



Curso Java Básico

Uma introdução prática usando
BlueJ





Interação entre objetos

Criando objetos cooperadores

Principais conceitos a serem abordados

- Abstração e Modularização
- Classes definem tipos
- Diagrama de classe e de objeto
- Tipos primitivos e tipos objeto
- Criação de objetos
- Sobrecarga de construtor e método
- Interface pública e implementação da classe

Um relógio digital

- Construiremos o mostrador de um relógio digital.
- O mostrador exibe horas e minutos separados por dois pontos.



11:03

Tratando problemas complexos

- À medida que um problema se torna mais complexo, torna-se mais difícil lidar com todos os detalhes ao mesmo tempo.
- A solução para tratar da complexidade é dividir cada problema em subproblemas, até que os problemas individuais se tornem suficientemente pequenos para serem fáceis.
- A técnica dividir para conquistar usa abstração e modularização.

Abstração e modularização

- **Abstração** é a habilidade de ignorar detalhes sobre as partes para concentrar a atenção no nível mais alto de um problema.
- **Modularização** é o processo de dividir um todo em partes bem definidas, que podem ser construídas e examinadas separadamente e que interagem de uma maneira bem definida.

Modularização no exemplo do relógio

11:03

Um mostrador de quatro dígitos ?

Ou dois mostradores de dois dígitos ?

11

03



Modularização no exemplo do relógio



Como funcionaria um mostrador de dois dígitos para as horas ?
E um para os minutos ?
O que há em comum ?



Que informações um mostrador de dois dígitos precisa armazenar ?

Implementação — NumberDisplay

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

    Construtor e
    métodos omitidos.
}
```

Modularização no exemplo do relógio



11:03

Como funcionaria um mostrador completo para o relógio ?



PROJETO
(a solução)

Implementação — ClockDisplay

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

    Construtor e
    métodos omitidos.
}
```

Implementação – ClockDisplay

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

    Construtor e  
métodos omitidos.
}
```



Classes
definem
tipos

Diagrama de objetos

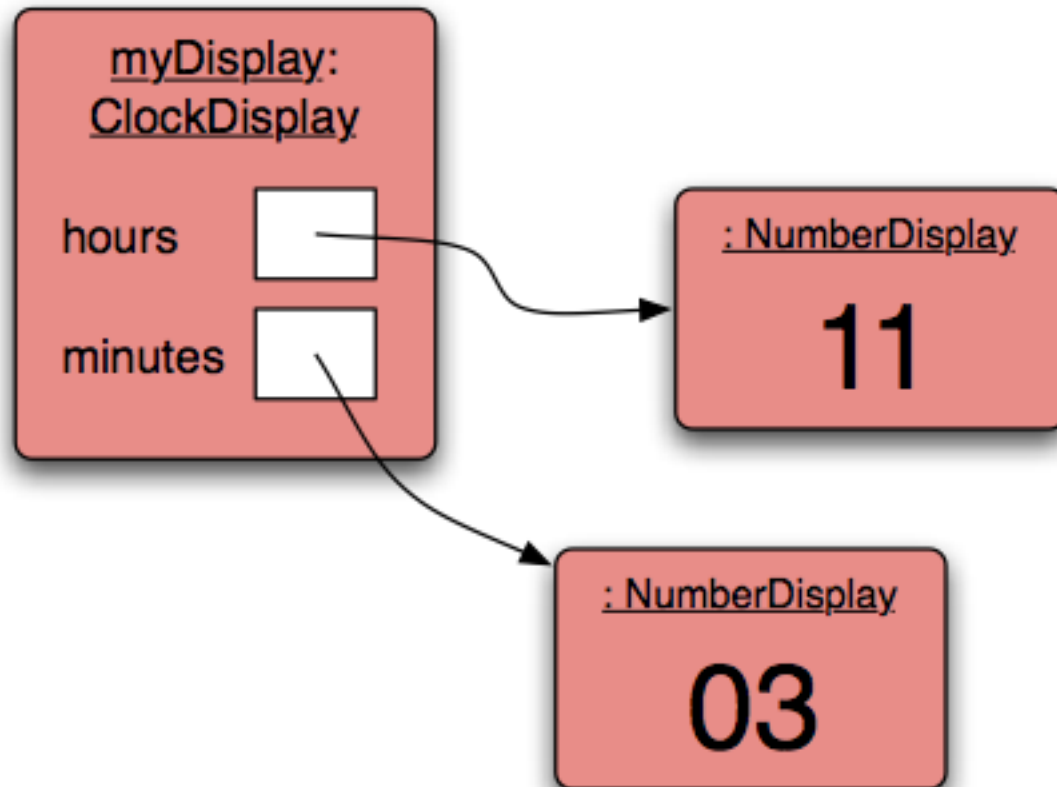
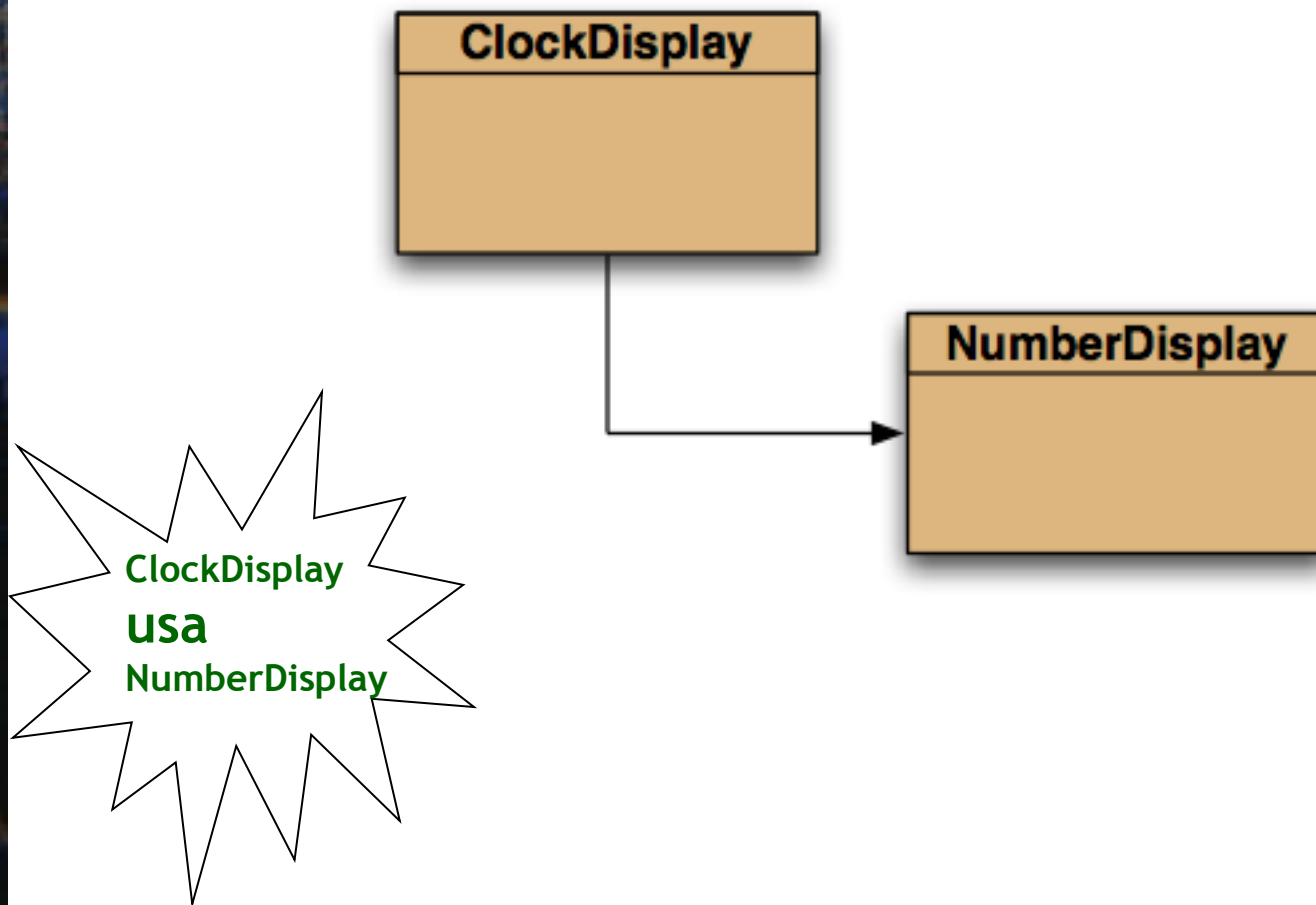


Diagrama de classes



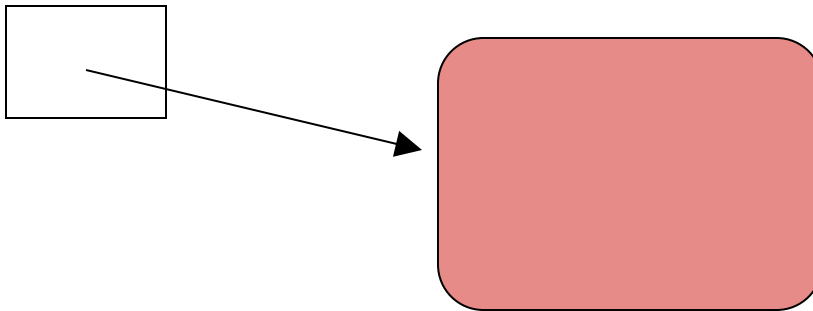
Tipos primitivos *versus* tipos objeto

- Variáveis de tipos primitivos podem armazenar valores daquele tipo.
- Variáveis de tipos objeto podem armazenar referências para objetos daquela classe.
- Tipos objeto também são conhecidos como de tipos de referência.

Tipos primitivos *versus* tipos objeto

`SomeObject obj;`

Tipo objeto



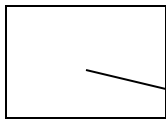
`int i;`

Tipo primitivo

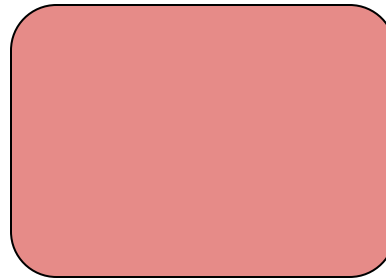
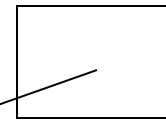


Tipos primitivos *versus* tipos objeto

`ObjectType a;`



`ObjectType b;`



`b = a;`

`int a;`



`int b;`



Exercício

- **Mostrador de relógio digital**
 - Inicie o **BlueJ**, abra o projeto *clock-display* e crie uma instância de *ClockDisplay* usando o construtor sem parâmetros.
 - Use a função *Inspect* para verificar os campos das instâncias de *ClockDisplay* e de *NumberDisplay*.
 - Chame os métodos *timeTick* e *setTime*, observando os campos *displayString* (*ClockDisplay*) e *value* (*NumberDisplay*).

Exercício

- **Mostrador de relógio digital**
 - Edite a classe *ClockDisplay*, comente a chamada ao método *updateDisplay* no construtor sem parâmetros e compile a classe.
 - Crie uma instância de *ClockDisplay* usando o construtor sem parâmetros e inspecione o campo *displayString*. Qual é o valor apresentado ? Porque ?
 - Edite a classe *ClockDisplay* e desfaça o comentário.

Palavra-chave *null*

- A palavra-chave *null* é um valor especial em Java. Significa ‘nenhum objeto’.
- Este é o valor default para uma variável objeto, o qual é assumido se ela não for explicitamente inicializada.
- A qualquer momento, pode-se:
 - atribuir o valor *null* a uma variável objeto
 - testar se é *null* o valor atual de uma variável objeto

Exercício

- **Mostrador de relógio digital**
 - Crie uma instância de *NumberDisplay* informando ao construtor um valor limite.
 - Use a função *Inspect* para verificar os campos da instância de *NumberDisplay*.
 - Chame o método *setValue* e configure valores abaixo e acima do limite. O que acontece em cada caso ?
 - Como você implementaria o método *setValue* ?

Código-fonte: NumberDisplay

```
public NumberDisplay(int rollOverLimit)
{
    limit = rollOverLimit;
    value = 0;
}

public void setValue(int replacementValue)
{
    if ((replacementValue >= 0) ↓
        ↙ Operador relacional
        Operador lógico → && (replacementValue < limit)) {
        value = replacementValue;
    }
}
```

Expressões booleanas (ou condicionais)

- Resultam *true* ou *false* e podem usar:
- Operadores relacionais (operandos aritméticos)
 - `==` (igual a) `!=` (diferente de)
 - `<` (menor que) `<=` (menor ou igual que)
 - `>` (maior que) `>=` (maior ou igual que)
- Operadores lógicos (operandos booleanos)
 - `&` (e) `&&` (e curto-circuito)
 - `|` (ou) `||` (ou curto-circuito)
 - `^` (ou exclusivo)
 - `!` (não)

Exercício

- **Mostrador de relógio digital**
 - Com a mesma instância de *NumberDisplay*, configure um valor próximo ao limite e use o método *increment* para alcançar o valor limite. O que ocorre neste caso ?
 - Como você implementaria o método *increment* ?

Código-fonte: NumberDisplay

```
public NumberDisplay(int rollOverLimit)
{
    limit = rollOverLimit;
    value = 0;
}
```

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

↑
Operador módulo

Exercício

- **Testando expressões: o CodePad**
 - Usando o **CodePad**, teste algumas expressões com o operador módulo (%).

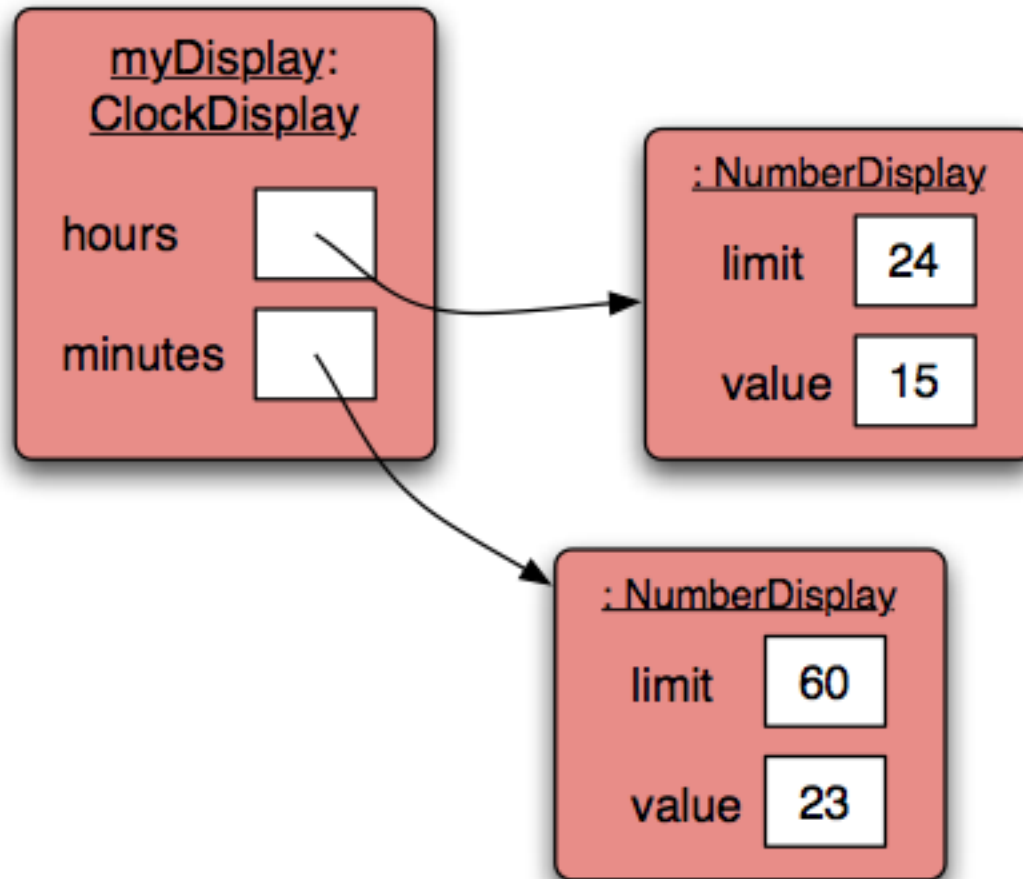
Exercício

- **Mostrador de relógio digital**
 - Com a mesma instância de *NumberDisplay*, configure (*setValue*) um valor de um dígito e verifique a diferença entre os retornos dos métodos *getValue* e *getDisplayValue*.
 - Como você implementaria o método *getDisplayValue* ?

Código-fonte: NumberDisplay

```
public String getDisplayValue()  
{  
    if(value < 10) {  
        return "0" + value;  
    }  
    else {  
        return "" + value;  
    }  
}
```


Objetos criando objetos



Código-fonte: ClockDisplay

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;

    public ClockDisplay()
    {
        Operador new
        hours = new NumberDisplay(24) ;
        minutes = new NumberDisplay(60) ;
        updateDisplay() ;
    }
}
```



Objeto criando objetos

Criação de objetos

- Operador **new**

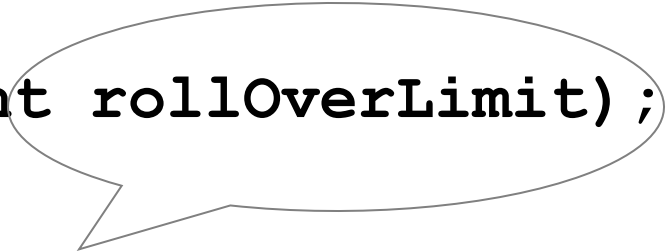
- Instancia um novo objeto da classe nomeada, alocando memória para variáveis de instância
- Inicializa o objeto, executando o construtor da classe nomeada com mesma assinatura
 - se a assinatura contiver parâmetros formais, a criação (operador **new**) deve fornecer os parâmetros reais
- Retorna uma referência para o objeto criado

***new** className (parameter-list)*

Objetos criando objetos

in class NumberDisplay:

```
public NumberDisplay(int rollOverLimit);
```



parâmetro formal

in class ClockDisplay:

```
hours = new NumberDisplay(24);
```



parâmetro real

Código-fonte: ClockDisplay

- A classe ***ClockDisplay*** contém dois construtores, disponibilizando modos alternativos de inicializar uma instância da classe.



```
public ClockDisplay() {...}
```

```
public ClockDisplay(int hour, int minute) {...}
```

Sobrecarga de construtor

- Uma classe pode ter mais de um construtor, desde que tenham assinaturas diferentes. Isto é *sobrecarga de construtor*.
- **Assinatura** é composta de dois elementos: nome e lista dos tipos dos parâmetros:

className (type1, type2, ...)

Exercício

- **Mostrador de relógio digital**
 - Limpe a bancada de objetos e crie duas instâncias de *ClockDisplay*, uma com cada construtor.
 - Edite *ClockDisplay* e identifique as semelhanças e diferenças entre os construtores. Porque não há nenhuma chamada para o método *updateDisplay* no construtor com parâmetros ?

Código-fonte: ClockDisplay

```
public ClockDisplay()  
{  
    hours = new NumberDisplay(24);  
    minutes = new NumberDisplay(60);  
    updateDisplay();  
}  
  
public ClockDisplay(int hour, int minute)  
{  
    hours = new NumberDisplay(24);  
    minutes = new NumberDisplay(60);  
    setTime(hour, minute);  
}
```

Código-fonte: ClockDisplay

```
public void setTime(int hour, int minute)
{
    hours.setValue(hour) ;
    minutes.setValue(minute) ;
    updateDisplay() ;
}

private void updateDisplay()
{
    displayString =
        hours.getDisplayValue() + ":" +
        minutes.getDisplayValue() ;
}
```

Sobrecarga de método

- Uma classe pode ter mais de um método com o mesmo nome, desde que tenham assinaturas diferentes. Isto é *sobrecarga de método*.
- Assinatura é composta de dois elementos: nome e lista dos tipos dos parâmetros:

methodName (type1, type2, ...)

Exercício

- **Mostrador de relógio digital**
 - Edite *ClockDisplay* e crie um método *setValue* sem parâmetros que configure o relógio para o mesmo horário que o construtor sem parâmetros.
 - Compile e teste o método criado.

Exercício

- **Mostrador de relógio digital**
 - A classe *ClockDisplay* contém o método *timeTick*, que seria chamado a cada 60 segundos se a classe fosse usada por um relógio real.
 - A responsabilidade de *timeTick* é incrementar em um minuto o horário do mostrador, delegando tarefas aos objetos que criou através de chamadas a seus métodos.
 - Como você implementaria o método *timeTick* ?

Código-fonte: ClockDisplay

```
public void timeTick()  
{  
    minutes.increment();  
    if(minutes.getValue() == 0) {  
        // acaba de voltar a zero!  
        hours.increment();  
    }  
    updateDisplay();  
}
```

Chamada de método externo

Chamada de método externo

Chamada de método interno

Chamadas de método

- Chamadas de método interno
(definido na mesma classe)

`updateDisplay() ;`

- Chamadas de método externo
(definido em outra classe)

`minutes.increment() ;`

object . methodName (parameter-list)

↑
Operador *ponto*
(*acesso a um membro*)

Resumo do mostrador de relógio digital

- Um objeto de uma classe (`ClockDisplay`) pode criar objetos de outra classe (`NumberDisplay`) e chamar seus métodos (`increment`, `getValue`) sem conhecer como foram implementados.
- Estas duas classes podem até ser escritas por pessoas diferentes, desde que haja concordância sobre a *assinatura* e o *comportamento* de cada método usado em outra classe (*interface pública*).

Interface pública e implementação da classe

- Podemos pensar nos objetos como uma “caixa-preta” com uma *interface pública* e uma *implementação* oculta.
- Objetos definem sua interação com o mundo externo através dos métodos que expõem. Estes métodos formam sua *interface pública* com o mundo exterior.

Interface pública e implementação da classe

- ***Interface pública:***
 - os métodos que podem ser chamados (a rigor, os construtores e os membros públicos; entretanto os métodos expõem o comportamento).
- ***Implementação:***
 - o código e os dados que fazem os métodos funcionarem.

Exercício

- **Mostrador de relógio digital**
 - Caso a classe *ClockDisplay* fosse usada por um relógio real, que métodos seriam usados como ***interface pública*** entre ela e a classe simuladora deste relógio ?
 - Feche o projeto *clock-display* e abra o projeto *clock-display-with-GUI*.
 - Crie uma instância de *Clock* e experimente os botões de sua janela.
 - Edite a classe *Clock* e verifique o método *step*.

Um sistema de correio eletrônico

- Vamos examinar um outro projeto. Simularemos um sistema de correio eletrônico:
 - neste sistema, um usuário utiliza um cliente de correio eletrônico para enviar itens de e-mail a um servidor a fim de que eles possam ser entregues a outro cliente de correio eletrônico.

Exercício

- **Sistema de correio eletrônico**

- Feche o projeto anterior, abra o projeto *mail-system* e crie uma instância de *MailServer*.
- Crie uma instância de *MailClient*, informando a instância de *MailServer* já criada e um nome de usuário. Repita com outro nome de usuário.
- Experimente enviar (*sendMailItem*) e receber (*getNextMailItem* ou *printNextMailItem*) mensagens usando os dois clientes de correio.

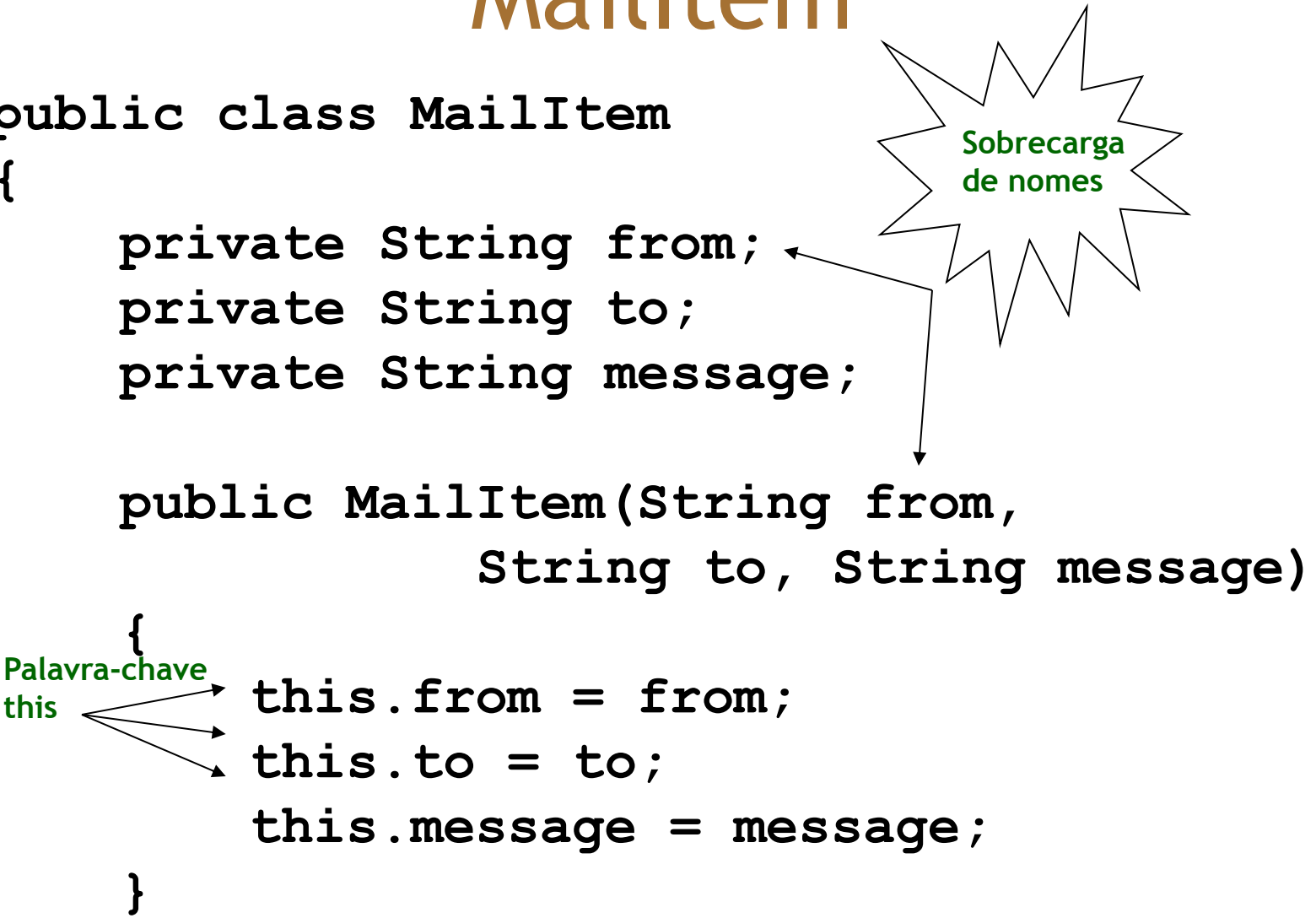
Um sistema de correio eletrônico

- As três classes do projeto têm graus diferentes de complexidade.
- ***MailServer*** usa conceitos que veremos em capítulos a frente. Apenas confiaremos que ela faz o seu trabalho.
- ***MailClient*** e ***MailItem*** serão examinadas, começando com a última, a mais simples.

Código-fonte: MailItem

```
public class MailItem
{
    private String from;
    private String to;
    private String message;

    public MailItem(String from,
                    String to, String message)
    {
        this.from = from;
        this.to = to;
        this.message = message;
    }
}
```



The diagram illustrates the relationship between the constructor and the instance variables. A starburst shape labeled "Sobrecarga de nomes" (Name Overload) points to the parameters `from`, `to`, and `message` in the constructor signature. Another arrow points from the `this` keyword in the constructor body to the same three variables, indicating that `this` refers to the current object and its attributes.

Palavra-chave *this*

- A palavra-chave *this* é uma referência ao objeto corrente.
- Usada (seguida de ponto) para se referir a membros do objeto corrente:

`this.from = from;`

`this.updateDisplay();`

↑
Operador *ponto*

- Usada (seguida de parêntesis) em um construtor para chamar outro da mesma classe:

`this(0, 0);`

Palavra-chave *this*

```
public ClockDisplay()  
{  
    this(0, 0);  
}
```



Invocação
explícita de
construtor

```
public ClockDisplay(int hour, int minute)  
{  
    hours = new NumberDisplay(24);  
    minutes = new NumberDisplay(60);  
    setTime(hour, minute);  
}
```

Duplicação do código

- A duplicação do código:
 - é um indicador de design ruim
 - torna a manutenção mais difícil
 - pode levar à introdução de erros durante a manutenção

Exercício

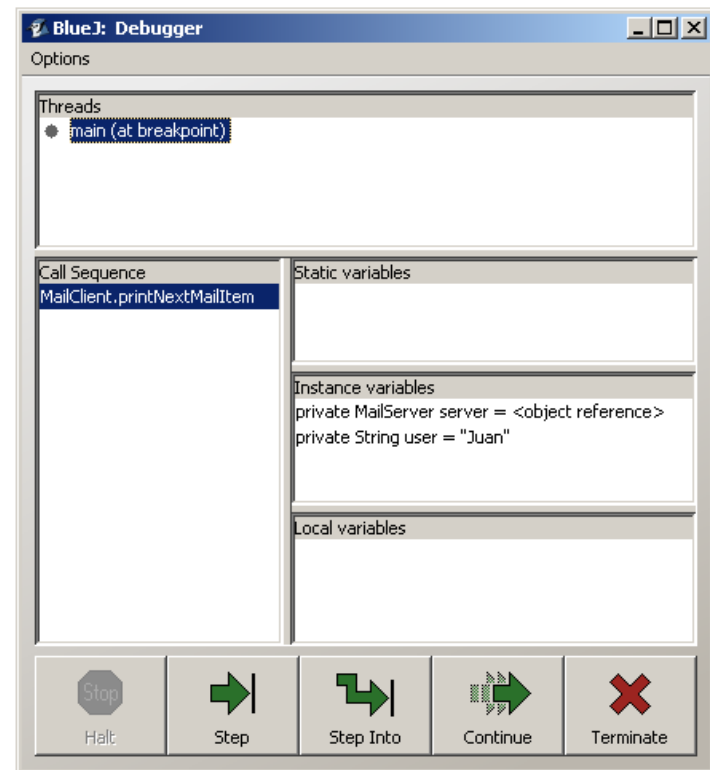
- **Mostrador de relógio digital**
 - Feche o projeto atual, abra o projeto *clock-display* e edite a classe *ClockDisplay* .
 - Elimine duplicação de código nos construtores. Dica: use invocação explícita de construtor.
 - Elimine duplicação de código nos métodos *setValue*. Dica: faça do corpo de um apenas uma chamada ao outro.
 - Compile a classe e teste o construtor e o método alterado.

Depurador

- Investigaremos *MailClient* usando um depurador.
- Um depurador permite interromper a execução de um programa para inspecionar o conteúdo de variáveis.

Depurador

- No BlueJ, usamos a função *Debugger* que pode ser visualizada a partir do menu View.



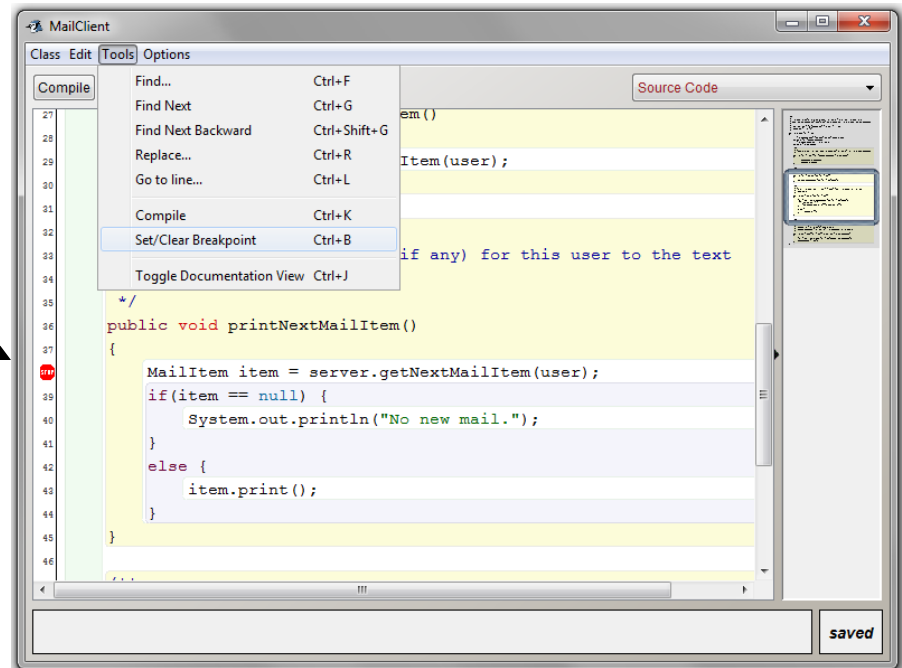
Exercício

- **Sistema de correio eletrônico**
 - Feche o projeto atual e abra o projeto *mail-system*.
 - Crie duas instâncias, uma para a usuária Maria e outra o usuário João.
 - Envie uma mensagem de Maria para João (não leia a mensagem ainda).

Exercício

- **Sistema de correio eletrônico**
 - Edite a classe *MailClient* e configure um ponto de interrupção (*breakpoint*) na primeira linha de *printNextMailItem*.

Selecione a linha e use a opção Set/Clear Breakpoint ou apenas clique na área do símbolo STOP



Exercício

- **Sistema de correio eletrônico**
 - Avance uma linha na execução do método *printNextMailItem* pressionando o botão **Step** do depurador.
 - Qual variável local é apresentada ?
 - Qual linha será executada a seguir ? E qual será a próxima ?
 - Avance até o fim da execução do método *printNextMailItem*. Chame-o novamente no cliente de João e avance até o fim da execução. O que ocorreu ?

Depurador

- Um depurador permite visualizar uma chamada de método de dois modos:
 - uma instrução (no método chamador)
 - várias instruções (no método chamado)
- Caso o depurador visualize diversos métodos chamados, estes aparecem empilhados.

Exercício

- **Sistema de correio eletrônico**
 - Envie outra mensagem de Maria para João e chame *printNextMailItem* no cliente de João.
 - Percorra o código com o botão **Step**. Porém, use o botão **Step Into** ao alcançar a linha: `item.print();`
 - O que você observa ?

Exercício

- **Sistema de correio eletrônico**
 - Edite a classe *MailClient* e configure um ponto de interrupção (*breakpoint*) na primeira linha de *sendMailItem*.
 - Invoque *sendMailItem* e use **Step Into** para visualizar o construtor do item de correio. Observe a inicialização das variáveis de instância a partir dos parâmetros de mesmo nome.

Revisão (1)

- Podemos tratar problemas complexos dividindo-os em subproblemas.
- Variáveis de tipos primitivos armazenam valores daquele tipo.
- Variáveis de tipos objeto armazenam referências a objetos daquele tipo.
- O operador new aloca memória, chama o construtor e retorna uma referência para o objeto criado.

Revisão (2)

- Métodos e construtores podem ser sobrecarregados caso tenham assinaturas diferentes.
- As classes possuem uma interface pública e uma implementação oculta.
- A duplicação de código torna a manutenção mais difícil e sujeita a erros.
- Um depurador pode inspecionar a execução de um programa.



Contatos

Câmara dos Deputados
CENIN - Centro de Informática

Carlos Renato S. Ramos

carlosrenato.ramos@camara.gov.br