

# CITS3213: CONCURRENT PROGRAMMING

## Concurrency in Java

Java supports concurrent programming via the following main features.

- Dynamically created threads.
- Mutual exclusion via a variant of monitors
  - Each object has a lock
  - Only one thread can be in the “synchronized” methods of each object
  - Unsynchronized methods are also allowed
  - No condition variables
  - Instead use `wait()` and `notifyAll()` to suspend and resume threads, with woken threads repeatedly calling `wait()` until their condition is satisfied

## Creating Threads: Subclassing Thread

```
public class Example1{

    public static void main(String args[]) {
        MyThread thread1=new MyThread("thread1: ");
        MyThread thread2=new MyThread("thread2: ");
        thread1.start();
        thread2.start();
    }
}

class MyThread extends Thread {
    static String message[] = {"Java", "is", "..."};
    public MyThread(String id){
        super(id);
    }
    public void run() {
        String name = getName();
        for(int i=0;i<message.length;i++){
            randomWait();
            System.out.println(name+message[i]);
        }
    }
    void randomWait(){
        try{
            sleep((int)(1000*Math.random()));
        } catch (InterruptedException x) {}
    }
}
```

## Creating Threads: Implementing Runnable

```
public class Example4 {

    public static void main(String args[]) {
        Thread thread1=new Thread(new MyThread("thread1: "));
        Thread thread2=new Thread(new MyThread("thread2: "));
        thread1.start();
        thread2.start();
    }
}

class MyThread implements Runnable {
    static String message[] = {"Java", "is", "..."};
    String name;
    public MyThread(String id){
        name=id;
    }
    public void run() {
        for(int i=0;i<message.length;i++){
            randomWait();
            System.out.println(name+message[i]);
        }
    }
    void randomWait(){
        try{
            Thread.sleep((int)(1000*Math.random()));
        } catch (InterruptedException x) { }
    }
}
```

## Synchronization: methods

- So far our threads are allowed to both be running in the same object
- In many cases we need some kind of mutual exclusion
- Java uses a variant of monitors
- Only one thread can be in the “synchronized” methods of each object

```
public class Counter {
    private int count=0;

    public synchronized void inc() {
        count = count+1;
    }

    public synchronized void dec() {
        count = count-1;
    }

    public int getCount() { // not synchronized
        return count;
    }
}
```

## Synchronization: `wait` and `notifyAll`

- When a thread holding the lock on an object needs to wait for some condition, it calls `wait()`
- There are no condition variables, so the usual pattern is to call `notifyAll()` whenever a change occurs that might be what a thread is waiting for.
- Then, each waiting thread should have a loop that repeatedly waits, checking a condition each time. (See following example.)
- Just like standard monitors, threads release mutual exclusion (on all objects) when they wait.
- Only threads that have mutual exclusion on an object may wait or notify on that object.
- The `notify()` function is NOT recommended: no guarantee that the thread is scheduled immediately.

## Producer-Consumer in Java

```
public class Buffer {
    private static final int N = 5;
    private int[] B = new int[N];
    private int InPtr=0, OutPtr=0;
    private int Count=0;

    public synchronized void append(int value) {
        while (Count==N) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        B[InPtr] = value;
        InPtr = (InPtr+1) % N;
        Count = Count+1;
        notifyAll();
    }

    // class Buffer continued on page 2.
```

## Producer-Consumer in Java

```
// class Buffer continued from page 1.

public synchronized int take () {
    while (Count==0) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    int I = B[OutPtr];
    OutPtr = (OutPtr+1) % N;
    Count = Count-1;
    notifyAll();
    return I;
}
}
```

## Producer-Consumer in Java

```
public class Producer extends Thread {
    private Buffer buffer;

    public Producer(Buffer c) {
        buffer = c;
    }

    public void run() {
        for (int i = 0; i < 100; i++) {
            buffer.append(i);
            System.out.println("Produced:" +i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

## Producer-Consumer in Java

```
public class Consumer extends Thread {
    private Buffer buffer;

    public Consumer(Buffer c) {
        buffer = c;
    }

    public void run() {
        for (int i = 0; i < 100; i++) {
            int v = buffer.take();
            System.out.println("Consumed: " + v);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

## Producer-Consumer in Java

```
public class ProducerConsumerTest {  
    public static void main(String[] args) {  
        Buffer buf = new Buffer();  
        Producer p1 = new Producer(buf);  
        Consumer c1 = new Consumer(buf);  
  
        p1.start();  
        c1.start();  
    }  
}
```