Урок 9.
Многопоточность

Гарбузов Николай
Donriver Inc.

DonRiver

Agenda

- Потоки исполнения в Java
- Класс Thread, интерфейс Runnable
- Управление потоком
- Синхронизации
- Блокировки
- volatile и атомики
- Mutable vs immutable
- Concurrent collections

Потоки исполнения в Java

- Потоки позволяют реализовать асинхронную работу внутри приложения
- Потоки более легковесны чем процессы
- Всегда существует один Главный поток

Потоки исполнения в Java

- Потоки позволяют реализовать асинхронную работу внутри приложения
- Потоки более легковесны чем процессы
- Всегда существует один Главный поток

- В однопроцессорной и одноядерной системе достигается лишь псевдо многопоточность

Потоки исполнения в Java

- Потоки позволяют реализовать асинхронную работу внутри приложения
- Потоки более легковесны чем процессы
- Всегда существует один Главный поток

- В однопроцессорной и одноядерной системе достигается лишь псевдо многопоточность

- Потоки можно создавать, запускать и прерывать
- Можно указывать приоритет потока
- Механизмы синхронизации позволяют контролировать совместное выполнение потоков

Класс Thread, интерфейс Runnable

Поток в Java реализуется классом Thread

- **run** - содержит код для выполнения потоком
- **start** - запускает поток
- **sleep** - приостанавливает поток

Класс Thread, интерфейс Runnable

Поток в Java реализуется классом Thread

- **run** - содержит код для выполнения потоком
- **start** - запускает поток
- **sleep** - приостанавливает поток
- **join** - ожидает завершения потока
- **yield** - рекомендация планировщику потоков “уступить” выполнение текущего потока в пользу другого

Класс Thread, интерфейс Runnable

Поток в Java реализуется классом Thread

- **run** - содержит код для выполнения потоком
- **start** - запускает поток
- **sleep** - приостанавливает поток
- **join** - ожидает завершения потока
- **yield** - рекомендация планировщику потоков “уступить” выполнение текущего потока в пользу другого

- **Object.wait** - заставляет текущий поток ждать пока не будет вызван **Object.notify** или **Object.notifyAll**

Класс Thread, интерфейс Runnable

Поток в Java реализуется классом Thread

- **currentThread** - возвращает текущий поток
- **getName** - возвращает имя потока

Класс Thread, интерфейс Runnable

Реализовав Runnable можно определить метод run и передавать реализацию потоку Thread для выполнения

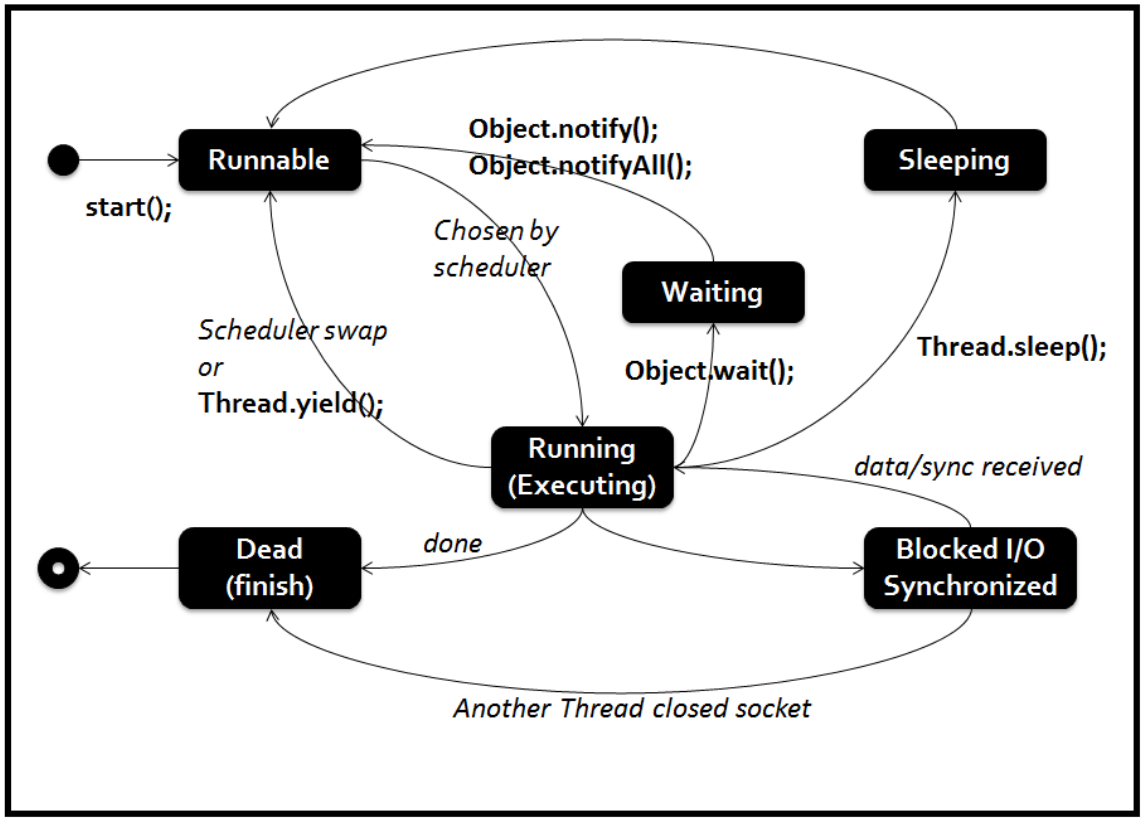
Класс Thread, интерфейс Runnable

Реализовав Runnable можно определить метод run и передавать реализацию потоку Thread для выполнения

- Thread имплементирует Runnable
- Код в переопределенном методе Thread.run имеет больший приоритет при выполнении перед выполнением переданного Runnable

Управление потоком

Диаграмма состояний потока (getState)



DonRiver

Управление потоком

Q:
Потоки завершаются независимо друг от друга.
Как заставить один поток дождаться завершения другого?

Управление потоком

Q:
Потоки завершаются независимо друг от друга.
Как заставить один поток дождаться завершения другого?

A:
Использовать **Thread.join** для ожидания завершения потока

Синхронизации

- Каждый объект в Java имеет связанный с ним монитор (Intrinsic lock)

Синхронизации

- Каждый объект в Java имеет связанный с ним монитор (Intrinsic lock)

- Чтобы получить монитор объекта нужно войти в synchronized секцию (метод или блок кода)
- Допускается повторный захват монитора одним и тем же потоком! (Reentrant lock)

Синхронизации

- Synchronized секции порождают отношение happens-before

Синхронизации

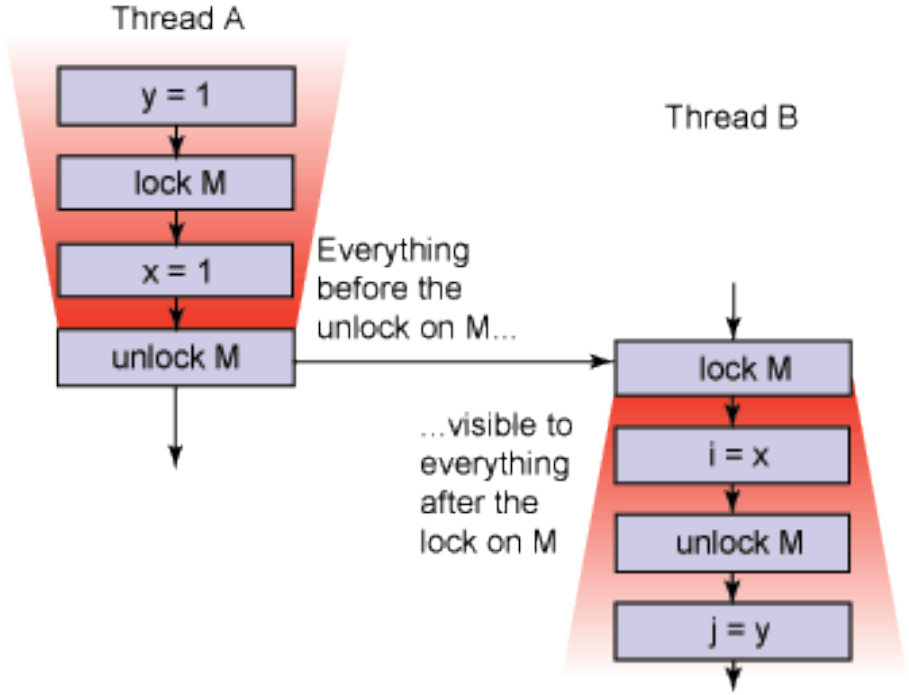
- Synchronized секции порождают отношение happens-before

Пусть есть поток **X** и поток **Y**. И пусть есть операции **A** в потоке **X** и **B** в потоке **Y**.

В таком случае, **A happens-before B** означает, что все изменения, выполненные потоком **X** до момента операции **A** и изменения, которые повлекла эта операция, видны потоку **Y** в момент выполнения операции **B** и после выполнения этой операции.

Синхронизации

- Synchronized секции порождают отношение happens-before



DonRiver

Синхронизации

- Synchronized секции порождают отношение happens-before

Пусть есть поток **X** и поток **Y** (не обязательно отличающийся от потока **X**). И пусть есть операции **A** (выполняющаяся в потоке **X**) и **B** (выполняющаяся в потоке **Y**).

В таком случае, **A happens-before B** означает, что все изменения, выполненные потоком **X** до момента операции **A** и изменения, которые повлекла эта операция, видны потоку **Y** в момент выполнения операции **B** и после выполнения этой операции.

Синхронизации

- Synchronized секции порождают отношение happens-before

Пусть есть поток **X** и поток **Y** (не обязательно отличающийся от потока **X**). И пусть есть операции **A** (выполняющаяся в потоке **X**) и **B** (выполняющаяся в потоке **Y**).

В таком случае, **A happens-before B** означает, что все изменения, выполненные потоком **X** до момента операции **A** и изменения, которые повлекла эта операция, видны потоку **Y** в момент выполнения операции **B** и после выполнения этой операции.

- Атомарность
- Видимость
- Оптимизации (перестановки) байт-кода

Блокировки

Используются для организации взаимодействия между потоками

- **Object.wait** - заставляет текущий поток ждать пока не будет вызван Object.notify или Object.notifyAll
- **Object.notify** возобновляет один из потоков выполнения
- **Object.notifyAll** возобновляет все потоки

Блокировки

Используются для организации взаимодействия между потоками

- **Object.wait** - заставляет текущий поток ждать пока не будет вызван Object.notify или Object.notifyAll
- **Object.notify** возобновляет один из потоков выполнения
- **Object.notifyAll** возобновляет все потоки

Иногда возникают “спонтанные пробуждения” (spurious wakeups)

Блокировки

Специальные коллекции могут использоваться для организации взаимодействия между потоками

- **Array, Linked, Priority blocking queue** читатель будет ждать добавление элементов писателем
- **LinkedBlockingDeque** читатель будет ждать добавление элементов писателем
- **DelayQueue** читатель будет получать только элементы, у которых истек “срок действия”
- **SynchronousQueue** блокирующая очередь с 1 элементом

Блокировки

Блокировки могут приводить к **взаимным блокировкам**

Блокировки

Блокировки могут приводить к **взаимным блокировкам**

- **Deadlock** полная взаимная блокировка
- **Starvation** ресурсное“голодание”
- **Livelock** не полная блокировка, но зацикливание выполнения кода

volatile и атомики

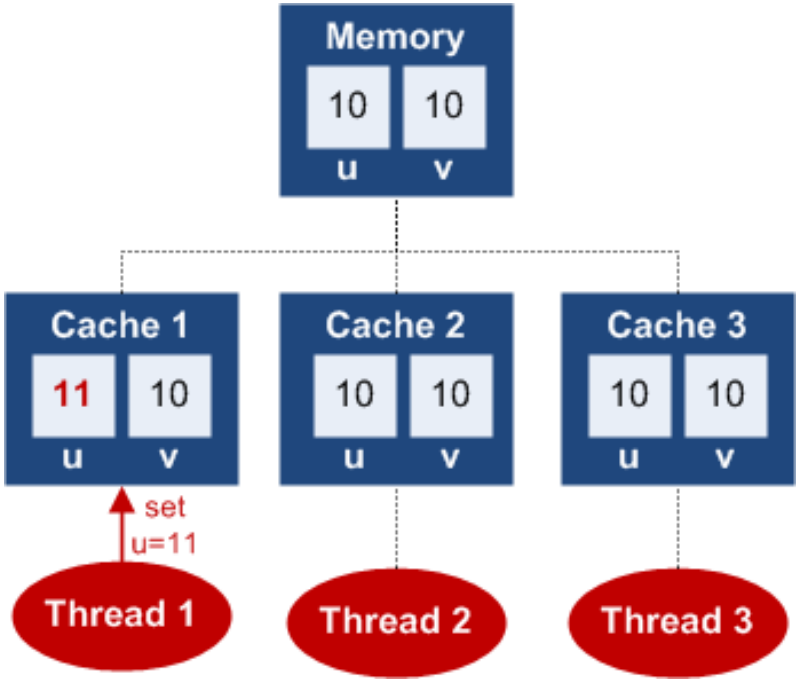
- Синхронизации обеспечивают эксклюзивный доступ к ресурсу
- Но синхронизация ухудшает конкурентное многопоточное выполнение кода

volatile и атомики

- Синхронизации обеспечивают эксклюзивный доступ к ресурсу
- Но синхронизация ухудшает конкурентное многопоточное выполнение кода

Нужны средства позволяющие писать
корректные программы, поддерживающие
многопоточность,
без синхронизаций

volatile и атомики



DonRiver

volatile и атомики

- **volatile** гарантирует что все потоки будут видеть одно и то же значение переменной
- **volatile** приводят к отношению happens-before

- **volatile** гарантируют атомарность при записи long & double в 32 битных системах!

volatile и атомики

- **volatile** гарантирует что все потоки будут видеть одно и то же значение переменной
- **volatile** приводят к отношению happens-before

- **volatile** гарантируют атомарность при записи long & double в 32 битных системах!

Q:
Можно ли использовать volatile для операций инкремента?

volatile и атомики

- **volatile** гарантирует что все потоки будут видеть одно и то же значение переменной
- **volatile** приводят к отношению happens-before

- **volatile** гарантируют атомарность при записи long & double в 32 битных системах!

Q:
Можно ли использовать volatile для операций инкремента?

A:
Нельзя! Потому что операция инкремента не атомарна!

volatile и атомики

- **atomic variable** гарантирует корректное поведение при работе с переменными в многопоточной среде, использует volatile внутри

- **atomic variable** используют compare-and-set механизм для реализации не-атомарных операци!

Q:
Можно ли использовать atomic variable для операций инкремента?

A:
Можно и нужно!

Mutable vs immutable

- **Mutable** (изменяемые) объекты не безопасно использовать в многопоточной системе без дополнительных синхронизаций
    - определяют зависимые поля в объекте, формирующие его состояние
    - используют единую синхронизацию для каждой группы полей
- **Immutable** (не изменяемые) объекты безопасно использовать в многопоточнойй среде
    - используют **final** при объявлении полей для достижения неизменяемость + получения happens-before при работе с полями

Concurrent collections

- Обычные коллекции не подходят и не безопасны для конкурентого использования
- Синхронизованные коллекции безопасны для конкурентого использования, но фактически не подходят для этого

Concurrent collections

- Обычные коллекции не подходят и не безопасны для конкурентого использования
- Синхронизованные коллекции безопасны для конкурентого использования, но фактически не подходят для этого

- Конкурентные коллекции подходят и безопасны для конкурентного использования

Concurrent collections

- **CopyOnWriteArrayList** копирует массив при изменении
- **ConcurrentHashMap** использует volatile и compare-and-swap для достижения корректного многопоточного доступа
- **ConcurrentLinkedQueue** использует compare-and-swap для атомарного обновления элементов

Home work

Многопользовательский сетевой чат.

Использовать

- Socket (https://docs.oracle.com/javase/tutorial/networking/sockets/readingWriting.html)
- 2 потока на сервере - один читает с сокета и кладет сообщения в потоко-безопасную очередь `BlockingQueue`, другой отправляет ответы через сокет
- 2 потока на клиенте - один читает ввод с клавиатуры, другой выводит на консоль пришедшие с сервера сообщения