Concurrent Programming

CSE 216 : Programming Abstractions

Department of Computer Science Stony Brook University

Dr. Ritwik Banerjee

Concurrency

A. Single core machine without any concurrency
   - runs only one application at a time (no more coding *while* listening to music)

B. Single application with concurrency
   - read digital audio off the network *and*
   - decompress it *and*
   - manage playback *while*
   - updating display.

- We will be using Java to study concurrent programs, since it provides a very strong framework for multithreaded programming.
  - Based on its **`java.util.concurrent`** library.

© 2019 Ritwik Banerjee

Processes & Threads

- There are two fundamental units of execution: **processes** and **threads**.
- Processes are more of a concern at the OS-level. They are not within the scope of our syllabus.
- In simpler (non-OS) programs, concurrent programming *mostly* deals with threads.
- A computer system normally has many active processes and threads. This is the case even in a single-core computer.
  - In a single core system, only one thread executes at any given moment.
  - The single core is shared among processes and threads through *time slicing*, a feature of the operating system's implementation.

© 2019 Ritwik Banerjee

Processes

- A process has its own self-contained execution environment.
- Generally, a process will have a complete (and typically private) set of basic runtime resources. In particular, a process has its *own memory space*.
  - Keep in mind that a "process" is not the same as a "program" or an "application".
  - A single application may actually be a whole set of processes working together.
- In an operating system, processes often need to communicate with each other. This is done through an *inter-process communication (IPC) protocol*.
  - This protocol is OS-specific.
- In Java, most virtual machines run as a single process.
- A concurrent Java application will typically work with multiple threads.
  - But there is the provision to use a **`ProcessBuilder`** object for situations that call for spawning multiple processes.

© 2019 Ritwik Banerjee

Threads

- We can think of a thread as a *lightweight* process.
  - Creating a new thread requires fewer resources than creating a new process.
- A thread exists *within* a process, and every process has at least one thread.
- Unlike processes, threads share the process resources (memory, open files, database access, etc.).
- Using multiple threads can increase efficiency by working on several things in parallel, but this also introduces additional communication issues.
- Java programs:
  - You start with a single thread called the **main** thread (and there are some additional `System` threads responsible for memory management).
  - The **`main`** thread can then create additional threads to start off a concurrent Java program.

© 2019 Ritwik Banerjee

A/Synchronous tasks

I. Synchronous
   - *Synchronously* means "using the same clock". So, when two instructions are **synchronous**, they use the same clock and one of the two *must* happen after the other.
   - In other words, the program *must* wait for the first instruction to finish, before it can move on the other.

II. Asynchronous
   - *Asynchronously* means *not* "using the same clock". That is, two **asynchronous** instructions are not concerned with being in step with each other.
   - In other words, the program can move on from a task $\mathbf{t_1}$ to another task $\mathbf{t_2}$ before $\mathbf{t_1}$ finishes.

© 2019 Ritwik Banerjee

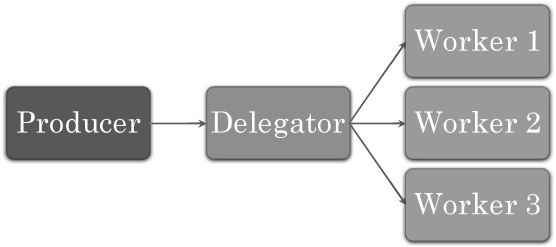
Concurrency or parallelism?

- Sometimes, these get used interchangeably, but they mean two different things. The difference may seem subtle, but is important:
- **Concurrency** is about handling *asynchronous* tasks, sometimes with *non-deterministic* behavior.
  - This can happen with reactive programming: a user may or may not click a UI button; if the button is clicked, the window must respond in some way even if it was busy displaying something else.
- **Parallelism** is about the *asymptotic efficiency* of a program with *deterministic* behavior.
  - Deals with the *dependencies* among the subtasks of a deterministic task.
  - Parallelism is an abstraction, and concurrency is used to implement it.
  - Parallelism does not need to be aware of concurrency, even if the engineers who build parallel computing systems must deal with concurrency.

© 2019 Ritwik Banerjee

Concurrency model: asynchronous workers

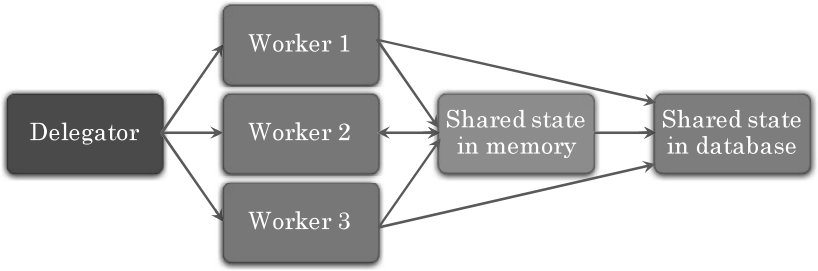
- There are various models of concurrency, with each model explaining how tasks are distributed among threads (from a general outside perspective).
- The first model is that of **asynchronous workers**.
  - A producer creates tasks somewhere, and a delegator distributes and assigns tasks to workers, who work asynchronously without depending on each other.
  - Each worker runs on a different thread



© 2019 Ritwik Banerjee

Concurrency model: asynchronous workers

- This is a conceptually simple model, but problems crop up because when you share a task among workers, the worker will typically depend on some common *shared* data.
- The workers (i.e., threads) that need access to shared data need to ensure that the changes made by one is visible to the others.
  - e.g., changes are pushed to main memory and not in the CPU cache
- The workers need to avoid shared state concurrency problems like **deadlock**, **race conditions**, etc.



© 2019 Ritwik Banerjee

Problems with asynchronous models

- Threads can interfere with one another – e.g., one thread is partly done with a computation when another thread decides to access shared data.

```
class Counter {
    private int c = 0;
    public void increment() { c++; }
    public void decrement() { c--; }
    public int value() { return c; }
}
```

- Even simple operations like **c++** consist of multiple smaller steps: (1) retrieve current value, (2) increment it, and (3) store the new value back into the memory location.
- In other words, it is not an atomic operation.

© 2019 Ritwik Banerjee

Atomicity and critical regions

- An **atomic operation** is something that cannot be broken down into smaller sub-operations. It either runs completely, or not at all.
- Any part of a program that need to be atomic (for concurrency to work properly) is called a **critical** area or region of the code.

© 2019 Ritwik Banerjee

Problems with asynchronous models

```
class Counter {
    private int c = 0;
    public void increment() { c++; }
    public void decrement() { c--; }
    public int value() { return c; }
}
```

- Suppose this program is run on two threads, **A** and **B**, and they both call **`increment()`** at around the same time.
- Due to lack of atomicity, the actual sequence of operations may be something like
  1) **A** retrieves **c**.
  2) **B** retrieves **c**.
  3) **A** **`increment()`**s **c** with the result being **1**.
  4) **B** **`decrement()`**s **c** with the result being **-1**.
  5) **A** stores result in **c**, so now, **`c = 1`**.
  6) **B** stores result in **c**, so now, **`c = -1`**.

Thread interference
leads to the result of A's work being overwritten by B, and lost.

© 2019 Ritwik Banerjee

Problems with asynchronous models

- A **memory inconsistency error** occurs when different threads have inconsistent views of what *should* be the same data.
- These can be avoided by clearly establishing when one event **happens before** another, thereby guaranteeing that data written by one operation is visible to the other.
- Again, suppose this is run on two threads, **`A`** and **`B`**, and
  1) **`A`** increments **`c`** using the unary operation, and
  2) **`B`** executes **`System.`*`out`*`.println(c);`**
- If **`A`**'s operation does not "happen before" that of **`B`** – and there's guarantee that it does – the printed value may still be zero.
- We will later see how to implement code to create specific "happens before" relations between two operations.

© 2019 Ritwik Banerjee

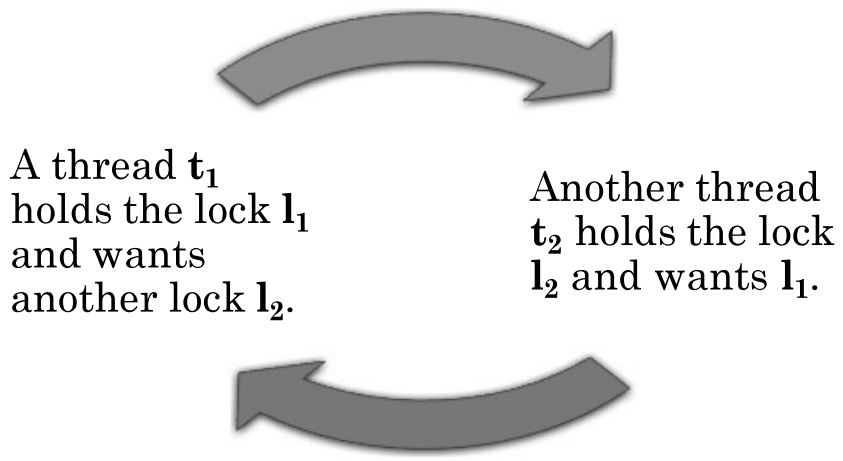
Problems with asynchronous models

- A **race condition** is a situation that occurs when a program attempts to perform multiple operations at the same time, but because of the nature of the hardware or software being used, the operations must be done in the proper sequence to be done correctly.
- In order to avoid race conditions, systems often use the concept of a **lock** (also called **mutex**, for ***mut***ual ***ex***clusion), which gives one thread exclusive access to the locked resource.
- But locks must be used carefully! Otherwise, we may adversely affect a concurrent program's ability to *actually* work concurrently, correctly, and in a timely manner – a property called **liveness**.
- The most important enemies of liveness are **deadlock**, **starvation**, and **livelock**.

© 2019 Ritwik Banerjee

Problems with asynchronous models

- A **deadlock** is a state in which each member of a group is waiting for another member (including itself) to take action, such as sending a message or releasing a lock.



© 2019 Ritwik Banerjee

Problems with asynchronous models

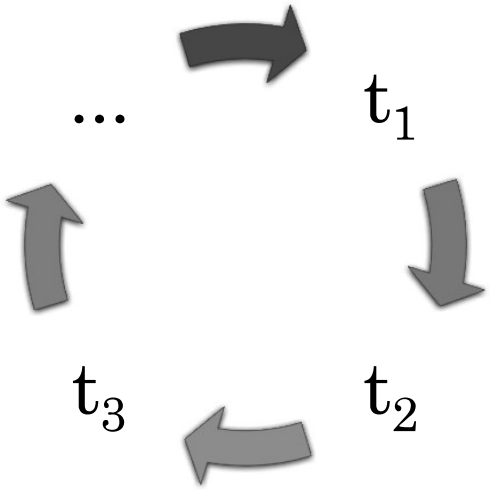
- **Starvation** is a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.
- This happens when a shared resource is made unavailable for long periods of time by "greedy" threads. E.g., if a thread *locks* a resource for a long time, denying access to other threads.

© 2019 Ritwik Banerjee

Problems with asynchronous models

- **Livelock** is a special type of starvation caused by *over*-communication, where the threads are so busy conveying messages to each other that they are not doing their actual job!
- Think of two overly polite people walking in opposite directions in a corridor, and when they approach each other
  - A moves to his left to let B pass while B moves to her right to let A pass.
  - Seeing that they are still blocking each other, A moves to his right, while B moves to her left.
  - They're still blocking each other, so it goes on ... neither person willing to become less polite.
- The responses could form a communication chain.
- Or even an infinite cycle.

© 2019 Ritwik Banerjee

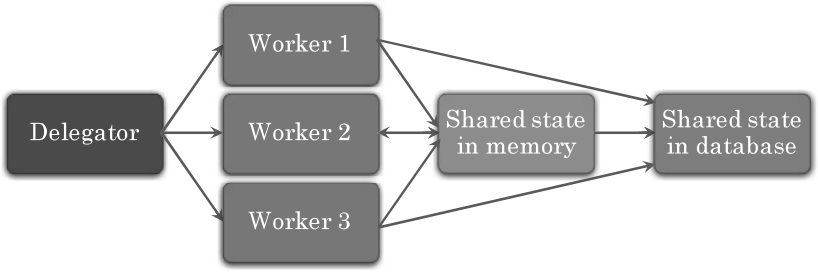


A livelock can form a long communication chain, or even a cycle.

© 2019 Ritwik Banerjee

Concurrency model: asynchronous workers

- Concurrency may also be partly lost when the threads start waiting for each other to access the shared data.
- Many concurrent data structures are **blocking**, i.e., only one thread can access the data at any given time.
- This leads to a struggle for access to the data. Too much of this will effectively lead to serial execution of the part of the code that accesses the shared data.
- In this picture, all workers need access to the shared state, so serial execution will effectively mean that all concurrent is lost, and we might as well forget about multithreading.



© 2019 Ritwik Banerjee

Concurrency model: asynchronous workers

- There are two ways around this problem: *non-blocking algorithms and data structures* (which are *really* difficult to implement), or *persistent data structures*.
- A **persistent data structure** is one that preserves its own immediately previous version upon modification.
- So, if threads $\mathbf{t_1}$ and $\mathbf{t_2}$ refer to the same persistent object, and $\mathbf{t_1}$ changes something, then $\mathbf{t_1}$ gets a reference to the new version while $\mathbf{t_2}$ sees the older one. Neither thread will see the data structure in an internally inconsistent state.

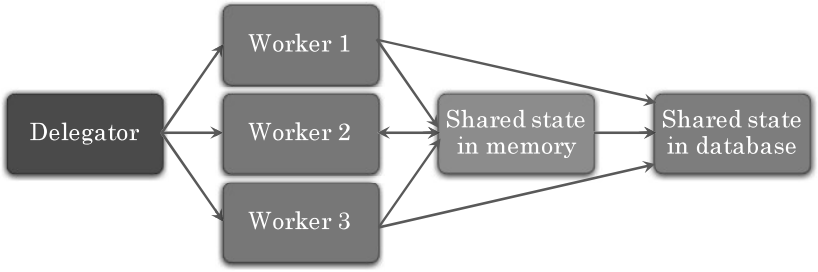
Exercise: think of a persistent list where only one of the multiple thread adds new elements to the head of the list.

- What data structure would you use to implement this list?
- If you didn't worry about multithreading and persistence, would you have gone for another option? would that option be more efficient for many list operations?

© 2019 Ritwik Banerjee

Concurrency model: asynchronous workers

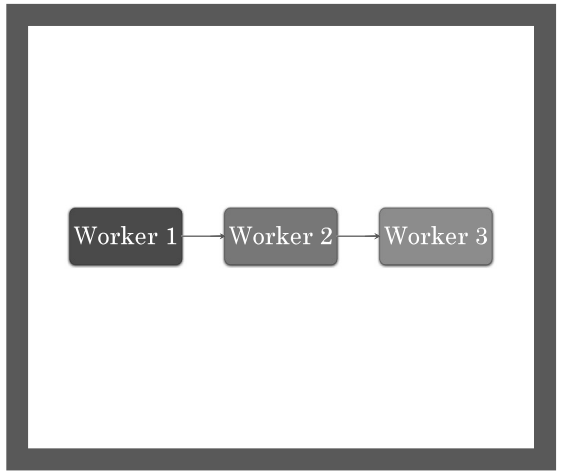
- In a picture like this, "concurrency" means that the order in which tasks (or even statements within the tasks) are completed is *non-deterministic*.
- This makes it difficult to establish if one execution **happens before** another.
- And without this "happens before" ordering among events, it becomes difficult to reason about the system's behavior.

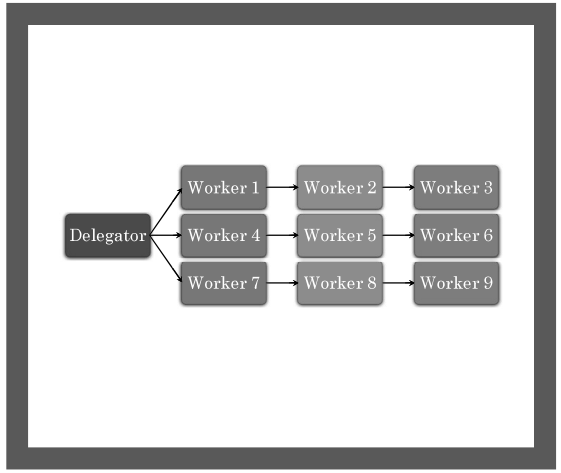


© 2019 Ritwik Banerjee

Concurrency model: assembly line

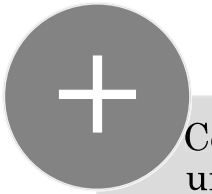
- The workers are arranged like an assembly line in a factory, with each worker doing only a part of the entire work before it passing it along to the next worker in the line.
- In reality, though, multiple assembly lines are often used.
- These systems can get quite complex, with situations like a thread $\mathbf{t_1}$ from one line depending on threads $\mathbf{t_2}$ and $\mathbf{t_3}$ from other lines.





© 2019 Ritwik Banerjee

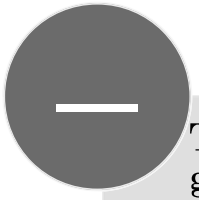
Concurrency model: assembly line



Conforms better with underlying hardware since each assembly line works like a single-threaded system.

Workers can keep in memory the data they need to work with, making the system more efficient (I/O operations are typically very slow).

Sequential ordering of subtasks is usually possible.



Tracing the code runtime can get very difficult because a single task is spread across multiple workers (i.e., thread instances).

This leads to tracing through multiple function callbacks and callback handlers.

Unlike this, with the asynchronous workers model, a programmer can pretty much read the entire code being executed in one place.

© 2019 Ritwik Banerjee

Concurrency model: functional parallelism

- Implement the program using function calls, where each function is seen as an "agent" who is sending a message to another one (implemented as a function call).
- All parameters passed to a function are copied, so no outside entity can modify the data. This automatically avoids race conditions on shared data.
  - Note that this effectively makes each function an atomic operation.
  - Therefore, a function call can be executed completely independent of another function call, which in turn means that an algorithm implemented functionally can be parallelized on, say, multiple CPUs.
- There's a caveat, as always:
  - it's not always obvious to figure out which function calls to parallelize.
  - coordinating function calls across CPUs has a performance overhead, and the functional model of parallelism is not worth it unless the work done by a function is enough to compensate for that overhead.

© 2019 Ritwik Banerjee

Next, we will see how these various models of concurrency are implemented in Java, and how we can create multithreaded Java programs that avoid the associated problems like deadlock, starvation, memory inconsistency, etc.

© 2019 Ritwik Banerjee

Threads in Java

Direct thread creation & thread management	Create a **`lava.lang.Thread`** object whenever the program needs to start an *asynchronous* task.
Abstraction for thread creation and thread management, separating it from the rest of the program. (Modular code)	There are a few different ways of doing this. E.g., pass the *asynchronous* task to a **`java.lang.concurrent.Executor`**. The executor is an interface, and must be implemented according to the task your program needs to accomplish.

© 2019 Ritwik Banerjee

Thread Creation

1

Implement **`java.lang.Runnable`** and create an instance of that class.

2

Extend **`java.lang.Thread`** and create an instance of that subclass.

© 2019 Ritwik Banerjee

Implementing your own **`Runnable`**

- The interface **`java.lang.Runnable`** is a functional interface with just one defined method, **`run()`**, whose return type is **`void`**.
- The runnable instance is passed to the Thread constructor:

```
public class RunnableImpl implements Runnable {

    public void run() {
        System.out.println("A runnable implementation.");
    }

    public static void main(String... args) {
        (new Thread(new RunnableImpl())).start();
    }
}
```

© 2019 Ritwik Banerjee

Extending the **`Thread`** class

- The **`Thread`** class itself actually implements **`Runnable`**, but its **`run()`** method does nothing.
- So, if you write a subclass of **`Thread`**, it should provide its own implementation overriding the empty **`run()`** method in the superclass.

```
public class ThreadImpl extends Thread {

    public void run() {
        System.out.println("A thread subclass implementation");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

© 2019 Ritwik Banerjee

```java
public class RunnableImpl implements Runnable {

    public void run() {
        System.out.println("A runnable implementation.");
    }

    public static void main(String... args) {
        (new Thread(new RunnableImpl())).start();
    }
}
```

1

```java
public class ThreadImpl extends Thread {

    public void run() {
        System.out.println("A thread subclass implementation");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

2

Comparing the two ways of creating threads

1) More flexible: the **`Runnable`** can extend a class other that **`Thread`** *while* implementing an asynchronous task.
   - This is a nice abstraction, which decouples the actual task being run, from the object that's running it.

2) Extending the **`Thread`** class is the easier option for simpler tasks.
   - It is not as flexible as the first option, but the **`Thread`** class already has a number of methods useful for thread management.

© 2019 Ritwik Banerjee

Pausing a thread

A currently running thread will suspend execution if **`Thread.sleep(long)`** is invoked. The processor time can then be made available to other threads.

It's an overloaded method:

- **`Thread.sleep(long milliseconds)`**, and
- **`Thread.sleep(long milliseconds, int nanoseconds)`**.

Do not assume that invoking sleep will suspend the thread for precisely that specified amount of time.

- Time facilities are OS-dependent.
- The sleep period can be *interrupted* by other

```
public static class SleepMessages {
    public static void main(String[] args) throws InterruptedException {
        String[] sentences = {"Mares eat oats",
                              "Does eat oats",
                              "Little lambs eat ivy",
                              "Kids will eat ivy too"};
        for (String s : sentences) {
            Thread.sleep(4000); // suspend execution for 4 seconds
            System.out.println(s);
        }
    }
}
```

© 2019 Ritwik Banerjee

Pausing a thread

- When a sleeping thread is interrupted, it throws **`InterruptedException`**. In this example, there is no need to catch it since there is only one thread.
- Interruption can be invoked by Thread.interrupt(). This is meant to indicate that the thread should stop what it's doing and do something else.
  - What is this "something else"? Up to the programmer.
  - A common practice is to simply terminate the thread.
  - For interruption to work properly, a thread must support its own interruption.

```
public static class SleepMessages {
    public static void main(String[] args) throws InterruptedException {
        String[] sentences = {"Mares eat oats",
                              "Does eat oats",
                              "Little lambs eat ivy",
                              "Kids will eat ivy too"};
        for (String s : sentences) {
            Thread.sleep(4000); // suspend execution for 4 seconds
            System.out.println(s);
        }
    }
}
```

© 2019 Ritwik Banerjee

How does a thread support its own interruption?

- Generally speaking, this will depend on what the thread is doing. A common practice is to simply return from the run() method:

```
public class SleepMessages {
    public static void main(String[] args) throws InterruptedException {
        String[] sentences = {"Mares eat oats",
                              "Does eat oats",
                              "Little lambs eat ivy",
                              "Kids will eat ivy too"};
        for (String s : sentences) {
            try {
                Thread.sleep(4000); // suspend execution for 4 seconds
            } catch (InterruptedException e) {
                return; // no more printing once interrupted
            }
            System.out.println(s);
        }
    }
}
```

© 2019 Ritwik Banerjee

Thread interruption example

```
public class JavaFXThreadInterrupt extends Application {

    private static boolean _first_click = true;

    @Override
    public void start(Stage primaryStage) {
        BorderPane borderPane  = new BorderPane();
        Button     resetButton = new Button("Start Counter");

        Thread countingThread = new Thread(new Runnable() {
            @Override public void run() {
                for (int i = 0; ; i++) {
                    try {
                        System.out.printf("%d ", i);
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        i = 0;
                        System.println();
                    }
                }
            }
        });
```

© 2019 Ritwik Banerjee

Thread interruption example

```
        resetButton.setOnMouseClicked(e -> {
            resetButton.setText("Reset Counter");
            if (_first_click) {
                countingThread.start();
                _first_click = false;
            } else
                countingThread.interrupt();
        });

        borderPane.setCenter(resetButton);

        Scene scene = new Scene(borderPane, 400, 200);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Counting Interruption Example");
        primaryStage.show();
    }
}
```

© 2019 Ritwik Banerjee

Thread priorities and scheduling

- In a Java application, **`main`** is a thread on its own.
- Once multiple threads are running, the runtime *thread scheduler* decides which thread is scheduled to run.
- This is decided based on the **priority** of a thread (relative to other threads in the application).
- The priority of a thread is a constant **`int`** ranging from **`MIN_PRIORITY`** (= 1) to **`MAX_PRIORITY`** (= 10).
  - The default value is **`NORM_PRIORITY`** (= 5)
- When a thread is created by another, it inherits the priority of the creator thread.
- The **`Runnable`** thread with the highest priority is selected to run. Only when this thread **`stop()`**s, **`yield()`**s (or becomes un-runnable in some other way) does a thread with lower priority get to run.

© 2019 Ritwik Banerjee

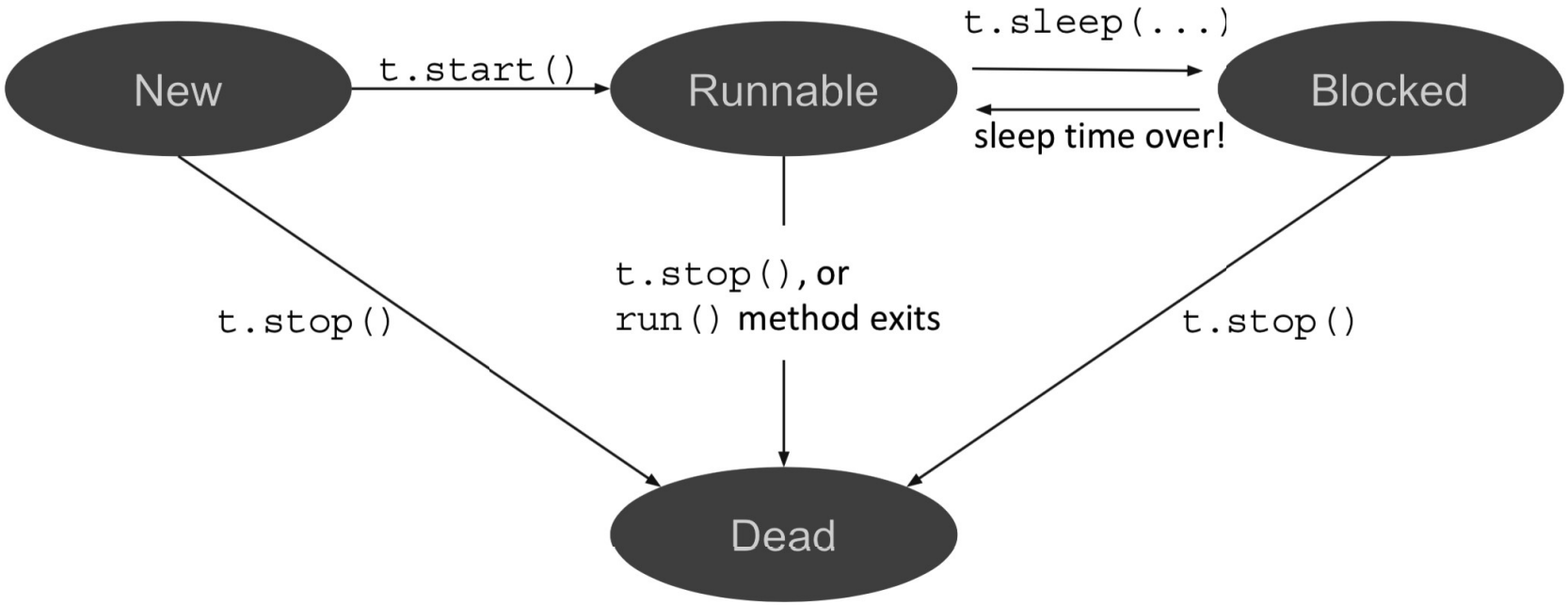
Thread priorities and scheduling

- If multiple threads with the same priority are ready to run, the scheduler picks one randomly, and the chosen thread will run until one of the following happens:
  - a higher priority thread becomes **`Runnable`**
  - this thread **`stop()`**s, **`yield()`**s or its **`run()`** method finishes running and exits.
  - the time-slicing of the OS decides this thread's time allocation is over.
- At any given time, the highest priority thread is running. But there is *no guarantee*! The thread scheduler may some times run a lower-priority thread to avoid starvation.

> As a programmer, you should use priorities for efficiency, but never exclusively rely on them for algorithmic correctness.

© 2019 Ritwik Banerjee

```
Thread t = new Thread();
```

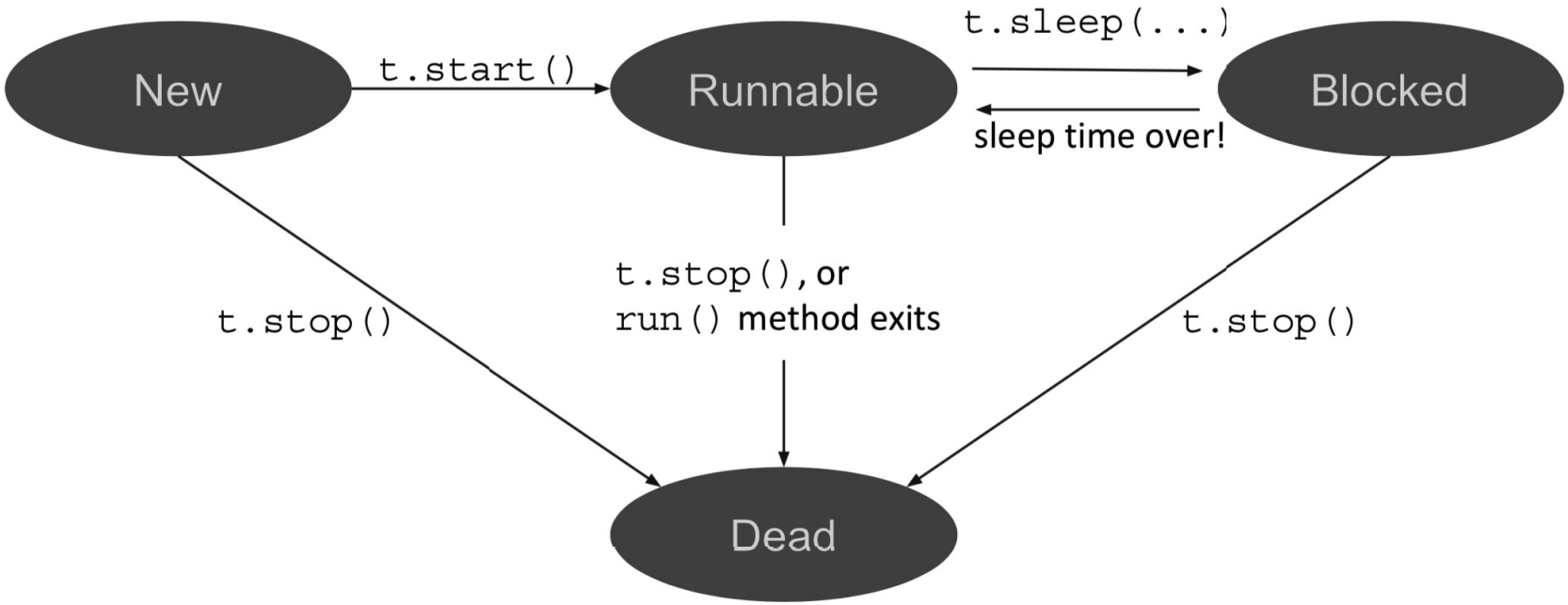


State transition diagram

- This is a slightly simplified picture. For details, see the `Thread.State` enumerable type.

© 2019 Ritwik Banerjee

```
Thread t = new Thread();
```

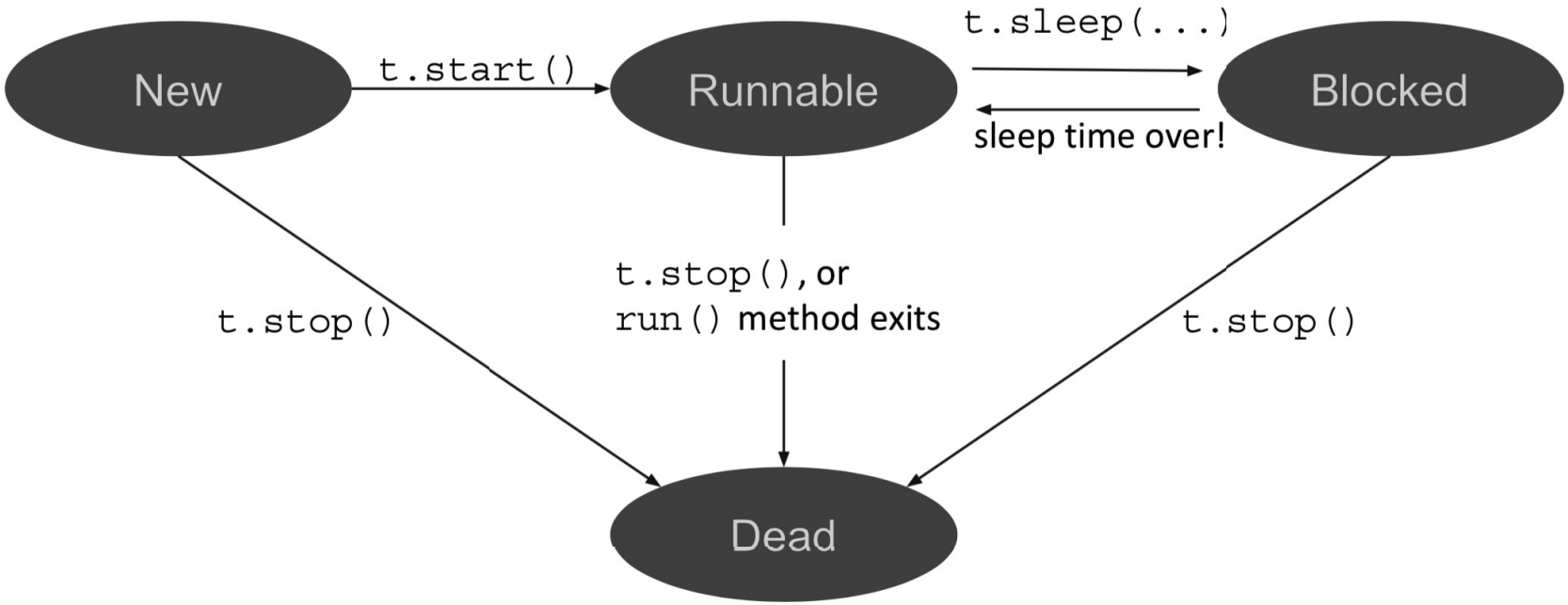


State transition diagram

- In the **new** state, the thread has not been **`start()`**ed, and therefore, is *not* considered to be **`Runnable`** yet.
- In this state, the thread scheduler doesn't know about this thread.

© 2019 Ritwik Banerjee

```
Thread t = new Thread();
```

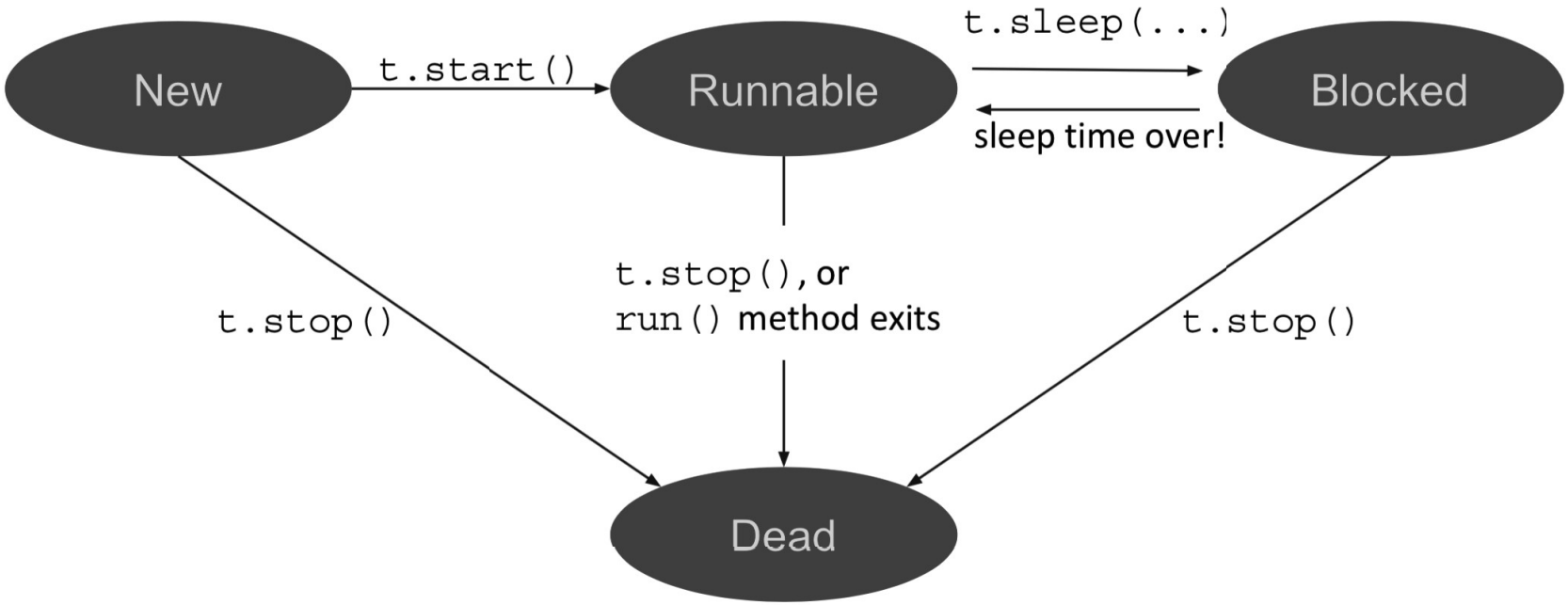


State transition diagram

- In the **<u>runnable</u>** state, the thread may not actually be running, but it is known to the thread scheduler, and it *can be scheduled to run immediately*.
- At any given point in time, multiple threads can simultaneously be in this state.

© 2019 Ritwik Banerjee

```
Thread t = new Thread();
```



State transition diagram

- Lastly, a thread arrives at the **dead** state when its **`run()`** method exits.
- Once here, the thread can't be scheduled to run again.
- You may see **`stop()`** being used to reach this state, but it's a deprecated method, so please avoid it!

© 2019 Ritwik Banerjee

```java
private class Wool extends Thread {

    @Override
    public void run() {
        System.out.printf("Running thread %d.%n", Thread.currentThread().getId());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

/** Put break points and use the debugger to see how many threads are running. */
public static void main(String... args) {
    Wool t = new Wool();
    System.out.printf("Running thread %d.%n", Thread.currentThread().getId());
    t.run();
    t.start();
}
```

Spawning a new thread

- We saw how to create a new thread (either by extending the **`Thread`** class, or by implementing the **`Runnable`** interface).
- But that alone doesn't ensure that the task will actually be running as a separate thread!

© 2019 Ritwik Banerjee

Joining threads

© 2019 Ritwik Banerjee

Concurrent Implementations

- We have discussed some of the issues plaguing asynchronous tasks, and example/overview of simple multithreaded Java programs.
- It's time to put the two together.

© 2019 Ritwik Banerjee

Creating the "happens before" relation

Invoke a thread's **`start()`** method.

- Everything that happens before this invocation is within a single thread, so the transactions are executed in the order of the code (i.e., sequentially, line by line).
- Every statement before calling **`start()`** ***happens before*** every statement in the newly spawned thread.

By using synchronization.

- This prevents thread interference and memory inconsistency errors.
- But if used incorrectly, can cause thread contention issues including *starvation*, *livelock*, and *deadlock*.
- There are some specific types of inter-thread synchronization that happen when threads wait for each other. We will see this later in the **`join()`**, **`notify()`**, and **`wait()`** methods.

© 2019 Ritwik Banerjee

Synchronization

Synchronization is built around an entity called the **intrinsic lock** or the **monitor lock**.

Every object has an intrinsic lock associated with it.

If a thread needs exclusive access to an object, it acquired that object's intrinsic lock. After accomplishing its task, it releases the lock.

There are other types of lock too. E.g., a **ReentrantLock**.

© 2019 Ritwik Banerjee

Synchronized methods

- Just add the **`synchronized`** keyword to the method declaration.
- Each call to a synchronized method is *atomic*.
- When one thread is executing a synchronized method for **`this`**, all other threads that invoke synchronized methods for the same object suspend execution until the first thread is done.
- When a synchronized method exits, it establishes a *happens before* relation with any future invocation of a synchronized method for the same object.

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() { c++; }

    public synchronized void decrement() { c--; }

    public synchronized int value() {
        return c;
    }
}
```

© 2019 Ritwik Banerjee

Synchronized blocks

- Code blocks that are not methods can be synchronized too.
- A synchronized statement can be used to acquire a lock on *any* object, not just **this**, when executing a block of the code in a method:

```
synchronized (expr) {
    /* code block */
}
```

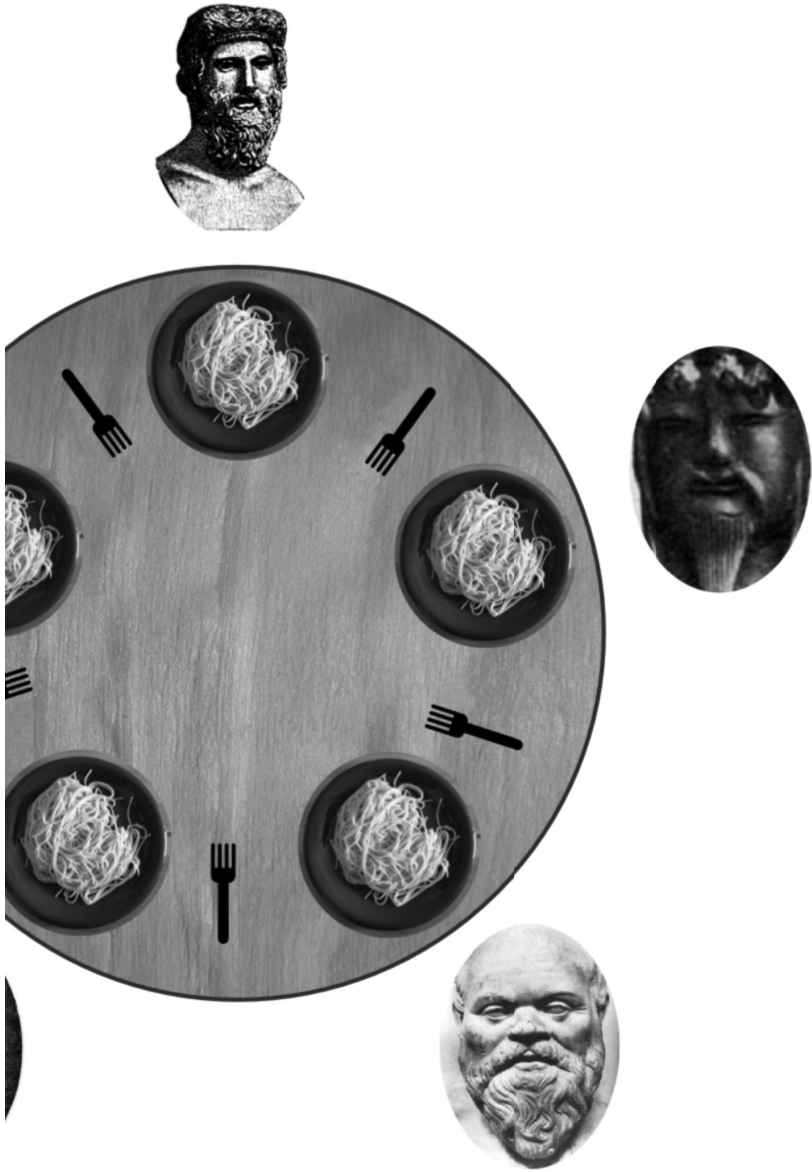
- In such a code, expr must evaluate to an object reference.
  - If that object is already locked by another thread, the thread executing this code block remains blocked until the lock is released.
  - When the thread executing this code block obtains the lock on the object, the statements in the synchronized block are executed.
  - This execution is *atomic*.
  - Once the work is done, the thread releases the lock.

© 2019 Ritwik Banerjee

The Dining Philosophers problem

Five (in general, n) silent philosophers sit at a table around a bowl of spaghetti, and a fork is placed between each pair of adjacent philosophers.

- A philosopher must either think or eat (can't do both at the same time).
- A philosopher can only eat spaghetti when he has both left and right forks.
- A philosopher can grab the fork on his right or the one on his left as they become available, but can't start eating before getting both of them.
- The possibility of a **deadlock**: if all 5 philosophers pick up the left fork at the same time and wait until the right fork is available, then no progress is possible again (**starvation**).
- This was originally formulated by Dijkstra, who gave this as an exam problem in 1965.



© 2019 Ritwik Banerjee

Inter-thread communication & coordination

- Communication and coordination always happens through some 'language' that serves as a common medium.
- Threads, too, need to communicate, and this is done through the 'language' of certain conventions. Without these, we may end up in the worrisome situations (e.g., livelock).
- One common convention is the **guarded block**, which keeps checking for some condition to be satisfied.
  - A thread may run only when that condition is, indeed, satisfied.
  - It's a simple enough concept, but we must be careful with the sequence of steps to follow when implementing this convention.

© 2019 Ritwik Banerjee

Guarded blocks

- Let's say we have a method **`guardedJoy()`** that *must not proceed unless* a Boolean shared variable **`joy`** is set to **`true`** by another thread.
- It's a simple concept. We don't even need to know any implementation related to threads.
- Of course we can guard against a condition with a loop, but this wastes CPU cycles.
  - Which is exactly what we want to avoid in multithreaded applications!
  - **Don't do this!**

```
public void guardedJoy() {
    while (!joy) { /* do nothing */ }
    System.out.println("I am happy now!");
}
```

© 2019 Ritwik Banerjee

```
public synchronized void guardedJoy() {
    while (!joy) {
        try { wait(); }
        catch (InterruptedException e) {
            /* do nothing */
        }
    }
    System.out.println("I am happy now!");
    System.out.println("And efficient!");
}
```

Guarded blocks

- If the thread that changes **`joy`** could notify our thread once the change is made, then we can avoid checking the value constantly in a loop, and instead just wait.
- There are two methods to do precisely that: **`notify()`** and **`wait()`**, defined for all objects. A more efficient guard will use these.

© 2019 Ritwik Banerjee

```
public synchronized void guardedJoy() {
    while (!joy) {
        try { wait(); }
        catch (InterruptedException e) {
            /* do nothing */
        }
    }
    System.out.println("I am happy now!");
    System.out.println("And efficient!");
}
```

Waiting

Why do we need the try-catch inside the loop? Can't we just **`wait()`** and then print inside the catch block?

- Other threads may issue notifications for many things, *not just* the variable our current thread cares about.
- An **`InterruptedException`** may occur even if **`joy`** remains unchanged.
- This guard loops once *per notification*, but it will still be orders of magnitude more efficient than our first code (in terms of CPU cycles).

© 2019 Ritwik Banerjee

```
public synchronized void guardedJoy() {
    while (!joy) {
        try { wait(); }
        catch (InterruptedException e) {
            /* do nothing */
        }
    }
    System.out.println("I am happy now!");
    System.out.println("And efficient!");
}
```

Waiting

Why do we need the method to be **`synchronized`**?

- Notice that **`wait()`** is an instance method. So the question is, which object's **`wait()`** are we calling?
- We are calling **`this.wait()`**, and a thread calling it *must* own the intrinsic lock of **`this`**.
- Otherwise, an exception gets thrown.

© 2019 Ritwik Banerjee

```
public synchronized void guardedJoy() {
    while (!joy) {
        try { wait(); }
        catch (InterruptedException e) {
            /* do nothing */
        }
    }
    System.out.println("I am happy now!");
    System.out.println("And efficient!");
}
```

Waiting

Why did the language designers want **`wait()`** to only be allowed when the thread calling it owns the object's intrinsic lock?

- The **`wait()`** method actually releases the lock *while* it waits, so that other threads can acquire the lock and do their job.
- When *another* thread finishes its job, it will issue a notification.
- That, in turn, will interrupt our thread and make it check the value of joy.

© 2019 Ritwik Banerjee

Notifications

- Notifications can be sent by a thread in two forms: **`notify()`** and **`notifyAll()`**.
- These are also instance methods. That is, a thread is invoking them in the form of **`obj.notify()`** or **`obj.notifyAll()`**, for some object **`obj`**.
- The thread running the **`spreadJoy()`** method will notify every thread that is waiting on the intrinsic lock of **`this`** object.

```
public synchronized void spreadJoy() {
    joy = true;
    notifyAll();
}
```

© 2019 Ritwik Banerjee

Notifications

- Notifications can be sent by a thread in two forms: **`notify()`** and **`notifyAll()`**.
- The notify() method, in contrast, wakes up a single thread out of possibly many waiting on this object's intrinsic lock.
  - The choice of the thread is random.
  - This is useful only in *massively* parallel applications where a very large number of threads are all performing similar tasks.
  - We don't necessarily care which specific thread gets woken up, but we don't want all those threads to be interrupted when, finally, only one of them will actually be scheduled.
  - The cost of that many interruptions (because the number of threads is so high) is not worth it.

© 2019 Ritwik Banerjee

The producer-consumer model through an example

- In this example, our data is going to be a series of text messages.
- The **producer** creates the data, and the **consumer** takes that data and does something with it.
- The two threads communicate using a shared object.
  - Communication is critical to avoid many invalid scenarios, e.g., the consumer trying to retrieve the data before it has been produced!
- To model the shared entity used by the producer and the consumer threads, we are going to create an object called **`Drop`**.

© 2019 Ritwik Banerjee

```java
public static class Drop {

    /** The message sent from producer to consumer */
    private String message;

    /**
     * A status that is <code>true</code> if the consumer should wait for the
     * producer to send a message, and <code>false</code> if the producer should
     * wait for the consumer to retrieve a message.
     */
    private boolean empty = true;

    // cont. next slide
```

The shared object

© 2019 Ritwik Banerjee

```java
/**
 * This method waits until the next message is available. It then toggles the
 * {@link #empty} status and issues a notification that the message is now
 * available.
 *
 * @return the currently available message.
 */
public synchronized String take() {
    while (empty)
        try { wait(); } catch (InterruptedException ignore) { /* do nothing */ }
    empty = true;
    notifyAll();
    return message;
}

// cont. next slide
```

The shared object

© 2019 Ritwik Banerjee

```java
    /**
     * This method waits until a message has been retrieved. It then toggles the
     * {@link #empty} status, stores the next message, and issues a notification
     * that the next message is now available for retrieval.
     *
     * @param message the current message to be retrieved next by the consumer.
     */
    public synchronized void put(String message) {
        while (!empty)
            try { wait(); } catch (InterruptedException ignore) { /* do nothing */ }
        empty = false;
        this.message = message;
        notifyAll();
    }
}
```

The shared object

© 2019 Ritwik Banerjee

```java
public class Producer implements Runnable {

    protected static final String DONE = "done";

    private Drop drop;

    public Producer(Drop drop) { this.drop = drop; }

    public void run() {
        Random r = new Random();
        List<String> messages = Arrays.asList("Mares eat oats.",
                                              "Does eat oats.",
                                              "Little lambs eat ivy.",
                                              "Kids will eat ivy too.");
        messages.forEach((String msg) -> {
            drop.put(msg);
            try { Thread.sleep(r.nextInt(5000)); }
            catch (InterruptedException ignore) { /* do nothing */ }
        });
        drop.put(DONE);
    }
}
```

The producer thread

© 2019 Ritwik Banerjee

```java
public class Consumer implements Runnable {

    private Drop drop;

    public Consumer(Drop drop) { this.drop = drop; }

    public void run() {
        Random r = new Random();
        for (String msg = drop.take(); !msg.equals(Producer.DONE); msg = drop.take()) {
            System.out.printf("MESSAGE RECEIVED: %s\n", msg);
            try {
                Thread.sleep(r.nextInt(5000));
            } catch (InterruptedException ignore) {/* do nothing */ }
        }
    }
}
```

The consumer thread

© 2019 Ritwik Banerjee

```
public static void main(String... args) {
    Drop drop = new Drop();
    (new Thread(new Producer(drop))).start();
    (new Thread(new Consumer(drop))).start();
}
```

The main thread

© 2019 Ritwik Banerjee

```java
public static void main(String... args) {
        Thread t1 = new Thread(() -> {
            try {
                Thread.sleep(new Random().nextInt(2000));
            } catch (InterruptedException ignore) {
                /* do nothing */
            }
            System.out.println("Running t1.");
        });
        Thread t2 = new Thread(() -> {
            try {
                Thread.sleep(new Random().nextInt(4000));
            } catch (InterruptedException ignore) {
                /* do nothing */
            }
            System.out.println("Running t2.");
        });

        t1.start();
        t2.start();

        System.out.println("Running the main thread.");
    }
```

Thread joins

- The calling thread waits (i.e., pauses its own execution) until the referenced thread dies.
- To understand its effects, let's look at this code, which doesn't join the threads at all.
- This is a possible output:

```
Running the main thread.
Running t1.
Running t2.
```

© 2019 Ritwik Banerjee

Thread joins

- The same code with joins:
  - the main thread creates and starts **t1** and **t2**
  - the two threads **t1** and **t2** may run concurrently
  - the main thread waits for **t1** to finish running (if **t1** finished running before **join()** was called, then the call to **join()** will return immediately to the main thread)
  - it also waits for **t2** to finish running (it is possible that this happens before **t1** finished running)
  - no matter the order of **t1** and **t2**, it is the main thread that is waiting for both to finish.
- Now, the output will always be

```
Running t1.
Running t2.
Running the main thread.
```

```java
public static void main(String... args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        try {
            Thread.sleep(new Random().nextInt(2000));
        } catch (InterruptedException ignore) {
            /* do nothing */
        }
        System.out.println("Running t1.");
    });
    Thread t2 = new Thread(() -> {
        try {
            Thread.sleep(new Random().nextInt(4000));
        } catch (InterruptedException ignore) {
            /* do nothing */
        }
        System.out.println("Running t2.");
    });

    t1.start();
    t2.start();

    t1.join();
    t2.join();

    System.out.println("Running the main thread.");

}
```

© 2019 Ritwik Banerjee