CT326 Programming III

MULTITHREADED PROGRAMMING LECTURES

DR ADRIAN CLEAR
SCHOOL OF COMPUTER SCIENCE

Multithreaded Programming

- A *process* is an instance of a program being executed.
  - In a *multitasking* environment, multiple processes can be running *concurrently*.
    - Actually, at a given instance in time, only one process is running on a single CPU. Time slicing between processes makes it appear as if they are all running at the same time.
- A *thread* is a “lightweight process.”
  - It’s like a process, but it doesn’t have all the *overhead* that a process has.

Processes vs. Threads

- Both processes and threads have their own independent CPU state.
  - i.e. their own processor stack, instruction pointer, and CPU register values.
- Multiple threads can share the same memory address space (i.e. share the same variables).
- Processes, in general, do not share their address space with other processes.

Threads and Java

- Java has language level support for threads.
- In Java it is easy to create multiple independent threads of execution.
  - Any class that is a subclass of `java.lang.Thread` or implements the `java.lang.Runnable` interface can can be used to create threads.

java.lang.Runnable

- To create a thread, you instantiate the `java.lang.Thread` class.
  - One of `Thread`'s constructors takes objects that implement the `Runnable` interface.
  - The Runnable interface contains a single method called `run()`:

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

Creating and starting a thread

```
class Fred implements Runnable {
  //..
  public void run() {
    //..
  }

  public static void main(String args[]) {
    Thread t = new Thread(new Fred());
    t.start();
  }
}
```

`start()` and `run()`

- To create a thread, create an instance of the `Thread` class.
- To start a thread, call the `start()` method on the thread.
  - When the thread starts, the `run()` method is invoked.
- The thread terminates when the `run()` method terminates.

Thread priority and daemons

- Every thread has a *priority*.
  - Threads with *higher priority* are executed in *preference* to threads with *lower priority*.
- A thread may be marked as a *daemon*.
  - Daemon threads are *background threads* that are not expected to *exit.*
- A *new thread* has its priority initially set equal to the priority of the *creating thread*.
  - The new thread is a daemon thread if and only if the creating thread is a daemon thread.

JVM and threads

- When the JVM starts up there is usually a single non-daemon (user) thread initially.
  - This thread calls the `main()` method of an application.
- The JVM continues to execute threads until either
  - The `exit()` method of the class `Runtime` is called,
  - or all non-daemon (user) threads have died.

Creating a new thread
Method 1

- Create a subclass of `java.lang.Thread`
  - This subclass should *override* the `run()` method.
- An instance of this subclass can be created and started.

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
       this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
        . . .
    }
}
```

Method 1, continued...

- The following code creates a thread and starts it running
  - The thread executes until its `run()` method terminates or it is hit over the head by calling the thread's `stop()` method.

```
PrimeThread p = new PrimeThread(143);
p.start();
```

Creating a new thread
Method 2

- The other way to create a thread is to declare a class that implements the `Runnable` interface.
- An instance of the class can then be allocated, passed as an argument when instantiating the `Thread` class, and started.

Method 2 continued...

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
       this.minPrime = minPrime;
    }
    public void run() {
       // compute primes larger than minPrime
       . . .
    }
}
. . .
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

The death of a thread

- Threads can be killed by calling their `stop()` method, but this is not a good idea in general.
- Allow for a thread to *die naturally* by having its `run()` method return.
  - This often done by altering some variable that causes a while loop to terminate in the `run()` method.

Simple Animation Demo

- When a program creates a thread, it can pass the *Thread* class constructor an object whose statements will be executed.
- Using the *thread* class *start* function, the program can start a thread's execution.
- The *start* function, in turn, will call the *Thread* objects *run* function.
- Rule of Thumb:
  - If an application or applet performs a time-consuming task, it should create and use its own thread of execution to perform that task.

Simple Animation Demo

Synchronization

- Multi-threaded coding requires special care
  - (also more difficult to debug)
- You want to prevent multiple threads from altering the state of an object at the same time.
  - Sections of code that should not be executed simultaneously are called *critical sections*.
  - You want *mutual exclusion* of concurrent threads in *critical sections* of code.
  - This is done in Java using *synchronized methods* or *synchronized statements*.

The `synchronized` statement

```
synchronized (expression) statement
```

- `expression`: must resolve to an object or an array.
- `statement`: the critical section, usually a statement block (i.e. surrounded by { and }).
- A synchronized statement attempts to acquire a *lock* for the object or array specified by the `expression`.
  - `Statement` is not executed until the lock is obtained.

More on synchronized

- You do not have to use the synchronized statement unless multiple threads share data.

```
public static void sortArray(int[] a) {
  synchronized (a) {
    // sort the array here
  }
}
```

Synchronized methods

- The `synchronized` keyword is most often used as a *method modifier* in Java.
  - Indicates that the *entire method* is a *critical section*.
  - *Static synchronized methods*: Java obtains a lock for the *class*.
  - *Instance methods*: Java obtains an exclusive lock for the *class instance*.

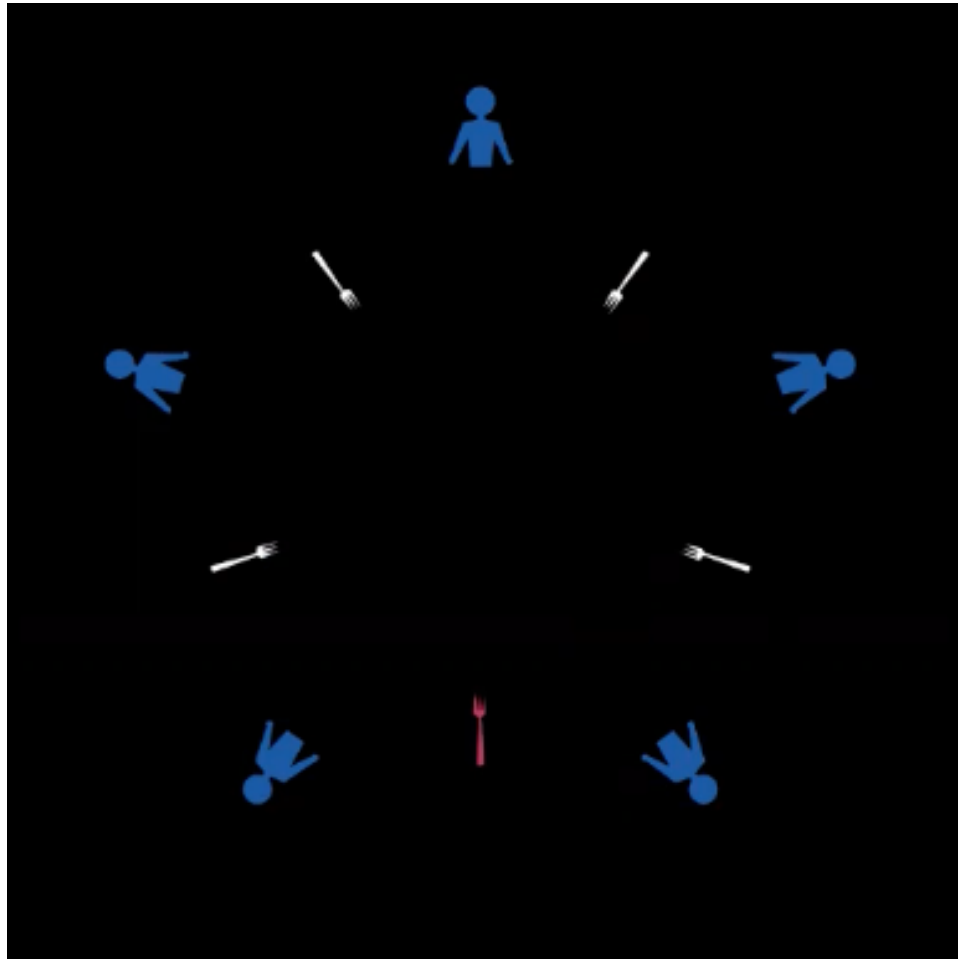
```
public synchronized void sort() {
  // whole method a critical section
}
```

Deadlocks

- Using synchronization can actually cause problems.
  - Only the thread that holds the lock can execute the critical section.
  - When more than one lock is used a situation called *deadlock* can occur.
  - This happens when two or more threads are all waiting to acquire a lock that is held by one of the other waiting threads.
    - Each thread waiting for a lock will never release the locks they currently hold (impasse).

Dining Philosophers Problem

- The story goes like this:
  - Five philosophers are sitting at a round table.
  - In front of each philosopher is a bowl of rice.
  - Between each pair of philosophers is one chopstick.
  - Before an individual philosopher can take a bite of rice he must have two chopsticks one taken from the left, and one taken from the right.
  - The philosophers must find some way to share chopsticks such that they all get to eat.



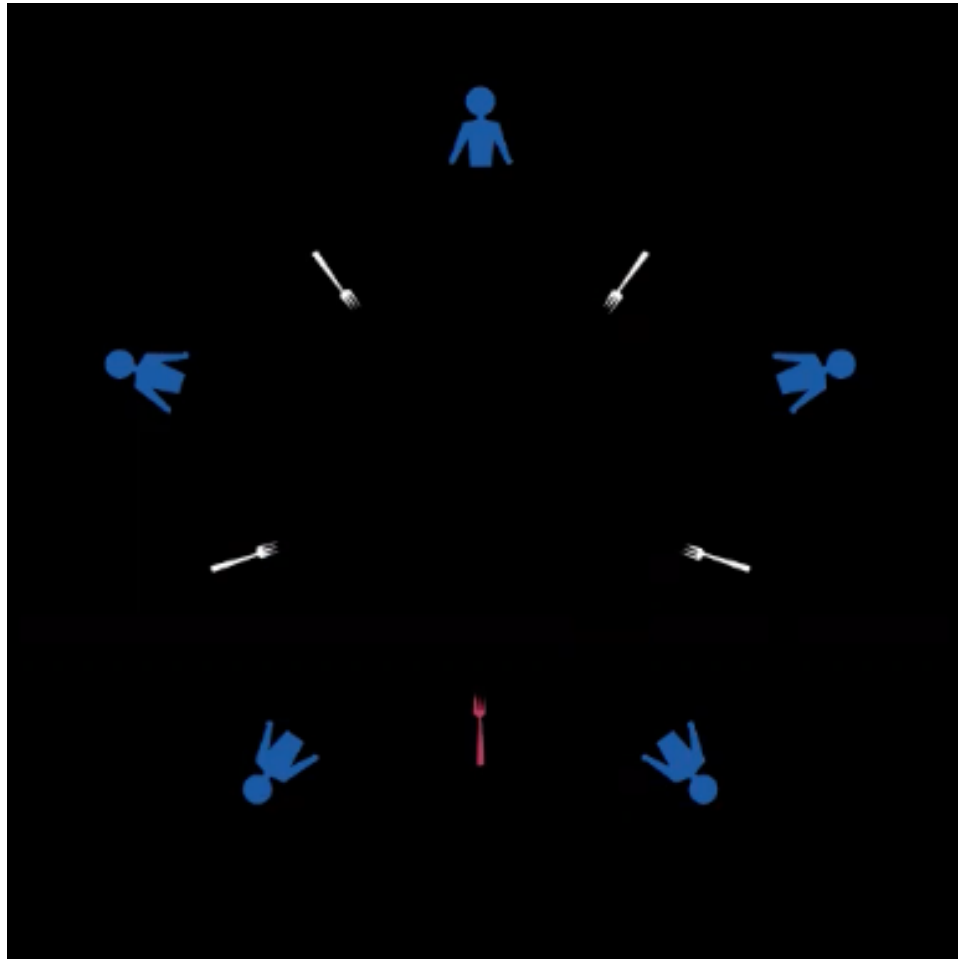
Dining Philosophers Problem

- The algorithm works as follows:
  - The philosopher always reaches for the chopstick on his right first.
    - If the chopstick is there, he takes it and raises his right hand.
    - Next, he tries for the left chopstick.
    - If the chopstick is available, he picks it up and raises his other hand.
  - Now that the philosopher has both chopsticks, he takes a bite of rice and then puts both chopsticks down, allowing either of his two neighbours to get the chopsticks.

Dining Philosophers Problem

- The philosopher then starts all over again by trying for the right chopstick.
- Between each attempt to grab a chopstick, each philosopher pauses for a random period of time.

- This algorithm always ends up in deadlock!
  - All the philosophers are frozen with their right hand in the air. Why?
    - Because each philosopher immediately has one chopstick and is waiting on a condition that cannot be satisfied ...
    - They are all waiting for the left chopstick, which is held by the philosopher to their left.

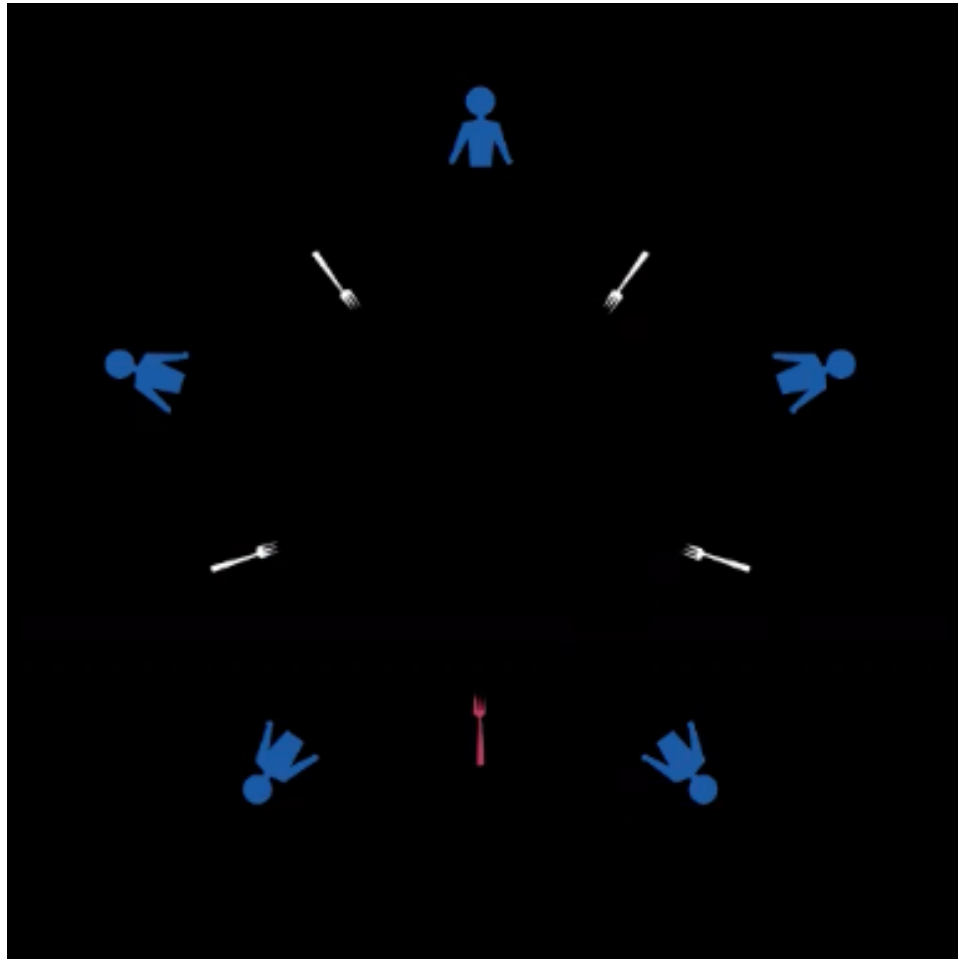


Solution to Dining Philosophers Problem

- For most Java programmers, the best choice is to prevent deadlock rather than to try and detect it.
  - Deadlock detection is complicated and the simplest approach to to preventing deadlock is to impose ordering on the condition variables.
  - In the dining philosopher algorithm, there is no ordering imposed on the condition variables because the philosophers and the chopsticks are arranged in a circle.
  - All chopsticks are equal.

Solution to Dining Philosophers Problem

- We can number the chopsticks 1 through 5 and insisting that the philosophers pick up the chopstick with the lower number first.
  - The philosopher who is sitting between chopsticks 1 and 2 and the philosopher who is sitting between chopsticks 1 and 5 must now reach for the same chopstick first (chopstick 1) rather than picking up the one on the right.
  - Whoever gets chopstick 1 first is now free to take another one.
  - Whoever doesn't get chopstick 1 must now wait for the first philosopher to release it.
- Deadlock is not possible ...



Producer Consumer Problem

- The Producer generates an integer between 0 and 9 (inclusive), stores it in an IntBuffer object, and prints the generated number.
- To make the synchronization problem more interesting, the Producer sleeps for a random amount of time between 0 and 100 mS before repeating the number generating cycle:
- The Consumer, being ravenous, consumes all integers from the IntBuffer (the exact same object into which the Producer put the integers in the first place) as quickly as they become available.

Producer Consumer Problem

```
public class Producer extends Thread {
    private IntBuffer cubbyhole;
    private int number;

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

Producer Consumer Problem

```
    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) { }
        }
    }
}
```

Producer Consumer Problem

```
public class Consumer extends Thread {
    private IntBuffer cubbyhole;
    private int number;

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

Producer Consumer Problem

```
    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" +
          this.number + " got: " + value);
        }
    }
}
```

Producer Consumer Problem

- The Producer and Consumer in this example share data through a common IntBuffer object.
- Note that neither the Producer nor the Consumer makes any effort whatsoever to ensure that the Consumer is getting each value produced once and only once.
- The synchronization between these two threads actually occurs at a lower level, within the get() and put() methods of the IntBuffer object.

Producer Consumer Problem

- Race conditions arise from multiple, asynchronous executing threads trying to access a single object at the same time and getting the wrong result.
- Race conditions in the producer/consumer example are prevented by having the storage of a new integer into the IntBuffer by the Producer be synchronized with the retrieval of an integer from the IntBuffer by the Consumer.
- The Consumer must consume each integer produced by the Producer exactly once.

wait() and notify()

- Java monitors are re-entrant:
  - Once a thread obtains a monitor (lock) it can release it again and wait, using wait(), pending the completion of some other event or action by another thread.
  - Useful in situations where separate threads are supplying and consuming data or events.
  - Can only be executed within synchronized code or methods and helps prevent deadlock and uncoordinated access to shared data ...
  - IntBuffer implementation uses these methods.

IntBuffer Implementation

- Here's the code skeleton for the IntBuffer class:

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

    public synchronized int get() {
        ...
    }

    public synchronized void put(int value) {
        ...
    }
}
```

IntBuffer Implementation

- Note that the method declarations for both put and get contain the synchronized keyword.
- Hence, the system associates a unique lock with every instance of IntBuffer (including the one shared by the Producer and the Consumer).
- Whenever control enters a synchronized method, the thread that called the method locks the object whose method has been called.
- Other threads cannot call a synchronized method on the same object until the object is unlocked.

IntBuffer Implementation

- When the Producer calls IntBuffer's put method, it locks the IntBuffer, thereby preventing the Consumer from calling the IntBuffer's get method:

```
public synchronized void put(int value) {
    // IntBuffer locked by the Producer
    ..
    // IntBuffer unlocked by the Producer
}
```

- When the put method returns, the Producer unlocks the IntBuffer.

IntBuffer Implementation

- Similarly, when the Consumer calls IntBuffer's get method, it locks the IntBuffer, thereby preventing the Producer from calling put:

```
public synchronized int get() {
    // IntBuffer locked by the Consumer
    ...
    // IntBuffer unlocked by the Consumer
}
```

IntBuffer Implementation

- The IntBuffer stores its value in a private member variable called *contents*.
- IntBuffer has another private member variable, *available*, that is a boolean.
- *available* is true when the value has just been put but not yet gotten and is false when the value has been gotten but not yet put.
- There follows one possible implementation for the put and get methods:

IntBuffer Implementation

```
public synchronized int get() {    // won't work!
    if (available == true) {
        available = false;
        return contents;
    }
}
public synchronized void put(int value) {    // won't work!
    if (available == false) {
        available = true;
        contents = value;
    }
}
```

IntBuffer Implementation

- As implemented, these two methods won't work. Look at the *get*() method…
- What happens if the Producer hasn't put anything in the IntBuffer and *available* isn't true? *get*() does nothing.
- Similarly, if the Producer calls *put*() before the Consumer got the value, *put*() does nothing.
- You really want the Consumer to wait until the Producer puts something in the IntBuffer and the Producer should then notify the Consumer when it's done so.

IntBuffer Implementation

- Similarly, the Producer must wait until the Consumer takes a value (and notifies the Producer of its activities) before replacing it with a new value.
- The two threads must co-ordinate more fully and can use Object's wait() and notifyAll() methods to do so.
- There follows the new implementations of *get*() and *put*() that wait on and notify each other of their activities:

IntBuffer Implementation

```
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;
}
```

IntBuffer Implementation

```
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();
    }
```

IntBuffer Implementation

- The code in the get() method loops until the Producer has produced a new value - each time through the loop, get() calls the wait() method.
- The wait() method relinquishes the lock held by the Consumer on the IntBuffer (thereby allowing the Producer to get the lock and update the IntBuffer) and then waits for notification from the Producer.
- When the Producer puts something in the IntBuffer, it notifies the Consumer by calling notifyAll().

IntBuffer Implementation

- The Consumer then comes out of the wait state, available is now true, the loop exits, and the get() method returns the value in the IntBuffer.
- The put() method works in a similar fashion, waiting for the Consumer thread to consume the current value before allowing the Producer to produce a new one.
- The notifyAll() method wakes up all threads waiting on the object in question (in this case, the IntBuffer).

IntBuffer Implementation

- The awakened threads compete for the lock - one thread gets it, and the others go back to waiting.
- The `java.lang.Object` class also defines the notify() method, which arbitrarily wakes up only one of the threads waiting on this object.
- The `Object` class also contains two other versions of the wait method:
  - wait(long timeout) waits for notification or until the timeout period (in mS) has elapsed.
  - wait(long timeout, int nanos) waits for notification or until timeout mS plus nS nanoseconds have elapsed.

Concurrency Utilities

- The Java 2 platform now includes a new package of concurrency utilities.
  - These are classes which are designed to be used as building blocks in building concurrent classes or applications.
  - The Concurrency Utilities include a high-performance, flexible thread pool; a framework for asynchronous execution of tasks; a host of collection classes optimized for concurrent access; synchronization utilities such as counting semaphores; atomic variables; locks; and condition variables.

Concurrency Utilities

- Using the Concurrency Utilities, instead of developing components such as thread pools yourself, offers a number of advantages:
  - Reduced programming effort. It is far easier to use a standard class than to develop it yourself.
  - Increased performance. The implementations in the Concurrency Utilities were developed and peer-reviewed by concurrency and performance experts; these implementations are likely to be faster and more scalable than a typical implementation, even by a skilled developer.
  - Increased reliability. Developing concurrent classes is difficult -- the low-level concurrency primitives provided by the Java language (synchronized, volatile, wait(), notify(), and notifyAll()) are difficult to use correctly, and errors using these facilities can be difficult to detect and debug.

Concurrency Utilities

- By using standardized, extensively tested concurrency building blocks, many potential sources of threading hazards such as deadlock, starvation, race conditions, or excessive context switching are eliminated.
- Improved maintainability. Programs which use standard library classes are easier to understand and maintain than those which rely on complicated, homegrown classes.
- Increased productivity. Developers are likely to already understand the standard library classes, so there is no need to learn the API and behavior of ad-hoc concurrent components.

Thread Pools

- A thread pool is a managed collection of threads that are available to perform tasks. Thread pools usually provide:
- Improved performance when executing large numbers of tasks due to reduced per-task invocation overhead.
- A means of bounding the resources, including threads, consumed when executing a collection of tasks.
- In addition, thread pools relieve you from having to manage the life cycle of threads. They allow to take advantage of threading, but focus on the tasks that you want the threads to perform, instead of the thread mechanics.

Thread Pools

- To use thread pools, you instantiate an implementation of the ExecutorService interface and hand it a set of tasks.
- The choices of configurable thread pool implementations are ThreadPoolExecutor and ScheduledThreadPoolExecutor.
- These implementations allow you to set the core and maximum pool size, the type of data structure used to hold the tasks, how to handle rejected tasks, and how to create and terminate threads.
- However, it is recommended that you use the more convenient factory methods of the Executors class listed in the following table. These methods preconfigure settings for the most common usage scenarios.

Thread Pools

- Factory Methods in the Executors Class

Method	Description
newFixedThreadPool(int)	Creates a fixed size thread pool.
newCachedThreadPool	Creates unbounded thread pool, with automatic thread reclamation.
newSingleThreadExecutor	Creates a single background thread.

Thread Pools

```
public class WorkerThread implements Runnable {
    private int workerNumber;

    WorkerThread(int number) {
        workerNumber = number;
    }

    public void run() {
        for (int i=0;i<=100;i+=20) {
        // Perform some work ...
            System.out.println("Worker number: "
                + workerNumber
                + ", percent complete: " + i );
            try {
                Thread.sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

Thread Pools

```
import java.util.concurrent.*;
public class ThreadPoolTest {
    public static void main(String[] args) {
        int numWorkers = Integer.parseInt(args[0]);
        int threadPoolSize = Integer.parseInt(args[1]);

        ExecutorService tpes =
            Executors.newFixedThreadPool(threadPoolSize);

        WorkerThread[] workers = new WorkerThread[numWorkers];
        for (int i = 0; i < numWorkers; i++) {
            workers[i] = new WorkerThread(i);
            tpes.execute(workers[i]);
        }
        tpes.shutdown();
    }
}
```