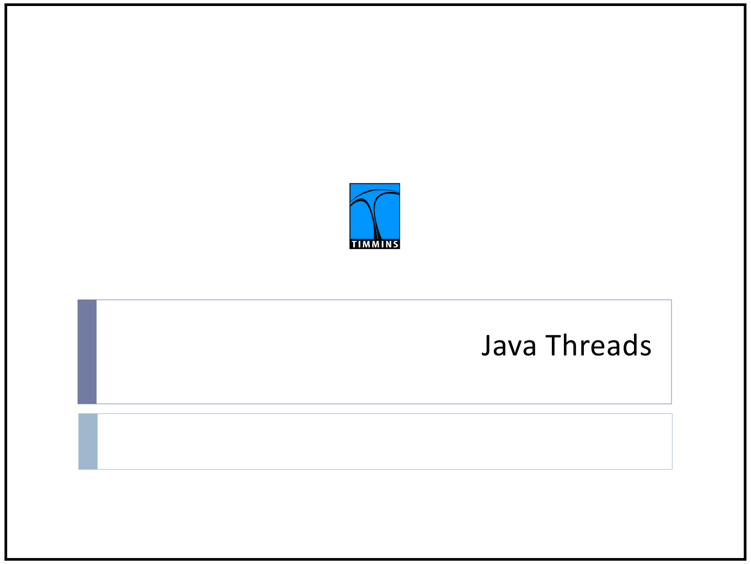
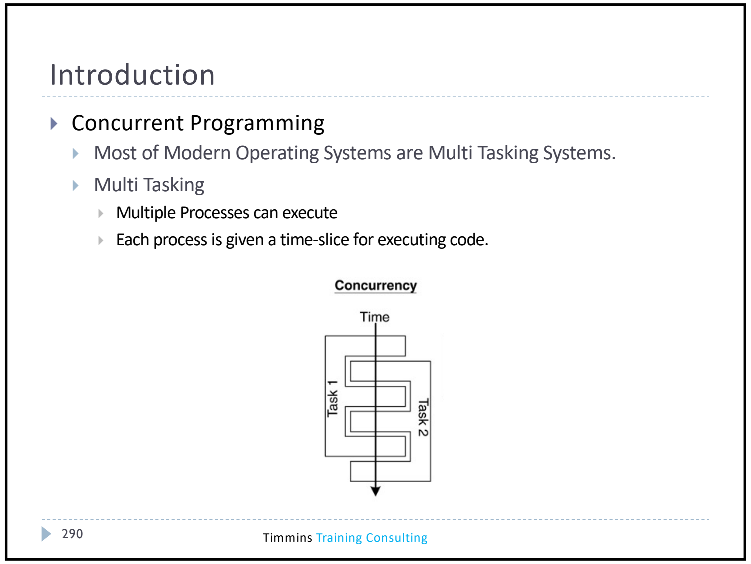


Java Threads

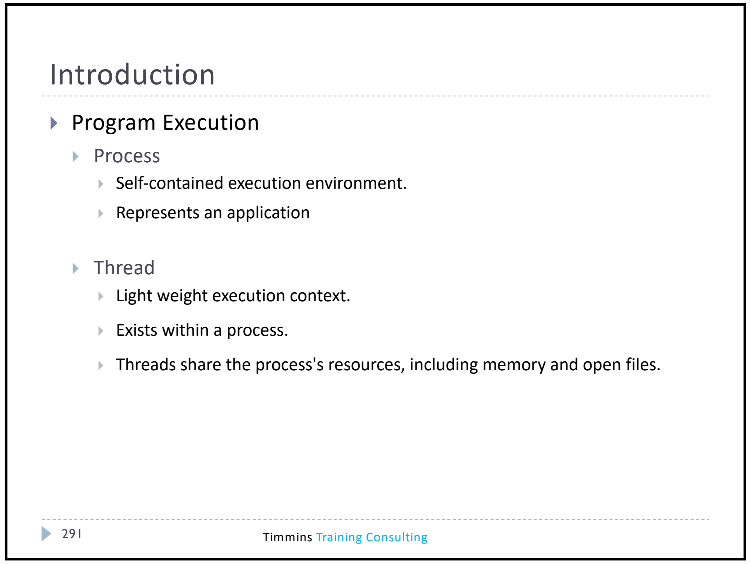
289



Introduction

- Concurrent Programming
  - Most of Modern Operating Systems are Multi Tasking Systems.
  - Multi Tasking
    - Multiple Processes can execute
    - Each process is given a time-slice for executing code.

290

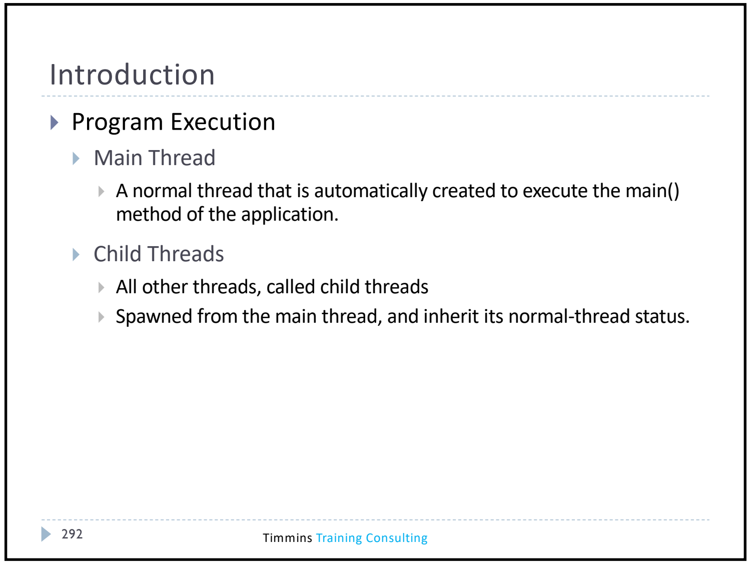


Introduction

- Program Execution
  - Process
    - Self-contained execution environment.
    - Represents an application
  - Thread
    - Light weight execution context.
    - Exists within a process.
    - Threads share the process's resources, including memory and open files.

291 Timmins Training Consulting

291

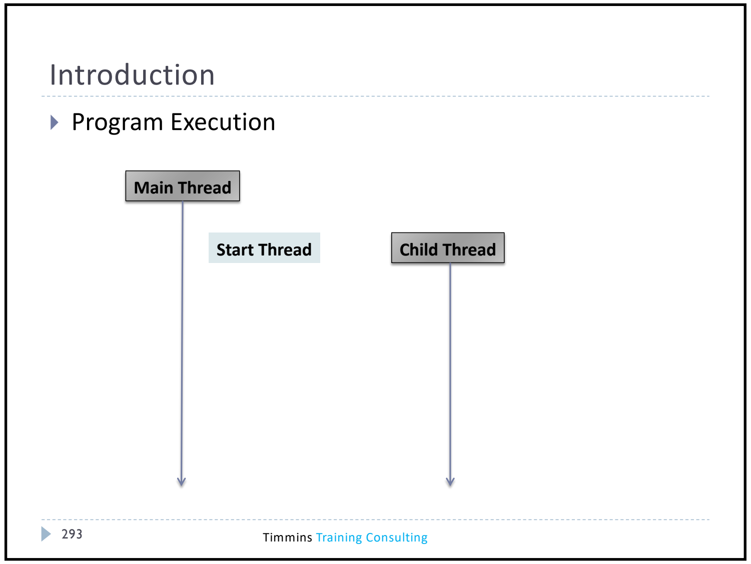


Introduction

- Program Execution
  - Main Thread
    - A normal thread that is automatically created to execute the main() method of the application.
  - Child Threads
    - All other threads, called child threads
    - Spawned from the main thread, and inherit its normal-thread status.

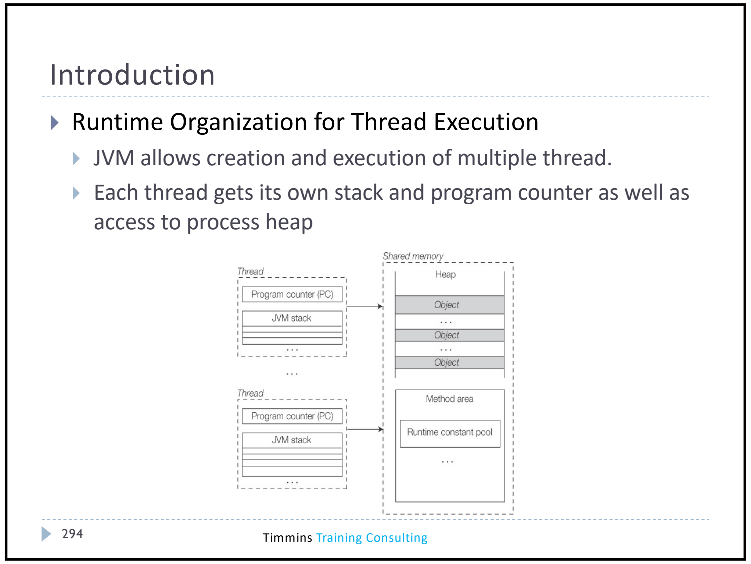
292 Timmins Training Consulting

292



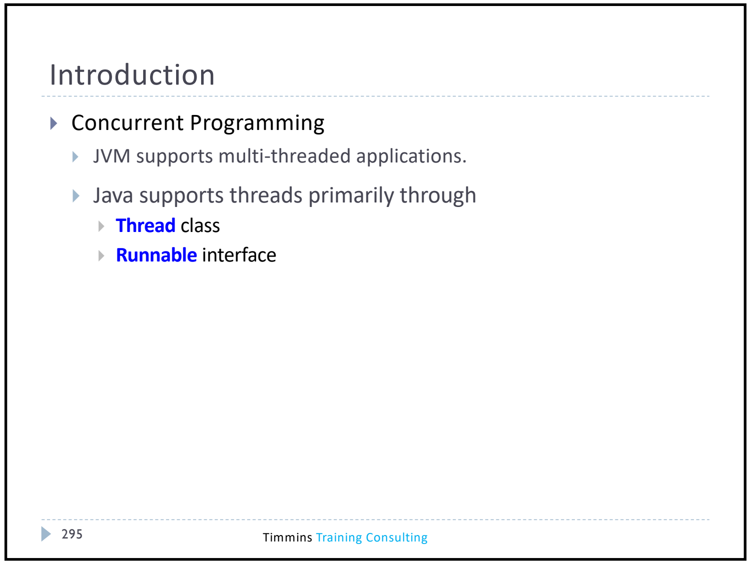
Introduction

- Program Execution



Introduction

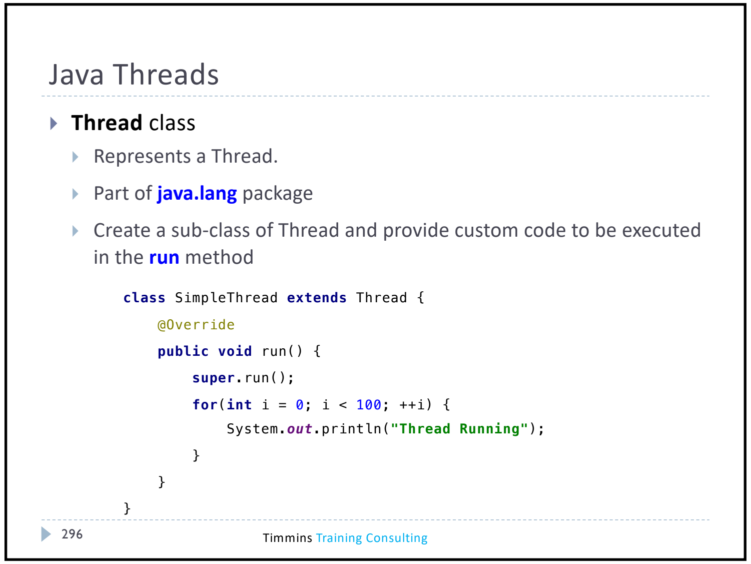
- Runtime Organization for Thread Execution
  - JVM allows creation and execution of multiple thread.
  - Each thread gets its own stack and program counter as well as access to process heap



Introduction

- Concurrent Programming
  - JVM supports multi-threaded applications.
  - Java supports threads primarily through
    - **Thread** class
    - **Runnable** interface

295

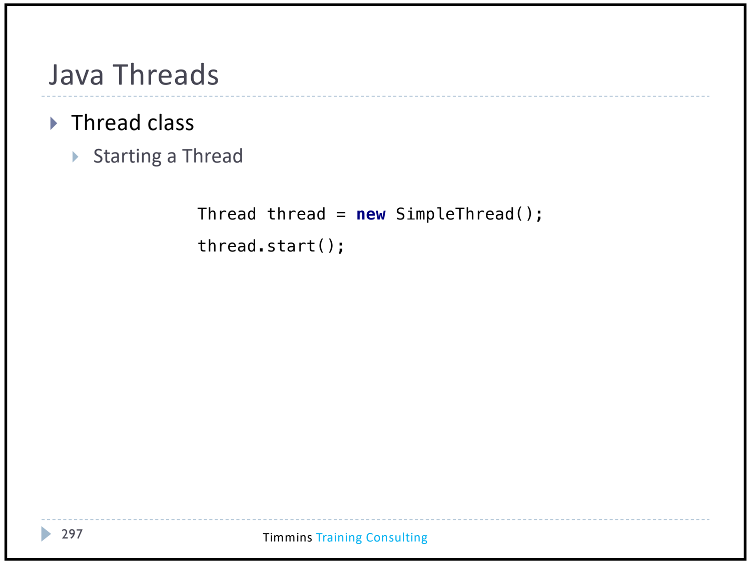


Java Threads

- **Thread** class
  - Represents a Thread.
  - Part of **java.lang** package
  - Create a sub-class of Thread and provide custom code to be executed in the **run** method

```
class SimpleThread extends Thread {
    @Override
    public void run() {
        super.run();
        for(int i = 0; i < 100; ++i) {
            System.out.println("Thread Running");
        }
    }
}
```

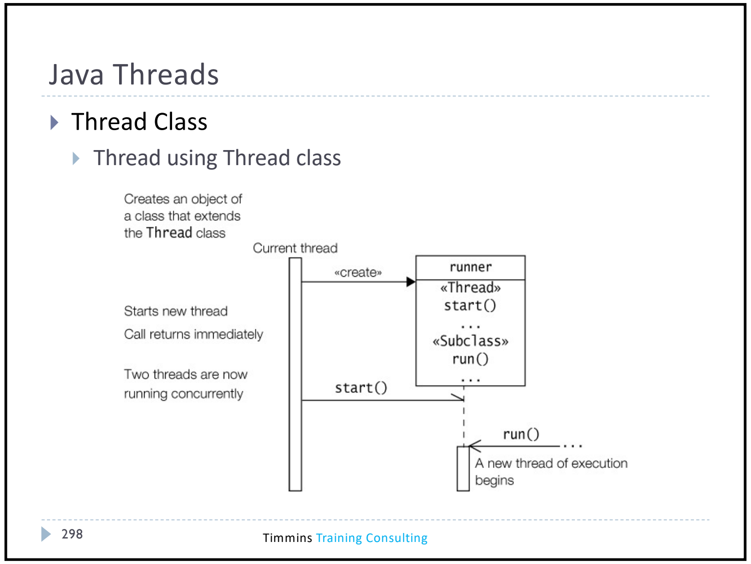
296



Java Threads

- Thread class
  - Starting a Thread

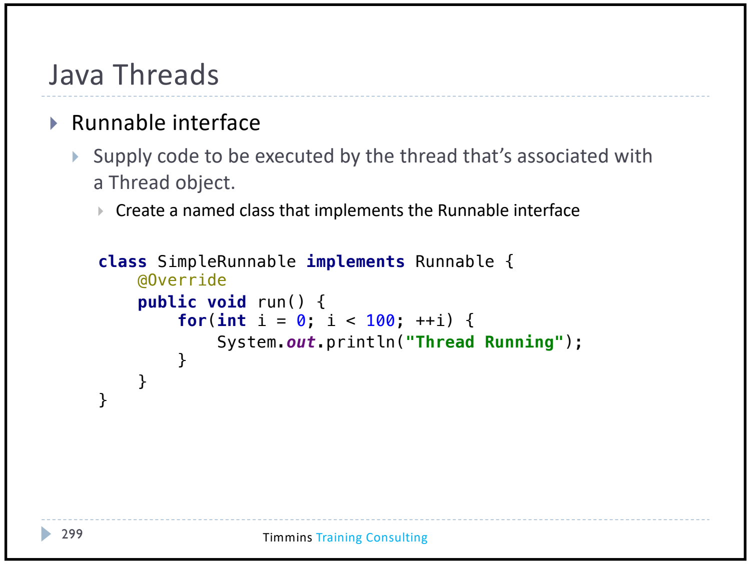
```
Thread thread = new SimpleThread();
thread.start();
```



Java Threads

- Thread Class
  - Thread using Thread class

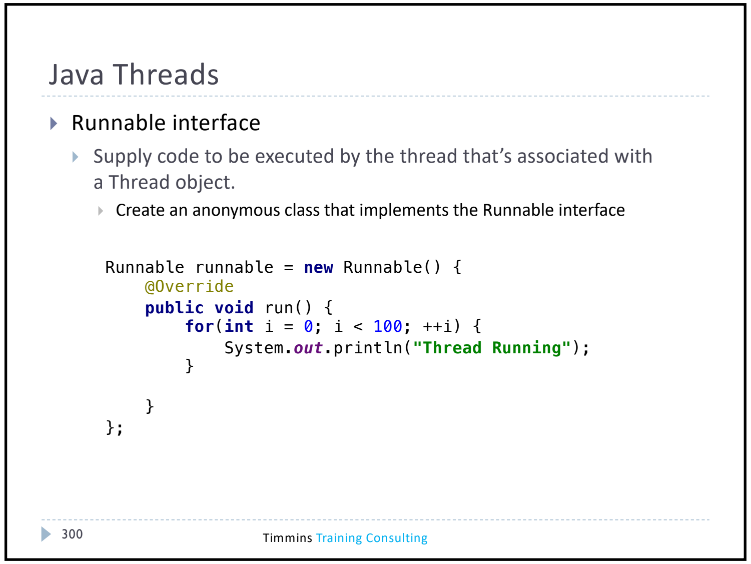
Creates an object of a class that extends the Thread class
Current thread
«create»
runner
«Thread»
start()
...
«Subclass»
run()
...
Starts new thread
Call returns immediately
Two threads are now running concurrently
start()
run()
...
A new thread of execution begins



Java Threads

- Runnable interface
  - Supply code to be executed by the thread that’s associated with a Thread object.
    - Create a named class that implements the Runnable interface

```
class SimpleRunnable implements Runnable {
    @Override
    public void run() {
        for(int i = 0; i < 100; ++i) {
            System.out.println("Thread Running");
        }
    }
}
```

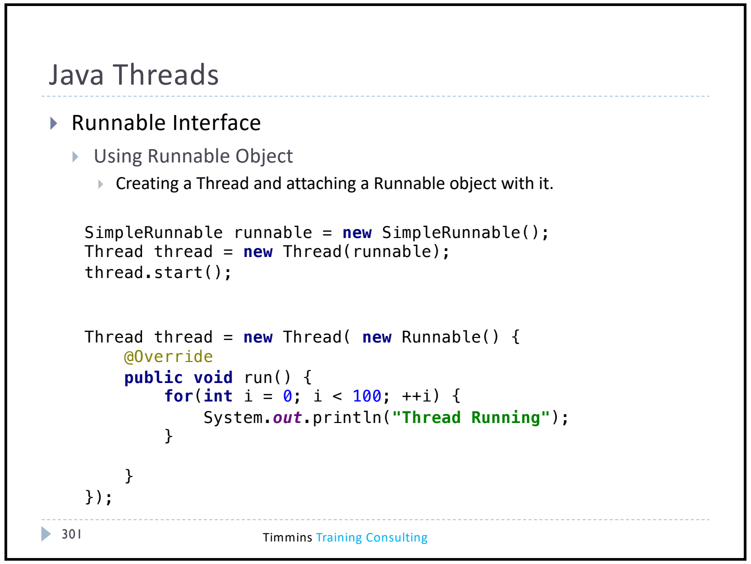


Java Threads

- Runnable interface
  - Supply code to be executed by the thread that’s associated with a Thread object.
    - Create an anonymous class that implements the Runnable interface

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        for(int i = 0; i < 100; ++i) {
            System.out.println("Thread Running");
        }

    }
};
```



Java Threads

- Runnable Interface
  - Using Runnable Object
    - Creating a Thread and attaching a Runnable object with it.

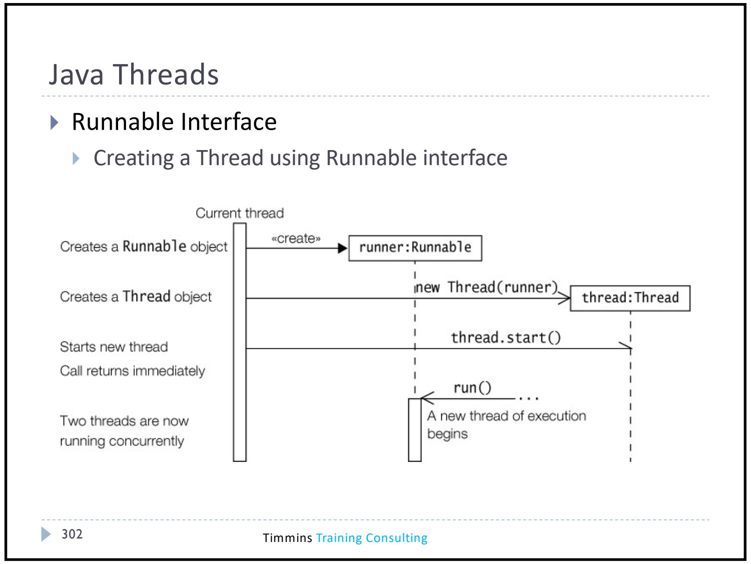
```
SimpleRunnable runnable = new SimpleRunnable();
Thread thread = new Thread(runnable);
thread.start();


Thread thread = new Thread( new Runnable() {
    @Override
    public void run() {
        for(int i = 0; i < 100; ++i) {
            System.out.println("Thread Running");
        }

    }
});
```

301 Timmins Training Consulting

301

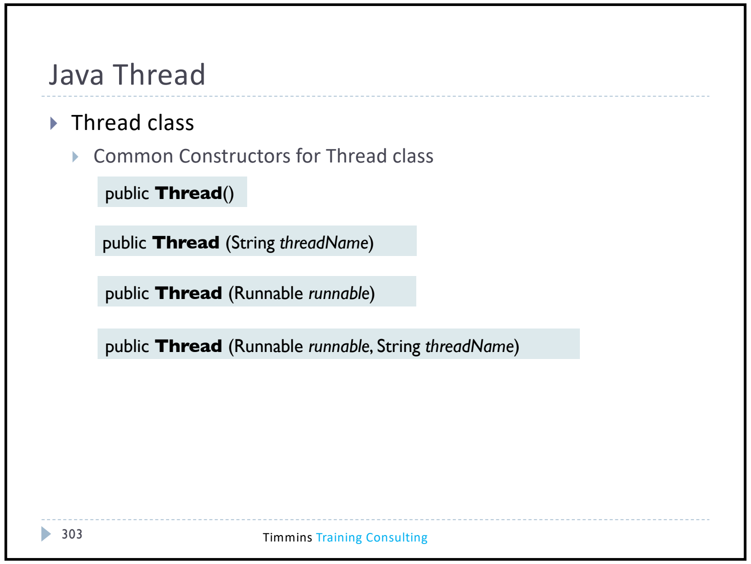


Java Threads

- Runnable Interface
  - Creating a Thread using Runnable interface

302 Timmins Training Consulting

302



Java Thread

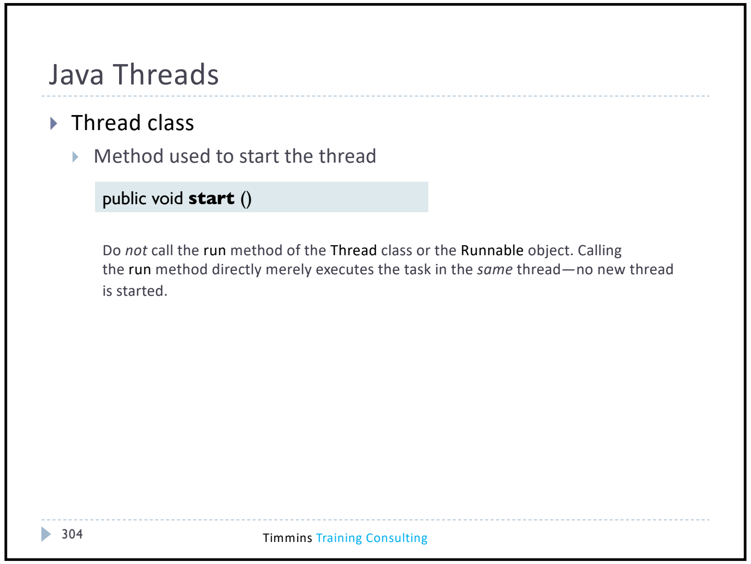
- Thread class
  - Common Constructors for Thread class

```
public Thread()

public Thread (String threadName)

public Thread (Runnable runnable)

public Thread (Runnable runnable, String threadName)
```

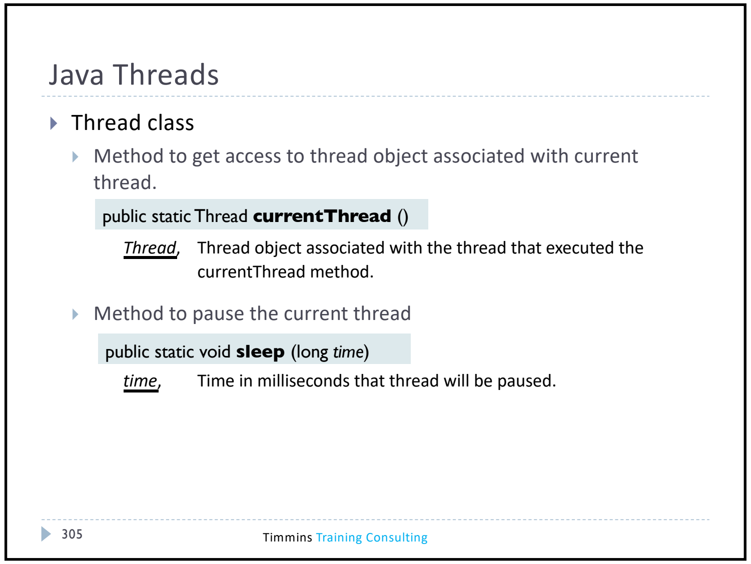


Java Threads

- Thread class
  - Method used to start the thread

```
public void start ()
```

Do *not* call the **run** method of the **Thread** class or the **Runnable** object. Calling the **run** method directly merely executes the task in the *same* thread—no new thread is started.



Java Threads

- Thread class
  - Method to get access to thread object associated with current thread.

public static Thread **currentThread** ()

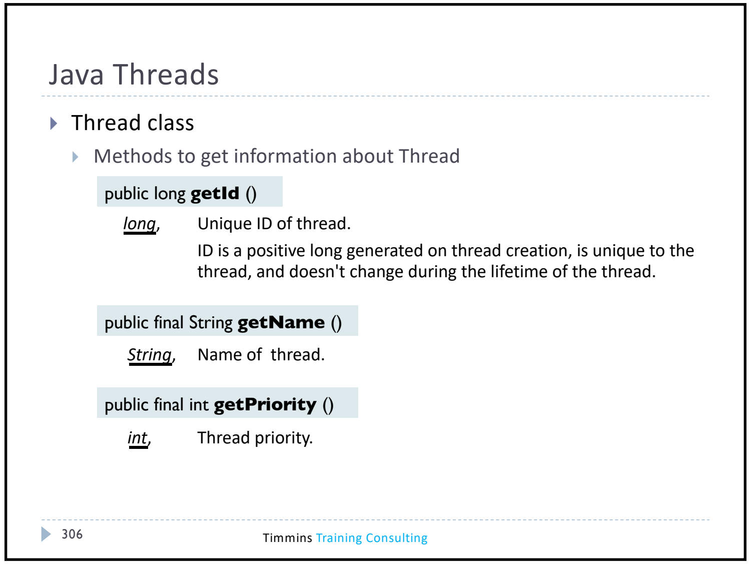
Thread, Thread object associated with the thread that executed the currentThread method.

  - Method to pause the current thread

public static void **sleep** (long *time*)

time, Time in milliseconds that thread will be paused.

305



Java Threads

- Thread class
  - Methods to get information about Thread

public long **getId** ()

long, Unique ID of thread.

ID is a positive long generated on thread creation, is unique to the thread, and doesn't change during the lifetime of the thread.

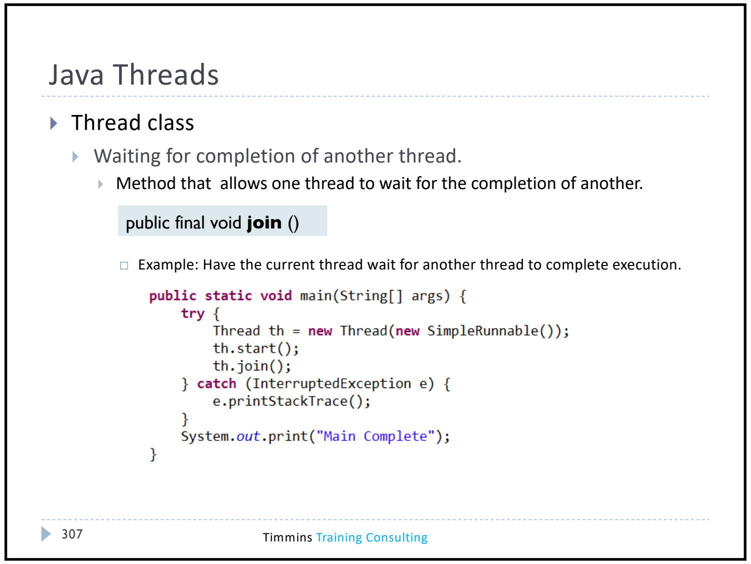
public final String **getName** ()

String, Name of thread.

public final int **getPriority** ()

int, Thread priority.

306



Java Threads

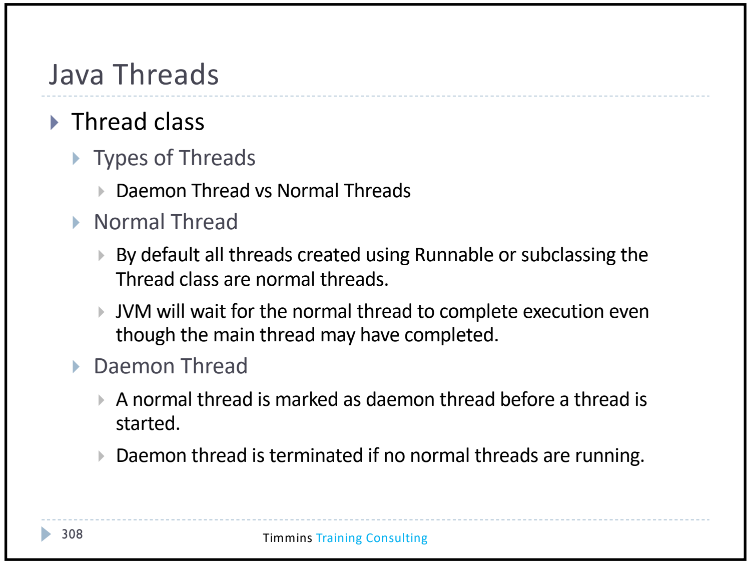
- Thread class
  - Waiting for completion of another thread.
    - Method that allows one thread to wait for the completion of another.

public final void **join** ()

□ Example: Have the current thread wait for another thread to complete execution.

```
public static void main(String[] args) {
    try {
        Thread th = new Thread(new SimpleRunnable());
        th.start();
        th.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.print("Main Complete");
}
```

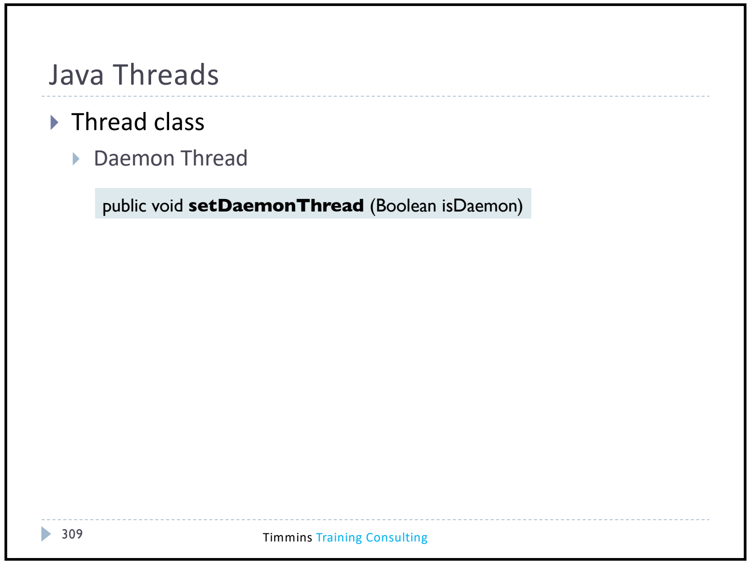
307



Java Threads

- Thread class
  - Types of Threads
    - Daemon Thread vs Normal Threads
  - Normal Thread
    - By default all threads created using Runnable or subclassing the Thread class are normal threads.
    - JVM will wait for the normal thread to complete execution even though the main thread may have completed.
  - Daemon Thread
    - A normal thread is marked as daemon thread before a thread is started.
    - Daemon thread is terminated if no normal threads are running.

308

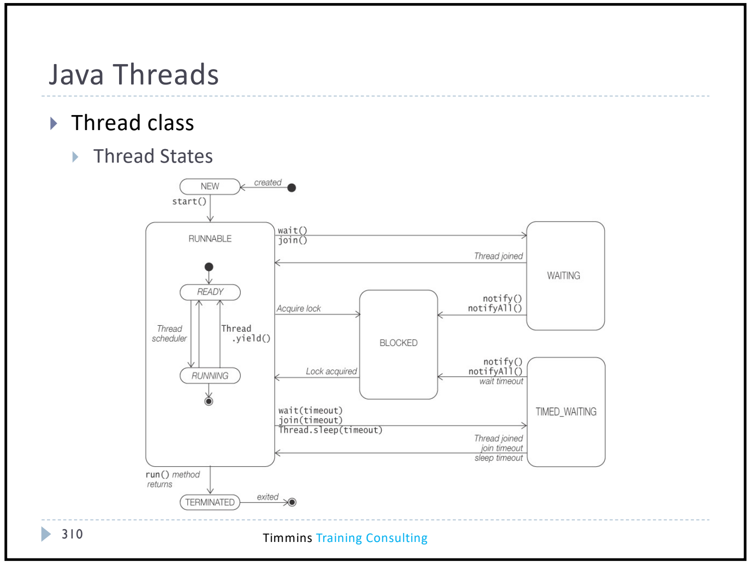


Java Threads

- Thread class
  - Daemon Thread

public void **setDaemonThread** (Boolean isDaemon)

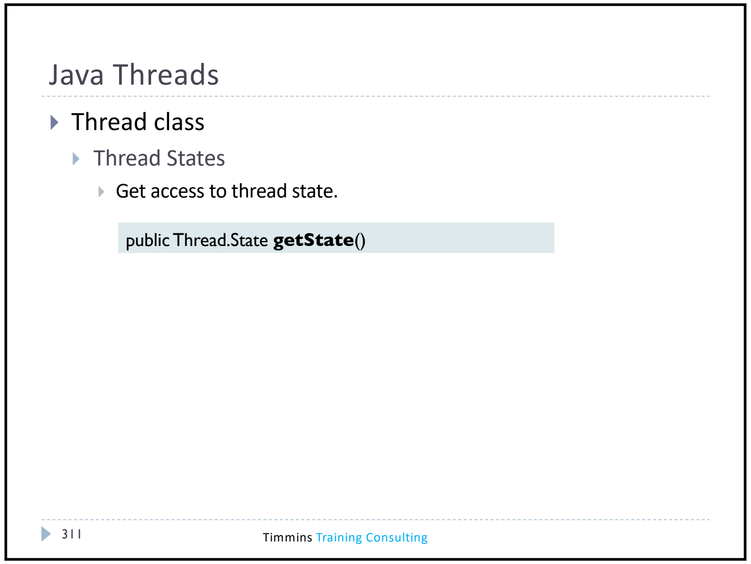
309



Java Threads

- Thread class
  - Thread States

310

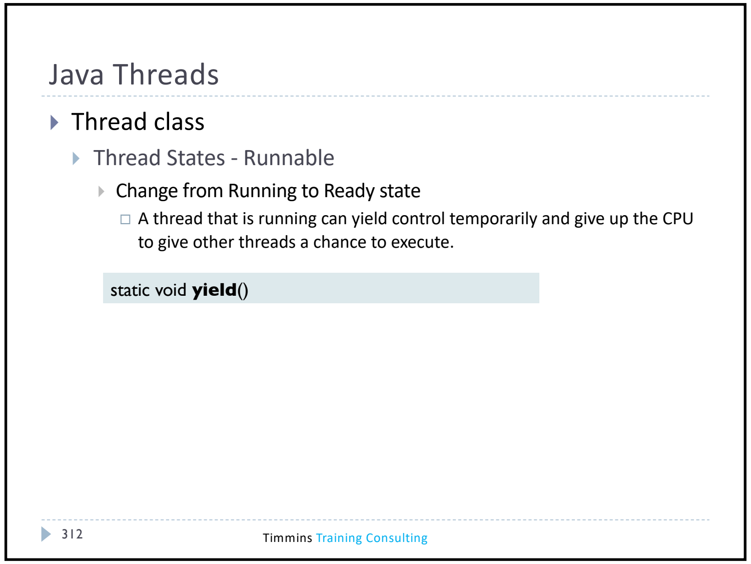


Java Threads

- Thread class
  - Thread States
    - Get access to thread state.

```
public Thread.State getState()
```

311

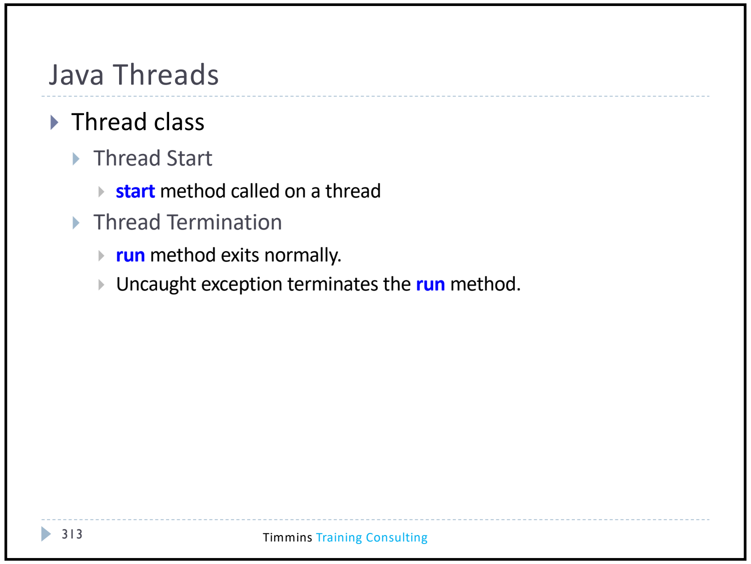


Java Threads

- Thread class
  - Thread States - Runnable
    - Change from Running to Ready state
      - □ A thread that is running can yield control temporarily and give up the CPU to give other threads a chance to execute.

```
static void yield()
```

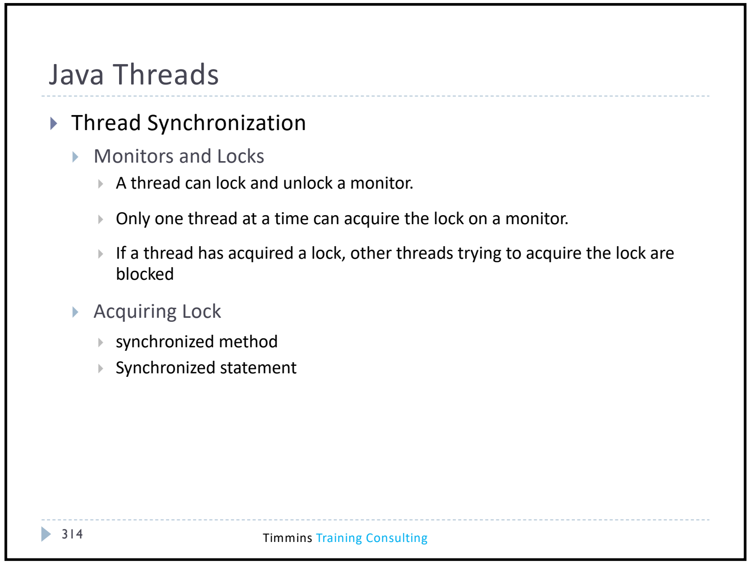
312



Java Threads

- Thread class
  - Thread Start
    - **start** method called on a thread
  - Thread Termination
    - **run** method exits normally.
    - Uncaught exception terminates the **run** method.

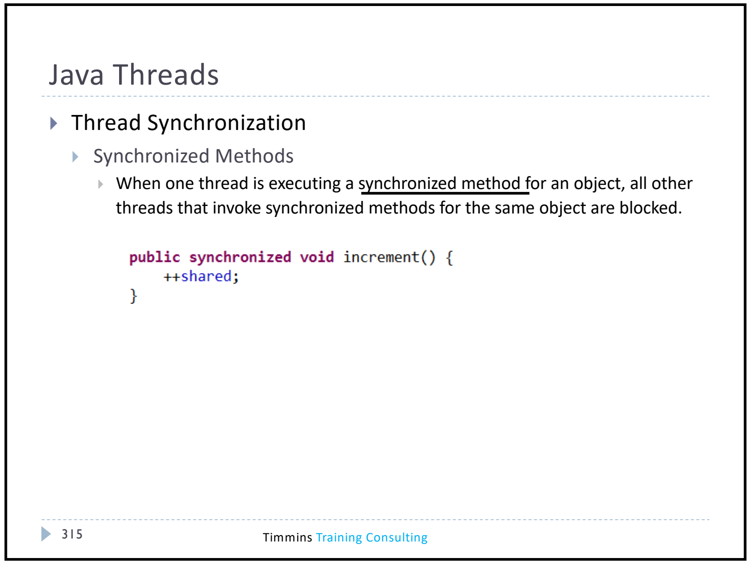
313



Java Threads

- Thread Synchronization
  - Monitors and Locks
    - A thread can lock and unlock a monitor.
    - Only one thread at a time can acquire the lock on a monitor.
    - If a thread has acquired a lock, other threads trying to acquire the lock are blocked
  - Acquiring Lock
    - synchronized method
    - Synchronized statement

314

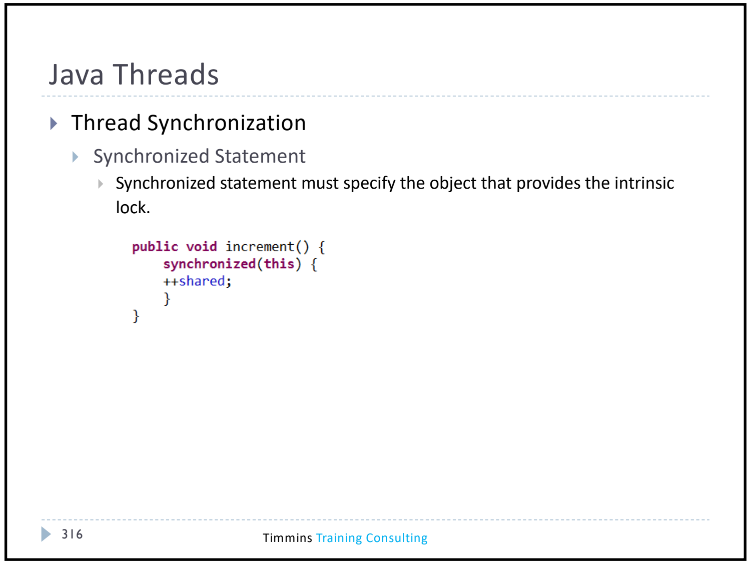


Java Threads

- Thread Synchronization
  - Synchronized Methods
    - When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object are blocked.

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

315

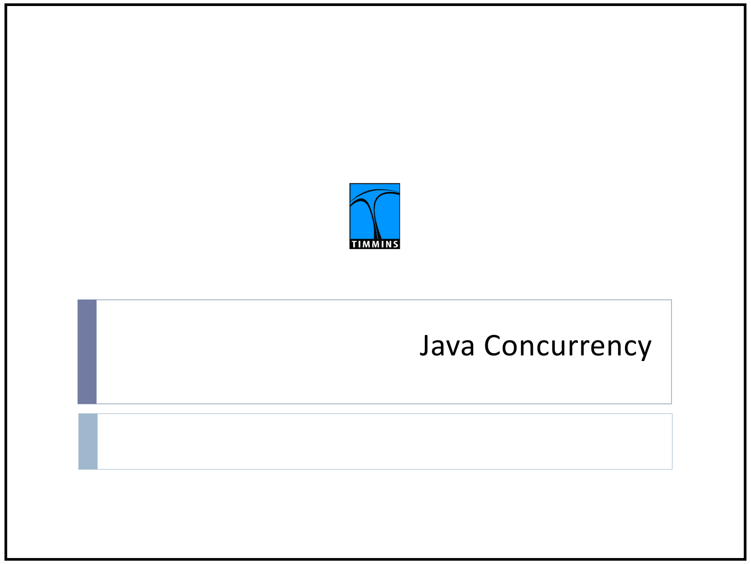


Java Threads

- Thread Synchronization
  - Synchronized Statement
    - Synchronized statement must specify the object that provides the intrinsic lock.

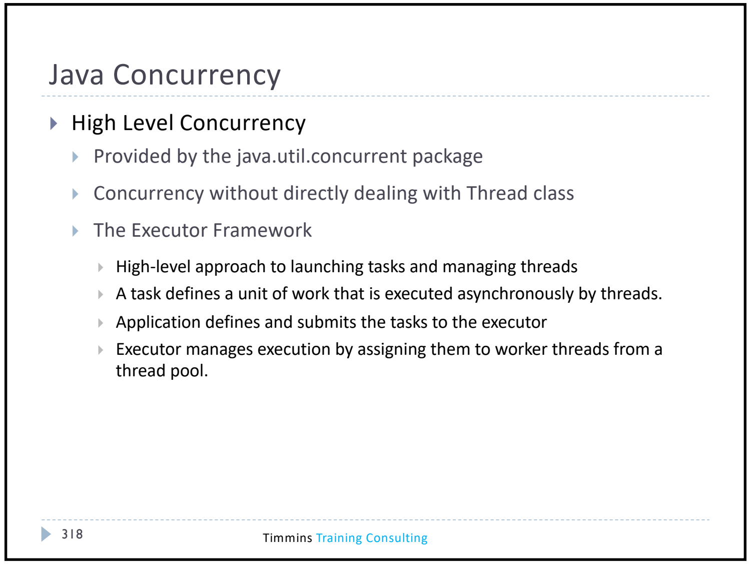
```
public void increment() {
    synchronized(this) {
    ++shared;
    }
}
```

316



Java Concurrency

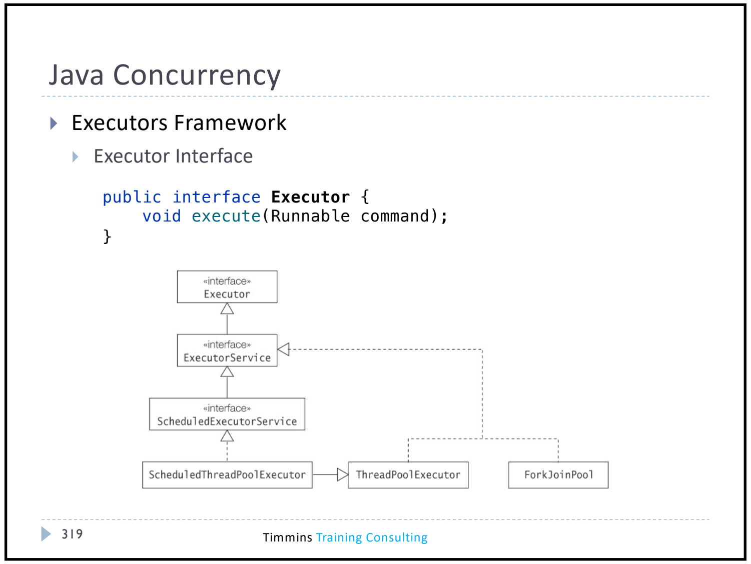
317



Java Concurrency

- High Level Concurrency
  - Provided by the java.util.concurrent package
  - Concurrency without directly dealing with Thread class
  - The Executor Framework
    - High-level approach to launching tasks and managing threads
    - A task defines a unit of work that is executed asynchronously by threads.
    - Application defines and submits the tasks to the executor
    - Executor manages execution by assigning them to worker threads from a thread pool.

318



Java Concurrency

- Executors Framework
  - Executor Interface

```
public interface Executor {
    void execute(Runnable command);
}
```

«interface»
Executor

«interface»
ExecutorService

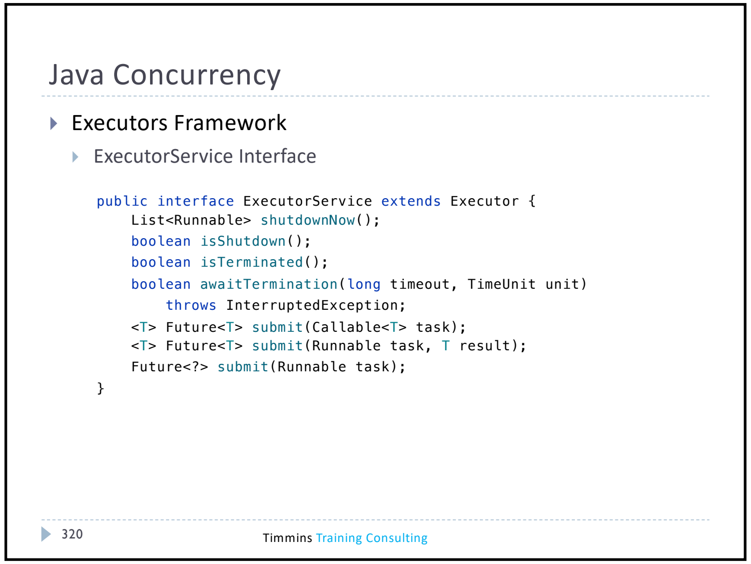
«interface»
ScheduledExecutorService

ScheduledThreadPoolExecutor

ThreadPoolExecutor

ForkJoinPool

319

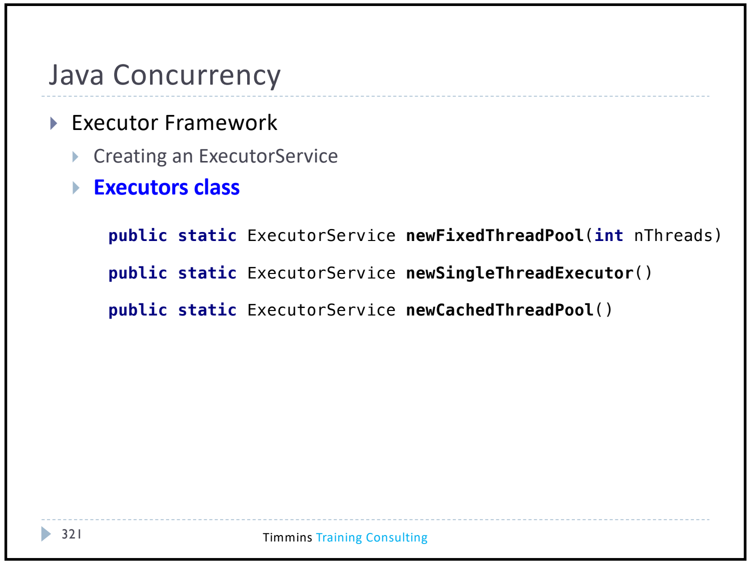


Java Concurrency

- Executors Framework
  - ExecutorService Interface

```
public interface ExecutorService extends Executor {
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;
    <T> Future<T> submit(Callable<T> task);
    <T> Future<T> submit(Runnable task, T result);
    Future<?> submit(Runnable task);
}
```

320



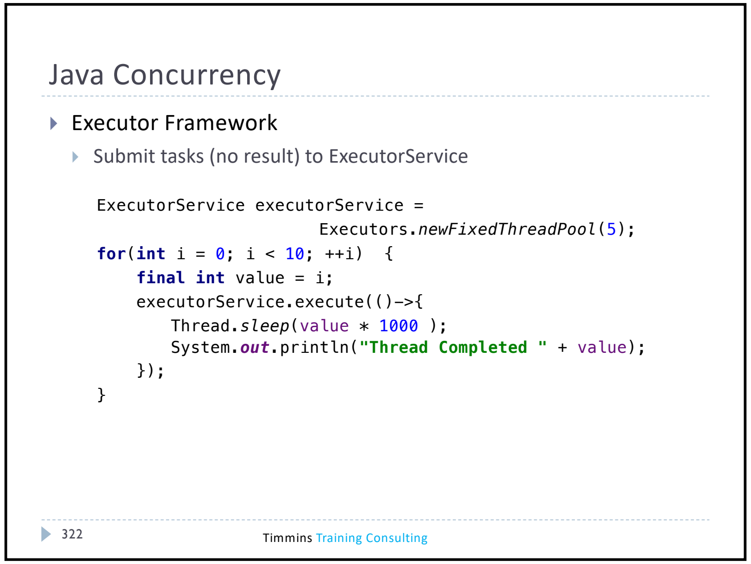
Java Concurrency

- Executor Framework
  - Creating an ExecutorService
  - **Executors class**

```
public static ExecutorService newFixedThreadPool(int nThreads)
public static ExecutorService newSingleThreadExecutor()
public static ExecutorService newCachedThreadPool()
```

321 Timmins Training Consulting

321



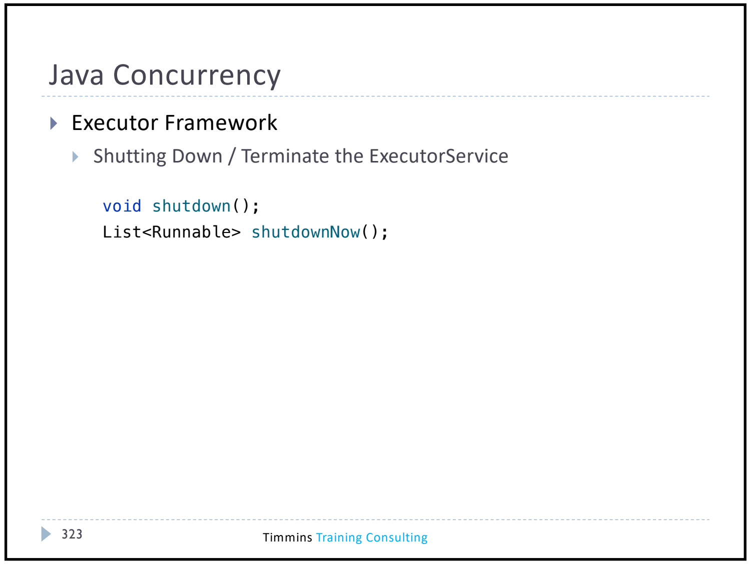
Java Concurrency

- Executor Framework
  - Submit tasks (no result) to ExecutorService

```
ExecutorService executorService =
                    Executors.newFixedThreadPool(5);
for(int i = 0; i < 10; ++i)  {
    final int value = i;
    executorService.execute(()->{
       Thread.sleep(value * 1000 );
       System.out.println("Thread Completed " + value);
    });
}
```

322 Timmins Training Consulting

322

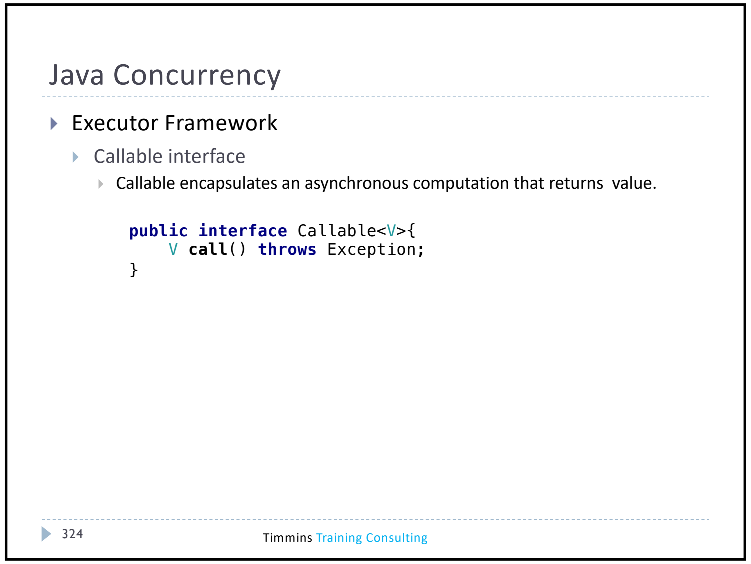


Java Concurrency

- Executor Framework
  - Shutting Down / Terminate the ExecutorService

```
void shutdown();
List<Runnable> shutdownNow();
```

323

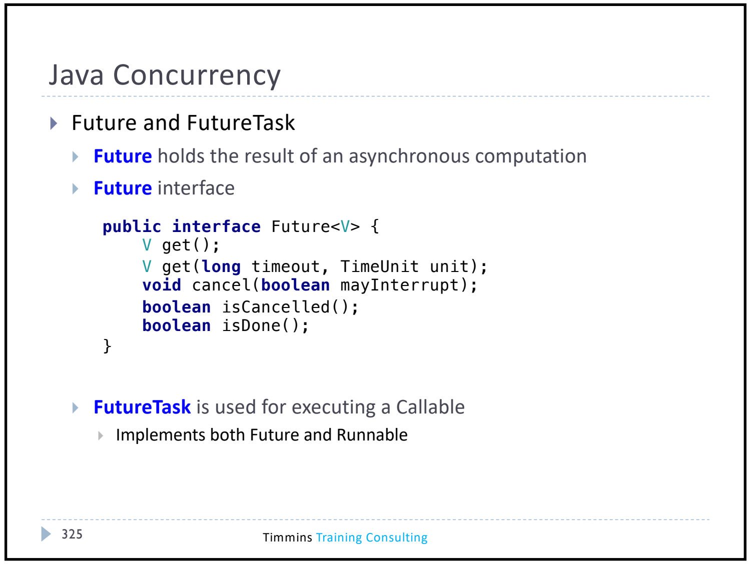


Java Concurrency

- Executor Framework
  - Callable interface
    - Callable encapsulates an asynchronous computation that returns value.

```
public interface Callable<V>{
    V call() throws Exception;
}
```

324



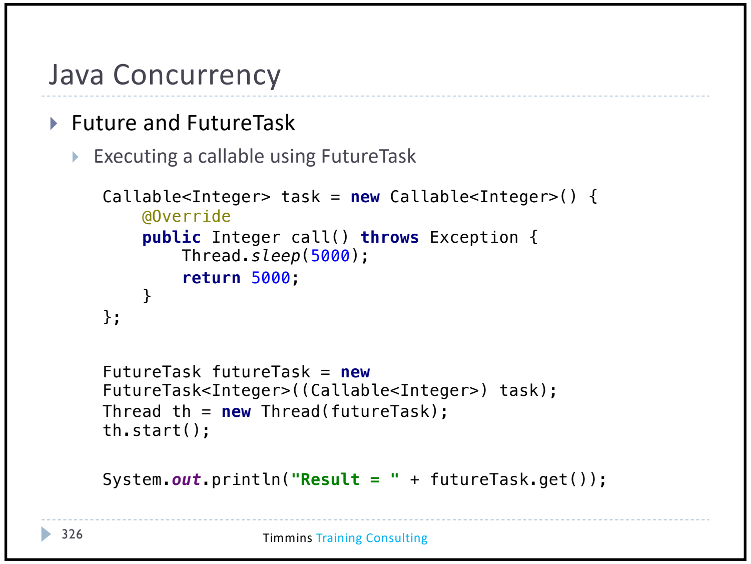
Java Concurrency

- Future and FutureTask
  - Future holds the result of an asynchronous computation
  - Future interface

```
public interface Future<V> {
    V get();
    V get(long timeout, TimeUnit unit);
    void cancel(boolean mayInterrupt);
    boolean isCancelled();
    boolean isDone();
}
```

  - FutureTask is used for executing a Callable
    - Implements both Future and Runnable

325



Java Concurrency

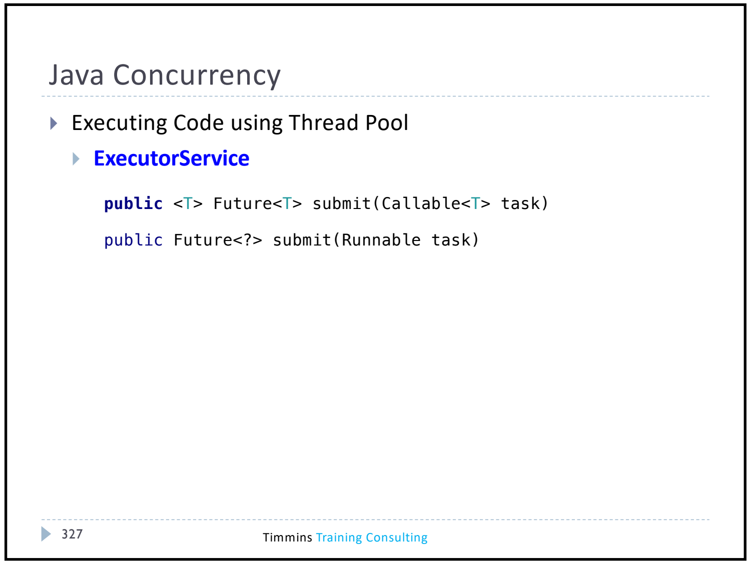
- Future and FutureTask
  - Executing a callable using FutureTask

```
Callable<Integer> task = new Callable<Integer>() {
    @Override
    public Integer call() throws Exception {
        Thread.sleep(5000);
        return 5000;
    }
};

FutureTask futureTask = new
FutureTask<Integer>((Callable<Integer>) task);
Thread th = new Thread(futureTask);
th.start();

System.out.println("Result = " + futureTask.get());
```

326



Java Concurrency

- Executing Code using Thread Pool
  - **ExecutorService**

```
public <T> Future<T> submit(Callable<T> task)

public Future<?> submit(Runnable task)
```