JAVA - Threads

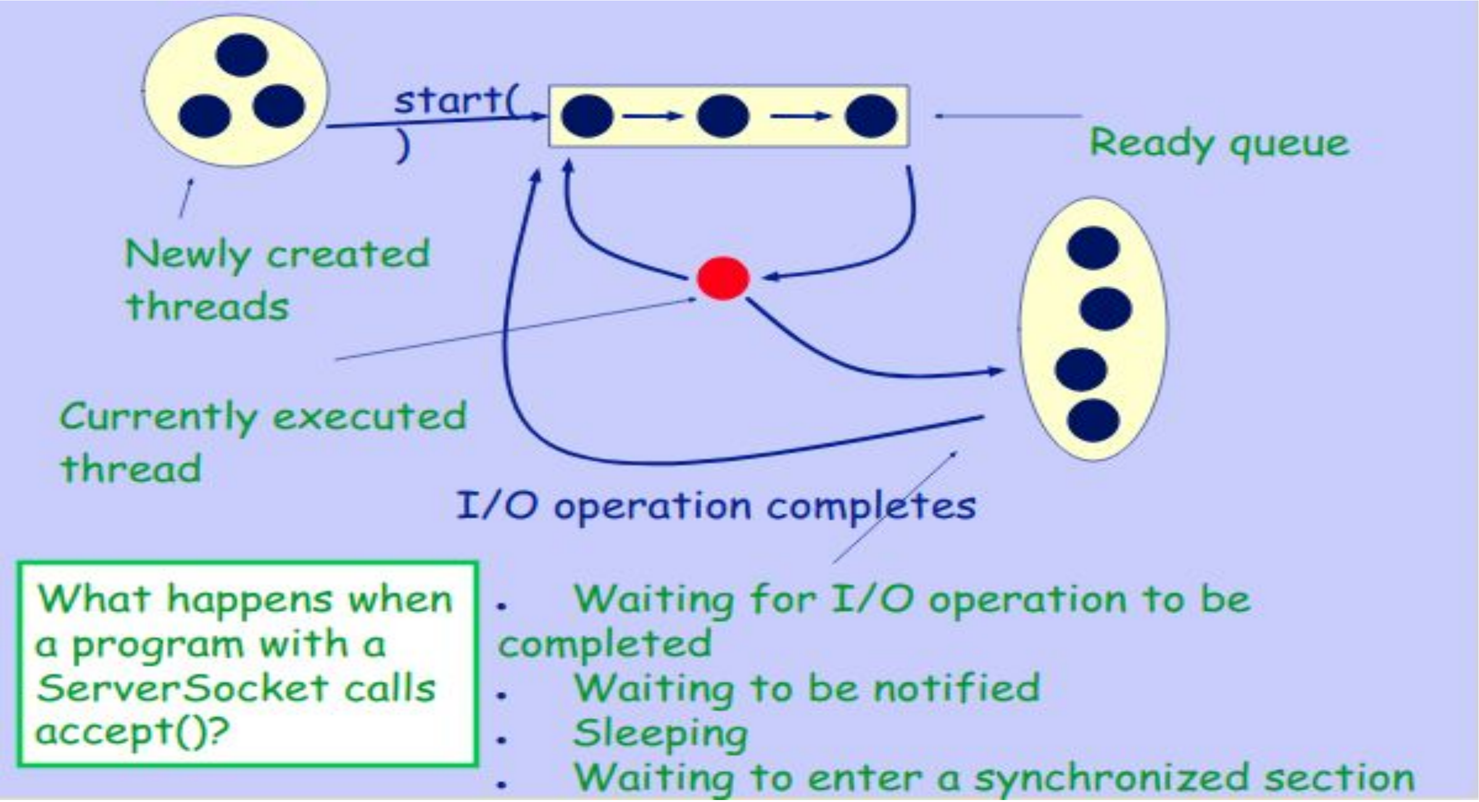
Topics

- ❖ Java Concurrency: Using threads in Java, Life cycle of thread
- ❖ Advantages and issues
- ❖ Thread class, thread groups
- ❖ The Runnable interface
- ❖ Synchronizing, Inter-Thread communication Parallel Fork/Join Framework

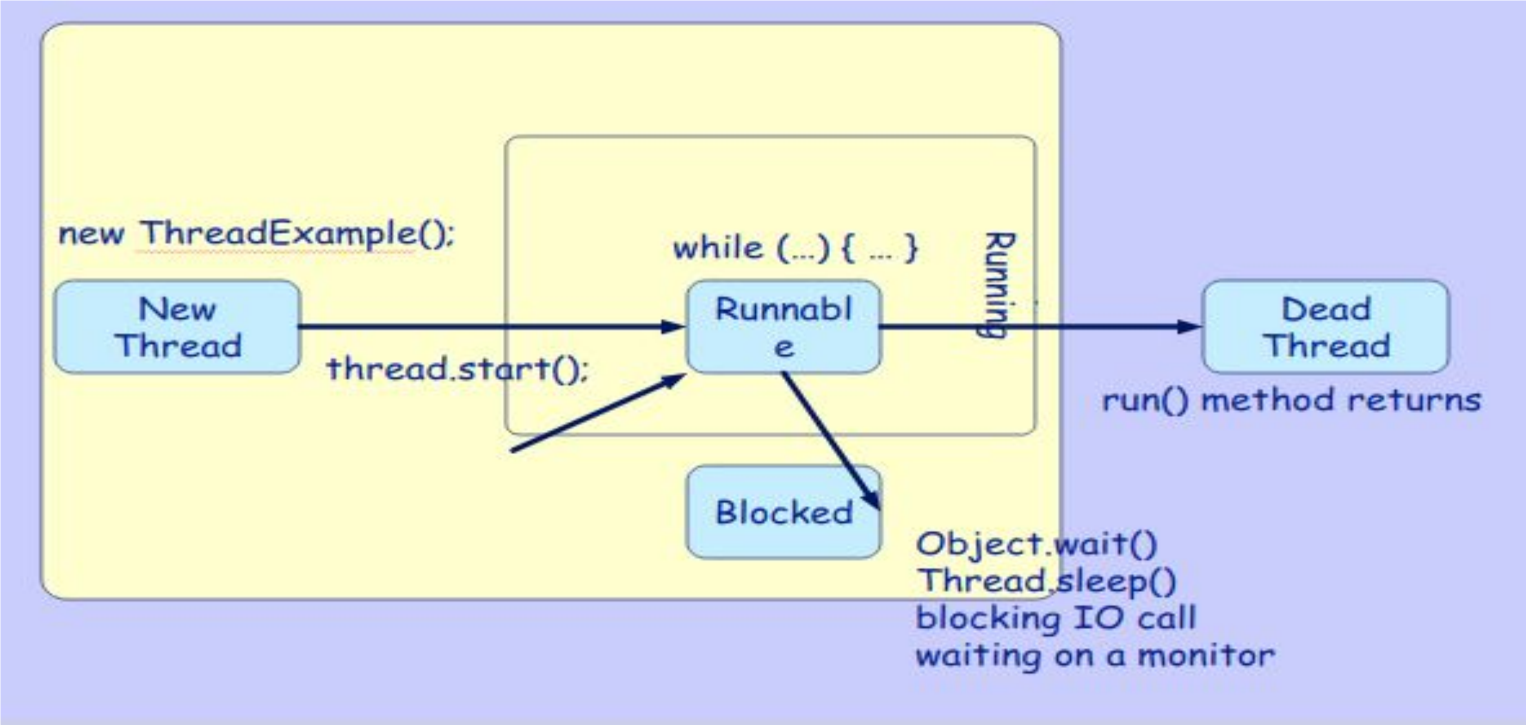
Threads

- A thread is a single sequence of execution within a program , refers to multiple threads of control within a single program.
- Each program can run multiple threads of control within it, e.g., Web Browser
- Each thread has its private run-time stack
- If two threads execute the same method, each will have its own copy of the local variables the methods uses
- However, all threads see the same dynamic memory, i.e., heap (are there variables on the heap?)
- Two different threads can act on the same object and same static fields concurrently

Thread Lifecycle



Thread State Diagram



```
Runnable task = () -> {
    try {
        String name = Thread.currentThread().getName();
        System.out.println("Foo " + name);
        Thread.sleep(10000);
        System.out.println("Bar " + name);
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
};
Thread thread = new Thread(task);
thread.start();
```

Creating a Threads

1) extends the Thread class
2) Implements Runnable interface

```
public class ThreadExample extends Thread {
  public void run () {
    for (int i = 1; i <= 100; i++) {
      System.out.println("---");
    }
  }
}
```

Thread Methods

- ❖ void start()
- ❖ Void run()
- ❖ Void stop()
- ❖ Void yield()
- ❖ Void sleep(int milliseconds)

Runnable Interface Implementation

```
public class RunnableExample implements Runnable {
        public void run () {
                for (int i = 1; i <= 100; i++) {
                        System.out.println ("***");
      }
   }
}
```

```
public class ThreadsStartExample {
        public static void main (String argv[]) {
          new ThreadExample ().start ();
        new Thread(new RunnableExample ()).start ();
        }
}
```

Thread Scheduling

- Thread scheduling is the mechanism used to determine how runnable threads are allocated CPU time
- A thread-scheduling mechanism is either preemptive or nonpreemptive
- Preemptive scheduling – the thread scheduler preempts (pauses) a running thread to allow different threads to execute
- Nonpreemptive scheduling – the scheduler never interrupts a running thread
- The nonpreemptive scheduler relies on the running thread to yield control of the CPU so that other threads can execute

Thread Issues

Starvation

- ❖ A nonpreemptive scheduler may cause starvation (runnable threads, ready to be executed, wait to be executed in the CPU a very long time, maybe even forever)
- ❖ Sometimes, starvation is also called a livelock

Race Condition

- A race condition – the outcome of a program is affected by the order in which the program's threads are allocated CPU time
- Two threads are simultaneously modifying a single object
- Both threads “race” to store their value

Time sliced scheduling

- Time-sliced scheduling
  - the scheduler allocates a period of time that each thread can use the CPU
  - when that amount of time has elapsed, the scheduler preempts the thread and switches to a different thread
- Non time-sliced scheduling
  - the scheduler does not use elapsed time to determine when to preempt a thread
  - it uses other criteria such as priority or I/O status

Thread Priority

- ➢Every thread has a priority
- ➢When a thread is created, it inherits the priority of the thread that created it
- ➢The priority values range from 1 to 10, in increasing priority

- ❖ **public static int MIN_PRIORITY**
- ❖**public static int NORM_PRIORITY**
- ❖**public static int MAX_PRIORITY**

- The priority can be adjusted subsequently using the setPriority() method
- The priority of a thread may be obtained using getPriority()
- Priority constants are defined:
- MIN_PRIORITY=1
- MAX_PRIORITY=10
- NORM_PRIORITY=5

The **main** thread is created with priority NORM_PRIORITY

Critical Section

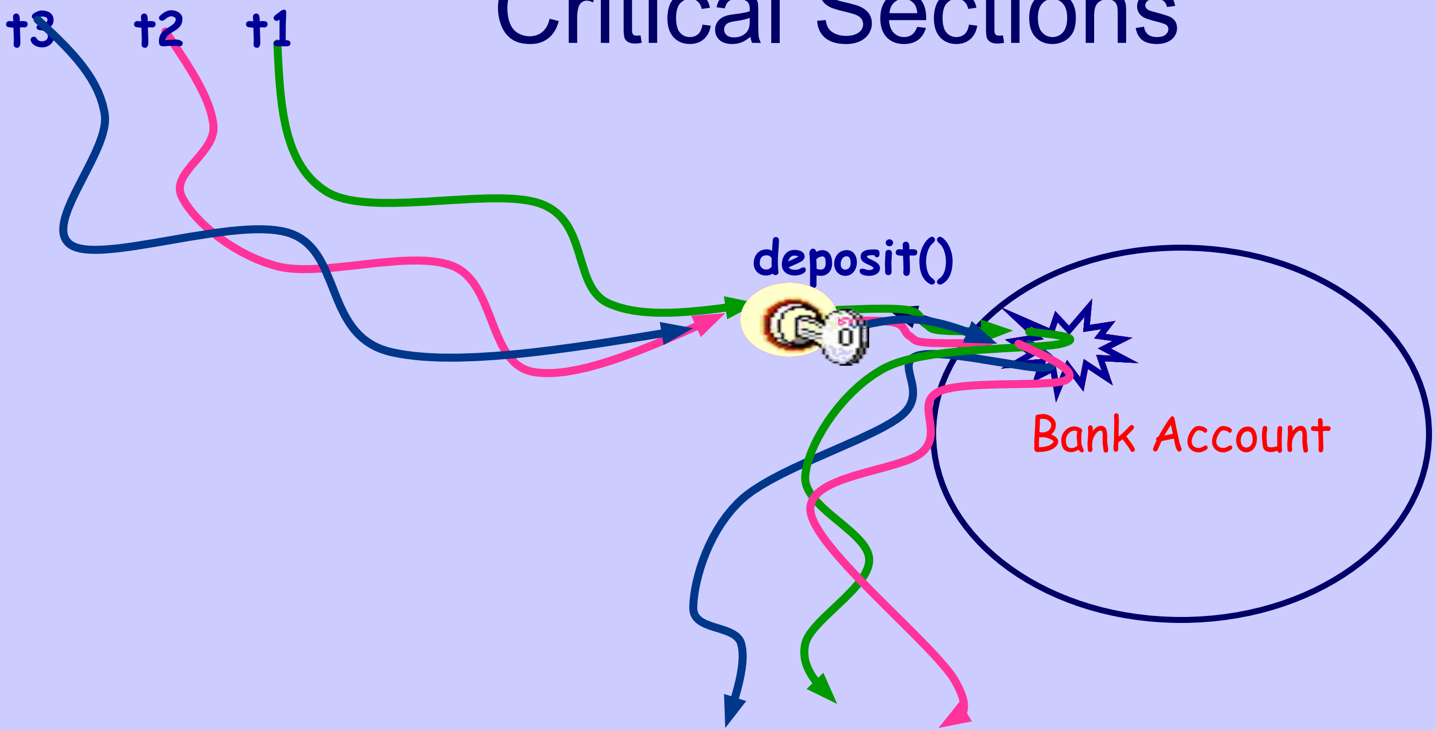
- The synchronized methods define critical sections
- Execution of critical sections is mutually exclusive. Why?

Example

```
public class BankAccount {
    private float balance;
    public synchronized void deposit(float
amount){
        balance += amount;
    }

    public synchronized void withdraw(float
amount){
        balance -= amount;
    }
}
```

Critical Sections



The Followings are Equivalent

```
public synchronized void a() {
 //… some code …
 }
```

```
public void a() {

     synchronized (this) {

         //… some code …
     }
 }
```

The Followings are Equivalent

```
public static synchronized void a() {

    //… some code …
 }
```

```
public void a() {

    synchronized (this.getClass()) {

        //… some code …
    }
 }
```

Example

```java
public class MyPrinter {
     public MyPrinter() {}
     public synchronized void printName(String name) {
       for (int i=1; i<100 ; i++) {
         try {
           Thread.sleep((long)(Math.random() * 100));
         } catch (InterruptedException ie) {}
             System.out.print(name);
       }
     }
}
```

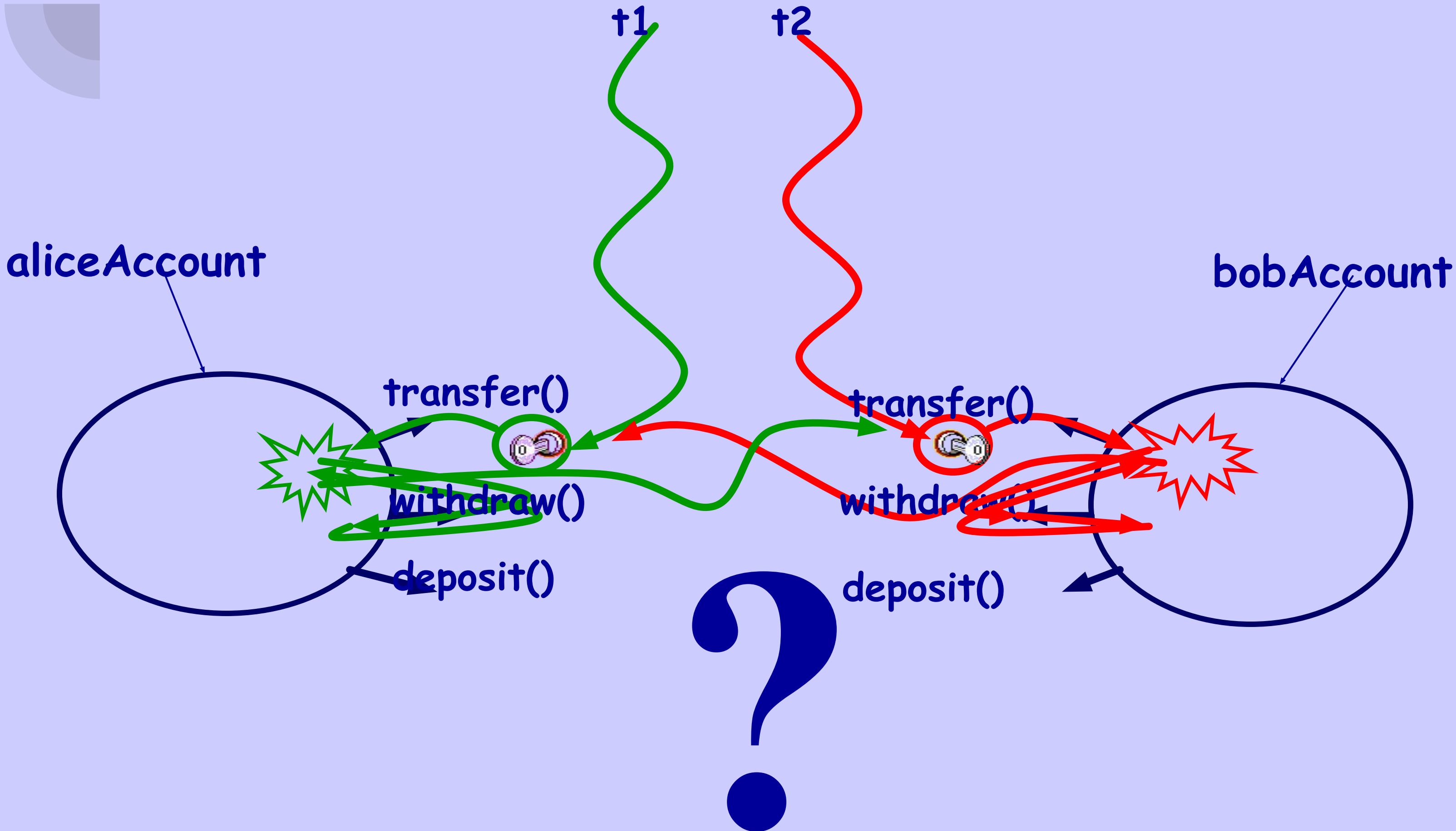
Deadlock Example

```
public class BankAccount {
    private float balance;
    public synchronized void deposit(float amount) {
        balance += amount;
    }
    public synchronized void withdraw(float amount) {
        balance -= amount;
    }
    public synchronized void transfer
                (float amount, BankAccount target) {
        withdraw(amount);
        target.deposit(amount);
    }
}
```

```java
public class MoneyTransfer implements Runnable {
    private BankAccount from, to;
    private float amount;
    public MoneyTransfer(
      BankAccount from, BankAccount to, float amount){
        this.from = from;
        this.to = to;
        this.amount = amount;
    }
    public void run() {
        source.transfer(amount, target);
    }
}
```

```java
BankAccount aliceAccount = new BankAccount();
BankAccount bobAccount = new BankAccount();
...
// At one place
Runnable transaction1 =
    new MoneyTransfer(aliceAccount, bobAccount, 1200);
Thread t1 = new Thread(transaction1);
t1.start();
// At another place
Runnable transaction2 =
    new MoneyTransfer(bobAccount, aliceAccount, 700);
Thread t2 = new Thread(transaction2);
t2.start();
```

Deadlocks



t1
t2
aliceAccount
bobAccount
transfer()
withdraw()
deposit()
transfer()
withdraw()
deposit()
?

Thread Synchronization

- We need to synchronized between transactions, for example, the consumer-producer scenario

Wait and Notify

- Allows two threads to cooperate
- Based on a single shared lock object
    - Marge put a cookie wait and notify Homer
    - Homer eat a cookie wait and notify Marge
        - Marge put a cookie wait and notify Homer
        - Homer eat a cookie wait and notify Marge

The **wait()** Method

- The **wait()** method is part of the **java.lang.Object** interface
- It requires a lock on the object’s monitor to execute
- It must be called from a synchronized method, or from a synchronized segment of code.

Why?

The wait() Method

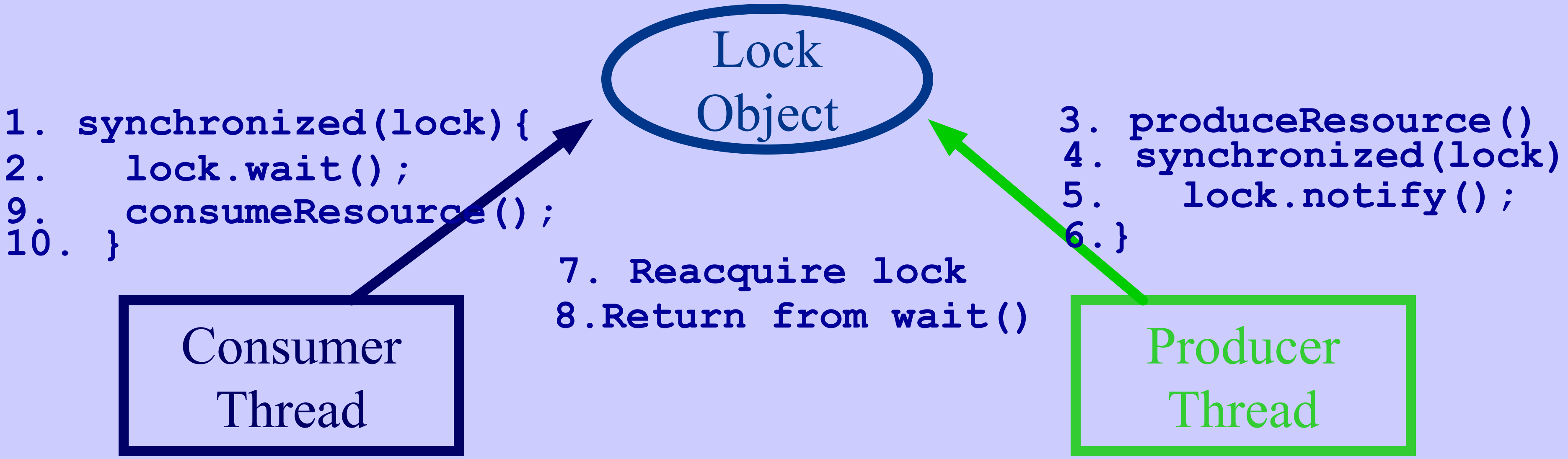
- wait() causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object
- Upon call for wait(), the thread releases ownership of this monitor and waits until another thread notifies the waiting threads of the object

The **wait()** Method

What is the difference between **wait** and **sleep**?

- **wait()** is also similar to **yield()**
  - Both take the current thread off the execution stack and force it to be rescheduled
- However, **wait()** is not automatically put back into the scheduler queue
  - **notify()** must be called in order to get a thread back into the scheduler's queue
  - The objects monitor must be reacquired before the thread's run can continue

Wait/Notify Sequence



Wait/Notify Sequence

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

Lock Object

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

```
7. Reacquire lock
8. Return from wait()
```

Consumer Thread

Producer Thread

Wait/Notify Sequence

Lock Object

```
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()
```

Consumer Thread

Producer Thread

Wait/Notify Sequence

Lock Object

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

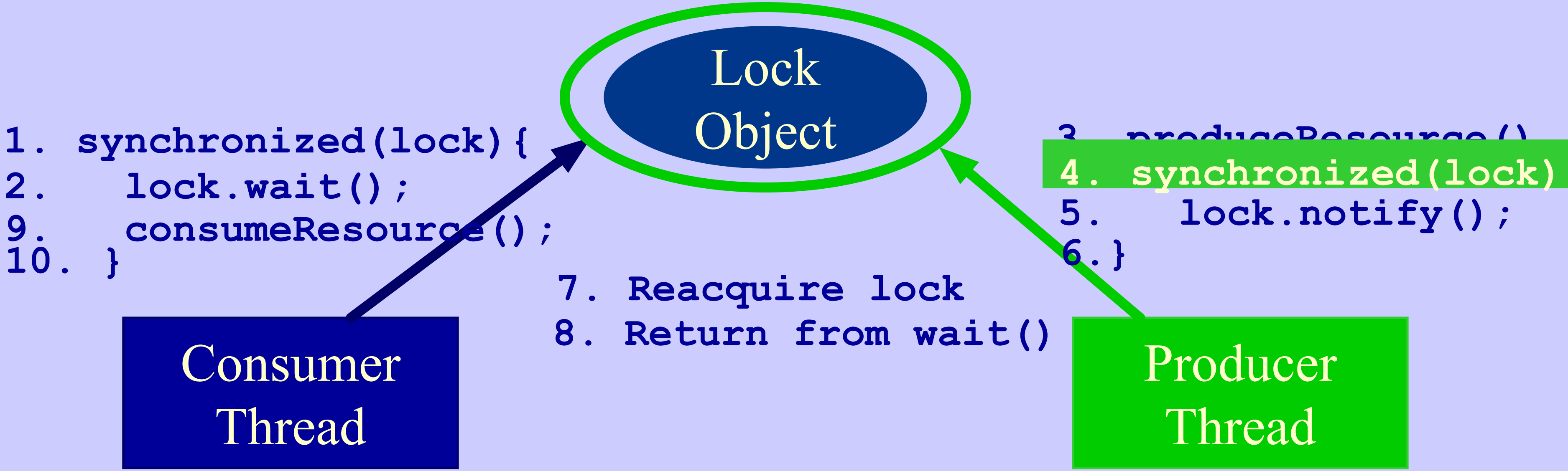
```
3. produceResource()
4. synchronized(lock)
5.    lock.notify();
6.}
```

```
7. Reacquire lock
8. Return from wait()
```

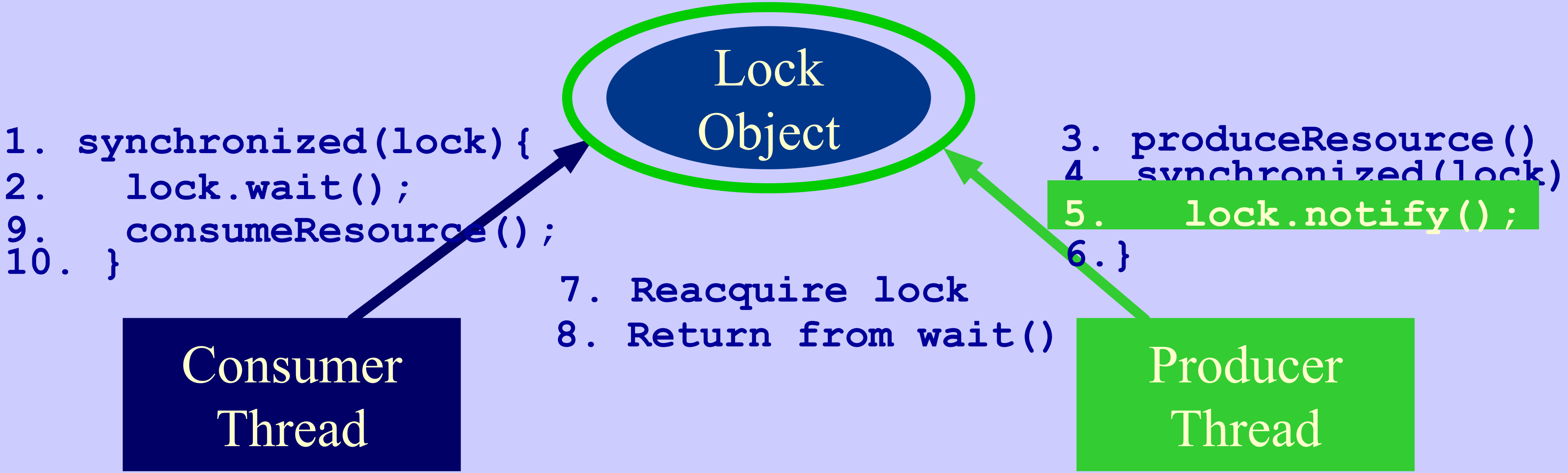
Consumer Thread

Producer Thread

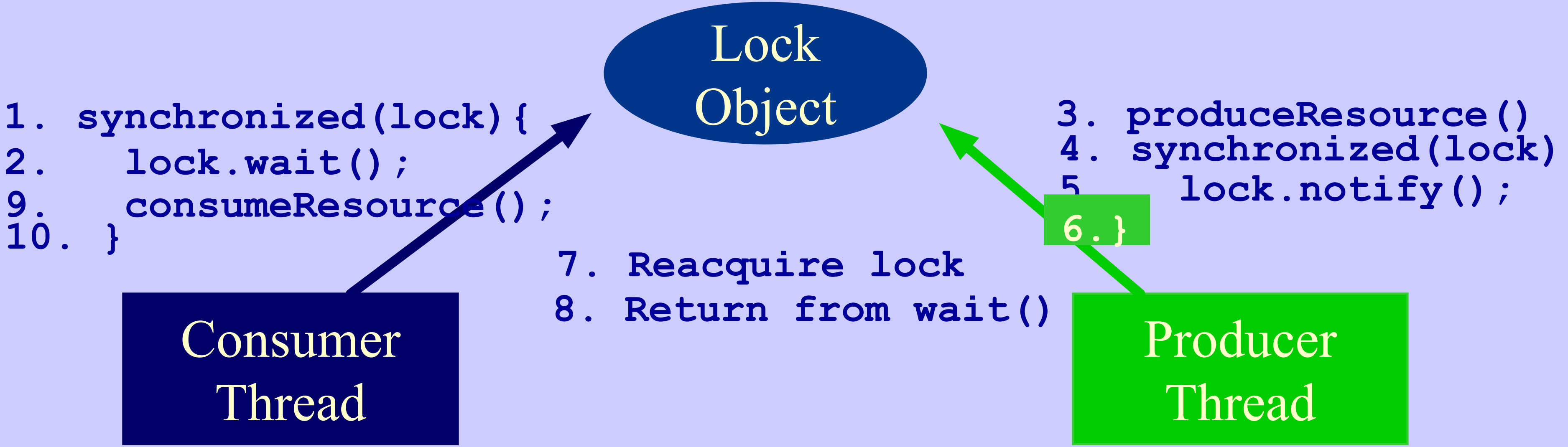
Wait/Notify Sequence



Wait/Notify Sequence



Wait/Notify Sequence



Wait/Notify Sequence

Lock Object

```
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()
```

Consumer Thread

Producer Thread

Wait/Notify Sequence

Lock Object

```
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()

Consumer Thread

Producer Thread

Wait/Notify Sequence

Lock Object

```
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()
```

Consumer Thread

Producer Thread

Wait/Notify Sequence

Lock Object

```
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()
```

Consumer Thread

Producer Thread

Timers and TimerTask

- The classes Timer and TimerTask are part of the java.util package
- Useful for
  - performing a task after a specified delay
  - performing a sequence of tasks at constant time intervals

Scheduling Timers

- The schedule method of a timer can get as parameters:
  - Task, time
  - Task, time, period
  - Task, delay
  - Task, delay, period

What to do	When to start	At which rate

Timer Example

```
import java.util.*;
public class CoffeeTask extends TimerTask {
    public void run() {
        System.out.println(“Time for a Coffee Break”);
    }
    public static void main(String args[]) {
        Timer timer = new Timer();
        long hour = 1000 * 60 * 60;
        timer.schedule(new CoffeeTask(), 0, 8 * hour);
      timer.scheduleAtFixedRate(new CoffeeTask(),new Date(), 24 * hour);
    }
}
```

Stopping Timers

- A Timer thread can be stopped in the following ways:
  - Apply cancel() on the timer
  - Make the thread a daemon
  - Remove all references to the timer after all the TimerTask tasks have finished
  - Call System.exit()