Tópicos selecionados de programação em Java

Threads em Java

Padrões, anti-padrões e boas práticas

Helder da Rocha

Julho 2005

1. Entenda o significado de synchronized (PJ)

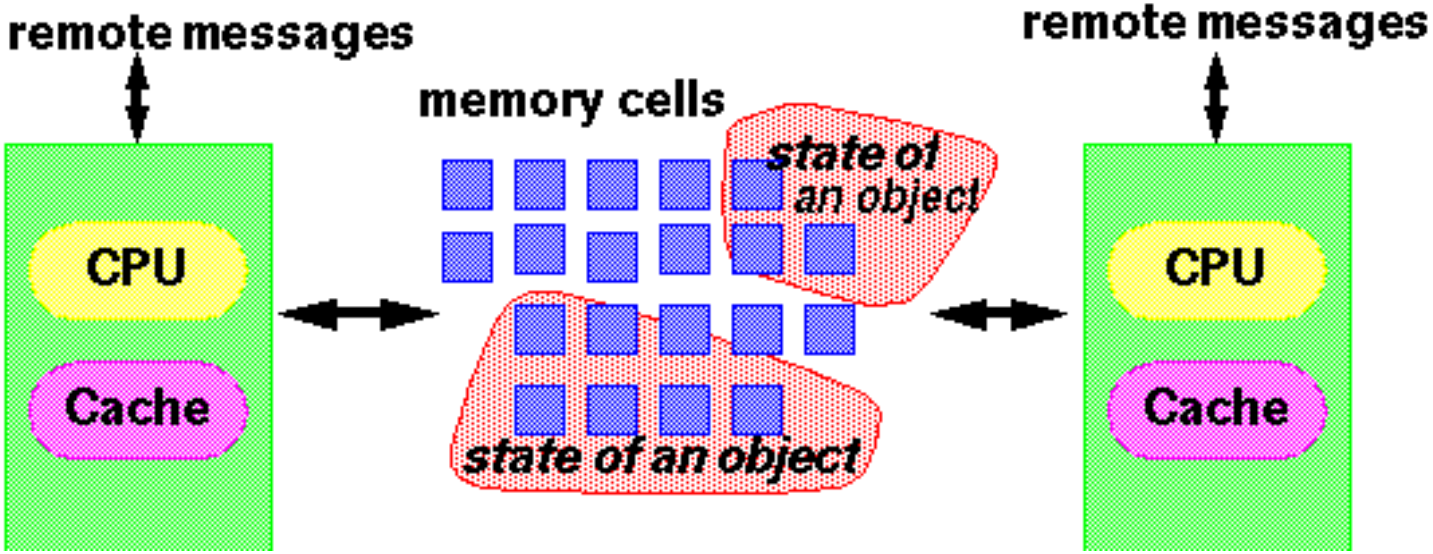
- Para entender corretamente synchronized é preciso entender o modelo de memória
  - Java possui um modelo de memória que descreve o comportamento de aplicações concorrentes e paralelas em relação à memória compartilhada
  - Nesse modelo, cada CPU (real ou virtual) tem uma cópia dos dados compartilhados em cache
  - Não há garantia que os dados em cache estejam em dia com os dados compartilhados a não ser que estejam em um bloco synchronized, ou sejam 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: [LEA1]

2. Sincronize o acesso a dados mutáveis e compartilhados (EJ 48)

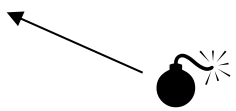
- Dados compartilhados nunca devem ser observados em um estado inconsistente
  - É importante que as mudanças ocorram de um estado consistente para outro
- Existem apenas duas maneiras de garantir que mudanças em dados compartilhados sejam vistos por todos os threads que os utilizam
  - Realizar as alterações 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**

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();}
}
```

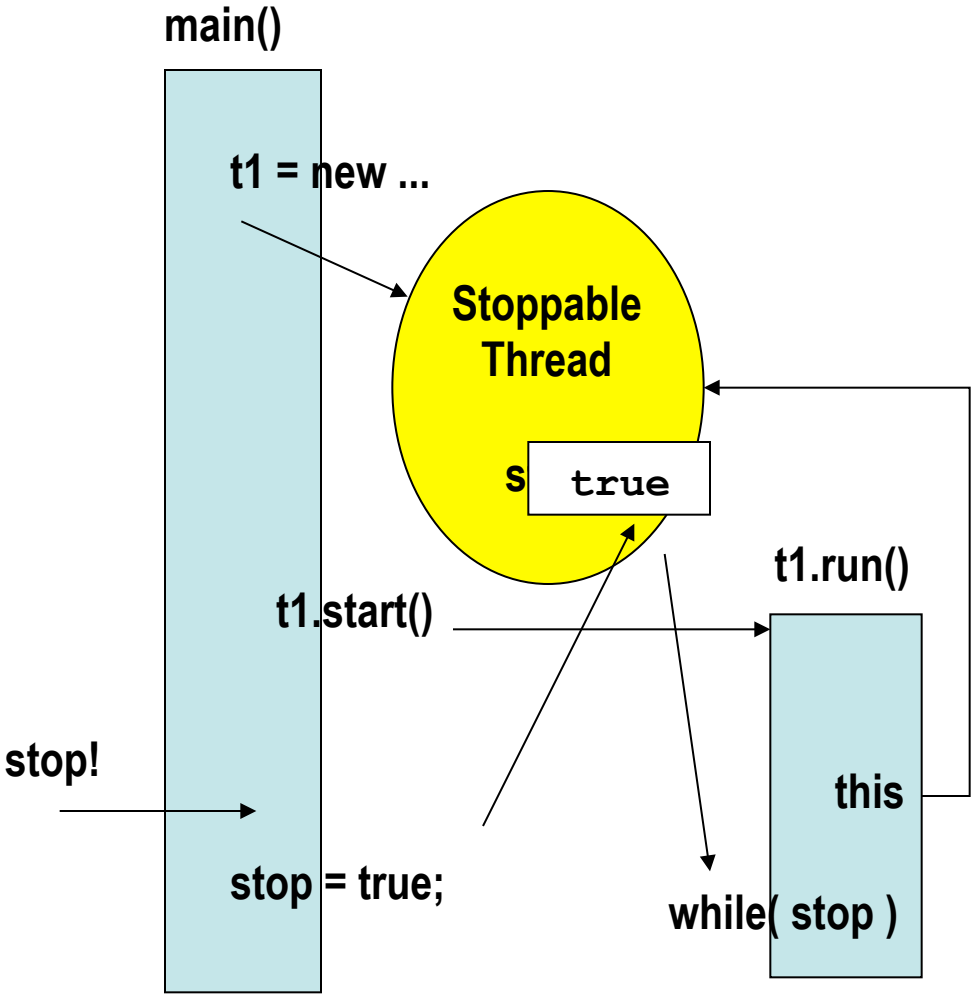
Falha de comunicação

- Esse problema é demonstrado no padrão comum usado para interromper um thread, usando uma variável booleana
  - Como a gravação e leitura é atomica, 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 não acontecer em sistemas monoprocessados

Threads vs. objetos



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;
    }
}
```

Soluções (2)

- Uma outra solução 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!

13. Não use o 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**

O double-check idiom não funciona

- Se um thread lê a referência sem sincronização e depois chama um método no objeto referenciado, o método poderá observar o objeto em um estado parcialmente inicializado (e falhar)
- Pode não parecer óbvio o vazamento
  - O objeto é construído completamente antes da referência ser publicada em um campo onde poderá ser lido por outros threads
  - Porém, sem sincronização, a leitura da referência de objeto publicada não garante que o thread que lê verá seus valores mais recentes
- O problema seria resolvido se
  - A variável compartilhada fosse um inteiro, um byte ou outro valor primitivo atômico
  - A variável fosse volatile, garantindo a correta ordenação entre a inicialização e construção do objeto, mas a performance seria pior que antes (volatile funciona como um bloco synchronized)

Alternativas

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

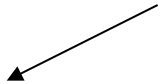
```
private static final Resource resource = new Resource();
public static Resource getResource() {
   return resource;
}
```

- Instanciamento lazy corretamente sincronizado
  - Há custo de sincronizacão em cada chamada

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

- *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.

3. Diferencie a sincronização em métodos estáticos e de instância (PJ)

- Quando synchronized é chamado em um método de instância, o bloco ao qual se aplica a trava é restrito ao objeto
  - Outros objetos da mesma classe podem ser livremente usados
- Quando synchronized é chamado em um método de classe (estático) todo o estado da classe é travado
  - Impede que qualquer outro thread tenha acesso a dados da classe inteira

4. Encapsule seus dados

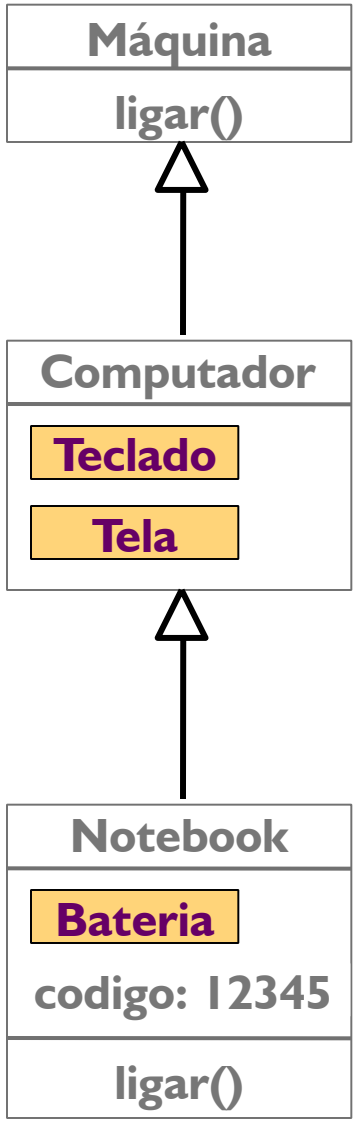
- Não adianta usar synchronized se seus dados estiverem incorretamente encapsulados
  - Declare o estado crítico de um objeto com modificador private
  - Não deixe referências para dados de instância vazarem pelo construtor (não atribua this a variáveis estáticas durante a construção do objeto)
  - Não chame métodos que possam ser sobrepostos dentro de construtores (construa objetos corretamente!)
- Exemplo de quebra de encapsulamento na inicialização de instâncias (a seguir)

Inicialização de instâncias

- O que acontece quando um objeto é criado usando new NomeDaClasse() ?
  - 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

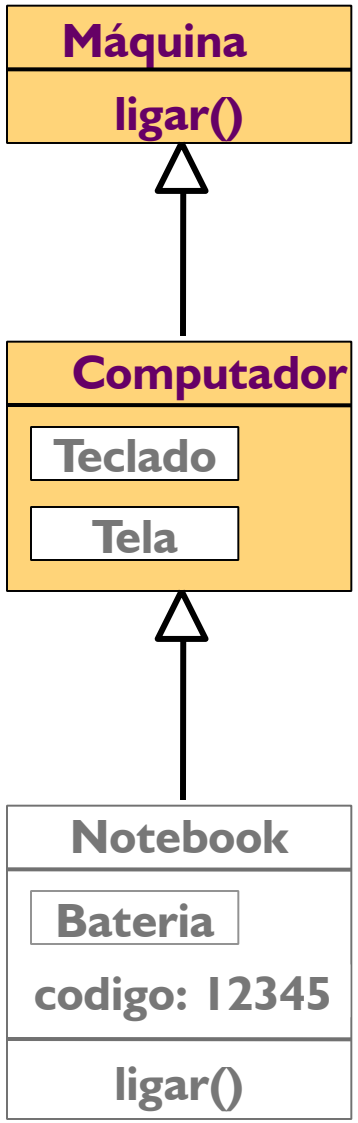
Exemplo (1)

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

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

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



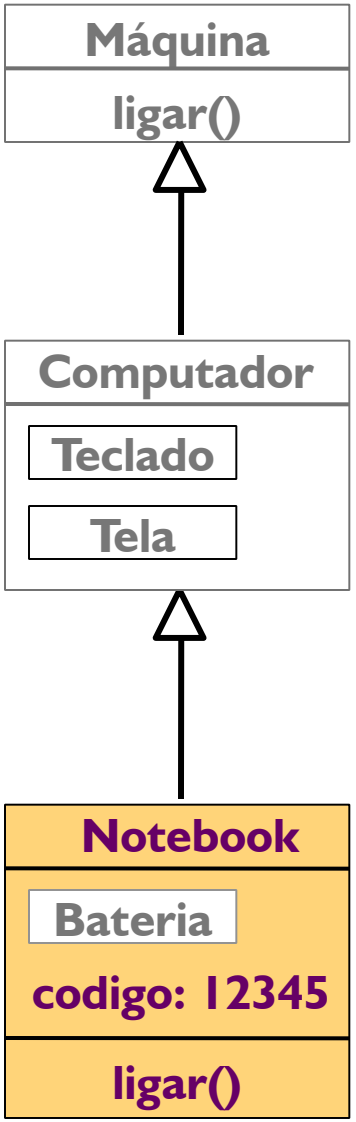
Exemplo (2)

```
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()

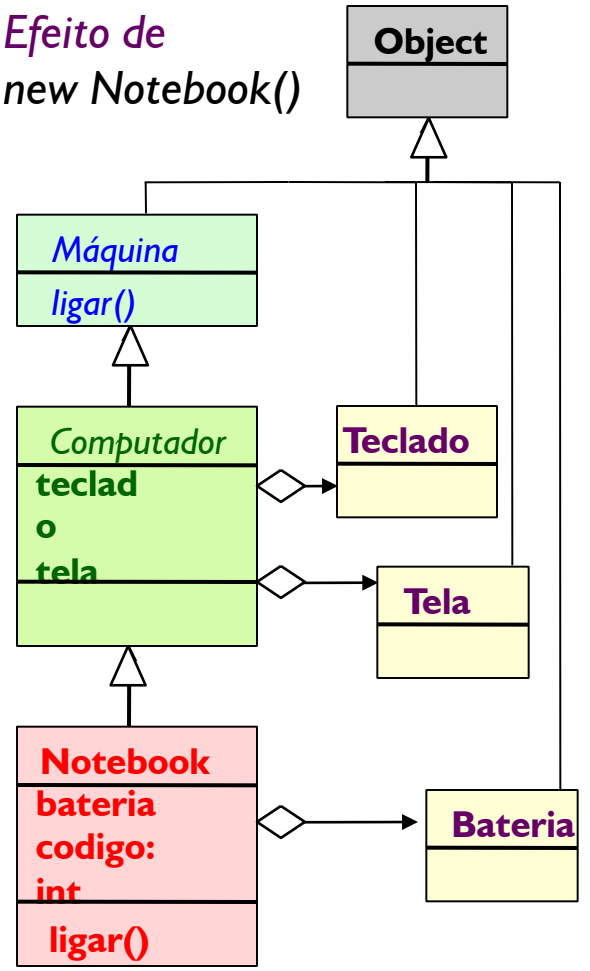
N5. Construtor de Bateria chamado

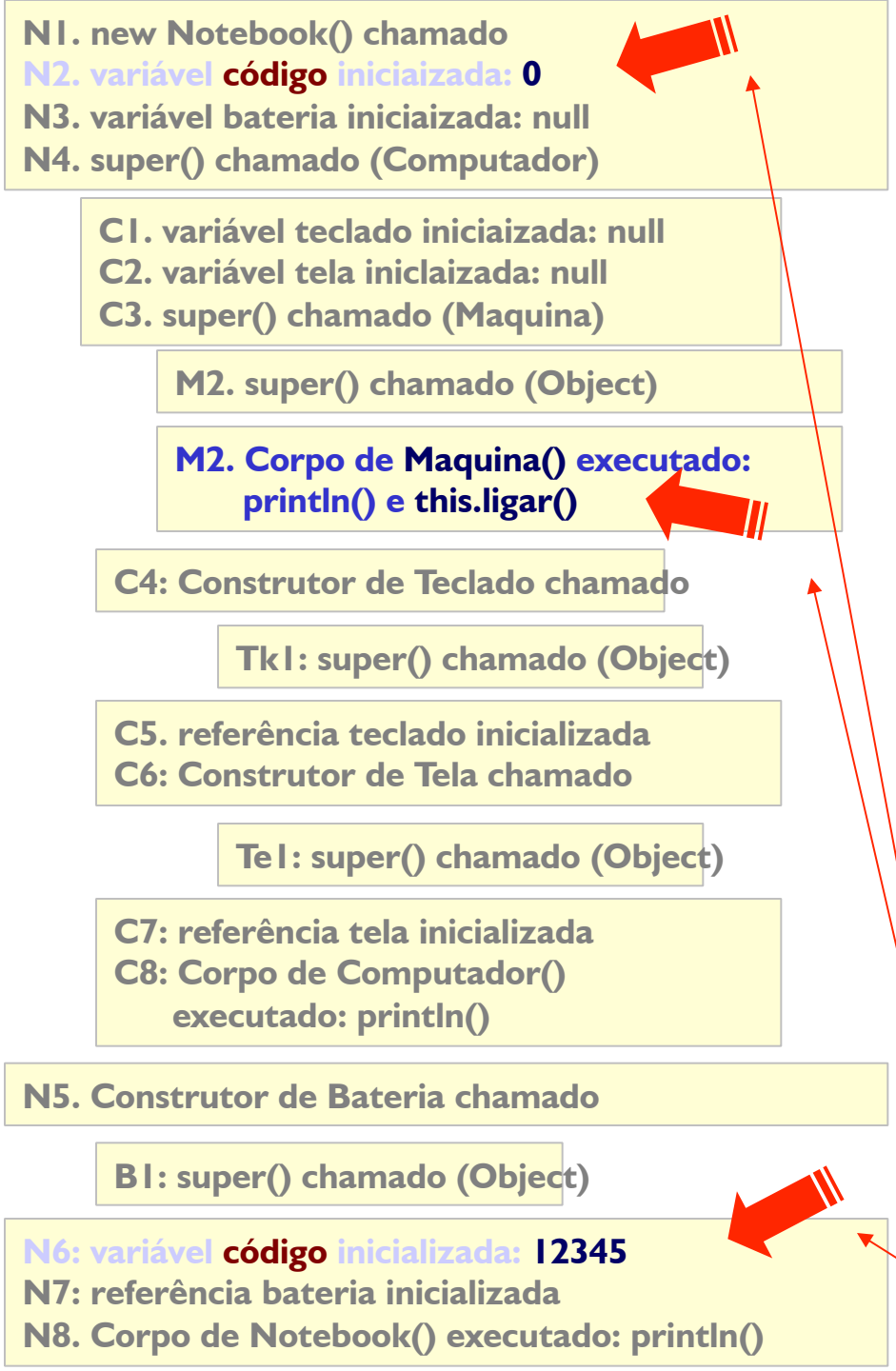
B1: super() chamado (Object)

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 (em aplicações concorrentes, os valores são não determinísticos)
- Use apenas métodos finais em construtores
  - Métodos declarados com modificador final não podem ser sobrepostos em subclasses

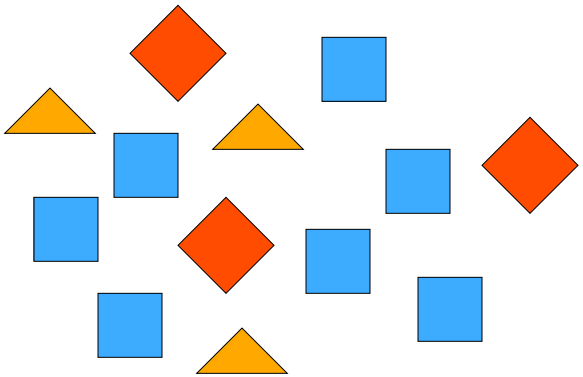
5. Utilize sempre que possível objetos imutáveis (EJ 13)

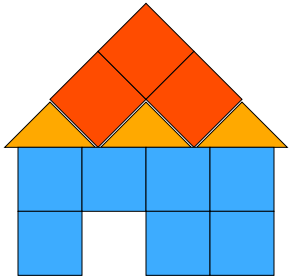
- São **thread safe**: não requerem sincronização
  - Não podem ser corrompidos por múltiplos threads concorrentes
  - Podem ser compartilhados livremente
  - Seu estado interno pode ser compartilhado
- São **simples**
  - Um objeto imutável pode estar em exatamente um estado: o estado no qual foi criado
  - São ideais para servir de blocos de montagem para objetos maiores, como elementos de conjuntos, chaves de mapas
- **Desvantagem**: cada valor distinto requer um objeto próprio
  - Valores iguais são reutilizados (pools, caches)
  - Construir novos objetos requer criar vários objetos durante o processo para depois descartá-los (ex: concatenação de String). Solução: prover uma classe companheira mutável (ex: StringBuffer)

Recomendações de design

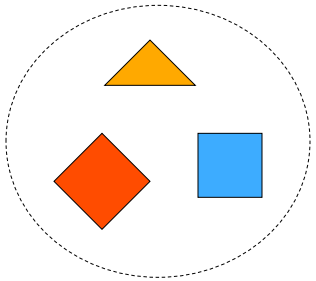
- Classes devem ser imutáveis a menos que haja uma boa razão para que não sejam
  - Pequenos objetos devem sempre ser imutáveis
  - Considere fazer objetos maiores imutáveis também
  - Forneça uma classe companheira mutável somente depois de confirmar que é necessária para alcançar performance satisfatória
- Se uma classe não puder ser imutável, você ainda deve limitar sua mutabilidade o quanto for possível
  - Construtores devem criar objetos completamente inicializados e não passar instâncias parcialmente construídas a outros métodos

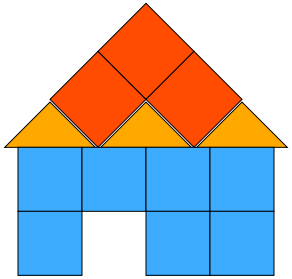
Aplicações





Pool de objetos imutáveis compartilhados

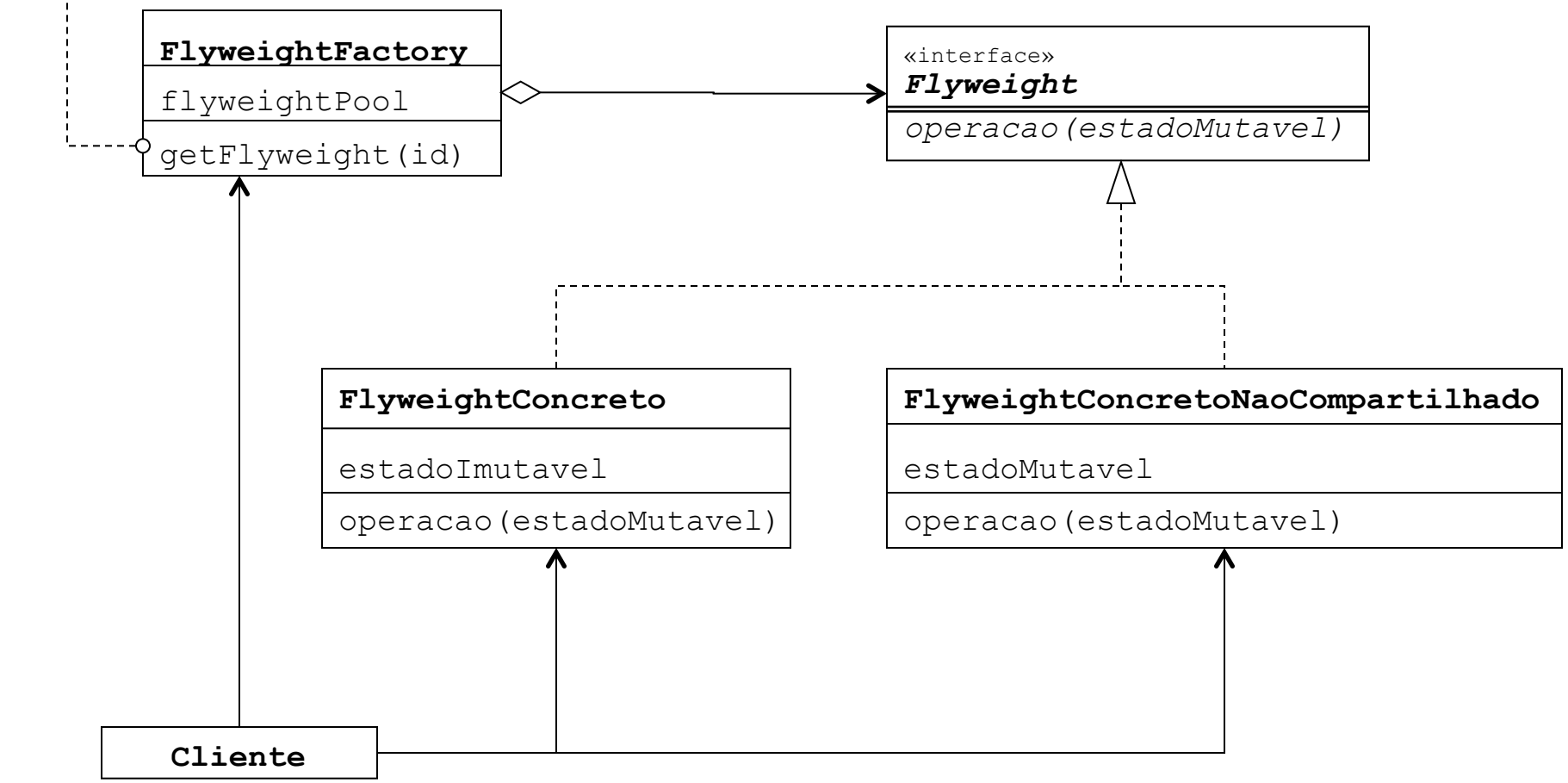




```
if(flyweightPool.containsKey(id)) {
    return (Flyweight)flyweightMap.get(id);
} else {
    Flyweight fly = new FlyweightConcreto( genKey() );
    flyweightPool.put(fly.getKey(), fly);
    return fly;
}
```

Flyweight

GoF Design Pattern



Como garantir objetos imutáveis

- Ponto de partida: objetos “puros”
  - Métodos puros não têm efeitos colaterais
  - Construtores puros só tem efeitos colaterais em estados recém-alocados
  - Classes puras têm apenas métodos puros, construtores puros e subclasses puras
- Não garante imutabilidade!

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

É puro! Não tem efeitos colaterais. Mas não é imutável!

Como garantir objetos imutáveis

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

Acesso direto ao estado do objeto foi fechado!

- Classe aparentemente bem encapsulada, mas ainda não é imutável!
  - Por que?

Final é necessário!

- 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

```
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

Tipos de imutabilidade

- Imutabilidade rasa
- Imutabilidade profunda
- Imutabilidade baseada em estado
  - Estado do objeto inclui apenas seus atributos imutáveis (referências para objetos externas não são parte de seu estado)

Imutabilidade rasa

- Uma classe pura tem imutabilidade rasa se
  1. Todos os campos de dados de instância são finais, e
  2. Construtores não deixam vazar a referência **this**

```
public class BankTransfer {
    private final char[] src, dest;
    private final Integer amount;
...
    char[] getDest() {
        return dest;
    }
    Integer getAmount() {
        return amount;
    }
}
```

Objetos não são imutáveis, apenas as referências!

Imutabilidade profunda

- Uma classe pura é profundamente imutável se
  1. tiver imutabilidade rasa, e
  2. todos os campos de dados que forem referências
     i. possuem tipos imutáveis, ou
     ii. não podem ser atribuídas a outras referências

```
public class BankTransfer {
    private final BankAccount src, dest;
    private final Integer amount;
...
    BankAccount getDest() {
        return dest;
    }
    BankAccount getSrc() {
        return amount;
    }
}
```

Requer que destino e fonte da transferência sejam imutáveis!

Imutabilidade profunda é excessiva neste caso

Imutabilidade baseada em estado

- Solução: meio-termo
- Uma classe pura tem imutabilidade baseada em estado se
  1. todos os campos de instância são finais, e
  2. construtores não deixam vazar a referência **this**, e
  3. todos os atributos de instância que têm referências
     i. possuem tipos imutáveis, ou
     ii. não podem ser atribuídas a outras referências (finais), ou
     iii. são excluídas do “estado” do objeto

6. Evite sincronização excessiva

(EJ 49)

- Sincronização excessiva pode causar redução de performance, deadlock ou até mesmo comportamento não determinístico
- O principal risco é o deadlock. Para evitá-lo
  - Nunca ceda controle ao cliente dentro de um método ou bloco sincronizado (não chame métodos public ou protected que foram criados para serem sobrepostos – chame apenas métodos final e private)
  - Chame métodos externos fora da região sincronizada (open call). Além de prevenir deadlocks, *open calls* permitem aumento da concorrência
- Outro risco ocorre com o uso de travas recursivas
  - Método externo, que retém a trava, chamado quando as invariantes protegidas pela região sincronizada estão temporariamente inválidas
  - Risco ocorre quando método entra novamente na classe onde ele retém a trava
- Regra geral
  - Faça apenas o necessário dentro de regiões sincronizadas.
  - Obtenha a trava, analise os dados, altere o que precisar, libere a trava

Anti-pattern: potencial de deadlock

```
public abstract class WorkQueue {
    public final List queue = new LinkedList();
    private boolean stopped = false;
    protected WorkQueue() { new WorkerThread().start(); }
    public final void enqueue(Object workItem) {
        synchronized(queue) {
            queue.add(workItem);
            queue.notify();
        }
    }
    public final void stop() {
        synchronized(queue) {
            stopped = true;
            queue.notify();
        }
    }
    protected abstract void processItem(Object workItem)
                            throws InterruptedException;
    // Broken!
    private class WorkerThread extends Thread {
        public void run() {
            while(true) {
                synchronized(queue) {
                    try {
                        while(queue.isEmpty() && !stopped) queue.wait();
                    } catch (InterruptedException e) {
                        return;
                    }
                    if (stopped) return;
                    Object workItem = queue.remove(0);
                    try {
                        processItem(workItem); // lock held
                    } catch (InterruptedException e) {
                        return;
}}}}}}
```

Deadlock Queue

- Para usar a classe é preciso implementar o método processItem

```
class DisplayQueue extends WorkQueue {
    protected void processItem(Object
workItem)
        throws InterruptedException {
        System.out.println(workItem);
        Thread.sleep(1000);
}}
```

- Esta implementação causa deadlock

```
class DeadlockQueue extends WorkQueue {
    protected void processItem(Object workItem)
        throws InterruptedException {
        Thread child = new Thread() {
            public void run() { enqueue(workItem); }
        };
        child.start();
        child.join(); // deadlock!
}}
```

Solução

```
private class WorkerThread extends Thread { ...
    public void run() {
        while(true) {
            synchronized(queue) {
                try {
                    while(queue.isEmpty() && !stopped)
                       queue.wait();
                } catch (InterruptedException e) {
                    return;
                }
                if (stopped) return;
                   Object workItem = queue.remove(0);
            }
            try {
                processItem(workItem); // no lock held
            } catch (InterruptedException e) {
                return;
}}}}
```

- **Regra geral:** para evitar corrupção de dados e deadlock, nunca chame um método externo de uma região sincronizada

Thread-safe ou thread compatible?

- 1) Se você está escrevendo uma classe que será usada em circunstâncias que requerem sincronização e também em circunstâncias onde ela não é requerida
  - Ofereça versões thread-safe (com sincronização interna) e thread-compatible (não sincronizadas)
  - Uma forma de fazer isto é através de uma classe Wrapper (encapsuladora)
- 2) Para classes que não serão estendidas ou reimplementadas
  - Ofereça uma classe não sincronizada e uma subclasse contendo apenas métodos sincronizados que chamam suas implementações na superclasse
- 3) Se uma classe ou método estático depende de um campo estático mutável
  - Ele deve ser sincronizado internamente, mesmo que seja tipicamente usado por um único thread

Use a nova API

- Considere a utilização das implementações da interface Lock (pacote de utilitários de concorrência do Java 5.0)
  - O uso das APIs torna a ocorrência de vários dos problemas descritos mais rara
- A API oferece várias implementações eficientes de travas; dentre elas
  - Implementações de ReadWriteLock
  - ReentrantLock (efeito equivalente a synchronized)
- Coleções são implementadas em versões thread-compatible e thread-safe, por exemplo:
  - HashMap (thread-compatible)
  - Hashtable (thread-safe)
  - ConcurrentHashMap (thread-safe, da nova API)

7. Nunca chame wait() fora de um loop (EJ 50)

- O método Object.wait() é usado para fazer a thread esperar por uma determinada condição
  - Ele precisa ser chamado dentro de uma região sincronizada que trava o objeto em que é chamado.
  - Técnica padrão para usar o método wait:

```
synchronized (obj) {
    while ( [condição não atendida] )
        obj.wait();
        // Realiza ação apropriada à condição
}
```

- Sempre chame use a técnica padrão para chamar wait()
  - **Nunca** chame-o fora de um loop

Viveza e segurança

- Testar a condição antes de esperar e pular o wait() se a condição foi atendida é necessário para garantir a **viveza** (liveness) do thread.
  - Se a condição já foi atendida e um notify() já foi chamado antes do thread começar a esperar, ele pode nunca acordar do wait()
- Testar a condição depois de esperar e esperar de novo se a condição não foi atendida é necessário para garantir a **segurança**
  - Se o thread continua com a ação quando a condição não mais se aplica, pode destruír as invariantes protegidas pela trava.

Acordando o thread fora de hora

- Há várias razões que podem levar um thread a acordar quando a condição do loop não for atendida
  - Outro thread pode ter obtido a trava e mudado o estado protegido no intervalo em que um thread chamou notify() e o thread que esperava acordou
  - Outro thread pode ter chamado notify() acidentalmente (ou maliciosamente) quando a condição não fora atendida (qualquer wait() contido em um bloco synchronized de um objeto publicamente acessível é vulnerável a esse problema)
  - O thread notificador foi excessivamente generoso ao acordar todos os threads com notifyAll()
  - O thread que espera poderia acordar na ausência de uma notificação (*spurious wakeup*)

notify vs. notifyAll

- Use sempre o notifyAll()
  - Este é o conselho mais conservador considerando que todas as chamadas ao wait() estão dentro de loops,
  - Garante que as threads que precisam ser reativadas sempre serão reativadas.
  - Também serão reativadas outras threads, mas isso não influi no resultado do programa, já que a condição será testada antes de continuar qualquer execução.
- notifyAll() pode afetar a performance.
  - Em estruturas de dados onde algumas threads possuem um status especial e as outras precisam esperar (número de threads saindo da espera é quadrático).
- Para otimizar
  - Pode-se decidir usar o notify() ao invés do notifyAll() se todas as threads esperam a mesma condição e apenas uma thread de cada vez pode se beneficiar da condição tornar-se true.
  - Ainda assim pode ser recomendado utilizar o notifyAll(), pois protege o código de uma chamada indevida ao wait() por thread não relacionada

*

Use a nova API

- Dê preferência à interface java.util.concurrent.locks.Lock e Condition para implementar monitores
  - Condition.await() é similar a wait()
  - Condition.signal()/signalAll() é similar a notify()/notifyAll()
- Veja exemplos na apresentação sobre utilitários de concorrência

8. Nunca dependa do agendador de threads (EJ 51)

- Depender do agendamento de threads é criar aplicações não portáveis
- Quando múltiplas threads são executáveis, o agendador de threads determina qual irá rodar e por quanto tempo.
  - A política usada varia muito de acordo com a implementação da JVM,
  - Para que o programa seja portável, é importante que ele não dependa do agendador de threads para executar corretamente ou ter boa performance.
- Para não depender demais de agendamento, uma aplicação multitarefa deve ter o mínimo de threads executáveis em qualquer momento.
  - A principal forma de atingir esse objetivo é possuir threads que executam uma quantidade de trabalho pequena e então esperam por alguma condição (wait) ou pela passagem de certo intervalo de tempo (sleep).
- Evite usar yield()
  - Funciona, mas torna o programa não-portável
- Menos portável ainda é o ajuste de prioridade (setPriority())
  - Uma melhor abordagem é reestruturar o programa para reduzir o número de threads executáveis ao mesmo tempo.

*

9. Documente a segurança em situações de concorrência (EJ 52)

- O comportamento de uma classe quando suas instâncias e métodos estáticos são expostos ao uso concorrente é parte importante do contrato que a classe estabelece com seus clientes
- É importante documentar esse comportamento
  - Caso não exista essa documentação, o usuário das suas classes terão que tomar decisões baseadas em suposições
  - Se as suposições estiverem erradas, erros sérios podem resultar
- Não há como saber se um método é ou não synchronized olhando-se apenas para a interface
  - Javadoc não indica o modificador synchronized
  - Blocos podem estar ocultos nos métodos
  - synchronized apenas é insuficiente para saber do nível de segurança que um thread suporta
- Joshua Bloch (EJ) sugere cinco níveis de segurança a seguir

Níveis de segurança (1)

- **Immutable**
  - Objetos são constantes e não podem ser mudados
- **Thread-safe**
  - Objetos são mutáveis mas podem ser usados com segurança em um ambiente concorrente pois seus métodos são sincronizados
- **Conditionally thread-safe**
  - Objetos ou possuem métodos que são thread-safe ou métodos que são chamados em seqüência com a trava mantida pelo cliente
  - É preciso indicar quais sequencias de chamada requerem sincronização externa e quais travas precisam ser obtidas para impedir acesso concorrente
  - É importante também saber em qual objeto a trava é obtida (pode ou não ser o próprio objeto). Exemplo: private lock object idiom.

Níveis de segurança (2)

- **Thread-compatible**
  - Instâncias da classe não oferecem nenhuma sincronização
  - Porém podem ser usadas com segurança em um ambiente de concorrência, se o cliente oferecer a sincronização ao incluir cada método (ou seqüência de métodos) em um bloco de trava (synchronized ou similar)
- **Thread-hostile**
  - Instâncias da classe não devem ser usadas em um ambiente concorrente mesmo que o cliente forneça sincronização externa
  - Tipicamente, uma classe thread-hostile acessa dados estáticos ou o ambiente externo

10. Evite thread groups

(EJ 53)

- Junto com threads, travas e monitores, uma abstração básica oferecida pelo sistema de threads é a de grupos de threads.
  - Servem para aplicar primitivos de Thread para várias threads ao mesmo tempo.
  - A maioria desses primitivos foi deprecada e os que restaram são muito pouco usados
- Grupos de threads não fornecem muita funcionalidade útil.
  - Do ponto de vista de segurança ThreadGroup é fraca
  - Vários de seus métodos não funcionam corretamente
- ThreadGroup.uncaughtException()
  - É o único caso que justifica o uso de ThreadGroup
  - É chamado automaticamente quando uma thread do grupo lança uma exceção que não é tratada.
  - O comportamento padrão mostra a stack trace na saída padrão de erro
- Mas agora, Thread (Java 5.0) tem uncaughtException()
  - Não há mais razão para usar ThreadGroup!

Use Executor

- Executor (da nova API de concorrência) oferece serviços de execução de threads que possuem algumas das características de Thread groups e mais vantagens
  - Pools de tamanho fixo que inicializam automaticamente vários threads
  - Pools com agendamento para inicialização de threads após um intervalo de tempo
  - Caches de tamanho variável
  - Grupos de threads que precisam executar em seqüência (sem concorrência entre si, porém com possibilidade de concorrência com outros threads)
  - Suporte a objetos ativos que causam exceção e retornam valor (Callable e Future)

11. Evite usar threads

(EJ 30)

- Programação multithreaded é mais difícil que programação singlethreaded
  - Se houver uma classe de biblioteca que evitará que você faça programação de baixo nível, use-a!
- Não usar threads – do ponto de vista do programador - significa não trabalhar diretamente com estruturas como synchronized, locks, wait, notify, etc.
  - Threads são usados em todo lugar
  - Não usar threads – do ponto de vista da aplicação - implica em lidar com eles (evitando a sincronização)
- Assim, não usar threads == usar APIs e frameworks que tornem multithreading transparente
  - Aplicações mais simples, mais fácil de usar e estender
  - Menos vulnerável a bugs, mais fácil de testar
  - Isso existe?

Fontes de referência

[JLS] James Gosling, Bill Joy, Guy Steele, Java Language Specification second edition, Addison Wesley, 2001

[SDK] Documentação do J2SDK 5.0

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

- Vários padrões foram extraídos deste livro; veja o item correspondente em cada slide

[J133] Jeremy Manson and Brian Goetz, JSR 133 (Java Memory Model) FAQ, Feb 2004

- JSR 133 em forma de perguntas e respostas, com exemplos mais compreensíveis que na especificação.
- www.cs.umd.edu/users/pugh/java/memoryModel/jsr-133-faq.html

[LEA2] Doug Lea, Synchronization and the Java Memory Model, 1999.

- Um artigo compreensível sobre o JMM (antigo, mas didático)

[LEA1] Doug Lea, Concurrent Programming in Java (1st. ed), 1996

[BJ] Bruce Tate, Bitter Java, Manning, 2002

- Vários anti-patterns com refactorings que os consertam, destacando situações onde as regras podem (ou devem) ser quebradas

Threads em Java

Padrões, anti-padrões e boas práticas

Criado em 31 de julho de 2005