Concurrency in Java

MD. ANISUR RAHMAN

Concurrency

Two or more sequences of events occur "in parallel"

- Multiprogramming
  - Single processor runs several programs at the same time
  - Each program proceeds sequentially
  - Actions of one program may occur between two steps of another
- Multiprocessors
  - Two or more processors
  - Programs on one processor communicate with programs on another
  - Actions may happen simultaneously

Process: sequential program running on a processor

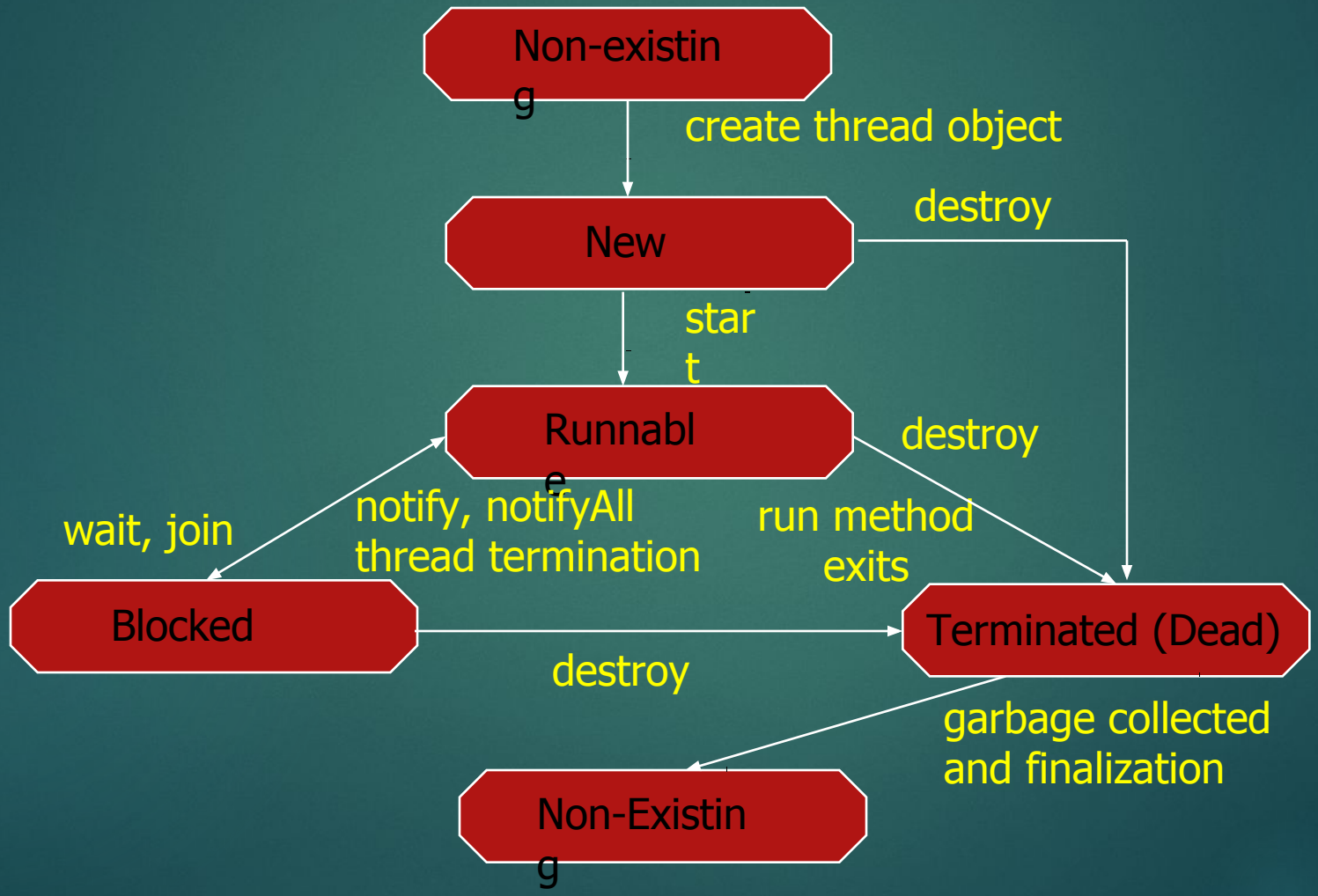
The Promise of Concurrency

- Speed
  - If a task takes time t on one processor, shouldn't it take time t/n on n processors?
- Availability
  - If one process is busy, another may be ready to help
- Distribution
  - Processors in different locations can collaborate to solve a problem or work together
- Humans do it so why can't computers?
  - Vision, cognition appear to be highly parallel activities

Java Threads

- Thread
  - Set of instructions to be executed one at a time, in a specified order
  - Special Thread class is part of the core language
- Methods of class Thread
  - start : method called to spawn a new thread
    - Causes JVM to call run() method on object
  - suspend : freeze execution (requires context switch)
  - interrupt : freeze and throw exception to thread
  - stop : forcibly cause thread to halt

States of a Java Thread



Creating and Starting Threads

- Creating a thread in Java is done like this:

```
Thread thread = new Thread();
```

- To start the Java thread

```
thread.start();
```

- This example doesn't specify any code for the thread to execute.
- Two ways to specify that:
  1. create a subclass of Thread and override the run() method.
  2. pass an object that implements *Runnable* to the *Thread* constructor

Thread Subclass

- The run() method is what is executed by the thread after you call start()

```
public class MyThread extends Thread {

   public void run(){
      System.out.println("MyThread running");
   }
}
```

- To create and start the above thread0

```
MyThread myThread = new MyThread();
  myTread.start();
```

Thread Subclass

- The start() call will return as soon as the thread is started.
- It will not wait until the run() method is done.
- The run() method will execute as if executed by a different CPU.
- When the run() method executes it will print out the text "MyThread running".

Runnable Interface Implementation

- create class that implements java.lang.Runnable

```
public class MyRunnable implements Runnable {

   public void run(){
      System.out.println("MyRunnable running");
   }
}
```

- To have the run() method executed by a thread, pass an instance of MyRunnable to a Thread in its constructor.

```
Thread thread = new Thread(new MyRunnable());
  thread.start();
```

Thread Names

- When you create a Java thread you can give it a name.
- The name can help you distinguish different threads from each other.
- For instance, if multiple threads write to System.out it can be handy to see which thread wrote the text.

```
Thread thread = new Thread("New Thread") {
    public void run(){
      System.out.println("run by: " + getName());
    }
  };

  thread.start();
  System.out.println(thread.getName());
```

```
MyRunnable runnable = new MyRunnable();
  Thread thread = new Thread(runnable, "New Thread");

  thread.start();
  System.out.println(thread.getName());
```

Thread.currentThread()

- The Thread.currentThread() method returns a reference to the Thread instance executing currentThread() .
- This way you can get access to the Java Thread object representing the thread executing a given block of code.
- Once you have a reference to the Thread object, you can call methods on it.

```
Thread thread = Thread.currentThread();

String threadName = Thread.currentThread().getName();
```

Java Thread Example

```
public class ThreadExample {
  public static void main(String[] args){
    System.out.println(Thread.currentThread().getName());
    for(int i=0; i<10; i++){
      new Thread("" + i){
        public void run(){
          System.out.println("Thread: " + getName() + " running");
        }
      }.start();
    } } }
```

- Even if the threads are started in sequence (1, 2, 3 etc.) they may not execute sequentially meaning thread 1 may not be the first thread to write its name to System.out.
- This is because the threads are in principle executing in parallel and not sequentially.
- The JVM and/or operating system determines the order in which the threads are executed. This order does not have to be the same order in which they were started.

Race Conditions and Critical Sections

- A race condition is a special condition that may occur inside a critical section
- A critical section is a section of code that
  - is executed by multiple threads and
  - where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.
- When the result of multiple threads executing a critical section may differ depending on the sequence in which the threads execute, the critical section is said to contain a race condition.

Critical Sections

- problems arise when multiple threads access the same resources. For instance the same memory (variables, arrays, or objects), systems (databases, web services etc.) or files.
- problems only arise if one or more of the threads write to these resources.

Critical Section example

```
public class Counter {

   protected long count = 0;

   public void add(long value){
       this.count = this.count + value;
   }
}
```

- Imagine if two threads, A and B, are executing the add method on the same instance of the Counter class.
- The code in the add() method is not executed as a single atomic instruction by the Java virtual machine.
- Rather it is executed as a set of smaller instructions
  1. Read this.count from memory into register.
  2. Add value to register.
  3. Write register to memory.

Critical Section example

- Observe what happens with the following mixed execution of threads A and B:

```
this.count = 0;

A:  Reads this.count into a register (0)
B:  Reads this.count into a register (0)
B:  Adds value 2 to register
B:  Writes register value (2) back to memory. this.count now equals 2
A:  Adds value 3 to register
A:  Writes register value (3) back to memory. this.count now equals 3
```

- The two threads wanted to add the values 2 and 3 to the counter. Thus the value should have been 5 after the two threads complete execution. However, since the execution of the two threads is interleaved, the result ends up being different.

Preventing Race Conditions

- To prevent race conditions from occurring you must make sure that the critical section is executed as an atomic instruction.
- Race conditions can be avoided by proper thread synchronization in critical sections.
- Thread synchronization can be achieved using a synchronized block of Java code.

```
public class TwoSums {
    private int sum1 = 0;
    private int sum2 = 0;
    public void add(int val1, int val2){
        synchronized(this){
            this.sum1 += val1;
        }
        synchronized(this){
            this.sum2 += val2;
        }
    }
}
```