



Presents

Concurrency

Concurrency

- ▶ Concurrency is doing more than one thing at a time
 - ✓ With modern multicore architectures, concurrency is a fundamental construct in almost all programming languages
- ▶ Java was designed as a multi-threaded programming language
 - ✓ Remember that Java runs on the JVM
 - ✓ The JVM is a multithreaded environment and maps threads in the JVM to host OS threads
 - ✓ In the early years, this was not always possible since some OS did not support.

Concurrency

- ▶ There are background threads running in the JVM
 - ✓ For example, the garbage collector that reclaims memory
- ▶ Processes
 - ✓ Created by a system call and uses IPC (inter-process communication) which requires system calls
 - ✓ Isolated execution space and does not share data
 - ✓ Has its own stack, heap and data map
 - ✓ Processes are managed by the OS process scheduler
 - ✓ Heavyweight – takes time to start up and shut down

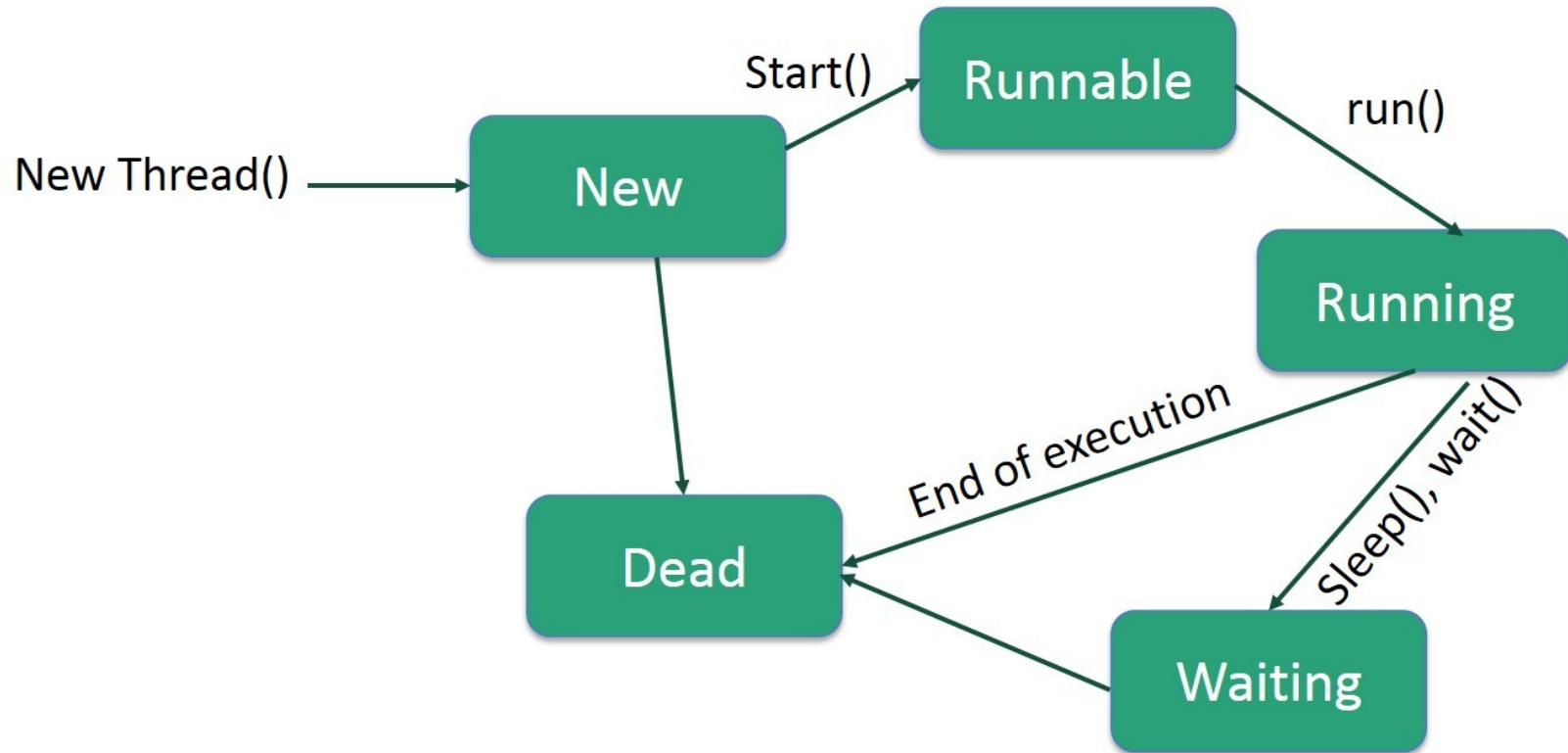
Concurrency

- ▶ Threads
 - ✓ All peer threads are treated as a single process
 - ✓ Each thread has its own stack and registers
 - ✓ Peer threads share memory and instruction space
 - ✓ Threads can share data through shared memory
 - ✓ Threads are lightweight
 - ✓ Threads are faster to start up and shut down
- ▶ Threads can be user defined as part of the overall application architecture

Concurrency

- ▶ Example, two development teams are like processes
 - ✓ Each operates independently but they can communicate
 - ✓ Each has its own budget, work area and resources
 - ✓ Starting up and shutting down a project can be complex
- ▶ The members of a team are like threads
 - ✓ They do their work concurrently
 - ✓ They share some resources but also have some unshared resources
 - ✓ Tasks are easily assigned, and operate within the context of a project

Thread Lifecycle



Thread Lifecycle

- ▶ New
 - ✓ Initial state of a thread
 - ✓ The thread stays in this state until the program starts the thread
- ▶ Runnable
 - ✓ After the thread is started, it is considered to be executing.
- ▶ Waiting
 - ✓ When waiting for another thread to complete a task.
 - ✓ Transition back to runnable when it receives a signal that it can proceed.

Thread Lifecycle

▶ Timed Waiting

- ✓ Enters the waiting state for a specified interval of time
- ✓ Transitions back to the runnable state when either the time interval expires or when the event it is waiting for occurs.

▶ Terminated

- ✓ A thread that has been stopped or has completed its task

Thread Priorities

- ▶ Each thread has a priority
 - ✓ Lowest is MIN_PRIORITY (1)
 - ✓ Default is NORM_PRIORITY (5)
 - ✓ Highest is MAX_PRIORITY (10)
- ▶ Priority is used to schedule threads
 - ✓ Higher priority threads get priority in begin able to access processing resources
 - ✓ However, priorities cannot ensure the order of execution
 - ✓ The environment the JVM is executing in has an impact.

Creating Threads

- ▶ There are two basic ways to implement a thread
 - ✓ Extending the Thread class
 - ✓ Implementing the Runnable interface
- ▶ Every program has at least one thread
 - ✓ Called the main thread
 - ✓ Created when a program begins
 - ✓ Additional threads are created from the main thread

Thread Class

- ▶ Has two methods we generally want to override
 - ✓ The run() method that actually executes the thread
 - ✓ The start() method that initializes the thread
 - ✓ Just calling the run() method runs the code in the main thread
 - ✓ The start() method sets up the thread environment
- ▶ By default
 - ✓ The run() method doesn't do anything
 - ✓ The start() methods calls the run() method
 - ✓ The run() method needs to be overridden

Thread Class Methods

Method	Meaning
final String getName()	Obtains a thread's name.
final int getPriority()	Obtains a thread's priority.
final boolean isAlive()	Determines whether a thread is still running.
final void join()	Waits for a thread to terminate.
void run()	Entry point for the thread.
static void sleep(long <i>milliseconds</i>)	Suspends a thread for a specified period of milliseconds.
void start()	Starts a thread by calling its run() method.

Extending Thread

```
public class runner {  
    public static void main(String[] args) {  
        MyThread m = new MyThread();  
        m.start();  
    }  
}  
  
class MyThread extends Thread {  
    public void run() {  
        System.out.println("This is my thread");  
    }  
}
```

Runnable Interface

- ▶ Classes that implement the Runnable Interface can be made into threads
 - ✓ The interface marks a unit of code that can be encapsulated in a Thread object
 - ✓ The implementing class must override the run() method
- ▶ This is the more common way to implement threads
 - ✓ An object implementing the Runnable interface is created
 - ✓ It is then “wrapped” in a Thread object

Implementing Runnable

```
public class RunThreads {  
  
    public static void main(String[] args) {  
        MyThread m = new MyThread();  
        Thread t = new Thread(m);  
        t.start();  
    }  
}  
  
class MyThread implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("This is the running thread");  
    }  
}
```

The Resource Problem

- ▶ Java threads have to be mapped to OS threads:
 - ✓ Each thread takes up 1MB of memory overhead
 - ✓ Has to be scheduled on a core like any other process
 - ✓ If we only have four cores, only four thread can be run at the same time
 - ✓ More threads than cores means that we have to schedule thread on a core
 - ✓ We have memory limitations and processor limitations on threads
- ▶ The more threads we have, the more overhead and competition for resources we will have

The Problem

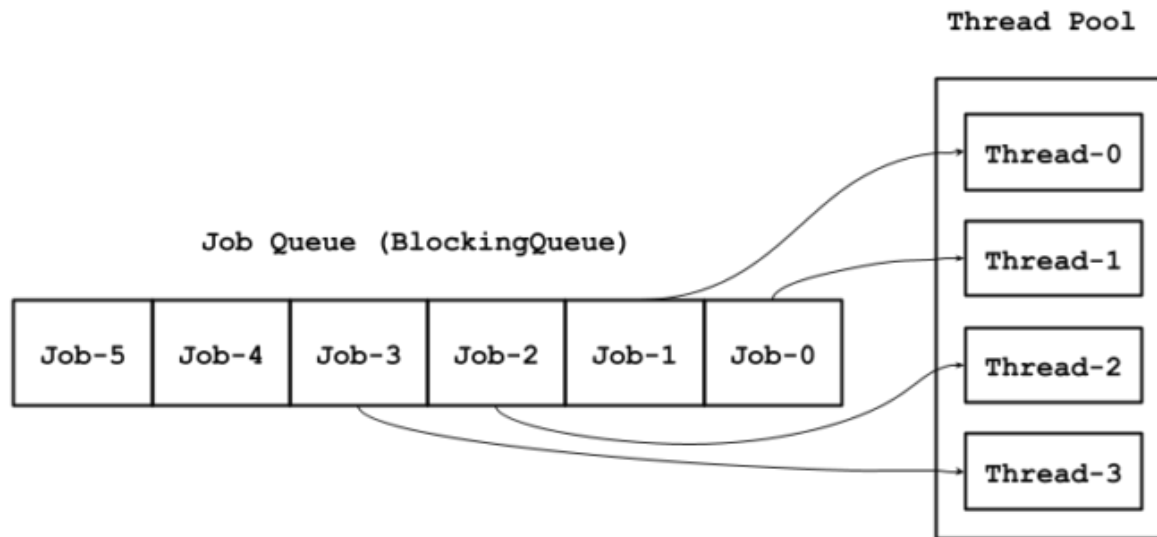
- ▶ Threads are often used for task that are:
 - ✓ Short in duration
 - ✓ Called very frequently
- ▶ The problems with managing any kind of resource with these characteristics are
 - ✓ The amount of time spent creating and shutting down threads starts to become significant – the system starts to “thrash” trying to manage the threads
 - ✓ Too many threads can cause out of memory issues

Pooling Resources

- ▶ A resource pool is a collection of pre-created resources that are reusable
 - ✓ Standard architectural pattern
 - ✓ Flyweight design pattern
- ▶ A thread pool reuses previous created threads
 - ✓ Delay introduced by thread creating is eliminated
 - ✓ Reduces thread life-cycle overhead and thrashing
 - ✓ Allows programmer to concentrate on the executable code instead of thread management

Java Thread Pools

- ▶ A number of threads are created when the application starts
 - ✓ When a request comes in via a job queue, it is allocated to a thread which performs the task.
 - ✓ When the task is finished, the thread is returned to the pool



Pooling Resources

- ▶ A resource pool is a collection of pre-created resources that are reusable
 - ✓ Standard architectural pattern
 - ✓ Flyweight design pattern
 - ✓ Separates the thread code from the “business” or functional code
- ▶ A thread pool reuses previous created threads
 - ✓ Delay introduced by thread creating is eliminated
 - ✓ Reduces thread life-cycle overhead and thrashing
 - ✓ Allows programmer to concentrate on the executable code instead of thread management

Thread Pools

- ▶ The idea of running threads is broken down into two parts
 - ✓ The “task” (often called a job) or the code that would be in the run() method
 - ✓ The “executor” or the actual thread object that runs the runnable tasks
- ▶ We delegate the process of running the task in a thread to the thread pool object

Executor Service

- ▶ Java provides a concurrency library which supports a built-in Java thread pool
- ▶ Implemented as three interfaces
 - ✓ An Executor interface that provides a replacement to the standard thread syntax.
 - `(new Thread(runnablecode)).start()` can be replaced by
 - `e.execute(runnablecode)` where e is an instance of Executor
 - ✓ ExecutorService interface extends the Executor interface to include a submit method for call-able which return a value
 - ✓ ScheduledExecutorService which adds methods to allow scheduling of threads

Using the ExecutorService

- ▶ Start by allocating a Thread pool using one of the constructors or a factory
 - ✓ The following code implements a fixed size Executor service
 - ✓ The shutdown() message
 - Stops the service from accepting new tasks
 - Shuts the service down when all the executing threads have exited

```
public static void main(String[] args) {  
    // Creates a new Thread Pool with 3 executors  
    ExecutorService myPool = Executors.newFixedThreadPool(3);  
  
    // Shuts the pool down once all the threads have terminated  
    myPool.shutdown();  
}
```

Submit a Task

- ▶ Once the pool is started
 - ✓ Runnable tasks are submitted via the `execute()` method
 - ✓ Execute essentially queues up the task, and when a thread is available, passes the task to the thread the effectively executes the `start()` method on it

```
// Creates a new Thread Pool with 3 executors

ExecutorService myPool = Executors.newFixedThreadPool(3);
for (int i = 1; i < 5; i++) {
    myPool.execute(new MyTask("Task " + i));
}
// Shuts the pool down once all the threads have terminated
myPool.shutdown();
```


Customized Executor

- ▶ We can also create our own service with customized parameters
 - ✓ Core thread – the number of threads to start with
 - ✓ Max threads – the number of threads that can be created
 - ✓ Keep alive – the amount of time to keep an executor running when idle
 - ✓ Time units – the time units used to measure the keep alive
 - ✓ BlockingQueue – the queue object to be used by the pool

Customized Executor

- ▶ We can also create our own service with customized parameters
 - ✓ Core thread – the number of threads to start with
 - ✓ Max threads – the number of threads that can be created
 - ✓ Keep alive – the amount of time to keep an executor running when idle
 - ✓ Time units – the time units used to measure the keep alive
 - ✓ BlockingQueue – the queue object to be used by the pool
- ▶ There are a number of other features of an Executor service that can be customized.

Submit a Task

- ▶ Once the pool is started
 - ✓ Runnable tasks are submitted via the `execute()` method
 - ✓ Execute essentially queues up the task, and when a thread is available, passes the task to the thread the effectively executes the `start()` method on it

```
public static void main(String[] args) {  
  
    int corePoolSize = 3;  
    int maxPoolSize = 5;  
    long keepAliveTime = 3000;  
    BlockingQueue<Runnable> pool = new ArrayBlockingQueue<Runnable>(100);  
  
    ExecutorService myPool = new ThreadPoolExecutor(  
        corePoolSize, maxPoolSize, keepAliveTime, TimeUnit.MILLISECONDS, pool);  
}
```

Questions

