Java

Thread

Multitasking & Multithreading

- Multitasking allows several activities to occur concurrently on the computer
- A multithreaded program contains two or more parts that can run concurrently
  - Each part of such a program is called a thread
  - Each thread defines a separate path of execution
- Multithreading is a specialized form of multitasking

Process-based multitasking

- Allows your computer to run two or more programs (processes) concurrently
  - Enables to run the Java compiler at the same time that you are using a text editor or visiting a web site
- Program is the smallest unit of code that can be dispatched by the scheduler
- Java makes use of process-based multitasking environments but no direct control over it

Thread-based multitasking

- Allows parts of the same process (threads) to run concurrently
  - Thread is the smallest unit of dispatchable code
- A single program can perform two or more tasks simultaneously
  - A text editor can format text at the same time that it is printing (if performed by two separate threads)
- Java supports thread-based multitasking and provides high-level facilities for multithreaded programming

Multithreading

- Advantages of multithreading
  - Threads share the same address space
  - Context switching and communication between threads is usually inexpensive
- Java works in an interactive, networked environment
  - Data transmission over networks, read/write from local file system, user input - all slower than computer processing
  - In a single-threaded environment, the program has to wait for a task to finish before proceeding to the next
  - Multithreading helps reduce the idle time because another thread can run when one is waiting

Multithreading in Multicore

- Java’s multithreading work in both single-core and multi-core systems
- In single-core systems
  - Concurrently executing threads share the CPU, with each thread receiving a slice of CPU time
  - Two or more threads do not run at the same time, but idle CPU time is utilized
- In multi-core systems
  - Two or more threads do execute simultaneously
  - It can further improve program efficiency and increase the speed of certain operations

Main Thread

- When a Java program starts up, one thread begins running immediately
- This is called the main thread of the program
- It is the thread from which the child threads will be spawned
- Often, it must be the last thread to finish execution

Main Thread

```java
public class MainThread {
    public static void main(String[] args) {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
        // change the name of the thread
        t.setName("My Thread");
        System.out.println("After name change: " + t);
        try {
            for(int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep( millis: 1000);
            }
        }catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
}
```

sleep() method

- Thread pause is accomplished by the sleep() method
  - The argument to sleep() specifies the delay period in milliseconds
- The sleep() method might throw an InterruptedException
  - It would happen if some other thread wanted to interrupt this sleeping one
- The sleep() method causes the thread from which it is called to suspend execution for the specified period of milliseconds

How to create Thread

1. By extending the **Thread** class
2. By implementing **Runnable** Interface

- Extending Thread
  - Need to override the public void run() method
- Implementing Runnable
  - Need to implement the public void run() method
- Which one is better?

Extending Thread

```java
class NewThread2 extends Thread {
    NewThread2() {
        super( name: "Extends Thread");
        start();
    }
    // This is the entry point for the thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep( millis: 500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

public class ExtendsThread {
    public static void main(String[] args) {
        new NewThread2();
    }
}
```

Implementing Runnable

```java
class NewThread1 implements Runnable {
    Thread t;
    NewThread1() {
        t = new Thread( target: this);
        t.start();
    }
    // This is the entry point for the thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep( millis: 500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

public class ImplementsThread {
    public static void main(String[] args) {
        new NewThread1();
    }
}
```

Other ways

```java
class NewThread3 implements Runnable {
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep( millis: 500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

public class ImplementsThread2 {
    public static void main(String[] args) {
        Runnable r = new NewThread3();
        Thread t = new Thread(r);
        t.start();
    }
}
```

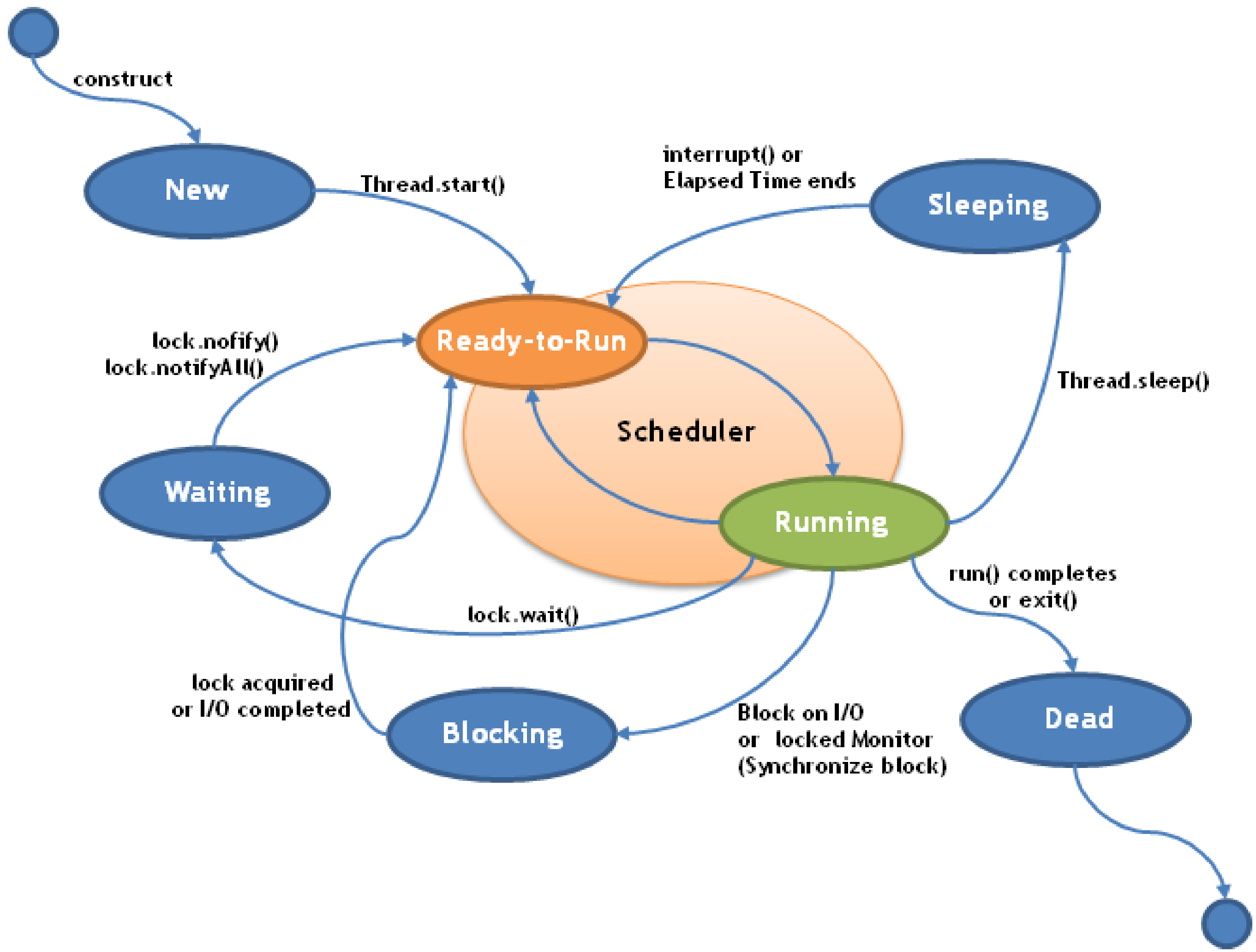
```java
public class CreateThread {
    public static void main(String[] args) {
        CreateThread ct = new CreateThread();
        new Thread(ct::f1, name: "T1").start();
    }

    public void f1() {
        for(int i = 5; i > 0; i--) {
            System.out.println(i);
            try {
                Thread.sleep( millis: 500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}
```

Multiple Threads

- It is possible to create more than one thread inside the main
- In multiple threads, often you will want the main thread to finish last. This is accomplished by
  - using a large delay in the main thread
  - using the **join()** method, this method waits until the thread on which it is called terminates
- Whether a thread has finished or not can be known using **isAlive()** method
- ***Example****: MultipleThreads.java, JoinAliveThreads.java*

Thread States

Source: https://avaldes.com/java-thread-states-life-cycle-of-java-threads/



Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time
- The process by which this is achieved is called synchronization
- Key to synchronization is the concept of the monitor
- A monitor is an object that is used as a mutually exclusive lock
  - Only one thread can own a monitor at a given time

Synchronization

- When a thread acquires a lock, it is said to have entered the monitor
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor
- These other threads are said to be waiting for the monitor
- A thread that owns a monitor can reenter the same monitor if it so desires

Synchronization

- Two ways to achieve synchronization
- Synchronized method

  synchronized void call(String msg) { }
- Synchronized block

  public void run() {

  synchronized(target) { target.call(msg); } }
- ***Example**: NonSynchronized.java, SynchronizedMethod.java, SynchronizedBlock.java, SynchronizedBlock2.java, SynchronizedTest.java*

Synchronized Method

- All objects have an implicit monitor with them
  - To enter an object’s monitor, call a synchronized method
  - All other threads that try to call it (or any other synchronized method) on the same instance have to wait
  - To exit the monitor, the owner returns from the method
- A thread enters any synchronized method on an instance
  - No other thread can enter any other synchronized method on the same instance
  - Non-synchronized methods on that instance will continue to be callable

Synchronized Statement

- Synchronized methods will not work in all cases
  - To synchronize access to objects of a class not designed for multithreading (class doesn’t use synchronized method)
  - No access to the source code, so not possible to synchronized appropriate methods within the class
- How can access to an object of this class be synchronized?
  - Put calls to the methods defined by this class inside a synchronized block

Inter Thread Communication

- One way is to use polling
  - Loop to check some condition repeatedly, wastes CPU time
  - Once the condition is true, appropriate action is taken
- Java includes an elegant inter-thread communication mechanism via the **wait()**, **notify()** and **notifyAll()** methods
- These methods are implemented as final methods in Object, so all classes have them
- All three methods can be called only from within a synchronized method

Inter Thread Communication

- ***wait()***
  - tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll( )
- ***notify()***
  - wakes up a thread that called wait( ) on the same object
- ***notifyAll()***
  - wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access first
- ***Example***: *IncorrectPC.java, CorrectPC.java*

Wait within Loop

- wait() waits until notify() or notifyAll() is called
- In very rare cases the waiting thread could be awakened due to a spurious wakeup
  - A waiting thread resumes without notify( ) or notifyAll() having been called
  - The thread resumes for no apparent reason
  - Java API documentation recommends that calls to wait() should take place within a loop that checks the condition on which the thread is waiting
  - *Best practice is to use wait() within loop and notifyAll()*

Deadlock

- Deadlock occurs when two threads have a circular dependency on a pair of synchronized objects
  - Thread-1 enters the monitor on object X, and Thread-2 enters the monitor on object Y
  - Thread-1 calls any synchronized method on Y; it will block
  - Thread-2 calls any synchronized method on X; it will block
  - Two threads wait forever – to access X, Thread-2 have to release its lock on Y so that Thread-1 could complete
  - If multithreaded program locks up occasionally, deadlock is one of the first conditions to check
- ***Example***: *Deadlock.java*

Suspend, Resume and Stop

- Suspend – ***Thread t; t.suspend();***
  - Locks are not released
- Resume – ***Thread t; t.resume();***
- Stop – ***Thread t; t.stop();***
  - Cannot be resumed later, locks are released
- Methods are deprecated
  - Suspend and stop can cause serious system failures
  - Deadlocks due to unreleased locks of suspended threads
  - Corrupted data structures due to stopping thread
- ***Example***: *SuspendResume.java*

Java Concurrency Utilities *

- The concurrency utilities are contained in *java.util.concurrent*, *java.util.concurrent.atomic*, and *java.util.concurrent.locks* (all in the *java.base*)
- *java.util.concurrent* defines the core features that support alternatives to the built-in approaches to synchronization and interthread communication
  - Synchronizers
  - Executors
  - Concurrent Collections
  - The Fork/Join Framework

Synchronizers *

- Synchronizers offer high-level ways of synchronizing the interactions between multiple threads
- Synchronization objects are supported by:
  - **Semaphore**
  - **CountDownLatch**
  - **CyclicBarrier**
  - **Exchanger**
  - **Phaser**
- Collectively, they enable to handle several formerly difficult synchronization situations with ease

Executors *

- Executor initiates and controls the execution of threads
  - Executor offers an alternative to managing threads through the Thread class
- At the core of an executor is the Executor interface
  - The ExecutorService interface extends Executor by adding methods that help manage and control the execution of threads
  - Java provides Thread Pool implementation with ExecutorService

Thread Pool *

- Thread Pools are useful when you need to limit the number of threads running in your application
  - Performance overhead starting a new thread
  - Each thread is also allocated some memory for its stack
- Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool
  - As soon as the pool has any idle threads the task is assigned to one of them and executed
- Thread pools are often used in multithreaded servers

ExecutorService *

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Running task");
        for (int j = 5; j > 0; j--) {
            System.out.println(j);
        }
    }
}

public class ExecutorServiceTest {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 10);
        for (int i = 0; i < 20; i++) {
            executorService.execute(new MyRunnable());
        }
        executorService.shutdown();
    }
}
```

Callable and Future *

- Runnable cannot return a result to the caller
- **Callable** object allows to return values after completion
- Callable task returns a **Future** object to return result
- The result can be obtained using get() that remains blocked until the result is computed
- Check completion by isDone(), cancel by cancel()
- ***Example***: *CallableFutures.java*

Concurrent Collections *

- java.util.concurrent defines several concurrent collection classes
  - **ConcurrentHashMap**
  - **BlockingQueue**
  - **BlockingQueue** etc.
- **BlockingQueue** can be used to solve the producer-consumer problem
  - No need to use wait(), notify(), notifyAll()
- ***Example***: *PCBlockingQueue.java*

TimeUnit *

- To better handle thread timing, java.util.concurrent defines the TimeUnit enumeration
  - The concurrent API defines several methods that take TimeUnit as argument, which indicates a time-out period
- TimeUnit is an enumeration that is used to specify the granularity (or resolution) of the timing
- It can be one of the following values:
  - DAYS, HOURS, MINUTES, SECONDS, MICROSECONDS, MILLISECONDS, NANOSECONDS
- **TimeUnit.SECONDS.sleep(1)** is same as **sleep(1000)**

Atomic *

- java.util.concurrent.atomic offers an alternative to the other synchronization features when reading or writing the value of some types of variables
  - This package offers methods that compare the value of a variable in one uninterruptible (atomic) operation
  - No lock or other synchronization mechanism is required
- Atomic operations are accomplished through:
- **Classes**: AtomicInteger, AtomicLong
- **Methods**: get(), set(), compareAndSet(), decrementAndGet(), incrementAndGet(), getAndSet() etc.

Lock *

- java.util.concurrent.locks provides support for locks, which are objects that offer an alternative to using synchronized to control access to a shared resource
- The **Lock** interface defines a lock. The methods are:
  - To acquire a lock, call *lock()*. If the lock is unavailable, *lock()* will wait
  - To release a lock, call *unlock( )*
  - To see if a lock is available, and to acquire it if it is, call *tryLock()*. This method will not wait for the lock if it is unavailable, it returns true if acquired and false otherwise

Lock *

- **ReentrantLock** is a lock that can be repeatedly entered by the thread that currently holds the lock
- **ReentrantReadWriteLock** is a **ReadWriteLock** that maintains separate locks for read and write access
  - Multiple locks are granted for readers of a resource as long as the resource is not being written
- The advantage to using these methods is greater control over synchronization
- ***Example***: *SynchronizationLock.java*

The Fork/Join Framework *

- Fork/Join framework supports parallel programming
- It enhances multithreaded programming
  - Simplifies the creation and use of multiple threads
  - Enables applications to automatically scale to make use of the number of available processors
- Recommended approach to multithreading when parallel processing is desired
- **Classes**: ForkJoinTask, ForkJoinPool, RecursiveTask, RecursiveAction
- ***Example***: *ForkJoinTest.java*