

Chapter 19

Java Never Ends (Multithreading)

Prof. Choonhwa Lee

Dept. of Computer Science and Engineering
Hanyang University

Threads

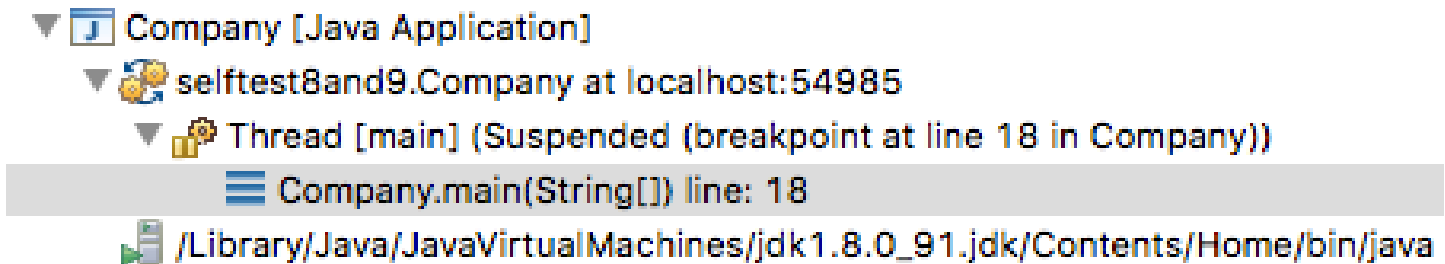
- A *thread* is a separate computation process
 - A thread is a single sequential flow of control within a program.
 - A thread *does not have its own address space* but uses the memory and other resources of the process in which it executes.
 - There may be several threads in one process
- Threads are lightweight processes, as the overhead of switching between threads is less

Multithreading

- In Java, programs can have multiple threads
- Threads are often thought of as computations that run in parallel
 - Although they usually do not really execute in parallel
 - Instead, the computer switches resources between threads, so that each one does a little bit of computing in turn
- Multithreading is a programming concept where a program (process) is divided into two or more subprograms (threads), which can be implemented at the same time in parallel

Multithreading

- We have experienced threads
 - The Java Virtual Machine spawns a thread, when your program is run, called the Main Thread



Multithreading

Single Thread

class ABC

{

.....

.....

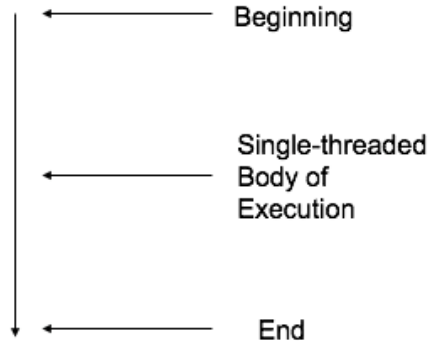
.....

.....

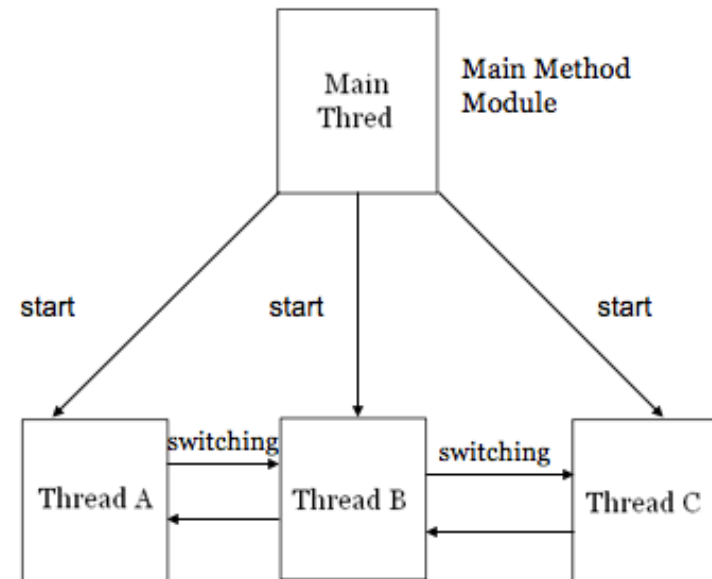
.....

.....

}



Multi-thread



Multithreading

- Why do we need threads?
 - To enhance parallel processing
 - To increase response to the user
 - To utilize the idle time of the CPU
 - Prioritize your work depending on priority
- Examples
 - Video games
 - Web server
 - The Web server listens for request and serves it

Threads in Java

- Two ways to use threads

1. Extending the class **Thread**.

```
public class MyThreadClass extends Thread  
{  
  
    ...  
  
}
```

2. Implementing **Runnable** interface.

```
public class MyThreadClass implements Runnable  
{  
  
    ...  
  
}
```

Threads in Java

- The **Thread** class and **Runnable** interface contains the method `run()`
 - The `public void run()` method acts like the main method of a traditional sequential program
 - We must *override or implement* the `run()` method so that our thread can perform the actions we need
- It is usually more preferred to implement the **Runnable** Interface, so that we can extend properties from other classes

The Class **Thread**

- In Java, a thread is an object of the class **Thread**
- Usually, a derived class of **Thread** is used to program a thread
 - The methods **run** and **start** are inherited from **Thread**
 - The derived class overrides the method **run** to program the thread
 - The method **start** initiates the thread processing and invokes the **run** method

The Runnable Interface

- Another way to create a thread is to have a class implement the **Runnable** interface
 - The **Runnable** interface has one method heading:
`public void run();`
- A class that implements **Runnable** must still be run from an instance of **Thread**
 - This is usually done by passing the **Runnable** object as an argument to the thread constructor

The Runnable Interface

```
public class ClassToRun extends SomeClass implements
    Runnable
{ . . .
    public void run()
    {
        // Fill this as if ClassToRun
        // were derived from Thread
    }
    . . .
    public void startThread()
    {
        Thread theThread = new Thread(this);
        theThread.start();
    }
    . . .
}
```

Thread Lifecycle

When threads are created they may go through the following states.

New (Newborn)

Runnable

Running

Blocked

Terminated (Dead)

The states of threads may be controlled by these methods.

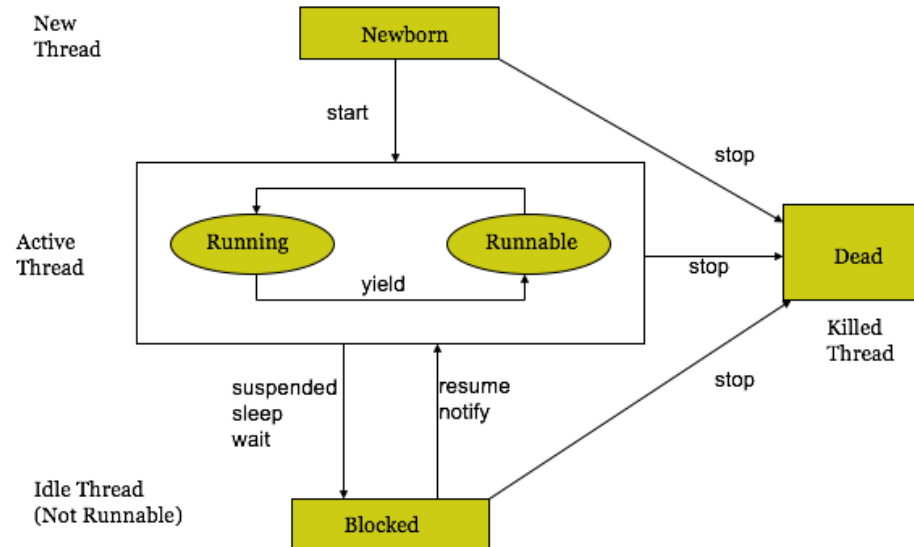
start()

yield()

suspend(), sleep()

resume()

stop()



Thread Lifecycle

```
class MyThread implements Runnable{
    public void run(){
        System.out.println("Thread Started");
    }
}

class MainClass {
    public static void main(String args[]){
        Thread t = new Thread(new MyThread());
        t.start(); // starts the thread by running the run method
    }
}
```

- Calling `t.run()` does not start a thread. It is just a simple method call.
- Creating an object does not create a thread. Calling `start()` method creates the thread.

Race Conditions

- When multiple threads change a shared variable, it is sometimes possible that the variable will end up with the wrong (and often unpredictable) value.
- This is called a race condition, because the final value depends on the sequence in which the threads access the shared value.
- We will use the Counter class to demonstrate a race condition.

Counter Class

Display 19.4 The Counter Class

```
1  public class Counter
2  {
3      private int counter;
4      public Counter()
5      {
6          counter = 0;
7      }
8      public int value()
9      {
10         return counter;
11     }
12     public void increment()
13     {
14         int local;
15         local = counter;
16         local++;
17         counter = local;
18     }
19 }
```

Race Condition Example


1. Create a single instance of the Counter class.
2. Create an array of many threads (30,000 in the example) where each thread references the single instance of the Counter class.
3. Each thread runs and invokes the increment() method.
4. Wait for each thread to finish and then output the value of the counter. If there were no race conditions, then its value should be 30,000. If there were race conditions, then the value will be less than 30,000.

Race Condition Test Class (1 of 3)

Display 19.5 The RaceConditionTest Class

```
1 public class RaceConditionTest extends Thread
2 {
3     private Counter countObject;
4
5     public RaceConditionTest(Counter ctr)
6     {
7         countObject = ctr;
8     }
9 }
```

*Stores a reference to a
single Counter object.*



Race Condition Test Class (2 of 3)

```
8     public void run()
9     {
10         countObject.increment();
11     }

12     public static void main(String[] args)
13     {
14         int i;
15         Counter masterCounter = new Counter();
16         RaceConditionTest[] threads = new RaceConditionTest[30000];

17         System.out.println("The counter is " + masterCounter.value());
18         for (i = 0; i < threads.length; i++)
19         {
20             threads[i] = new RaceConditionTest(masterCounter);
21             threads[i].start();
22         }
```

Invokes the code in Display 19.4 where the race condition occurs.

The single instance of the Counter object.

Array of 30,000 threads.

Give each thread a reference to the single Counter object and start each thread.

Race Condition Test Class (3 of 3)

```
23      // Wait for the threads to finish
24      for (i = 0; i < threads.length; i++)
25      {
26          try
27          {
28              threads[i].join(); ← Waits for the thread to complete.
29          }
30          catch (InterruptedException e)
31          {
32              System.out.println(e.getMessage());
33          }
34      }
35      System.out.println("The counter is " + masterCounter.value());
37  }
38 }
```

Sample Dialogue (output will vary)

```
The counter is 0
The counter is 29998
```

Thread Synchronization

- The solution is to make each thread wait, so only one thread can run the code in `increment()` at a time.
- This section of code is called a **critical region**. Java allows you to add the keyword **synchronized** around a critical region to enforce that only one thread can run this code at a time.

Synchronized

- Two solutions:

```
public synchronized void increment()
{
    int local;
    local = counter;
    local++;
    counter = local;
}
```

```
public void increment()
{
    int local;
    synchronized (this)
    {
        local = counter;
        local++;
        counter = local;
    }
}
```

Producer & Consumer

- In this producer & consumer example
 - We have a producer thread that creates a random number.
 - We also have a consumer thread that consumes the random number.

```
package multithread;

public class Producer extends Thread {

    public Producer(){ }

    public void produce(){

        for(int i=0; i<10; i++){
            prodConTest.NUMBER = Math.random()*100;
            System.out.println("Producer: "+prodConTest.NUMBER );
        }

    }

    public void run(){
        produce();
    }

}
```

```
package multithread;

public class Consumer extends Thread {

    public Consumer(){ }

    public void consume(){
        for (int i=10; i>=0; i--){
            System.out.println("Consumer: "+prodConTest.NUMBER);
        }

    }

    public void run(){
        consume();
    }

}
```

Producer & Consumer

- The main method of the producer & consumer creates two threads.
 - One producer thread
 - One consumer thread

```
package multithread;

public class prodConTest {
    static double NUMBER;

    public static void main(String[] args){

        Producer producer = new Producer();
        Consumer consumer = new Consumer();
        producer.start();
        consumer.start();

    }
}
```

```
Consumer: 0.0
Consumer: 82.26335058969953
Producer: 82.26335058969953
Consumer: 82.26335058969953
Producer: 23.827262626261348
Producer: 32.16717390573557
Consumer: 23.827262626261348
Producer: 91.34480443971357
Consumer: 91.34480443971357
Producer: 62.256633263452656
Consumer: 62.256633263452656
Producer: 72.88430578958373
Consumer: 72.88430578958373
Producer: 86.29986539827502
Consumer: 86.29986539827502
Producer: 25.044011441522727
Consumer: 25.044011441522727
Producer: 83.09676926554043
Consumer: 83.09676926554043
Producer: 60.64252656797524
Consumer: 60.64252656797524
```

Something's fishy with our results...

Synchronization

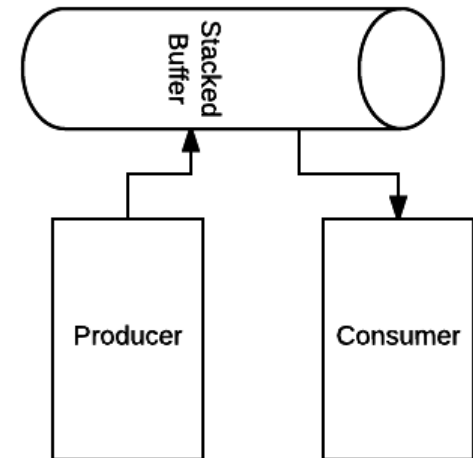
- Synchronization in Java is the capability of controlling the access of a shared resource by multiple threads.
- We must force synchronicity, if needed, which is done by using the keyword **synchronized**
- Why do we use synchronization?
 - To prevent thread interference
 - To prevent consistency problems
 - To prevent data corruption

Synchronization

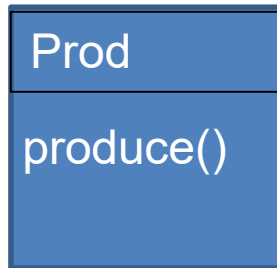
- When a thread begins to execute a synchronized method, it automatically acquires a lock on that object.
- The lock is relinquished, when the method ends.
- Only one thread can have the lock at a particular time
- Therefore, only one thread can execute the synchronized instance method of the same object at a particular time.
 - Synchronization allows only one thread to perform an operation on a object at a time.
 - If multiple threads require an access to an object, synchronization helps in maintaining consistency.

Synchronized Producer & Consumer

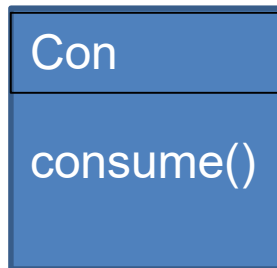
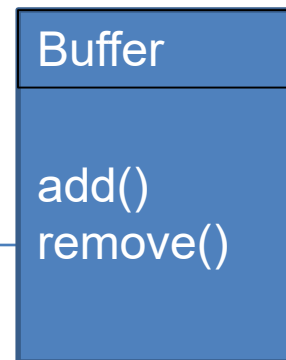
- This version of the producer-consumer has a single stacked buffer.
- The producer adds data to the buffer and waits, if it is full.
- The consumer takes data from the buffer and waits, if empty.
- To fully synchronize this multithreaded program, we need
 - **synchronized**
 - Makes a method synchronous
 - **wait()**
 - Forces the thread to go into the blocked state
 - **notify() / notifyAll()**
 - Forces the thread back into the active state.



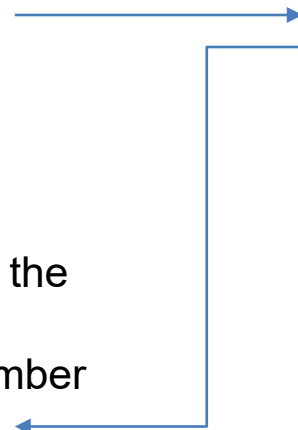
Synchronized Producer & Consumer



- I will produce a number
- Is the buffer full?
- If yes, then I must **wait** for the consumer
- If not, then add the number to the buffer and **notify**.



- I will consume a number
- Is the buffer empty?
- If yes, then I must **wait** for the producer
- If not, then remove the number from the buffer and **notify**.



Synchronized Producer & Consumer

```
package multithread1;

public class Buffer {

    private int loc = 0;
    private double[] data;

    public Buffer(int size){
        data = new double[size];
    }

    public int getSize(){return data.length;}

    public synchronized void add(double toAdd) throws InterruptedException{
        if(loc >= data.length){
            System.out.println("Buffer is full.");
            wait();
        }
        System.out.println("Adding item "+toAdd);
        System.out.flush();
        data[loc++] = toAdd;
        notifyAll();
    }

    public synchronized double remove() throws InterruptedException{
        if (loc <= 0){
            System.out.println("Buffer is empty.");
            wait();
        }
        double hold = data[--loc];
        data[loc] = 0.0;
        System.out.println("Removing item "+hold);
        System.out.flush();
        notifyAll();
        return hold;
    }

    public synchronized String toString(){
        String toReturn = "";
        for(int i=0; i<data.length; i++){
            toReturn += String.format("%2.2f", data[i])+" ";
        }
        return toReturn;
    }
}
```

Synchronized Producer & Consumer

```
package multithread1;

public class Producer extends Thread {

    private int pNum;
    private final Buffer buffer;

    public Producer(Buffer buffer){
        this.buffer = buffer;
    }

    public void produce() throws InterruptedException{
        for(int i=0; i<buffer.getSize(); i++){
            buffer.add(Math.random()*100);
        }
    }

    public void run(){
        try {
            produce();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
package multithread1;

public class Consumer extends Thread {

    private int pNum;
    private Buffer buffer;

    public Consumer(Buffer buffer){
        this.buffer = buffer;
    }

    public void consume() throws InterruptedException{
        for (int i=buffer.getSize(); i>=0; i--){
            buffer.remove();
        }
    }

    public void run(){
        try {
            consume();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Synchronized Producer & Consumer

```
package multithread2;  
  
public class prodConTest {  
  
    public static void main(String[] args){  
  
        Buffer buff = new Buffer(10);  
  
        Producer producer = new Producer(buff);  
        Consumer consumer = new Consumer(buff);  
        producer.start();  
        consumer.start();  
  
    }  
}
```

```
Buffer is empty.  
Adding item 77.8472326959853  
Removing item 77.8472326959853  
Buffer is empty.  
Adding item 26.92239748059311  
Adding item 20.580438594134364  
Removing item 20.580438594134364  
Removing item 26.92239748059311  
Buffer is empty.  
Adding item 80.29504857263319  
Adding item 40.42235604592893  
Adding item 1.8605782544117155  
Adding item 8.794070081571315  
Adding item 12.065152909808718  
Adding item 79.15242043184627  
Adding item 57.59898790728746  
Removing item 57.59898790728746  
Removing item 79.15242043184627  
Removing item 12.065152909808718  
Removing item 8.794070081571315  
Removing item 1.8605782544117155  
Removing item 40.42235604592893  
Removing item 80.29504857263319  
Buffer is empty.
```