Multi Tasking
Shared resources
Lecture goal:

- Learn about the basics of concurrency in computer-programs
  - Especially in Java programs

Overview:

- Execution context
- Sharing computation resources
- Threads in Java
  - Threads?
  - The Life cycle of a thread
  - Thread priority
  - Sharing resources
    - Synchronizing
Concurrency...

- The question is how to fix so several things to happen simultaneously on one computer
- Several users on one computer
- One user running several programs on one computer
  - Mail client
  - Excel
- Operating systems
- Foreground/background

Concurrency...

- Several CPUs mounted in one Computer
  - Supercomputers
    - Thousands of CPUs
  - Desktop computers
    - 1-4 CPUs

- Sharing one CPU
  - Processes/Threads
  - Time slices
  - Priority queues
Concurrency - Process...

✓ A Program in execution
  o Task
  o Job

✓ A process has:
  o a virtual address space
  o program counter
  o executable code
  o data
  o a base priority
  o execution stack
  o registers
Concurrency - Threads...

✓ A thread is a single sequential flow of control within a program
  o Lightweight processes
  o Execution context
✓ All threads of a process share its virtual address space and system resources
✓ Threads has:
  o Registers
  o Program counter
  o Execution Stack
Concurrency...

A Program

Two Threads

A Thread
Threads - how to in Java

✓ Subclassing the thread class and override the run-method
  o Works for stand alone apps
✓ Implementing the Runnable Interface
Threads - subclassing 1/3

1. public class SimpleThread extends Thread {
2.     public SimpleThread(String str) {
3.         super(str);
4.     }
5.     public void run() {
6.         for (int i = 0; i < 5; i++) {
7.             System.out.println(i + " " + getName());
8.             try {
9.                 sleep((long)(Math.random() * 1000));
10.            } catch (InterruptedException e) {}  
11.         }
12.         System.out.println("DONE! " + getName());
13.     }
14.}
Threads - subclassing 2/3

1. public class TwoThreadsDemo {
2.     public static void main (String[] args) {
3.         new SimpleThread("Jamaica").start();
4.         new SimpleThread("Fiji").start();
5.     }
6. }

Threads - subclassing 3/3

0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
4 Fiji
3 Jamaica
DONE! Fiji
4 Jamaica
DONE! Jamaica
Threads - Impl. Runnable 1

```java
public class threadSample2 implements Runnable {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " Thread.currentThread().getName());
            try {
                Thread.currentThread().sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + Thread.currentThread().getName());
    }
    public static void main(String args[]) {
        (new Thread(new threadSample2())).start();
        (new Thread(new threadSample2())).start();
        (new Thread(new threadSample2())).start();
    }
}
```
Anonymous Implementation of Runnable

```java
public static void main (String[] args)
{
    new Thread(new Runnable()
    {
        public void run()
        {
            doThings();
        }
    }).start();
}
```
Threads - Rule of Thumb

✓ Subclassing or Implementing the Runnable

  Why??
Threads - the life cycle

✓ Creating
✓ Starting
✓ Not Runnable
✓ Stopping

Diagram:
- New Thread
- Runnable
- Not Runnable
- Dead

States:
- Running
- Yield
- The run method terminates
Threads - Creating a Thread

```java
public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
}
```

- Empty thread object
  - No resources allocated
  - Only start is possible
Threads - Starting a Thread

```java
public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
}
```

- Creates system resources
- Schedules the thread to run
- Calls the thread’s run-method
- Returns “running”
- All threads can’t run simultaneously
Threads - running

public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // the VM doesn't want us to sleep anymore,
            // so get back to work
        }
    }
}

✓ May be waiting for its turn in a pqueue
Threads - Making a Thread Not Runnable

```java
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // the VM doesn't want us to sleep anymore,
            // so get back to work
        }
    }
}
```

✔ sleep(x)
  o Waiting x milliseconds
✔ wait()
  o notify() or notifyAll()
✔ IO-blocked
Threads - Stopping a Thread 1/2

public void run() {
    int i = 0;
    while (i < 100) {
        i++;
        System.out.println("i = " + i);
    }
}

✓ The run method terminates
Threads - Stopping a Thread 2/2

```java
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        ...
    }
}

public void stop() {
    // applets' stop method
    clockThread = null;
}
```

![Diagram showing the life cycle of a thread, from new to running, yield, and stop](image)
Thread Priority - sharing CPUs

✓ Java supports, a fixed priority scheduling
  o Relative priority
    • Highest first
    • Round robin
      – Equal priority
  o Preemptive
  o Selfish threads but
    • Avoids starvation
  o Time slicing not granted

✓ A thread:
  o Inherits its priority
  o Modify
    • setpriority(…)
      – MIN_PRIORITY
      – MAX_PRIORITY
  o Executes until
    • Higher priority runnable
    • Its run method exits
    • Time-slice ends
    • It yields
      – only to threads at the same priority level
Threads - shared resources

✓ Producer/consumer situation
  o A consumer can’t consume from an empty resource
  o A producer can’t produce into a full resource
  o Critical sections
    • Code segments that access the same object

✓ Examples
  o Event queues in OS
  o Distributed data bases

✓ Java can block critical sections
  o The synchronized keyword

✓ Fairness - must avoid
  o Starvation
  o Deadlock
Threads- locking objects

- Java uses monitors
- Semaphores is an alternative possible to implement with monitors
Threads- locking objects example (producer)

public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer "+this.number
                                + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) {
            }
        }
    }
}
Threads- locking objects example (consumer)

public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer "+ this.number
                                + " got: " + value);
        }
    }
}
Threads- locking objects example (main)

```java
public class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
}
```
Threads- locking objects example

```java
public class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        // CubbyHole locked by the Consumer
        if (available == true) {///<DEADLOCK
            available = false;
            return contents;
        }
        // CubbyHole unlocked by the Consumer
    }

    public synchronized void put(int value) {
        // CubbyHole locked by the Producer
        if (available == false) {///<DEADLOCK
            available = true;
            contents = value;
        }
        // CubbyHole unlocked by the Producer
    }
}
```
Threads - notifying other threads

```java
public synchronized int get() {
    while (available == false) {
        try {
            // wait for Producer to put value
            wait();
        } catch (InterruptedException e) { }
    }
    available = false;
    // notify Producer that value has been retrieved
    notifyAll();
    return contents;
}

public synchronized void put(int value) {
    while (available == true) {
        try {
            // wait for Consumer to get value
            wait();
        } catch (InterruptedException e) { }
    }
    contents = value;
    available = true;
    // notify Consumer that value has been set
    notifyAll();
}
```
Threads - notifying & waiting

✓ sleep(long time)
  - A sleeping thread can’t be awakened prematurely
  - Must finish the sleeping period before work again!

✓ wait()
  - Waits for notification

✓ wait(long time)
  - Waits for notification or the time period has elapsed
Threads - deadlock & starvation

- **Deadlock** - every thread waiting for someone else to release something
  - Hard to detect
  - Better to avoid

- **Starvation** - someone stands forever in the queue without doing any work
  - Quite easy to detect
  - Java avoid it
    - Round robin
    - Not fully preemptive
Threads- avoiding deadlocks

- Impose the ordering of the condition variables
- Rearrange the problem
  - “Numbering the chopsticks”
    - 1..5 and take the lowest stick first and not
    - The right one first
  - Using tickets for the table
    - 4-valued semaphore for the table