More on Threads

Synchronized methods

Wait and notify

Concurrent data types

What was a Thread?
Do you remember?

- Individual and separate unit of execution that is part of a process
  - multiple threads can work together to accomplish a common goal
- Video Game example
  - one thread for graphics
  - one thread for user interaction
  - one thread for networking

Thread Synchronization

- Monitors
  - Object with **`synchronized`** methods
    - Any object can be a monitor
  - Methods declared **`synchronized`**
    - **`public synchronized int myMethod( int x )`**
    - Only one thread can execute a **`synchronized method`** at a time
      - *Obtaining the lock* and *locking* an object
    - If multiple **`synchronized`** methods, only one may be active
  - Java also has **`synchronized`** blocks of code

Thread Synchronization

- Thread may decide it cannot proceed
  - May voluntarily call **`wait`** while accessing a **`synchronized`** method
    - Removes thread from contention for monitor object and processor
    - Thread in waiting state
  - Other threads try to enter monitor object
    - Suppose condition first thread needs has now been met
    - Can call **`notify`** to tell a single waiting thread to enter ready state
    - **`notifyAll`** - tells all waiting threads to enter ready state

Daemon Threads

- Daemon threads
  - Threads that run for benefit of other threads
    - E.g., garbage collector
  - Run in background
    - Use processor time that would otherwise go to waste
  - Unlike normal threads, do not prevent a program from terminating - when only daemon threads remain, program exits
  - Must designate a thread as daemon before **`start`** called:

```
void setDaemon( true );
```

  - Method **`boolean isDaemon()`**
    - Returns **`true`** if thread is a daemon thread

Synchonized blocks

- Synchronized blocks of code

```
synchronized( monitorObject ){
  ...
}
```

  - **monitorObject**- Object to be locked while thread executes block of code
- Suspending threads
  - In earlier versions of Java, there were methods to stop/suspend/resume threads
    - Why have these methods been deprecated?
      - Dangerous, can lead to deadlock
  - Instead, use **wait** and **notify**
    - **wait** causes current thread to release ownership of a monitor until another thread invokes the **notify** or **notifyAll** method

Implementation of Java synchronization

- Every object has an intrinsic lock associated with it.

- A thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it is done with them.

Example using synchronized methods

On-line banking

•Several entities can access account potentially simultaneously (maybe a joint account, maybe automatic debits, …)

•Suppose three entities each trying to perform an operation, either:

- deposit()
- withdraw()
- enquire()

Create three threads, one for each action

```
public class InternetBankingSystem {

    public static void main(String[] args)
    {

        Account accountObject = new Account();
        Thread t1 = new Thread(new MyThread(accountObject) );
        Thread t2 = new Thread(new YourThread(accountObject) );
        Thread t3 = new Thread(new HerThread(accountObject) );

        t1.start();
        t2.start();
        t3.start();

    }
```

Three threads work with a shared object

```
static class MyThread implements Runnable  {
     Account account;
            public MyThread (Account s) {  account = s;}
            public void run() { account.deposit(100); }
    } // end class MyThread

static class YourThread implements Runnable  {
     Account account;
            public YourThread (Account s) { account = s;}
            public void run() { account.withdraw(100); }
    } // end class YourThread

static class HerThread implements Runnable  {
     Account account;
            public HerThread (Account s) { account = s; }
            public void run() {account.enquire(); }
    } // end class HerThread
```

Each thread executes a different action on the object

IT TALENTS Training Camp

Synchronized account methods

```
public class Account {
       int balance;
       Account()
       {
           System.out.println(" initialized an account with balance = " + balance);
       }
    // if 'synchronized' is removed, outcome unpredictable

     public synchronized void deposit(int deposit_amount ) {
         System.out.println("depositing " + deposit_amount);
        balance += deposit_amount;
     }

   public synchronized void withdraw(int deposit_amount ) {
         System.out.println("withdraw " + deposit_amount);
        balance -= deposit_amount;
   }
   public synchronized void enquire( ) {
        System.out.println("balance = " + balance);
   }

}
```

What happens when methods are not synchronized

```
 initialized an account with balance = 0
depositing 100
withdraw 100
balance = 100
```

```
 initialized an account with balance = 0
withdraw 100
balance = 0
depositing 100
```

When methods are syncronyzed the results are adequate

```
 initialized an account with balance = 0
depositing 100
withdraw 100
balance = 0
```

```
 initialized an account with balance = 0
withdraw 100
balance = -100
depositing 100
```

```
 initialized an account with balance = 0
depositing 100
balance = 100
withdraw 100
```

Synchronized Statements

Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
synchronized (object) {

    //do something here

}
```

Evaluate to an object or an array.
Used to identify lock

Critical section

```
public void addName(String name) {
    synchronized(this)
    {
        lastName = name;
        nameCount++;
    }
    System.out.println(name);
}
```

Only this part
Is synchronized

Deadlock ?

What happens if you have two things that do this - deadlock

```
public void doSomething() {
Synchronized (a) {
  Synchronized (b) {
  // code
  } }
}
```

```
public void doOther() {
Synchronized (b) {
  Synchronized (a) {
  // code
  } }
}
```

Atomic action

An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

Read/writes can be declared atomic with the **volatile** keyword, e.g.

```
private volatile int x;
```

Sometimes can be more efficient than synchronized methods.

Volatile

- What is the Java volatile keyword?
  - Essentially, volatile is used to indicate that a variable's value will be modified by different threads.
  - Declaring a volatile Java variable means:
    - The value of this variable will never be cached thread-locally: all reads and writes will go straight to "main memory";
    - Access to the variable acts as though it is enclosed in a synchronized block, synchronized on itself.

Volatile

- Difference between synchronized and volatile

Characteristic	Synchronized approach	Volatile approach
Type of variable	Object	Object or primitive
Null allowed ?	No	Yes
Can block ?	Yes	No
When synchronizations happens ?	When you explicitly enter/exit a synchronized block	Whenever a volatile variable is accessed.
Can be used to combine several operations in an atomic operation ?	Yes	Possible – Atomic get-set of volatiles

Volatile

- What did that all meant ?
  - a primitive variable may be declared volatile (whereas you can't synchronize on a primitive with synchronized);
  - an access to a volatile variable never has the potential to block: we're only ever doing a simple read or write, so unlike a synchronized block we will never hold on to any lock;
  - because accessing a volatile variable never holds a lock, it is not suitable for cases where we want to read-update-write as an atomic operation (unless we're prepared to "miss an update");
  - a volatile variable that is an object reference may be null (because you're effectively synchronizing on the reference, not the actual object).
  - Attempting to synchronize on a null object will throw a NullPointerException.

Volatile vs Static

- Difference Between Static and Volatile :
  - Static Variable: If two Threads(suppose t1 and t2) are accessing the same object and updating a variable which is declared as static then it means t1 and t2 can make their own local copy of the same object(including static variables) in their respective cache, so update made by t1 to the static variable in its local cache wont reflect in the static variable for t2 cache .
  - Static variables are used in the context of Object where update made by one object would reflect in all the other objects of the same class but not in the context of Thread where update of one thread to the static variable will reflect the changes immediately to all the threads (in their local cache).

Volatile vs Static

- Difference Between Static and Volatile :
  - Volatile variable: If two Threads(suppose t1 and t2) are accessing the same object and updating a variable which is declared as volatile then it means t1 and t2 can make their own local cache of the Object except the variable which is declared as a volatile . So the volatile variable will have only one main copy which will be updated by different threads and update made by one thread to the volatile variable will immediately reflect to the other Thread.

Coordinating threads

- Sometimes we need a thread to stop running and wait for an event before continuing.
- **wait()** and **notify()** methods are methods of class Object.
- Every object can maintain a list of waiting threads.
- **Wait()** - When a thread calls **wait()** method of an object, any locks the thread holds are temporarily released and the thread is added to a list of waiting threads for that object and stops running.
- **Notify()** - When another thread calls **notify()** method on the same object, object wakes up one of the waiting threads and allows it to continue.

Join

- Sometimes one thread needs to stop and wait for another thread to complete.

- **join()** -- waits for a thread to die, i.e. **thr1.join()** waits for thread **thr1** to die.

- Calling **return()** from the run method implicitly causes the thread to exit.

Example for wait/notify

```
public class Drop {

    // Message sent from producer to consumer
    private String message;

    // A flag, True if consumer should wait for
    // producer to send message, False if producer
    // should wait for consumer to retrieve message
    private boolean empty = true;
```

Flag must be used, never Count only on the notify

Example for wait/notify

Must be in synchronized context

```
public synchronized String take() {
    // Wait until message is available
    while (empty) {
        // we do nothing on InterruptedException
        // since the while condition is checked anyhow
        try { wait(); } catch (InterruptedException e) {}
    }
    // Toggle status and notify on the status change
    empty = true;
    notifyAll();
    return message;
}
```

Example for wait/notify

Must be in synchronized context

```
public synchronized void put(String message) {
    // Wait until message has been retrieved
    while (!empty) {
        // we do nothing on InterruptedException
        // since the while condition is checked anyhow
        try { wait(); } catch (InterruptedException e) {}
    }
    // Toggle status, store message and notify consumer
    empty = false;
    this.message = message;
    notifyAll();
}
```

Concurrent collections

- ConcurrentHashMap
  - The most popular collection class and most used.
  - Provides a concurrent alternative of Hashtable or Synchronized Map classes with aim to support higher level of concurrency by implementing fined grained locking. Multiple readers can access the Map concurrently while a portion of the Map gets locked for write operations.
  - Iterator of ConcurrentHashMap are fail-safe iterators which doesn't throw ConcurrencModificationException thus eliminates another requirement of locking during iteration which result in further scalability and performance.

Concurrent collections

- CopyOnWriteArrayList and CopyOnWriteArraySet
  - a concurrent alternative of synchronized List.
  - Provides better concurrency by allowing multiple concurrent readers and replacing the whole list on write operation. Yes, write operation is costly, but it performs better when there are multiple readers and requirement of iteration is more than writing.
  - Since CopyOnWriteArrayList Iterator also don't throw ConcurrencModificationException it eliminates need to lock the collection during iteration.

Concurrent collections

- BlockingQueue
  - Makes it easy to implement producer-consumer design pattern by providing inbuilt blocking support for put() and take() method. put() method will block if Queue is full while take() method will block if Queue is empty.
  - Two concrete implementations - ArrayBlockingQueue and LinkedBlockingQueue, both FIFO. Consider using BlockingQueue to solve producer Consumer problem in Java instead of writing your won wait-notify code.

Concurrent collections

- Deque and BlockingDeque
  - Support insertion and removal from both end of Queue referred as head and tail.
  - Can be used efficiently to increase parallelism in program by allowing a set of worker threads to help each other by taking some of the work load by utilizing Deque double end consumption property. So if all Threads have their own set of task Queue and they are consuming from head; helper thread can also share some work load via consumption from tail.

Concurrent collections

- ConcurrentSkipListMap and ConcurrentSkipListSet
  - Just like ConcurrentHashMap provides a concurrent alternative of synchronized HashMap.
  - Provide concurrent alternative for synchronized version of SortedMap and SortedSet. For example instead of using TreeMap or TreeSet wrapped inside synchronized Collection, You can consider using ConcurrentSkipListMap or ConcurrentSkipListSet from java.util.concurrent package.