Core Java 8 and Development Tools

Lesson 20 : Multithreading

Capgemini

Lesson Objectives

After completing this lesson, participants will be able to

- Understanding threads
- Thread life cycle
- Scheduling threads- Priorities
- Controlling threads using sleep(),join()
- Synchronization concept
- Inter Thread Communications
- Implementations of wait(),notify(),notifyAll() in Producer Consumer problem

20.1: Understanding Threads?
Thread and Process

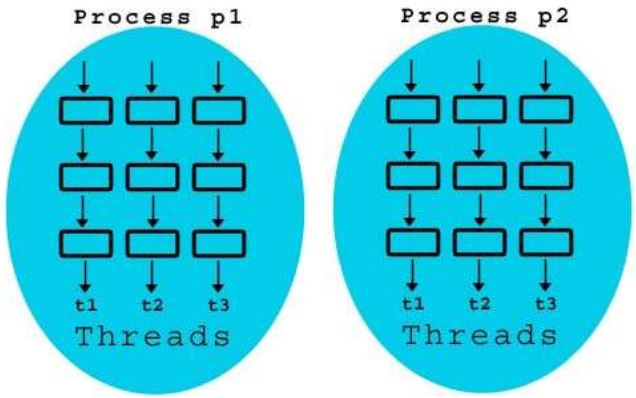
Thread

- A thread is a single sequential flow of control within a process and it lives within the process
- A light weight process which runs under resources of main process
- Inter process Communication is always slower than intra process communication
- Actual thread execution highly depends on OS and hardware support.
- The JVM of Thread-non-supportive OS takes care of thread execution.
- On single processor, threads may be executed in time sharing manner

20.1: Understanding Threads?
Thread Application

Applications of Multithreading

- Playing media player while doing some activity on the system like typing a document.
- Transferring data to the printer while typing some file.
- Running animation while system is busy on some other work
- Honoring requests from multiple clients by an application/web server.



20.1: Understanding Threads?
Create Thread using Extending Thread

Extending Thread class to create threads :

- Inherited class should:
  - Override the *run()* method.
  - Invoke the *start()* method to start the thread.

```
public class HelloThread extends Thread     {

            public void run(){

              System.out.println("Hello ..  Welcome to Capgemini.");

             }
          public static void main(String… args)  {
              new HelloThread().start();
           }
}
```

20.1: Understanding Threads?
Creating Thread Implementing Runnable

Another way to Create Thread.

- Create a class that implements the runnable interface.
- Class to implement only the run method that constitutes the new thread.

```
public class HelloRunnable implements Runnable {

@Override
public void run() {
        System.out.println("Hello .. Welcome to Capgemini ..");
}

}}
```

```
public static void main(String… args){

        HelloRunnable   hello = new HelloRunnable();
         Thread     helloThread = new Thread(hello);

          helloThread . start();
}
```

20.1: Understanding Threads?
Thread Extending Vs. Implementing

Extending Thread	Implementing Runnable
Basically for creating worker thread.	Basically for defining task.
It itself is a Thread. Simple syntax	Thread object wraps Runnable object
Can not extend any other class	Can extend any other class
A functionality is executed only once on a thread instance.	A functionality can be executed more than once by multiple worker threads.
Concurrent framework does have limited support.	Concurrent framework provide extensive support.
Thread's life cycle methods like interrupt() can be overridden.	Only run() method can be overridden.

20.1: Understanding Threads?
Thread

Thread API :

- Thread Class
  - run()
  - start() --- It causes this thread to begin execution; the JVM calls the run method of this thread.
  - sleep()
  - join()
  - stop() (It is a Deprecated method .).
  - getName() - It returns the Thread name in string format.
  - isAlive() -- It returns thread is alive or not .
  - currentThread() - It returns the current Thread object.

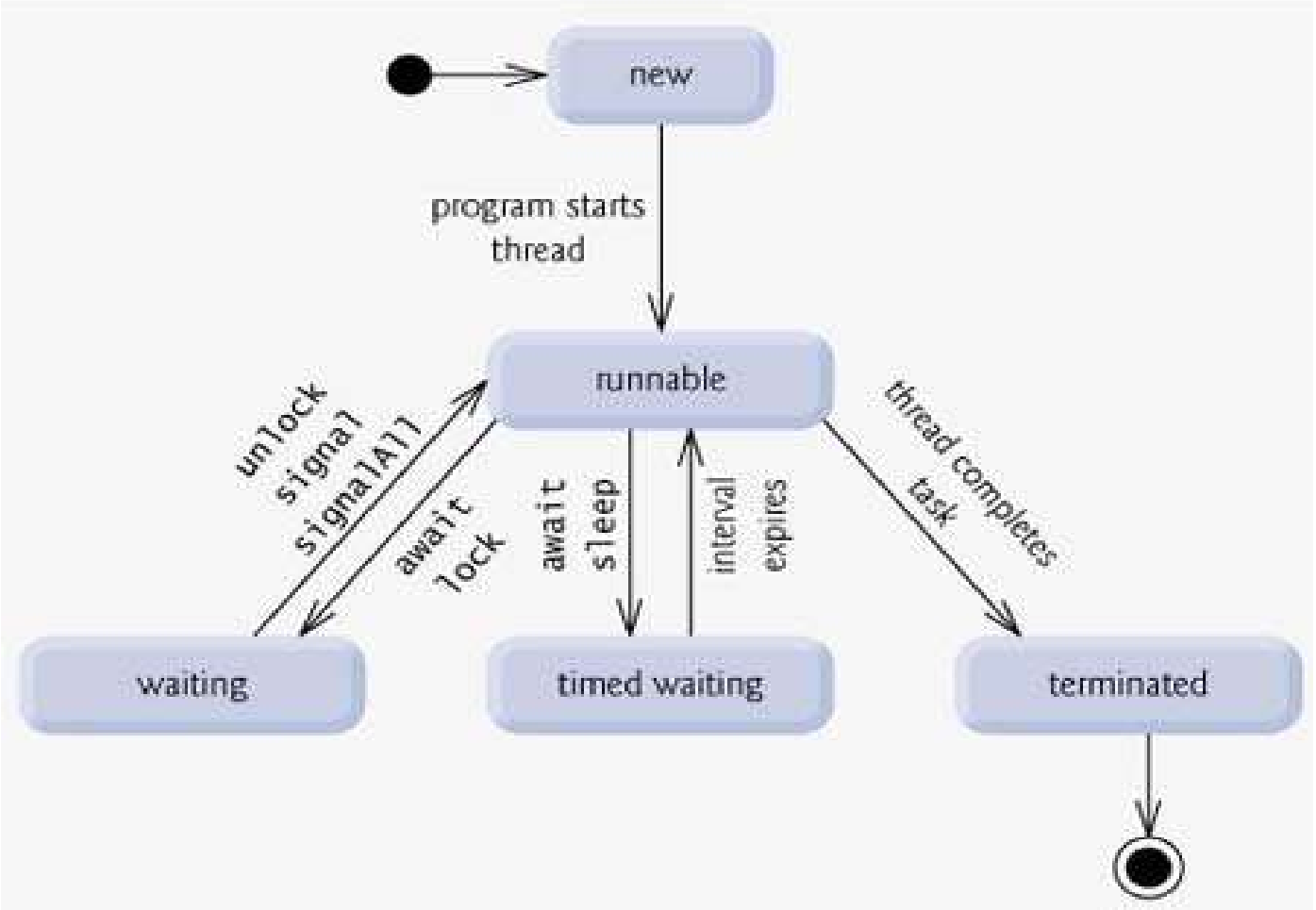
Runnable Interface

- run() - This is the only one method available in this interface .
- Common Exceptions in Threads:
- InterruptedException .
- IllegalStateException

20.1: Understanding Threads? Demo

HelloThread.java
HelloRunnable.java

20.2: Thread Life cycle
Thread Life Cycle



20.3: Scheduling threads- Priorities
Thread Priorities

Thread runs in a priority level . Each thread has a priority which is a number starts from 1 to 10 .

- Thread Scheduler schedules the threads according to their priority .

There are three constant defined in Thread class

- MIN_PRIORITY
- NORM_PRIORITY

MAX_PRIORITY

- Method which is used to set the priority

setPriority(PRIORITY_LEVEL);

Do not rely on thread priorities when you design your multithreaded application .

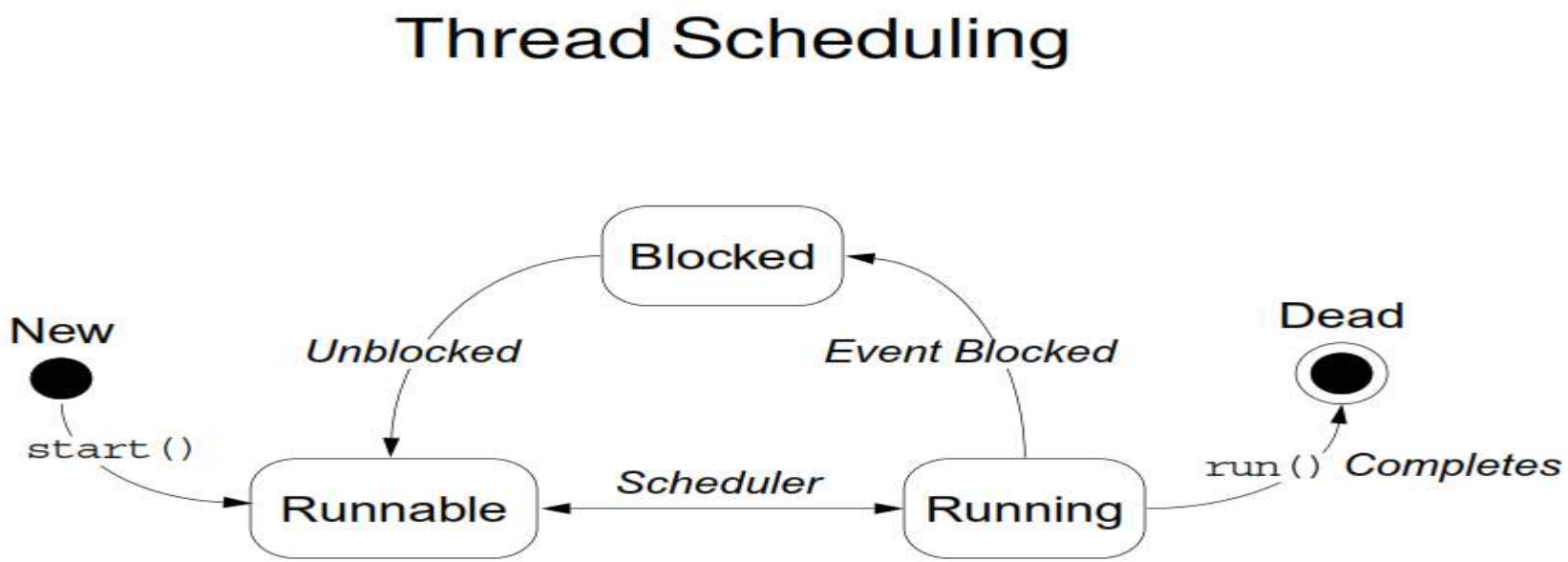
20.3: Scheduling threads- Priorities Demo

ThreadPriorityDemo

20.4: Controlling threads using sleep(),join()
Threads in Blocked Stage

Number of ways a thread can be brought to blocked state.

1. By calling sleep() method
2. By calling suspend() method
3. By calling wait() method



20.4: Controlling threads using sleep(),join()
Controlling thread using sleep()

The Thread class provides two methods for sleeping a thread:

- public static void sleep(long milliseconds)throws InterruptedException
- public static void sleep(long milliseconds, int nanos)throws InterruptedException

20.2: Thread Life cycle Demo

ThreadLifeCycleDemo.java

20.4: Controlling threads using sleep(),join()
Controlling Thread using join()

In Tread join method can be used to pause the current thread execution until unless the specified thread is dead.

There are three overloaded join functions.

- **public final void join()**
- **public final synchronized void join(long millis)**
- **public final synchronized void join(long millis, int nanos)**

20.4: Controlling threads using sleep(),join() Demo :

ThreadJoinDemo . Java
SleepDemo.java

20.5: Synchronization concept
Problems with Shared Data

Shared data must be accessed cautiously.
Instance and static fields:

- Are created in an area of memory known as heap space
- Can potentially be shared by any thread
- Might be changed concurrently by multiple threads
  - There are no compiler or IDE warnings.
  - “Safely” accessing shared fields is developer’s responsibility.

Without Synchronization Demo

```
public class Display {
   public void wish(String name) {
      for (int i = 0; i < 10; i++) {
         System.out.print("Hello ");
         try {
            Thread.sleep(2000);
         } catch (InterruptedException ie) {
            ie.printStackTrace();
         }
         System.out.println(name);
      }
   }
}
```

```
public class MyThread extends Thread {
   private Display display;
   private String name;
   public MyThread(Display display,String name) {
      this.display=display;
      this.name=name;
   }
   public void run(){
      display.wish(name);
   }
}
```


```
public class Client {
   public static void main(String[] args) {
      Display display=new Display();
      MyThread myThread1=new MyThread(display, "Dhoni");
      MyThread myThread2=new MyThread(display, "Kohli");
      myThread1.start();
      myThread2.start();
   }
}
```

Without Synchronization Demo output


```
Hello Hello Kohli
Hello Dhoni
Hello Dhoni
Hello Kohli
Hello Dhoni
Hello Kohli
Hello Kohli
Hello Dhoni
Hello Kohli
Dhoni
Hello Hello Dhoni
Kohli
Hello Hello Dhoni
Hello Kohli
Hello Kohli
Hello Dhoni
Hello Kohli
Dhoni
Hello Hello Dhoni
Kohli
```

20.5: Synchronization concept
Multithreading without Synchronization

NonSyncDemo

20.5: Synchronization concept
Synchronization

If multiple threads are trying to operate simultaneously on given java object then there may be a chance of data inconsistency problem.

To over come this problem one should go for SYNCHRONIZED keyword. The main advantage of synchronized keyword is we can overcome data inconsistency problem

In synchronized method the whole object gets locked with thread .
in synchronized block , the lock occurred for a particular resource

Synchronization makes a program to work very slow as only one thread can access the data at a time

20.5: Synchronization concept
Object Monitor Locking

Every object in Java has a lock

Using synchronization enables the lock and allows only one thread to access that part of code.

Each object in Java is associated with a monitor, which a thread can lock or unlock.

20.5: Synchronization concept
Object Monitor Locking

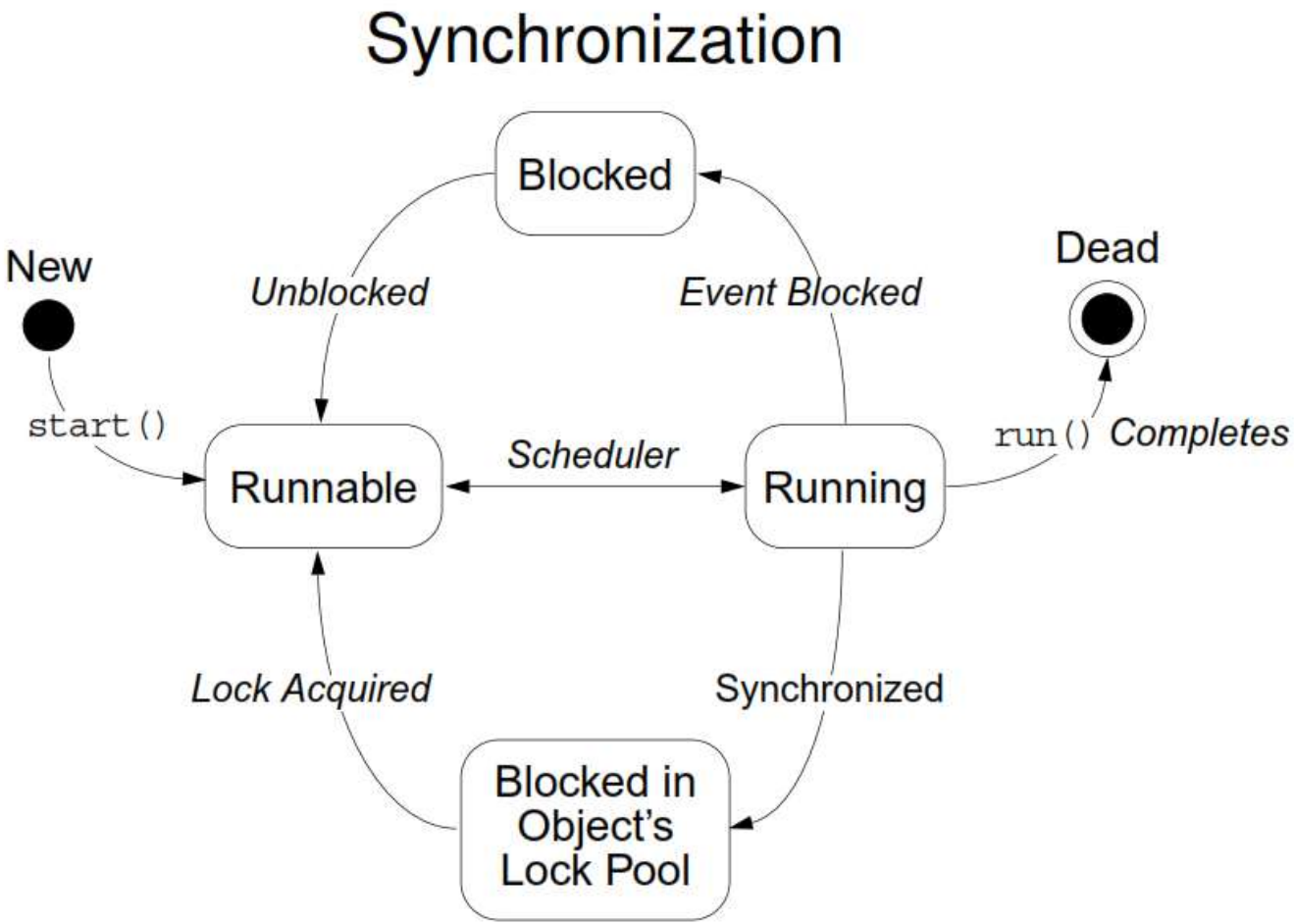
Each object in Java is associated with a monitor, which a thread can lock or unlock.

- synchronized methods use the monitor for the this object.
- static synchronized methods use the classes' monitor.
- synchronized blocks must specify which object's monitor to lock or unlock and it can be nested .

```
synchronized ( this ) {

}
```

20.5: Synchronization concept
Synchronization



Synchronized Method

The synchronized keyword in the method declaration. This tells Java that the method is synchronized.

A synchronized instance method in Java is synchronized on the instance (object) owning the method.

```
public synchronized void add(int value){
    this.count += value;
 }
```

20.5: Synchronization concept
Synchronized Method

Implementation of concurrency control mechanism is very simple because every Java object has its own implicit monitor associated with it.

If a thread wants to enter an object's monitor, it has to just call the synchronized method of that object

While a thread is executing a synchronized method, all other threads that are trying to invoke that particular synchronized method or any other synchronized method of the same object, will have to wait

With Synchronization – Method Level Demo

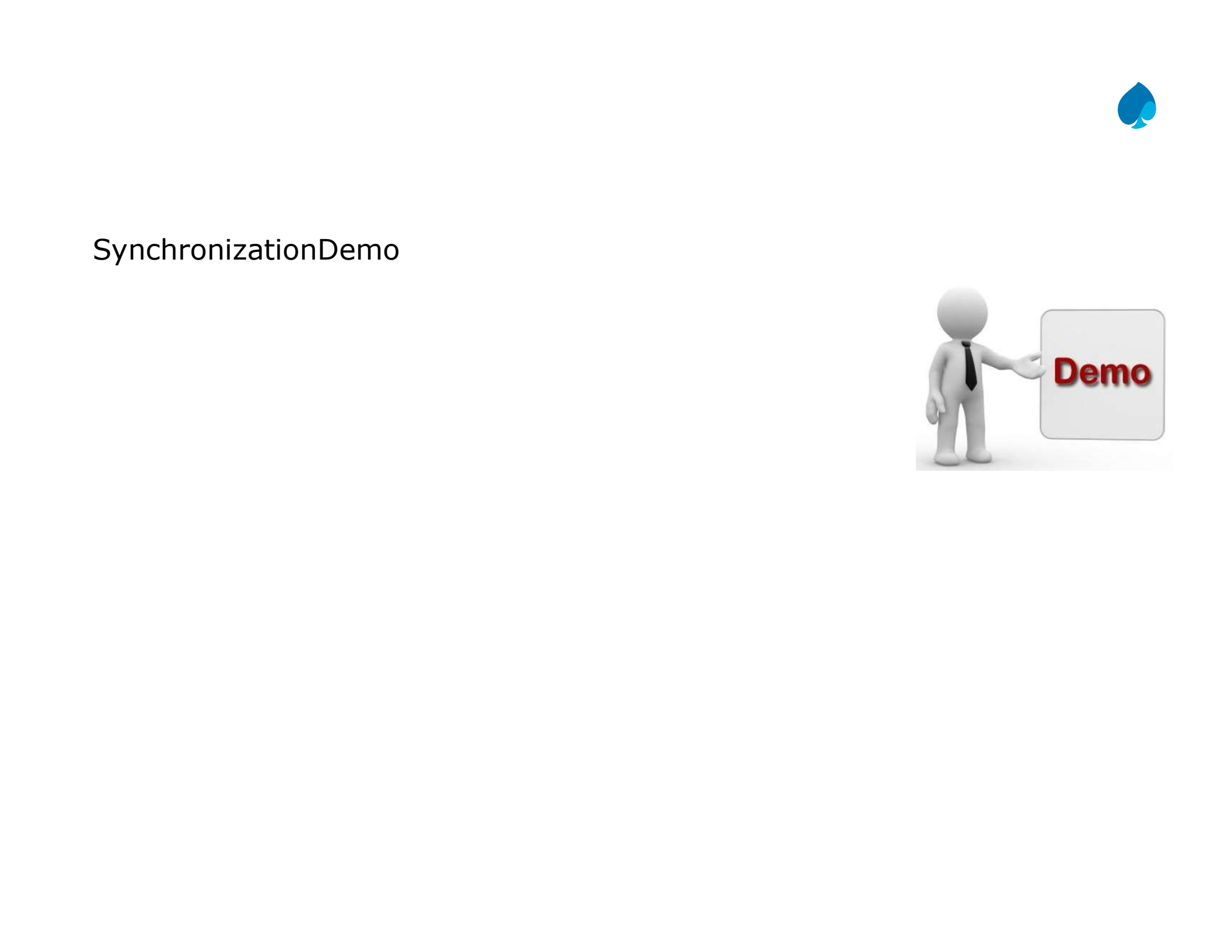
```
public class Display {
   public  synchronized void wish(String name) {
      for (int i = 0; i < 10; i++) {
         System.out.print("Hello ");
         try {
            Thread.sleep(2000);
         } catch (InterruptedException ie) {
            ie.printStackTrace();
         }
         System.out.println(name);
      }
   }
}
```

```
public class MyThread extends Thread {
   private Display display;
   private String name;
   public MyThread(Display display,String name) {
      this.display=display;
      this.name=name;
   }
   public void run(){
      display.wish(name);
   }
}
```


```
public class Client {
   public static void main(String[] args) {
      Display display=new Display();
      MyThread myThread1=new MyThread(display, "Dhoni");
      MyThread myThread2=new MyThread(display, "Kohli");
      myThread1.start();
      myThread2.start();
   }
}
```

With Synchronization – Method Level Demo output

```
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Kohli
Hello Kohli
Hello Kohli
Hello Kohli
Hello Kohli
Hello Kohli
Hello Kohli
Hello Kohli
Hello Kohli
Hello Kohli
```



SynchronizationDemo

20.5: Synchronization concept
With Synchronization – Static Method Level

The synchronized keyword in the method declaration. This tells Java that the method is synchronized.

Synchronized static methods are synchronized on the class object of the class where the synchronized static method present.

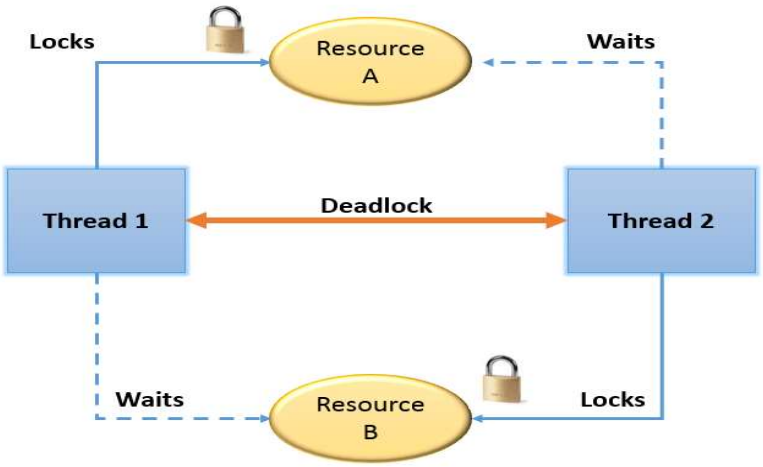
```
public static synchronized void add(int value){
      count += value;
  }
```

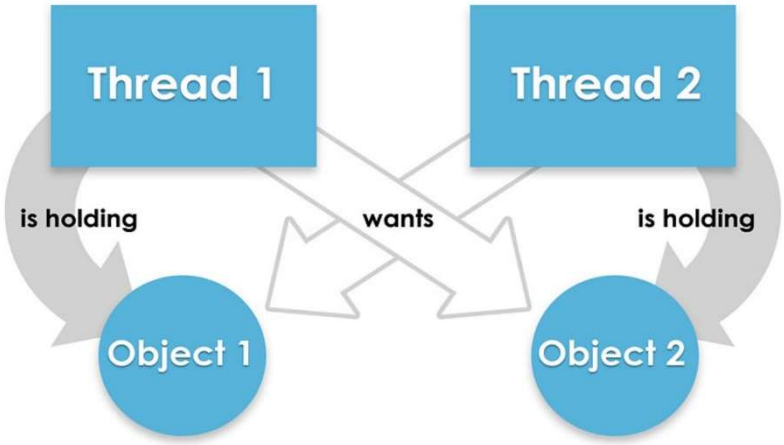
20.6: Inter thread communication
Deadlock

Deadlock results when two or more threads are blocked forever, waiting for each other.

A deadlock has the below characteristics

- It is two threads , each waiting for a lock from the other.
- It is not detected or avoided





20.6: Inter thread communication
Inter-Thread Communication

When more than one thread uses a shared resource they need to synchronize with each other.

While using a shared resource the threads need to communicate with each other, to get the expected behaviour of the application.

Java provides some methods for the threads to communicate

20.7: Implementations of wait(),notify(),notifyAll() Inter-Thread Communication

Methods for Interthread Communication
`public final void wait()`
Causes this thread to wait until some other thread calls the *notify* or *notifyAll* method on this object. May throw *InterruptedException*.
`public final void notify()`
Wakes up a thread that called the *wait* method on the same object.
`public final void notifyAll()`
Wakes up all threads that called the *wait* method on the same object.

20.7: Implementations of wait(),notify(),notifyAll() Inter-Thread Communication

Threads are often interdependent.

Wait() and notify() encapsulate the two central concepts to thread communication .

To avoid polling, Java’s elegant inter-thread communication mechanism uses:
wait( ) ,
notify( ), and
notifyAll( )

20.7: Implementations of wait(),notify(), Inter-Thread Communication (Contd.).

wait( ) : directs the calling thread to surrender the monitor, and go to sleep until some other thread enters the monitor of the same object, and calls notify( )

notify( ) : wakes up the other thread which was waiting on the same object(that had called wait() previously on the same object)

20.7: Implementations of wait(),notify()
Producer-Consumer Problem

Producing thread may write to buffer (shared memory)
Consuming thread reads from buffer
If not synchronized, data can become corrupted

- Producer may write before consumer read last data
  - Data lost
- Consumer may read before producer writes new data
  - Data "doubled"

20.7: Implementations of wait(),notify() Producer-Consumer Problem (Contd.).

Using synchronization

- If producer knows that consumer has not read last data, calls wait (awaits a notify command from consumer)
- If consumer knows producer has not updated data, calls wait (awaits notify command from producer)

20.7: Implementations of wait(),notify() Producer-Consumer Problem (Contd.).

Incorrect Implementation

```
synchronized int get() {
System.out.println("Got: " + n);
return n;
}
synchronized void put(int n) {
this.n = n;
System.out.println("Put: " + n);
}
```

20.7: Implementations of wait(),notify() Producer-Consumer Problem (Contd.).

```
Put: 0
Put: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Put: 8
Put: 9
Put: 10
Put: 11
Put: 12
Put: 13
Put: 14
Put: 15
Put: 16
Put: 17
Put: 18
Put: 19
Put: 20
Put: 21
Put: 22
Put: 23
Put: 24
```

Only Producer is doing work

```
Put: 164
Put: 165
Put: 166
Put: 167
Put: 168
Put: 169
Put: 170
Put: 171
Got: 171
Put: 172
Got: 172
Put: 173
Got: 173
Put: 174
Got: 174
Put: 175
Got: 175
Got: 175
Got: 175
Got: 175
Got: 175
Got: 175
```

Though producer and consumer are working, there is no sync between them

20.7: Implementations of wait(),notify() Producer-Consumer Problem (Contd.).

Correct Implementation

```
synchronized int get() {
if(!valueSet)
try {
wait();
} catch(InterruptedException e)
{ System.out.println("InterruptedExce
ption caught");
}
System.out.println("Got: " + n);
valueSet = false;
notify();
return n;
}
```

20.7: Implementations of wait(),notify() Producer-Consumer Problem (Contd.).

```
Put: 0
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
Put: 6
Got: 6
Put: 7
Got: 7
Put: 8
Got: 8
Put: 9
Got: 9
Put: 10
Got: 10
Put: 11
Got: 11
Put: 12
Got: 12
Put: 13
Got: 13
Put: 14
Got: 14
```

put and get are in synch

20.7: Implementations of wait(),notify() Inter-Thread Communication (Contd.).

The following sample program implements a simple form of the Producer/Consumer problem

It consists of four classes namely:

- Factory , that you are trying to synchronize
- Producer, the threaded object that is producing data
- Consumer, the threaded object that is consuming data
- Main, the class that creates the single Factory, Producer, and Consumer

20.7 Implementations of wait(),notify() Demo

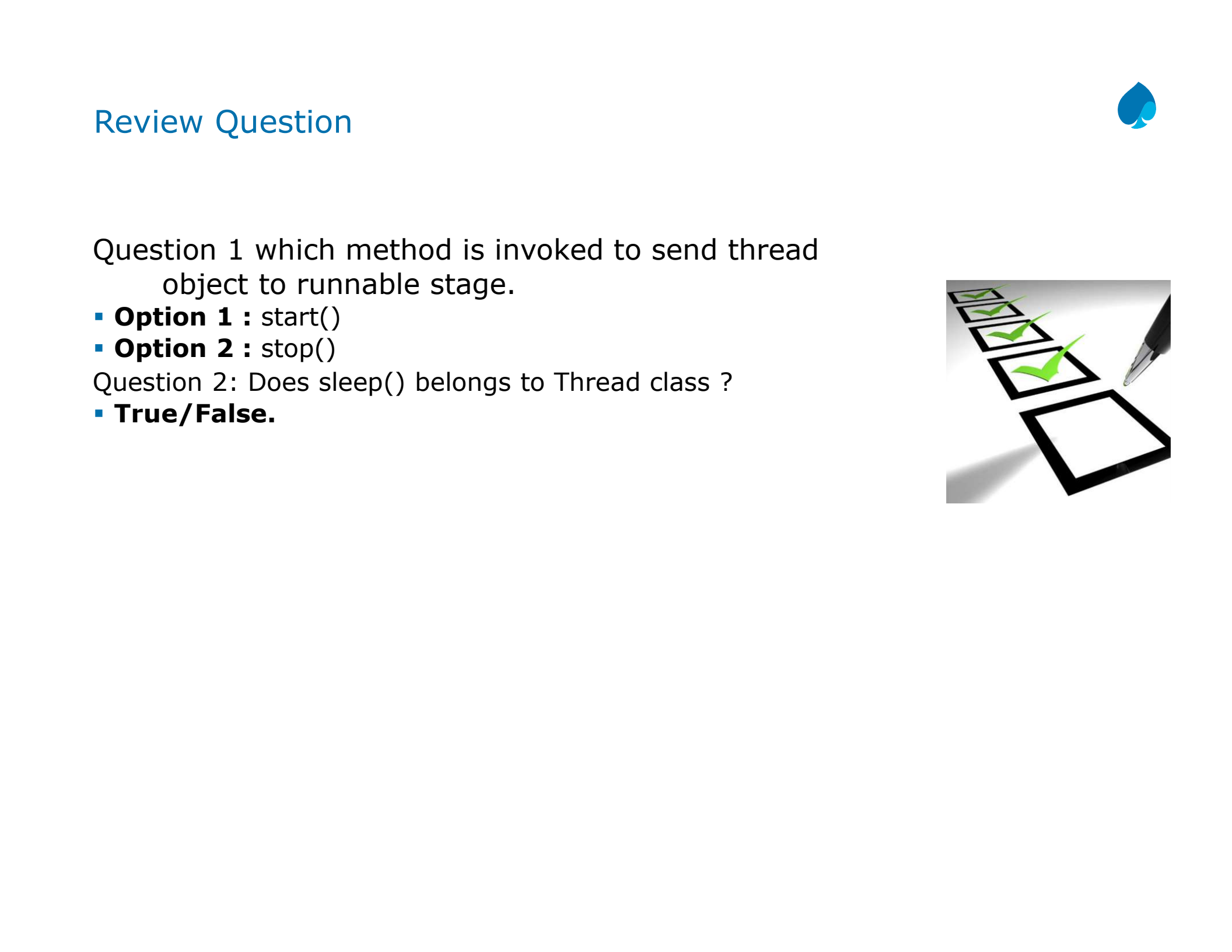
Factory.java
Producer.java
Consumer.java
Main.java

Summary

In this lesson, you have learnt the following:

- What is Thread and use of Multithread
- how to create a Thread program and Lifecycle?
- Thread Priorities Implementation in Multi Threading environment .
- Use of sleep() , join()
- Synchronization concept
- Different types of Synchronization
- Inter thread communication
- Producer – consumer problem by using wait(),notify(), notifyAll()

Review Question



Question 1 which method is invoked to send thread object to runnable stage.

- **Option 1 :** start()
- **Option 2 :** stop()

Question 2: Does sleep() belongs to Thread class ?

- **True/False.**

Review Question

Question 3 which of the following is not a type of synchronization.

- **Option 1 :** by block
- **Option 2 :** by method()
- **Option 3 :** by variable

Question 4: Which of the following variable types are never shared.

- **Option 1:** Local variables**.**
- **Option 2 :** Instance variables
- **Option 3 :** Method parameters