Multithreaded Programming

CSE 219
Stony Brook University, Department of Computer Science
Concurrent programs

- Single core machine *without* concurrency
  - only one application at a time
  - no more coding while listening to music

- Single application *with* concurrency
  - read digital audio off the network
  - decompress it
  - manage playback
  - update display

- Java has excellent support for concurrency: `java.util.concurrent`
Processes and Threads

- In concurrent programming, there are two basic units of execution
  - Processes
  - Threads

- In Java, concurrent programming is *mostly* concerned with threads

- A computer system normally has many active processes and threads
  - Even in a single-core machine
    - i.e., only one thread actually executing at any given moment
  - The single core is shared among processes and threads through *time slicing*
    - OS feature
Processes

- Have self-contained execution environment
- Generally have a complete, private set of basic run-time resources
  - in particular, have their own memory space
- Not the same as a “program” or an “application”
  - a single application may actually be a set of processes working together
  - Inter-Process Communication (IPC) are OS-specific (e.g., pipes and sockets)
- In Java
  - Most JVMs run as a single process
  - A Java application can create multiple processes using a ProcessBuilder object.
Threads

- “Lightweight” processes
  - Creating a new thread requires fewer resources than creating a new process

- Exist within processes
  - Every process has at least one thread
  - Threads share the process resources, (memory, open files, etc.)
    - Increases efficiency but also creates communication issues

- In Java
  - You start with just one thread, called the main thread
    - more if you include System threads responsible for memory management
  - This main thread can create additional threads
Threads

- "Lightweight" processes
  - Creating a new thread requires fewer resources than creating a new process

- Exist within processes
  - Every process has at least one thread
  - Threads share the process resources, (memory, open files, etc.)
    - Increases efficiency but also creates communication issues

- In Java
  - You start with just one thread, called the main thread
    - more if you include System threads responsible for memory management
  - This main thread can create additional threads

Remember this bit … this is about to become very important
Synchronous and Asynchronous Tasks

- "Synchronously" means "using the same clock"
  - so when two instructions are synchronous they use the same clock and must happen one after the other

- Synchronous task
  - you wait for the first task to finish before moving on to another one

- "Asynchronous" means "not using the same clock"
  - so the instructions are not concerned with being in step with each other

- Asynchronous task
  - you can move on to another task before the first task finishes
Threads

- Each thread is an instance of the `java.lang.Thread` class
- To create a concurrent application, there are two main strategies for using these thread objects
  - Direct thread creation and thread management
    - Create a `java.lang.Thread` whenever the application needs to start an asynchronous task
  - “Abstract out” thread management from the rest of your application
    - There are a few ways of doing this
    - One important way to do this is to pass the application’s tasks to an `executor`
      - `java.util.concurrent.Executor` (an interface)
Thread Creation

- To directly create a Thread instance, we have to provide the code to run that thread.

- There are two ways of doing this
  - Provide a `java.lang.Runnable` object
  - Create a subclass of `java.lang.Thread`
Thread Creation with Runnable

-Runnable is an interface that defines a single method: `run()`
- The Runnable object is passed to the Thread constructor

```java
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```
Creating a subclass of Thread

- The Thread class itself implements Runnable
  - But its run() method does nothing
- An application can subclass Thread
  - providing its own implementation of run()

```java
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```
### Thread Creation: a comparison

**Runnable** object can be a subclass of something other than `Thread`.

**Decouples** the Runnable task from the Thread object that executes the task.

Applicable to the high-level thread management APIs of Java.

**Easier to use in simple applications**

But has relatively limited flexibility because your task class must be a descendant of `Thread`.

The `Thread` class defines a number of methods useful for thread management, including some static methods.

```java
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }

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

public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }

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

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

public class SleepMessages {

  public static void main(String args[]) throws InterruptedException {
    String importantInfo[] = {
      "Mares eat oats", "Does eat oats",
      "Little lambs eat ivy", "Kids will eat ivy too"};
    for (String anImportantInfo : importantInfo) {
      Thread.sleep(4000); // suspend thread execution for 4 seconds
    }
  }

- Thread.sleep causes the current thread to suspend execution for a specified period
  - Can be used to make processor time available to the other threads
- Overloaded
  - Thread.sleep(long millis) and Thread.sleep(long millis, int nanos)
- Do NOT assume that invoking sleep will suspend the thread for precisely the time period specified
  - The time facilities are provided by the underlying OS
  - The sleep period can be interrupted by other methods
Pausing Execution: Thread.sleep

```java
public class SleepMessages {

    public static void main(String[] args) throws InterruptedException {
        String importantInfo[] = {
            "Mares eat oats", "Does eat oats",
            "Little lambs eat ivy", "Kids will eat ivy too"};

        for (String anImportantInfo : importantInfo) {
            Thread.sleep(4000); // suspend thread execution for 4 seconds
            System.out.println(anImportantInfo);
        }
    }
}
```

- Thread.sleep causes the current thread to suspend execution for a specified period.
- Can be used to make processor time available to the other threads.
- Overloaded:
  - Thread.sleep(long millis)
  - Thread.sleep(long millis, int nanos)
- Do NOT assume that invoking sleep will suspend the thread for precisely the time period specified.
- The time facilities are provided by the underlying OS.
- The sleep period can be interrupted by other methods.

This is an exception that sleep throws when another thread interrupts the current thread while sleep is active.

This example has only one thread, so there is no need to catch the InterruptedException.
Interruption

- `Thread.interrupt()` indicates to a thread that it should stop its current task and do something else.
  - The programmer can decide how a thread responds to an interrupt
  - A common practice is to simply make the thread terminate
  - A thread sends an interrupt by invoking `Thread.interrupt()`
    - For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption
How does a thread support its own interruption?

- Depends on what the thread is currently doing
  - If it is frequently calling methods that throw InterruptedException, it will simply return from the run() method after catching the exception

```java
for (String anImportantInfo : importantInfo) {
    try {
        Thread.sleep(4000);  // suspend thread execution for 4 seconds
    } catch (InterruptedException e) {
        return;  // there has been an interruption, so no more message printing
    }
    System.out.println(anImportantInfo);
}
```
How does a thread support its own interruption?

- Depends on what the thread is currently doing
  - If it is frequently calling methods that throw InterruptedException, it will simply return from the run() method after catching the exception

```java
for (String anImportantInfo : importantInfo) {
    try {
        Thread.sleep(4000); // suspend thread execution for 4 seconds
    } catch (InterruptedException e) {
        return; // there has been an interruption, so no more message printing
    }
    System.out.println(anImportantInfo);
}
```

- Methods like `sleep` are designed to cancel their current operation and return immediately when an interrupt is received
Joins

- The `join()` method makes one thread wait for another to complete

  Let \( t \) be the Thread object currently executing. Then,

  \[
  t.join();
  \]

  causes the current thread to pause execution until \( t \)'s thread terminates.

- Overloaded method: other versions specify a waiting period

- Like `sleep`

  - don’t assume that the `join` will wait for exactly as long as you specify
  - `join` responds to an interrupt by exiting with an `InterruptedException`
Thread Communication and Synchronization

- There is a need for inter-thread communication
  - This is done mainly by sharing fields and object references
  - This is a very efficient form of communication
    - Why? Because it avoids ACTUALLY sending and receiving data!
  - But it gives rise to the possibility of serious errors

- Broadly speaking, there are two types of errors
  - Thread interference
    - when multiple threads access shared data
  - Memory inconsistency
    - Errors resulting from inconsistent views of shared memory

**Synchronization** is a technique used to prevent these errors