MultiThreading

Presented by
VAISHALI TAPASWI

FANDS INFONET Pvt.Ltd.

www.fandsindia.com
fands@vsnl.com

Ground Rules

- Turn off cell phone. If you cannot, please keep it on silent mode. You can go out and attend your call.
- If you have questions or issues, please let me know immediately.
- Let us be punctual.

Contents

What is Multi-Processing?

- A multi-processing Operating System can run several processes at the same time
- Each process has its own address/memory space
- The OS's scheduler decides when each process is executed
- Only one process is actually executing at any given time. However, the system appears to be running several programs simultaneously
- Separate processes to not have access to each other's memory space
- Many OSes have a shared memory system so that processes can share memory space

What is Multithreading?

- In a multithreaded application, there are several points of execution within the same memory space.
- Each point of execution is called a thread
- Threads can share memory access

Why use Multithreading?

- In a single threaded application, one thread of execution must do everything
  - A single task with lots of processing -> application appears to be "sluggish" or unresponsive.
  - Several tasks -> Time to required to get to the task
- Each task can be performed by a separate thread
  - If one thread is executing a long process, it does not make the entire application wait for it to finish.
- If a multithreaded application is being executed on a system that has multiple processors, the OS may execute separate threads simultaneously on separate processors.

Which Applications Use Multithreading?

- Any kind of application which has distinct tasks which can be performed independently
- Any application with a GUI.
  - Threads dedicated to the GUI can delegate the processing of user requests to other threads.
  - The GUI remains responsive to the user even when the user's requests are being processed
- Any application which requires asynchronous response
- Network based applications are ideally suited to multithreading.
  - Data can arrive from the network at any time.
  - Dedicated thread for listening on the network port

How does it all work?

- Each thread is given its own "context"
- A thread's context includes virtual registers and its own calling stack
- The "scheduler" decides which thread executes at any given time
- The VM may use its own scheduler
- Since many OSes now directly support multithreading, the VM may use the system's scheduler for scheduling threads
- The scheduler maintains a list of ready threads (the run queue) and a list of threads waiting for input (the wait queue)
- Note: the programmer cannot make assumptions about how threads are going to be scheduled. Typically, threads will be executed differently on different platforms.

Single Threaded/MultiThreaded

- JVM is multithreaded
  - – Code Compilation
  - – Garbage Collection
  - – Class loaders
- Main
  - – Main method will run in main thread
  - – Single threaded by default

Java Threads

- A thread is **<u>not</u>** an object
- A thread is a flow of control
- A thread is a series of executed statements
- A thread is a nested sequence of method calls

The Thread Object

- A thread is not an object
- A Thread **<u>is</u>** an object

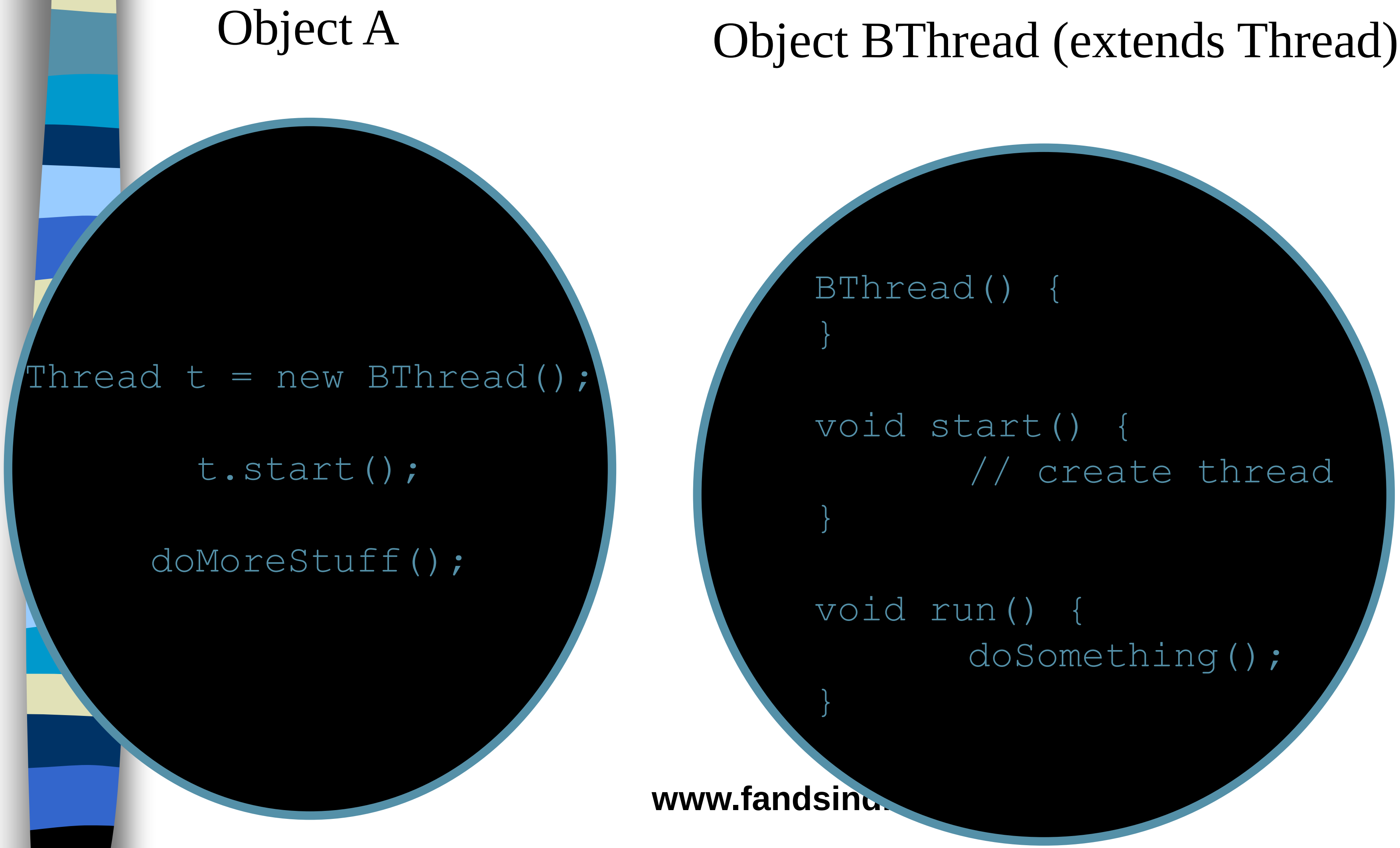
```
void start()
```

  – Creates a new thread and makes it runnable

```
void run()
```

  – The new thread begins its life inside this method

Thread Creation Diagram



Object A

```
Thread t = new BThread();

      t.start();

   doMoreStuff();
```

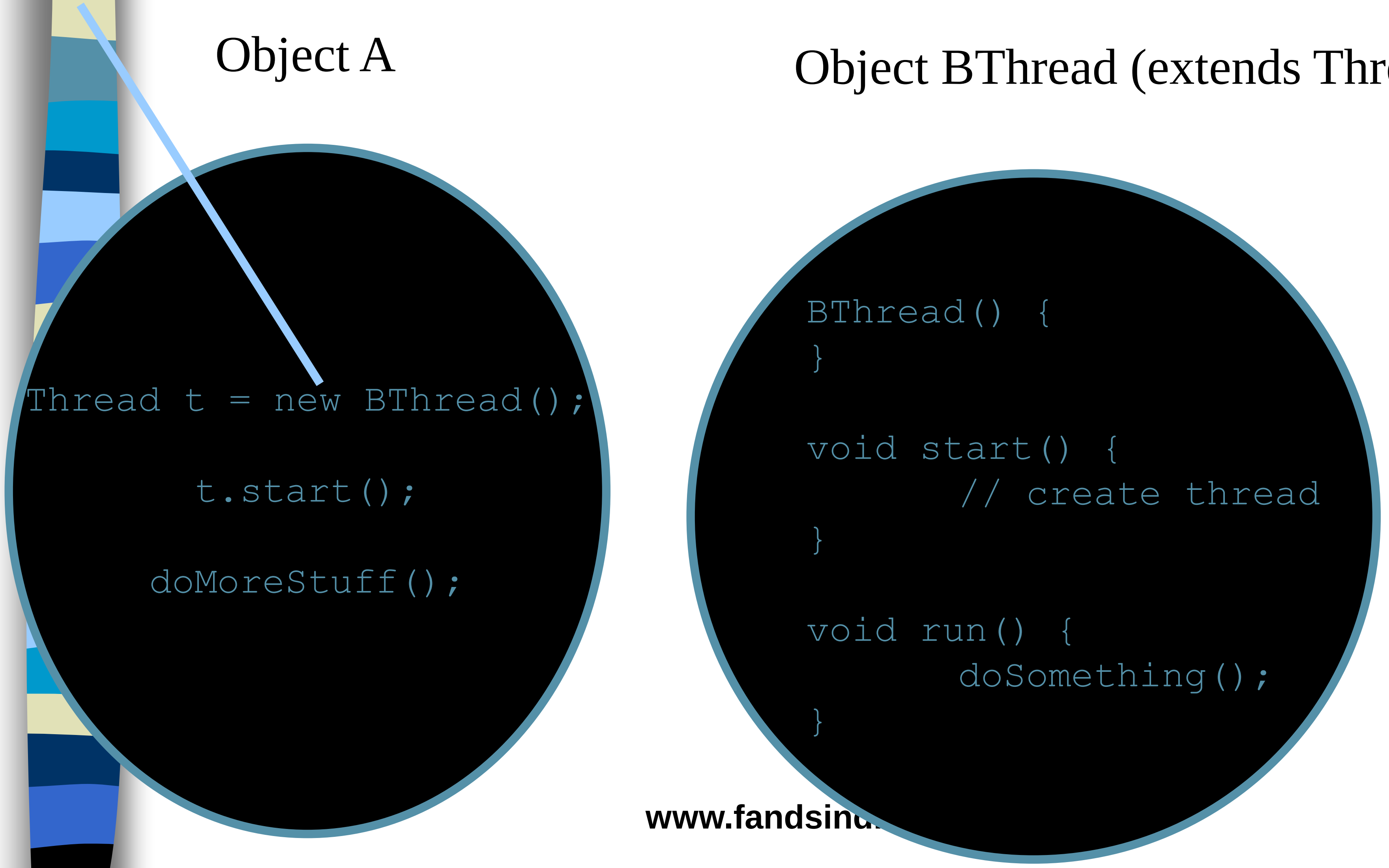
Object BThread (extends Thread)

```
BThread() {
}

void start() {
       // create thread
}

void run() {
       doSomething();
}
```

Thread Creation Diagram

Object A

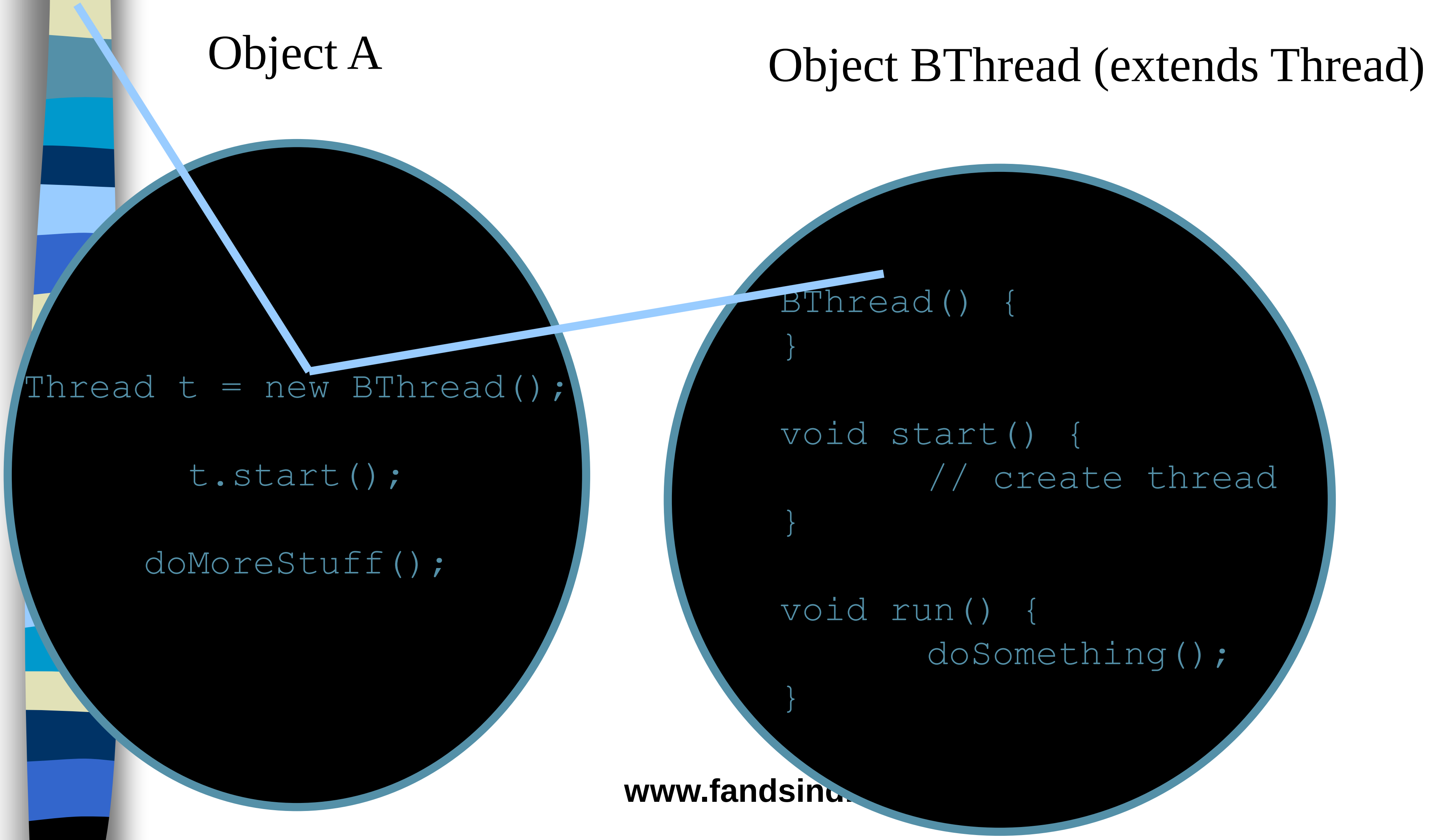
Object BThread (extends Thread)



```
Thread t = new BThread();

       t.start();

     doMoreStuff();
```

```
BThread() {
}

void start() {
       // create thread
}

void run() {
       doSomething();
}
```

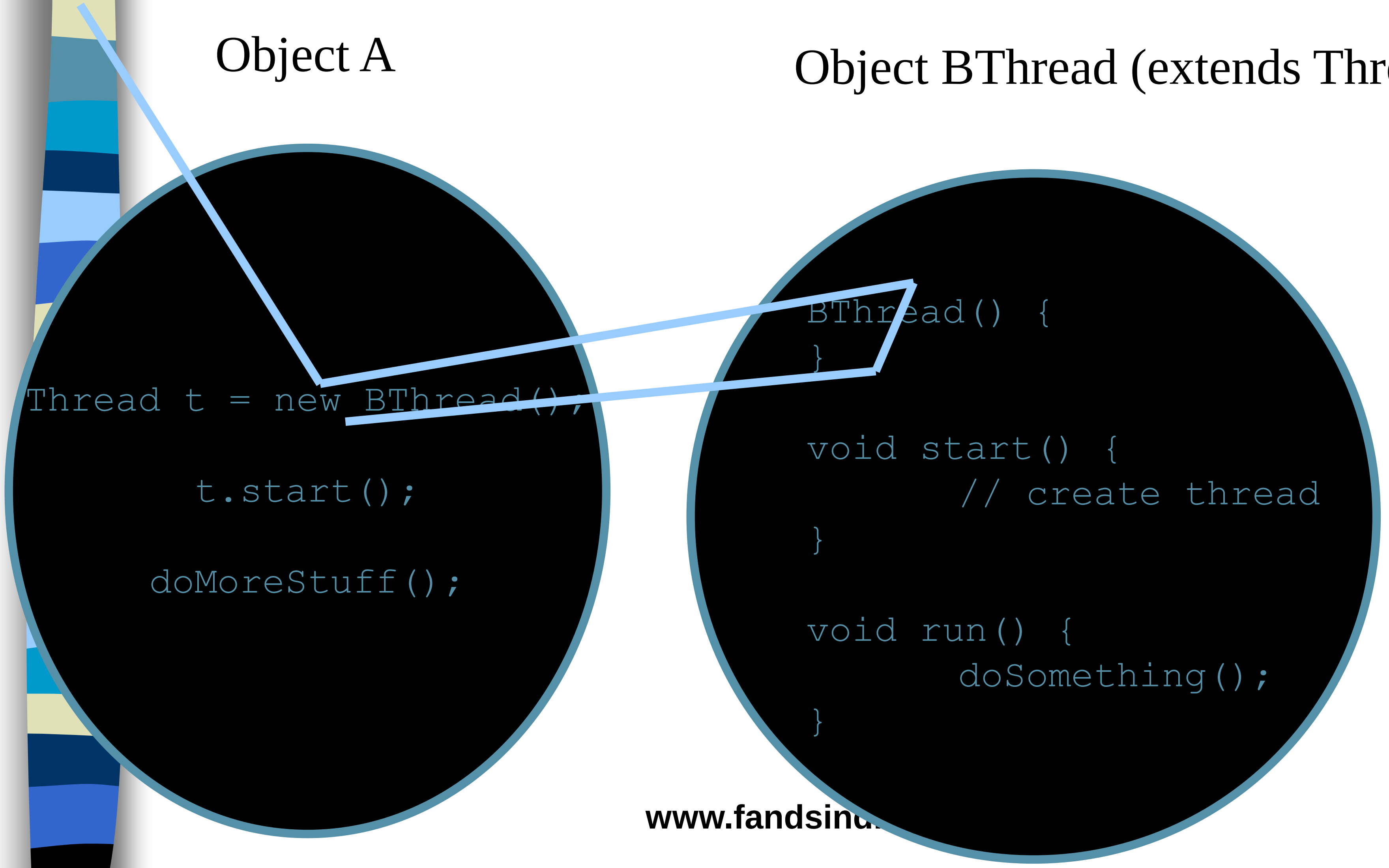
Thread Creation Diagram



```
Thread t = new BThread();

        t.start();

      doMoreStuff();
```

```
BThread() {
}

void start() {
        // create thread
}

void run() {
        doSomething();
}
```

Thread Creation Diagram



Object A

```
Thread t = new BThread();

     t.start();

   doMoreStuff();
```

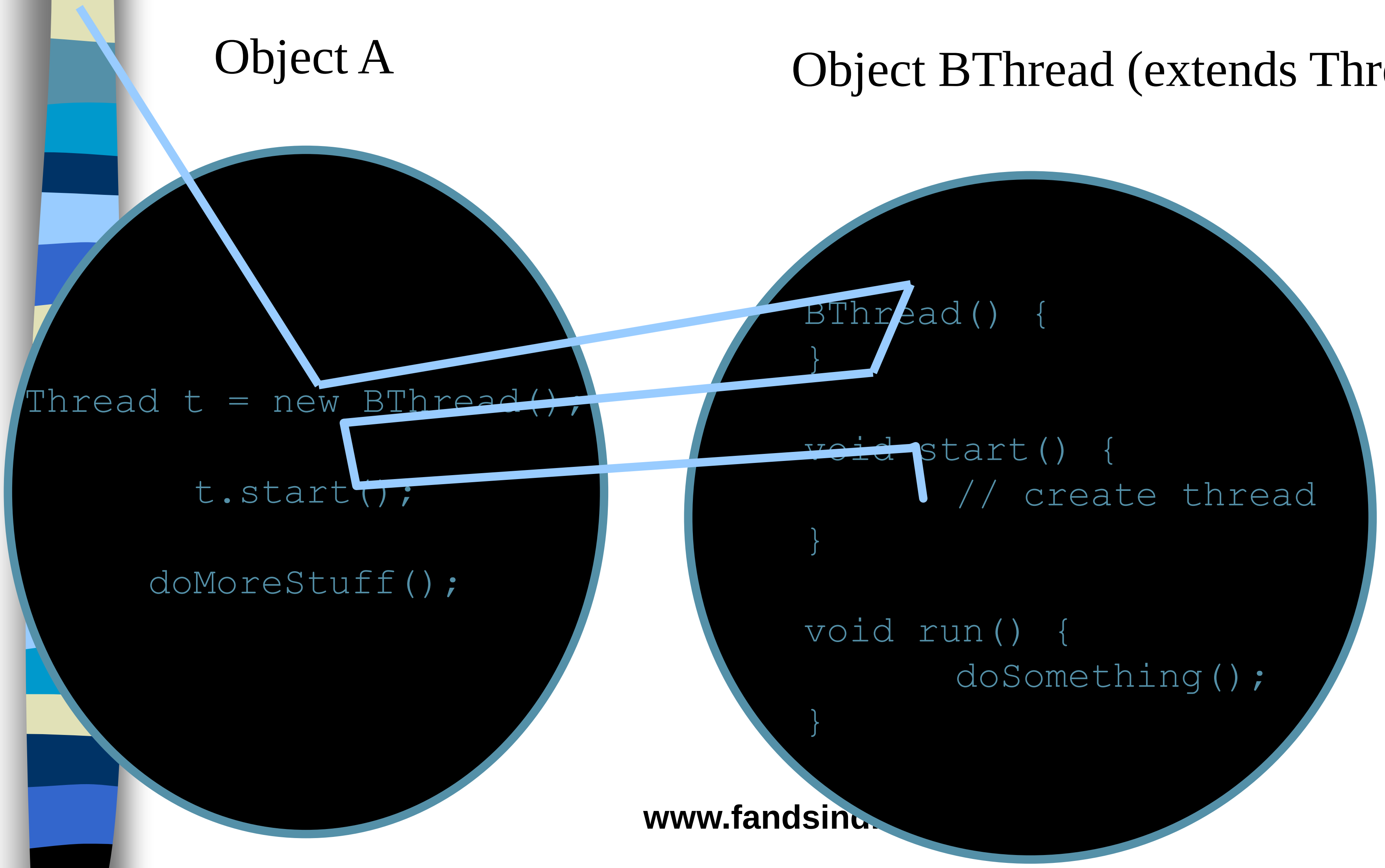
Object BThread (extends Thread)

```
BThread() {
}

void start() {
        // create thread
}

void run() {
        doSomething();
}
```

Thread Creation Diagram

Object A

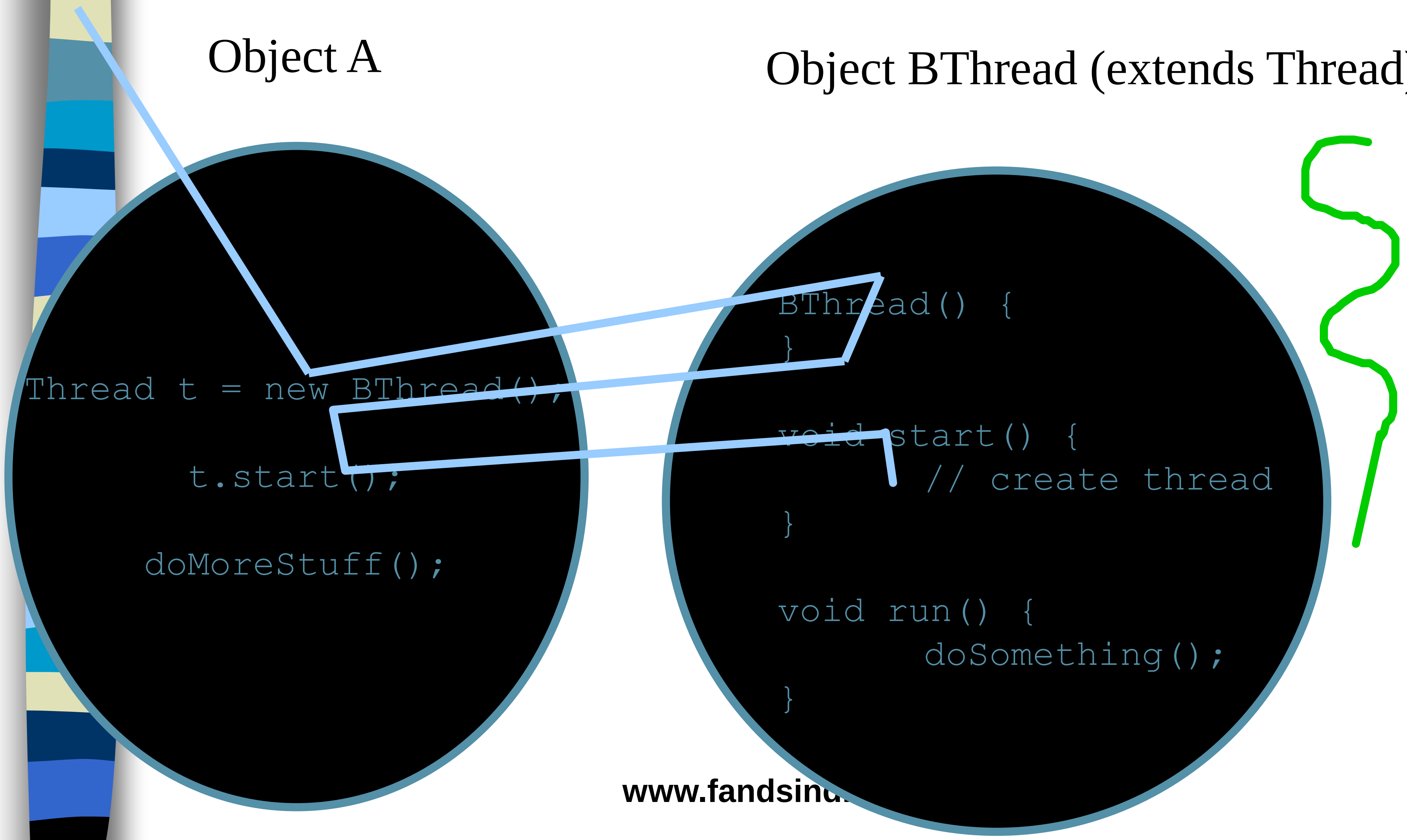
Object BThread (extends Thread)



```
Thread t = new BThread();

     t.start();

    doMoreStuff();
```

```
BThread() {
}

void start() {
       // create thread
}

void run() {
       doSomething();
}
```

Thread Creation Diagram



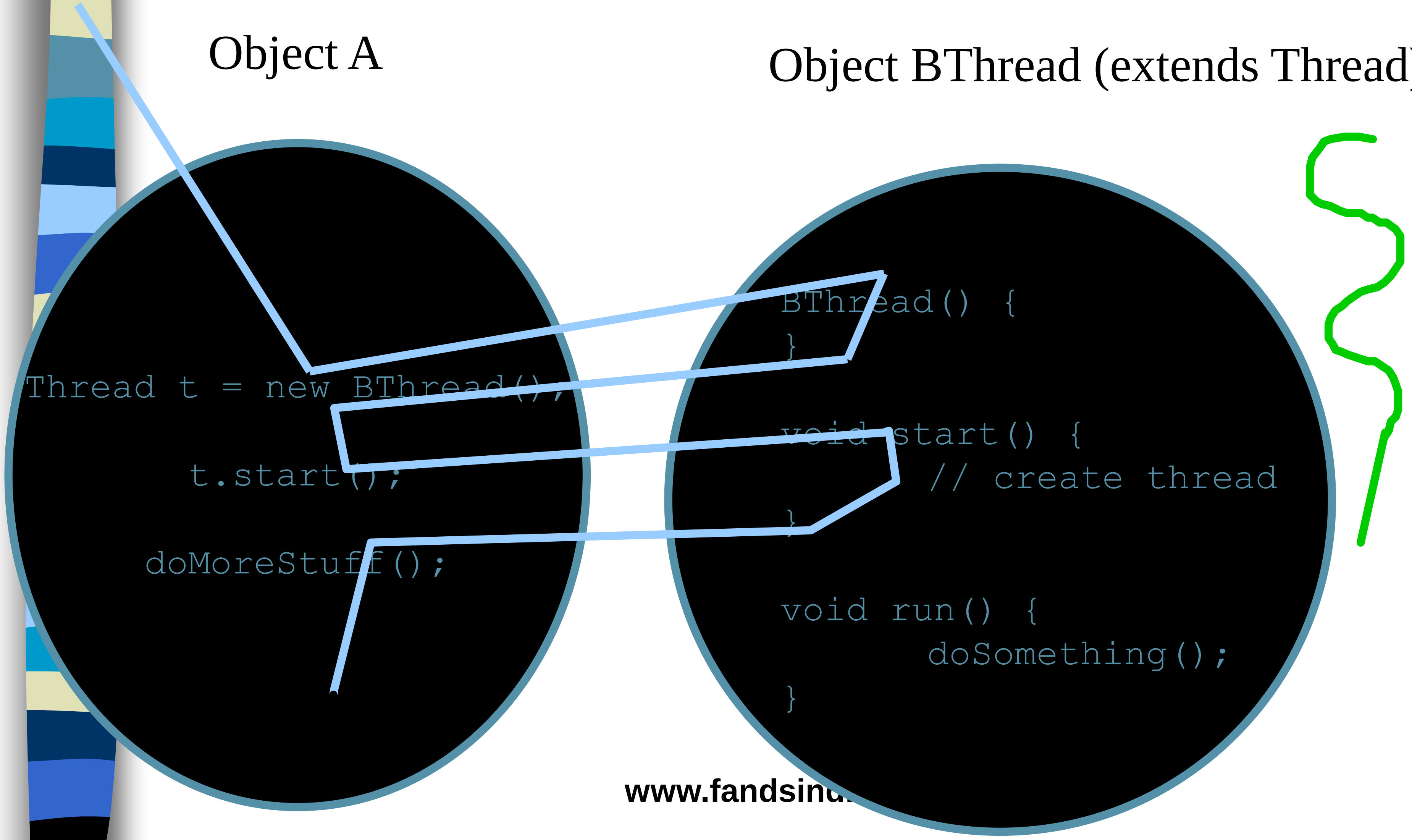
```
Thread t = new BThread();

    t.start();

  doMoreStuff();
```

```
BThread() {
}

void start() {
       // create thread
}

void run() {
       doSomething();
}
```

Thread Creation Diagram

Object A

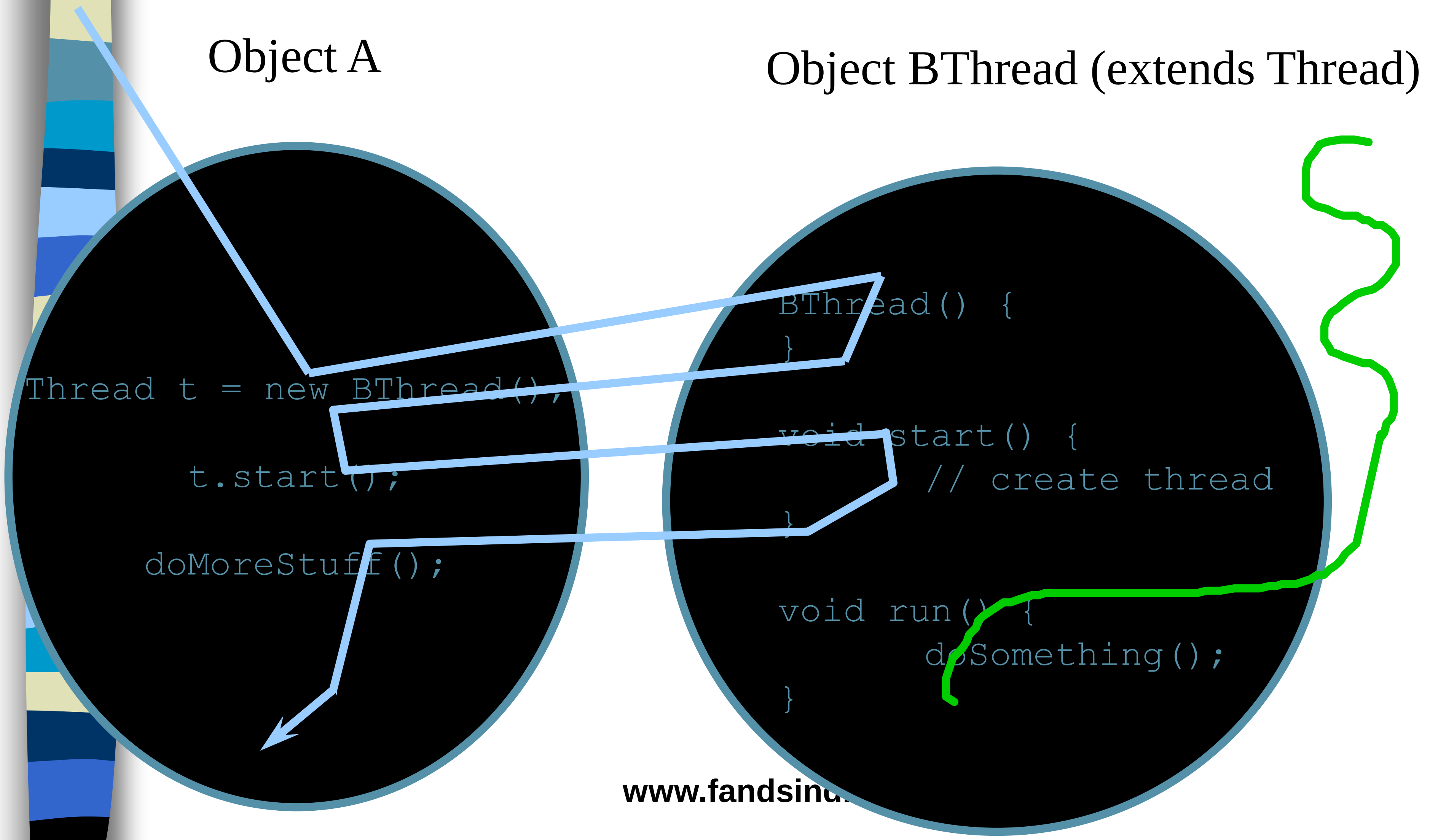
Object BThread (extends Thread)



```
Thread t = new BThread();

        t.start();

      doMoreStuff();
```

```
BThread() {
}

void start() {
        // create thread
}

void run() {
        doSomething();
}
```

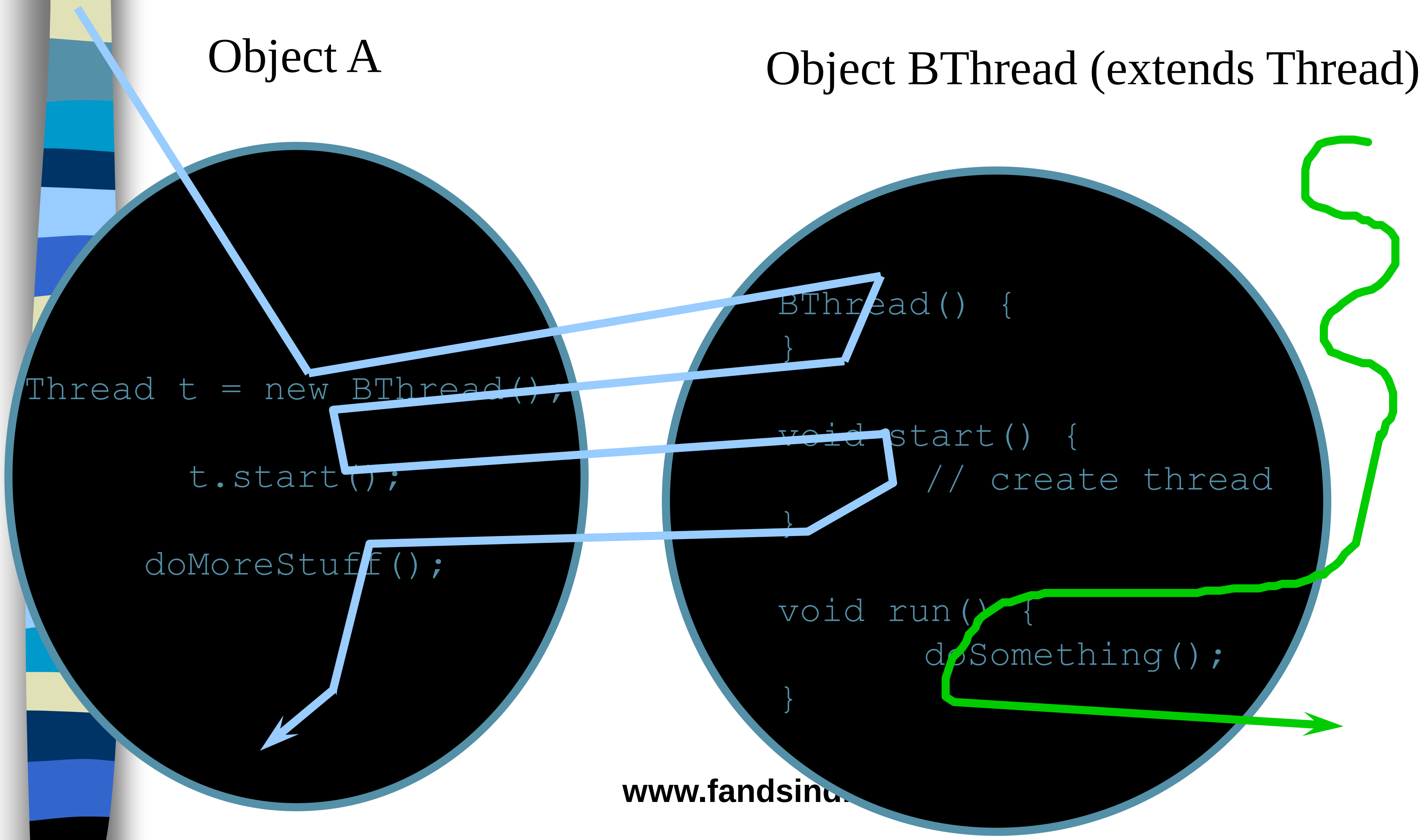
Thread Creation Diagram



```
Thread t = new BThread();

    t.start();

  doMoreStuff();
```

```
BThread() {
}

void start() {
      // create thread
}

void run() {
      doSomething();
}
```

Thread Creation Diagram



```
Thread t = new BThread();

    t.start();

  doMoreStuff();
```

```
BThread() {
}

void start() {
      // create thread
}

void run() {
      doSomething();
}
```

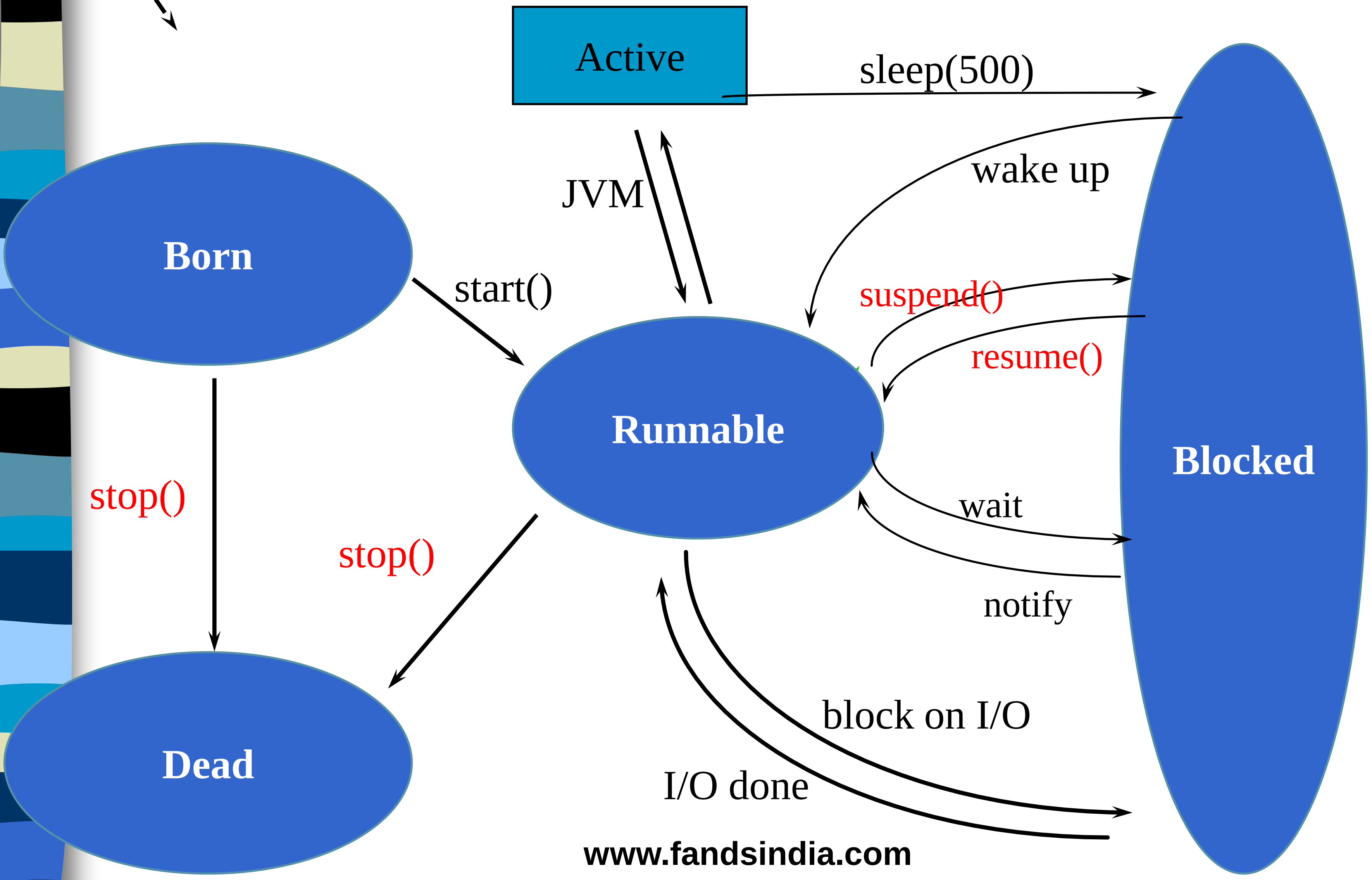
Runnable Interface

- A helper to the thread object
- The Thread object’s run() method calls the Runnable object’s run() method
- Allows threads to run inside any object, regardless of inheritance

Runnable Example

```
Talker talker = new Talker();
Thread t = new Thread(talker);
t.Start();
---
class Talker implements Runnable {
  public void run() {
    while (true) {
      System.out.println(“hello world”);
    } } }
----
Runnable r = ()->{while(true)
  System.out.println(“hw functional”);  }
```

Thread Lifecycle



www.fandsindia.com

Wait and Notify

- Allows two threads to cooperate
- Based on a single shared lock object

Wait and Notify: Code

- Consumer:

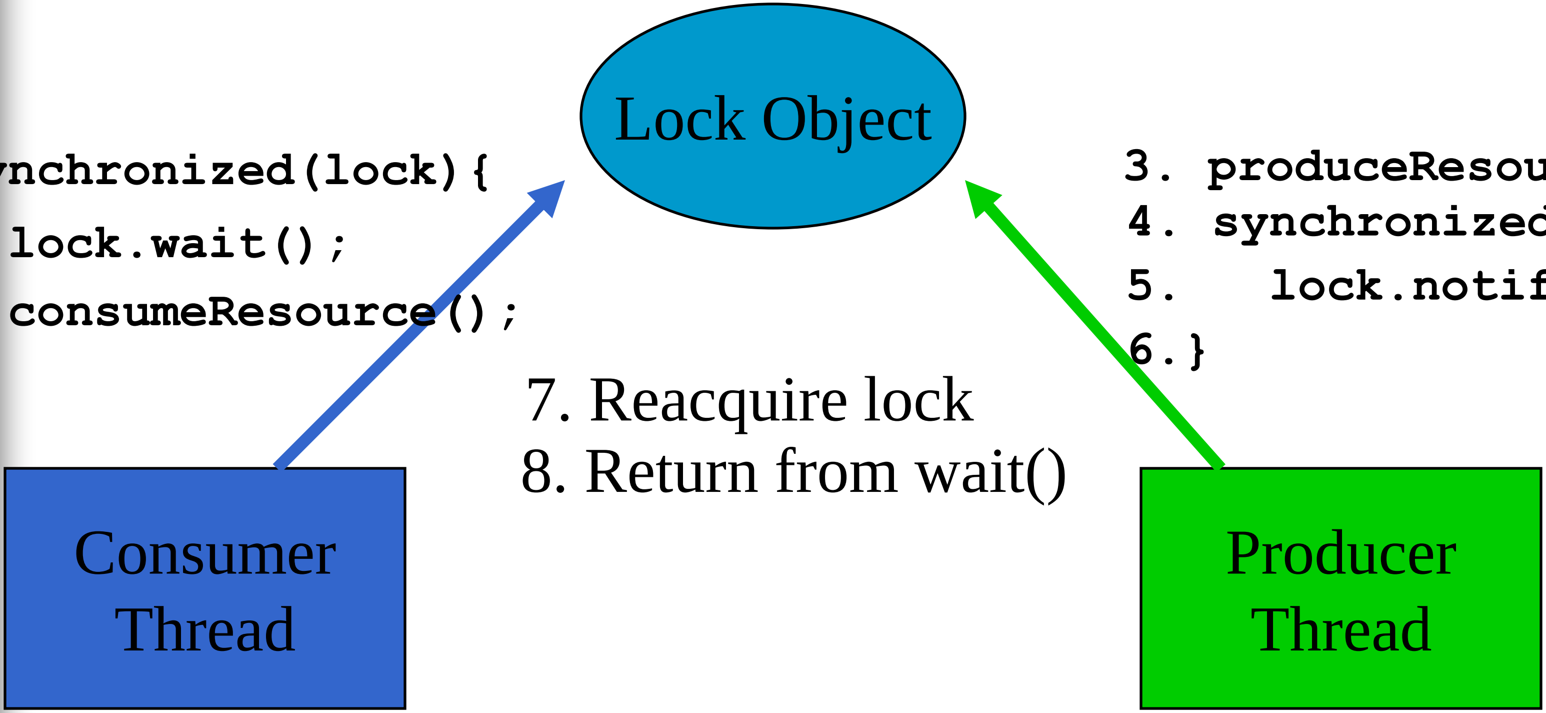
```
synchronized (lock) {
  while (!resourceAvailable()) {
    lock.wait();
  }
  consumeResource();
}
```

Wait and Notify: Code

- Producer:

```
produceResource();
synchronized (lock) {
  lock.notifyAll();
}
```

Wait/Notify Sequence



```
1. synchronized(lock){
2.     lock.wait();
9.     consumeResource();
1[illegible]
```

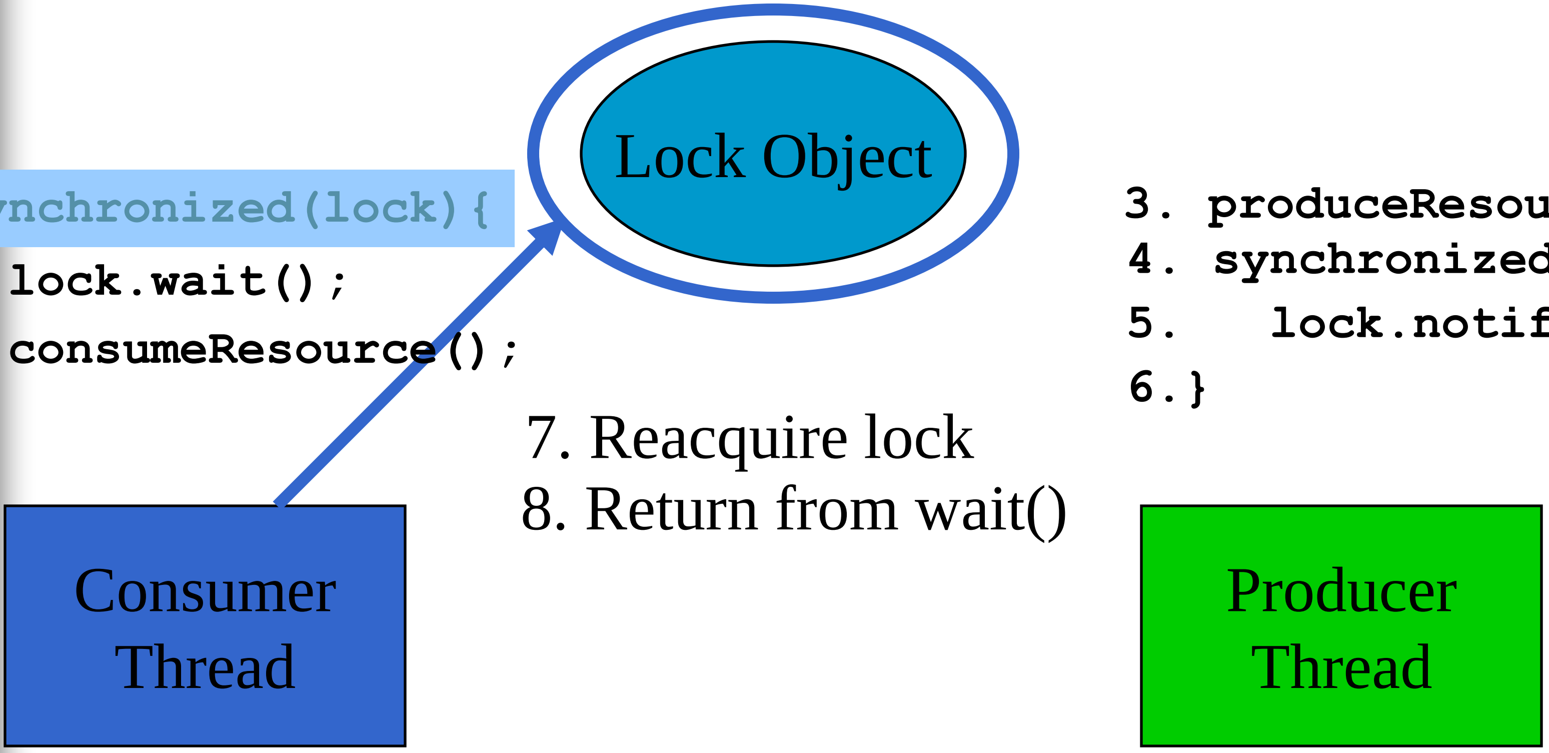
```
3. produceResource()
4. synchronized(lock)
5.     lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()

Wait/Notify Sequence

```
1.  synchronized(lock){
2.      lock.wait();
9.      consumeResource();
1
```

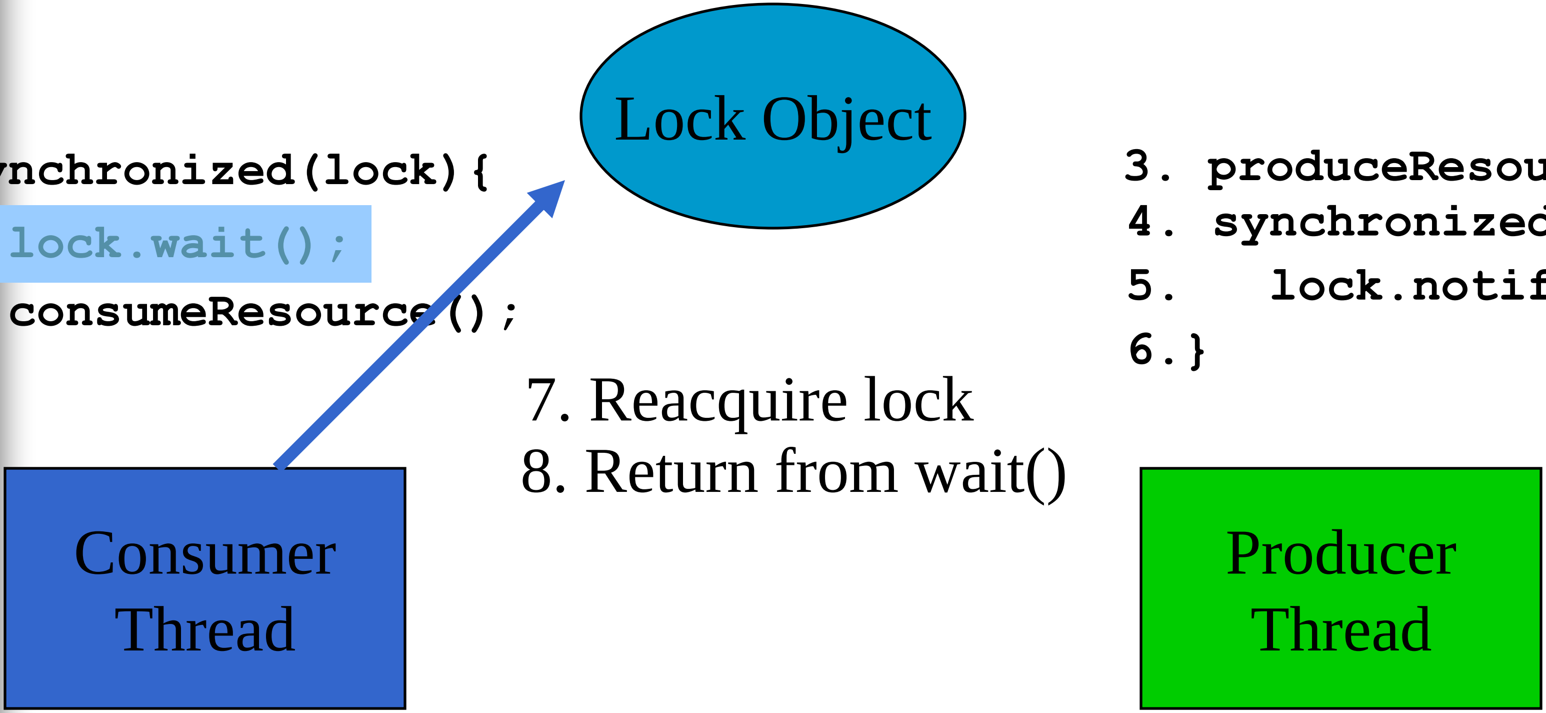
```
3.  produceResource()
4.  synchronized(lock)
5.     lock.notify();
6.}
```



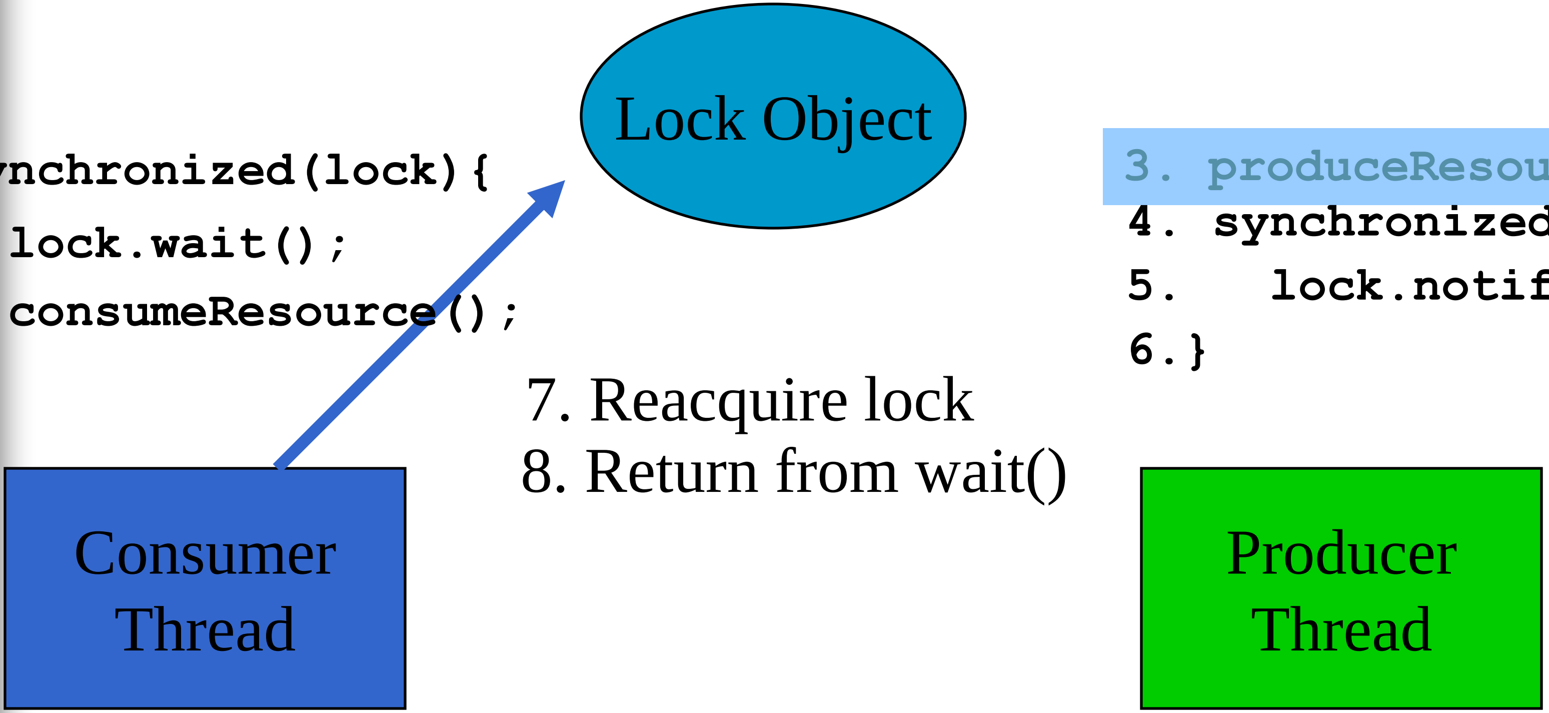
Wait/Notify Sequence

```
1. synchronized(lock){
2.    lock.wait();
9.    consumeResource();
1[illegible]}
```

```
3. produceResource()
4. synchronized(lock)
5.    lock.notify();
6.}
```



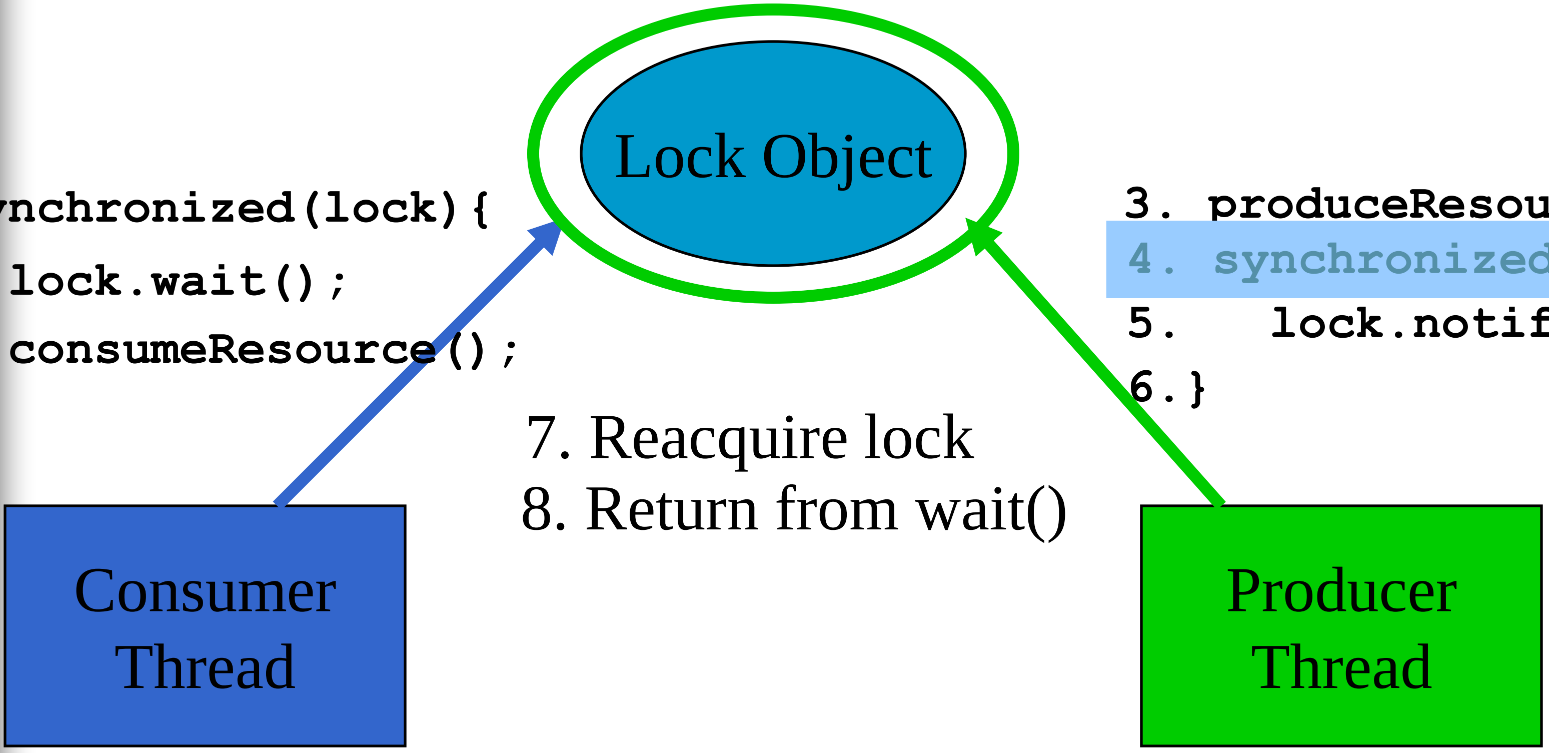
Wait/Notify Sequence



```
1.  synchronized(lock){
2.      lock.wait();
9.      consumeResource();
1
```

```
3.  produceResource()
4.  synchronized(lock)
5.     lock.notify();
6.}
```

Wait/Notify Sequence

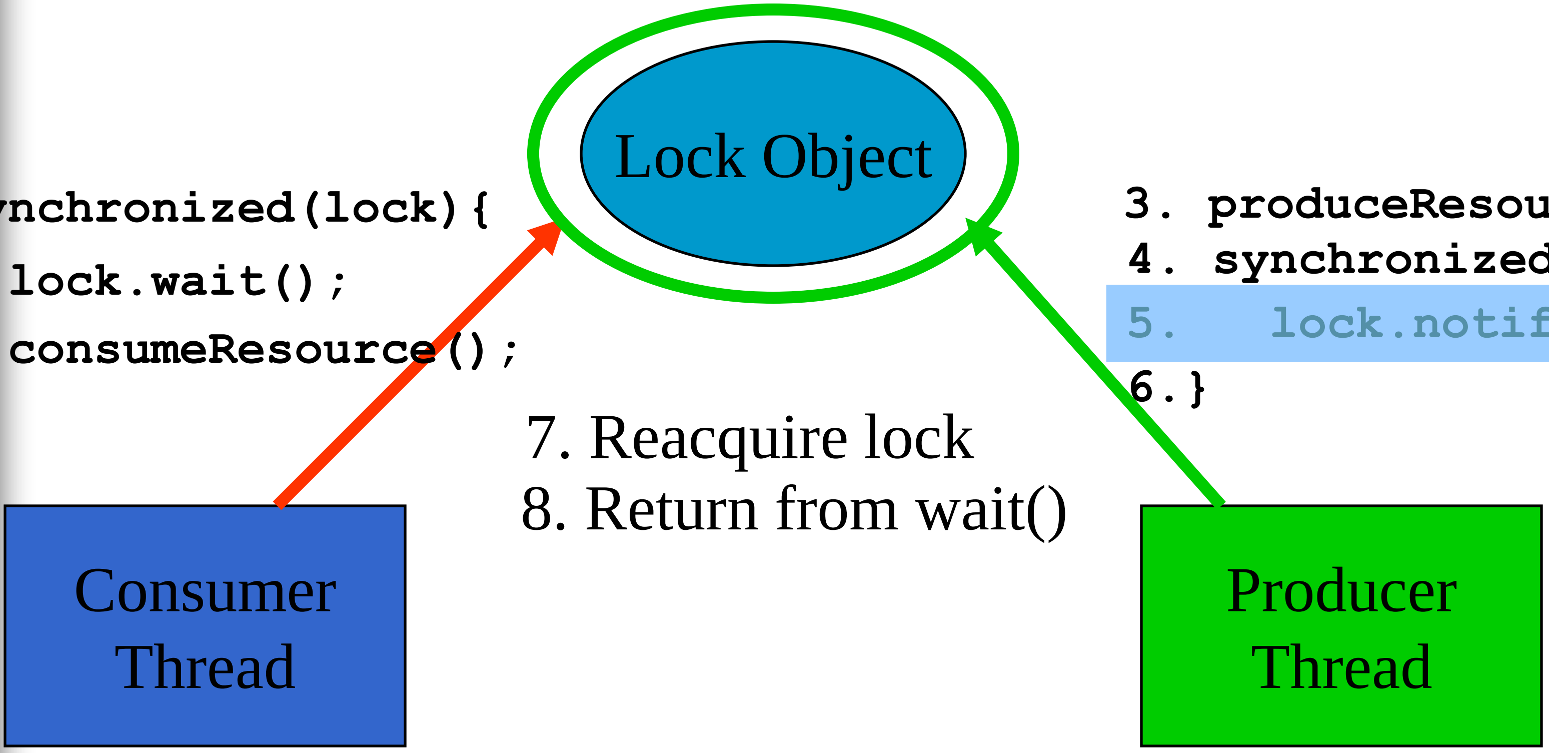


```
1. synchronized(lock){
2.     lock.wait();
9.     consumeResource();
1[illegible]
```

```
3. produceResource()
4. synchronized(lock)
5.     lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()

Wait/Notify Sequence

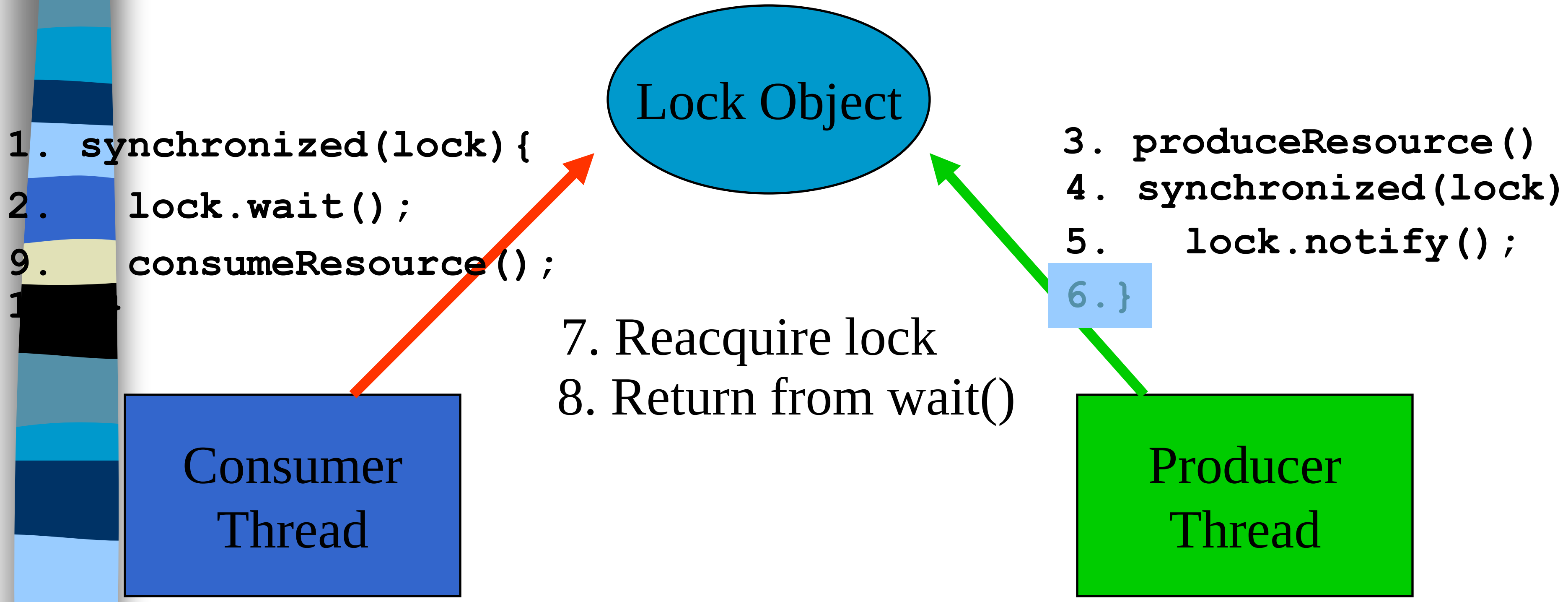


```
1.  synchronized(lock){
2.      lock.wait();
9.      consumeResource();
1[illegible]
```

```
3.  produceResource()
4.  synchronized(lock)
5.      lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()

Wait/Notify Sequence



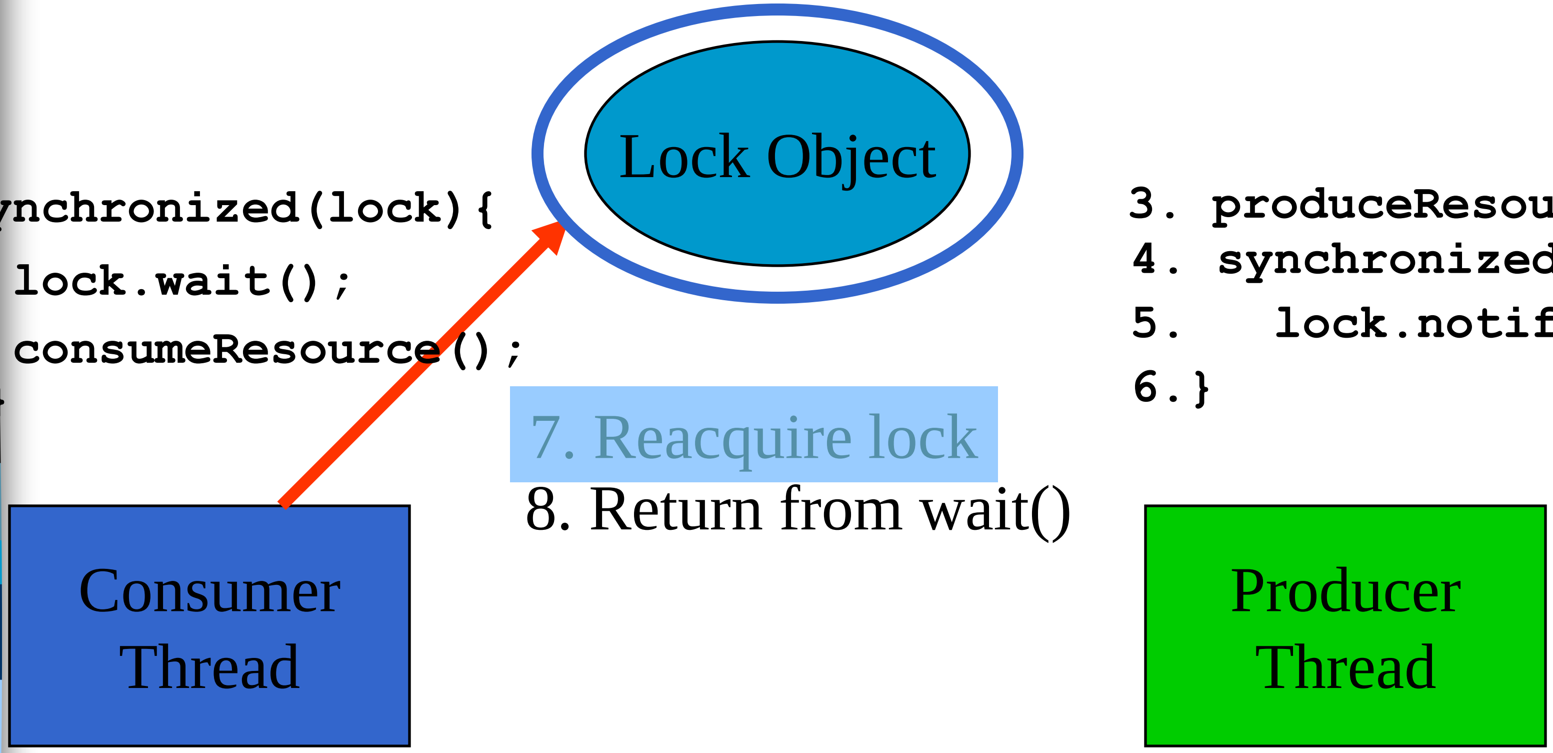
```
1.  synchronized(lock){
2.      lock.wait();
9.      consumeResource();
1[illegible] }
```

```
3.  produceResource()
4.  synchronized(lock)
5.      lock.notify();
6.  }
```

Wait/Notify Sequence

```
1.  synchronized(lock){
2.      lock.wait();
9.      consumeResource();
1[illegible]
```

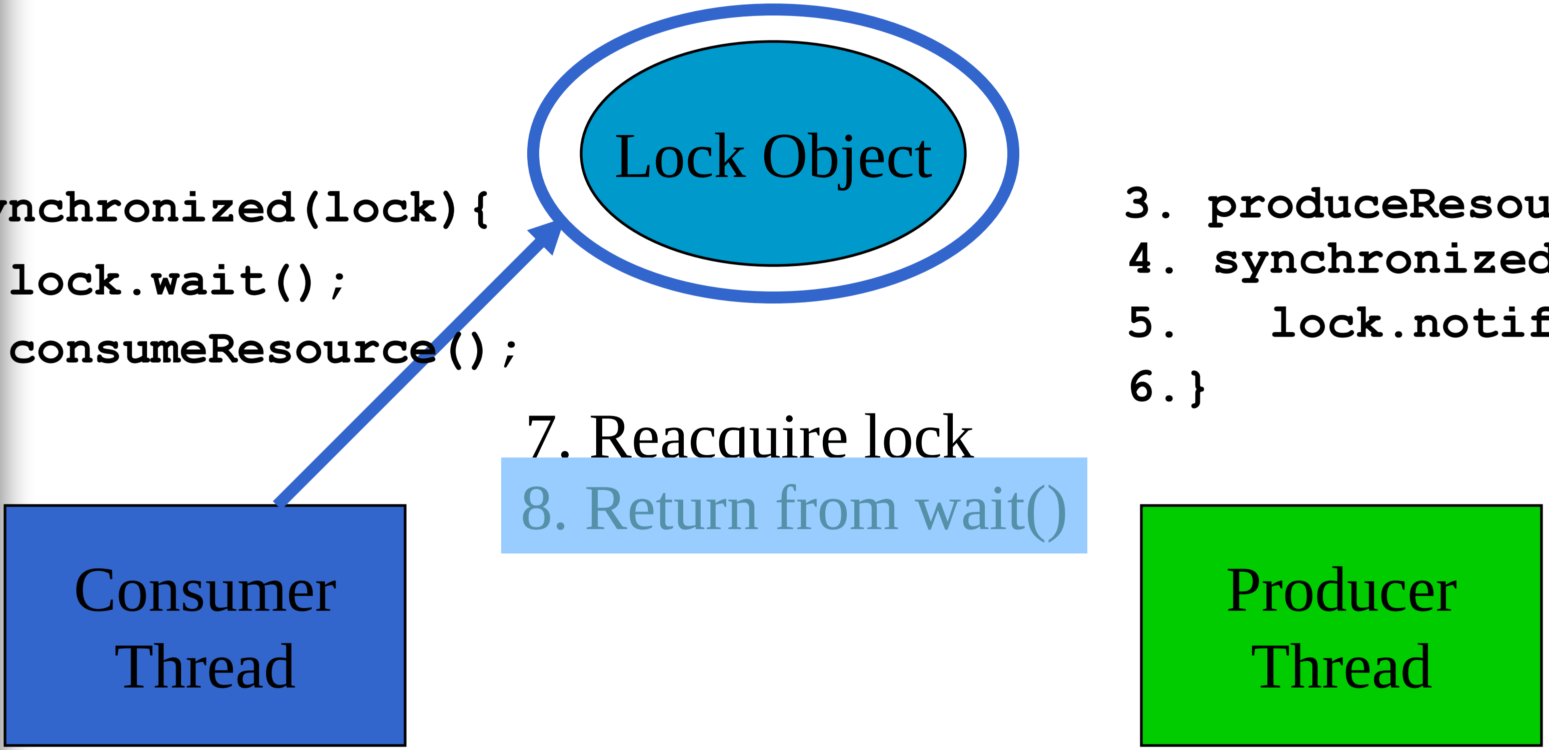
```
3.  produceResource()
4.  synchronized(lock)
5.      lock.notify();
6.}
```



Wait/Notify Sequence

```
1. synchronized(lock){
2.     lock.wait();
9.     consumeResource();
1
```

```
3. produceResource()
4. synchronized(lock)
5.    lock.notify();
6.}
```

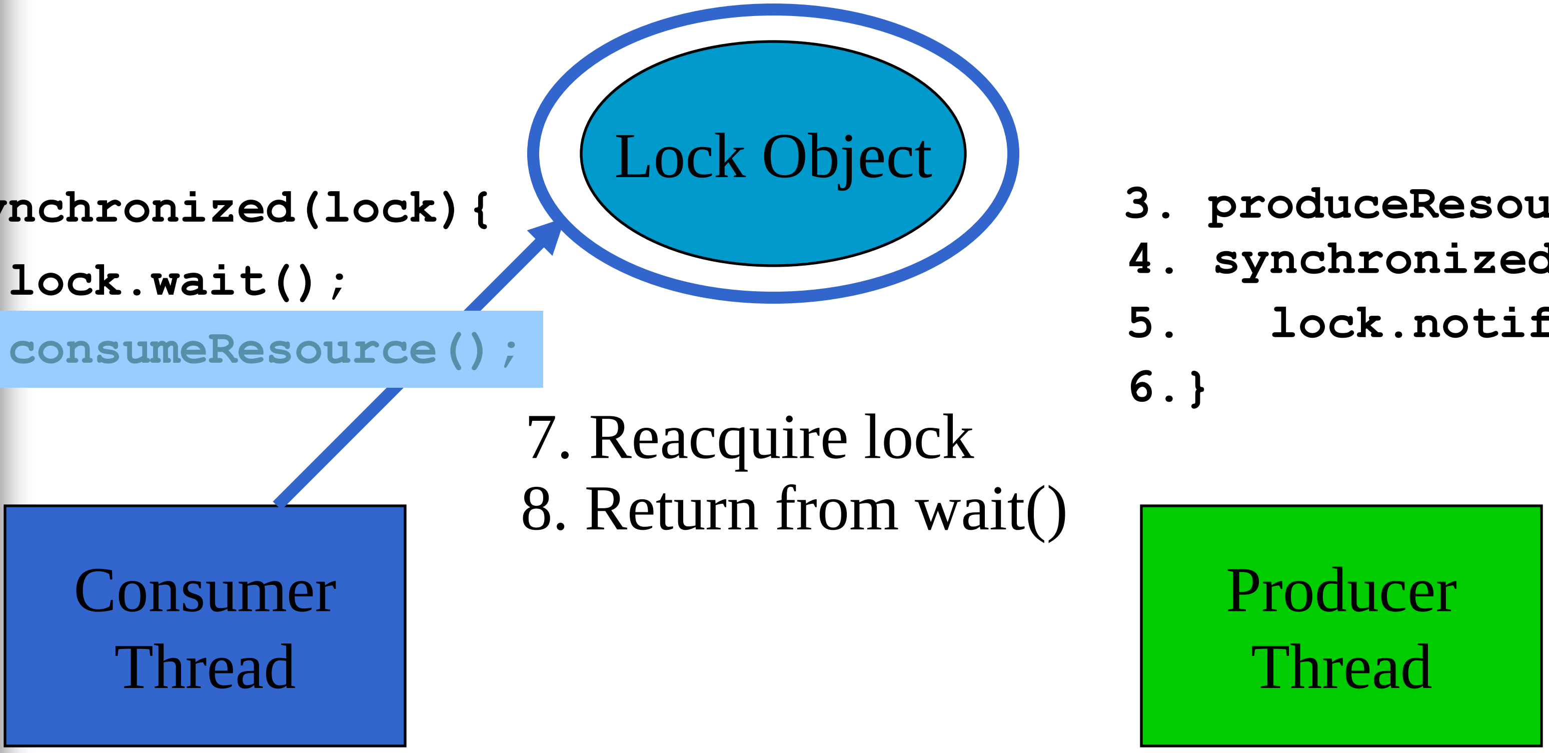


Wait/Notify Sequence

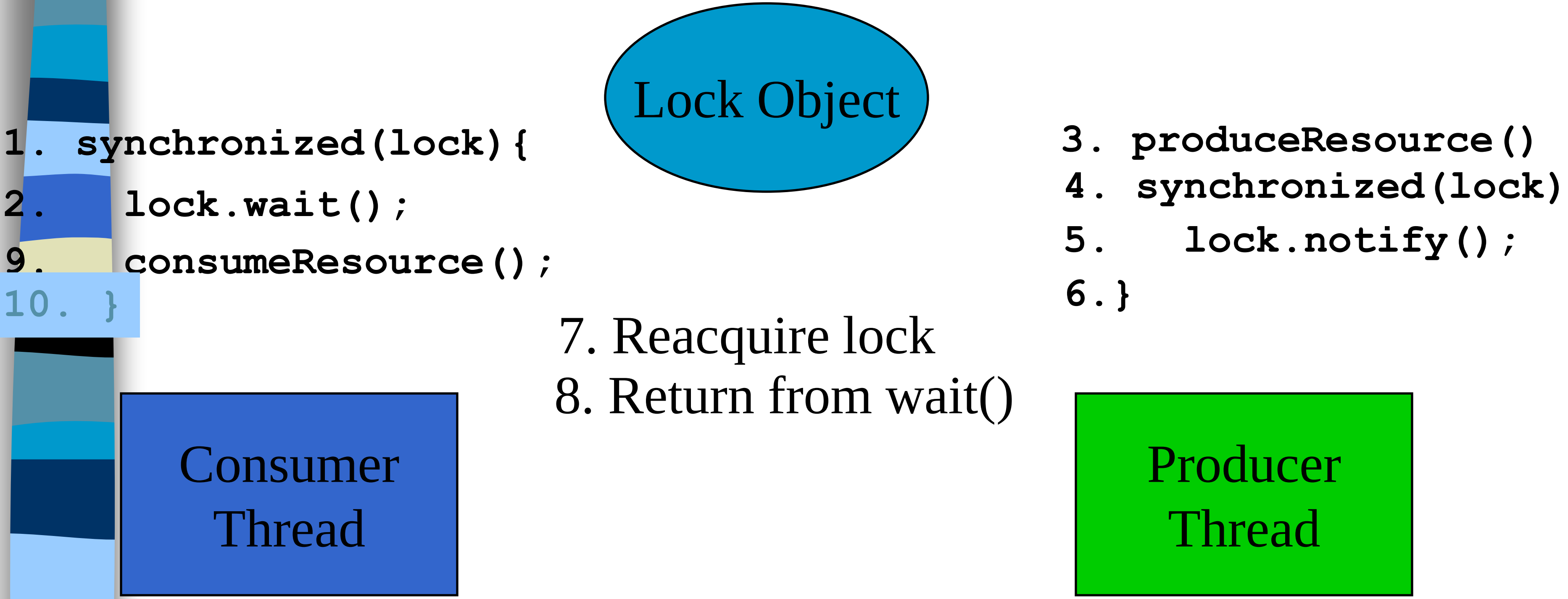
```
1. synchronized(lock){
2.      lock.wait();
9.      consumeResource();
10.}
```

```
3. produceResource()
4. synchronized(lock)
5.     lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()



Wait/Notify Sequence



```
1. synchronized(lock){
2.     lock.wait();
9.     consumeResource();
10. }
```

```
3. produceResource()
4. synchronized(lock)
5.     lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()

Wait/Notify: Details

- Often the lock object is the resource itself
- Sometimes the lock object is the producer thread itself

Wait/Notify: Details

- Must loop on wait(), in case another thread grabs the resource...
  - After you are notified
  - Before you acquire the lock and return from wait()
- Use lock.notifyAll() if there may be more than one waiting thread

Wait/Notify Example: Blocking Queue

```
class BlockingQueue extends Queue {
  public synchronized Object remove() {
    while (isEmpty()) {
      wait();     // really this.wait()
    }
    return super.remove();
  }
  public synchronized void add(Object o) {
    super.add(o);
    notifyAll();  // this.notifyAll()
  }
}
```

Thread Scheduling

- In general, the runnable thread with the highest priority is active (running)
- Java is priority-preemptive
  – If a high-priority thread wakes up, and a low-priority thread is running then the high-priority thread gets to run immediately
- Allows on-demand processing
  – Efficient use of CPU

Blocking Threads

- When reading from a stream, if input is not available, the thread will <u>block</u>
- Thread is suspended ("blocked") until I/O is available
- Allows other threads to automatically activate
- When I/O available, thread wakes back up again
  - Becomes "<u>runnable</u>"
  - Not to be confused with the Runnable interface

Thread

- Threads can be in one of four states
  - – Created
  - – Running
  - – Blocked
  - – Dead

Important Methods

- Start
- Run
- Yield
- Join
- Sleep
- Stop

Stop

- The Thread class had a stop() method
  - One thread could terminate another by invoking its stop() method.
  - using stop() could lead to deadlocks
  - The stop() method is now deprecated.
- Right Way
  - the run method should terminate
  - Add a boolean variable which indicates whether the thread should continue or not
  - Provide a set method for that variable which can be invoked by another thread

Thread Priorities

- Every thread is assigned a priority (1 to 10)
- The default is 5 (NORM_PRIORITY)
- The higher the number, the higher the priority
- Can be set with setPriority(int aPriority)
- The standard mode of operation is that the scheduler executes threads with higher priorities first.
  - This simple scheduling algorithm can cause problems. Specifically, one high priority thread can become a "CPU hog".
- A thread using vast amounts of CPU can share CPU time with other threads by invoking the yield() method on itself.

Thread Priority and OS

- Most OSes do not employ a scheduling algorithm as simple as this one
- Most modern OSes have thread aging
- The more CPU a thread receives, the lower its priority becomes
- The more a thread waits for the CPU, the higher its priority becomes
- Because of thread aging, the effect of setting a thread's priority is dependent on the platform

Thread Priorities: General Strategies

- Threads that have more to do should get lower priority
- Counterintuitive
- Cut to head of line for short tasks
- Give your I/O-bound threads high priority
  - Wake up, immediately process data, go back to waiting for I/O

Thread Group

- Every Java thread is a member of a thread group.
- Thread groups provide a mechanism for collecting multiple threads into a single object and invoked operations on all the threads at once
- The runtime system puts a thread into a thread group during thread construction.

Thread Group

```
public Thread(ThreadGroup group, Runnable target)
```

- Either allow the runtime system to put the new thread in some reasonable default group or you can explicitly set the new thread's group.
- The thread is a permanent member of whatever thread group it joins upon its creation -> you cannot move a thread to a new group after the thread has been created.

Daemon Thread

- Daemon thread is a low priority thread that runs in background to perform tasks such as garbage collection.
- Features
  - They can not prevent the JVM from exiting when all the user threads finish their execution.
  - JVM terminates itself when all user threads finish their execution
  - If JVM finds running daemon thread, it terminates the thread and after that shutdown itself.
  - It is an utmost low priority thread.

Daemon Thread

- Methods for Daemon
  - void setDaemon(boolean status)
    - If you call the setDaemon() method after starting the thread, it would throw IllegalThreadStateException
  - boolean isDaemon()

Daemon vs User Threads

- Priority
  - When the only remaining threads in a process are daemon threads, the interpreter exits. This makes sense because when only daemon threads remain, there is no other thread for which a daemon thread can provide a service.
- Usage
  - Daemon thread is to provide services to user thread for background supporting

Concurrency Control

- Concurrency is the ability to run several programs or several parts of a program in parallel. If a time consuming task can be performed asynchronously or in parallel, this improve the throughput and the interactivity of the program. A modern computer has several CPU's or several cores within one CPU

Thread Starvation

- If a high priority thread never blocks then all other threads will <u>starve</u>
- Must be judicious
- Example
  – How to decide right priority

Runnable Vs Callable

- Run method does not return anything
- Callable can return data

```
Callable<T>{
  public T call(){
} }
```

- Rather than passed to Thread objects, Callable classes are launch using the submit(Callable) method of an ExecutorService

Futures

- *ExecutorService* interface also provides

```
public interface ExecutorService {
  …
  Future<T> submit(Callable<T> task);
  Future<?> submit(Runnable task);
  Future<T> submit(Runnable task, T result);
}
```

Race Conditions

- Two threads are simultaneously modifying a single object
- Both threads “race” to store their value
- In the end, the last one there “wins the race”
- (Actually, both loose)

Race Condition Example

```
class Account {
  int balance;
  public void deposit(int val)
  {
    int newBal;
    newBal = balance + val;
    balance = newBal;
  }
}
```

Thread Synchronization

- Language keyword: `synchronized`
- Takes out a monitor lock on an object
  - Exclusive lock for that thread
- If lock is currently unavailable, thread will block

Thread Synchronization

- Protects access to code, not to data
  - Make data members private
  - Synchronize accessor methods
- Puts a “force field” around the locked object so no other threads can enter
  - Actually, it only blocks access to other synchronizing threads

Thread Deadlock

- If two threads are competing for more than one lock
- Example: bank transfer between two accounts
  - Thread A has a lock on account 1 and wants to lock account 2
  - Thread B has a lock on account 2 and wants to lock account 1

Avoiding Deadlock

- No universal solution
- Ordered lock acquisition
- Encapsulation (“forcing directionality”)
- Spawn new threads
- Check and back off
- Timeout
- Minimize or remove synchronization

Synchronization Constraints

- Synchronized keyword is quite rigid in its use. E.g. a thread can take a lock only once. Synchronized blocks don’t offer any mechanism of a waiting queue and after the exit of one thread, any thread can take the lock. This could lead to starvation of resources for some other thread for a very long period of time.

Reentrant Locks

- The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the shared resource is surrounded by calls to lock and unlock method. This gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.

Reentrant Locks

- ReentrantLock allow threads to enter into lock on a resource more than once. When the thread first enters into lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlock request, hold count is decremented by one and when hold count is 0, the resource is unlocked.

Reentrant Locks

- Reentrant Locks also offer a fairness parameter, by which the lock would abide by the order of the lock request i.e. after a thread unlocks the resource, the lock would go to the thread which has been waiting for the longest time. This fairness mode is set up by passing true to the constructor of the lock.

Reentrant Locks Vs Synchronization

- ReentrantLock is a better replacement for synchronization, which offers many features not provided by synchronized. However, the existence of these obvious benefits are not a good enough reason to always prefer ReentrantLock to synchronized. Instead, make the decision on the basis of whether you need the flexibility offered by a ReentrantLock.

CountDownLatch

- CountDownLatch is used to make sure that a task waits for other threads before it starts.
- E.g. a server where the main task can only start when all the required services have started.
- CountDownLatch latch = new CountDownLatch(4);

CountDownLatch

- Creating an object of CountDownLatch by passing an int to its constructor (the count), is actually number of invited parties (threads) for an event.
- The thread, which is dependent on other threads to start processing, waits on until every other thread has called count down. All threads, which are waiting on await() proceed together once count down reaches to zero

CyclicBarrier

- Used to make threads wait for each other.
- Used when different threads process a part of computation and when all threads have completed the execution, the result needs to be combined in the parent thread.
- Used when multiple thread carry out different sub tasks and the output of these sub tasks need to be combined to form the final output.
- After completing its execution, threads call await() method and wait for other threads to reach the barrier. Once all the threads have reached, the barriers then give the way for threads to proceed.

Working of CyclicBarrier

- Create new instance
  - CyclicBarrier newBarrier = new CyclicBarrier(numberOfThreads);
- Working
  - Each and every thread does some computation and after completing it's execution, calls await() methods as shown:

```
public void run() {
    // thread does the computation
    newBarrier.await();
}
```

CyclicBarrier vs CountDownLatch

- CyclicBarrier allows a number of threads to wait on each other, whereas CountDownLatch allows one or more threads to wait for a number of tasks to complete.
- In short, CyclicBarrier maintains a count of threads whereas CountDownLatch maintains a count of tasks.

Phaser (java.util.concurrent.Phaser)

- Phaser’s primary purpose is to enable synchronization of threads that represent one or more phases of activity. It lets us define a synchronization object that waits until a specific phase has been completed. It then advances to the next phase until that phase concludes. It can also be used to synchronize a single phase, and in that regard, it acts much like a CyclicBarrier.

Phaser

- Create a phaser
  - – public Phaser()
  - – public Phaser(int parties) throws IllegalArgumentException
- Working
  - – phaser.register()
  - – phaser.arriveAndAwaitAdvance();
  - – phaser.arriveAndDeregister();

Phaser – Important Methods

- int register()
  - register parties after a phaser has been constructed. It returns the phase number of the phase to which it is registered.
- int arrive()
  - signals that a thread has completed some portion of the task. It does not suspend the execution of the calling thread. It returns the current phase number or a negative value if the phaser has been terminated.
- int arriveAndDeregister()
  - enables a thread to arrive at a phase and deregister itself, without waiting for other threads to arrive. It returns the current phase number or a negative value if the phaser has been terminated.

Phaser – Important Methods

- int arriveAndAwaitAdvance()
  - suspends the execution of the thread at a phase, to wait for other threads. It returns the current phase number or a negative value if the phaser has been terminated.
- final int getPhase()
  - returns the current phase number. A negative value is returned if the invoking phasers terminated.
- boolean onAdvance(int phase, int parties)
  - helps in defining how a phase advancement should occur. To do this, the user must override this method. To terminate the phaser, onAdvance() method returns true, otherwise, it returns false;

Synchronization and Collections

- ArrayList, LinkedList, HashSet,LinkedHashset and TreeSet in Collection Interface and HashMap, LinkedHashMap and Treemap are all non-synchronized. All collection classes (except Vector and Hashtable) in the java.util package are not thread-safe. The only two legacy collections are thread-safe: Vector and Hashtable

Synchronization and Collections

- Java platform provides strong support for this scenario through different synchronization wrappers implemented within the Collections class.
- These wrappers make it easy to create synchronized views of the supplied collections by means of several static factory methods

Synchronized Collections

- Synchronized collections achieve thread-safety through intrinsic locking, and the entire collections are locked.
- Intrinsic locking is implemented via synchronized blocks within the wrapped collection's methods.

```
Collection<Integer> syncCollection =
            Collections.synchronizedCollection(new ArrayList<>());
List<Integer> syncList =
            Collections.synchronizedList(new ArrayList<>());
Map<Integer, String> syncMap =
            Collections.synchronizedMap(new HashMap<>());
```

A Fast AND Steady Approach

Executor Framework

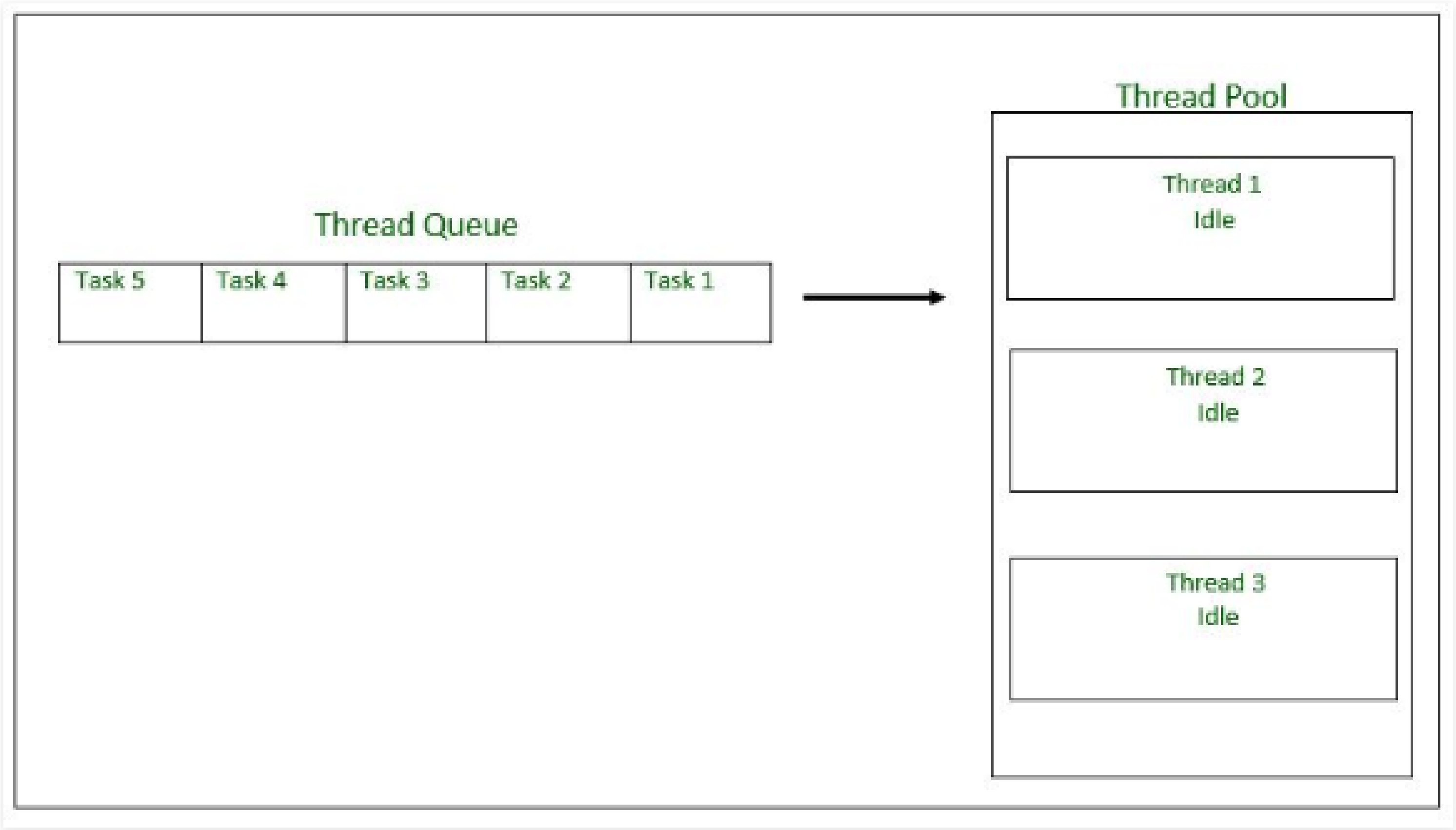
Need of Executor Framework

- Server Programs such as database and web servers repeatedly execute requests from multiple clients and these are oriented around processing a large number of short tasks.
- An approach for building a server application would be to create a new thread each time a request arrives and service this new request in the newly created thread.
  - If server creates a new thread for every request, it would spend more time and consume more system resources in creating and destroying threads than processing actual requests.

Thread Pool

- A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing
- Executor Framework
  - centered around the Executor interface, its sub-interface ExecutorService and the class ThreadPoolExecutor, which implements both of these interfaces.
  - one only has to implement the Runnable objects and send them to the executor to execute.
  - Take advantage of threading, but focus on the tasks that you want the thread to perform, instead of thread mechanics.

Thread Pool


Thread Queue
Task 5
Task 4
Task 3
Task 2
Task 1
Thread Pool
Thread 1
Idle
Thread 2
Idle
Thread 3
Idle

Provided Thread Pool Executors

- `newFixedThreadPool`
  - Bounded size
  - Replace if thread dies
- `newCachedThreadPool`
  - Demand driven variable size
- `newSingleThreadExecutor`
  - Just one thread
  - Replace if thread dies
- `newScheduledThreadPool`
  - Delayed and periodic task execution
  - Good replacement for class `Timer`

Executor Shut-down

- Executor is a service provider
- Executor abstracts thread management
- To maintain the abstraction, the service should generally provide methods for shutting down the service
- JVM cannot shut down until threads do

ExecutorService Notes

- Implies three states:
  - Running
  - Shutting down
  - Terminated
- `shutdown()` runs tasks in queue
- `shutdownNow()` returns unstarted tasks
- `awaitTermination()` blocks until the service is in the terminated state

Risks in using Thread Pools

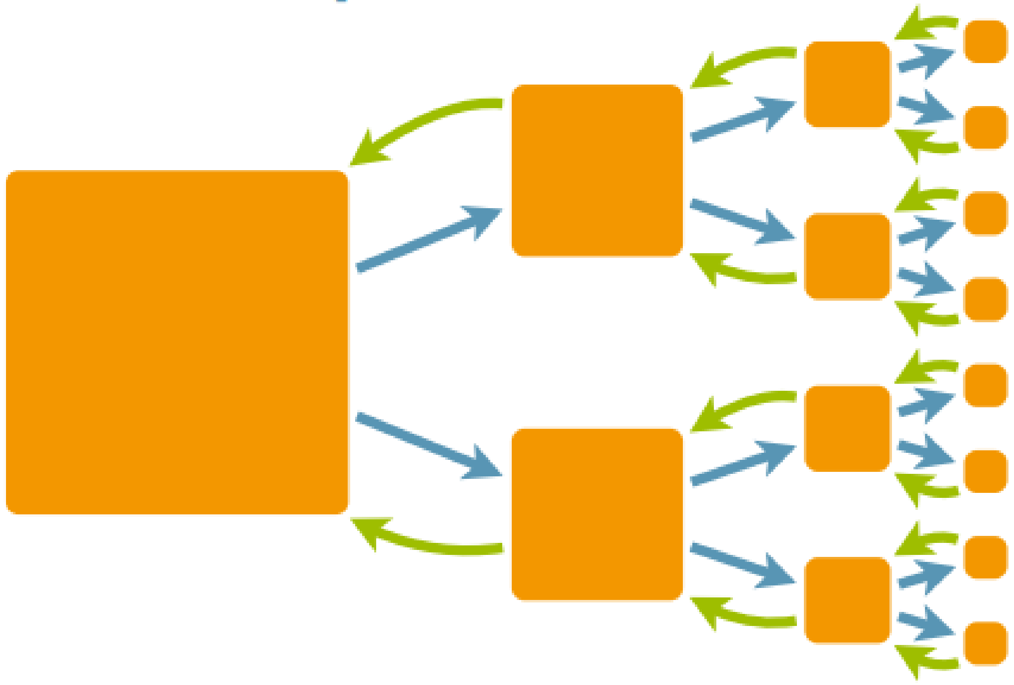
- Deadlock (in addition to normal)
  - All the executing threads are waiting for the results from the blocked threads waiting in the queue due to the unavailability of threads for execution.
- Thread Leakage
  - Occurs if a thread is removed from the pool to execute a task but not returned to it when the task completed. A thread throws an exception and pool class does not catch this exception, then the thread will simply exit, reducing pool size by one.
- Resource Thrashing
  - If the thread pool size is very large then time is wasted in context switching between threads. Having more than the optimal number of threads may cause starvation problem leading to resource thrashing.

ForkJoinPool Framework

- Introduced in Java 7.
- Aim is to speed up parallel processing by attempting to use all available processor cores using a “divide and conquer” approach.
- framework first “forks”, recursively breaking the task into smaller independent subtasks until they are simple enough to be executed asynchronously.
- In “join” part, results of all subtasks are recursively joined into a single result, or in the case of a task which returns void, the program simply waits until every subtask is executed.

Fork & Join

Divide & conquer



ORACLE

Working of ForkJoinPool

- Worker threads can execute only one task at the time, but the ForkJoinPool doesn’t create a separate thread for every single subtask. Instead, each thread in the pool has its own double-ended queue (or deque, pronounced deck) which stores tasks.
- This architecture is vital for balancing the thread’s workload with the help of the work-stealing algorithm.

Performance considerations

- Choosing the sequential threshold
  - Smaller tasks increase parallelism
  - Larger tasks reduce coordination overhead
  - Ultimately you must profile your code

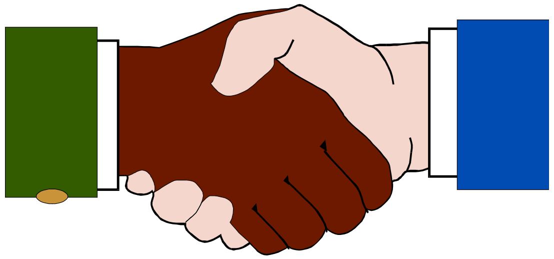
fands™

QUESTION / ANSWERS

www.fandsindia.com

fands™

THANKING YOU !



www.fandsindia.com