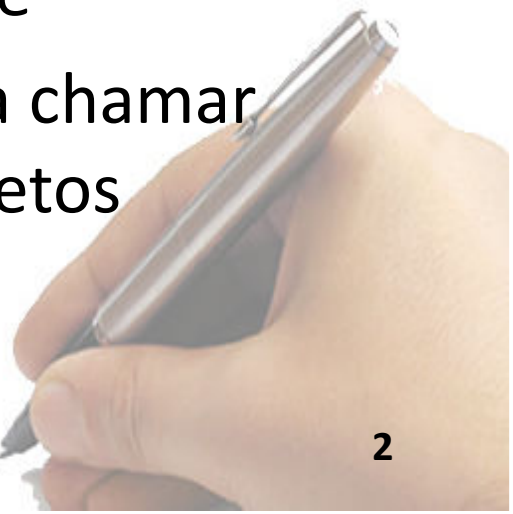
Programação Orientada a Objetos e a Linguagem Java

Rodrigo da Cruz Fujioka
fujiokabr@gmail.com

Criação e Destruição de Objetos.

- Este módulo explora detalhes da construção de classes e objetos:
  - Construtores
  - Implicações da herança
  - Palavras super e this, usadas como referências para o objeto corrente e a super classe
  - Instruções super() e this() usadas para chamar construtores durante a criação de objetos



Criação e Destruição de Objetos.

- Operações especiais: construtores
  - Utilizados para a inicialização de objetos
  - Executados automaticamente durante a criação de um objeto
  - Sem valor de retorno
  - Possuem o MESMO nome da classe

- Quando não definimos um construtor para uma classe, ela terá, implicitamente, um construtor padrão.

Criação e Destruição de Objetos.

- Para a criação de novos objetos, Java garante que cada classe tenha um construtor
  - O construtor default recebe zero argumentos
  - Faz apenas a inicialização da superclasse

- Programador pode criar um construtor explicitamente e determinar suas operações de inicialização
  - Inicialização pela superclasse continua garantida
  - Construtor default deixa de existir

- Quando definimos um construtor explicitamente, a classe NÃO terá o construtor padrão.

Criação e Destruição de Objetos.


```
class Rock {
 Rock(int i) {
    System.out.println("Creating Rock
  number " + i);
   }
}
public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
     new Rock(i);
   }
}
```

Criação e Destruição de Objetos.

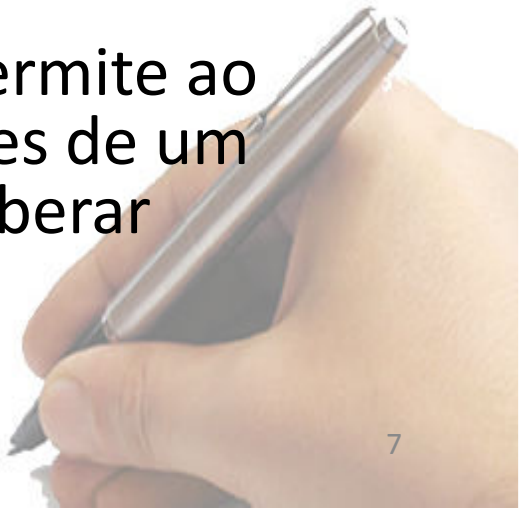

```
C:\...>javac SimpleConstructor.java
C:\...>java SimpleConstructor
Creating Rock number 0
Creating Rock number 1
Creating Rock number 2
Creating Rock number 3
Creating Rock number 4
Creating Rock number 5
Creating Rock number 6
Creating Rock number 7
Creating Rock number 8
Creating Rock number 9
```

Criação e Destruição de Objetos.

- Objetos são destruídos automaticamente.

- Algumas vezes, antes de um objeto ser destruído, precisamos liberar alguns recursos que foram alocados por este objeto. Exemplos:
  - Fechar uma conexão com um banco de dados.
  - Fechar um arquivo.

- O método finalize(), herdado de Object, permite ao programador realizar algumas tarefas antes de um objeto ser coletado, como por exemplo, liberar recursos alocados anteriormente.



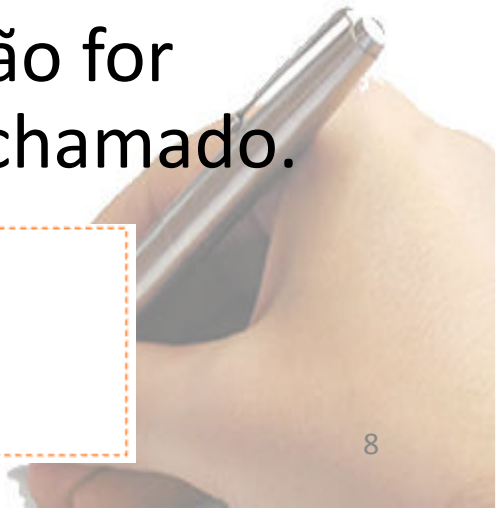
Criação e Destruição de Objetos.

- O método **finalize()** é sempre chamado antes do garbage collector coletar o objeto.

- O **coletor de lixo** não tem hora certa para executar. Não é possível determinar a hora exata de sua execução.

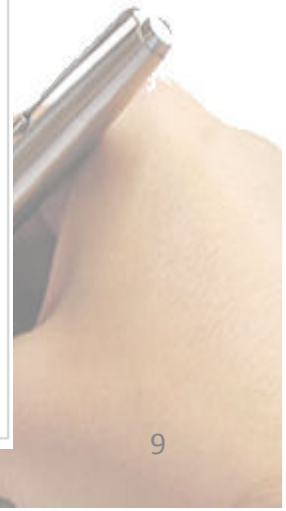
- Se o objeto (por uma razão qualquer) não for coletado, o método **finalize()** não será chamado.

```
protected void finalize()
{
// libera recursos alocados
}
```



Criação e Destruição de Objetos.


```
class Book {
  boolean checkedOut = false;
  Book( boolean checkOut) {
    checkedOut = checkOut;
  }
  void checkIn() {
    checkedOut = false;
  }
  public void finalize() {
    if(checkedOut)
      System.out.println("Error: checked out");
  }
}
```



Criação e Destruição de Objetos.

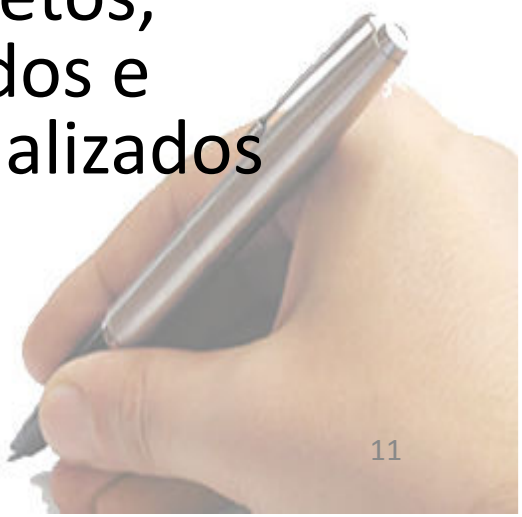

```
public class DeathCondition {
     public static void main(String[] args) {
       Book b1 = new Book(true);
       b1.checkIn();
       Book b2 = new Book(true);
       b1 = null;
       b2 = null;
       // Força coleta de lixo e, conseqüentemente, a
                 // finalização
       System.gc();
}
```

```
C:\...>javac DeathCondition.java
C:\...>java DeathCondition
Error: checked out
```

Criação e Destruição de Objetos.

- **System.gc()**
  - Chama explicitamente o coletor de lixo para coletar todos os objetos que não são mais referenciados (**objetos lixo**)
  - Antes de cada **objeto lixo** ser coletado o seu método **finalize** será executado

- Se a execução do coletor de lixo não tivesse sido forçada, através do **System.gc()**, os objetos, possivelmente, não teriam sido coletados e conseqüentemente não teriam sido finalizados



Sobrecarga de métodos

- Um classe pode ter vários métodos com o mesmo nome.
- Isto é o que chamamos sobrecarga (overload) de métodos
- Exemplo: método lavar. Nós podemos lavar o carro, os pratos, a roupa, etc.
  - Solução sem sobrecarga: **lavarCarro(), lavarPratos(), lavarRoupa()**
  - Solução com sobrecarga: **lavar(Carro c), lavar(Pratos p), lavar(Roupa r)**

Sobrecarga de métodos

- A assinatura de um método consiste do:
  - Tipo de retorno
  - Nome
  - Tipo de argumentos
  - Quantidade de argumentos
- Na sobrecarga a distinção entre os métodos é feita APENAS pelo número e tipo de argumentos

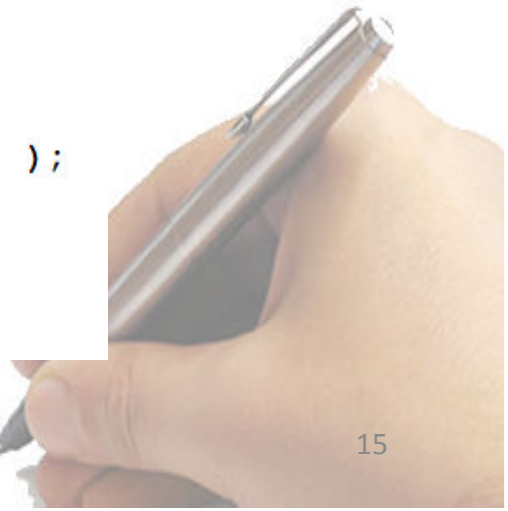
Sobrecarga de Construtores

- Uma classe pode ter vários construtores. Nestes casos teremos uma sobrecarga (overload) de construtores.

- Na sobrecarga de construtores a distinção entre construtores também é feita pelo número e tipo de argumentos.

Sobrecarga de Construtores


```
import java.util.*;
class Tree {
  int height;
  Tree() {
    System.out.println( "Planting a seedling" );
    height = 0;
  }
  Tree( int i) {
    System.out.println ("Creating new Tree that is" +
                i + " feet tall");
    height = i;
  }
  void info() {
    System.out.println ( "Tree is " + height + " feet tall" );
  }
```



Sobrecarga de Construtores


```
void info(String s) {
    System.out.println (s +  "Tree is " + height +
                "  feet tall" );
  }
public class Overloading {
  public static void main(String[] args) {
    for(int i = 0; i < 4; i++) {
      Tree t = new Tree(i);
      t.info();
      t.info( "overloaded method" );
    }
    new Tree();
  }
}
```

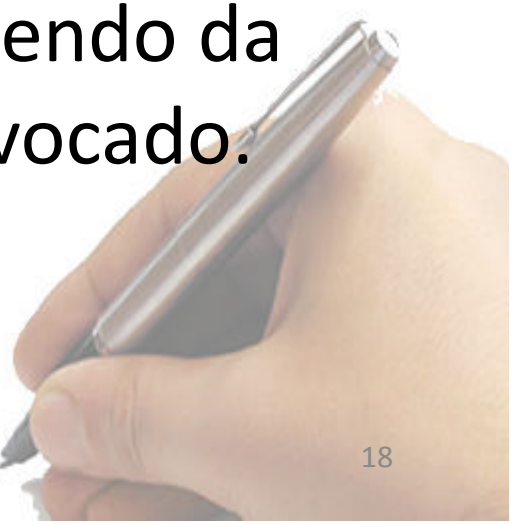
Sobrecarga de Construtores


```
C:\...>java Overloading
Creating new Tree that is 0 feet tall
Tree is 0 feet tall
overloaded method: Tree is 0 feet tall
Creating new Tree that is 1 feet tall
Tree is 1 feet tall
overloaded method: Tree is 1 feet tall
Creating new Tree that is 2 feet tall
Tree is 2 feet tall
overloaded method: Tree is 2 feet tall
Creating new Tree that is 3 feet tall
Tree is 3 feet tall
overloaded method: Tree is 3 feet tall
Planting a seedling
```

This

- A palavra-chave **this** é usada para acessar o objeto atual.

- O **this** não pode ser usado em um método estático, pois como o método é estático ele pertence a uma classe, não dependendo da existência de um objeto para ser invocado.



This


```
public class Leaf {
  int i = 0;
  public Leaf  increment() {
    i++;
    return this;
  }
  public static void main(String[] args) {
    Leaf x = new Leaf();
    System.out.println("i = " +
                       x.increment().increment().i);
  }
}
```

```
C:\...>java Leaf
i = 2
```

This

- Em classes com múltiplos construtores, podemos usar this(*parametros*) *dentro de um construtor para* chamar outro construtor da mesma classe.
- O this(*parametros*) *deve ser a primeira linha dentro* de um construtor e não é permitido usá-lo duas vezes dentro do mesmo construtor.

This

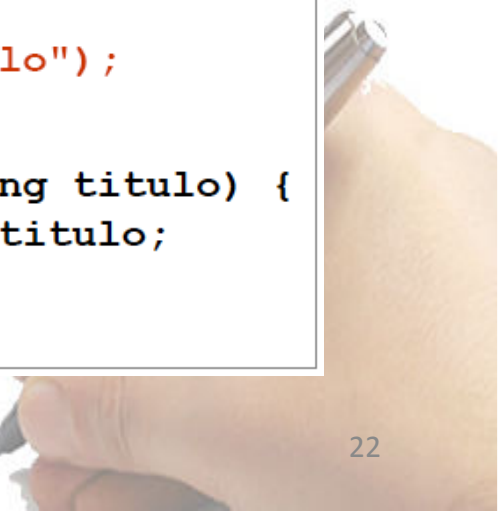

```
public class Circle {
    private int x;
    private int y;
    private double radius;
    public Circle( int x, int y, double radiusValue )
      {
      this (x, y);
      setRadius( radiusValue );
    }
    public Circle( int x, int y ) {
      this.x = x;
      this.y = y;
    }
    public void SetRadius( double radius )
    {
      this.radius = radius;
    }

}
```

This

- Em classes com múltiplos construtores, que realizam tarefas semelhantes, this() pode ser usado para chamar outro construtor local, identificado pela sua assinatura (número e tipo de argumentos)

```
public class Livro {
   private String titulo;
   public Livro() {
      titulo = "Sem titulo";
   }

   public Livro(String titulo) {
      this.titulo = titulo;
   }
}
```

```
public class Livro {
   private String titulo;
   public Livro() {
      this("Sem titulo");
   }

   public Livro(String titulo) {
      this.titulo = titulo;
   }
}
```



Super

- Todo construtor chama algum construtor de sua superclasse
  - Por default, chama-se o construtor sem argumentos, através do comando super() (implícito)
  - Pode-se chamar outro construtor, identificando-o através dos seus argumentos (número e tipo) na instrução super()
  - super(), se presente, deve sempre ser a primeira instrução do construtor (substitui o super() implícito)
- Se a classe tiver um construtor explícito, com argumentos, subclasses precisam chamá-lo diretamente
  - Não existe mais construtor default na classe

Super


```
class Shape {
   private int x;
   private int y;
   public Shape ( int x, int y) {
     mover(x,y);
   }
   public void mover( int x, int y) {
     this.x = x;
     this.y = y;
   }

}
public classe Circle extends Shape {
   private double radius;
   public Circle( int x, int y, double radiusValue ) {
     super (x, y);
     this.radius = radius;
   }

}
```

Super

- A palavra super também é usada para referenciar variáveis ou métodos da classe mãe

```
class Numero {
    public int x = 10;
}
```

```
class OutroNumero extends Numero {
    public int x = 20;
    public int total() {
        return this.x + super.x;
    }
}
```

20

10

```
public class Circle extends Shape {
    public void moverPosicaoInicial() {
      super.mover (0, 0);
    }

}
```

This e Super

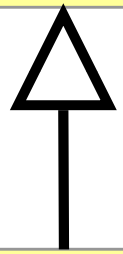
Não confunda this e super com this() e super()

- Os últimos são usados apenas em construtores!

This e Super


```
public class Ponto {
  private int x, y;
  public String toString()
   { return “Coordenadas:” + “[” + x + “,” + y + “]”; }
}
```

Ponto.java

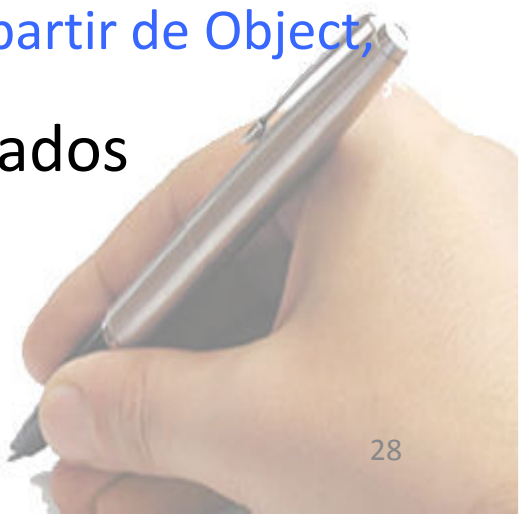


```
public class Circulo extends Ponto {
  private double raio;
  public String toString()
   { return  super.toString() + “Raio:” + raio; }
}
```

Circulo.java

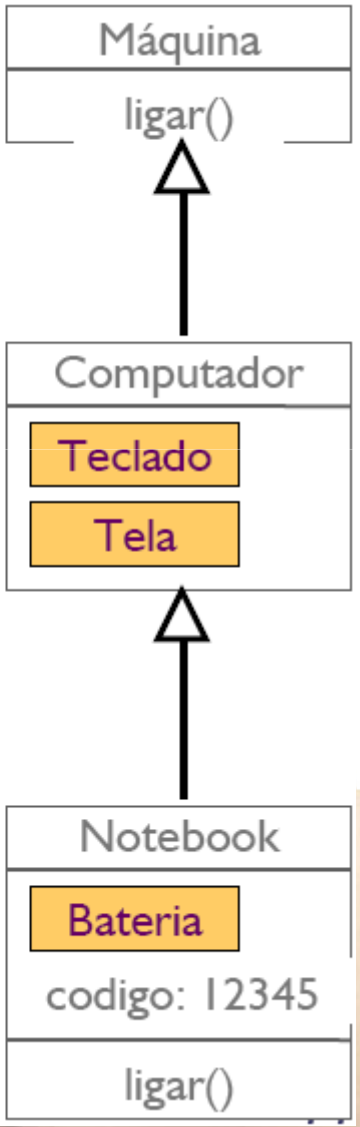
Inicialização de Instâncias

- O que acontece quando um objeto é criado usando new NomeDaClasse()?
  - 1. Inicialização default de campos de dados (0, null, false)
  - 2. Chamada recursiva ao construtor da superclasse (até Object)
    - 2.1 Inicialização default dos campos de dados da superclasse (recursivo, subindo a hierarquia)
    - 2.2 Inicialização explícita dos campos de dados
    - 2.3 Execução do conteúdo do construtor (a partir de Object, descendo a hierarquia)
  - 3.Inicialização explícita dos campos de dados
  - 4.Execução do construtor



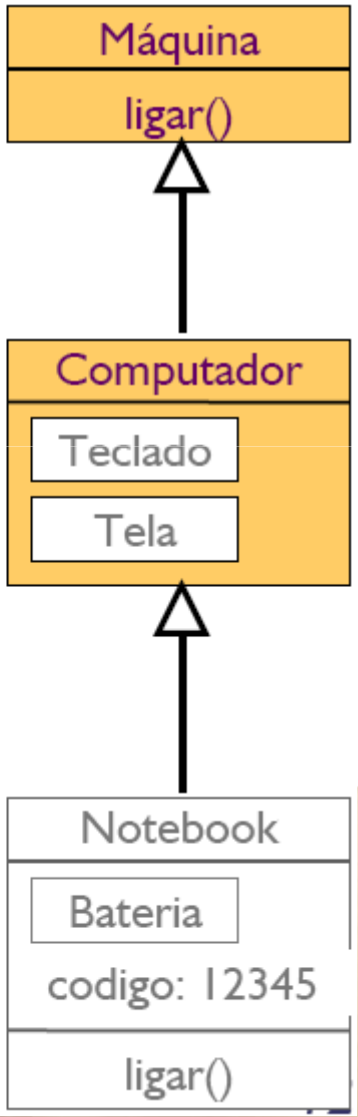
Exemplo (1)


```
class Bateria {
  public Bateria() {
    System.out.println("Bateria()");
  }
}

class Tela {
  public Tela() {
    System.out.println("Tela()");
  }
}

class Teclado {
  public Teclado() {
    System.out.println("Teclado()");
  }
}
```



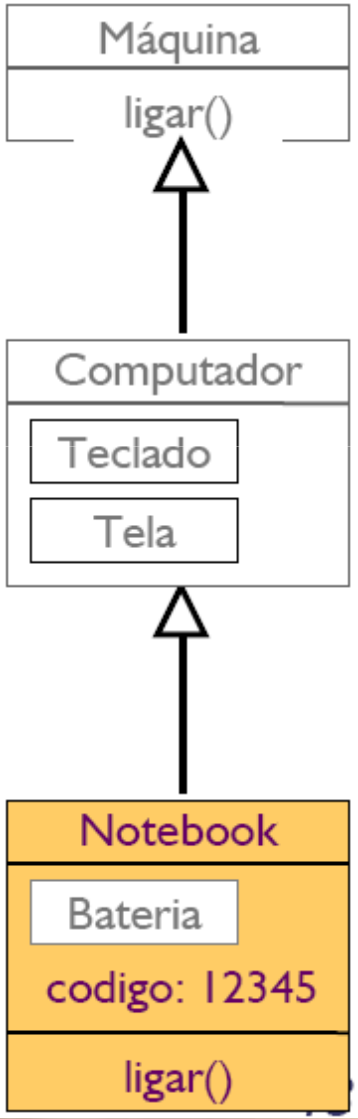
Exemplo (2)


```
class Maquina {
  public Maquina() {
     System.out.println("Maquina()");
     this.ligar();
  }
  public void ligar() {
    System.out.println("Maquina.ligar()");
  }
}

class Computador extends Maquina {
  public Tela tela = new Tela();
  public Teclado teclado = new Teclado();
  public Computador() {
     System.out.println("Computador()");
  }
}
```

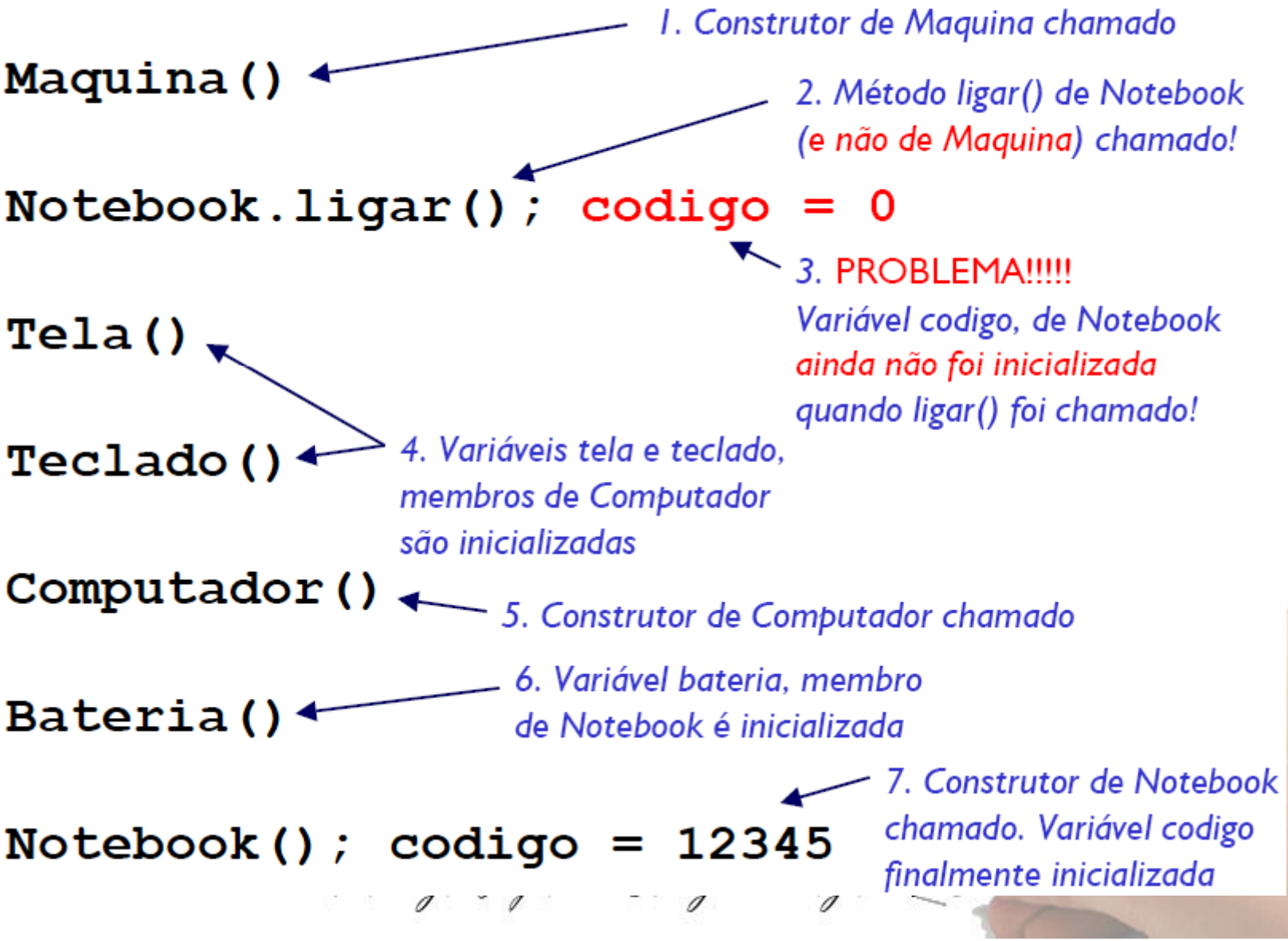


Exemplo(3)


```
class Notebook extends Computador {
  int codigo = 12345;
  public Bateria bateria = new Bateria();
  public Notebook() {
    System.out.print("Notebook(); " +
                     "codigo = "+codigo);
  }
  public void ligar() {
    System.out.println("Notebook.ligar();" +
                       " codigo = "+ codigo);
  }
}

public class Run {
  public static void main (String[] args) {
  new Notebook();
  }
}
```



Resultado de new Notebook()



Detalhes

N1. new Notebook() chamado
N2. variável código iniciaizada: 0
N3. variável bateria iniciaizada: null
N4. super() chamado (Computador)

C1. variável teclado iniciaizada: null
C2. variável tela iniclaizada: null
C3. super() chamado (Maquina)

M2. super() chamado (Object)

M2. Corpo de Maquina() executado:
println() e this.ligar()

C4: Construtor de Teclado chamado

Tk1: super() chamado (Object)

C5. referência teclado inicializada
C6: Construtor de Tela chamado

Te1: super() chamado (Object)

C7: referência tela inicializada
C8: Corpo de Computador()
executado: println()

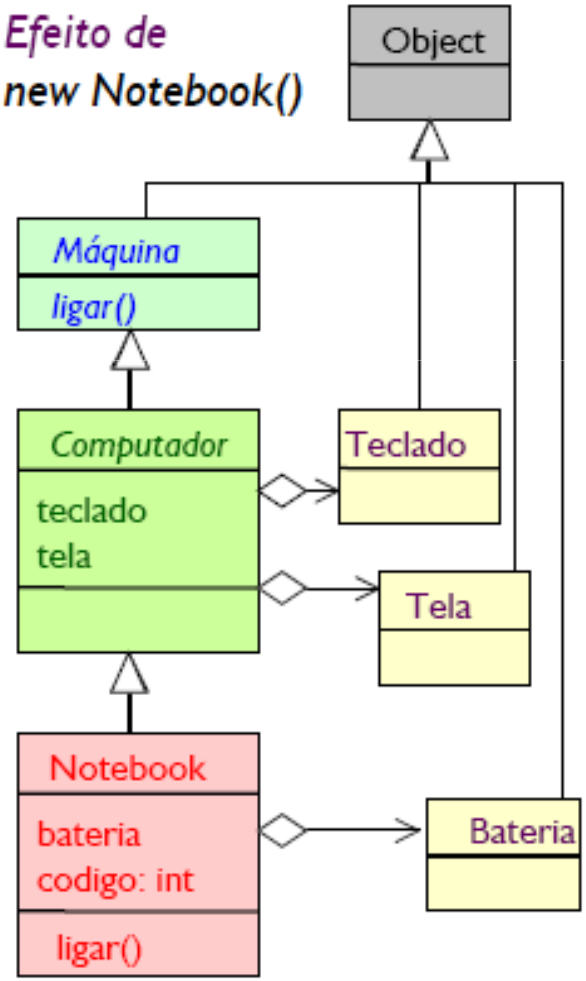
N5. Construtor de Bateria chamado

B1: super() chamado (Object)

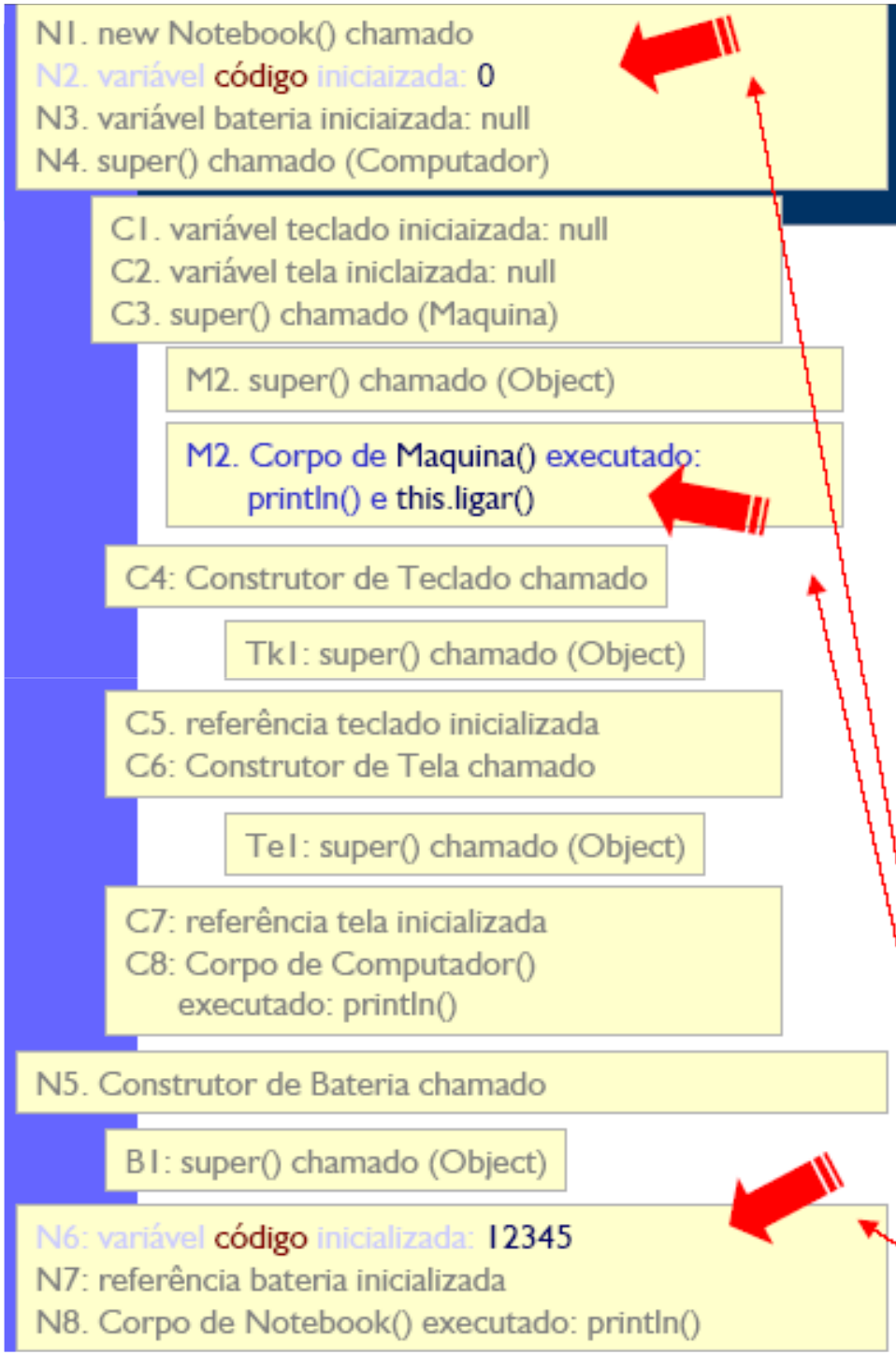
N6: variável código inicializada: 12345
N7: referência bateria inicializada
N8. Corpo de Notebook() executado: println()

O1. Campos inicializados
O2. Corpo de Object() executado

Efeito de new Notebook()



Problemas com inicialização



- *método ligar() é chamado no construtor de Maquina, mas ...*
- *... a versão usada é a implementação em Notebook, que imprime o valor de código (e não a versão de Maquina como aparenta)*
- *Como código ainda não foi inicializado, valor impresso é 0!*

Preste atenção nos pontos críticos!

Inicialização de Instâncias

- Na inicialização de um atributo:
  - Pode ser usado um valor (54, 2, “um valor”, true)
  - É permitido a chamada de um método ou a criação de um objeto

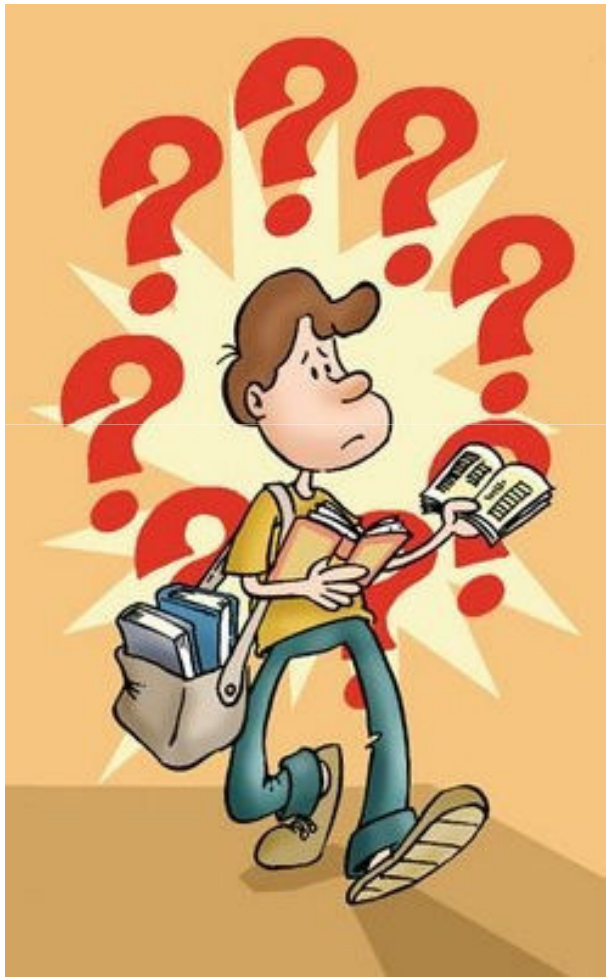
```
class Measurement{
  Depth depth = new Depth(); //criação de um objeto
  boolean b = true; //valor
  int j = 10; //valor
  int i = f(); //Chamada a um método
```

Inicialização de Instâncias

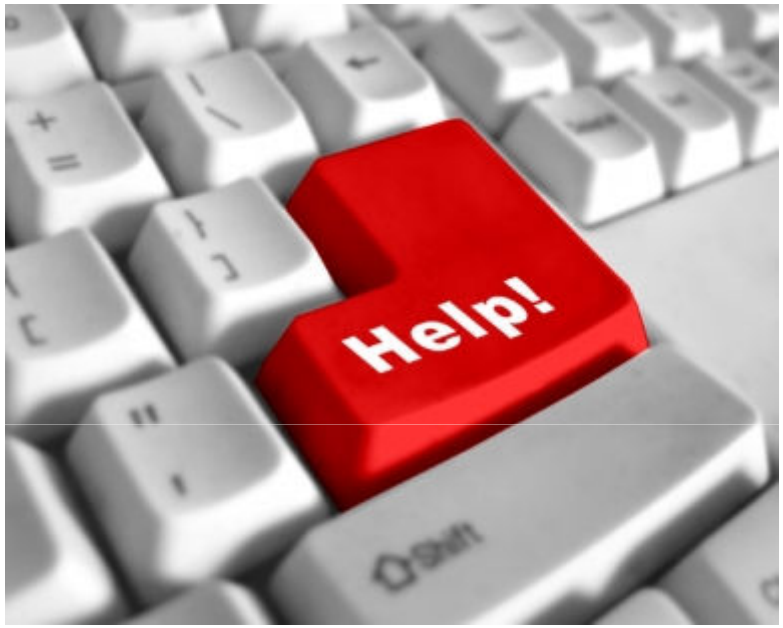
- Na inicialização de um atributo:
  - Pode ser usado um valor (54, 2, “um valor”, true)
  - É permitido a chamada de um método ou a criação de um objeto

```
class Measurement{
  Depth depth = new Depth(); //criação de um objeto
  boolean b = true; //valor
  int j = 10; //valor
  int i = f(); //Chamada a um método
```

Dúvidas?



Ajuda!



rodrigofujioka@gmail.com
ou
rcf4@cin.ufpe.br