Programming in Java

1. Multi-Threading 1

Java Multi-Threading

- As Java has evolved, there have been several versions of how Java implements multi-threading
  - In this module, we will look at the classic model
  - This is not used anymore for new Java code
  - But you will see it if you need to work with Java legacy code
- Knowing how this code works provides a conceptual foundation for the more modern concepts introduced in a subsequent module

Basic Definitions

- Agent
  - Anything capable of executing a task – person, CPU, host, etc.
- Multi-tasking
  - One agent working on more than one task over a period of time
  - Only one task is being executed at any given moment in time
  - The agent switches between tasks according some strategy
- Preemptive time slicing
  - Most common strategy for multi-tasking in operating systems
  - A time period is divided into equal time slices (e.g. 1 second is divided into 1000 slices)
  - Each task gets exclusive access to system resources for one time slice
  - Then it is “preempted” and forced to give up the resources so another task can run
  - Higher priority tasks get more slices (more turns to use resources) rather than longer slices
  - Necessary to prevent any task from locking up the system

Multitasking versus Parallel Processing

- Working from home with multiple tasks needing to be done
  - Cooking dinner, writing code, doing laundry, grocery shopping, etc
  - You have to switch between tasks – but you are the only agent doing these tasks
  - You will use some sort of switching strategy to move from task to task
  - You need to switch environments (kitchen, store, office) to switch between tasks
  - This is concurrent processing, tasks are being done concurrently by switching from one to another in some sort of order
- Parallel Processing
  - A set of agents are available
  - Each task can be executed exclusively by one agent
  - Multi-tasking without having to switch tasks
  - Working from home with a staff
    - *My chef cooks dinner. My housekeeper does the laundry My chauffeur does the grocery shopping*
    - *I just write code*

Multiprocessing versus Multi-threading

- Multiprocessing
  - Each task is self-contained in an isolated environment
  - Switching tasks requires saving the executing environment and loading the new task's environment
  - Can be very slow. Often called heavyweight multi-tasking
- Recall multi-tasking example
  - To switch from coding to grocery shopping to cooking dinner requires a complete change environment
- Multi-threading
  - Often called lightweight multi-processing
  - A group of related tasks in the same environment are called threads
  - The agent can switch between tasks without having to change the environment
  - The agent just switches from task to task inside the same environment

Java Multi-Threading

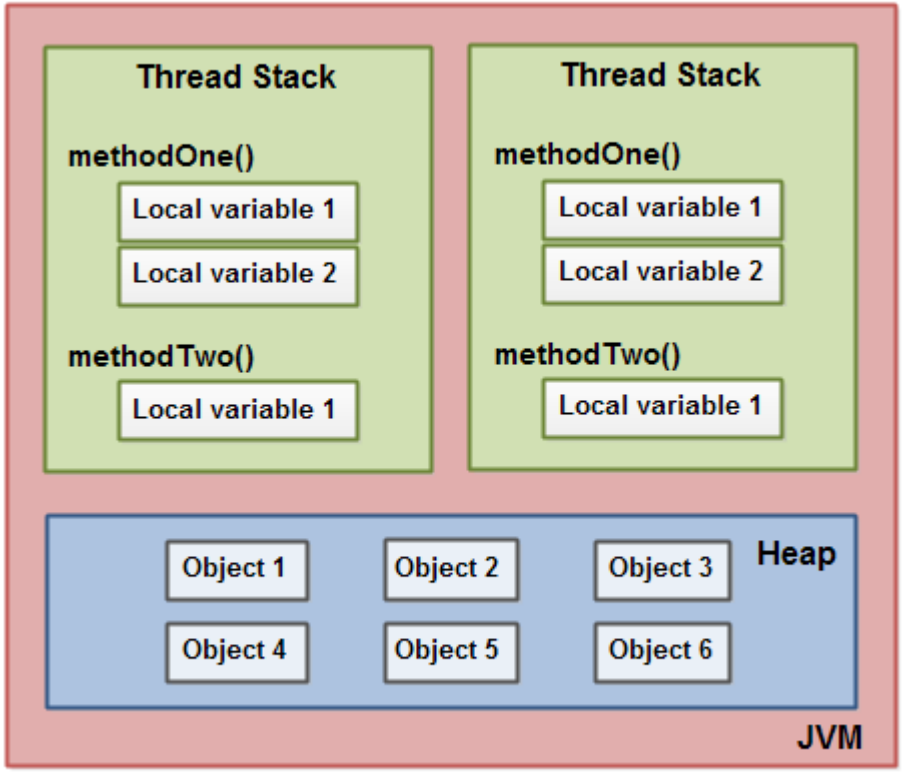
- Concurrency is doing more than one thing at a time
- With modern multi-core architectures, concurrency is a fundamental construct in almost all programming languages
- Java was designed as a multi-threaded programming language
  - Remember that Java runs on the JVM
  - The JVM is a multi-threaded environment 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 multi-threading
- 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

Java Multi-Threading

- Threads
  - All peer threads are treated as a single process because they run in the same execution process space at the OS level
  - Each thread has its own stack memory and registers
  - Peer threads share heap memory and instruction space
  - Threads can share data through shared heap memory
  - Threads are lightweight
  - Threads are faster to start up and shut down
- Threads are defined by the user while processes are defined by the OS

Java Multi-Threading Memory

- Thread Stack
  - Each thread keeps track of its current execution state
  - This means function calls and local variables
  - When swapped out, these all have to be saved
  - And then restored when the thread resumes executing
  - By having its own stack, the execution environment (variables, function calls etc) will not be corrupted



https://jenkov.com/images/java-concurrency/java-memory-model-2.png

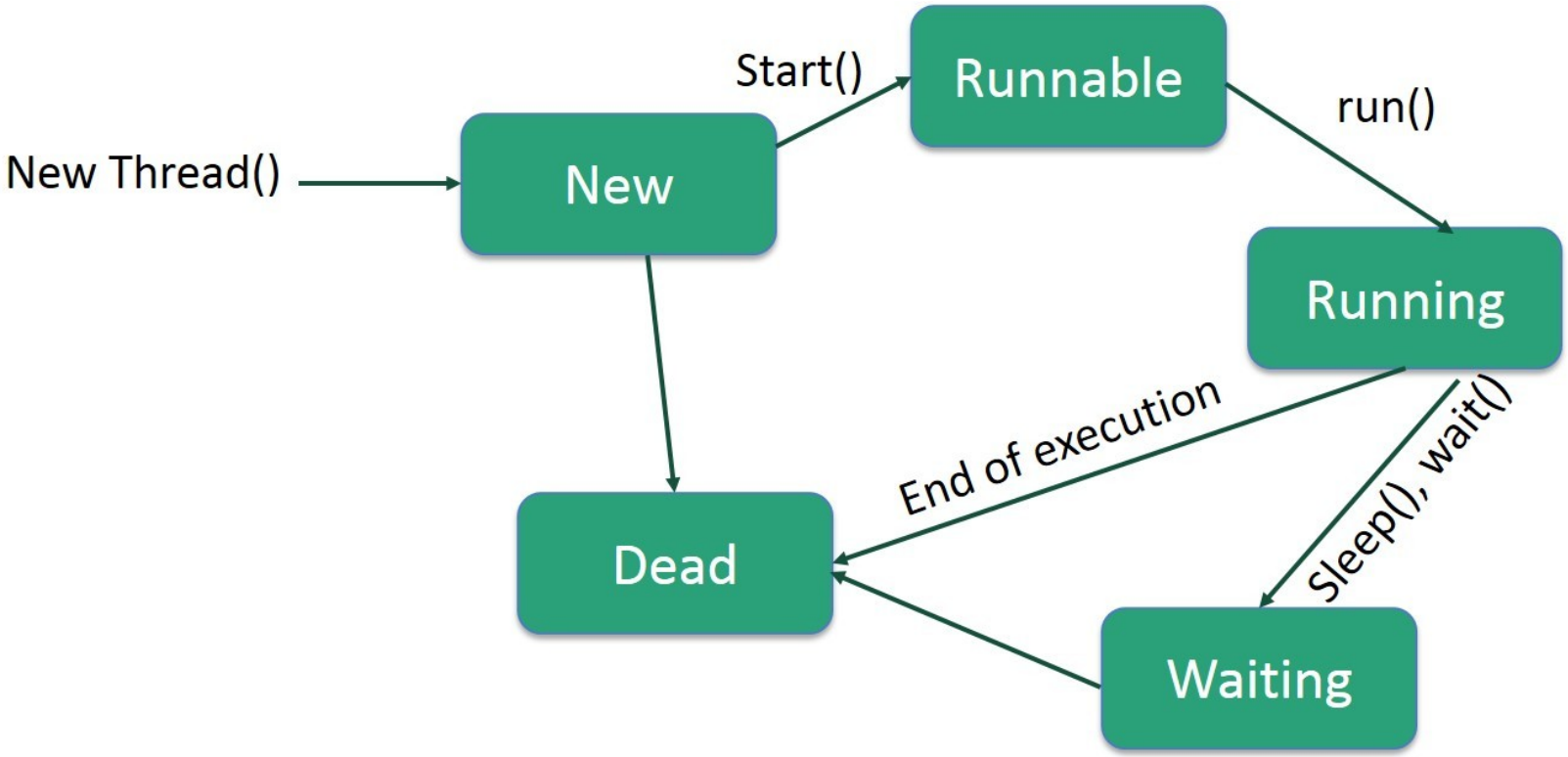
Java Threads

- The Java Thread class is used to create and manage Java threads
- A Java Thread is a single sequence of execution – it is the smallest unit of processing that can be scheduled for a time slice
- A Java Thread Group is a group of threads that are managed as a unit
  - We often want to start and stop all the threads in a group together
- *User* threads are defined by the user
  - The *main()* method creates a user thread when it starts
  - If no user threads are running, the JVM exits
- *Daemon* threads perform background tasks
  - Memory management, garbage collection, etc
  - A JVM exits if only daemon threads are running

The Thread Class

- The Thread class has all of the functionality to create and run a thread
  - We create threads by extending the Thread class or instantiating it directly
- The *start()* method
  - Initializes the thread object so that it can be scheduled to run as a task
  - Once the thread is started, the code in the *run()* method is executed
- The *run()* method
  - Contains the code we want the thread to execute
  - We are limited to a single method that returns void and takes no arguments
- The *x.join()* method
  - Waits until the thread x stops before running
  - Used to synchronize threads that have to work together

Thread States



Thread States

- New
  - Initial state of a thread
  - The thread stays in this state until the JVM schedules the thread
- Runnable
  - After the thread is scheduled, it can be executed and is considered runnable
- Waiting
  - When waiting for another thread to complete a task (the join() operation)
  - Thread switches to runnable when it receives a signal that it can proceed
- 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 killed or has completed its task

Using Threads

- There are two basic ways to implement a thread
  - Extending the Thread class
  - Implementing the Runnable interface
- Extending Thread
  - Thread has an empty implementation of the run() method
  - When we extend Thread, we provide an implementation of the run() method
- The Runnable Interface has a single `void run();` method defined
  - When a class implements the the Runnable Interface it is said to be runnable
  - A runnable object is passed to a Thread object in the Thread object’s constructor
  - A runnable object provides an implementation of the `run();` method

Lab 1-1

Creating Threads

Thread Attributes

- Threads have a number attributes that we can access
  - Name: we can set a printable name for a thread, or let the JVM generate one (like “Thread-0”)
  - ID: A unique id for the thread object
  - State: The current state of the thread
  - Daemon: Whether the thread is daemon or not
- 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 more turns to run
  - However, priorities cannot ensure any specific order of execution
  - The environment the JVM is executing in has an impact on order of execution

Thread Attributes

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 *milliseconds*)	Suspends a thread for a specified period of milliseconds.
void start( )	Starts a thread by calling its **run( )** method.

- The JVM will not exit until all the user threads have completed
- Throwing an exception in one thread does not cause other threads to stop
- Threads spawned by the main thread will continue if the main thread exits

Thread Attributes

- The JVM will not exit until all the user threads have completed
- Throwing an exception in one thread does not cause other threads to stop
- Threads spawned by the main thread will continue if the main thread exits

Lab 1-2

Thread Properties

Race Conditions

- Objects created by a runnable object are only accessible to that object
  - The object is created on the shared heap
  - But only the thread's stack has a reference to it
  - No other threads can access the object because they don't have a reference to it.
- If the object exists on the heap prior to the creation of the runnable object
  - Then the runnable object can be passed a reference to the object
  - Then all references in the threads resolve to the same object
  - Even though each thread has its own copy of the reference, they all point to the same object
- This is not problem when none of the threads are modifying the object
  - But when one or more threads are modifying the object, it can create something called a race condition

Race Conditions

- Modifying or writing data in a shared resource may result in corruption
  - Result of how the threads are interleaved
  - Some instructions from one thread may overlap with instructions from and another thread
- The problem is that because race conditions are environment dependent
  - They are hard to test for because they are not predictable
  - They may be transient (not occur every time)
- Consider a bank account
  - The balance is $100
  - Thread A is going to update the balance by adding $10
  - It reads the balance from the bank account, but then has to yield to thread B
  - Thread B updates the balance to $500, then yields back to thread A
  - Using the value it had previously read from the account, it updates the balance to $110
  - The balance is now incorrect

Mutex

- Mutex (short for *mut*ual *e*xclusion) is a programming language primitive
  - Used in concurrent programming to prevent multiple threads from accessing a shared resource at the same time.
  - A mutex is a locking mechanism that ensures only one thread can access a critical section (shared resource or block of code) at any given time.
  - When a thread locks a mutex, other threads attempting to acquire it are blocked until the mutex is released.
    - *Exclusive access: Only one thread can hold the mutex at a time.*
    - *Blocking behavior: Other threads trying to acquire the mutex are blocked until it's available.*
    - *Used to prevent race conditions: Ensures data consistency in multithreaded environments.*
- Mutex code can break
  - If a thread locks a resource and then fails to release it when it’s finished using it
  - Or a thread terminates before it can release the mutex

Java Synchronized Keyword

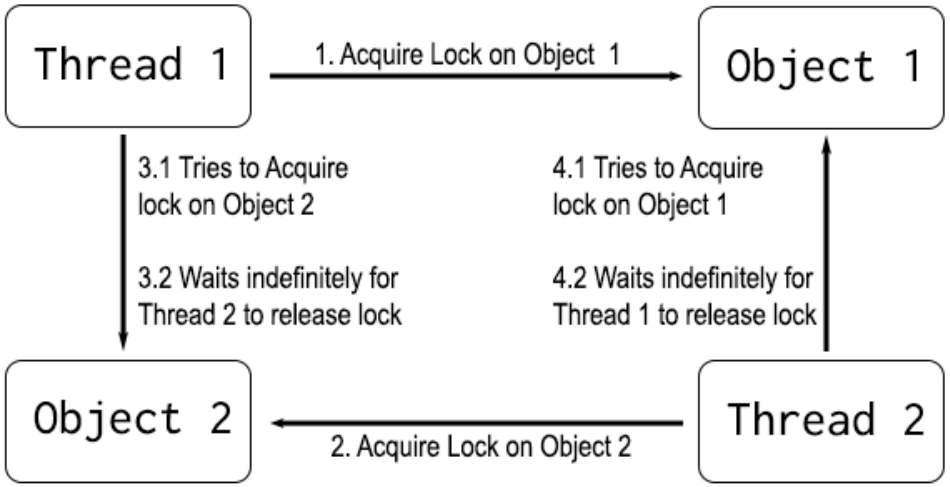
- Used to lock an object or method
  - Allowing only one thread at a time to execute the synchronized code block or method.
  - Essentially a mutex managed by Java
  - When a thread begins executing a synchronized method or block
  - All other threads are blocked from executing it until the thread exits the synchronized code
- The lock is released automatically
  - When the thread exits the block or method
  - Or if the thread terminates

```
public class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }
}
```

```
public class Counter {
    private int count = 0;
    private final Object lock = new Object();

    public void increment() {
        synchronized (lock) {
            count++;
        }
    }
}
```

Deadlocks

- Can occur when using a mutex
  - Can occur when using synchronized
- Thread 1 gets a lock on object 1
  - But can't release the lock until it gets access to object 2
- Thread 2 gets a lock on object 2
  - But can't release the lock until it gets access to object 1
- Each thread is waiting for the other to release their lock



Lab 1-3

Synchronized Threads

Semaphores

- The problem with using the synchronized keyword is that theads block waiting for their turn
  - This can create slowdowns in processing
  - The threads are idle while waiting to use the synchronized resource
- A semaphore is an integer variable shared by the threads
  - Semaphores regulate access to a shared resource
  - Eliminates busy waiting
  - Can be binary taking the values 0 and 1
  - Or can be a count of available resources
- Used to control access to
  - A limited resource, like data base connections
  - Access to a shared resource, usually a binary semphore

Java Semaphores

- There are several ways to use semaphore
  - aquire(): Tries to get a permit, blocks until one is available
  - tryAquire(): Tries to get a permit
    - *Returns true if successful, false if no permit is available*
    - *Does not block if it can't get a permit*
  - tryAcquire(timeout, unit): Blocks for a specific period of time
    - *Used when the code can wait for a specific period of time before resuming*
- When no longer needed, a permit is released
  - Uses the release() method

Lab 1-4

Semaphores

Atomic Access

- An atomic action executes as a single unit of execution.
  - It cannot stop in the middle or be interrupted
  - it either happens completely, or it doesn't happen at all
  - No side effects of an atomic action are visible until the action is complete
- In Java
  - All primitive types generally use atomic reads and writes except for longs and doubles
  - Atomic access is not guaranteed unless the variable is declared volatile
- Volatile variables
  - Threads may cache variables locally, volatile forces write to main memory
  - One thread might not see the most recent value cached by another thread.
  - Without volatile, changes to a shared variable might not be visible immediately to other threads.
  - With volatile, writes by one thread are visible to all other threads immediately.

Atomic Access

- Atomic Variables
  - Java provides atomic classes that support atomic operations on single variables
  - They have get() and set() methods that operate like atomic reads and writes
  - Ensures thread safety when multiple threads are accessing a shared variable
- Concurrent Collection
  - Memory Consistency Errors
    - *If multiple threads are accessing a collection*
    - *We may have two threads trying to add a “last” object into the same position while another thread is reading the last object*
    - *It’s not clear what the “last object” will actually to each thread*
  - Concurrent collections ensure thread access to the collection is sychronized

Atomic Access

- Atomic Variables
  - Java provides atomic classes that support atomic operations on single variables
  - They have get() and set() methods that operate like atomic reads and writes
  - Ensures thread safety when multiple threads are accessing a shared variable
- Concurrent Collection
  - Memory Consistency Errors
    - *If multiple threads are accessing a collection*
    - *We may have two threads trying to add a “last” object into the same position while another thread is reading the last object*
    - *It’s not clear what the “last object” will actually to each thread*
  - Concurrent collections ensure thread access to the collection is synchronized to avoid race conditions.

Atomic Access

- Atomic Variables
  - Java provides atomic classes that support atomic operations on single variables
  - They have get() and set() methods that operate like atomic reads and writes
  - Ensures thread safety when multiple threads are accessing a shared variable
- Concurrent Collection
  - Memory Consistency Errors
    - *If multiple threads are accessing a collection*
    - *We may have two threads trying to add a “last” object into the same position while another thread is reading the last object*
    - *It’s not clear what the “last object” will actually to each thread*
  - Concurrent collections ensure thread access to the collection is synchronized to avoid race conditions.


Lab 1-5
Atomic Access


Java™