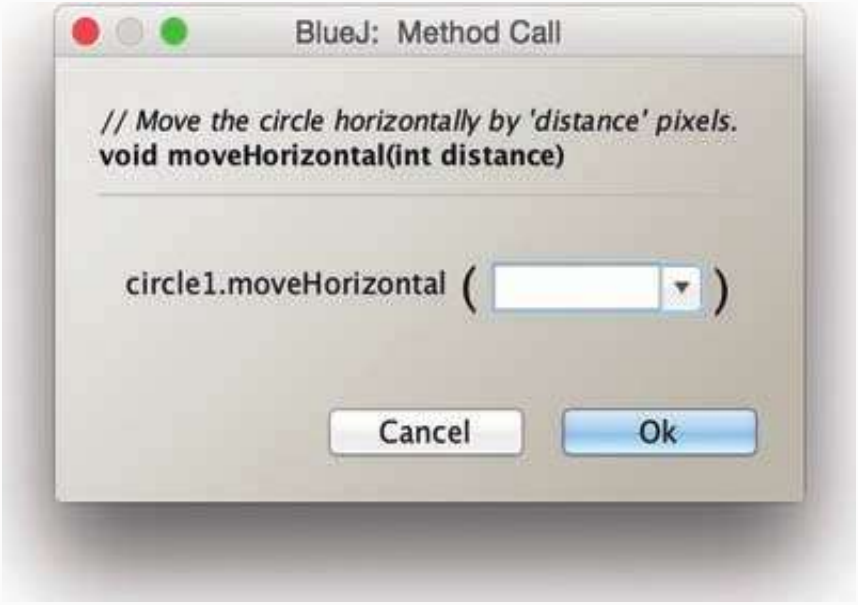
Fonte: BlueJ.

1.4 Parâmetros

Agora invoque o método *moveHorizontal*. Você verá uma caixa de diálogo solicitando alguma entrada (Figura 1.5), digite 50 e clique em OK. Você verá o círculo se mover 50 pixels para a direita.

O método *moveHorizontal*, que acabou de ser chamado, é escrito de tal forma que requer mais informações para ser executado. Nesse caso, a informação necessária é a distância - até onde o círculo deve ser movido. Portanto, o método *moveHorizontal* é mais flexível que os métodos *moveRight* e *moveLeft*. Este último, sempre move o círculo a uma distância fixa, enquanto *moveHorizontal* permite especificar até onde você deseja mover o círculo.

Figura 1.5 – Método *moveHorizontal*



Fonte: BlueJ

Exercício 1.3: tente chamar os métodos *moveVertical*, *slowMoveVertical* e *changeSize* antes de continuar lendo. Descubra como você pode usar o *moveHorizontal* para mover o círculo de 70 pixels para a esquerda.

Os valores adicionais que alguns métodos exigem são chamados de parâmetros. Um método indica que tipos de parâmetros ele requer. Ao chamar, por exemplo, o método *moveHorizontal*, como mostra a Figura 1.5, a caixa de diálogo exibe a linha próxima ao topo.

***void moveHorizontal* (distância *int*)**

Isso é chamado de cabeçalho do método. O cabeçalho fornece algumas informações sobre o método em questão. A parte entre parênteses (distância *int*) é a informação sobre o parâmetro requerido. Para cada parâmetro, ele define um tipo e um nome. O cabeçalho acima, afirma que o método requer um parâmetro do tipo *int* chamado *distance*. O nome fornece uma dica sobre o significado dos dados esperados. Juntos, o nome de um método e os tipos de parâmetros encontrados em seu cabeçalho são chamados de assinatura do método.

1.5 Tipos de dados

Um tipo específico de dados que pode ser passado para um parâmetro. O tipo *int* significa números inteiros (daí a abreviação "int"). No exemplo acima, a assinatura do método, *moveHorizontal,* afirma que, antes que o método possa ser executado, precisamos fornecer um número inteiro, especificando a distância a ser movida. O campo de entrada de dados, mostrado na Figura 1.5, permite inserir esse número. Nos exemplos até agora, o único tipo de dados que vimos foi *int*. Os parâmetros dos métodos de movimentação e o método *changeSize* são desse tipo. Uma inspeção mais detalhada do menu *pop-up* do objeto, mostra que as entradas do

método, no menu, incluem os tipos de parâmetros. Se um método não tiver parâmetro, o nome do método será seguido por um conjunto vazio de parênteses. Se tiver um parâmetro, o tipo e o nome desse parâmetro é exibido. Na lista de métodos para um círculo, você verá um método com um tipo de parâmetro diferente: o método *changeColor* possui um parâmetro do tipo *String*. O tipo *String* indica que uma seção de texto (por exemplo, uma palavra ou sentença) é esperada. As *strings* sempre são colocadas entre aspas duplas. Por exemplo, para inserir a palavra vermelho como uma sequência, digite "vermelho". A caixa de diálogo de chamada de método também inclui uma seção de texto, denominada comentário, acima do cabeçalho do método. Os comentários são incluídos para fornecer informações ao leitor (humano). O comentário do método *changeColor* descreve quais nomes de cores o sistema conhece.

Um erro comum para iniciantes é esquecer as aspas duplas ao digitar um valor de dados do tipo *String*. Se você digitar verde, ao invés de "verde", receberá uma mensagem de erro dizendo algo como "Erro: não é possível encontrar o símbolo - variável verde". Java suporta vários outros tipos de dados, incluindo números decimais e caracteres.

1.6 Várias instâncias

Depois de ter uma classe, você pode criar quantos objetos (ou instâncias) dessa classe desejar. Na classe Círculo, você pode criar muitos círculos. No *Square*, você pode criar muitos quadrados.

Cada um desses objetos tem sua própria posição, cor e tamanho. Você altera um atributo de um objeto (como seu tamanho) chamando um método nesse objeto. Isso afetará esse objeto em particular, mas não outros. Você também pode observar um detalhe adicional sobre os parâmetros. Dê uma olhada no método *changeSize* do triângulo.

void changeSize (int newHeight, int newWidth)

Aqui está um exemplo de um método com mais de um parâmetro. Este método possui dois e uma vírgula os separa no cabeçalho. Os métodos podem, de fato, ter qualquer número de parâmetros.

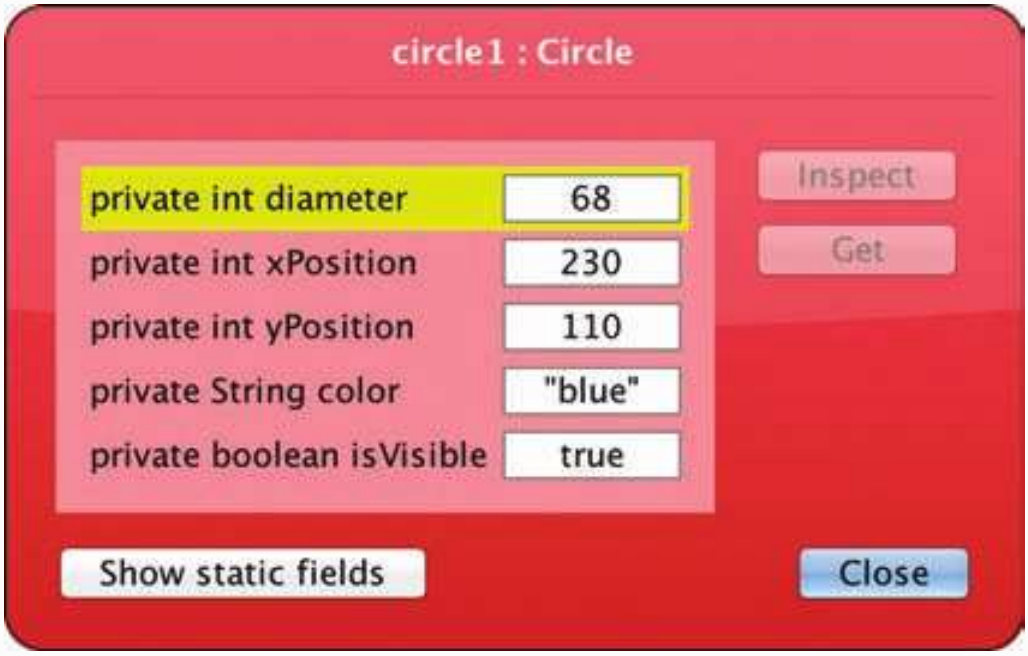
1.7 Estado

O conjunto de valores de todos os atributos que definem um objeto (como posição x, posição y, cor, diâmetro e status de visibilidade de um círculo) também é chamado de estado do objeto. Este é outro exemplo de terminologia comum que usaremos a partir de agora.

No BlueJ, o estado de um objeto pode ser inspecionado, selecionando a função “Inspecionar” no menu *pop-up* do objeto. Quando um objeto é inspecionado, um inspetor de objetos é exibido. O inspetor de objetos é uma visão ampliada do objeto que mostra os atributos armazenados dentro dele (Figura 1.6).

Alguns métodos, quando chamados, alteram o estado de um objeto. Por exemplo, *moveLeft* altera o atributo *xPosition*. A linguagem Java refere-se a esses atributos de objeto como campos.

Figura 1.6 - Atributos de objeto



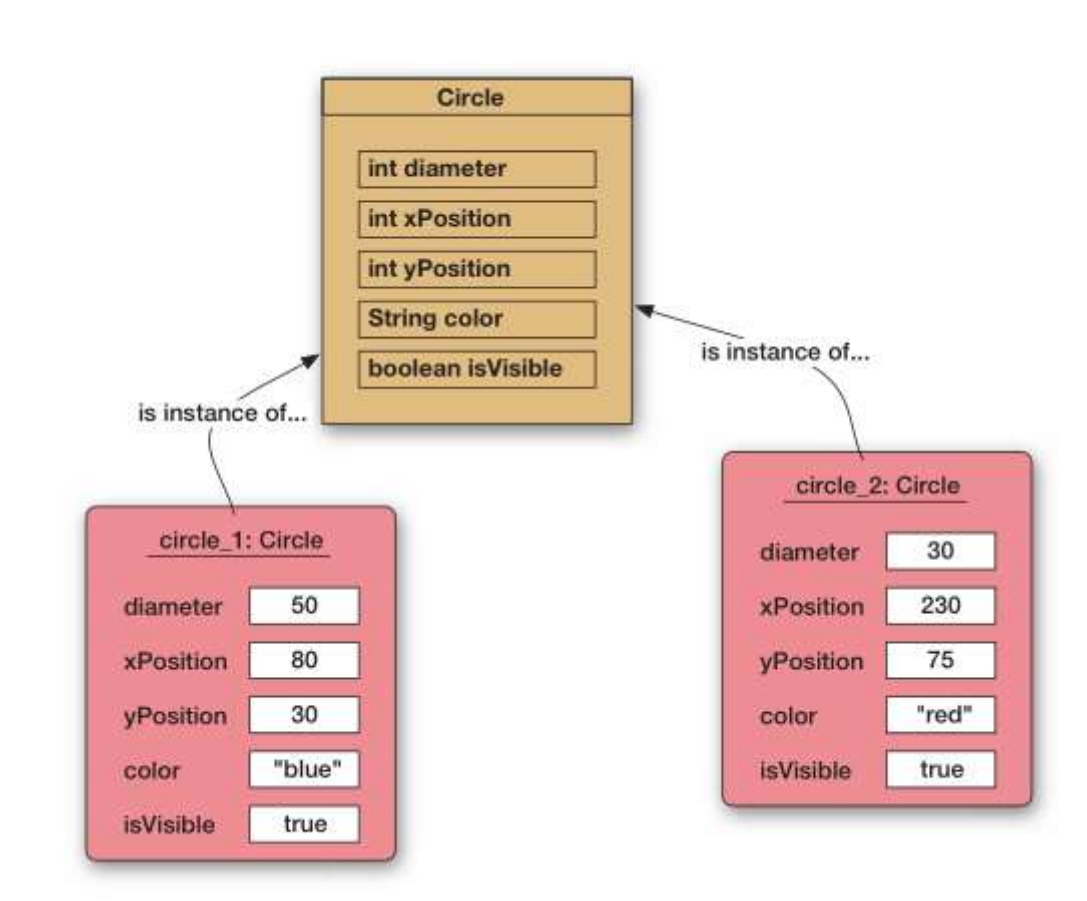
Fonte: BlueJ.

1.8 O que há em um objeto?

Ao inspecionar objetos diferentes, você notará que todos os objetos da mesma classe têm os mesmos campos. Ou seja, o número, o tipo e os nomes dos campos são os mesmos, enquanto o valor real de um campo específico em cada objeto pode ser diferente. Por outro lado, objetos de uma classe diferente podem ter campos diferentes. Um círculo, por exemplo, possui um campo de "diâmetro", enquanto um triângulo possui campos de "largura" e "altura".

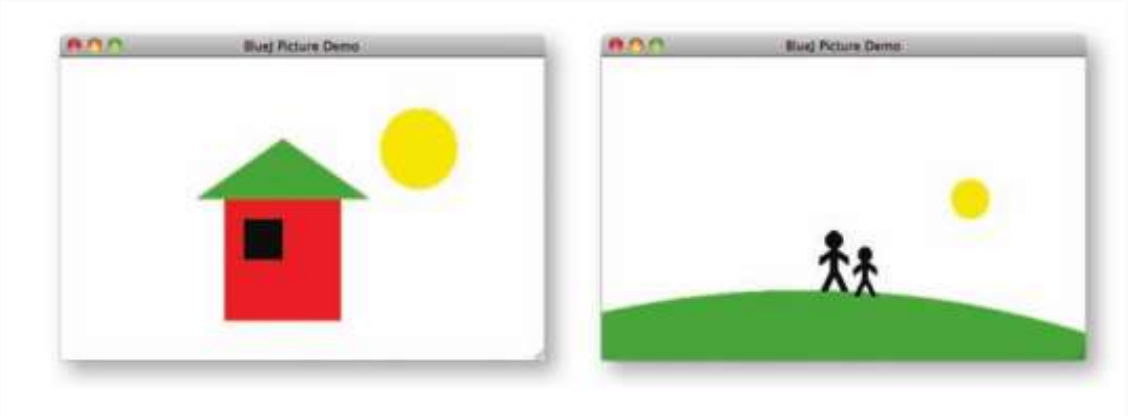
O motivo é que o número, os tipos e os nomes dos campos sejam definidos em uma classe, não em um objeto. Portanto, a classe *Circle* define que cada objeto de círculo terá cinco campos, denominados: diâmetro, *xPosition, yPosition*, cor e *isVisible*. Ele também define os tipos para esses campos. Ou seja, especifica que os três primeiros são do tipo *int,* enquanto a cor é do tipo *String* e o sinalizador *isVisible* é do tipo booleano. (booleano é um tipo que pode representar dois valores: verdadeiro e falso. Discutiremos mais detalhadamente adiante.) Quando um objeto da classe *Circle* é criado, o objeto terá automaticamente esses campos. Os valores dos campos são armazenados no objeto. Isso garante que cada círculo tenha uma cor, por exemplo, e cada um possa ter uma cor diferente (Figura 1.7).

Figura 1.7 – Exemplos de *Circle*



Fonte: BlueJ.

Figura 1.8 – BlueJ *Picture Demo*



Fonte: BlueJ.

A história é semelhante para métodos. Os métodos são definidos na classe do objeto. Como resultado, todos os objetos de uma determinada classe têm os mesmos métodos. No entanto, os métodos são chamados em objetos. Isso deixa claro qual objeto alterar quando, por exemplo, um método *moveRight* é chamado.

1.9 Código Java

Quando programamos em Java, basicamente escrevemos instruções para chamar métodos em objetos, assim como fizemos com nossos objetos de figura acima. No entanto, não fazemos isso de maneira interativa, escolhendo métodos em um menu com o mouse, mas, ao invés disso, digitamos os comandos na forma de texto. Podemos ver como esses comandos se parecem em forma de texto usando o Terminal BlueJ.

1.10 Interação de objeto

Para a próxima seção, trabalharemos com um exemplo de projeto diferente. Feche o projeto de figuras se você ainda o tiver aberto e abra o projeto chamado *House*.

Cinco das classes no projeto da casa são idênticas às classes no projeto das figuras. Mas agora temos uma classe adicional: *Picture*.

Na realidade, se queremos uma sequência de tarefas executadas em Java, normalmente não faríamos isso manualmente. Ao invés disso, criaremos uma classe que faz isso por nós. Esta é a classe *Picture*.

A classe Picture é escrita para que, quando você criar uma instância, a instância crie dois objetos quadrados (um para a parede, outro para a janela), um triângulo e um círculo. Então, quando você chama o método *draw*, ele os move e muda sua cor e tamanho, até que a tela se pareça com a imagem que vimos na Figura 1.8.

Os pontos importantes aqui são os seguintes: objetos podem criar outros objetos; e eles podem chamar os métodos um do outro. Em um programa Java normal, você pode ter centenas ou milhares de objetos. O usuário de um programa apenas inicia o programa (que normalmente cria um primeiro objeto) e todos os outros objetos são criados, direta ou indiretamente, por esse objeto. A grande questão agora é esta: como escrevemos a classe para esse objeto?

1.11 Código fonte

Cada classe tem algum código fonte associado a ela. O código fonte é um texto que define os detalhes da classe. No BlueJ, o código-fonte de uma classe pode ser visualizado selecionando a função *Open Editor* no menu *pop-up* da classe ou clicando duas vezes no ícone da classe.

Nota sobre “Sobre a compilação”: quando as pessoas escrevem programas de computador, geralmente usam uma linguagem de programação de "nível superior", como Java. Um problema disso é que um computador não pode executar o código fonte Java diretamente. O Java foi projetado para ser razoavelmente fácil de ler para humanos, não para computadores. Os computadores, internamente, trabalham com uma representação binária de um código de máquina, que parece bem diferente do Java. O problema para nós é que parece tão complexo que não queremos escrevê-lo diretamente. Nós preferimos escrever Java. O que podemos fazer sobre isso? A solução é um programa chamado compilador. O compilador converte o código Java em código de máquina. Podemos escrever Java e executar o compilador - que gera o código

da máquina - e o computador pode ler o código da máquina. Como resultado, toda vez que alteramos o código fonte, devemos primeiro executar o compilador antes de podermos usar a classe novamente para criar um objeto. Caso contrário, a versão do código da máquina necessária ao computador não existirá.

1.12 Outro exemplo

Nesta unidade, já discutimos um grande número de novos conceitos. Para ajudar a entender esses conceitos, agora os revisitaremos em um contexto diferente. Para isso, usamos outro exemplo. Feche o projeto da casa, se ainda estiver aberto, e abra o projeto das aulas de laboratório.

Este projeto é uma parte simplificada de um banco de dados de alunos, desenvolvido para acompanhá-los nas aulas de laboratório e imprimir listas de turmas.

1.13 Retornar valores

Como antes, você pode criar vários objetos. E, novamente, os objetos têm métodos que você pode chamar nos menus *pop-up*. Exercício: crie alguns objetos de Aluno. Chame o método *getName* em cada objeto.

Ao chamar o método *getName* da classe *Student*, notamos algo novo: os métodos podem retornar um valor de resultado. De fato, o cabeçalho de cada método nos diz se retorna ou não um resultado e qual é o tipo do resultado. O cabeçalho de *getName* (como mostrado no menu *pop-up* do objeto) é definido como: *String getName ().*

String getName ()

A palavra *String* antes do nome do método especifica o tipo de retorno. Nesse caso, ele afirma que a chamada desse método retornará um resultado do tipo *String*. O cabeçalho de *changeName* declara: *void changeName* (*String ReplacementName*).

A palavra, nulo, indica que este método não retorna nenhum resultado.

Métodos com valores de retorno nos permitem obter informações de um objeto por meio de uma chamada de método. Isso significa que podemos usar métodos para alterar o estado de um objeto ou para descobrir sobre seu estado. O tipo de retorno de um método não faz parte de sua assinatura.

1.14 Objetos como parâmetros

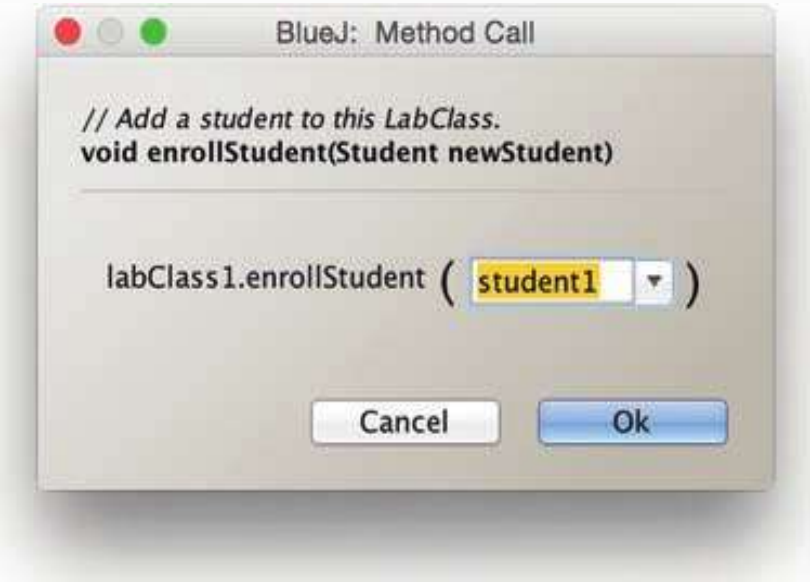
Exercício: crie um objeto da classe *LabClass*. Como a assinatura indica, você precisa especificar o número máximo de alunos nessa turma (um número inteiro).

Exercício: veja a assinatura do método *registerStudent*. Você notará que o tipo do parâmetro esperado é “Aluno”. Verifique se você tem dois ou três alunos e um objeto *LabClass* no banco de objetos, e chame o método *registerStudent* do objeto *LabClass*. Com o cursor de entrada no

campo da caixa de diálogo, clique em um dos objetos do aluno; isso insere o nome do objeto aluno no campo de parâmetro do método *registerStudent* (Figura 1.9). Clique em OK e você adicionou o aluno à *LabClass*. Adicione um ou mais alunos.

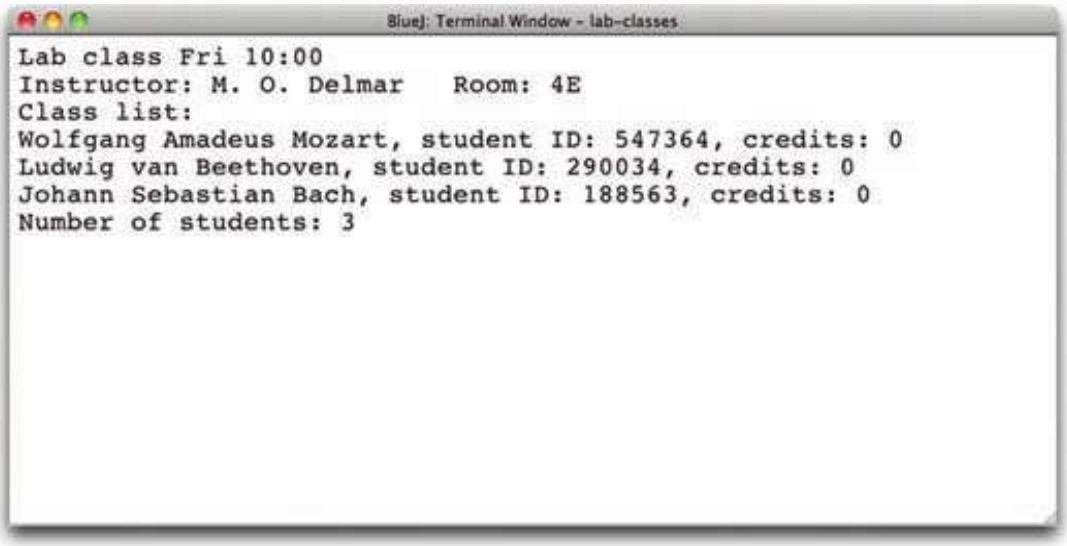
Exercício, chame o método *printList* do objeto *LabClass*. Você verá uma lista de todos os alunos dessa turma impressa na janela do terminal BlueJ (Figura 1.10).

Figura 1.9 - Parâmetro do método *registerStudent*



Fonte: BlueJ.

Figura 1.10 - Janela do terminal BlueJ



Fonte: BlueJ.

Como mostram os exercícios, os objetos podem ser passados como parâmetros para os métodos de outros objetos. No caso em que um método espera um objeto como parâmetro, o nome da classe do objeto esperado é especificado como o tipo de parâmetro na assinatura do método.

Explore este projeto um pouco mais. Tente identificar os conceitos discutidos no exemplo das figuras, neste contexto.

1.15 Sumário

Neste capítulo, exploramos o básico de classes e objetos. Discutimos o fato de que os objetos são especificados por classes. Classes, representam o conceito geral de coisas, enquanto objetos, representam instâncias concretas de uma classe. Podemos ter muitos objetos de qualquer classe. Os objetos têm métodos que usamos para nos comunicar com eles. Podemos usar um método para fazer uma alteração no objeto ou obter informações do objeto. Os métodos podem ter parâmetros e os parâmetros têm tipos. Os métodos têm tipos de retorno, que especificam que tipo de dados eles retornam. Se o tipo de retorno for nulo, eles não retornam nada. Objetos armazenam dados em campos (que também possuem tipos). Todos os valores de dados de um objeto juntos são chamados de estado do objeto.

Os objetos são criados a partir de definições de classe que foram escritas em uma linguagem de programação específica. Grande parte da programação em Java é sobre aprender a escrever definições de classe.

Um grande programa Java terá muitas classes, cada uma com muitos métodos que são chamados de várias maneiras diferentes.

Para aprender a desenvolver programas em Java, precisamos aprender a escrever definições de classe, incluindo campos e métodos, e como reuni-las bem. O restante deste, lida com esses problemas.

Videoaula 1

Utilize o QR Code para assistir!

Videoaula 2

Utilize o QR Code para assistir!

Videoaula 3

Utilize o QR Code para assistir!

Aula 02 - Definindo classes

Neste capítulo, examinamos primeiro o código fonte de uma classe. Discutiremos os elementos básicos das definições de classe: campos, construtores e métodos. Os métodos contêm instruções e, inicialmente, analisamos métodos que contêm apenas instruções aritméticas e de impressão simples. Posteriormente, apresentamos instruções condicionais que permitem escolhas entre diferentes ações a serem feitas nos métodos.

Começaremos examinando um novo projeto em uma quantidade razoável de detalhes. Este projeto representa uma implementação simples de uma máquina de tickets automatizada. Quando começamos a introduzir os recursos mais básicos das classes, descobrimos rapidamente que essa implementação é deficiente de várias maneiras. Portanto, procederemos à descrição de uma versão mais sofisticada da máquina de tickets que representa uma melhoria significativa. Por fim, para reforçar os conceitos introduzidos neste capítulo, examinamos as partes internas do exemplo de classe de laboratório encontrado na aula 01.

2.1 Máquinas de bilhetes (*Ticket machines*)

As estações de trem, geralmente, fornecem máquinas de bilhetes que imprimem um bilhete quando um cliente insere o dinheiro correto para sua tarifa. Neste capítulo, definiremos uma classe que modela algo como essas máquinas de bilhetes. Como examinaremos nossas primeiras classes de exemplo Java, manteremos nossa simulação bastante simples para começar. Isso nos dará a oportunidade de fazer algumas perguntas sobre como esses modelos diferem das versões do mundo real e como podemos mudar nossas classes para tornar os objetos que eles criam mais parecidos com os reais.

Nossas máquinas de bilhetes trabalham pelos clientes, "inserindo" dinheiro nelas e solicitando a impressão de um bilhete. Cada máquina mantém uma quantidade total de dinheiro que coletou durante sua operação. Na vida real, muitas vezes, acontece que uma máquina de bilhetes oferece uma seleção de diferentes tipos de bilhetes, a partir da qual os clientes escolhem o que desejam. Nossas máquinas simplificadas imprimem bilhetes por apenas um preço. É significativamente mais complicado programar uma classe para poder emitir *tickets* de valores diferentes, do que oferecer um preço único. Por outro lado, com a programação orientada a objetos, é muito fácil criar várias instâncias da classe, cada uma com sua própria configuração de preço, para atender à necessidade de diferentes tipos de *tickets*.

2.1.1 Explorando o comportamento de uma máquina de bilhetes simples

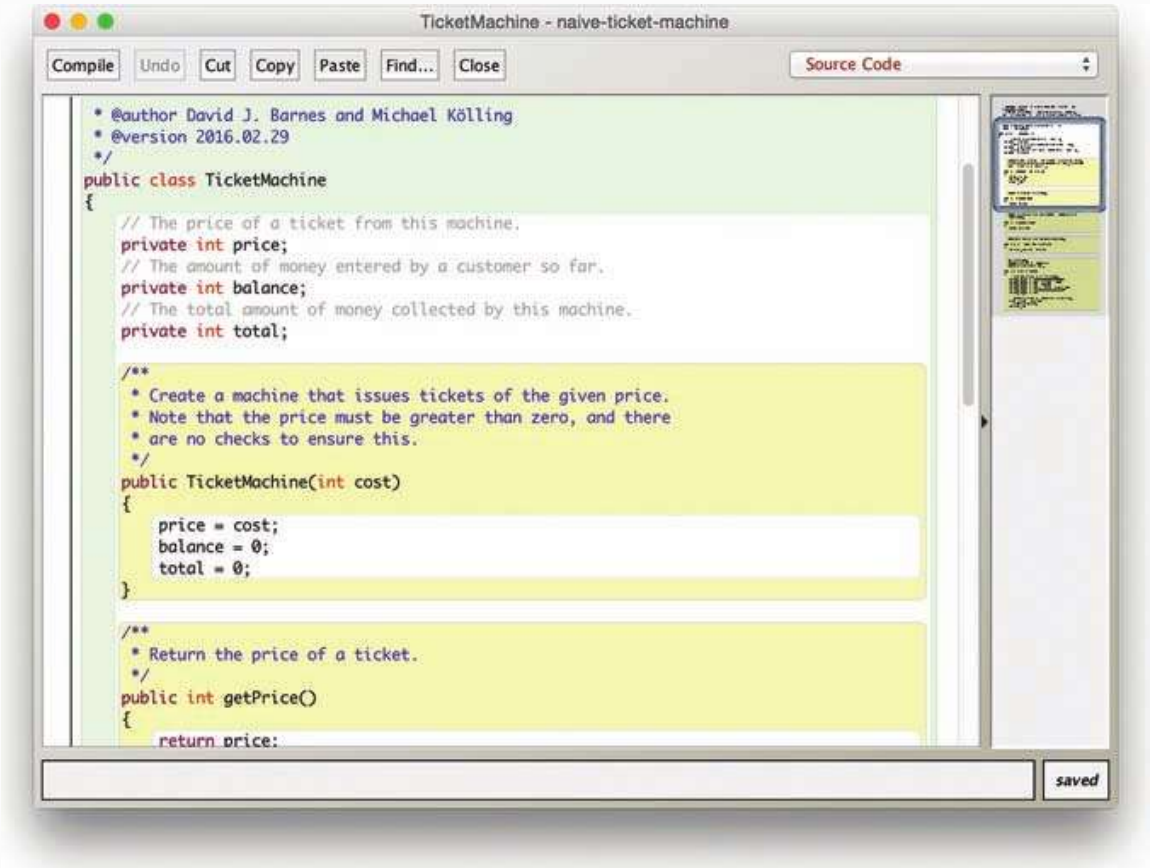
Abra o projeto *naíve-ticket-machine* no BlueJ. Este projeto contém apenas uma classe *TicketMachine* - que você poderá explorar de maneira semelhante aos exemplos discutidos na aula 01. Ao criar uma instância do *TicketMachine*, você será solicitado a fornecer um número que corresponda ao preço dos ingressos que serão emitidos por essa máquina específica. O preço é considerado um número de centavos; portanto, um número inteiro positivo, como 500, seria apropriado como um valor para trabalhar.

2.2 Examinando uma definição de classe

No final da seção anterior vimos que os objetos, *TicketMachine,* apenas se comportam da maneira que esperamos, se inserirmos a quantia exata de dinheiro que corresponde ao preço de um ingresso. Ao explorarmos os detalhes internos da classe nesta seção, veremos por que isso acontece desta forma.

Veja no código fonte da classe *TicketMachine,* clicando duas vezes em seu ícone no diagrama de classes no BlueJ. Deve parecer algo como a Figura 2.1. O texto completo da classe é mostrado no Código 2.1. Observando o texto da definição de classe, peça por peça, podemos detalhar alguns dos conceitos orientados a objetos discutidos na aula 01. Essa definição de classe contém muitos dos recursos do Java, que veremos repetidamente, portanto, pegará muito para estudá-lo cuidadosamente.

Figura 2.1 - Diagrama de classes no BlueJ



Fonte: BlueJ.

Código 2.1

```
/**
* TicketMachine models a naive ticket machine that issues
* flat-fare tickets.
* The price of a ticket is specified via the constructor.
* It is a naive machine in the sense that it trusts its users
* to insert enough money before trying to print a ticket.
* It also assumes that users enter sensible amounts.
*
* @author David J. Barnes and Michael Kolling
* @version 2008.03.30
*/
public class TicketMachine
{
  // The price of a ticket from this machine.
  private int price;
  // The amount of money entered by a customer so far.
  private int balance;
  // The total amount of money collected by this machine.
  private int total;


  /**
   * Create a machine that issues tickets of the given price.
   * Note that the price must be greater than zero, and there
   * are no checks to ensure this.
   */
```

```
public TicketMachine(int ticketCost)
{
    price = ticketCost;
    balance = 0;
    total = 0;
}


/**
 * Return the price of a ticket.
 */
public int getPrice()
{
    return price;
}


/**
 * Return the amount of money already inserted for the
 * next ticket.
 */
public int getBalance()
{
    return balance;
}


/**
 * Receive an amount of money in cents from a customer.
 */
public void insertMoney(int amount)
{
    balance = balance + amount;
}
```

```
    /**
     * Print a ticket.
     * Update the total collected and
     * reduce the balance to zero.
     */
    public void printTicket()
    {
        // Simulate the printing of a ticket.
        System.out.println("##################");
        System.out.println("# The BlueJ Line");
        System.out.println("# Ticket");
        System.out.println("# " + price + " cents.");
        System.out.println("##################");
        System.out.println();


        // Update the total collected with the balance.
        total = total + balance;
        // Clear the balance.
        balance = 0;
    }
}
```

2.3 O cabeçalho da classe

O texto de uma classe pode ser dividido em duas partes principais: um pequeno invólucro externo que simplesmente nomeia a classe (aparecendo em um fundo verde) e uma parte interna muito maior que faz todo o trabalho. Nesse caso, o invólucro externo aparece da seguinte maneira:

```
public class TicketMachine
{

    // Parte interna da classe omitida.
}
```

Os invólucros externos de diferentes classes parecem todos iguais. O invólucro externo contém o cabeçalho da classe, cujo objetivo principal é fornecer um nome para a classe. Por uma convenção amplamente seguida, sempre começamos os nomes das classes com uma letra maiúscula. Desde que seja usada de forma consistente, essa convenção permite que os nomes de classes sejam facilmente distinguidos de outros tipos de nomes, como nomes de variáveis e nomes de métodos, que serão descritos em breve. Acima do cabeçalho da classe, há um comentário (mostrado como texto em azul) que nos diz algo sobre a classe.

2.3.1 Palavras-chave

As palavras "público" e "classe" fazem parte da linguagem Java, enquanto a palavra "*TicketMachine*" não é - a pessoa que escreveu a classe escolheu esse nome específico. Chamamos palavras como palavras-chave "públicas" e "de classe" ou palavras reservadas - os termos são usados com frequência e de forma intercambiável. Existem cerca de 50 deles em Java, e em breve você poderá reconhecer a maioria deles. Um ponto que vale a pena lembrar é

que as palavras-chave Java nunca contêm letras maiúsculas, enquanto as palavras que escolhemos (como "*TicketMachine*") costumam ser uma mistura de letras maiúsculas e minúsculas.

2.4 Campos, construtores e métodos

A parte interna da classe é onde definimos os campos, construtores e métodos que dão aos objetos dessa classe suas próprias características e comportamento particulares. Podemos resumir os recursos essenciais desses três componentes de uma classe da seguinte maneira:

- Os campos armazenam dados persistentemente dentro de um objeto.

- Os construtores são responsáveis por garantir que um objeto seja configurado corretamente quando ele for criado.

- Os métodos implementam o comportamento de um objeto; eles fornecem sua funcionalidade.

No BlueJ, os campos são mostrados como texto em um fundo branco, enquanto os construtores e métodos são exibidos como caixas amarelas.

Em Java, existem poucas regras sobre a ordem em que você pode escolher para definir os campos, os construtores e os métodos dentro de uma classe. Na classe *TicketMachine*, optamos por listar os campos primeiro, os construtores em segundo e, finalmente, os métodos (Código 2.2). Essa é a ordem que seguiremos em todos os nossos exemplos. Outros autores optam por adotar estilos diferentes, e isso é principalmente uma questão de preferência. Nosso estilo não é necessariamente melhor que todos os outros. No entanto, é importante escolher um estilo e usá-lo de forma consistente, pois suas aulas serão mais fáceis de ler e entender.

Código 2.2

```
public class ClassName
{
    Fields
    Constructors
    Methods
}
```

2.4.1 Campos

Os campos armazenam dados dentro de um objeto. A classe *TicketMachine* possui três campos: preço, saldo e total. Os campos também são conhecidos como variáveis de instância, porque a palavra variável é usada como um termo geral para itens que armazenam dados em um programa. Definimos os campos logo no início da definição da classe (Código 2.3). Todas essas variáveis estão associadas a itens monetários com os quais um objeto de máquina de *ticket* deve lidar:

- *price*, armazena o preço fixo de um bilhete;

- *balance*, armazena a quantidade de dinheiro inserida na máquina por um usuário antes de solicitar a impressão de um bilhete;

- total, armazena a quantia total de dinheiro inserida na máquina por todos os usuários desde que o objeto da máquina foi construído (excluindo qualquer saldo atual). A ideia é que, quando um bilhete é impresso, qualquer dinheiro no saldo é transferido para o total.

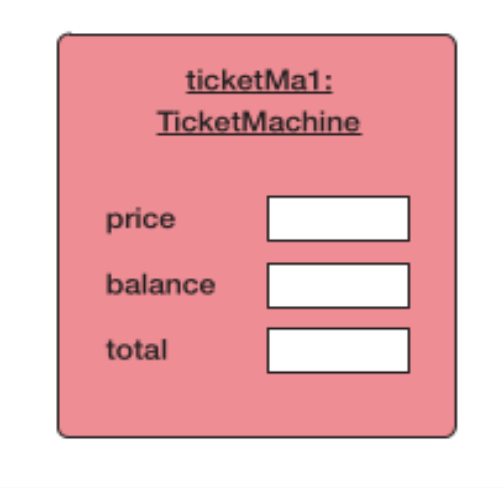
Código 2.3

```
public class TicketMachine
{
    private int price;
    private int balance;
    private int total;

    Constructors and methods omitted.
}
```

Os campos são pequenas quantidades de espaço dentro de um objeto que podem ser usados para armazenar dados persistentemente. Todo objeto terá espaço para cada campo declarado em sua classe. A Figura 2.2, mostra uma representação esquemática de um objeto bilheteira, com seus três campos. Os campos ainda não receberam nenhum valor; uma vez que eles tenham, podemos escrever cada valor na caixa que representa o campo. A notação é semelhante à usada no BlueJ para mostrar objetos no banco de objetos, exceto que mostramos um pouco mais detalhadamente aqui. No BlueJ, por motivos de espaço, os campos não são exibidos no ícone do objeto. No entanto, podemos vê-los abrindo uma janela do inspetor (Seção 1.7).

Figura 2.2 - Objeto de máquina de *ticket*



Fonte: BlueJ.

Cada campo tem sua própria declaração no código fonte. Na linha acima de cada um, na definição de classe completa, adicionamos uma única linha de texto - um comentário - para benefício dos leitores humanos da definição de classe:

// O preço de um bilhete desta máquina.

private int price;

Um comentário de linha única é introduzido pelos dois caracteres “//”, escritos sem espaços entre eles. Comentários mais detalhados, geralmente com várias linhas, são escritos na forma de comentários com várias linhas. Eles começam com o par de caracteres "/ *" e terminam com o par "* /". Há um bom exemplo que precede o cabeçalho da classe no Código 2.1.

As definições dos três campos são bastante semelhantes:

- Todas as definições indicam que são campos privados do objeto; teremos mais a dizer sobre o que isso significa no capítulo 6, mas, por enquanto, simplesmente diremos que sempre definimos os campos como privados.

- Todos os três campos são do tipo *int*, que é outra palavra-chave e representa o tipo de dados de número inteiro. Isso indica que cada um pode armazenar um único valor de número inteiro, o que é razoável, uma vez que desejamos que eles armazenem números que representam quantias de dinheiro em centavos.

Os campos podem armazenar valores que podem variar ao longo do tempo, portanto, eles também são conhecidos como variáveis. O valor armazenado em um campo pode ser alterado de seu valor inicial, se necessário. Por exemplo, à medida que mais dinheiro é inserido em uma máquina de *tickets*, queremos alterar o valor armazenado no campo de saldo. É comum ter campos cujos valores mudam frequentemente, como saldo e total, e outros que mudam raramente ou não, como preço. O fato de o valor do preço não variar, uma vez definido, não altera o fato de ainda ser chamado de variável. Nas seções a seguir, também conheceremos outros tipos de variáveis além dos campos, mas todos compartilharão o mesmo objetivo fundamental de armazenar dados. Os campos preço, saldo e total são todos os itens de dados que um objeto da máquina de tickets precisa para cumprir sua função de receber dinheiro de um cliente, imprimir tickets e manter um total de todo o dinheiro que foi investido nele. Nas seções a seguir, veremos como o construtor e os métodos usam esses campos para implementar o comportamento de máquinas de ticket simples.

A partir das definições de campos que vimos até agora, podemos começar a montar um padrão que será aplicado sempre que definirmos uma variável de campo em uma classe:

- Eles geralmente começam com a palavra reservada particular;
- Eles incluem um nome de tipo (como *int*, *String*, *Person* etc.);
- Eles incluem um nome escolhido pelo usuário para a variável de campo;
- Eles terminam com ponto e vírgula.

Lembrar desse padrão o ajudará quando você escrever suas próprias aulas.

De fato, ao examinarmos de perto o código fonte de diferentes classes, você verá padrões como este, surgindo repetidamente. Parte do processo de aprender a programar envolve olhar para esses padrões e usá-los em seus próprios programas. Essa é uma das razões pelas quais estudar o código fonte em detalhes é tão útil nesta fase.

2.4.2 Construtores

Os construtores têm um papel especial a cumprir. Eles são responsáveis por garantir que um objeto seja configurado corretamente quando é criado pela primeira vez; em outras palavras, para garantir que um objeto esteja pronto para ser usado imediatamente após sua criação. Esse processo de construção também é chamado de inicialização.

Um construtor é responsável por garantir que o novo objeto venha a existir adequadamente. Depois que um objeto é criado, o construtor não desempenha nenhum papel adicional na vida desse objeto e não pode ser chamado novamente. O código 2.4 mostra o construtor da classe *TicketMachine*.

Uma das características distintivas dos construtores é que eles têm o mesmo nome da classe em que são definidos - neste caso, *TicketMachine*. O nome do construtor segue imediatamente a palavra público, sem nada no meio.

Devemos esperar uma conexão estreita entre o que acontece no corpo de um construtor e os campos da classe. Isso ocorre porque uma das principais funções do construtor é inicializar os campos. Em alguns campos, como saldo e total, será possível definir valores iniciais sensíveis atribuindo um número constante - neste caso, zero. Com outras pessoas, como o preço do ingresso, não é tão simples, pois não sabemos o preço que os ingressos de uma determinada máquina terão até que essa máquina seja construída. Lembre-se de que podemos criar vários objetos de máquina para vender bilhetes com preços diferentes, para que nenhum preço inicial esteja sempre correto. Ao experimentar a criação de objetos da Máquina de *tickets* no BlueJ, você precisará fornecer o custo dos *tickets* sempre que criar uma nova máquina de *tickets*. Um ponto importante a ser observado aqui é que o preço de um *ticket* é inicialmente determinado externamente e deve ser repassado ao construtor. No BlueJ, você decide o valor e o insere em uma caixa de diálogo. Parte da tarefa do construtor é receber esse valor e armazená-lo no campo de preço da máquina de *tickets* recém-criada, para que a máquina possa lembrar qual era esse valor sem que você precise lembrá-lo.

Código 2.4

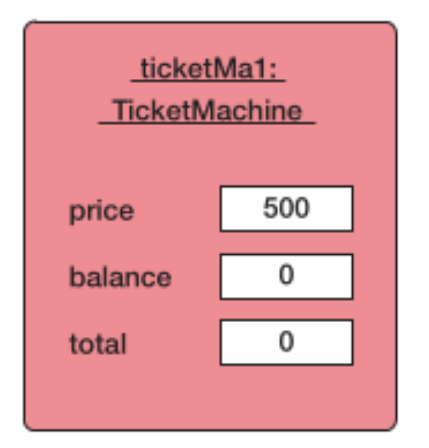
```
public class TicketMachine
{
    Fields omitted.
    /**
     * Create a machine that issues tickets of the given price.
     * Note that the price must be greater than zero, and there
     * are no checks to ensure this.
     */
    public TicketMachine(int cost)
    {
        price = cost;
        balance = 0;
        total = 0;
    }
    Methods omitted.
}
```

Podemos ver com isso, que uma das funções mais importantes de um campo é lembrar informações externas passadas para o objeto, para que essas informações estejam disponíveis para um objeto durante toda a sua vida útil. Os campos, portanto, fornecem um local para armazenar dados duradouros (ou seja, persistentes).

A Figura 2.3, mostra um objeto de máquina de *tickets* após a execução do construtor. Os valores agora foram atribuídos aos campos. Neste diagrama, podemos dizer que a máquina de *tickets* foi criada passando 500, como o valor do preço do *ticket*.

Nota: em Java, todos os campos são inicializados automaticamente com um valor padrão se não forem explicitamente inicializados. Para campos inteiros, esse valor padrão é zero. Portanto, estritamente falando, poderíamos ter feito sem definir o saldo e o total como zero, contando com o valor padrão para nos dar o mesmo resultado. No entanto, preferimos escrever as atribuições explícitas de qualquer maneira. Não há desvantagem e serve bem para documentar o que realmente está acontecendo. Não confiamos que um leitor da classe saiba qual é o valor padrão e documentamos que realmente queremos que esse valor seja zero e não esquecemos de inicializá-lo.

Figura 2.3 - Um objeto de máquina de *tickets* após a execução do construtor



Fonte: BlueJ.

2.5 Parâmetros: recebendo dados

Construtores e métodos, desempenham papéis bastante diferentes na vida de um objeto, mas a maneira pela qual ambos recebem valores de fora é a mesma: via parâmetros. Você deve se lembrar que encontramos brevemente parâmetros na aula 01 (Seção 1.4). Os parâmetros são outro tipo de variável, assim como os campos, e também são usados para armazenar dados. Parâmetros são variáveis definidas no cabeçalho de um construtor ou método: *public TicketMachine (int cost).*

Esse construtor possui um único parâmetro, *cost*, que é do tipo *int* - o mesmo tipo que o campo de preço, que será usado para definir. Um parâmetro é usado como uma espécie de mensageiro temporário, transportando dados originados de fora do construtor ou método e disponibilizando-os dentro dele.

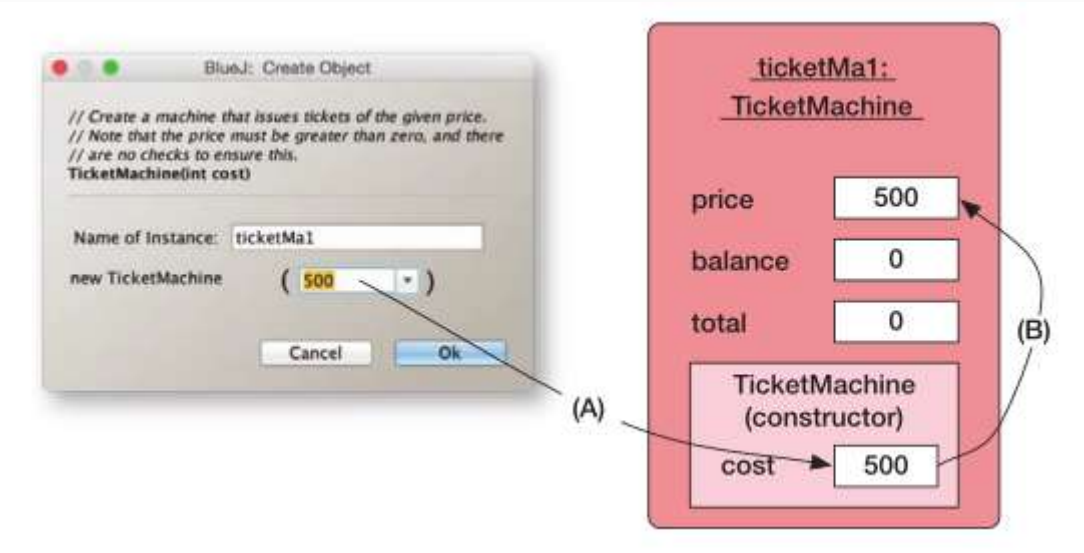
A Figura 2.4 ilustra como os valores são passados por meio de parâmetros. Nesse caso, um usuário BlueJ insere o valor externo na caixa de diálogo ao criar uma nova máquina de *tickets* (mostrada à esquerda) e esse valor é copiado no parâmetro *cost* do construtor da nova máquina. Isso é ilustrado com a seta rotulada (A). A caixa no objeto *TicketMachine* na Figura 2.4, denominada "*TicketMachine* (construtor)", representa espaço adicional para o objeto que é criado apenas quando o construtor é executado. Vamos chamá-lo de espaço construtor do objeto (ou espaço de método quando falamos de métodos, ao invés de construtores, pois a situação é a mesma). O espaço do construtor é usado para fornecer lugar a fim de armazenar os valores para os parâmetros do construtor. Em nossos diagramas, todas as variáveis são representadas por caixas brancas.

Distinguimos entre os nomes dos parâmetros dentro de um construtor ou método, e os valores dos parâmetros fora referindo-se aos nomes como parâmetros formais e aos valores como parâmetros reais. Portanto, o custo é um parâmetro formal e um valor fornecido pelo usuário, como 500, é um parâmetro real.

Um parâmetro formal está disponível para um objeto apenas dentro do corpo de um construtor ou método que o declara. Dizemos que o escopo de um parâmetro é restrito ao corpo do construtor ou método no qual ele é declarado. Por outro lado, o escopo de um campo é toda a definição de classe - ele pode ser acessado de qualquer lugar da mesma classe. Essa é uma diferença muito importante entre esses dois tipos de variáveis.

Um conceito relacionado ao escopo variável é vida útil variável. A vida útil de um parâmetro é limitada a uma única chamada de um construtor ou método. Quando um construtor ou método é chamado, o espaço extra para as variáveis de parâmetro é criado e os valores externos copiados para esse espaço. Depois que a chamada termina sua tarefa, os parâmetros formais desaparecem e os valores que eles mantêm são perdidos. Em outras palavras, quando o construtor termina a execução, todo o espaço do construtor é removido, juntamente com as variáveis de parâmetro contidas nele (veja a Figura 2.4).

Figura 2.4 - todo o espaço do construtor é removido, juntamente com as variáveis de parâmetro contidas nele



Fonte: BlueJ.

Por outro lado, a vida útil de um campo é igual à vida útil do objeto ao qual ele pertence. Quando um objeto é criado, o mesmo acontece com os campos e eles persistem por toda a vida útil do objeto. Segue-se que, se quisermos lembrar o custo dos tickets retidos no parâmetro *cost*, devemos armazenar o valor em algum lugar persistente - ou seja, no campo *price*.

Assim como esperamos ver um vínculo estreito entre um construtor e os campos de sua classe, esperamos ver um vínculo estreito entre os parâmetros do construtor e os campos, porque valores externos geralmente serão necessários para definir os valores iniciais de um ou mais desses campos. Onde for esse o caso, os tipos de parâmetro corresponderão estreitamente aos tipos dos campos correspondentes.

2.5.1 Escolhendo nomes de variáveis

Uma das coisas que você deve ter notado é que os nomes de variáveis que usamos para campos e parâmetros têm uma estreita conexão com o objetivo da variável. Nomes como preço, custo, título e ativo, informam algo útil sobre as informações armazenadas nessa variável. Isso facilita a compreensão do que está acontecendo no programa. Dado que temos um alto grau de liberdade na escolha de nomes de variáveis, vale a pena seguir esse princípio de escolher nomes que comuniquem um senso de propósito, ao invés vez de combinações arbitrárias e sem sentido de letras e números.

2.6 Atribuição

Na seção anterior, observamos a necessidade de copiar o valor de curta duração, armazenado em uma variável de parâmetro para um local mais permanente - uma variável de campo. Para fazer isso, o corpo do construtor contém a seguinte instrução de atribuição:

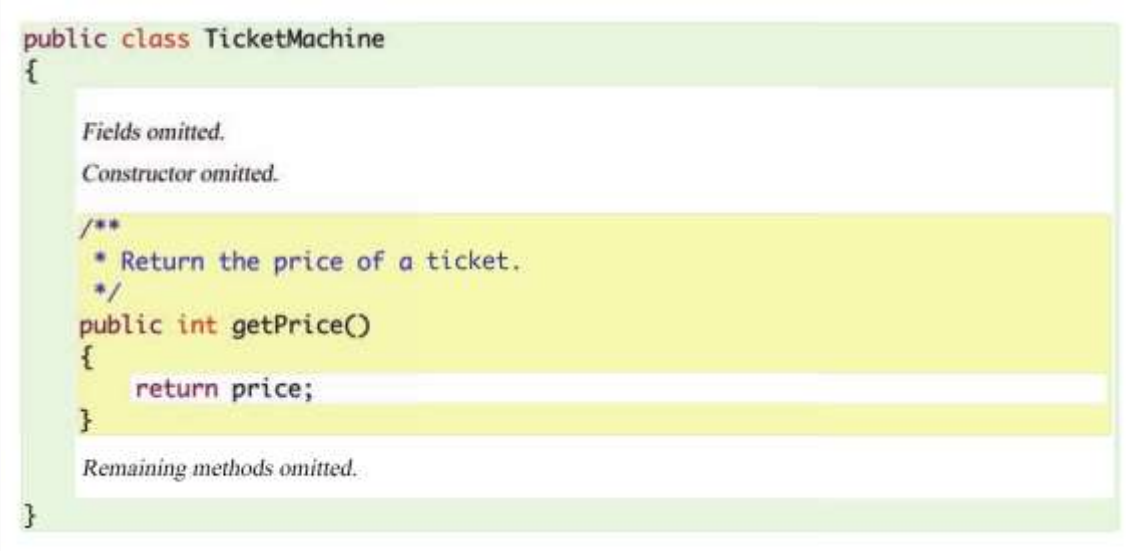
preço = custo;

As instruções de atribuição são usadas frequentemente na programação, como um meio de armazenar um valor em uma variável. Eles podem ser reconhecidos pela presença de um operador de atribuição, como "=" no exemplo acima. As instruções de atribuição funcionam pegando o valor do que aparece no lado direito do operador e copiando esse valor na variável no lado esquerdo. Isso é ilustrado na Figura 2.4, pela seta rotulada (B). O lado direito é chamado de expressão. Em sua forma mais geral, expressões são coisas que computam valores, mas, neste caso, a expressão consiste em apenas uma variável única, cujo valor é copiado na variável *price*. Veremos exemplos de expressões mais complicadas mais adiante nesta aula. Uma regra sobre as instruções de atribuição é que o tipo da expressão no lado direito deve corresponder ao tipo da variável à qual seu valor está atribuído. Já conhecemos três tipos diferentes e comumente usados: *int*, *String* e (muito brevemente) booleano. Essa regra significa que não temos permissão para armazenar uma expressão *int-type* em uma variável *String -type*, por exemplo. Essa mesma regra também se aplica entre parâmetros formais e parâmetros reais: o tipo de uma expressão de parâmetro real deve corresponder ao tipo da variável de parâmetro formal. Por enquanto, podemos dizer que os tipos de ambos devem ser os mesmos, embora veremos nos capítulos posteriores que essa não é toda a verdade.

2.7 Métodos

A classe *TicketMachine* possui quatro métodos: *getPrice*, *getBalance*, *insertMoney* e *printTicket*. Você pode vê-los no código fonte da turma (código 2.1), como caixas amarelas. Começaremos nossa análise do código fonte dos métodos considerando o *getPrice* (código 2.5).

Código 2.5



```
public class TicketMachine
{
    Fields omitted.
    Constructor omitted.
    /**
     * Return the price of a ticket.
     */
    public int getPrice()
    {
        return price;
    }
    Remaining methods omitted.
}
```

Os métodos têm duas partes: um cabeçalho e um corpo. Aqui está o cabeçalho do método para *getPrice*, precedido por um comentário descritivo:

```
/**
* Return the price of a ticket (Retorne o preço de um ticket).
* /
public int getPrice ()
```

É importante distinguir entre cabeçalhos de método e declarações de campo, porque eles podem parecer bastante semelhantes. Podemos dizer que *getPrice* é um método e não um campo, porque os cabeçalhos dos métodos sempre incluem parênteses - “(” e “)” - e nenhum ponto e vírgula no final do cabeçalho.

O corpo do método é o restante do método após o cabeçalho. É sempre delimitada por um par de colchetes: "{" e "}". Os corpos do método contêm as declarações e instruções que definem o que um objeto faz quando esse método é chamado. As declarações são usadas para criar espaço variável temporário adicional, enquanto as instruções descrevem as ações do método. Em *getPrice*, o corpo do método contém uma única instrução, mas em breve veremos exemplos em que o corpo do método consiste em muitas linhas de declarações e declarações.

Qualquer conjunto de declarações e declarações entre um par de colchetes correspondentes é conhecido como um bloco. Portanto, o corpo da classe *TicketMachine*, os corpos do construtor e todos os métodos dentro da classe são blocos.

Há pelo menos duas diferenças significativas entre os cabeçalhos do construtor *TicketMachine* e o método *getPrice*:

public TicketMachine (*custo int*)

public int getPrice ()

- O método possui um tipo de retorno *int*; o construtor não tem tipo de retorno. Um tipo de retorno é gravado antes do nome do método. Essa é uma diferença que se aplica a todos os casos.

- O construtor tem um único parâmetro formal, *cost*, mas o método não possui - apenas um par de parênteses vazios. Essa é uma diferença que se aplica nesse caso específico.

É uma regra em Java que um construtor não tenha um tipo de retorno. Por outro lado, os construtores e os métodos, podem ter qualquer número de parâmetros formais, incluindo nenhum. Dentro do corpo de *getPrice*, há uma única instrução:

return price;

Isso é chamado de declaração de retorno. É responsável por retornar um valor inteiro para corresponder ao tipo de retorno *int* no cabeçalho do método. Onde um método contém uma declaração de retorno, é sempre a declaração final desse método, porque nenhuma outra declaração no método será executada depois que a declaração de retorno for executada.

Tipos de retorno e as instruções de retorno trabalham juntos. O tipo *int return* de *getPrice* é uma forma de promessa de que o corpo do método fará algo que, em última análise, resulta em um valor inteiro sendo calculado e retornado como o resultado do método. Você pode pensar em uma chamada de método como uma forma de pergunta para um objeto e o valor de retorno do método, como a resposta do objeto para essa pergunta. Nesse caso, quando o método *getPrice* é chamado em uma máquina de ticket, a pergunta é: quanto custa o *ticket*? Uma máquina de bilhetes não precisa executar nenhum cálculo para poder responder a isso, porque mantém a resposta em seu campo de preços. Portanto, o método responde apenas retornando o valor dessa variável. À medida que gradualmente desenvolvemos classes mais complexas, encontraremos inevitavelmente questões mais complexas, que exigem mais trabalho para fornecer suas respostas.

2.8 Métodos acessadores ("*getters*") e mutadores ("*setters*")

Geralmente, descrevemos métodos como os dois métodos "*get*" do *TicketMachine* (*getPrice* e *getBalance*) como métodos de acessador (ou apenas acessadores). Isso ocorre, porque eles retornam informações ao chamador sobre o estado de um objeto; eles fornecem acesso a informações sobre o estado do objeto. Um acessador geralmente contém uma declaração de retorno para retornar essas informações.

Muitas vezes há confusão sobre o que "retornar um valor" realmente significa na prática. As pessoas costumam pensar que isso significa que algo é impresso pelo programa. Não é esse o caso - veremos como a impressão é feita quando analisamos o método *printTicket*. Ao invés disso, retornar um valor, significa que algumas informações são passadas internamente entre duas partes diferentes do programa. Uma parte do programa solicitou informações de um objeto por meio de uma chamada de método, e o valor de retorno é o modo como o objeto repassa essas informações para o chamador. Os métodos *get* de uma máquina de tickets executam tarefas semelhantes: retornando o valor de um dos campos de seus objetos. Os métodos restantes - *insertMoney* e *printTicket* têm uma função muito mais significativa, principalmente porque alteram o valor de um ou mais campos de um objeto de máquina de tickets, cada vez que são chamados. Chamamos métodos que alteram o estado de seus métodos de mutadores de objetos (ou apenas mutadores).

Da mesma maneira que pensamos em uma chamada para um acessador como uma solicitação de informações (uma pergunta), podemos pensar em uma chamada para um mutador como uma solicitação de um objeto para alterar seu estado. A forma mais básica de mutador é aquela que usa um único parâmetro cujo valor é usado para substituir diretamente o que é armazenado em um dos campos de um objeto. Em um complemento direto aos métodos "*get*", esses são frequentemente chamados de métodos "*set*", embora o *TicketMachine* não possua nenhum deles, neste estágio.

Um efeito distintivo de um mutador é que um objeto geralmente exibe um comportamento ligeiramente diferente antes e depois de ser chamado.

O cabeçalho de *insertMoney* possui um tipo de retorno nulo e um único parâmetro formal, quantidade, do tipo *int*. Um tipo de retorno nulo significa que o método não retorna nenhum valor ao seu chamador. Isso é significativamente diferente de todos os outros tipos de retorno. No BlueJ, a diferença é mais perceptível, pois nenhum diálogo de valor de retorno é mostrado após uma chamada para um método nulo. Dentro do corpo de um método nulo, essa diferença se reflete no fato de que não há declaração de retorno.

Código 2.6

```
/**
 * Receive an amount of money from a customer.
 */
public void insertMoney(int amount)
{
    balance = balance + amount;
}
```

No corpo do *insertMoney*, há uma única instrução que é outra forma de instrução de atribuição. Sempre consideramos as declarações de atribuição examinando primeiro o cálculo no lado direito do símbolo de atribuição. Aqui, seu efeito é calcular um valor que seja a soma do número no parâmetro de quantidade e o número no campo de saldo. Esse valor combinado é então atribuído ao campo de saldo. Portanto, o efeito é aumentar o valor em equilíbrio pelo valor em quantidade.

2.9 Imprimir a partir de métodos

O código 2.7 mostra o método mais complexo da classe: *printTicket*. Para ajudar você a entender a discussão a seguir, certifique-se de ter chamado esse método em uma máquina de *tickets*. Você verá a seguinte mensagem na janela do terminal do BlueJ:

```
##################
# The BlueJ Line
# Ticket
# 500 cents.
##################
```

Este é o método mais longo que vimos até agora, portanto, o dividiremos em partes mais gerenciáveis:

- O cabeçalho indica que o método possui um tipo de retorno nulo e que não requer parâmetros.

- O corpo é composto por oito declarações e comentários associados.

- As seis primeiras instruções são responsáveis por imprimir o que você vê na janela do terminal BlueJ: cinco linhas de texto e uma sexta linha em branco.

- O sétimo extrato adiciona o saldo inserido pelo cliente (através de chamadas anteriores para *insertMoney*) ao total corrente de todo o dinheiro arrecadado, até agora pela máquina.

- O oitavo extrato redefine o saldo com um extrato básico de atribuição, em preparação para o próximo cliente.

Código 2.7

```
/**
 * Print a ticket.
 * Update the total collected and reduce the balance to zero.
 */
public void printTicket()
{
    // Simulate the printing of a ticket.
    System.out.println("##################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + price + " cents.");
    System.out.println("##################");
    System.out.println();

    // Update the total collected with the balance.
    total = total + balance;
    // Clear the balance.
    balance = 0;
}
```

Ao comparar a saída que aparece com as declarações que a produziram, é fácil ver que uma declaração como: System.out.println ("# The BlueJ Line"); literalmente imprime a sequência que aparece entre o par correspondente de caracteres de aspas duplas. A forma básica de uma chamada para *println* é *System.out.println* (“algo que queremos imprimir”); onde algo que queremos imprimir pode ser substituído por qualquer sequência arbitrária, entre um par de

caracteres de aspas duplas. Por exemplo, não há nada significativo no caractere "#" que está na *string* - é simplesmente um dos caracteres que desejamos ser impressos. Todas as instruções de impressão no método *printTicket* são chamadas para o método *println* do objeto *System.out* que é incorporado à linguagem Java, e o que aparece entre colchetes é o parâmetro para cada chamada de método, conforme o esperado.

No entanto, na quarta instrução, o parâmetro real para *println* é um pouco mais complicado e requer mais explicações: System.out.println ("# " + price + " centavos");

O que ele faz é imprimir o preço do bilhete, com alguns caracteres extras nos dois lados do valor. Os dois operadores "+" estão sendo usados para construir um único parâmetro real, na forma de uma sequência, a partir de três componentes separados:

- A *string* literal: "#" (observe o caractere de espaço após o *hash*);

- O valor do campo de *price* (observe que não há aspas no nome do campo porque queremos o valor do campo, não o nome);

- A *string* literal: "centavos" (observe o caractere de espaço antes da palavra "centavos").

Quando usado entre uma sequência e qualquer outra coisa, "+" é um operador de concatenação de sequências (ou seja, concatena ou une sequências para criar uma nova sequência), em vez de um operador de adição aritmética. Portanto, o valor numérico do preço é convertido em uma sequência e unido às duas sequências adjacentes.

Observe que a chamada final para *println* não contém parâmetro de *string*. Isso é permitido, e o resultado da chamada será deixar uma linha em branco entre essa saída e qualquer outra que se segue. Você verá facilmente a linha em branco se imprimir um segundo *ticket*.

2.10 Resumo do método

Vale a pena resumir alguns recursos dos métodos neste momento, porque os métodos são fundamentais para os programas que iremos escrever e explorar neste material. Eles implementam as ações principais de cada objeto.

Um método com parâmetros receberá dados transmitidos a ele do chamador do método e, em seguida, usará esses dados para ajudá-lo a executar uma tarefa específica. No entanto, nem todos os métodos aceitam parâmetros; muitos simplesmente usam os dados armazenados nos campos do objeto para realizar sua tarefa.

Se um método tiver um tipo de retorno nulo, ele retornará alguns dados para o local de origem - e esses dados certamente serão usados no chamador para cálculos ou manipulações de programa. Muitos métodos, no entanto, têm um tipo de retorno nulo e não retornam nada, mas ainda executam uma tarefa útil dentro do contexto de seu objeto. Os métodos de acesso têm tipos de retorno não nulos e retornam informações sobre o estado do objeto. Os métodos mutadores modificam o estado de um objeto. Os mutadores geralmente usam parâmetros cujos valores são usados na modificação, embora ainda seja possível escrever um método de mutação que não aceita parâmetros.

2.11 Resumo da máquina de ingresso simples

Agora examinamos a estrutura interna da classe *TicketMachine* com mais detalhes. Vimos que a classe tem uma pequena camada externa que dá nome à classe e um corpo interno mais substancial, contendo campos, um construtor e vários métodos. Os campos são usados para armazenar dados que permitem que os objetos mantenham um estado que persiste entre chamadas de método. Construtores são usados para configurar um estado inicial quando um objeto é criado. Ter um estado inicial adequado permitirá que um objeto responda adequadamente a chamadas de método imediatamente após sua criação. Os métodos implementam o comportamento definido dos objetos da classe. Os métodos assessores fornecem informações sobre o estado de um objeto e os métodos mutadores alteram o estado de um objeto.

Vimos que os construtores se distinguem dos métodos por terem o mesmo nome da classe em que são definidos. Tanto os construtores quanto os métodos podem aceitar parâmetros, mas apenas métodos podem ter um tipo de retorno. Tipos de retorno não nulos nos permitem passar um valor de um método para o local de onde o método foi chamado. Um método com um tipo de retorno não nulo deve ter pelo menos uma declaração de retorno em seu corpo; essa será frequentemente a declaração final. Os construtores nunca têm um tipo de retorno de nenhum tipo - nem mesmo nulo.

2.12 Refletindo sobre o projeto da máquina de bilhetes

Do nosso estudo sobre os componentes internos da classe *TicketMachine*, você deveria ter percebido o quão inadequado seria no mundo real. É deficiente de várias maneiras:

- Não contém verificação de que o cliente inseriu dinheiro suficiente para pagar um bilhete;
- Não reembolsa o dinheiro se o cliente paga muito por um ingresso;
- Não verifica se o cliente insere quantias sensatas de dinheiro. Experimente o que acontece se um valor negativo for inserido, por exemplo;
- Não verifica se o preço do bilhete passado ao seu construtor é sensato.

Se pudéssemos resolver esses problemas, teríamos um software muito mais funcional que serviria de base para a operação de uma máquina de bilhetes do mundo real. Nas próximas seções, examinaremos a implementação de uma classe aprimorada de máquinas de *tickets* que tenta lidar com algumas das inadequações da implementação simples. Abra o projeto *Better-Ticket-Machine*. Como antes, este projeto contém uma única classe: *TicketMachine*. Antes de examinar os detalhes internos da classe, experimente criar algumas instâncias e veja se você percebe alguma diferença de comportamento entre esta versão e a versão anterior. Uma diferença específica é que a nova versão possui um método adicional, reembolso de saldo. Veja o que acontece quando você o chama.

Código 2.8

```
/**
* TicketMachine models a ticket machine that issues
* flat-fare tickets.
* The price of a ticket is specified via the constructor.
* Instances will check to ensure that a user only enters
* sensible amounts of money, and will only print a ticket
* if enough money has been input.
*
* @author David J. Barnes and Michael Kolling
* @version 2008.03.30
*/
public class TicketMachine
{
   // The price of a ticket from this machine.
   private int price;
   // The amount of money entered by a customer so far.
   private int balance;
   // The total amount of money collected by this machine.
   private int total;


   /**
    * Create a machine that issues tickets of the given price.
    */
   public TicketMachine(int ticketCost)
   {
      price = ticketCost;
      balance = 0;
      total = 0;
   }
```

```
/**
 * @Return The price of a ticket.
 */
public int getPrice()
{
   return price;
}


/**
 * Return The amount of money already inserted for the
 * next ticket.
 */
public int getBalance()
{
   return balance;
}
```

```
/**
 * Receive an amount of money in cents from a customer.
 * Check that the amount is sensible.
 */
public void insertMoney(int amount)
{
    if(amount > 0) {
        balance = balance + amount;
    }
    else {
        System.out.println("Use a positive amount: " +
                           amount);
    }
}


/**
 * Print a ticket if enough money has been inserted, and
 * reduce the current balance by the ticket price. Print
 * an error message if more money is required.
 */
public void printTicket()
{
    if(balance >= price) {
        // Simulate the printing of a ticket.
        System.out.println("####################");
        System.out.println("# The BlueJ Line");
        System.out.println("# Ticket");
        System.out.println("# " + price + " cents.");
        System.out.println("####################");
        System.out.println();
```

```
        // Update the total collected with the price.
        total = total + price;
        // Reduce the balance by the prince.
        balance = balance - price;
    }
    else {
        System.out.println("You must insert at least: " +
                           (price - balance) + " more cents.");

    }
}


/**
 * Return the money in the balance.
 * The balance is cleared.
 */
public int refundBalance()
{
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
}
```

2.13 Fazendo escolhas: a declaração condicional

O código 2.8 mostra os detalhes internos da melhor definição de classe da bilheteria. Grande parte dessa definição já lhe será familiar em nossa discussão sobre a máquina de bilhetes simples. Por exemplo, o invólucro externo que nomeia a classe é o mesmo, porque escolhemos atribuir o mesmo nome a essa classe. Além disso, ele contém os mesmos três campos para manter o estado do objeto e esses foram declarados da mesma maneira. O construtor e os dois métodos *get* também são os mesmos de antes.

A primeira alteração significativa pode ser vista no método *insertMoney*. Reconhecemos que o principal problema da bilheteria simples era a falha em verificar determinadas condições. Um desses cheques ausentes estava na quantidade de dinheiro inserida por um cliente, pois era possível inserir uma quantidade negativa de dinheiro. Corrigimos essa falha usando uma declaração condicional para verificar se o valor inserido tem um valor maior que zero:

```
if (quantidade > 0) {
  saldo = saldo + valor;
}
else {
   System.out.println ("Use uma quantidade positiva em vez de:" + montante);
}
```

Instruções condicionais também são conhecidas como instruções, "*if* - *else*", da palavra-chave usada na maioria das linguagens de programação para apresentá-las. Uma declaração condicional nos permite executar uma das duas ações possíveis, com base no resultado de uma verificação ou teste. Se o teste for verdadeiro, faremos uma coisa; caso contrário, fazemos algo diferente. Esse tipo de decisão deve ser familiar de situações da vida cotidiana: por exemplo, se eu tiver dinheiro suficiente, sairei para uma refeição; caso contrário, ficarei em casa e assistirei um filme. Uma instrução condicional tem a forma geral descrita no pseudocódigo a seguir:

```
if (execute algum teste que dê um resultado verdadeiro ou falso) {
   Faça as instruções aqui se o teste deu um resultado verdadeiro
}
else {
   Faça as instruções aqui se o teste deu um resultado falso
}
```

Certas partes desse pseudocódigo são bits apropriados do Java, e elas aparecerão em quase todas as instruções condicionais - as palavras-chave *if and else*, os colchetes ao redor do teste e os colchetes marcando os dois blocos - enquanto os outros três estão em itálico as partes serão aprimoradas de maneira diferente para cada situação específica que está sendo codificada.

Somente um dos dois blocos de declarações, após o teste, será realizado posteriormente a avaliação do teste. Portanto, no exemplo do método *insertMoney*, após o teste de uma quantia

inserida, apenas adicionaremos a quantia ao saldo ou imprimimos a mensagem de erro. O teste usa o operador maior que, ">", para comparar o valor em quantidade contra zero. Se o valor for maior que zero, ele será adicionado à balança. Se não for maior que zero, uma mensagem de erro será impressa. Usando uma declaração condicional, protegemos a alteração para equilibrar no caso em que o parâmetro não representa um valor válido. Os mais óbvios a serem mencionados neste momento são "<" (menor que), "<=" (menor ou igual a) e "> =" (maior que ou igual a). Todos são usados para comparar dois valores numéricos, como no método *printTicket*.

O teste usado em uma instrução condicional é um exemplo de expressão booleana. No início deste capítulo, introduzimos expressões aritméticas que produziram resultados numéricos. Uma expressão booleana possui apenas dois valores possíveis (verdadeiro ou falso): o valor da quantidade é maior que zero (verdadeiro) ou não é maior (falso). Uma declaração condicional utiliza esses dois valores possíveis para escolher entre duas ações diferentes.

2.14 Mais um exemplo de declaração condicional

O método *printTicket* contém um exemplo adicional de uma declaração condicional. Aqui está o esboço:

```
if (balance >= price) {
  ...Detalhes de impressão omitidos.
   // Atualize o total coletado com o preço.
  total = total + price;
  // Reduza o saldo pelo preço.
  balance = balance - price;
}
else {
  System.out.println ("Você deve inserir pelo menos:" + (price - balance) + " mais centavos.");
}
```

Com essa declaração *if*, corrigimos o problema de que a versão simples não verifica se um cliente inseriu dinheiro suficiente para um ticket antes da impressão. Esta versão verifica se o valor no campo de saldo é pelo menos tão grande quanto o valor no campo de preço. Se estiver, não há problema em imprimir um *ticket*. Caso contrário, imprimimos uma mensagem de erro.

A impressão da mensagem de erro segue exatamente o mesmo padrão que vimos para imprimir o preço dos ingressos no método *printTicket*; é apenas um pouco mais detalhado.

System.out.println ("Você deve inserir pelo menos:" + (*price* - *balance*) + "mais centavos.");

O único parâmetro real para o método *println* consiste em uma concatenação de três elementos: duas literais de sequência de caracteres em ambos os lados de um valor numérico. Nesse caso, o valor numérico é uma subtração que foi colocada entre parênteses para indicar que é o valor resultante que queremos concatenar com as duas *strings*.

O método *printTicket* reduz o valor do saldo pelo valor do preço. Como consequência, se um cliente inserir mais dinheiro que o preço do ingresso, algum dinheiro será deixado na balança que poderia ser usada no preço de um segundo ingresso. Como alternativa, o cliente pode solicitar o reembolso do saldo remanescente, e é isso que o método *refundBalance* faz, como veremos na seção 2.16.

2.15 Destaque do escopo

Você já deve ter notado que o editor BlueJ exibe o código fonte com alguma decoração adicional: caixas coloridas em torno de alguns elementos, como métodos e instruções *if* (consulte, por exemplo, o Código 2.8).

Essas anotações coloridas são conhecidas como destaque do escopo e ajudam a esclarecer as unidades lógicas do seu programa. Um escopo (também chamado de bloco) é uma unidade de código normalmente indicada por um par de colchetes. O corpo inteiro de uma classe é um escopo, assim como o corpo de cada método e as partes *if* e *else* de uma instrução *if*.

Como você pode ver, os escopos são frequentemente aninhados: a instrução if está dentro de um método, dentro de uma classe. O BlueJ ajuda a distinguir diferentes tipos de escopos com cores diferentes. Um dos erros mais comuns no código dos programadores iniciantes é errar os colchetes - colocando-os no lugar errado ou com um colchete ausente por completo. Duas coisas podem ajudar muito a evitar esse tipo de erro:

- Preste atenção ao recuo adequado do seu código. Sempre que um novo escopo for iniciado (após um colchete aberto), indente o código a seguir em mais um nível. Fechar o escopo traz de volta o recuo. Se o seu recuo estiver completamente esgotado, use a função "Layout automático" do BlueJ (encontre-o no menu do editor) para corrigi-lo;

- Preste atenção ao destaque do escopo. Você vai se acostumar à aparência do código bem estruturado. Tente remover um colchete no editor ou adicionar um em um local arbitrário e observe como a coloração muda. Acostume-se a reconhecer quando os escopos parecem errados.

2.16 Variáveis locais

Até agora, encontramos dois tipos diferentes de variáveis: campos (variáveis de instância) e parâmetros. Agora vamos introduzir um terceiro tipo. Todos têm em comum o armazenamento de dados, mas cada tipo de variável tem um papel específico a desempenhar.

A Seção 2.7 observou que um corpo de método (ou, em geral, um bloco) pode conter declarações e declarações. Até este ponto, nenhum dos métodos que examinamos contém nenhuma declaração. O método *refundBalance* contém três instruções e uma única declaração.

A declaração introduz um novo tipo de variável:

```
public int refundBalance ()
{
  int amountToRefund;
  amountToRefund = balance;
  balance = 0;
  return amountToRefund;
}
```

Que tipo de variável é *amountToRefund*? Sabemos que não é um campo, porque os campos são definidos fora dos métodos. Também não é um parâmetro, pois esses são sempre definidos no cabeçalho do método. A variável *amountToRefund* é o que é conhecida como variável local, porque é definida dentro de um corpo de método.

As declarações de variáveis locais são semelhantes às declarações de campo, mas elas nunca têm privado ou público como parte delas. Os construtores também podem ter variáveis locais. Como parâmetros formais, as variáveis locais têm um escopo limitado às instruções do método ao qual pertencem. O tempo de vida útil deles é o tempo de execução do método: eles são criados quando um método é chamado, destruído quando um método é concluído. Você pode se perguntar, por que há necessidade de variáveis locais se temos campos? As variáveis locais são usadas principalmente como armazenamento temporário, para ajudar um único método a concluir sua tarefa; pensamos neles como armazenamento de dados para um único método. Por outro lado, os campos são usados para armazenar dados que persistem ao longo da vida de um objeto inteiro. Os dados armazenados nos campos são acessíveis a todos os métodos do objeto. Tentamos evitar declarar como campos variáveis os que realmente possuem apenas um uso local (no nível do método), cujos valores não precisam ser retidos além de uma única chamada de método. Portanto, mesmo que dois ou mais métodos da mesma classe usem variáveis locais para uma finalidade semelhante, não é apropriado defini-las como campos se seus valores não precisarem persistir além do final dos métodos.

No método *refundBalance*, *amountToRefund* é usado brevemente para reter o valor do saldo imediatamente, antes de este ser definido como zero. O método então retorna o valor lembrado da balança.

É bastante comum inicializar variáveis locais dentro de suas declarações. Para que pudéssemos abreviar as duas primeiras instruções de reembolso de saldo como: *int amountToRefund = balance;* mas ainda é importante ter em mente que existem duas etapas aqui: declarar a variável *amountToRefund* e atribuir um valor inicial a ela.

Uma variável local, com o mesmo nome que um campo, impedirá que o campo seja acessado de dentro de um construtor ou método. Consulte a Seção 3.13.2 para obter uma maneira de contornar isso, quando necessário.

O que você sabe sobre as declarações de retorno que ajudam a explicar por que esta versão não é compilada?

2.17 Campos, parâmetros e variáveis locais

Com a introdução de *amountToRefund* no método *refundBalance*, vimos agora três tipos diferentes de variáveis: campos, parâmetros formais e variáveis locais. É importante entender as semelhanças e diferenças entre esses três tipos. Aqui está um resumo de seus recursos:

- Todos os três tipos de variáveis são capazes de armazenar um valor apropriado aos seus tipos definidos. Por exemplo, um tipo definido de *int* permite que uma variável armazene um valor inteiro;
- Os campos são definidos fora dos construtores e métodos;
- Os campos são usados para armazenar dados que persistem ao longo da vida de um objeto. Como tal, eles mantêm o estado atual de um objeto. Eles têm uma vida útil que dura tanto quanto seu objeto;
- Os campos têm escopo de classe: eles são acessíveis em toda a classe, para que possam ser usados em qualquer um dos construtores ou métodos da classe em que estão definidos;
- Desde que sejam definidos como privados, os campos não podem ser acessados de qualquer lugar fora de sua classe de definição;
- Parâmetros formais e variáveis locais persistem apenas pelo período em que um construtor ou método executa. Sua vida útil é apenas enquanto uma única chamada, portanto, seus valores são perdidos entre as chamadas. Eles agem como locais de armazenamento temporário e não permanente;
- Parâmetros formais são definidos no cabeçalho de um construtor ou método. Eles recebem seus valores de fora, sendo inicializados pelos valores reais dos parâmetros que fazem parte da chamada do construtor ou do método.

Os parâmetros formais têm um escopo limitado ao construtor ou método definidor.

- Variáveis locais são definidas dentro do corpo de um construtor ou método. Eles podem ser inicializados e usados apenas dentro do corpo de seu construtor ou método de definição. Variáveis locais devem ser inicializadas antes de serem usadas em uma expressão - elas não recebem um valor padrão;
- Variáveis locais têm um escopo limitado ao bloco em que estão definidas. Eles não são acessíveis de qualquer lugar fora desse bloco.

Os programadores iniciantes, geralmente, acham difícil determinar se uma variável deve ser definida como um campo ou como uma variável local. Tentação é definir todas as variáveis como campos, porque elas podem ser acessadas de qualquer lugar da classe. De fato, a abordagem oposta é uma regra muito melhor a ser adotada: defina variáveis locais para um método, a menos que sejam claramente uma parte genuína do estado persistente de um objeto. Mesmo se você antecipar o uso da mesma variável em dois ou mais métodos, defina uma versão separada localmente para cada método até ter certeza absoluta de que a persistência entre chamadas de método é justificada.

2.18 Resumo da máquina de bilhetes melhorada

Ao desenvolver uma versão melhor da classe *TicketMachine*, conseguimos resolver as principais inadequações da versão simples. Ao fazer isso, introduzimos duas novas construções de linguagem: a instrução condicional e variáveis locais.

- Uma declaração condicional nos fornece um meio de executar um teste e, com base no resultado desse teste, executar uma ou outra de duas ações distintas;
- Variáveis locais nos permitem calcular e armazenar valores temporários dentro de um construtor ou método. Eles contribuem para o comportamento implementado pelo método de definição, mas seus valores são perdidos quando o construtor ou o método termina sua execução.

2.20 Revendo um exemplo familiar

A essa altura do capítulo, você já conheceu muitos conceitos novos. Para ajudar a reforçá-los, agora revisitaremos alguns em um contexto familiar. Ao longo do caminho, preste atenção a mais um ou dois novos conceitos que abordaremos em mais detalhes nas próximas aulas! Abra o projeto de aulas de laboratório que apresentamos na aula 01 e, em seguida, examine a classe *Student* no editor (código 2.9).

Código 2.9

```
/**
* The Student class represents a student in a student administration system.
* It holds the student details relevant in our context.
*
* @author Michael Kolling and David Barnes
* @version 2008.03.30
*/
public class Student
{
  // the student's full name
  private String name;
  // the student ID
  private String id;
  // the amount of credits for study taken so far
  private int credits;


  /**
   * Create a new student with a given name and ID number.
   */
  public Student(String fullName, String studentID)
  {
     name = fullName;
     id = studentID;
     credits = 0;
  }


  /**
   * Return the full name of this student.
   */
  public String getName()
  {
     return name;
```

```
    }

    /**
     * Set a new name for this student.
     */
    public void changeName(String replacementName)
    {
        name = replacementName;
    }

    /**
     * Return the student ID of this student.
     */
    public String getStudentID()
    {
        return id;
    }

    /**
     * Add some credit points to the student's accumulated credits.
     */
    public void addCredits(int additionalPoints)
    {
        credits += additionalPoints;
    }

    /**
     * Return the number of credit points this student has accumulated.
     */
    public int getCredits()
    {
        return credits;
    }

    /**
     * Return the login name of this student. The login name is a combination
     * of the first four characters of the student's name and the first three
     * characters of the student's ID number.
     */
    public String getLoginName()
    {
        return name.substring(0,4) + id.substring(0,3);
    }

    /**
     * Print the student's name and ID number to the output terminal.
     */
    public void print()
    {
        System.out.println(name + " (" + id + ")");
    }
}
```

Neste exemplo, as informações que desejamos armazenar para um aluno são: o nome, o ID do aluno e o número de créditos do curso que eles obtiveram até agora. Todas essas informações são persistentes durante o período em que estudam, mesmo que algumas delas sejam alteradas durante esse período (o número de créditos). Queremos armazenar essas informações nos campos, portanto, para representar o estado de cada aluno.

A classe contém três campos: nome (*name*), ID e créditos (*credits*). Cada um deles é inicializado no construtor único. Os valores iniciais dos dois primeiros são definidos a partir dos valores dos parâmetros passados para o construtor. Cada um dos campos tem um método associado de acessador *get*, mas apenas o nome e os créditos têm métodos mutadores associados. Isso significa que o valor de um campo de identificação permanece fixo após a construção do objeto. Se o valor de um campo não puder ser alterado depois de inicializado, dizemos que é imutável. Às vezes tornamos imutável o estado completo de um objeto depois que ele é construído; a classe *String* é um exemplo importante disso.

2.21 Métodos de chamada

O método *getLoginName* ilustra um novo recurso que vale a pena explorar:

```
public String getLoginName ()
{
    return name.substring (0,4) + id.substring (0,3);
}
```

Estamos vendo duas coisas em ação aqui:

- Chamando um método em outro objeto, em que o método retorna um resultado;
- Usando o valor retornado como resultado e como parte de uma expressão.

O nome e o *id* são objetos *String*, e a classe *String* possui um método, *substring*, com o seguinte cabeçalho:

```
/ **
* Retorne uma nova string contendo os caracteres de
* beginIndex para (endIndex-1) dessa string.
* /
public String substring (int beginIndex, int endIndex)
```

Um valor de índice igual a zero representa o primeiro caractere de uma *string*, portanto, *getLoginName* pega os quatro primeiros caracteres da *string* de nome e os três primeiros caracteres da *string* de identificação e os concatena para formar uma nova *string*. Essa nova sequência é retornada como resultado do método. Por exemplo, se *name* for a sequência "Leonardo da Vinci" e *id* for a sequência "468366", a sequência "Leon468" será retornada por esse método. Aprenderemos mais sobre a chamada de método entre objetos na próxima aula.

2.22 Experimentando expressões: o *Code Pad*

Nas seções anteriores, vimos várias expressões para obter vários cálculos, como o cálculo do preço total + na máquina de *tickets* e o *name.substring*(0,4) expressão da classe *Student.*

No restante deste material, encontraremos muitas outras operações, algumas vezes escritas com símbolos de operador (como “+”) e outras escritas como chamadas de método (como *substring*). Quando encontramos novos operadores e métodos, geralmente, eles nos ajudam a experimentar com diferentes exemplos o que eles fazem.

O *Code Pad*, que usamos brevemente na aula 01, pode nos ajudar a experimentar expressões Java (Figura 2.5). Aqui, podemos digitar expressões, que serão avaliadas imediatamente e os resultados exibidos. Isso é muito útil para experimentar novos operadores e métodos.

Exercício: considere as seguintes expressões. Tente prever seus resultados e digite-os no *Code Pad* para verificar suas respostas:

99 + 3

"gato" + "peixe"

"gato" + 9

9 + 3 + "gato"

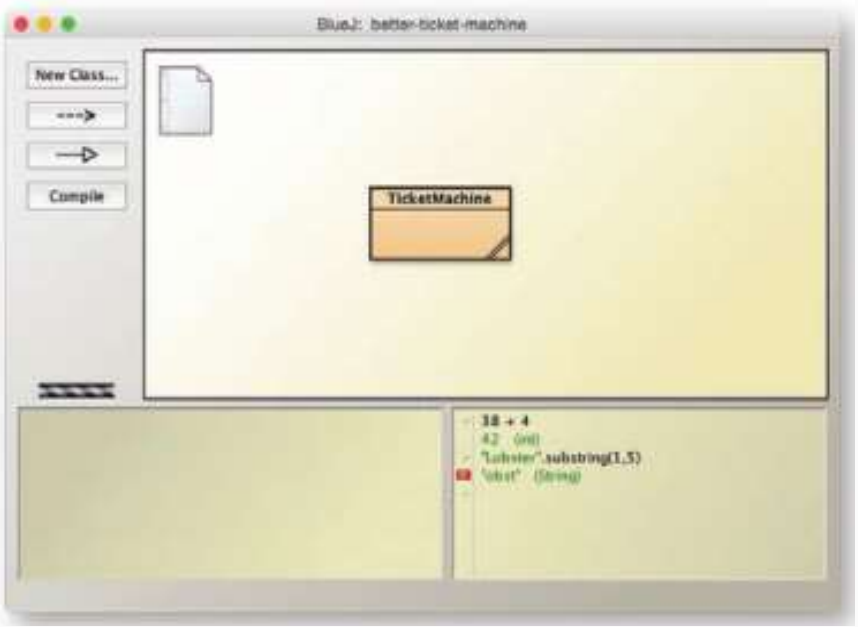
"gato" + 3 + 9

"bagre".substring (3,4)

"bagre".substring (3,8)

Você aprendeu alguma coisa que não esperava do exercício? Se sim, o que foi?

Figura 2.5 – Exemplo no BlueJ



Fonte: BlueJ.

Quando o resultado de uma expressão no *Code Pad* for um objeto (como uma *String*), ele será marcado com um pequeno símbolo de objeto vermelho ao lado da linha que mostra o resultado. Você pode clicar duas vezes nesse símbolo para inspecioná-lo ou arrastá-lo para o banco de objetos para uso posterior. Você também pode declarar variáveis e escrever instruções completas no *Code Pad*. Sempre que você encontrar novos operadores e chamadas de método, é uma boa ideia experimentá-los aqui para ter uma ideia do comportamento deles. Você também pode explorar o uso de variáveis no *Code Pad*. Tente o seguinte:

soma = 99 + 3;

Você verá a seguinte mensagem de erro:

Error: cannot find symbol - variable soma

Isso ocorre porque o Java exige que cada variável (soma, neste caso) receba um tipo antes de poder ser usada. Lembre-se de que toda vez que um campo, parâmetro ou variável local foi introduzido pela primeira vez na fonte, ele tinha um nome de tipo à sua frente, como *int* ou *String*. À luz disso, tente agora o seguinte no *Code Pad*:

int soma = 0;

soma = 99 + 3;

Desta vez, não há problema, porque a soma foi introduzida com um tipo e pode ser usada sem repetir o tipo posteriormente. Se você digitar soma em uma linha por si só (sem ponto e vírgula), você verá o valor que ela armazena atualmente.

Agora tente isso no *Code Pad*:

String nadador = "gato" + "peixe";

nadador

Novamente, demos um tipo apropriado ao nadador variável, permitindo-nos fazer uma atribuição a ele e descobrir o que ele armazena. Desta vez, escolhemos defini-lo com o valor que desejávamos ao mesmo tempo que o declarava. O que você esperaria ver após o seguinte?

String peixe = nadador;

peixe

Experimente. O que você acha que aconteceu na tarefa?

2.23 Resumo

Neste capítulo, abordamos o básico de como criar uma definição de classe. As classes contêm campos, construtores e métodos que definem o estado e o comportamento dos objetos. Dentro do corpo de um construtor ou método, uma sequência de instruções implementa essa parte do seu comportamento. Variáveis locais podem ser usadas como armazenamento temporário de dados para ajudar nisso. Cobrimos declarações de atribuição e declarações condicionais e iremos adicionar outros tipos de declarações nas próximas aulas.

Termos introduzidos nesta aula:

Campo, variável de instância, construtor, método, cabeçalho do método, corpo do método, parâmetro real, parâmetro formal, acessador, mutador, declaração, inicialização, bloco, declaração, declaração de atribuição, declaração condicional, declaração de retorno, tipo de retorno, comentário, expressão, operador, variável, variável local, escopo e tempo de vida.

Videoaula 1

Utilize o QR Code para assistir!

Videoaula 2

Utilize o QR Code para assistir!

Videoaula 3

Utilize o QR Code para assistir!

Encerramento

Chegamos ao final de nossa unidade, em nossa próxima aula veremos mais termos e formas de utilização da linguagem Java. Nos vemos na próxima.

Esperamos que este guia o tenha ajudado compreender a organização e o funcionamento de seu curso. Outras questões importantes relacionadas ao curso serão disponibilizadas pela coordenação.

Grande abraço e sucesso!

UniFil