

# **BlueJ + Definindo classes + Interação entre objetos**

Instituto Metr pole Digital

Disciplina: IMD0040 - Linguagem de Programação II

Docente: Emerson Alencar



# Revisando Aula 01

- Classe
- Objeto
- Instância
- Atividade BlueJ:
  - Projeto Shapes + Compilar + Criar um objeto circle1
  - Criar mais um circle2 + square1
- Métodos: Comunicação com o objeto invocando seus métodos, exemplo: `makeVisible()`
- Parâmetros: `moveHorizontal(int distance)`
- Assinatura do método
- Tipos de parâmetros: `void changeColor(String newColor)`

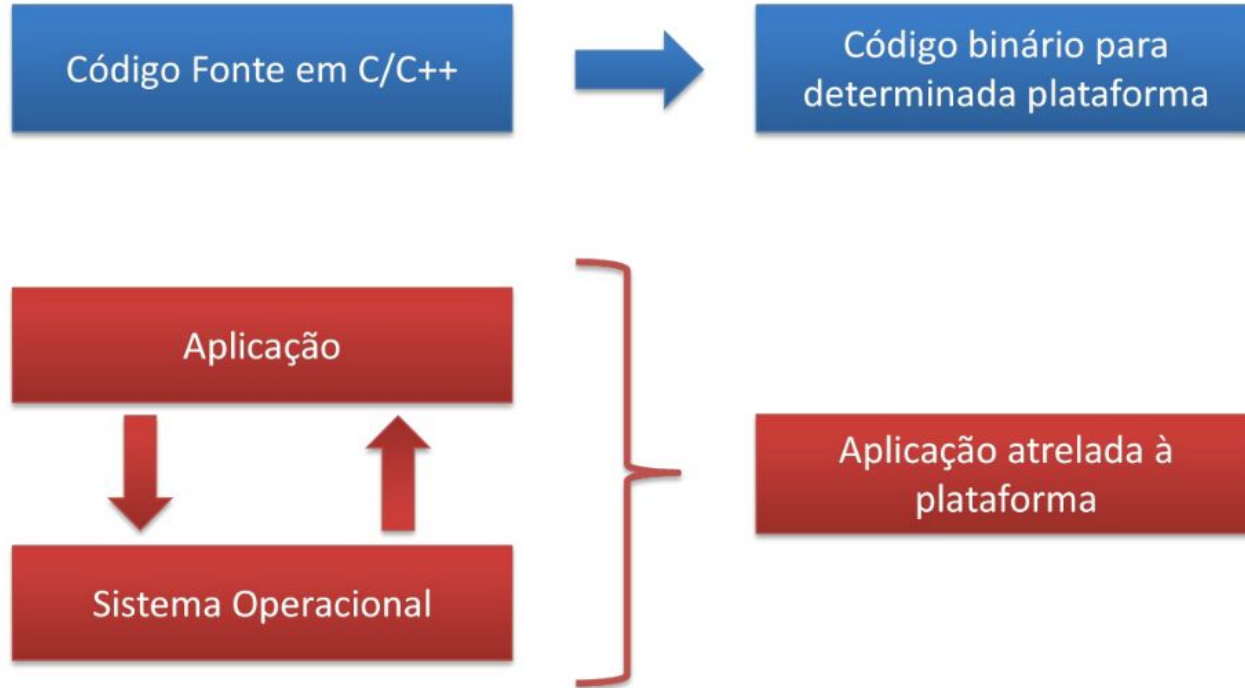
# Revisando Aula 01

- Estado: É representado ao armazenar os valores em campos (*inspect*)
- Atividade: Criar a imagem de uma casa
- Abrir projeto chamado *picture*
- Criar um objeto *picture*. Como a Classe *Picture* desenha a figura?
  - Os objetos podem criar outros objetos
- Código-fonte: texto que define as classes (*Open Editor*)
  - Como Editar no BlueJ
- Abrir o projeto *lab-classes*
  - Criar um objeto da Classe *Student*
- Valores de Retorno
  - Chamar o *getName()*
- Objetos como parâmetros
  - Criar um objetos *LabClass* e inserir o objeto *student1*

# Mais informações: JVM

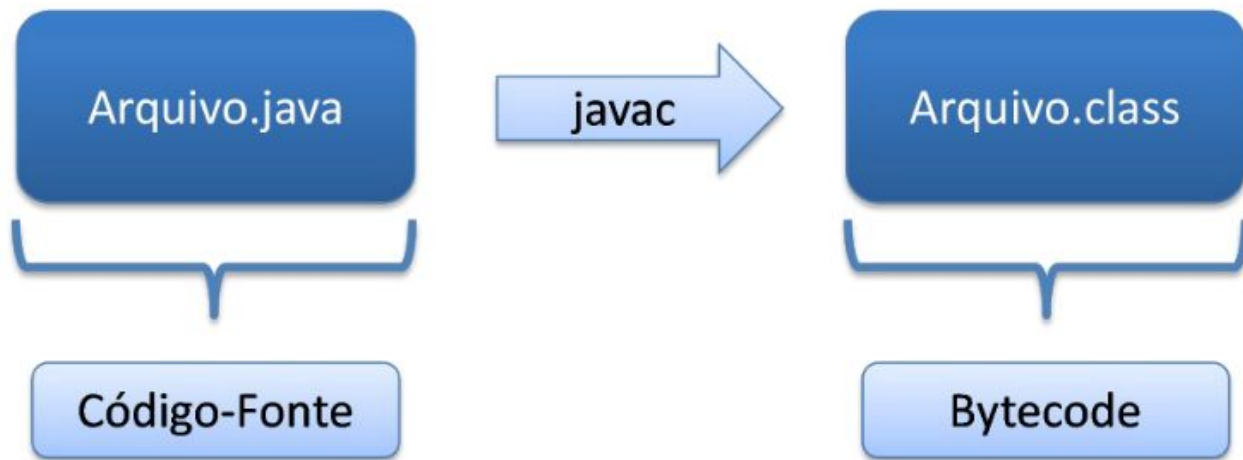
- JVM
  - Java Virtual Machine
- A Máquina Virtual é uma camada intermediária entre o sistema operacional e a aplicação. Quem executa a aplicação é a JVM;
- A aplicação se comunica apenas com a JVM;

# Aplicações Atreladas à Plataforma

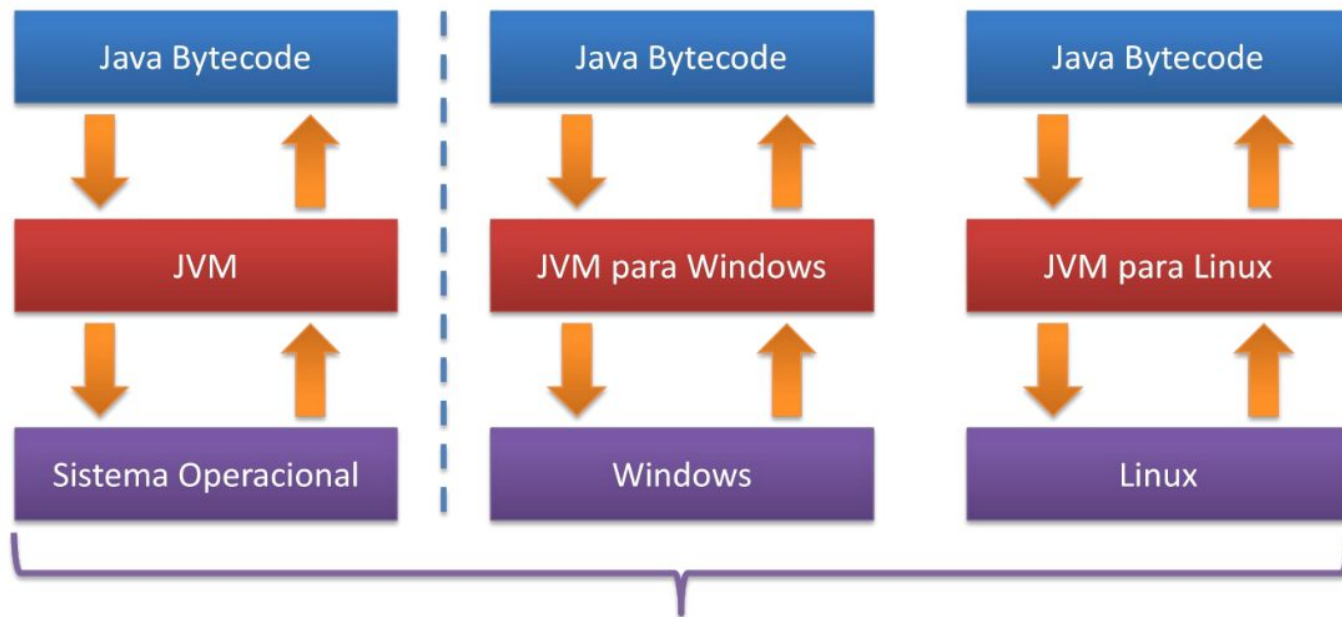


# Bytecode

- O bytecode é uma linguagem entendida pela JVM
- A geração do bytecode é feita através da compilação do código Java



# Mesmo Código entre Plataformas



Ao mudar a plataforma, o bytecode não precisa ser alterado

# Quais as vantagens da Máquina Virtual

- Isolamento total da aplicação;
- Como tudo passa pela JVM, é possível obter métricas e trabalhar com otimização;
- Garbage Collection;
- Princípio WORA
  - “Write Once, Run Anewhere”



# A JVM é uma especificação

- Existe um documento que define como a JVM, como deve se comportar e o que ela deve ter;
- Diversas empresas implementam a JVM
  - Oracle, IBM, etc.
- É possível trocar de JVM sem a necessidade de recompilar os códigos das aplicações

# A performance do Java

- A JVM, durante a execução da aplicação, usa dois elementos para otimizar a performance
  - HotSpot (Identifica código bastante executado)
  - JIT (Just in Time Compiler)
    - Compilar o código identificado pelo HotSpot para instruções nativas da plataforma
- Mito da Performance
  - “Java é uma linguagem com baixa performance”

# Evolução das Versões do Java

- Java 1.0 e 1.1
  - Primeiras Versões
- Java 2 (Java 1.2)
  - Aumento significativo no tamanho da API
- Java 2 (Java 1.3 e 1.4)
  - Melhorias na API
  - Java 1.3.1, 1.4.1, 1.4.2, etc.
- Java 5 (Java 1.5)
  - Diversas mudanças significativas
- Java 6 (Java 1.6)
  - Mais recursos na API
  - Muitas melhorias de performance da JVM

# Evolução das Versões do Java

- Java 7
  - Novas APIS
  - Novos Recursos na linguagem
  - Melhorias internas na JVM
- Java 8
  - Nova API de data e hora
  - Suporte a expressões lambda

# Ramificações do Java

- Java SE (Standard Edition)
  - Base do Java
  - Ambiente de execução e bibliotecas comuns
- Java EE (Enterprise Edition)
  - Aplicações corporativas e internet
- Java ME (Micro Edition)
  - Dispositivos Móveis

# Atividade

- Entendendo e Criando um projeto Java
- Compilando

# Abstração e Modularização

## **Abstração**

Capacidade de ignorar detalhes de partes para focalizar a atenção em um nível mais elevado de um problema

## **Modularização**

Processo de dividir um todo em partes bem definidas, que podem ser construídas e examinadas separadamente, e que interagem de maneiras também bem definidas

## **Exemplo: Um carro**

Motor, assentos, volante, freios, pneus

Motor: cilindros, velas, eixos, injeção, eletrônica

# Abstração em software

## **Sistema orientado a objetos**

Objetos interagindo (trocando mensagens)

## **Quais tipos (classes) eu preciso?**

Não trivial



# Modularizando o mostrador do relógio

11:03

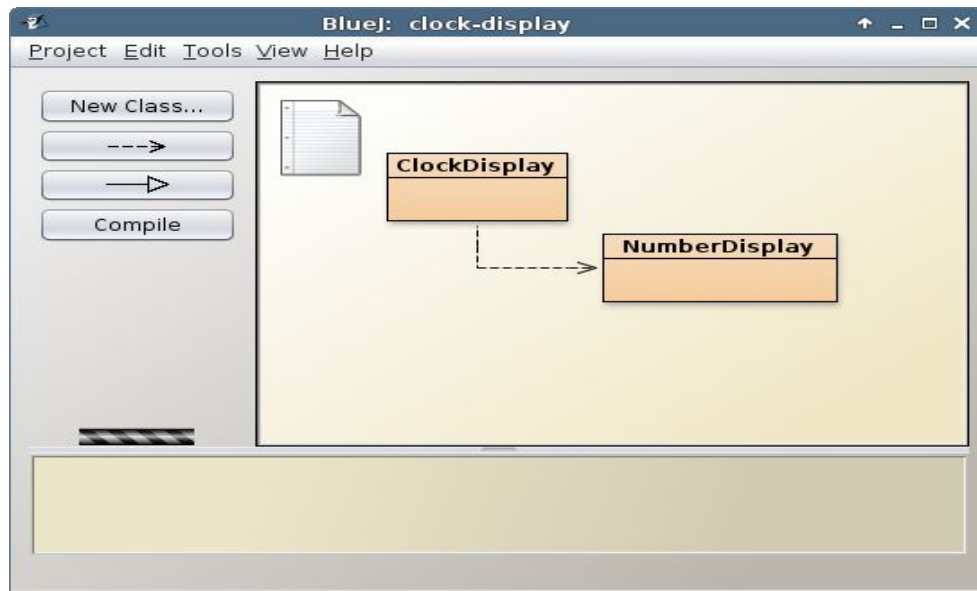
Um mostrador de quatro dígitos?

Dois mostradores de dois dígitos?

11 03

# Um relógio digital

- Projeto clock-display
- Abrir no BlueJ

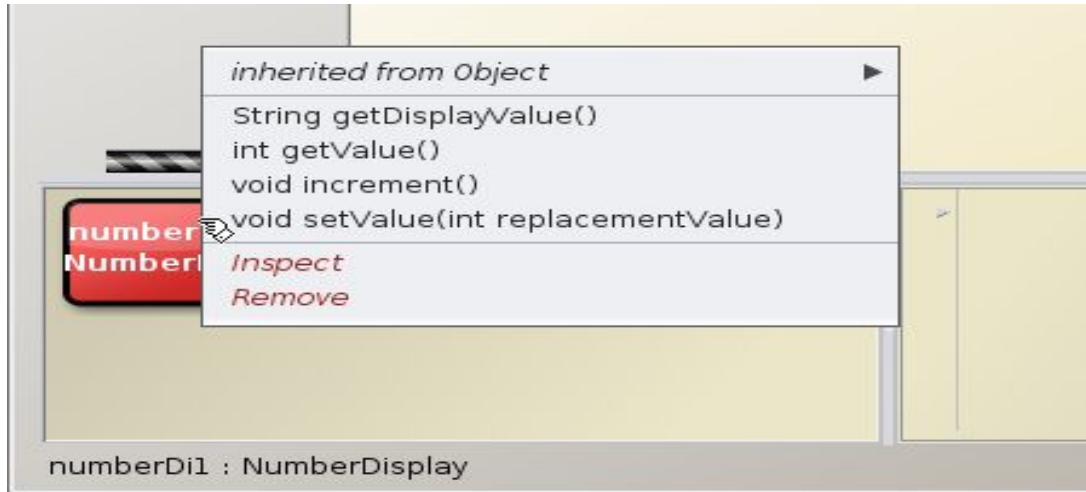


# Enxergando objetos

- Interagir com um objeto nos dá pistas sobre o seu comportamento
  - Visão externa
- Analisar as partes internas nos permite determinar como esse comportamento é fornecido (implementado)
  - Visão interna
- Todas as classes Java têm uma visualização interna semelhante
- Seguem uma estrutura básica

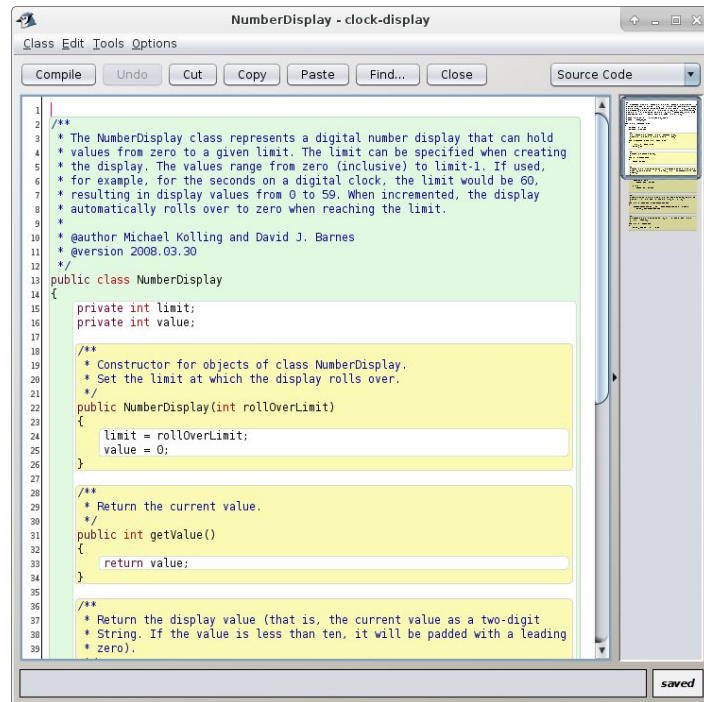
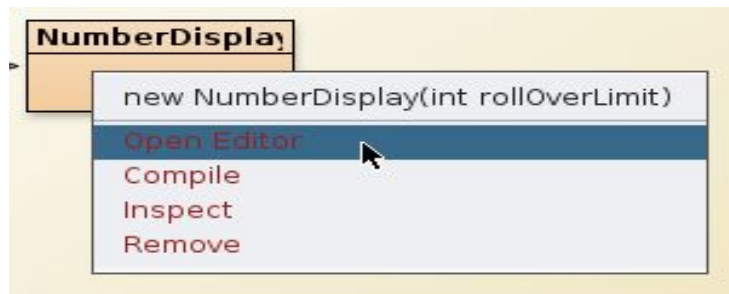
# Interagindo com objetos

- Interagir com um objeto nos dá pistas sobre o seu comportamento
- Visão externa
- Classe NumberDisplay



# Enxergando objetos

- Examinar o código de NumberDisplay



# Estrutura básica de uma classe

```
public class NumberDisplay
{
    A parte interna da classe omitida.
}
```

```
classe pública NomeDaClasse
{
    Campos
    Construtores
    Métodos
}
```

# Código-fonte: NumberDisplay

- Dois campos
  - `private int limit;`
  - `private int value;`
- Construtor que recebe um parâmetro
  - `public NumberDisplay(int rollOverLimit)`
- Métodos de acesso
  - `public int getValue()`
  - `public String getDisplayValue()`
- Métodos modificadores
  - `public void setValue(int replacementValue)`
  - `public void increment()`

# Campos

- Os campos armazenam dados para um objeto
- Campos também são conhecidos como variáveis de instância
- Utilize a opção *inspect* para visualizar os campos de um objeto
- Os campos definem o estado de um objeto

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

    //Detalhes adicionais omitidos
}
```

24

modificador  
de visibilidade      tipo      nome variável

private int limit;



# Construtores

- Construtores inicializam um objeto
- Têm o mesmo nome de sua classe
- Armazenam valores iniciais para os campos
- Costumam receber valores de parâmetro externo para isso

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

# Métodos de acesso

- Métodos implementam o comportamento de objetos
- Métodos de acesso fornecem informações sobre um objeto
- Métodos têm uma estrutura básica
  - Cabeçalho
    - Define a assinatura do método
    - `public int getValue()`
  - Corpo
    - Inclui as instruções do método

```
public int getValue()  
{  
    return value;  
}
```

# Código-fonte: NumberDisplay

- Métodos de acesso
- `public String getDisplayValue()`

modificador  
de visibilidade

tipo de  
retorno

nome do  
método

lista de parâmetro  
(vazia)

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

início e fim do corpo  
do método (bloco)

instrução  
de  
retorno

# Métodos modificadores

- Estrutura básica semelhante
- São usados para *modificar* o estado de um objeto
- São alcançados através da mudança de valor de um ou mais campos
  - Em geral, contêm instruções de atribuição
  - Normalmente, recebem parâmetros

# Código-fonte: NumberDisplay

- Métodos modificadores
- `public void setValue(int replacementValue)`

modificador  
de visibilidade

tipo de retorno

nome do  
método

parâmetro

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

campo sendo  
modificado

instrução de atribuição

- `public void increment()`

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

# Teste

- O que está errado aqui?

```
public class CokeMachine
{
    private price;

    public CokeMachine()
    {
        price = 300
    }

    public int getPrice
    {
        return Price;
    }
}
```

# Teste

- O que está errado aqui?

```
public class CokeMachine
{
    private price;

    public CokeMachine()
    {
        price = 300
    }

    public int getPrice
    {
        return Price;
    }
}
```

# Teste

- Alguma diferença?

```
public class CokeMachine
{
    private int price;

    public CokeMachine()
    {
        price = 300;
    }

    public int getPrice()
    {
        return price;
    }
}
```



# Antes de continuarmos...

- Observações

- Muitas **instâncias** podem ser criadas a partir de uma única classe
- Um objeto tem **atributos**: valores armazenados em *campos*
- A classe define quais campos um objeto tem
- Mas cada objeto armazena seu próprio conjunto de valores (o **estado** do objeto)

# Implementando o relógio

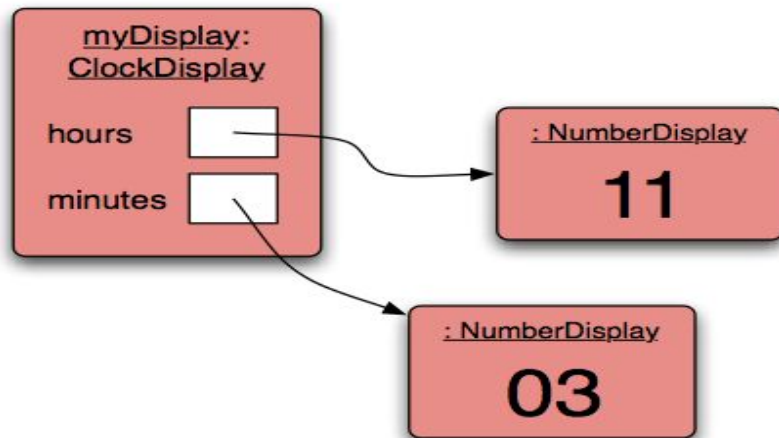
```
public class ClockDisplay
{
    private NumberDisplay hours;
    Private NumberDisplay minutes;

    //Construtores e métodos omitidos
}
```

# Diagrama de objetos/classes

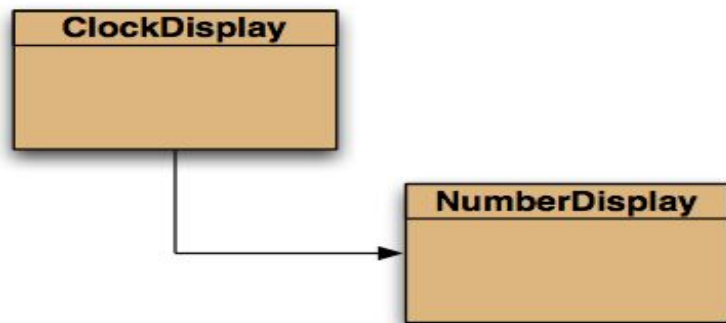
## Diagrama de objetos

- Visualização dinâmica
- Relação entre objetos
- Referência de objeto



## Diagrama de classes

- Visualização estática
- Relação entre classes

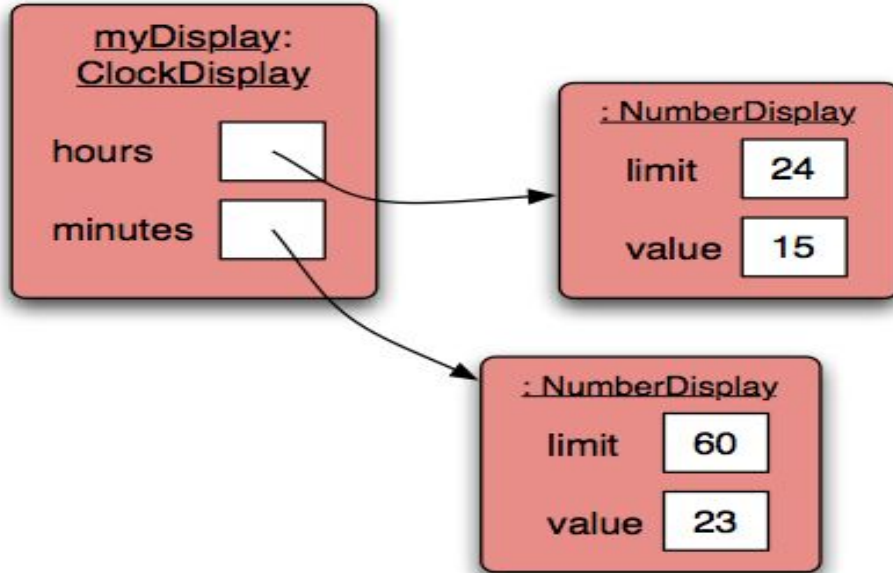


# ClockDisplay

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

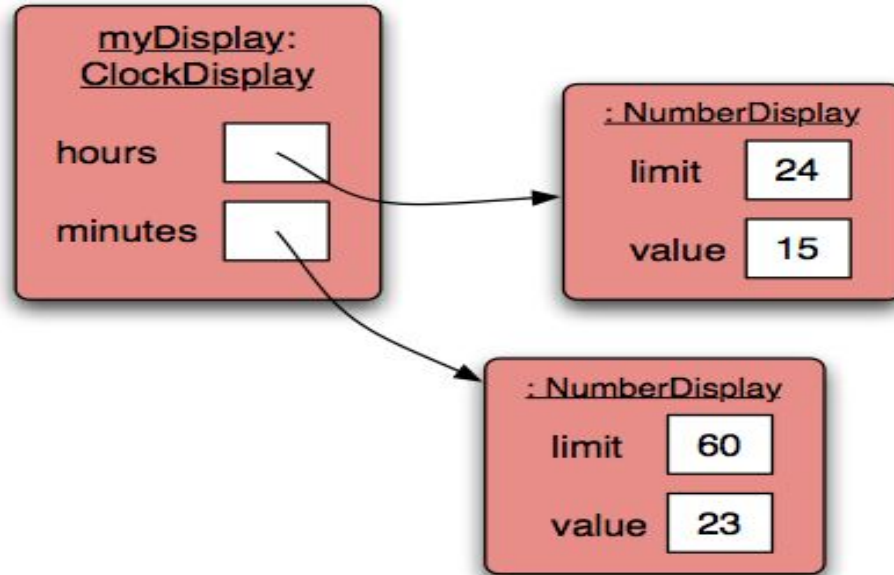
    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }
}
```

# Diagrama de objetos



# Diagrama de objetos

Quem cria esses objetos?



# Criação de Objetos

## Construtor

- Método chamado durante a criação de um objeto
- “primeiro” método chamado no objeto

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

## Operação “new”

- Cria um novo objeto da classe nomeada
- Executa o construtor da classe

# Criação de Objetos

## Na classe **NumberDisplay**

- `public NumberDisplay(int rollOverLimit);`
- Parâmetro formal

## Na classe **ClockDisplay**

- `hours = new NumberDisplay(24);`
- Parâmetro real



# Criação de Objetos

## Múltiplos construtores

- `new ClockDisplay();`
- `new ClockDisplay(hour, minute);`
- Maneiras alternativas de inicializar um objeto
- Sobrecarga de construtores
- Similar ao que vocês já conhecem de C++

# Imprimindo a partir de métodos

## Imprimindo em Java

- `System.out.println();`

## Imprime na tela o parâmetro recebido

- `System.out.println("Hello world");`
- `System.out.println("teste # 123");`

# Imprimindo a partir de métodos

```
public void meuMostrador()  
{  
    // Simule a impressão de um bilhete.  
    System.out.println("#####");  
    System.out.println("# O mostrador");  
    System.out.println("# Valor: " + value + ", ");  
    System.out.println("# Limite: " + limit + ".");  
    System.out.println("#####");  
    System.out.println();  
}
```

```
#####  
# O mostrador  
# Valor: 15,  
# Limite: 24.  
#####
```

# Concatenação de string

- 4 + 5
- 9
- “wind” + “ow”
- “window”
- “Resultado: ” + 6
- “Resultado: 6”
- “#” + price + “ cents”
- “# 500 cents”

# Teste

· System.out.println(5 + 6 + "hello");  
11hello

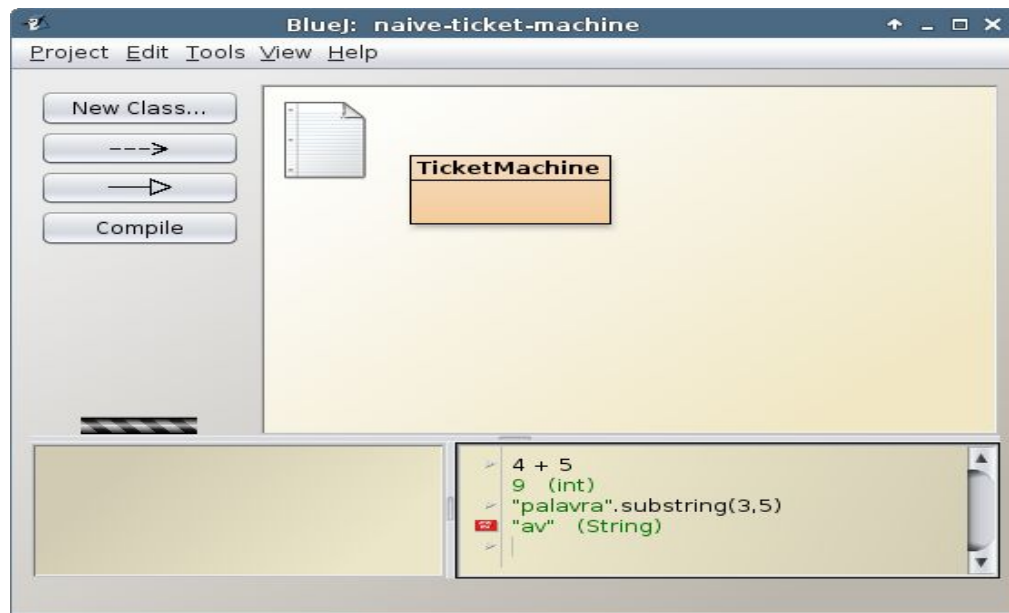
· System.out.println("hello" + 5 + 6);  
hello56

# Um pouco mais de BlueJ

- Testando expressões com o Code Pad

# Code Pad

- O BlueJ tem uma ferramenta que permite testar expressões: Code Pad



# Tipos em Java

## **Duas espécies de tipos**

- Tipos primitivos
  - Predefinidos pela linguagem
  - int, float, boolean
- Tipos de objeto
  - Definidos por classes
  - Classes padrão: String
  - Classes definidas pelo usuário



# Tipos em Java

- Diferenças entre tipos primitivos e de objeto
  - Armazenamento
  - Manipulação
- Tipos primitivos
  - Armazenado direto na variável
- Tipos de objeto
  - Armazenado através de uma referência

# Tipos em Java

- Qual o resultado?

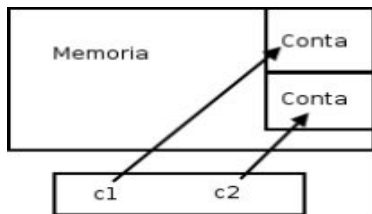
```
int a;  
int b;  
a = 32;  
b = a;  
a = a + 1;  
System.out.println(b);
```

```
Person a;  
Person b;  
a = new Person("Everett");  
b = a;  
a.changeName("Delmar");  
System.out.println(b.getName());
```

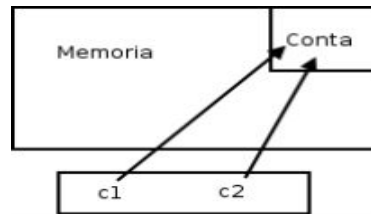
# Tipos em Java

## Outro exemplo

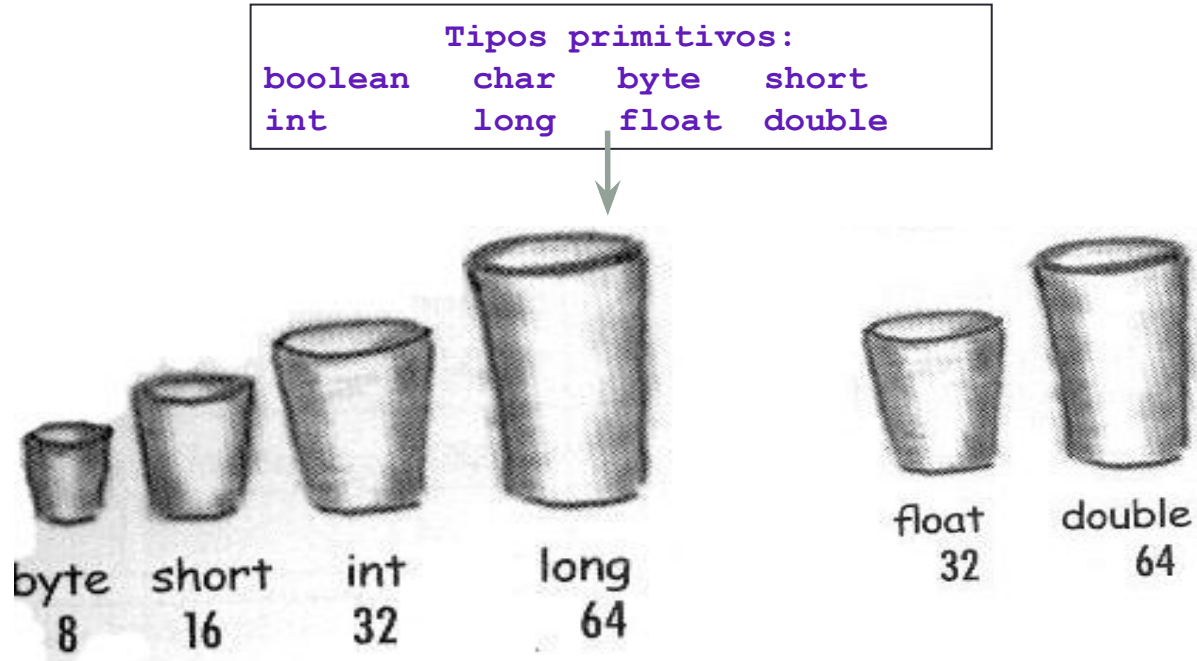
```
Conta c1;  
Conta c2;  
c1 = new Conta();  
c2 = new Conta();
```



```
Conta c1;  
Conta c2;  
c1 = new Conta();  
c2 = c1;
```



# Tipos em Java



# null

- null é um valor especial em Java
- Todas as variáveis de objetos são inicializadas como null
- Você pode atribuir e testar a existência de null

```
private NumberDisplay hours;  
  
if (hours == null) { ... }  
  
hours = null;
```

# Chamada de método

```
public void timeTick()
{
    minutes.increment();
    if(minutes.getValue() == 0) {
        hours.increment();
    }
    updateDisplay();
}
```

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

# Chamada de método

## Métodos externos

- minutes.increment();

## Métodos internos

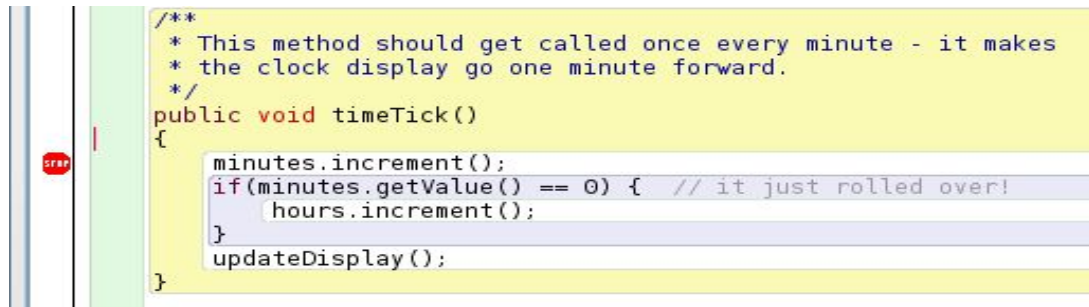
- updateDisplay();
- private*

## Forma geral:

- objeto.nomeDoMétodo( lista-de-parâmetros )

# Depurador

- Debugger
- Permite rastrear a execução de um programa
- “Executar linha a linha”
- Verificar valores de variáveis
- Pontos de interrupção

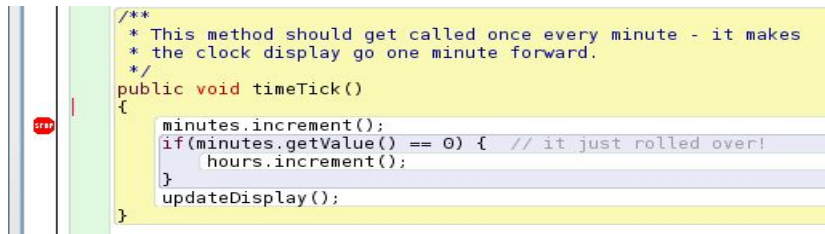


```
/**
 * This method should get called once every minute - it makes
 * the clock display go one minute forward.
 */
public void timeTick()
{
    minutes.increment();
    if(minutes.getValue() == 0) { // it just rolled over!
        hours.increment();
    }
    updateDisplay();
}
```



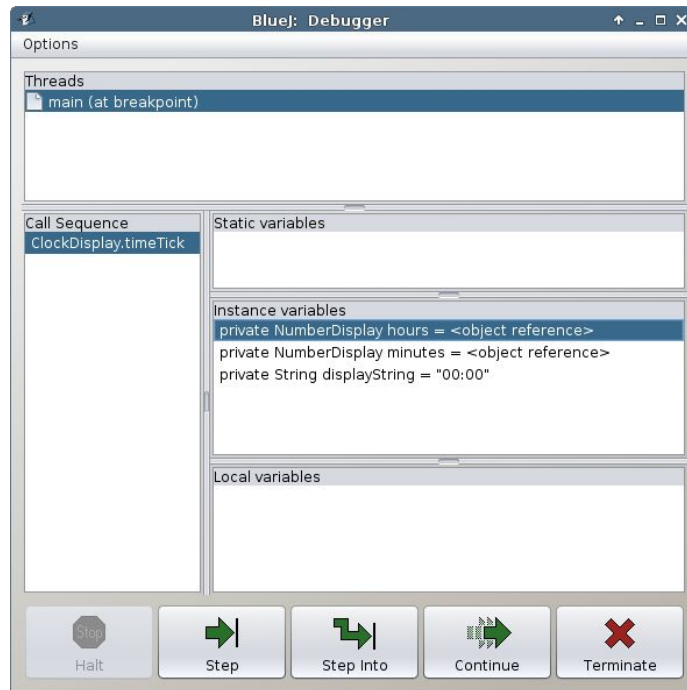
# Depurador

- Interface de debugger do BlueJ



The screenshot shows a code editor window with a yellow background. On the left margin, there is a red 'crash' button. The code is as follows:

```
/**
 * 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();
}
```



# Até aqui

- Capítulos 1, 2 e 3 do livro do BlueJ
- **Conceitos**
  - Criação de objeto, sobrecarga
  - Chamada de método interno/externo
  - Campos, construtores, métodos, parâmetros
  - Abstração, modularização
  - Chamadas de método
  - Diagramas de classe/objeto

**Por hoje é só...**