Linguagem de Programação Orientada a Objetos

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

UniFil

Unidade 2

Introdução da Unidade

Nesta unidade continuamos a desvendar o mundo da orientação a objeto, começando com uma discussão sobre abstração e modularização. Introduzindo os operadores lógicos e matemáticos em Java. Também trabalharemos com dois projetos, “Relógio” e “Cliente de mensagens”.

Objetivos

- Saber como abstrair o conhecimento que desejamos representar (abstração);
- Conhecer sobre modularização, sobre como organizar nossos objetos em partes;
- Abordar sobre os operadores matemáticos, lógicos e como fazer uso deles.

Conteúdo programático

Aula 01 – Abstração e modularização

Aula 02 – Classe *ClockDisplay*

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.

Aula 01 - Abstração e modularização

Nas aulas anteriores, examinamos o que são objetos e como eles são implementados. Em particular, discutimos campos, construtores e métodos quando analisamos as definições de classe.

Vamos agora dar um passo adiante. Para construir aplicativos interessantes, não basta criar objetos de trabalho individuais. Em vez disso, os objetos devem ser combinados para que eles cooperem para executar uma tarefa comum. Neste capítulo, criaremos um pequeno aplicativo a partir de três objetos e seus métodos, invocando outros métodos para atingir seu objetivo.

3.1 O exemplo do relógio

O projeto que usaremos para discutir a interação de objetos é uma exibição para um relógio digital. O visor mostra horas e minutos, separados por dois pontos (Figura 1). Para este exercício, primeiro construiremos um relógio com um mostrador de 24 horas em estilo europeu. Assim, o visor mostra o horário das “00:00” (meia-noite) às “23:59” (um minuto antes da meia-noite). Acontece, em uma inspeção mais detalhada, que construir um relógio de 12 horas é um pouco mais difícil do que um relógio de 24 horas; deixaremos isso até o final da aula.

Figura 1 - Exibição de um relógio digital

Fonte: BlueJ.

3.2 Abstração e modularização

Uma primeira ideia pode ser implementar a exibição do relógio inteiro em uma única classe. Afinal, é o que vimos até agora: como criar classes para fazer um trabalho.

No entanto, aqui abordaremos esse problema de maneira um pouco diferente. Veremos se podemos identificar subcomponentes no problema que podemos transformar em classes separadas. O motivo é a complexidade. À medida que avançamos neste material, os exemplos que usamos e os programas que criamos se tornam cada vez mais complexos. Tarefas triviais, como a máquina de tickets, podem ser resolvidas como um único problema. Você pode analisar a tarefa completa e criar uma solução usando uma única classe. Para problemas mais complexos, isso é simplista demais. À medida que o problema aumenta, fica cada vez mais difícil acompanhar todos os detalhes ao mesmo tempo.

A solução que usamos para lidar com o problema da complexidade é a abstração. Dividimos o problema em subproblemas, novamente em subproblemas e assim por diante, até que os problemas individuais sejam pequenos o suficiente para serem fáceis de lidar. Depois que resolvermos um dos subproblemas, não pensamos mais nos detalhes dessa parte, mas tratamos a solução como um único componente para o próximo problema. Essa técnica às vezes é chamada de dividir e conquistar.

Vamos discutir isso com um exemplo. Imagine engenheiros de uma empresa de automóveis projetando um carro novo. Eles podem pensar nas partes do carro, como a forma da carroçaria, o tamanho, localização do motor, o número e o tamanho dos assentos na área de passageiros, o espaçamento exato das rodas e assim por diante. Outro engenheiro, por outro lado, cujo trabalho é projetar o motor (equipe de), pensa nas várias partes de um motor: os cilindros, o mecanismo de injeção, o carburador, a eletrônica, etc. Eles pensam no motor não como uma entidade única, mas como um trabalho complexo de muitas partes. Uma dessas peças pode ser uma vela de ignição.

Depois, há um engenheiro (talvez em uma empresa diferente) que projeta a vela de ignição. Ele pensará na vela de ignição como um artefato complexo de muitas partes. Ele pode ter realizado estudos complexos para determinar exatamente que tipo de metal usar nos contatos ou que tipo de material e processo de produção a ser usado no isolamento.

O mesmo vale para muitas outras partes. Um projetista do mais alto nível considerará uma roda como uma peça única. Outro engenheiro muito mais adiante pode passar os dias pensando na composição química necessária para produzir os materiais certos para fabricar os pneus. Para o engenheiro de pneus, o pneu é uma coisa complexa. A empresa automobilística apenas comprará o pneu da empresa e o verá como uma única entidade. Isso é abstração.

O engenheiro da empresa automobilística abstrai os detalhes da fabricação do pneu para poder se concentrar nos detalhes da construção, por exemplo, da roda. O projetista que desenha a forma da carroceria abstrai dos detalhes técnicos das rodas e do motor para se concentrar no design da carroceria (ele só estará interessado no tamanho do motor e das rodas).

O mesmo vale para todos os outros componentes. Embora alguém possa se preocupar em projetar o espaço interno do passageiro, alguém pode trabalhar no desenvolvimento do tecido que será usado para cobrir os assentos.

A questão é que, se visto com detalhes suficientes, um carro consiste em tantas partes que é praticamente impossível para uma única pessoa conhecer todos os detalhes de cada peça ao mesmo tempo. Se isso fosse necessário, nenhum carro poderia ser construído.

A razão pela qual os carros são construídos com sucesso é que os engenheiros usam modularização e abstração. Eles dividem o carro em módulos independentes (volante, motor, caixa de velocidades, assento, etc.) e fazem com que pessoas separadas trabalhem em módulos de forma independente. Quando um módulo é construído, eles usam abstração. Eles veem esse módulo como um único componente usado para criar componentes mais complexos.

Modularização e abstração se complementam. Modularização é o processo de dividir coisas grandes (problemas) em partes menores, enquanto abstração é o processo de ignorar detalhes para focar na imagem maior.

3.3 Abstração em software

Os mesmos princípios de modularização e abstração discutidos na seção anterior são usados no desenvolvimento de software. Para nos ajudar a manter uma visão geral de programas complexos, tentamos identificar subcomponentes que podemos programar como entidades independentes. Em seguida, tentamos usar esses subcomponentes como se fossem partes simples, sem nos preocuparmos com suas complexidades internas.

Na programação orientada a objetos, esses componentes e subcomponentes são objetos. Se estivéssemos tentando construir um carro em software, usando uma linguagem orientada a objetos, tentaremos fazer o que os engenheiros de automóveis fazem. Em vez de implementar o carro como um único objeto monolítico, primeiro construímos objetos separados para um motor, caixa de velocidades, roda, assento e assim por diante, e depois montamos o objeto do carro a partir desses objetos menores.

Identificar que tipos de objetos (e com essas classes) você deve ter em um sistema de software para um determinado problema nem sempre é fácil, e teremos muito mais a dizer sobre isso mais adiante neste material. Por enquanto, começaremos com um exemplo relativamente simples. Agora, de volta ao nosso relógio digital.

3.4 Modularização no exemplo do relógio

Vamos dar uma olhada no exemplo do relógio. Usando os conceitos de abstração que acabamos de descrever, queremos tentar encontrar a melhor maneira de visualizar este exemplo, para que possamos escrever algumas classes e para implementá-lo. Uma maneira de ver isso é considerá-lo como consistindo em uma única exibição com quatro dígitos (dois dígitos para as horas, dois para os minutos). Se agora nos afastarmos dessa visão de nível muito baixo, podemos ver que há dois visores separados de dois dígitos (um par por horas e um par por minutos). Um par começa em 0, aumenta 1 a cada hora e volta a 0 após atingir seu limite de 23. O outro volta para 0 após atingir seu limite de 59. A similaridade no comportamento desses dois monitores pode nos levar a abstrair ainda mais a visualização da exibição de horas e minutos. Ao invés disso, podemos pensar neles como objetos que podem exibir valores de zero a um determinado limite. O valor pode ser incrementado, mas, se atingir o limite, ele passa para zero. Agora parece que atingimos um nível apropriado de abstração que podemos representar como uma classe: uma classe de exibição de dois dígitos.

Figura 2 - Uma exibição de número de dois dígitos

Fonte: BlueJ.

Para nossa exibição de relógio, primeiro programaremos uma classe para uma exibição numérica de dois dígitos (Figura 2), em seguida, forneceremos um método acessador, a fim de obter seu valor e dois métodos mutadores para definir o valor e incrementá-lo. Depois de definir essa classe, podemos apenas criar dois objetos da classe com limites diferentes para construir a exibição inteira do relógio.

3.5 Implementando a exibição do relógio

Como discutido acima, para construir a exibição do relógio, primeiro criaremos uma exibição numérica de dois dígitos. Essa exibição precisa armazenar dois valores. Um é o limite no qual ele pode contar antes de passar para zero. O outro é o valor atual. Representaremos esses dois como campos inteiros em nossa classe (Código 3.1).

Código 3.1 Classe para uma exibição de número de dois dígitos

```
public class NumberDisplay
{
  private int limit;
  private int value;
  ...
}
```

Veremos os detalhes restantes dessa classe posteriormente. Primeiro, vamos supor que podemos criar a classe *NumberDisplay* e depois pensar um pouco mais na exibição completa do relógio. Construiríamos uma exibição completa do relógio tendo um objeto que tenha, internamente, duas exibições numéricas (uma para as horas e outra para os minutos). Cada uma das exibições numéricas seria um campo na exibição do relógio (Código 3.2). Aqui, usamos um detalhe que não mencionamos antes: classes definem tipos.

Código 3.2 A classe *Clock-Display* contendo dois campos de *Number*

```
public class ClockDisplay
{
private NumberDisplay hours;
private NumberDisplay minutes;
...
}
```

Quando discutimos os campos na aula 02, dissemos que a palavra "privado" na declaração de campo é seguida por um tipo e um nome para o campo. Aqui usamos a classe *NumberDisplay* como o tipo para os campos nomeados horas e minutos. Isso mostra que os nomes das classes podem ser usados como tipos.

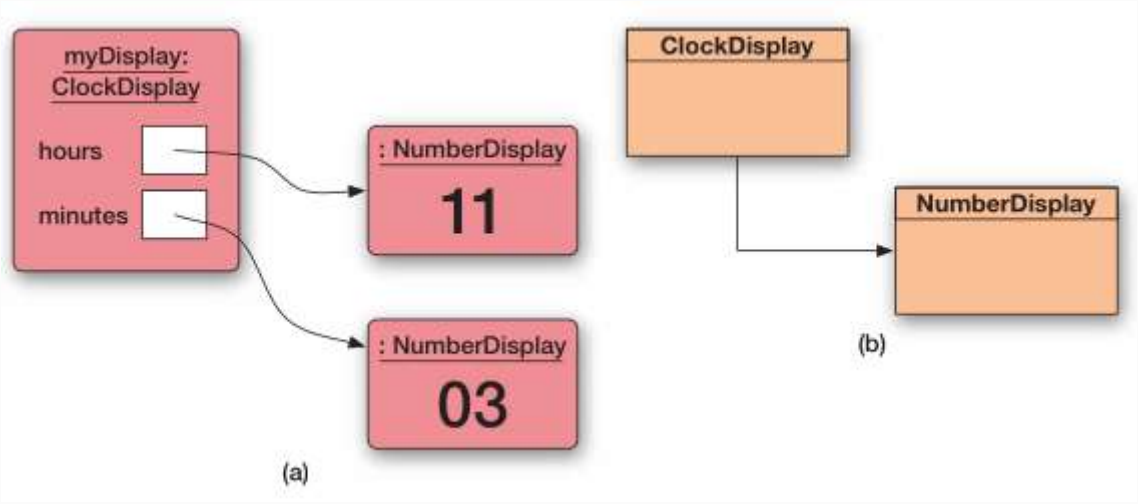
O tipo de um campo especifica que tipo de valores podem ser armazenados no campo. Se o tipo for uma classe, o campo poderá conter objetos dessa classe. Declarar um campo ou outra variável de um tipo de classe não cria automaticamente um objeto desse tipo, pelo contrário, o campo está inicialmente vazio. Ainda não temos um objeto *NumberDisplay*. O objeto associado terá que ser criado explicitamente, e veremos como isso é feito quando olharmos para o construtor da classe *ClockDisplay*.

3.6 Diagramas de classes versus diagramas de objetos

A estrutura descrita na seção anterior (um objeto *ClockDisplay* contendo dois objetos *NumberDisplay*) pode ser visualizada em um diagrama de objetos, como mostra a Figura 3a. Neste diagrama, você vê que estamos lidando com três objetos. A Figura 3b, mostra o diagrama de classes para a mesma situação.

Observe que o diagrama de classes mostra apenas duas classes, enquanto o diagrama de objetos mostra três objetos. Isso tem a ver com o fato de que podemos criar vários objetos da mesma classe. Aqui, criamos dois objetos (*NumberDisplay)* a partir da classe *NumberDisplay*.

Figura 3 - Diagrama de objetos e diagrama de classes para o *ClockDisplay*



Fonte: BlueJ.

Esses dois diagramas oferecem visualizações diferentes do mesmo aplicativo. O diagrama de classes mostra a visualização estática. Ele descreve o que temos no momento em que escrevemos o programa. Temos duas classes, e a seta indica que a classe *ClockDisplay* utiliza a classe *NumberDisplay* (*NumberDisplay* é mencionado no código fonte do *ClockDisplay*). Também dizemos que *ClockDisplay* depende de *NumberDisplay*.

Para iniciar o programa, criaremos um objeto da classe *ClockDisplay*. Vamos programar a exibição do relógio para que ele crie automaticamente dois objetos *NumberDisplay* para si. Portanto, o diagrama de objetos mostra a situação no tempo de execução (quando o aplicativo está sendo executado). Isso também é chamado de exibição dinâmica.

O diagrama do objeto também mostra outro detalhe importante: quando uma variável armazena um objeto, o objeto não é armazenado diretamente na variável, mas uma referência ao objeto é armazenada na variável. No diagrama, a variável é mostrada como uma caixa branca e a referência do objeto é mostrada como uma seta. O objeto referido é armazenado fora do objeto de referência e a referência do objeto vincula os dois.

É muito importante entender esses dois diagramas e visões diferentes. O BlueJ exibe apenas a visualização estática. Você vê o diagrama de classes em sua janela principal. Para planejar e entender os programas Java, você precisa construir diagramas de objetos no papel ou na sua cabeça. Quando pensamos no que nosso programa fará, pensamos nas estruturas de objetos que ele cria e como esses objetos interagem. Ser capaz de visualizar as estruturas de objetos é essencial.

3.7 Tipos primitivos e tipos de objetos

Java conhece dois tipos muito diferentes de tipos: tipos primitivos e tipos de objetos. Todos os tipos primitivos são predefinidos na linguagem Java. Eles incluem *int* e *boolean*. Tipos de objetos são aqueles definidos por classes. Algumas classes são definidas pelo sistema Java padrão (como *String*), outras são as classes que escrevemos.

Os tipos primitivos e os tipos de objeto podem ser usados como tipos, mas há situações em que eles se comportam de maneira diferente. Uma diferença é como os valores são armazenados. Como vimos em nossos diagramas, os valores primitivos são armazenados diretamente em uma variável (escrevemos o valor diretamente na caixa da variável - por exemplo, na aula 02, figura 2.3). Os objetos, por outro lado, não são armazenados diretamente na variável, mas uma referência ao objeto é armazenada (desenhada como uma seta em nossos diagramas, como na Figura 3a).

Veremos outras diferenças entre tipos primitivos e tipos de objetos posteriormente.

3.8 A classe *NumberDisplay*

Antes de começarmos a analisar o código fonte do projeto completo de exibição de relógio, você precisará entender a classe *NumberDisplay* e como seus recursos suportam a criação da classe ClockDisplay. Isso pode ser encontrado no projeto de exibição numérica separado.

O código 3.3 mostra o código fonte completo da classe *NumberDisplay*. No geral, essa classe é bastante direta, embora ilustre alguns novos recursos do Java. Ele possui os dois campos discutidos acima (Seção 3.5), um construtor e quatro métodos (*getValue*, *setValue*, *getDisplayValue* e incremento).

Código 3.3 Implementação da classe *NumberDisplay*

```
/**
* The NumberDisplay class represents a digital number display that can hold
* values from zero to a given limit. The limit can be specified when creating
* the display. The values range from zero (inclusive) to limit-1. If used,
* for example, for the seconds on a digital clock, the limit would be 60,
* resulting in display values from 0 to 59. When incremented, the display
* automatically rolls over to zero when reaching the limit.
*
* @author Michael Kolling and David J. Barnes
* @version 2008.03.30
*/
public class NumberDisplay
{
  private int limit;
  private int value;
  private int startingpoint;
  private int startingpoint;
  private int limitminusone;

  /**
   * Constructor for objects of class NumberDisplay
   * Set the limit at which the display rolls over.
   */
  public NumberDisplay(int rollOverLimit, int startingpoint)
  {
    this.limit = rollOverLimit;
    limitminusone = limit - 1;
    this.startingpoint = startingpoint;
    if( startingpoint == 0){
       value = startingpoint;
    }
    else if ( startingpoint !=0){
        value = rollOverLimit;
    }
  }


  /**
   * Return the current value.
   */
  public int getValue()
  {
     return value;
  }
```

```
/**
 * Return the display value (that is, the current value as a two-digit
 * String. If the value is less than ten, it will be padded with a leading
 * zero).
 */
public String getDisplayValue()
{
    if(value < 10) {
        return "0" + value;
    }
    else {
        return "" + value;
    }
}


/**
 * Set the value of the display to the new specified value. If the new
 * value is less than zero or over the limit, do nothing.
 */
public void setValue(int replacementValue)
{
    if((replacementValue >= 0) && (replacementValue < limit)) {
        value = replacementValue;
    }
}
```

```
/**
 * Increment the display value by one, rolling over to zero if the
 * limit is reached.
 */
public void increment()
{
    if(startingpoint == 0 | value < limitminusone){
        value = (value + 1) % limit;
    }
    else if(startingpoint != 0 && value == limitminusone ) {
        value = limit;
    }
    else if(startingpoint != 0 && value == limit ) {
        value = startingpoint;
    }
}
}
```

O construtor recebe o limite de sobreposição como um parâmetro. Se, por exemplo, 24 for passado como o limite de sobreposição, a exibição passará para 0 neste valor. Assim, o intervalo para o valor exibido seria de 0 a 23. Esse recurso nos permite usar essa classe para exibições de hora e minuto. Para a exibição da hora, criaremos um *NumberDisplay* com limite 24; para a exibição dos minutos, criaremos uma com limite 60. O construtor armazena o limite de sobreposição em um campo e define o valor atual da exibição como 0.

A seguir, segue um método simples de acessador para o valor de exibição atual (*getValue*). Isso permite que outros objetos leiam o valor atual da exibição. As seções seguintes discutem alguns novos recursos que foram usados nos métodos *setValue*, *getDisplayValue* e *increment*.

3.8.1 Os operadores lógicos

O seguinte método mutador, *setValue,* é interessante porque tenta garantir que o valor inicial de um objeto *NumberDisplay* seja sempre válido. Diz:

```
public void setValue (int replacementValue)
{
if ((replacementValue> = 0) && (replacementValue < limit)) {
value = replacementValue;
}
}
```

Aqui, passamos o novo valor para a exibição como um parâmetro para o método. No entanto, antes de atribuirmos o valor, precisamos verificar se o valor é legal. O intervalo legal para o valor, conforme discutido acima, é de zero a um, a menos que o limite. Usamos uma instrução *if* para verificar se o valor é legal antes de atribuí-lo. O símbolo "&&" é um operador lógico "e". Faz com que a condição na instrução *if* seja verdadeira se ambas as condições em ambos os lados do símbolo "&&" forem verdadeiras. Consulte a nota “Operadores lógicos”, a seguir, para obter detalhes.

Nota sobre Operadores lógicos: os operadores lógicos operam com valores booleanos (verdadeiro ou falso) e produzem um novo valor booleano como resultado.

```
&& (e)
|| (ou)
! (não)
```

A expressão:

a && b - é verdadeira se “aeb” são verdadeiros e falsos em todos os outros casos.

A expressão:

a || b - é verdadeira se “a ou b” ou ambos são verdadeiros e falso se ambos são falsos.

A expressão:

!a - é verdadeira se “a” for falso e falso se “a” for verdadeiro.

3.8.2 Concatenação de *strings*

O próximo método, *getDisplayValue*, também retorna o valor da exibição, mas em um formato diferente. O motivo é que queremos exibir o valor como uma sequência de dois dígitos. Ou seja, se o horário atual for “03:05”, queremos que a exibição leia: “03:05”, e não “3: 5”. Para nos permitir fazer isso facilmente, implementamos o método *getDisplayValue*. Este método retorna o valor atual como uma sequência e adiciona um 0 inicial se o valor for menor que 10. Aqui está a seção relevante do código:

```
if(valor < 10) {
return "0" + valor;
}
else {
return "" + valor;
}
```

Observe que o zero ("0") é escrito entre aspas duplas. Assim, escrevemos a *string* “0”, não o número inteiro 0. Então a expressão

"0" + valor

"Adiciona" uma *string* e um número inteiro (porque o tipo de valor é número inteiro). O operador mais (+), portanto, representa a concatenação de cadeias novamente, como visto na Seção 2.9. Antes de continuarmos, agora examinaremos a concatenação de cadeias um pouco mais de perto.

O operador mais (+) tem significados diferentes, dependendo do tipo de seus operandos. Se ambos os operandos são números, isso representa adição, como seria de esperar. Portanto, “42 + 12” adiciona esses dois números e o resultado é 54. No entanto, se os operandos forem cadeias, o significado do sinal de mais é concatenação de cadeias, e o resultado é uma cadeia única que consiste nos dois operandos presos juntos. Por exemplo, o resultado da expressão "Java" + "com BlueJ" é a cadeia única "Java com BlueJ"

Observe que o sistema não adiciona automaticamente um espaço entre as *strings*. Se você deseja um espaço, você deve incluí-lo em uma das *strings*. Se um dos operandos de uma operação positiva for uma sequência e o outro não, o outro operando será automaticamente convertido em uma sequência e, em seguida, uma concatenação de sequência será executada.

Portanto:

"resposta:" + 42

Resulta na *string:*

"resposta: 42"

Isso funciona para todos os tipos. Qualquer que seja o tipo "adicionado" a uma sequência, é automaticamente convertido em uma sequência e concatenado.

Voltar ao nosso código no método *getDisplayValue*. Se o valor contiver 3, por exemplo, a instrução “retornar "0" + valor;” retornará a *string* "03". No caso em que o valor é maior que 9, usamos um pequeno truque:

return "" + valor;

Aqui, concatenamos o valor com uma *string* vazia. O resultado é que o valor será convertido em uma sequência e nenhum outro caractere será prefixado. Estamos usando o operador mais com o único objetivo de forçar uma conversão do valor inteiro para um valor do tipo *String*.

3.8.3 O operador módulo

O último método na classe *NumberDisplay* aumenta o valor de exibição em 1. Ele cuida para que o valor seja redefinido para 0 quando o limite for atingido:

```
public void increment()
{
  valor = (valor + 1) % limit;
}
```

Este método usa o operador módulo (%). O operador módulo calcula o restante de uma divisão inteira. Por exemplo, o resultado da divisão “27/4”, pode ser expresso em números inteiros como resultado = 6, restante = 3.

O operador módulo retorna apenas o restante de uma divisão desse tipo. Assim, o resultado da expressão (27 % 4) é 3.

3.9 A classe *ClockDisplay*

Agora que vimos como podemos construir uma classe que define uma exibição numérica de dois dígitos, veremos mais detalhadamente a classe *ClockDisplay* - a classe que criará duas exibições numéricas para criar uma exibição em tempo integral. O código 3.4 mostra o código fonte completo da classe *ClockDisplay*, que pode ser encontrada no projeto de exibição do relógio.

Código 3.4 Implementação da classe *ClockDisplay*

```
/**
* The ClockDisplay class implements a digital clock display for a
* European-style 24 hour clock. The clock shows hours and minutes.
* The range of the clock is 00:00 (midnight) to 23:59 (one minute
* before midnight).
*
* The clock display receives "ticks" (via the timeTick method) every
* minute and reacts by incrementing the display. This is done in the
* usual clock fashion: the hour increments when the minutes roll
* over to zero.
*
* @author Michael Kölling and David J. Barnes
* @version 2001.05.26
*/
public class ClockDisplay
{
  private NumberDisplay hours;
  private NumberDisplay minutes;
  private String displayString;
  // simulates the actual display
  /**
   * Constructor for ClockDisplay objects. This constructor
   * creates a new clock set at 00:00.
   */
  public ClockDisplay()
  {
    hours = new NumberDisplay(12,1);
    minutes = new NumberDisplay(60,0);
    updateDisplay();
    System.out.println(" ");
    System.out.println(displayString);
  }
```

```
/**
 * Constructor for ClockDisplay objects. This constructor
 * creates a new clock set at the time specified by the
 * parameters.
 */
public ClockDisplay(int hour, int minute)
{
  hours = new NumberDisplay(12,1);
  minutes = new NumberDisplay(60,0);
  setTime(hour, minute);
  System.out.println(" ");
  System.out.println(displayString);
```

```
/**
 * This method should get called once every minute - it makes
 * the clock display go one minute forward.
 */
public void timeTick()
{
  minutes.increment();
  if(minutes.getValue() == 0) {  // it just rolled over!
    hours.increment();
  }
  updateDisplay();
  System.out.println(displayString);
}


public void hourTick()
{
  hours.increment();
  updateDisplay();
  System.out.println(displayString);
}
```

```
/**
 * Set the time of the display to the specified hour and
 * minute.
 */
public void setTime(int hour, int minute)
{
  hours.setValue(hour);
  minutes.setValue(minute);
  updateDisplay();
  System.out.println(displayString);
}


/**
 * Return the current time of this display in the format HH:MM.
 */
public String getTime()
{
  return displayString;
}
```

```
    /**
     * Update the internal string that represents the display.
     */
    private void updateDisplay()
    {
      displayString = hours.getDisplayValue() + ":" +
        minutes.getDisplayValue();
    }
}
```

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 - Classe *ClockDisplay*

Exercício: abra o projeto de exibição do relógio e crie um objeto *ClockDisplay* selecionando o seguinte construtor:

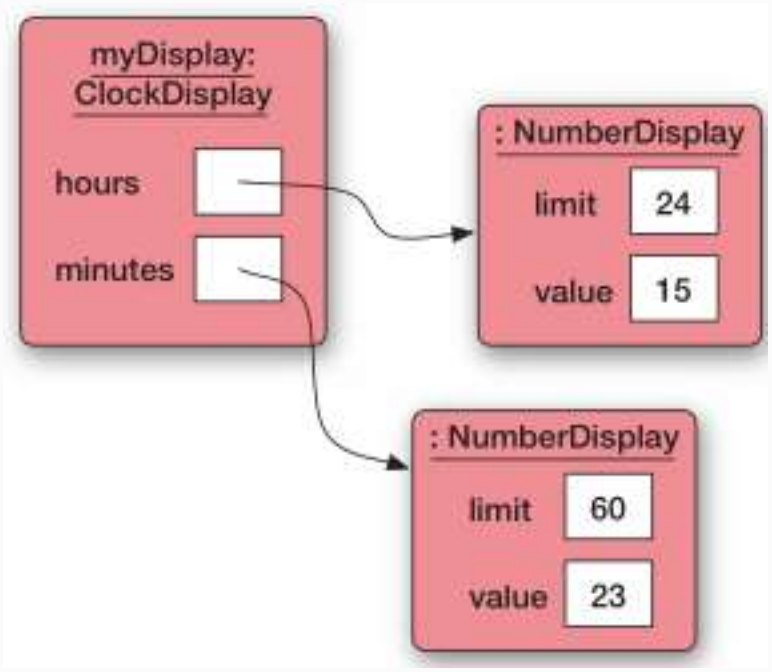
new ClockDisplay ()

Chame seu método *getTime* para descobrir a hora inicial em que o relógio foi definido. Você pode descobrir por que começa naquele momento específico?

Neste projeto, usamos o campo *displayString* para simular o dispositivo de exibição real do relógio. Se este *software* fosse executado em um relógio real, apresentaríamos a saída no visor do relógio real. Portanto, essa *string* serve como nossa simulação de software para o dispositivo de saída do relógio.

Além da sequência de exibição, a classe *ClockDisplay* possui mais dois campos: horas e minutos. Cada um pode conter uma referência a um objeto do tipo *NumberDisplay*. O valor lógico da exibição do relógio (a hora atual) é armazenado nesses objetos *NumberDisplay*. A Figura 4 mostra um diagrama de objeto, desse aplicativo, quando o horário atual é “15:23”.

Figura 4 - Diagrama de objetos da exibição do relógio



Fonte: BlueJ.

3.10 Objetos criando objetos

A primeira pergunta que devemos fazer é: de onde vêm os objetos *NumberDisplay* usados pelo *ClockDisplay*? Como usuário de uma exibição de relógio, quando criamos um objeto *ClockDisplay*, assumimos que nossa exibição de relógio tem horas e minutos. Assim,

simplesmente criando uma exibição de relógio, esperamos que tenhamos criado implicitamente duas exibições de números por horas e minutos.

No entanto, como escritores da classe *ClockDisplay*, temos que fazer isso acontecer. Para fazer isso, simplesmente escrevemos código no construtor do *ClockDisplay* que cria e armazena dois objetos *NumberDisplay*. Como o construtor é executado automaticamente quando um objeto *ClockDisplay* é criado, os objetos *NumberDisplay* serão criados automaticamente ao mesmo tempo. Aqui está o código do construtor *ClockDisplay* que faz este trabalho:

```
public class ClockDisplay
{
  private NumberDisplay hours;
  private NumberDisplay minutes;
  // … Resto dos campos omitidos
    public ClockDisplay()
    {
      hours = new NumberDisplay(24);
      minutes = new NumberDisplay(60);
      updateDisplay();
    }
  // ... Métodos omitidos
}
```

Cada uma das duas primeiras linhas do construtor cria um novo objeto *NumberDisplay* e o atribui a uma variável. A sintaxe de uma operação para criar um novo objeto é o novo *ClassName* (lista de parâmetros). A nova operação faz duas coisas:

1 - Ele cria um novo objeto da classe nomeada (aqui, *NumberDisplay*);

2 - Ele executa o construtor dessa classe.

Se o construtor da classe estiver definido como tendo parâmetros, os parâmetros reais deverão ser fornecidos na nova instrução. Por exemplo, o construtor da classe *NumberDisplay* foi definido para esperar um parâmetro inteiro:

public NumberDisplay (int rollOverLimit)

Portanto, a nova operação para a classe *NumberDisplay*, que chama esse construtor, deve fornecer um parâmetro real do tipo *int* para corresponder ao cabeçalho do construtor definido:

new NumberDisplay (24);

É o mesmo que para os métodos discutidos na Seção 2.5. Com esse construtor, alcançamos o que queríamos: se alguém agora criar um objeto *ClockDisplay*, o construtor *ClockDisplay* executará e criará automaticamente dois objetos *NumberDisplay*. Do ponto de vista de um usuário da classe *ClockDisplay*, a criação dos objetos *NumberDisplay* está implícita. No entanto, do ponto de vista de um escritor, a criação é explícita porque eles precisam escrever o código para que isso aconteça.

A instrução final no construtor é uma chamada para *updateDisplay* que configura o campo *displayString*. Esta chamada é explicada na Seção 3.12.1. Então o relógio está pronto para começar.

3.11 Vários construtores

Você deve ter notado, ao criar um objeto *ClockDisplay*, que o menu *pop-up* oferecia duas maneiras de fazer isso:

new ClockDisplay ()

new ClockDisplay (hora int, minuto int)

Isso ocorre porque a classe *ClockDisplay* contém dois construtores. O que eles fornecem são maneiras alternativas de inicializar um objeto *ClockDisplay*. Se o construtor sem parâmetros for usado, o horário de início exibido no relógio será "00:00". Se, por outro lado, você quiser ter um horário de início diferente, poderá configurá-lo usando o segundo construtor. É comum que as definições de classe contenham versões alternativas de construtores ou métodos que fornecem várias maneiras de realizar uma tarefa específica por meio de seus conjuntos distintos de parâmetros. Isso é conhecido como sobrecarregar um construtor ou método.

3.12 Chamadas de método interno

A última linha do primeiro construtor, *ClockDisplay,* consiste na instrução:

updateDisplay ();

Esta declaração é uma chamada de método. Como vimos acima, a classe *ClockDisplay* possui um método com a seguinte assinatura:

private void updateDisplay ()

A chamada do método acima chama esse método. Como ele está na mesma classe que a chamada do método, também a chamamos de chamada interna do método. As chamadas de método interno têm a sintaxe:

nomeMetodo (lista de parâmetros)

Uma chamada de método interna não possui um nome de variável e um ponto antes do nome do método, o que você observou em todas as chamadas de método registradas na janela Terminal. Uma variável não é necessária, porque com uma chamada de método interna, um objeto chama o método em si. Comparamos as chamadas de métodos internos e externos na próxima seção.

No nosso exemplo, o método não possui parâmetros, portanto a lista de parâmetros está vazia. Isso é representado pelo par de parênteses sem nada entre eles.

Quando uma chamada de método é encontrada, o método correspondente é executado e a execução retorna à chamada de método e continua na próxima instrução após a chamada. Para que uma assinatura de método corresponda à chamada de método, o nome e a lista de parâmetros do método devem corresponder. Aqui, as duas listas de parâmetros estão vazias e, portanto, correspondem. Essa necessidade de corresponder ao nome do método e às listas de

parâmetros é importante, pois pode haver mais de um método com o mesmo nome em uma classe - se esse método estiver sobrecarregado.

Em nosso exemplo, o objetivo desta chamada de método é atualizar a *string* de exibição. Após a criação das duas exibições numéricas, a sequência de exibição é configurada para mostrar a hora indicada pelos objetos de exibição numérica. A implementação do método *updateDisplay* será discutida abaixo.

3.12.1 Chamadas de método externas

Agora vamos examinar o próximo método: *timeTick*. A definição é:

```
public void timeTick()
{
  minutes.increment();
  if(minutes.getValue() == 0) { // acabou de rodar o tempo
    hours.increment();
  }
  updateDisplay();
}
```

Se esse monitor fosse conectado a um relógio real, esse método seria chamado uma vez a cada 60 segundos pelo timer eletrônico do relógio. Por enquanto, chamamos a nós mesmos para testar a tela. Quando o método *timeTick* é chamado, ele primeiro executa a instrução:

minutes.increment ();

Esta declaração chama o método de incremento do objeto de minutos. Assim, quando um dos métodos do objeto *ClockDisplay* é chamado, por sua vez, chama um método de outro objeto para fazer parte da tarefa. Uma chamada de método para um método de outro objeto é chamada de chamada externa. A sintaxe de uma chamada de método externa é:

objeto. nomeMetodo (lista de parâmetros)

Essa sintaxe é conhecida como notação de ponto. Consiste em um nome de objeto, um ponto, o nome do método e os parâmetros para a chamada. É particularmente importante notar que usamos o nome de um objeto aqui e não o nome de uma classe. Usamos o nome minutos em vez de *NumberDisplay*.

A diferença entre chamadas de método internas e externas é clara - a presença de um nome de objeto seguido de um ponto indica que o método que está sendo chamado faz parte de outro objeto. Portanto, no método *timeTick*, o objeto *ClockDisplay* solicita que os objetos *NumberDisplay* realizem parte da tarefa geral. Em outras palavras, a responsabilidade pela tarefa geral de manutenção de tempo é dividida entre a classe *ClockDisplay* e a classe *NumberDisplay*. Esta é uma ilustração prática do princípio de dividir e conquistar a que nos referimos anteriormente em nossa discussão sobre abstração.

O método *timeTick* possui uma instrução *if* para verificar se as horas também devem ser incrementadas. Como parte da condição na instrução *if*, ele chama outro método do objeto minute: *getValue*. Este método retorna o valor atual dos minutos. Se esse valor for zero, sabemos que a exibição acabou de rolar e devemos aumentar as horas. É exatamente isso que o código faz.

Se o valor dos minutos não for zero, terminamos. Não precisamos alterar as horas nesse caso. Portanto, a instrução *if* não precisa de outra parte. Agora também devemos entender os três métodos restantes da classe *ClockDisplay* (consulte o Código 3.4). O método *setTime* usa dois parâmetros - a hora e o minuto - e ajusta o relógio para o horário especificado. Observando o corpo do método, podemos ver que ele faz isso chamando os métodos *setValue* de ambas as exibições de números (aquele para as horas e o para os minutos). Em seguida, chama *updateDisplay* para atualizar a sequência de exibição de acordo, da mesma forma que o construtor.

O método *getTime* é trivial - apenas retorna a *string* de exibição atual. Como sempre mantemos a *string* de exibição atualizada, é tudo o que há para fazer.

Por fim, o método *updateDisplay* é responsável por atualizar a sequência de exibição, para que ela reflita corretamente o tempo, conforme representado pelos dois objetos de exibição numéricos. É chamado sempre que a hora do relógio muda. Mais uma vez, ilustra chamadas de método externas. Ele funciona chamando os métodos *getDisplayValue* de cada um dos objetos *NumberDisplay*. Esses métodos retornam os valores de cada exibição numérica separada. O método Display de atualização usa a concatenação de *string* para unir esses dois valores, separados por dois pontos, em uma única *string*.

3.12.2 Resumo do mostrador do relógio

Vale a pena revisar e ver como este exemplo usa a abstração para dividir o problema em partes menores. Observando o código fonte da classe *ClockDisplay*, você notará que apenas criamos um objeto *NumberDisplay,* sem estar particularmente interessado na aparência interna desse objeto. Podemos então chamar métodos (incremento, *getValue*) desse objeto para fazê-lo funcionar para nós. Nesse nível, simplesmente assumimos que o incremento aumentará corretamente o valor da exibição, sem nos preocuparmos com a forma como isso acontece.

Em projetos do mundo real, essas classes diferentes são frequentemente escritas por pessoas diferentes. Você já deve ter notado que todas essas duas pessoas precisam concordar sobre quais assinaturas de método a classe deve ter e o que deve fazer. Então uma pessoa pode se concentrar na implementação dos métodos, enquanto a outra pessoa pode apenas usá-los.

O conjunto de métodos que um objeto disponibiliza para outros objetos é chamado de interface. Discutiremos as interfaces com muito mais detalhes posteriormente no decorrer das aulas.

3.13 Outro exemplo de interação de objeto

Vamos agora examinar os mesmos conceitos com um exemplo diferente, usando ferramentas diferentes. Ainda estamos preocupados em entender como os objetos criam outros objetos e como os objetos chamam os métodos uns dos outros. Na primeira metade desta aula, usamos a técnica mais fundamental para analisar um determinado programa: leitura de código. A

capacidade de ler e entender o código fonte é uma das habilidades mais essenciais para um desenvolvedor de *software*, e precisamos aplicá-lo em todos os projetos em que trabalhamos. No entanto, às vezes é benéfico usar ferramentas adicionais para nos ajudar a entender melhor como o programa é executado. Uma ferramenta que veremos agora é um *debugger*.

Um *debugger* ou depurador é um programa que permite que os programadores executem um aplicativo em uma etapa de cada vez. Geralmente, fornece funções para parar e iniciar um programa em pontos selecionados no código fonte e para examinar os valores das variáveis.

Nota sobre o "*debugger*" em português depurador. Os erros nos programas de computador são comumente conhecidos como "*bugs*". Assim, os programas que ajudam na remoção de erros são conhecidos como "depuradores". Não está totalmente claro de onde vem o termo "*bug*". Há um caso famoso do que é conhecido como "o primeiro *bug* do computador" - um *bug* real (de fato, uma mariposa) - que foi encontrado dentro do computador Mark II por Grace Murray Hopper, pioneira em computação em 1945. Ainda existe um livro de registro no Museu Nacional de História Americana do Instituto Smithsonian que mostra uma entrada com essa mariposa colada no livro e a observação "primeiro caso real de *bug* encontrado". A redação, no entanto, sugere que o termo "*bug*" já estava em uso antes que este caso real causasse problemas no Mark II.

Para saber mais, faça uma pesquisa na Web por "primeiro *bug* do computador" - você encontrará até fotos da mariposa!

Depuradores variam amplamente em complexidade. Aqueles para desenvolvedores profissionais têm um grande número de funções para um exame sofisticado de muitas facetas de um aplicativo. O BlueJ possui um depurador embutido que é muito mais simples. Podemos usá-lo para interromper nosso programa, percorrer uma linha de código por vez e examinar os valores de nossas variáveis. Apesar da aparente falta de sofisticação do depurador, isso é suficiente para fornecer-nos uma grande quantidade de informações. Antes de começarmos a experimentar o depurador, veremos o exemplo que usaremos para depuração: uma simulação de um sistema de e-mail.

3.13.1 O exemplo do sistema de correio

Começamos investigando a funcionalidade do projeto do sistema de correio. Nesse estágio, não é importante ler a fonte, mas principalmente executar o projeto existente, a fim de entender o que ele faz.

Abra o projeto do sistema de correio, que pode ser encontrado na referência do material do curso. A ideia deste projeto é simular o ato de usuários enviando itens de correio um para o outro. Um usuário usa um cliente de e-mail para enviar itens de e-mail para um servidor, para entregar ao cliente de e-mail de outro usuário. Primeiro, crie um objeto *MailServer*. Agora crie um objeto *MailClient* para um dos usuários. Ao criar o cliente, você precisará fornecer uma instância do *MailServer* como uma área semelhante à que você acabou de criar. Você também precisa especificar um nome de usuário para o cliente de e-mail. Agora crie um segundo *MailClient* de maneira semelhante, com um nome de usuário diferente.

Experimente com os objetos *MailClient*. Eles podem ser usados para enviar itens de e-mail de um cliente de e-mail para outro (usando o método *sendMailItem*) e receber mensagens (usando os métodos *getNextMailItem* ou *printNextMailItem*).

Examinando o projeto do sistema de correio, vemos que:

- Um objeto de servidor de e-mail deve ser criado e usado por todos os clientes de e-mail. Ele lida com a troca de mensagens;
- Vários objetos de cliente de e-mail podem ser criados. Todo cliente de e-mail tem um nome de usuário associado;
- Os itens de e-mail podem ser enviados de um cliente de e-mail para outro por meio de um método na classe de cliente de e-mail;
- Os itens de correio podem ser recebidos por um cliente de e-mail do servidor, um de cada vez, usando um método no cliente de e-mail;
- A classe *MailItem* nunca é explicitamente instanciada pelo usuário. É usada internamente no cliente e no servidor de e-mail para criar, armazenar e trocar mensagens.

As três classes têm diferentes graus de complexidade. *MailItem* é bastante trivial. Discutiremos apenas um pequeno detalhe e deixaremos o resto para o leitor investigar. *MailServer* é bastante complexo nesta fase; faz uso dos conceitos discutidos posteriormente neste material. Não investigaremos essa classe em detalhes aqui. Ao invés disso, apenas confiamos que ele faz seu trabalho - outro exemplo da maneira como a abstração é usada para ocultar detalhes dos quais não precisamos estar cientes. A classe *MailClient* é a mais interessante e a examinaremos com mais detalhes.

Código 3.5 Campos e construtor da classe MailItem

```
/**
 * A class to model a simple mail item. The item has sender and recipiente
 * addresses and a message string.

 * @author David J. Barnes and Michael Kolling
 * @version 2008.03.30
 */
public class MailItem
{
   // The sender of the item.
   private String from;
   // The intended recipient.

   private String to;
   // The text of the message.
   private String message;

   /**
    * Create a mail item from sender to the given recipient,
    * containing the given message.
    * @param from The sender of this item.
```

```
     * @param to The intended recipient of this item.
     * @param message The text of the message to be sent.
     */
    public MailItem(String from, String to, String message)
    {
        this.from = from;
        this.to = to;
        this.message = message;
    }

    /**
     * @return The sender of this message.
     */
    public String getFrom()
    {
        return from;
    }

    /**
     * @return The intended recipient of this message.
     */
    public String getTo()
    {
        return to;
    }

    /**
     * @return The text of the message.
     */
    public String getMessage()

    {
        return message;
    }

    /**
     * Print this mail message to the text terminal.
     */
    public void print()
    {
        System.out.println("From: " + from);
        System.out.println("To: " + to);
        System.out.println("Message: " + message);
    }
}
```

3.13.2 A palavra-chave "*this*"

A única seção que discutiremos da classe *MailItem* é o construtor. Ele usa um construtor Java que não encontramos antes. O código fonte é mostrado no código 3.5. O novo recurso Java neste fragmento de código é o uso da palavra-chave *this*:

this.from = from;

A linha inteira é uma declaração de atribuição. Ele atribui o valor no lado direito (*from*) à variável à esquerda (*this.from*).

A razão para usar essa construção é que temos uma situação conhecida como sobrecarga de nome - o mesmo nome sendo usado para duas entidades diferentes. A classe contém três campos, nomeados "*from*"/de, "*to*"/para e "*message*"/mensagem. O construtor possui três parâmetros, também nomeados de, para e mensagem!

Então, enquanto executamos o construtor, quantas variáveis existem? A resposta é seis: três campos e três parâmetros. É importante entender que os campos e os parâmetros são variáveis separadas que existem independentemente uma da outra, apesar de compartilharem nomes semelhantes. Um parâmetro e um campo que compartilham um nome não são realmente um problema em Java.

O problema que temos, no entanto, é como fazer referência às seis variáveis para poder distinguir entre os dois conjuntos. Se simplesmente usarmos o nome da variável "*from*" no construtor (por exemplo, em uma instrução System.out.println (from)), qual variável será usada - o parâmetro ou o campo?

A especificação Java responde a esta pergunta. Especifica que a definição originada no bloco envolvente mais próximo será sempre usada. Como o parâmetro *from* é definido no construtor e o campo *from* é definido na classe, o parâmetro será usado. Sua definição é "mais próxima" da afirmação que a utiliza.

Agora tudo o que precisamos é de um mecanismo para acessar um campo quando houver uma variável mais bem definida com o mesmo nome. É para isso que a palavra-chave é usada. A expressão refere-se ao objeto atual. Escrever *this.from* refere-se ao campo *from* no objeto atual. Assim, essa construção nos fornece um meio de se referir ao campo em vez do parâmetro com o mesmo nome. Agora podemos ler a declaração de atribuição novamente:

this.from = from;

Esta declaração, como podemos ver agora, tem o seguinte efeito:

campo nomeado from = parâmetro nomeado from;

Em outras palavras, atribui o valor do parâmetro ao campo com o mesmo nome. Obviamente, é exatamente isso que precisamos fazer para inicializar o objeto corretamente.

Uma última pergunta permanece: por que estamos fazendo isso? Todo o problema poderia ser facilmente evitado apenas atribuindo nomes diferentes aos campos e aos parâmetros. O motivo é a legibilidade do código fonte.

Às vezes, há um nome que descreve perfeitamente o uso de uma variável - ela se encaixa tão bem que não queremos inventar um nome diferente para ela. Queremos usá-lo para o parâmetro, onde ele serve como uma dica para o chamador, indicando o que precisa ser

passado. Também queremos usá-lo para o campo, onde é útil como um lembrete para o implementador da classe, indicando para que o campo é usado. Se um nome descreve perfeitamente o uso, é razoável usá-lo para ambos e passar pelo problema de usar a palavra-chave *this* na atribuição, a fim de resolver o conflito de nomes.

3.14 Usando um depurador

A classe mais interessante no exemplo do sistema de e-mail é o cliente de e-mail. Agora, vamos investigar mais detalhadamente usando um depurador. O cliente de e-mail possui três métodos: *getNextMailItem*, *printNextMailItem* e *sendMailItem*. Primeiro, investigaremos o método *printNextMailItem*.

Antes de começarmos com o depurador, configure um cenário que possamos usar para investigar.

Configure um cenário para investigação: Crie um servidor de correio e, em seguida, crie dois clientes de correio para os usuários "Sophie" e "Juan" (você também deve nomear as instâncias “Sophie” e “juan”, para poder distingui-las melhor no banco de objetos). Em seguida, use o método *sendMailItem* da Sophie para enviar uma mensagem a Juan. Não leia a mensagem ainda.

Agora temos uma situação em que um item de e-mail é armazenado no servidor para Juan, aguardando para ser buscado. Vimos que o método *printNextMailItem* pega esse item de correio e o imprime no terminal. Agora, queremos investigar exatamente como isso funciona.

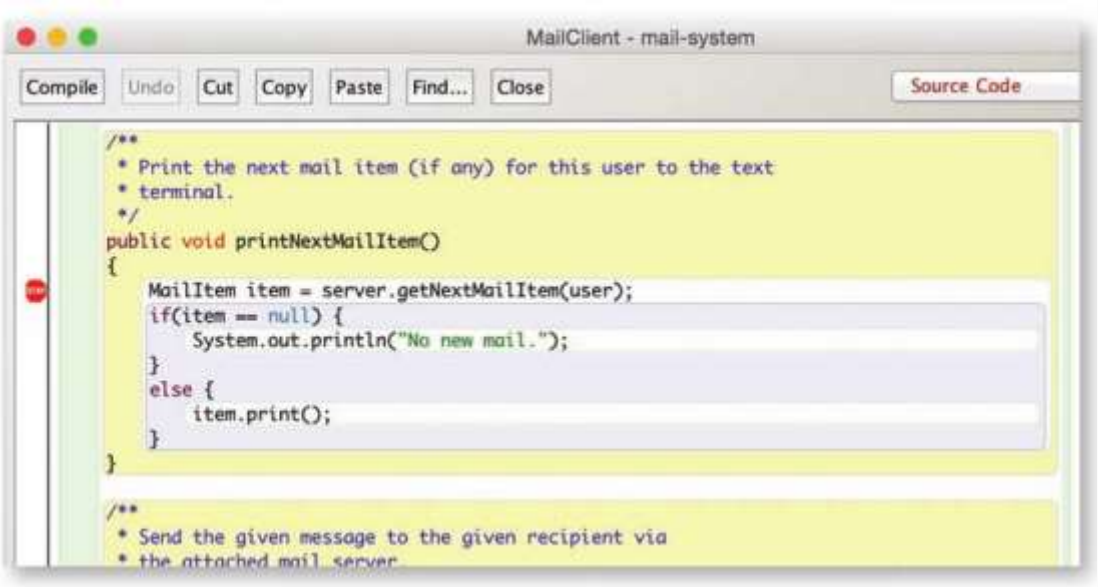
3.14.1 Configurando pontos de interrupção (*breakpoints*)

Para iniciar nossa investigação, estabelecemos um ponto de interrupção no Exercício abaixo. Um ponto de interrupção é um sinalizador anexado a uma linha de código fonte que interrompe a execução de um método nesse ponto quando for atingido. É representado no editor BlueJ como um pequeno sinal de parada (Figura 3.5).

Você pode definir um ponto de interrupção abrindo o editor BlueJ, selecionando a linha apropriada (no nosso caso, a primeira linha do método *printNextMailItem*) e, em seguida, selecionando *Set / Clear Breakpoint* no menu *Tools* do editor. Como alternativa, você também pode simplesmente clicar na área próxima à linha de código onde o símbolo do ponto de interrupção aparece, para definir ou limpar pontos de interrupção. Observe que a classe precisa ser compilada para fazer isso.

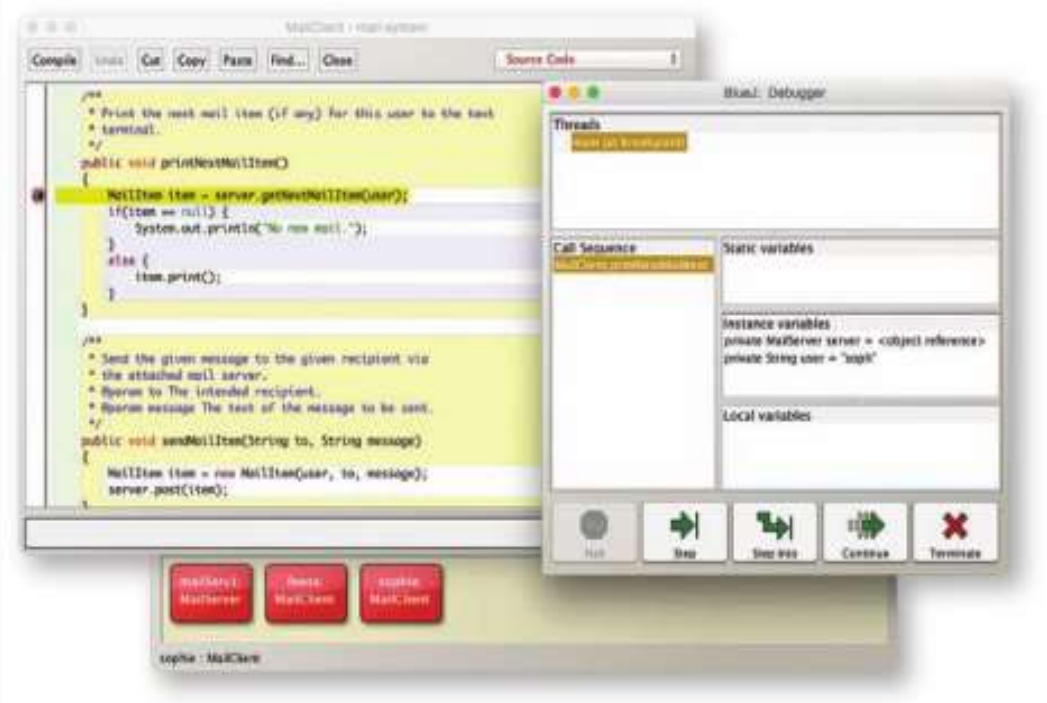
Exercício, abra o editor para a classe *MailClient* e defina um ponto de interrupção na primeira linha do método *printNextMailItem* como mostrado na janela Depois de definir o ponto de interrupção, chame o método *printNextMailItem* no cliente de correio de Juan. A janela do editor para a classe *MailClient* e uma janela do depurador serão exibidas (Figura 6).

Figura 5 - Um ponto de interrupção no editor BlueJ



Fonte: BlueJ.

Figura 6 - A janela do depurador, execução interrompida em um ponto de interrupção



Fonte: BlueJ.

Ao longo da parte inferior da janela do depurador, existem alguns botões de controle. Eles podem ser usados para continuar ou interromper a execução do programa.

No lado direito da janela do depurador, existem três áreas para exibição de variáveis, variáveis estáticas intituladas, variáveis de instância e variáveis locais. Vamos ignorar a área de variável estática por enquanto. Discutiremos variáveis estáticas posteriormente, e essa classe não possui nenhuma.

Vemos que esse objeto tem duas variáveis de instância (ou campos), servidor e usuário, e podemos ver os valores atuais. A variável de usuário armazena a cadeia "juan" e a variável de servidor armazena uma referência a outro objeto. A referência do objeto é o que desenhamos como uma seta nos diagramas de objetos.

Observe que ainda não há variável local. Isso ocorre porque a execução é interrompida antes da execução da linha com o ponto de interrupção. Como a linha com o ponto de interrupção contém a declaração da única variável local e essa linha ainda não foi executada, nenhuma variável local existe no momento.

O depurador não apenas nos permite interromper a execução do programa para que possamos inspecionar as variáveis, mas também nos permite avançar lentamente.

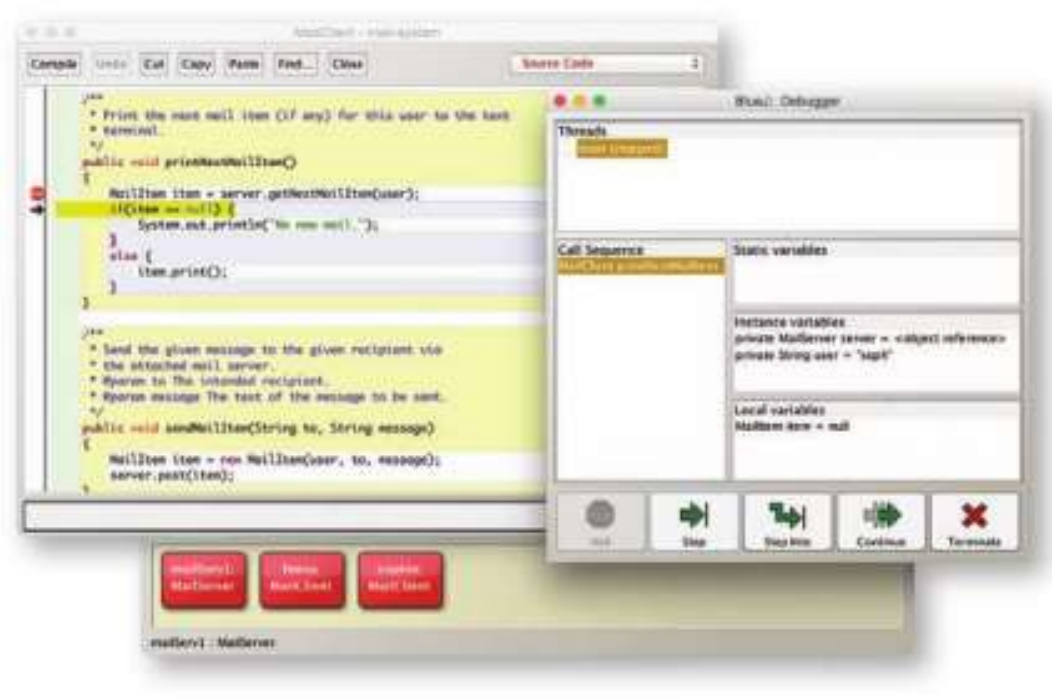
3.14.2 Passo único

Quando parado em um ponto de interrupção, clique no botão *Step* e execute uma única linha de código e depois para novamente.

Exercício: avance uma linha na execução do método *printNext MailItem* clicando no botão Etapa. O resultado da execução da primeira linha do método *printNextMailItem* é mostrado na Figura 7. Podemos ver que a execução avançou em uma linha (uma pequena seta preta ao lado da linha do código fonte indica a posição atual) e a lista de variáveis locais na janela do depurador indica que um item da variável local foi criado e um objeto atribuído a ele.

Exercício, preveja qual linha será marcada como a próxima linha a ser executada após a próxima etapa. Em seguida, execute outra etapa única e verifique sua previsão. Você estava certo ou errado? Explique o que aconteceu e por quê.

Figura 7 - Parado novamente após uma única etapa



Fonte: BlueJ.

Agora podemos usar o botão *Step* repetidamente para ir até o final do método. Isso nos permite ver o caminho que a execução segue. Isso é especialmente interessante em declarações condicionais: podemos ver claramente qual ramificação de uma instrução *if* é executada e usá-la para verificar se ela corresponde às nossas expectativas.

3.14.3 Entrando em métodos

Ao percorrer o método *printNextMailItem*, vimos duas chamadas de método para objetos de nossas próprias classes. A linha: *MailItem* item = server.getNextMailItem (user); inclui uma chamada para o método *getNextMailItem* do objeto do servidor. Verificando as declarações da variável da instância, podemos ver que o objeto do servidor é declarado da classe *MailServer*.

A linha: item.print (); chama o método de impressão do objeto de item. Podemos ver na primeira linha do método *printNextMailItem* que esse item é declarado como da classe *MailItem*.

Utilizando o comando *Step* no depurador, usamos a abstração: vimos o método print da classe do item como uma única instrução e pudemos observar que seu efeito é imprimir os detalhes (remetente, destinatário e mensagem) do item de correio.

Se estivermos interessados em mais detalhes, podemos aprofundar o processo e ver o próprio método de impressão ser executado passo a passo. Isso é feito usando o comando "*Step Into*" no depurador em vez do comando *Step*. "*Step Into*" entrará no método que está sendo chamado e parará na primeira linha dentro desse método.

3.15 Chamada de método (revisitada)

Nos experimentos da Seção 3.14, vemos outro exemplo de interação de objeto semelhante ao que vimos antes: objetos chamando métodos de outros objetos. No método *printNextMailItem*, o objeto *MailClient* fez uma chamada para um objeto *MailServer* para recuperar o próximo item de e-mail. Este método (*getNextMailItem*) retornou um valor - um objeto do tipo *MailItem*. Houve uma chamada para o método de impressão do item de correio. Usando abstração, podemos visualizar o método de impressão como um único comando. Ou, se estivermos interessados em mais detalhes, podemos ir para um nível mais baixo de abstração e examinar o método de impressão.

Em um estilo semelhante, podemos usar o depurador para observar um objeto criando outro. O método *sendMessage* na classe *MailClient* mostra um bom exemplo. Nesse método, um objeto *MailItem* é criado na primeira linha do código:

MailItem item = new MailItem (user, to, message);

A ideia aqui é que o item de e-mail seja usado para encapsular uma mensagem de e-mail. O item de e-mail contém informações sobre o remetente, o destinatário e a própria mensagem. Ao enviar uma mensagem, um cliente de e-mail cria um item de e-mail, com todas essas informações e armazena o item no servidor de e-mail. Desta forma, mais tarde, ele pode ser recolhido pelo cliente de e-mail do destinatário.

Na linha de código acima, vemos a palavra-chave "*new*" sendo usada para criar o novo objeto e os parâmetros sendo passados para o construtor. (Lembre-se: a construção de um objeto faz duas coisas - o objeto está sendo criado e o construtor é executado.) Chamar o construtor funciona de maneira muito semelhante à chamada de métodos. Isso pode ser observado usando o comando *Step Into* na linha em que o objeto está sendo construído.

TERMOS INTRODUZIDOS NESTE CAPÍTULO

Abstração, modularização, dividir e conquistar, diagrama de classes, diagrama de objetos, referência a objetos, sobrecarga, chamada interna de método, chamada externa de método, notação de ponto, depurador e ponto de interrupção.

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, parabéns!

Na segunda aula, discutimos como um problema pode ser dividido em subproblemas. Podemos tentar identificar subcomponentes nos objetos que queremos modelar e podemos implementar subcomponentes como classes independentes. Isso ajuda a reduzir a complexidade da implementação de aplicativos maiores, porque nos permite implementar, testar e manter classes individuais separadamente.

Vimos como essa abordagem resulta em estruturas de objetos trabalhando juntos para resolver uma tarefa comum. Os objetos podem criar outros objetos e podem invocar os métodos um do outro. A compreensão dessas interações com objetos é essencial no planejamento, implementação e depuração de aplicativos.

Podemos usar diagramas de caneta e papel, leitura de código e depuradores para investigar como um aplicativo é executado ou para rastrear *bugs*.

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