Tópicos selecionados de programação em Java

Threads e o Modelo de Consistência de Memória da Plataforma Java

Helder da Rocha

Agosto 2005

Objetivos

- Apresentar os conceitos fundamentais do Modelo de Consistência de Memória da plataforma Java (**JMM – Java Memory Model**)
- Discutir estratégias para construir programas corretamente sincronizados usando a nova especificação da JMM

Parte I

O que é o Java Memory Model

O que é um Modelo de Consistência de Memória?

- Modelo que define os valores que podem ser retornados na leitura de uma variável compartilhada
- É necessário para permitir otimizações do compilador e do processador
- Otimizações não afetam o modelo de memória de sistemas uniprocessados
  - Em sistemas seqüenciais, uma variável sempre possui o valor da última gravação
- Mas os mesmos compiladores e processadores são usados em sistemas paralelos
  - Em sistemas paralelos (multiprocessados, com buffers de gravação paralelos, etc.) o valor de uma variável pode ser indeterminado devido a reordenações e otimizações agressivas

Modelos de Consistência de Memória

- Consistência Sequencial (**CS**)
  - Modelo mais restritivo: não permite reordenamento de operações de memória dentro de um thread
  - Equivalente a um sistema seqüencial
- Outros modelos relaxam a CS para melhorar a performance
  - **TSO** (Total Store Order): permite que gravações sejam feitas em paralelo com leituras
  - **PSO** (Partial Store Order): permite que gravações sejam feitas em paralelo com outras gravações
  - **WO** (Weak Ordering): permite leituras não bloqueadas
  - **RC** (Release Consistency): não garante a ordem entre operações de travamento e operações de dados que a antecedem (e vice-versa)

Modelos de Memória para Linguagens

- Modelos de memória para processadores e para linguagens devem ser diferentes
- Garantias de alto-nível
  - Garantia de type-safety
  - Garantia de atomicidade de operações em estruturas invisíveis aos programadores (ex: operações em ponteiros em Java)
  - Garantia de semântica especial (ex: volatile, final) sem que seja necessário usar recursos de baixo nível para garanti-las (memory barrrier)
  - Garantia de compatibilidade com transformações realizadas por compiladores

Modelo de Memória em Linguagens

- Descrevem relacionamento entre variáveis de um programa e a memória de um computador
  - As variáveis são campos estáticos, campos de instância e elementos de vetores
  - As operações de memória são essencialmente a recuperação e gravação de dados
- O modelo de memória para Java deve garantir que
  - Seja impossível um thread ver uma variável antes que ela tenha sido inicializada a seu valor default (0, null)
  - O fato que a coleta de lixo pode realocar uma variável para uma nova localidade é imaterial e invisível ao modelo de memória

Por que Java tem um Modelo de Consistência de Memória?

- A linguagem Java suporta multithreading em memória compartilhada
  - Threads podem executar em sistemas uniprocessados ou em multiprocessados
- Mas sistemas multiprocessados possuem diferentes modelos de memória
  - Para garantir consistência seqüencial em todos eles seria necessário desabilitar certas otimizações explicitamente
  - Isto não é responsabilidade do programador Java
    - Contrato Write Once, Run Anywhere
    - Java não tem instruções MemBar (Memory Barrier)
- **Modelo de Memória do Java (JMM)** garante uma interface consistente, independente de plataforma, para o programador de aplicações

O que é o Java Memory Model?

- É uma especificação
  - Parte do capítulo 17 da Java Language Specification e capítulo 8 da Java Virtual Machine Specification
  - **Foi completamente reescrito para a versão 5.0! (JSR-133)**
- Define o conjunto de todos os comportamentos válidos que uma aplicação Java pode apresentar em qualquer plataforma
  - **Não é SC**: Comportamentos válidos podem produzir resultados supreendentes (que violam a SC) – parecido com RC!
  - Comportamentos válidos podem ou não ser resultados de programas corretos: a JMM define a semântica para programas corretos (sem data-races) e incorretos
  - Permite otimizações agressivas por parte dos multiprocessadores e compiladores e previsibilidade por parte dos programadores

Por que a JMM foi reescrita?

- Porque era excessivamente “forte”
  - Impedia otimizações comuns nos processadores e compiladores atuais
  - Era confusa e difícil de entender, e propunha controles muito difíceis de implementar (caros) o que levou a implementações incorretas
- Porque era excessivamente “fraca”
  - Não tinha uma semântica clara sobre programas incorretamente sincronizados (não usar synchronized poderia produzir resultados imprevisíveis e incompatíveis)
  - **volatile** não funcionava (não era clara a especificação e levou a implementações diferentes)
  - **final** não funcionava (poderia exibir valores diferentes ao longo do programa – conseqüência: String só no Java 5.0 é imutável)

Requerimentos do JMM

- Definição de **programas corretamente sincronizados**
  - Garantir consistência sequencial para programas corretamente sincronizados (livres de data-races)
  - Programadores só precisam se preocupar que eventuais otimizações do sistema terão impacto no seu código se esse código tiver data-races
- Garantias para **programas incorretos**
  - O foco da revisão do modelo de memória no Java 5.0
  - Determina como código deve se comportar quando não estiver corretamente escrito sem limitar excessivamente as otimizações realizadas pelos compiladores e hardware existentes
  - Busca garantir que um valor inesperado não surja “do nada” (a garantia é baseada em implementações atuais)

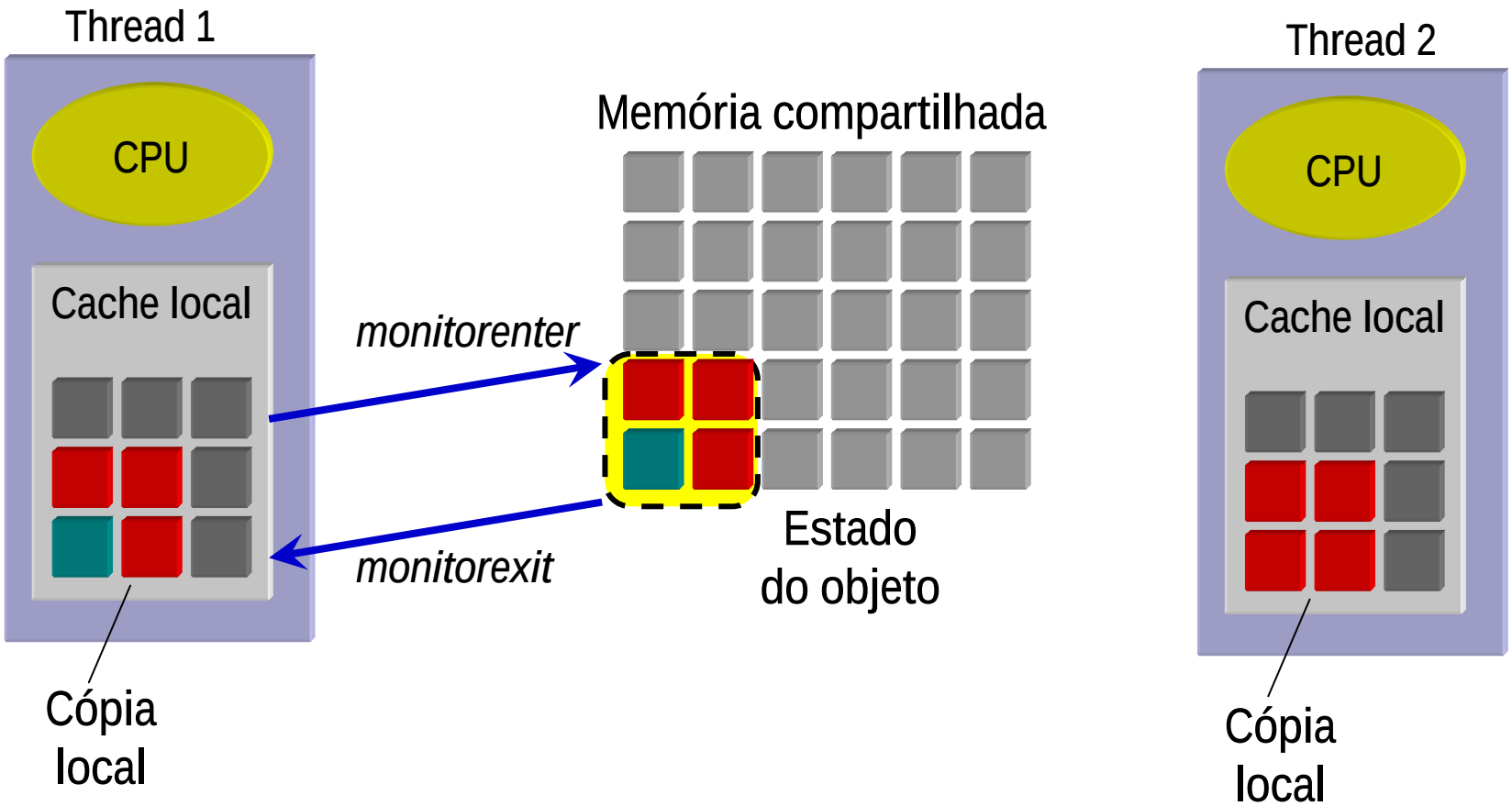
Comportamento de programas corretamente sincronizados

- Dentro de um **método ou bloco sincronizado**, leitura de memória compartilhada deve ler o valor da memória principal
  - Antes que um método ou bloco sincronizado termine, variáveis gravadas no seu interior devem ser escritas de volta na memória principal
- O sistema tem toda a liberdade para otimizar e reordenar o código de cada thread, desde que mantenha semântica equivalente ao funcionamento sequencial
  - Em programas sequenciais, o programador não será capaz de detectar otimizações e reordenações
  - Em sistemas concorrentes, **elas irão se manifestar**, a não ser que o programa esteja corretamente sincronizado

JMM: abstração fundamental

- Cada thread age como uma CPU virtual, que tem acesso a um **cache de memória local**, e à **memória principal** que é compartilhada entre todos os threads
  - Em multiprocessadores, as CPUs virtuais podem ser mapeadas pela JVM a CPUs reais
- O cache local é usado para guardar cópias dos dados que residem na memória principal compartilhada
- A JMM é uma **abstração** de dados guardados em registradores ou caches locais de um sistema multiprocessado
  - O sistema real pode ser implementado de outra forma mas deve comportar-se como a abstração

Modelo de Memória do Java



Fonte da imagem: [8] e [9] (Doug Lea)

Garantias da JMM: três aspectos essenciais da sincronização

- Atomicidade
  - Travar objeto para obter exclusão mútua durante operações críticas
- Visibilidade
  - Garantir que mudanças a campos de objetos feitos em um thread sejam vistos em outros threads
- Ordenação
  - Garantir que não haverá surpresa devido à ordem em que as instruções são executadas

JSR-133: Java Memory Model
Semântica informal: sincronização

1. Todos os objetos Java agem como **monitores** que suportam travas reentrantes
   - As únicas operações que podem ser realizadas no monitor são ações de travamento e destravamento
   - Uma ação de travamento bloqueia outros threads até que consigam obter a liberação
2. **Atomicidade**: As ações em monitores e campos voláteis são executados de maneira **seqüencialmente consistente**
   - Existe uma única, total e global ordem de execução sobre essas ações que é consistente com a ordem em que as ações ocorrem em seus threads originais
   - Ações sobre campos voláteis são sempre imediatamente visíveis em outros *threads*, e não precisam ser guardados por sincronização

JSR-133: Java Memory Model

Semântica informal: sincronização

3. **Visibilidade**: Se dois threads acessarem uma variável, e um desses acessos for uma gravação, então o programa deve ser sincronizado para que o primeiro acesso seja **visível** ao segundo
   - Quando um thread t1 adquire uma trava para um monitor m que era previamente mantido por outro thread t2, todas as ações visíveis a t2 no momento da liberação de m tornam-se visíveis a t1
4. **Ordenação**: Se um thread t1 inicia um thread t2, ações visíveis a t1 no momento em que ele inicia t2 tornam-se visíveis a t2 **antes** de t2 iniciar
   - Se t1 espera t2 terminar através de uma operação join(), todos os acessos visíveis a t2 quando terminar serão visíveis a t1 quando o join() terminar

JSR-133: Java Memory Model

Semântica informal: volatile e final

5. **Semântica de volatile**: Quando um thread t1 lê um campo volatile v que foi previamente gravado por um thread t2, todas as ações que eram visíveis a t2 no momento em que t2 gravou em v tornam-se visíveis a t1
   - Existe uma ordenação entre blocos sincronizados e campos voláteis
6. **Semântica de final**: Campos finais não podem retornar valores diferentes em nenhuma fase de sua construção
   - Quando um campo final for lido, o valor lido é o valor atribuído no construtor (nunca será retornado o tipo default do campo)
   - É preciso garantir que o construtor para um objeto foi concluído antes que outro objeto possa carregar uma referência para esse objeto

Conseqüências da nova JMM

- **Volatile** pode ser usada para garantir que uma variável seja vista entre threads
  - Mas o custo da comunicação é equivalente a um travamento e destravamento (bloco synchronized)
- **Final** é garantido desde que um objeto seja construído corretamente
  - Não deve haver vazamentos dentro de construtores
- Sem sincronização, o modelo de memória permite que o processador ou compilador reordene as instruções agressivamente
  - O modelo resultante **não tem consistência seqüencial** mas permite otimizações agressivas por parte do compilador e processador

Algoritmo de Dekker

- Clássico algorítmo de programação concorrente para exclusão mútua
  - Como o modelo de memória não garante CS, o algoritmo **não funciona em Java**, a menos que haja sincronização

```
x = y = 0;
void m1() {
    x = 1;
    if(y == 0) { ... seção crítica ... }
}
void m2() {
    y = 1;
    if (x == 0) {... seção crítica ...}
}
```

gravação

leitura

gravação

leitura

Por que não funciona? (1)

- Considere que um thread **t1** executa o método **m1()** e um thread **t2** executa o método **m2()** concorrentemente
- Quais são os valores possíveis para as **leituras** de **x** e **y**, ou seja, quais os valores que **i** e **j** podem ter?

```
x = y = 0; // valores iniciais de x  e y
void m1() {
    x = 1;
    j = y;
}
void m2() {
    y = 1;
    i = x;
}
```

gravação

leitura

gravação

leitura

O código foi reescrito para destacar as leituras e gravações

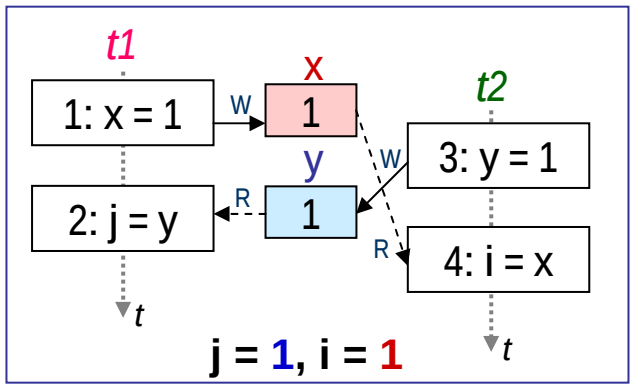
Cenários possíveis em CS

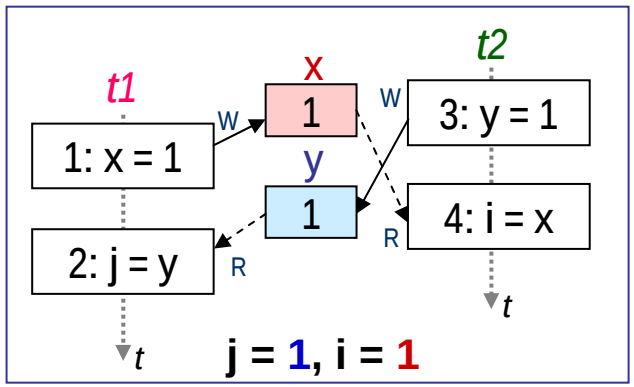
Inicialmente
x = y = 0

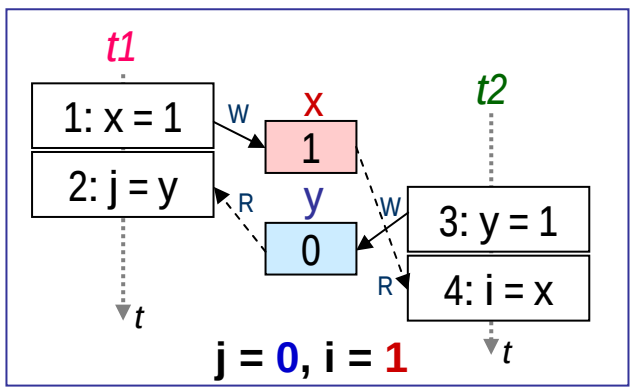
Todos os cenários consideram CS

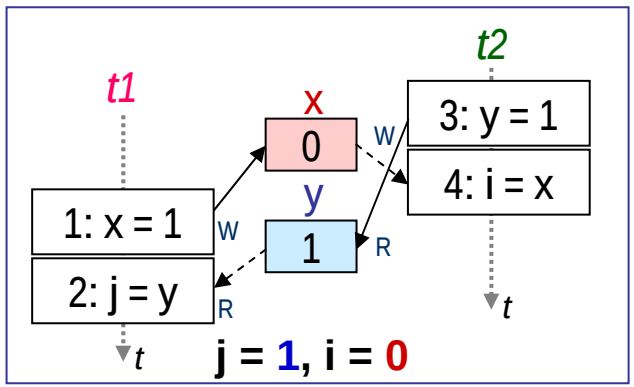
```
Thread t1:
 1: x = 1;
 2: j = y;

Thread t2:
 3: y = 1;
 4: i = x;
```

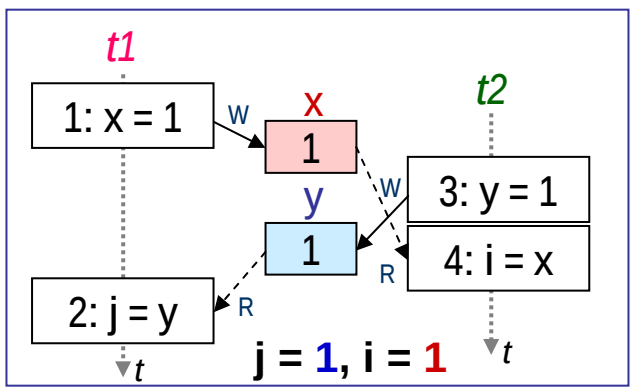
Mas a JMM não garante CS!
Que valores são possíveis sem CS?

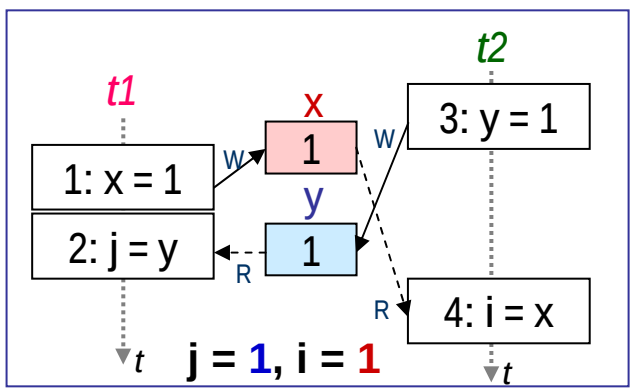




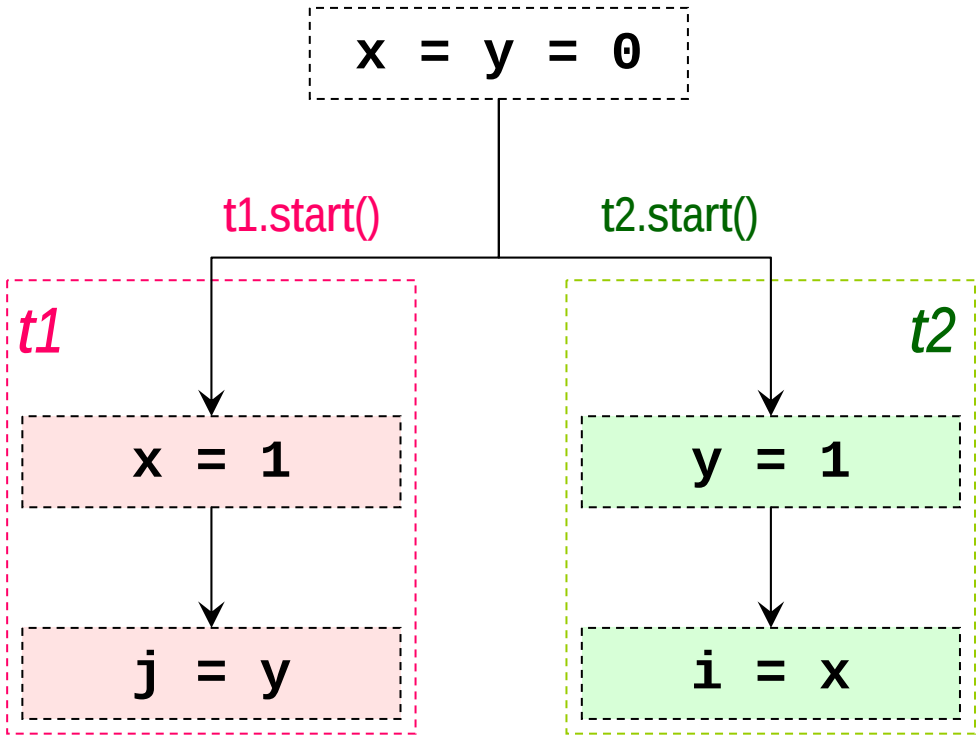








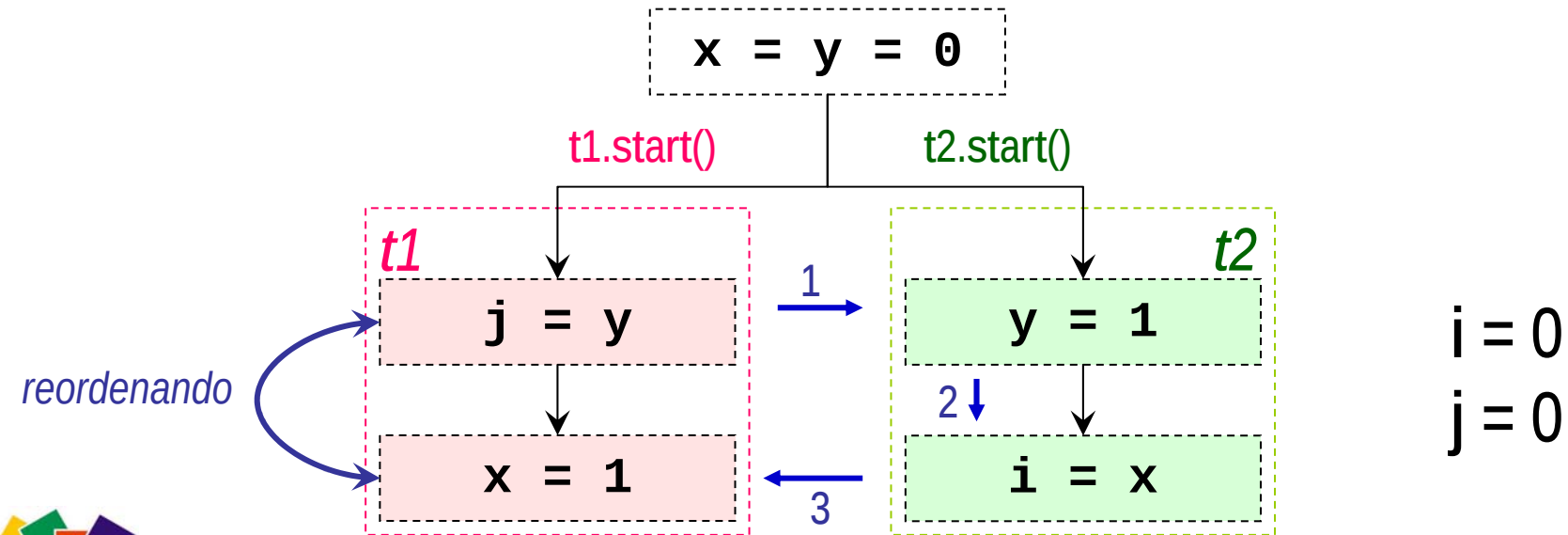
Lógica estranha em Java



- Poderia o resultado ser: **i = 0** e **j = 0**?

O JMM garante que poderia **sim**!

- Viola a consistência seqüencial
  - Resultado **i = 0** e **j = 0** não seria possível em sistemas CS
- Mas a JMM não garante CS!
  - Compilador *pode* **reordenar** instruções independentes
  - Processador *pode* usar **buffers de gravação** para leitura paralela se a leitura não depender da gravação



Como isto pode acontecer?

1. O **compilador** pode reordenar instruções ou manter os valores nos registradores
   - Diversas técnicas que visam melhorar eficiência de algoritmos
   - Não afeta funcionamento sequencial
2. O **processador** pode reordená-los
3. Em sistemas multiprocessados, os valores não são sincronizados na **memória global**

- O JMM é projetado para permitir otimizações agressivas
  - Inclusive otimizações que ainda não foram implementadas
  - Ótimo para performance;
    Ruim para raciocinar sobre código não sincronizado

Para que preciso do JMM?

- Entender os detalhes do Java Memory Model é necessário se você precisar
  - Construir um compilador, uma máquina virtual
  - Simular a máquina virtual
  - Escrever programas em Java que dispensam o uso de **synchronized** (não recomendado)
- Entender os princípios básicos do Java Memory Model é importante para
  - Usar corretamente **synchronized**, **final** e **volatile** e escrever aplicações que funcionem em qualquer plataforma

    O comportamento do Java Memory Model só é complexo em programas incorretamente sincronizados

Como usar corretamente o JMM?

- Usar corretamente o JMM não é difícil. É preciso
  - Entender o significado da palavra **synchronized**
  - Lidar corretamente com dados que são mutáveis e compartilhados
  - Entender o processo de construção de objetos para garantir a correta semântica de **final** e construir objetos realmente *imutáveis*
  - Entender o custo de performance das operações de sincronização e **volatile**
- Escrever código *multithreaded* não é simples
  - Use os utilitários de concorrência do Java 5.0 sempre que possível e evite a programação em baixo nível

Parte II

Como escrever programas corretamente sincronizados

O que significa synchronized?

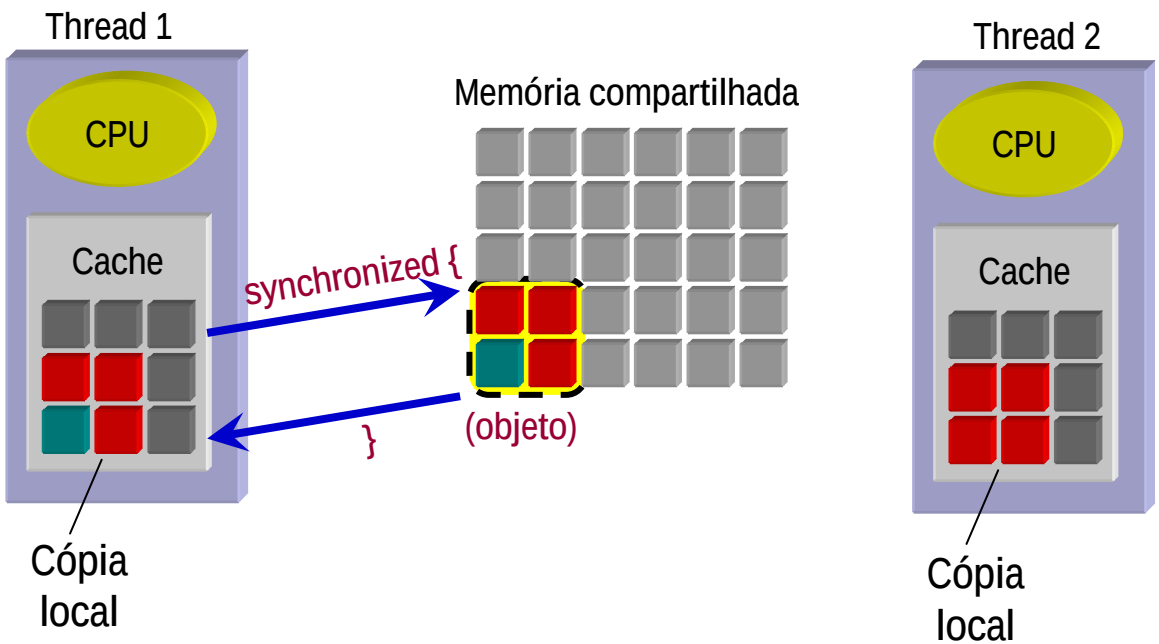
- **Synchronized** não é só uma trava
- Para entender corretamente synchronized é preciso entender o modelo de memória
  - No modelo de memória do Java, cada CPU (real ou virtual) tem **uma cópia** dos dados compartilhados em cache
  - Não há garantia que a cópia local esteja em dia com os dados compartilhados a não ser que acessos estejam em um bloco synchronized, ou sejam via variáveis voláteis
- Portanto
  - É possível acessar um objeto compartilhado fora de um bloco synchronized, mesmo que outro método tenha sua trava, mas **não há garantia** que o estado observado seja correto nem que quaisquer mudanças serão preservadas

Diagrama: modelo de memória

Funcionamento de **synchronized**

synchronized (objeto)

1. **{** Obtém trava
2. Atualiza cache local com dados da memória compartilhada
3. Manipula dados **localmente** (interior do bloco)
4. **}** Persiste dados locais na memória compartilhada
5. Libera trava

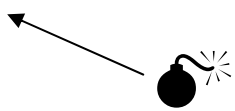


Fonte: [9]

Processos pseudo-atômicos

- A linguagem garante que ler ou escrever em uma variável de tipo primitivo(exceto long ou double) é um processo atômico.
  - Portanto, o valor retornado ao ler uma variável **é o valor exato** que foi gravado por alguma thread, mesmo que outras threads modifiquem a variável ao mesmo tempo sem sincronização.
- Cuidado com a **ilusão de atomicidade**

```
private static int nextSerialNumber = 0;
public static int generateSerialNumber() {
    return nextSerialNumber++;
}
```



- O método acima não é confiável sem sincronização
  - Por que? Como consertar?

Soluções

```
/* Solucao 1: synchronized */
private static int nextSerialNumber = 0;
public static synchronized int generateSerialNumber() {
    return nextSerialNumber++;
}

/* Solucao 2: objetos atômicos */
import java.util.concurrent.atomic.AtomicInteger;
private static AtomicInteger nextSerialNumber =
    new AtomicInteger(0);
public static int generateSerialNumber() {
    return nextSerialNumber.getAndIncrement();
}

/* Solucao 3: concurrent locks */
import java.util.concurrent.lock.*;
private static int nextSerialNumber = 0;
private Lock lock = new ReentrantLock();
public static int generateSerialNumber() {
    lock.lock();
    try { return nextSerialNumber++; }
    finally { lock.unlock();}
}
```

Dados mutáveis e compartilhados

- Dados compartilhados nunca devem ser observados em um estado inconsistente
  - É importante que as mudanças ocorram de um estado consistente para outro
- Existem apenas **duas** maneiras de garantir que mudanças em dados compartilhados sejam vistos por todos os threads que os utilizam
  - Realizar as alterações dentro de um bloco synchronized
  - Se os dados forem constituídos de apenas uma variável atômica, declarar a variável como volatile*

* Garantido apenas para JVM 5.x em diante

Liberação e aquisição

- Todos os acessos antes de uma liberação são ordenadas e visíveis a quaisquer novos acessos após uma aquisição correspondente
  - O thread vê os efeitos de todos os outros acessos sincronizados
- O destravamento de um monitor/trava é uma liberação, que é adquirida por qualquer trava seguinte daquele monitor/trava

Falha de comunicação

- Esse problema é demonstrado no padrão comum usado para interromper um thread, usando uma variável booleana
  - Como a gravação e leitura é atômica, pode-se cair na tentação de dispensar a sincronização

```
public class StoppableThread extends Thread {
    private boolean stopRequested = false;
    public void run() {
        boolean done = false;
        while (!stopRequested && !done) {
            // ... do it
        }
    }
    public void requestStop() {
        stopRequested = true;
    }
}
```

- Por não haver sincronização, não há garantia de quando o *thread* que se quer parar verá a mudança que foi feita pelo outro *thread*!
  - Esse problema poderá nunca acontecer em monoprocessadores

Soluções

- Uma solução é simplesmente sincronizar todos os acessos ao campo usado na comunicação
  - A sincronização neste caso está sendo usada apenas para seus efeitos de comunicação (e não para excusão mútua)

```
public class StoppableThread extends Thread {
    private boolean stopRequested = false;
    public void run() {
        boolean done = false;
        while (!stopRequested() && !done) {
            // ... do it
        }
    }
    public synchronized void requestStop() {
        stopRequested = true;
    }
    private synchronized boolean stopRequested() {
        return stopRequested;
    }
}
```

Campos voláteis

- Se um campo puder ser simultaneamente acessado por múltiplos threads, e pelo menos um desses acessos for uma operação de gravação, faça o campo **volátil**
- O que faz volatile?
  - Leituras e gravações vão diretamente para a memória: não é cacheado nos registros
  - Longs e doubles voláteis **são atomicos** (longs e doubles normais não são: apenas tipos primitivos menores o são)
  - Reordenamento de acessos voláteis pelo compilador é limitado

Garantia de visibilidade

- A solução ideal é declarar a variável como volatile
  - O modificador volatile equivale a uma aquisição e liberação de trava e tem a finalidade de resolver o problema da comunicação entre threads

```
public class StoppableThread extends Thread {
    private volatile boolean stopRequested = false;
    public void run() {
        boolean done = false;
        while (!stopRequested && !done) {
            // ... do it
        }
    }
    public void requestStop() {
        stopRequested = true;
    }
}
```

Solução recomendada
Java 5.0 em diante!

Garantia de ordenação

- Se um thread lê **data**, há uma liberação/aquisição em **ready** que garante visibilidade e ordenação

```
class Future {
  private volatile boolean ready;
  private Object data;
  public Object get() {
    if (!ready)
      return null;
    return data;
  }
  public synchronized void setOnce(Object o) {
    if (ready) throw … ;
      data = o;
    ready = true;
  }
}
```

Gravação vai sempre acontecer antes devido à ordenação!

Outras ações que causam liberação/aquisição

- Outras ações também formam pares libera-adquire
  - **Iniciar** um *thread* é uma liberação adquirida pelo método *run()* do thread
  - **Finalização** de um *thread* é uma liberação adquirida por qualquer thread que junta-se (joins) ao *thread* terminado

O famigerado multithreaded Singleton anti-pattern (JMM FAQ, EJ 48)

- Esse famoso padrão é um truque para suportar inicialização lazy evitando o overhead da sincronização
  - Parece uma solução inteligente (evita sincronização no acesso)
  - Mas não funciona! Inicialização de resource (null) e instanciamento podem ser reordenados no cache

```
class SomeClass {
  private static Resource resource = null;
  public static Resource getResource() {
    if (resource == null) {
      synchronized (Resource.class) {
        if (resource == null)
          resource = new Resource();
      }
    }
    return resource;
  }
}
```

* Também conhecido como o double-check idiom

Alternativas

- Não usar lazy instantiation
  - Melhor alternativa (deixar otimizações para depois)

```
private static final Resource resource = new Resource();
public static Resource getResource() {
   return resource;
}
```

- Instanciamento lazy corretamente sincronizado (somente Java 5.0)
  - Há custo de sincronizacão na variável volátil

```
private volatile static Resource resource = null;
public static Resource getResource() {
  if (resource == null)
    resource = new Resource();
  return resource;
}
```

(É tão lento quanto declarar o método getResource synchronized)

- *Initialize-on-demand holder class idiom*

```
private static class ResourceHolder {
  static final Resource resource = new Resource();
}
public static Resource getResource() {
  return ResourceHolder.resource;
}
```

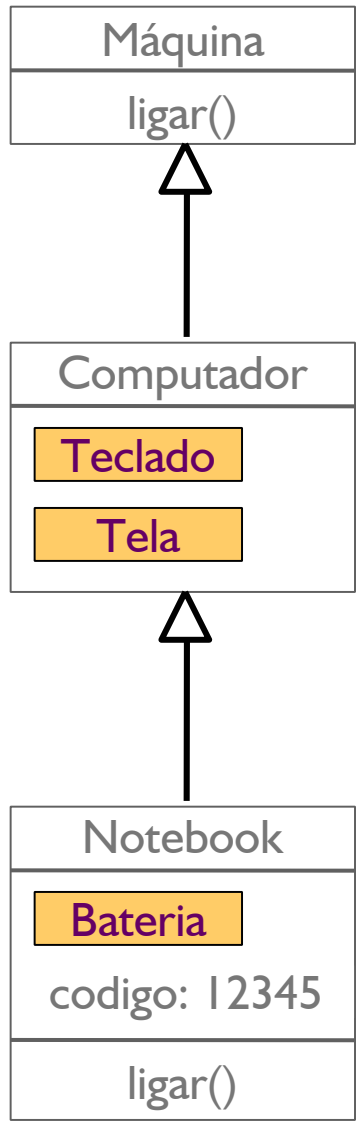

Esta técnica explora a garantia de que uma classe não é inicializada antes que seja usada.

Inicialização de instâncias

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

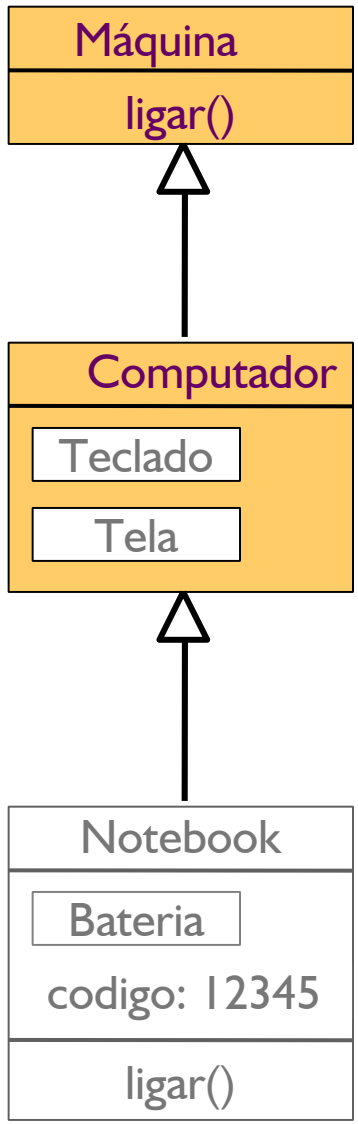
Exemplo (1)

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

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

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



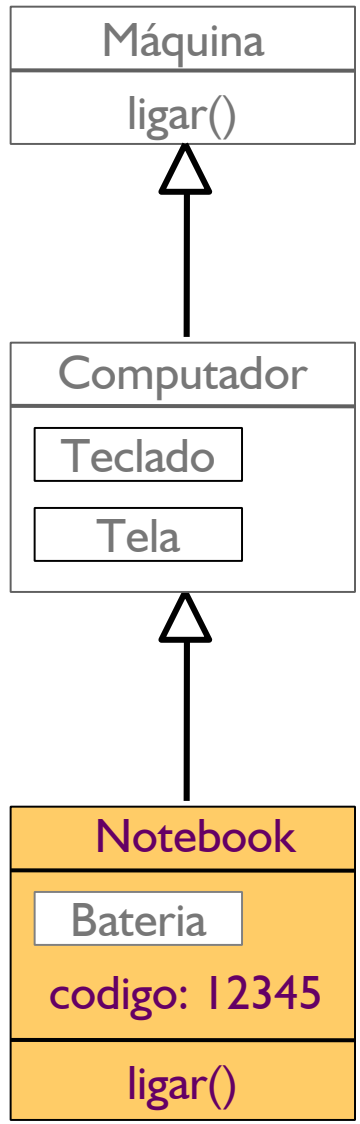
Exemplo (2)

```
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();
  }
}
```



```
new Notebook()
```

Isto foi executado, e...

```
Maquina()

Notebook.ligar(); codigo = 0

Tela()

Teclado()

Computador()

Bateria()

Notebook(); codigo = 12345
```

1. Construtor de Maquina chamado

2. Método ligar() de Notebook (e não de Maquina) chamado!

3. PROBLEMA!!!!! Variável codigo, de Notebook ainda não foi inicializada quando ligar() foi chamado!

4. Variáveis tela e teclado, membros de Computador são inicializadas

5. Construtor de Computador chamado

6. Variável bateria, membro de Notebook é inicializada

7. Construtor de Notebook chamado. Variável codigo finalmente inicializada

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()

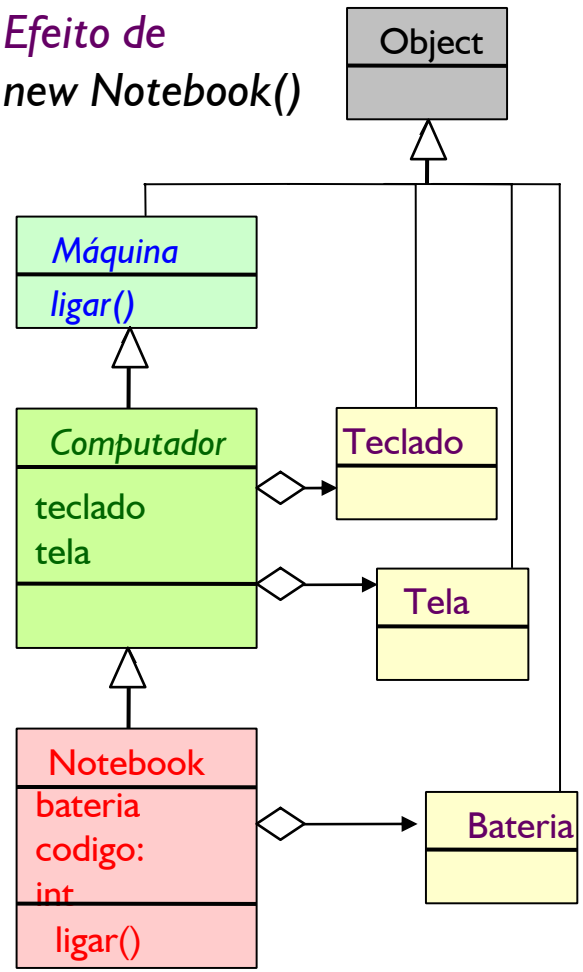
N5. Construtor de Bateria chamado

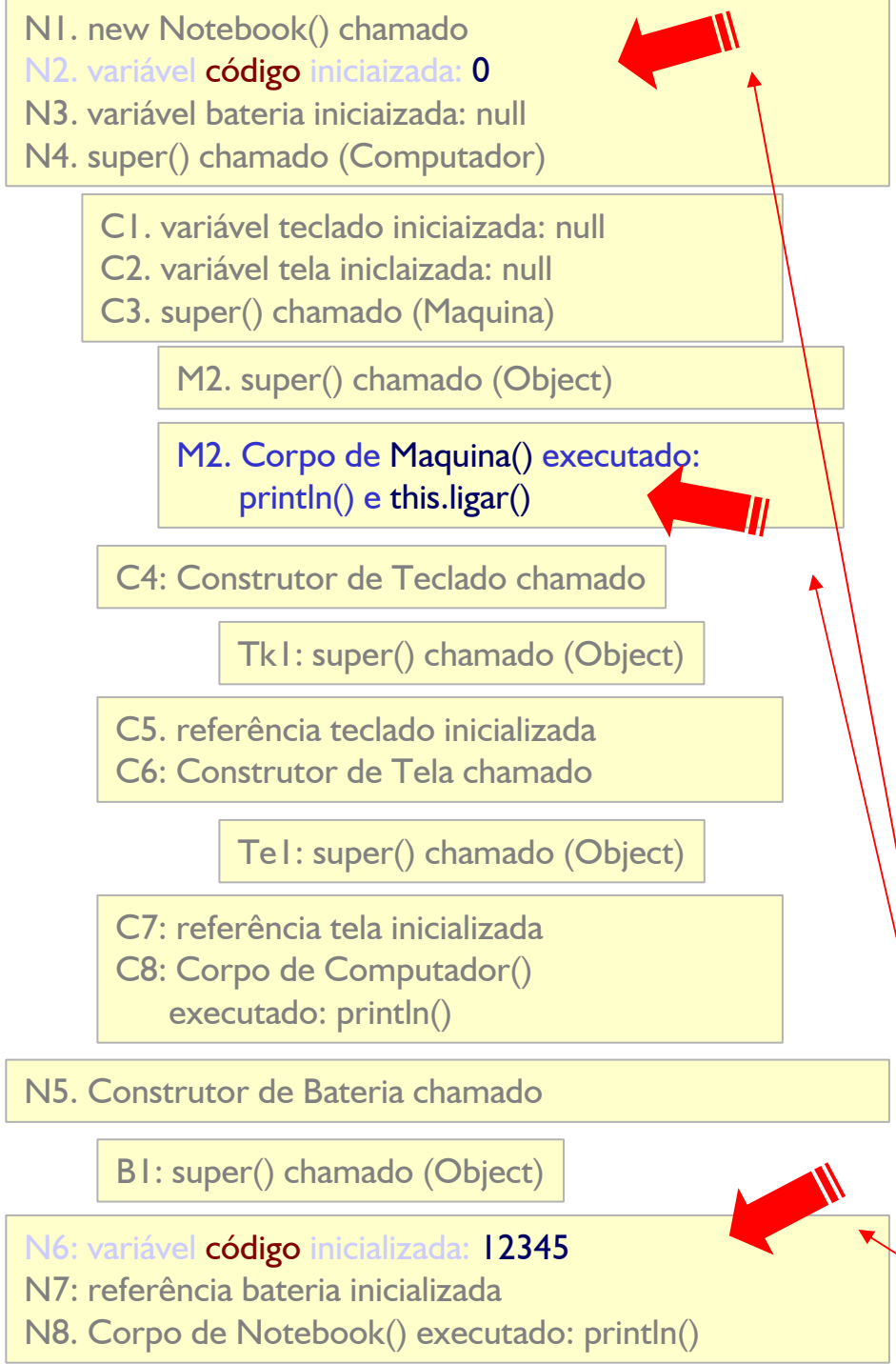
B1: super() chamado (Object)

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()





Quebra de encapsulamento!

- 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!

Como evitar o problema?

- Evite chamar métodos locais dentro de construtores
  - Construtor (qualquer um da hierarquia) sempre usa versão sobreposta do método
- Resultados inesperados se alguém estender a sua classe com uma nova implementação do método que
  - Dependa de variáveis da classe estendida
  - Chame métodos em objetos que ainda serão criados (provoca NullPointerException)
  - Dependa de outros métodos sobrepostos
  - Deixe vazar variáveis para campos estáticos (comportamento de final só é garantido na conclusão do Construtor!)
- Use apenas métodos finais em construtores
  - Métodos declarados com modificador final não podem ser sobrepostos em subclasses

Objetos imutáveis

- Processo de criação de um objeto
  - Objeto é instanciado; atributos são inicializados a valores default (ex: 0)
  - Objetos e construtores das superclasses são inicializados recursivamente
  - Atributos são inicializados a valores explícitos
  - Corpo do construtor é executado (possível nova atribuição)
- Atributos assumem até 3 valores diferentes durante criação do objeto
  - Se **i** não for final, uma chamada **new Integer(5)** pode fazer com que o valor de **i** apareça como **0** para alguns threads e **5** para outros

```
public class Integer {
    private final int i;
    public Integer(int j) {
        i = j;
    }
    public int getValue() {
        return i;
    }
}
```

Funciona no Java 5.0 devido ao novo modelo de memória (JMM)

Não deixe vazar referências na construção do objeto!

- Vazamento: **this**
  - Construtor deixou vazar um acesso à referência do próprio objeto: **final falha!**
  - JMM garante semântica de **final** para objetos **construídos corretamente** (ou seja, sem deixar vazar referências no construtor)

```
public class Integer {
    private final int i;
    public static Integer ultimo;
    public Integer(int j) {
        i = j;
        ultimo = this;
    }
    public int getValue() {
        return i;
    }
}
```

Threads que lêem esta referência não têm garantia de ver um valor fixo

Bug semelhante: criar novo Thread dentro do construtor!

Conclusões

- O JMM foi reescrito para garantir uma semântica simples e determinística para o funcionamento de programas (corretos e incorretos) aos programadores
  - Programas incorretamente sincronizados podem causar resultados surpreendentes, porém válidos pelo JMM
- O JMM garante a semântica esperada de **volatile**, **final** e **synchronized** para programas corretos
- Construir programas corretos requer
  - Conhecimento do funcionamento de *volatile* e *synchronized* para construir programas corretamente sincronizados
  - Construir objetos corretamente (garantindo encapsulamento total do construtor durante a criação de objetos)

Fontes de referência

[1] James Gosling, Bill Joy, Guy Steele, Java Language Specification third edition, Addison Wesley, 2005 – Capítulo 17

[2] Jeremy Manson, William Pugh. JSR-133 –The Java Memory Model and Thread Specification.

[3] William Pugh. Fixing the Java Memory Model.

[4] Sarita Adve et al. Shared Memory Consistency Models: A Tutorial

[5] Documentação do J2SDK 5.0

[6] Joshua Bloch, Effective Java Programming Guide, Addison-Wesley, 2001

[7] Jeremy Manson and Brian Goetz, JSR 133 (Java Memory Model) FAQ, 2004

- www.cs.umd.edu/users/pugh/java/memoryModel/jsr-133-faq.html

[8] Doug Lea, Synchronization and the Java Memory Model, 1999.

[9] Doug Lea, Concurrent Programming in Java (1st. ed), 1996