Programação Concorrente em Java

Aula 06

Segurança de Threads e Recursos Compartilhados

Introdução

- Qualquer código seguro de ser acessado por várias threads é chamado de thread-safe.
- Se um código é thread-safe, então ele tem condição de corrida entre threads.
- Condições de corrida ocorrem quando múltiplas threads compartilham um mesmo recurso.
- Portanto, é muito importante conhecer quais recursos são compartilhados em um ambiente multi-thread.

Variáveis Locais

- Variáveis locais são armazenadas na pilha de cada thread.
- Nunca são compartilhadas entre threads.

```
public void someMethod(){
  long threadSafeInt = 0;
  threadSafeInt++;
}
```

Referências a objetos locais

- A referência em si não é compartilhada.
- O objeto, no entanto, é armazenado numa heap compartilhada.
- Se um objeto criado localmente nunca escapa do método o qual foi criado, então ele é thread-safe.
- Você pode até passá-lo para outros métodos contanto que não passe para outras threads.

Referências a objetos locais

```
public void someMethod(){
  LocalObject localObject = new LocalObject();
  localObject.callMethod();
  method2(localObject);
}

public void method2(LocalObject localObject){
  localObject.setValue("value");
}
```

Referências a objetos locais

- The LocalObject instance in this example is not returned from the method, nor is it passed to any other objects that are accessible from outside the someMethod() method. Each thread executing the someMethod() method will create its own LocalObject instance and assign it to the localObject reference. Therefore the use of the LocalObject here is thread safe.
- In fact, the whole method someMethod() is thread safe. Even if the LocalObject instance is passed as parameter to other methods in the same class, or in other classes, the use of it is thread safe.
- The only exception is of course, if one of the methods called with the LocalObject as parameter, stores the LocalObject instance in a way that allows access to it from other threads.

Variáveis membros de objetos

- São armazenadas dentro da heap do objeto. Logo, se duas threads chamam, na mesma instância do obejto, um método que atualiza essa variável, o método não é thread-safe.

```
public class NotThreadSafe{
    StringBuilder builder = new StringBuilder();
    public add(String text){
        this.builder.append(text);
    }
}
```

Variáveis membros de objetos

```
NotThreadSafe sharedInstance = new NotThreadSafe();
new Thread(new MyRunnable(sharedInstance)).start();
new Thread(new MyRunnable(sharedInstance)).start();

public class MyRunnable implements Runnable{
  NotThreadSafe instance = null;
  public MyRunnable(NotThreadSafe instance){
    this.instance = instance;
  }
  public void run(){
    this.instance.add("some text");
  }
}
```

```
new Thread(new MyRunnable(new NotThreadSafe())).start();
new Thread(new MyRunnable(new NotThreadSafe())).start();
```

Regra de escape de controle

- Para determinar se o acesso a um determinado recurso pelo seu código é thread-safe, deve-se fazer a seguinte checagem:
- *If a resource is created, used and disposed within the control of the same thread, and never escapes the control of this thread, the use of that resource is thread safe.*
  - Dispose = tornar a referência ao objeto nula.

Regra de escape de controle

- Mesmo que dentro da thread, o uso do objeto seja seguro, caso o objeto aponte para um recurso compartilhado (arquivo ou banco de dados) a aplicação como um todo pode não ser thread-safe. Por exemplo:
  - Thread 1 e Thread 2 ambas criam suas conexões com um banco de dados.
  - O usa de seus objetos de conexões, por cada thread é obviamente thread-safe.
  - No entanto, o uso do banco de dados, pode não ser thread-safe.

Regra de escape de controle

- Exemplo Bando de Dados
  - Imagine o seguinte código executado por ambas as threads:

```
check if record X exists
if not, insert record X
```

  - Se as duas threads executarem simultâneamente, o seguinte cenário pode acontecer:

```
Thread 1 checks if record X exists. Result = no
Thread 2 checks if record X exists. Result = no
Thread 1 inserts record X
Thread 2 inserts record X
```

  - É importante destinguir se o objeto controlado pela thread é o recurso, ou se meramente **referencia** o recurso (no caso, o objeto conexão).

Imutabilidade

- Condição de corrida ocorre quando múltiplas threads querem **escrever** no recurso compartilhado.
- Operações de **leitura** não geram condições de corrida.
- Em Java é possível tornar o objeto imutável, ou seja, o valor do objeto não pode ser atualizado pelas threads que o compartilham.

Imutabilidade

- Exemplo

```
public class ImmutableValue{

  private int value = 0;
  public ImmutableValue(int value){
    this.value = value;
  }

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

```
public class ImmutableValue{
  private int value = 0;
  public ImmutableValue(int value){
    this.value = value;
  }
  public int getValue(){
    return this.value;
  }

  public ImmutableValue add(int valueToAdd){
    return new ImmutableValue(this.value + valueToAdd);
  }

}
```

Thread Safety

- A referência não é thread-safe! Mesmo um objeto imutável e, portanto, thread-safe, sua referência pode não ser.

```
public class Calculator{
  private ImmutableValue currentValue = null;
  public ImmutableValue getValue(){
    return currentValue;
  }
  public void setValue(ImmutableValue newValue){
    this.currentValue = newValue;
  }
  public void add(int newValue){
    this.currentValue = this.currentValue.add(newValue);
  }
}
```

Thread Safety

- A classe Calculator tem uma referência para uma instância do tipo ImmutableValue.
- É possível mudar essa mudar essa **referência** através dos métodos add e setValue (como?).

Referências

- http://tutorials.jenkov.com/java-concurrency/thread-safety.html