Java Thread Mode

- Thread: A *sequence* of execution in a program.
- Every java program contains at least one thread.
- The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.
Single Thread Example

Control Flow:
1. JVM creates an initial thread of control
2. The thread enters main()
3. Executes the operations
4. Exit from main()
5. Terminate

Example:
Public class HelloWorld{
    Public static void main (String[] args){
        System.out.println(“Hello World”); }
}
Threads in Java

• Thread Class
  – Every thread of control in a JVM is associated with a *Thread Object*.

• Thread Priority
  – Threads with higher priority are executed first.
  – Multi-level ready queue, Round Robin scheduling for each level.
  – New thread’s priority is initially set to be the same priority of the creating thread.
Thread Creation

• Two ways to construct threads
  – Explicitly create a new class that is derived from the Thread class (java.lang.thread).
  – Implement the Runnable interface (java.lang.runnable).
Subclass of the Thread Class

- Override the `run()` method of the Thread class.

```
Class WorkerThread1 extends Thread{
    Public void run() {
        System.out.println(“I am a worker thread.”);
    }
}
```
Subclass of the Thread Class (Cont.)

- Create an object
  
  `WorkerThread1 w1 = new WorkerThread1();`

- Call `start()` method for the new object to
  
  1. allocate memory and initialize a new thread in JVM;
  2. call the `run()` method (implicitly) making the thread eligible to be run by the JVM.

  `w1.start();`
Subclass of the Thread Class  
(Cont.)

Main class (Thread) :

```java
public class First {
    public static void main(String args[]) {
        WorkerThread1 w1 = new WorkerThread1();
        w1.start();
        System.out.println("I am the main Thread");
    }
}
```
Subclass of the Thread Class (Cont.)

• Results
  – Two threads are created by the JVM.
  – First: the thread that starts execution at the `main()` method;
  – Second: the \( w1 \) thread begins execution in its `run()` method.
Implement Runnable Interface

• Define a class that implements the `Runnable` interface:

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

• The class must define a `run()` method.

```java
Class WorkerThread2 implements Runnable {
    public void run() {
        System.out.println("I am a worker thread.");
    }
}
```
Implement Runnable Interface (Cont.)

- Thread creation
  
  Public class Second {

  public static void main(String args[]) {

  Runnable w2 = new WorkerThread2();

  Thread td = new Thread(w2);

  td.start();

  System.out.println(“I am the main Thread.”);
  
  }

  }

}
Java Thread States

- **New**: when an object for the thread is created.

- **Runnable**: when a thread’s `run()` method is invoked, the thread moves from New to Runnable.
  - The thread in eligible to be run by the JVM.
  - A running thread is still in the Runnable state.
Java Thread States (Cont.)

- **Blocked**: when the thread performs a blocking statement, e.g., doing I/O; or if it invokes certain Java Thread methods, s.t., `sleep()`.
  - Thread in such state cannot be executed by the JVM scheduler.

- **Dead**: when the thread’s `run()` method terminates.
  - Cannot be restart and is eligible for garbage collection.
Java Thread States (Cont.)

Diagram:
- New
  - start()
  - New
- Runnable
  - sleep()
  - I/O
  - resume()
- Blocked
  - Runnable
- Dead
  - stop()
Java Thread Scheduling

• JVM schedules threads using preemptive, priority-based scheduling algorithm.
  – Always schedule the thread with the highest priority.
  – Using FIFO for the threads having the highest priority.
When to Schedule?

• One of the followings events occur:
  1. The currently running thread exists the *Runnable* state:
     • Blocking for I/O
     • Existing run() method
     • Having sleep(), yield() invoked (relinquishes control of CPU, allow another thread of equal priority to run)
  2. A thread with higher priority than the currently running thread enters the *Runnable* state.
Timing Slicing

- Implementation dependent (JVM).
- May yield to give up CPU if there is no time slicing...cooperative multitasking.

```java
public void run(){
    while(true) {
        //perform a CPU-intensive task
        //now yield control of the CPU
        Thread.yield();
    }
}
```
Thread Priorities

- Threads are given a default priority when they are created and unless they are changed explicitly by the program.
- JVM does not dynamically alter priorities throughout the threads lifetime.

```java
public class HighThread extends Thread{
    public void run(){
        this.setPriority(Thread.NORM_PRIORITY+1);
        //remainder of run() method
    }
}
```
Lab 2

• Purpose: multiple robots coordinate to eat goals.
  – Three colors for robots (red, green, blue) and four for goals (red, green, blue, black).
  – Each robot can eat goals with either its color or black color.
  – Run until all goals are eaten.
What robots do?

• Make decision to take a next step
  – Four directions to choose, choose the position with the minimum non-negative value from distance matrix. \((\infty \text{ means it is an obstacle})\)
  – Move to that place. If there is a goal, eat it and signal that the goal has gone (need to reset the distance values).
  – If there is no goal to eat, the robots become dead and an obstacle for other color robots, also need to signal when robots die.
What need to be concerned about?

• Two robots cannot eat a same goal.
• Two robots cannot occupy the same cell(position) in the grid.
• Robots share the same map information.
• The group robots with the same color share one distance matrix to get/update the distance values for the goals.
## Distance Function

![Image of a grid with different symbols and numbers]

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>R</td>
<td>8</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(b)
```
Map_Data{
    Grid Info., int Grid[][];
    Abstract_Robot Robots[];
    Goal Goals;
    Distance Dists[];
    Semaphore sem;}

Distance{
    char cColor; //group color
    Map_Data myGrid; //referred map
    int Dist[][]; //dis. Matrix
    Grid size;
    Semaphore sem;
    void initDist(); //create the dis. Matrix
    int fromNeighbor(int, int);
    Void SetDist();}

Map{
    Map_Data md = new Map_Data;
    parse the map file;
    show the map;}

Robot extends Abstract_Robot{
    while{has eatable goals}
    { make decision for the next step;
        move;
        signal if eating a goal or becoming a dead robot
        (need to update distance matrix)}

Abstract_Robot{
    position;
    Map_Data myGrid;
    Distance my Dist;}
```