15. Threads and Locks

Threads in Java

Every thread in Java is created and controlled by a unique object of the java.lang.Thread class. When a standalone application is run, a user thread is automatically created to execute the main() method. This thread is called the main thread.

In Java, we can implement threads in one of two ways:

- Implement the java.lang.Runnable interface
- Extend java.lang.Thread class

Implementing the Runnable Interface

The Runnable interface has the following structure:

```
public interface Runnable {
  void run();
}
```

To create and use a thread using this interface:

1. Create a class that implements the Runnable interface. An object of this class is a Runnable object
2. Create an object of type Thread by passing a Runnable object as argument to the Thread constructor. The Thread object now has a Runnable object that implements the run() method.
3. The start() method is invoked on the Thread object created in the previous step.

Example:

```
public class RunnableThreadExample implements Runnable {
  public int count = 0;

  public void run() {
    System.out.println("Runnable Thread running");
    try {
      while (count < 5) {
        Thread.sleep(500);
        count++;
      }
    } catch (InterruptedException exc){
      System.out.println("Runnable Thread interrupted");
    }
    System.out.println("RunnableThread terminating");
  }
}

public static void main(String[] args) {
  RunnableThreadExample instance = new RunnableThreadExample();

  Thread thread = new Thread(instance);
  thread.start();

  while (instance.count != 5) {
    try {
      Thread.sleep(250);
    } catch (InterruptedException exc) {
      exc.printStackTrace();
    }
  }
}
```

All we have to do is have our class implement the `run()` method. Another method can then pass an instance of the class to the new `Thread(obj)` and call `start()`

Extending the Thread Class

This will almost always mean that we override the `run()` method, and the subclass may also call the constructor explicitly in its constructor.

```
public class ThreadExample extends Thread {
  int count = 0;

  public void run() {
    System.out.println("thread starting");
    try {
      while (count < 5) {
        Thread.sleep(500);
        System.out.println("In thread, count is " + count);
        count++;
      }
    } catch (InterruptedException exc) {
      System.out.println("Thread interrupted");
    }
    System.out.println("Thread terminating");
  }
}

public class ExampleB {
  public static void main(String[] args) {
    ThreadExample instance = new ThreadExample();
    instance.start();

    while (instance.count != 5) {
      try {
        Thread.sleep(250);
      } catch (InterruptedException exc) {
        exc.printStackTrace();
      }
    }
  }
}
```

The difference is that since we are extending the Thread class, we can call `start()` on the instance of the class itself (instead of needing to create new Thread and passing the instance)

Extending the Thread Class vs. Implementing the Runnable Interface

When creating threads, there are two reasons why implementing the Runnable interface maybe preferable to extending the Thread class.

1. Java does not support multiple inheritance. Therefore, extending the Thread class means that the subclass cannot extend any other class. A class implementing the Runnable will still be able to extend another class.
2. A class might only be interested in being runnable. Therefore, inheriting the full overhead of the Thread class would be excessive.

Synchronization and Locks

Threads within a given process share the same memory space, which is both a positive and a negative. It enables threads to share data, but it also creates the opportunity for issues when the two threads modify a resource at the same time. Java provides Synchronization in order to control access to shared resources.

The keyword `synchronized` and the `lock` form the basis for implementing synchronized execution of code.

Synchronized Methods

Most commonly, we restrict access to shared resources through the use of `synchronized` keyword. It can be applied to methods and code blocks, and restricts multiple threads from executing the code simultaneously on the same object.

`public static synchronized void foo(){ ... }`

Synchronized Blocks

A block of code can be synchronized. This operates very similarly to synchronizing a method.

```
public class MyClass extends Thread {
  ...
  public void run() {
    myObj.foo(name);
  }
}

public class MyObject {
  public void foo(String name) {
    synchronized(this) {
      ...
    }
  }
}
```

Like synchronizing a method, only one thread per instance of MyObject can execute the code within the synchronized block. If thread1 and thread2 have the same instance of MyObject, only one will be allowed to execute the code block at a time.

Locks

For more granular control, we can utilize a lock. A lock is used to synchronize access to a shared resource by first acquiring the lock associated with the resource. At any time, at most one thread can hold the lock, and therefore, only one thread can access the shared resource.

```
public class LockedATM{
  private Lock lock;
  private int balance = 100;

  public LockedATM() {
    lock = new ReentrantLock();
  }

  public int withdraw(int value) {
    lock.lock();
    int temp = balance;
    try {
      Thread.sleep(100);
      temp = temp - value;
      Thread.sleep(100);
      balance = temp;
    } catch (InterruptedException e) { ... }
```

```
      lock.unlock();
      return temp;
    }
  }

  public int deposit(int value) {
    lock.lock();
    int temp = balance;
    try {
      Thread.sleep(100);
      temp = temp + value;
      Thread.sleep(300);
      balance = temp;
    } catch (InterruptedException e) { ... }
    lock.unlock();
    return temp;
  }
```

Deadlocks and Deadlock Prevention

A deadlock is a situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds.

In order for a deadlock to occur, you must have all 4 of the following conditions met:

1. Mutual Exclusion: There is limited access to a resource.
2. Hold and Wait: Process already holding a resource can request additional resource without relinquishing their current resources.
3. No preemption: One process cannot forcibly remove another process's resource.
4. Circular Wait: 2 or more processes form a circular chain where each process is waiting on another resource in the chain.

Deadlock prevention entails removing any of the above conditions, but it's quite tricky. Removing #1 is difficult because many resourcecs can only be used by one process at a time (e.g. printers). Most deadlock prevention algorithms focus on avoiding condition 4: circular wait.

Questions

Question 1: Threads vs. Process: what is the difference?

A process can be thought of as an instance of a program in execution. A process is an independent entity to which system resources are allocated. Each process is executed in a separate address space, and one process cannot access the variables and data structures of another process. If a process wishes to access another process's resources, inter-process communications have to be used. These include pipes, files, sockets, and other forms.

A thread exists within a process and shares the process's resources. Multiple threads within the same process will share the same heap space. This is different from processes, which cannot directly access the memory of another process. Each thread still has its own registers and its own stack, but other threads can read and write the heap memory.

A thread is a particular execution path of a process. When one thread modifies a process resource, the change is immediately visible to sibling threads.

Question 3: Dining philosophers. A bunch of philosophers are sitting around a circular table with one chopstick between each of them. A philosopher needs both chopsticks to eat and always picks up the left chopstick before the right one. A deadlock could potentially occur if all the philosophers reached for the left chopstick at the same time. Using threads and locks, implement a simulation of the dining philosophers problem that prevents deadlocks.

First, we implement a simple simulation of the dining philosophers in which we don't care about deadlocks. We'll have Philosopher extend Thread and Chopstick call lock.lock() when it is picked up and lock.unlock() when it is put down.

```
class Chopstick {
  private Lock lock;

  public Chopstick(){
    lock = new ReentrantLock();
  }

  public void pickUp() {
    void lock.lock();
  }

  public void putDown() {
    void lock.unlock()
  }
}

class Philosopher extends Thread {
  private int bites = 10;
  private Chopstick left, right;

  public Philosopher(Chopstick left, Chopstick right) {
    this.left = left;
    this.right = right;
  }
  public void eat() {
    pickUp();
    chew();
    putDown();
  }

  public void pickUp(){
    left.pickUp();
    right.pickUp();
  }

  public void chew() { }

  public void putDown() {
    left.putDown();
    right.putDown();
  }

  public void run(){
    for (int i = 0; i < bites; i++){
      eat();
    }
  }
}
```

Running the above code may lead to a deadlock if all the philosophers have a left chopstick and are waiting for the right one.

Solution 1: All or Nothing

To prevent deadlocks, we can implement a strategy where a philosopher will put down his left chopstick if he is unable to obtain the right one.

```
  public class Chopstick {
    // same as before

    public boolean pickUp(){
      return lock.tryLock();
    }
  }

  public class Philosopher extends Thread {
    // same as before

    public void eat() {
      if(pickUp()) {
        chew();
        putDown();
      }
    }

    public boolean pickUp(){
      // attempt to pick up
      if (!left.pickUp()) {
        return false;
      }
      if (!right.pickUp()) {
        left.putDown();
        return false;
      }
      return true;
    }
  }
```

In the above code, we need to be sure to release the left chopstick if we can't pick up the right chopstick and to not call putDown() on chopstick if we never had them in the first place.

One issue with this is that if all the philosophers were perfectly synchronized, they could simultaneously pick up their left chopstick, be unable to pick up the right one, and then put back down the left one - only to have the process repeated again.

Solution 2: Prioritized Chopsticks

Alternatively, we can label the chopsticks with a number from 0 to N-1. Each philosopher attempts to pick up the lower numbered chopstick first.

```
  public class Philosopher extends Thread {
    private int bites = 10;
    private Chopstick lower, higher;
    private int index;
    public Philosopher(int i, Chopstick left, Chopstick right) {
      index = i;
      if(left.getNumber() < right.getNumber()) {
        this.lower = left;
        this.higher = right;
      } else {
        this.lower = right;
        this.higher = left;
      }
    }
    public void eat() {
      pickUp();
      chew();
      putDown();
    }
    public void pickUp(){
      lower.pickUp();
      higher.pickUp();
    }
    public void chew() { ... }

    public void putDown() {
      higher.putDown();
      lower.putDown();
    }

    public void run() {
      for (int i = 0; i < bites; i++){
        eat();
      }
    }
  }

  public class Chopstick {
    private Lock lock;
    private int number;

    public Chopstick(int n) {
      lock = new ReentrantLock();
      this.number = n;
    }
    public void pickUp() {
      lock.lock();
    }
    public void putDown() {
      lock.unlock();
    }
    public int getNumber(){
      return number;
    }
  }
```

With this solution, a philosopher can never hold the larger chopstick without holding the smaller one. This prevents the ability to have a cycle, since a cycle means that a higher chopstick would "point" to a lower one.

Question 5: Call in Order.

Suppose we have the following code:

```
public class Foo {
  public Foo() { ... }
  public void first() {...}
  public void second() {...}
  public void third() {...}
}
```

The same instance of Foo will be passed to 3 different threads. ThreadA will call first, threadB will call second, and threadC will call third.
Design a mechanism to ensure that first is called before second and second before third.

Solution
Initially we would think: The general logic is to check if first() has completed before executing second(), and second() before third(). Since we need to be careful about thread safety, we can't simply use boolean flags.

What about using a lock? It actually won't quite work due to the concept of lock ownership. One thread is actually performing the lock, but other threads attempt to unlock the locks. Instead, we can use semaphores.

A semaphore controls access to a shared resource through the use of a counter. If the counter > 0, then access is allowed. If counter == 0, then access is denied. What the counter is counting are permits that allow access to the shared resource.

```
public class Foo {
  public Semaphore sem1, sem2;

  public Foo() {
    try {
      sem1 = new Semaphore(1);
      sem2 = new Semaphore(2);

      sem1.acquire();
      sem2.acquire();
    } catch (...) {...}
  }
  public void first() {
    try {
      ...
      sem1.release();
    } catch (...) {...}
  }
  public void second() {
    try {
      sem1.acquire();
      sem1.release();
      ...
      sem2.release();
    } catch (...) {...}
  }

  public void third() {
    try {
      sem2.acquire();
      sem2.release();
      ...
    } catch (...) {...}
  }
}
```

Question 6: Synchronized Methods: You are given a class with synchronize method A and a normal method B. If you have two threads in one instance of a program, can they both execute A at the same time. Can they execute A and B at the same time?

By applying the word synchronized to a method, we ensure that two threads cannot execute synchronized methods on the same object instance at the same time.

First part: if the two threads have the same instance of the object, then no. However, if they have different instances of the object, then yes.

Second: we're asked if thread1 can execute synchronized method A while thread2 is executing non-synchronized method B. Since B is not synchronized, there is nothing to block thread1 from executing A while thread2 is executing B. This is true regardless of whether thread1 and thread2 have the same instance of the object.

Ultimately, the key concept to remember is that only one synchronized method can be in execution per instance of that object. Other threads can execute non-synchronized methods on that instance, or they can execute any method on a different instance of the object.