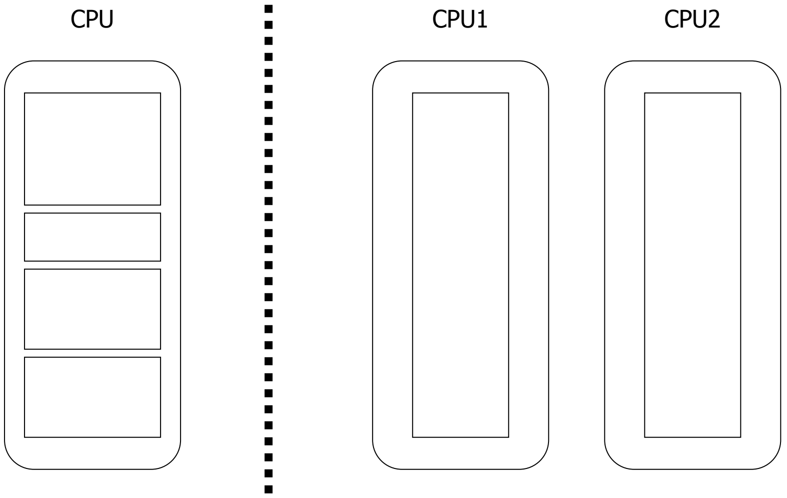
Java Threads

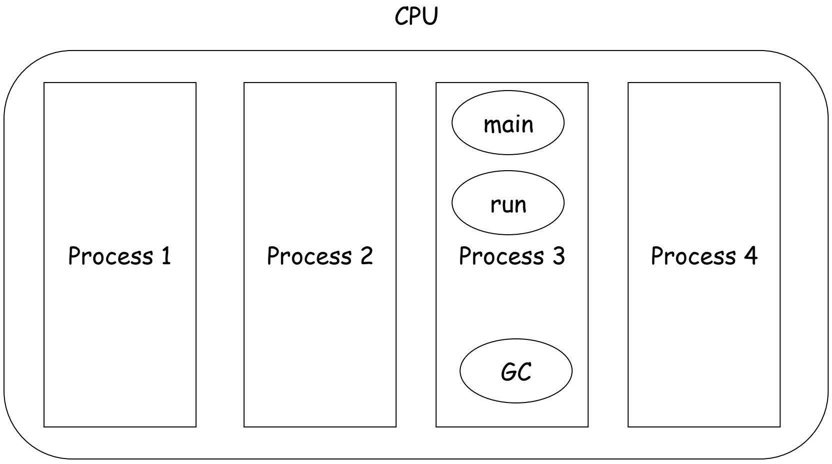
Multitasking vs Multithreading

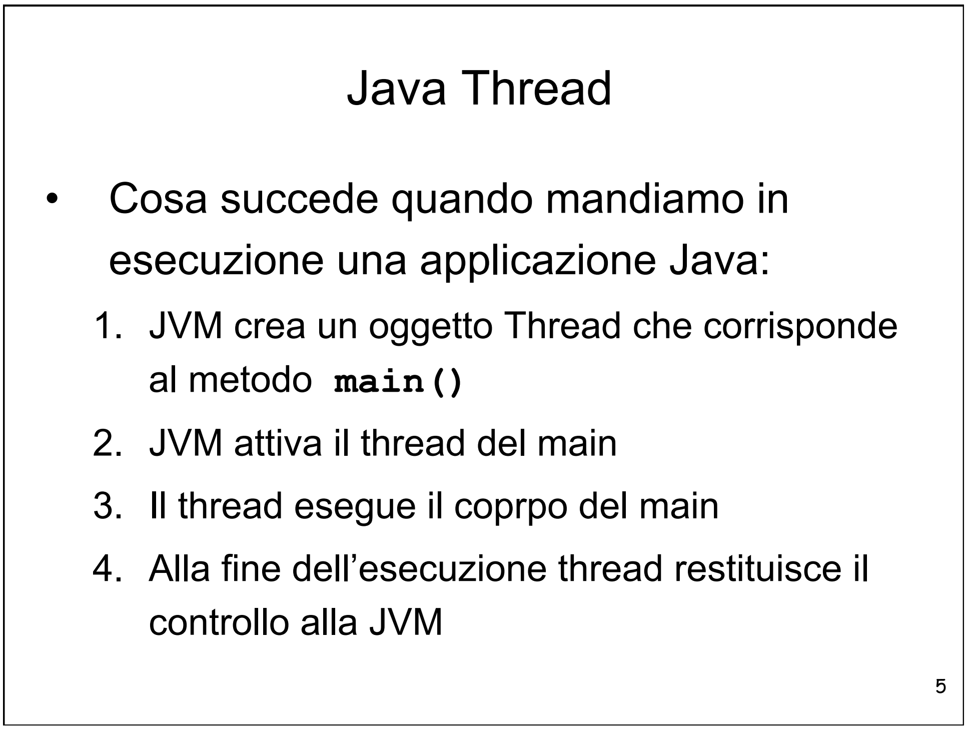
- Multitasking:
  - Avere la possibilita’ di eseguire contemporamente diverse attivita’ (job)
- Multithreading:
  - Un thread e’ un singolo flusso di esecuzione
  - Insieme multiplo di thread all’interno di un programma (esempio Web Browser)

Concorrenza vs. Parallelismo



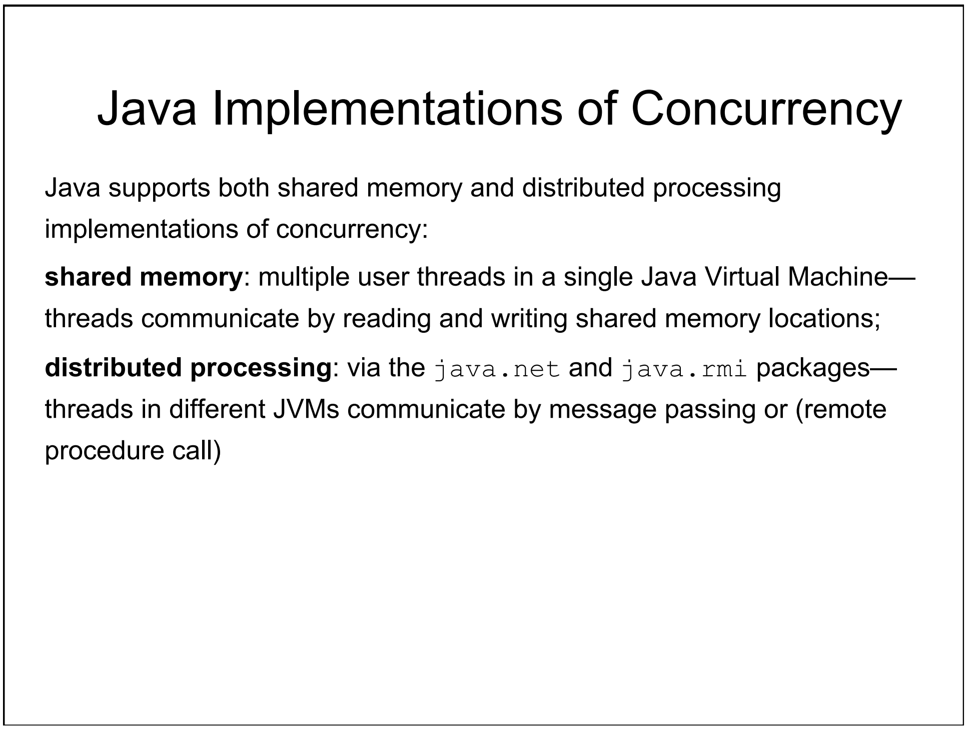
Thread e Processi





Java Thread

- Cosa succede quando mandiamo in esecuzione una applicazione Java:
  1. JVM crea un oggetto Thread che corrisponde al metodo **`main()`**
  2. JVM attiva il thread del main
  3. Il thread esegue il coprpo del main
  4. Alla fine dell’esecuzione thread restituisce il controllo alla JVM



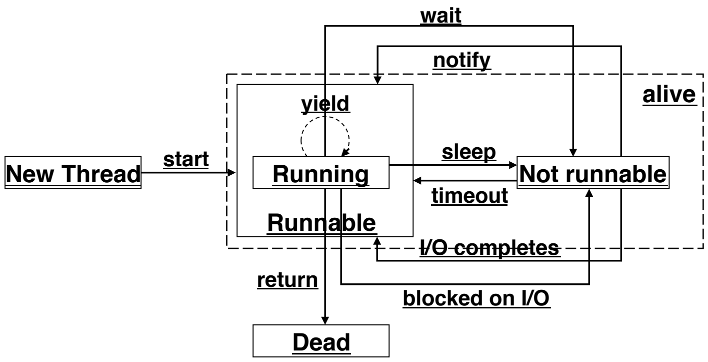
Java Implementations of Concurrency

Java supports both shared memory and distributed processing implementations of concurrency:

shared memory: multiple user threads in a single Java Virtual Machine—threads communicate by reading and writing shared memory locations;

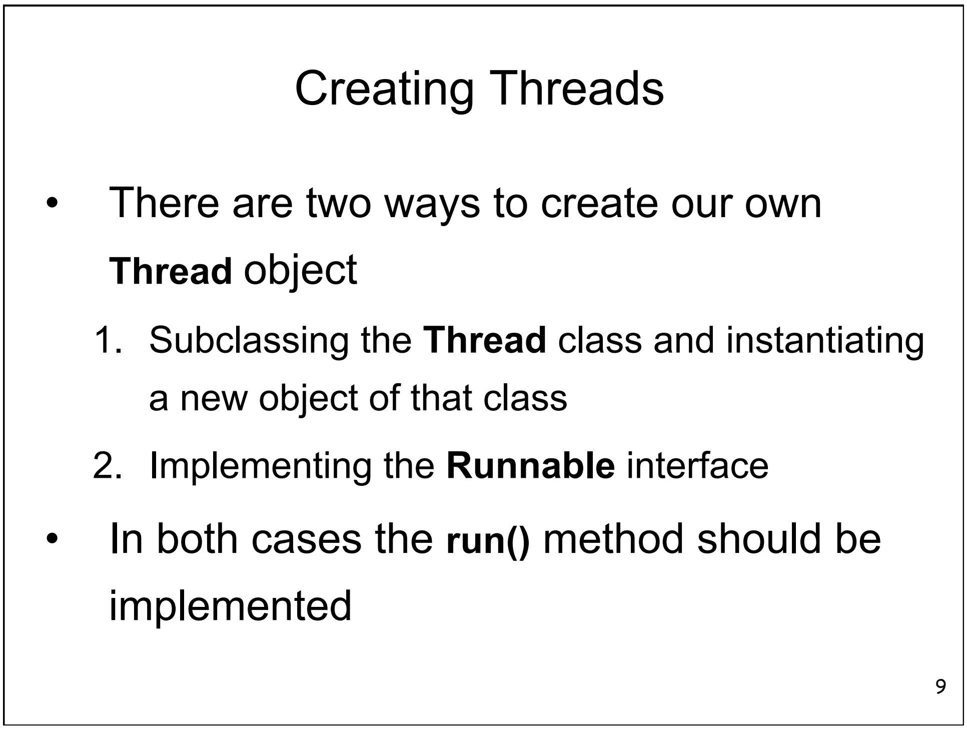
distributed processing: via the `java.net` and `java.rmi` packages—threads in different JVMs communicate by message passing or (remote procedure call)

`Thread` lifecycle



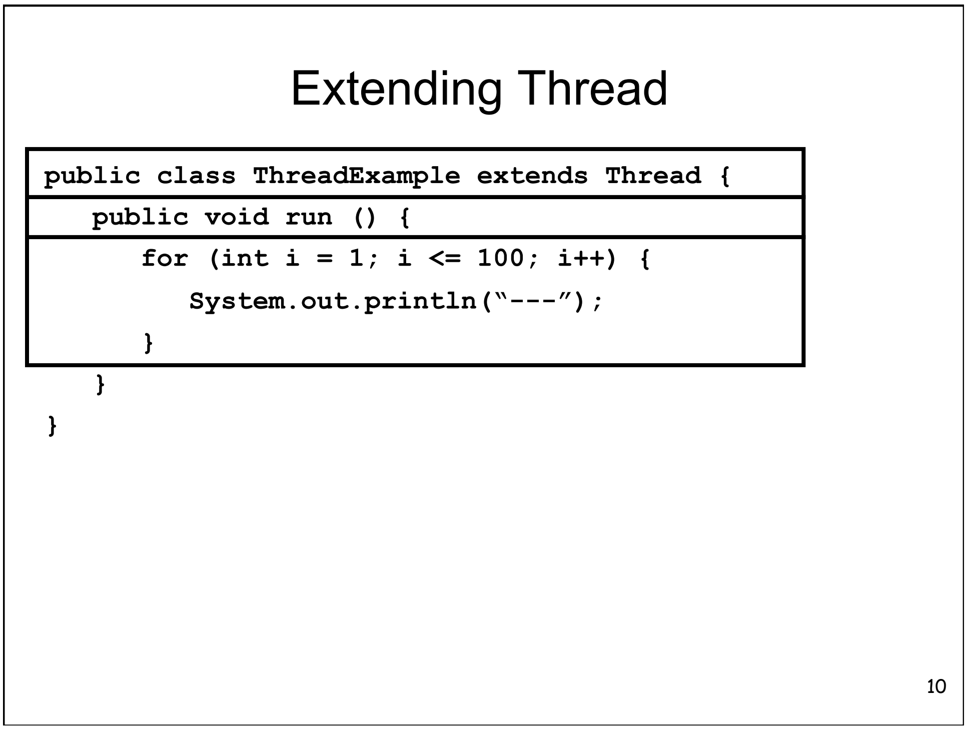
Thread Multipli

- Ogni thread ha il suo run-time stack privato
- Se due thread invocano l'esecuzione dello stesso metodo, ciascun thread avra' il controllo delle variabili locali usate dal metodo
- Tutti I thread condividono lo heap
- Thread possono agire concorrentemente sugli stessi oggetti



Creating Threads

- There are two ways to create our own **Thread** object
  1. Subclassing the **Thread** class and instantiating a new object of that class
  2. Implementing the **Runnable** interface
- In both cases the **run()** method should be implemented



Extending Thread

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

Thread Methods

void start()

- Creates a new thread and makes it runnable
- This method can be called only once

void run()

- The new thread begins its life inside this method

void stop() (deprecated)

- The thread is being terminated

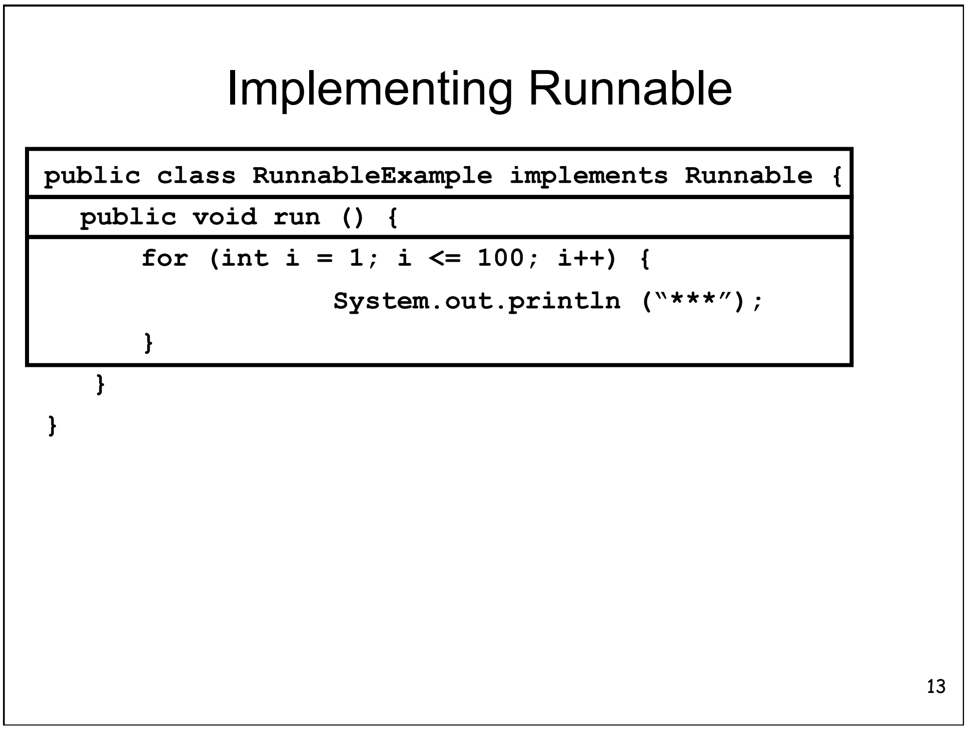
Thread Methods

void yield()

- Causes the currently executing thread object to temporarily pause and allow other threads to execute
- Allow only threads of the same priority to run

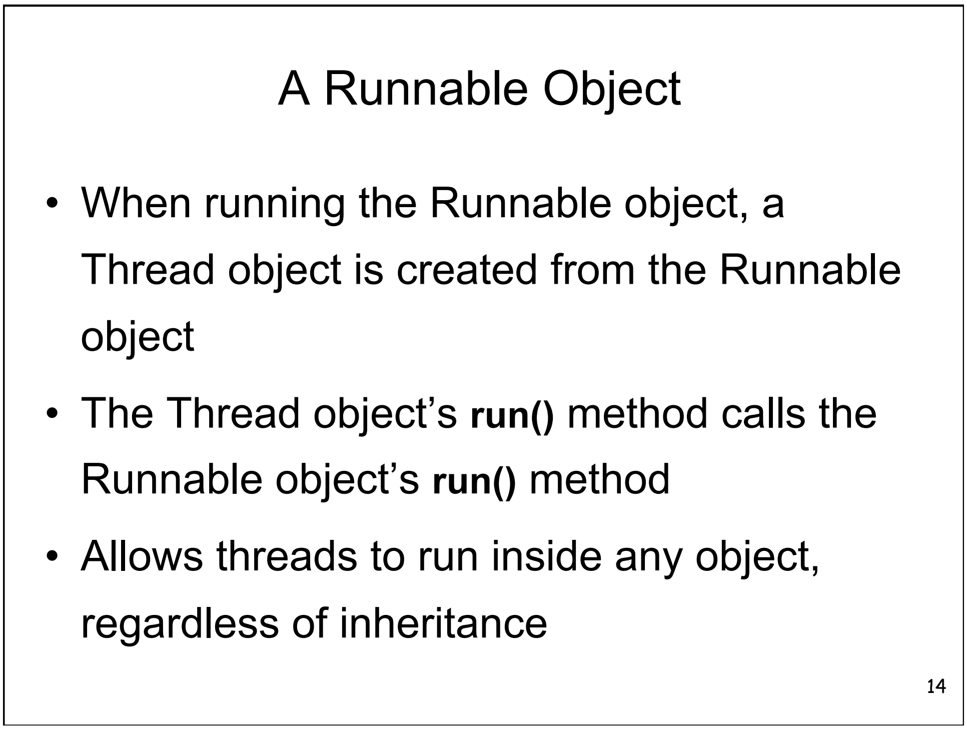
void sleep(int *m*) or sleep(int *m*, int *n*)

- The thread sleeps for *m* milliseconds, plus *n* nanoseconds



Implementing Runnable

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



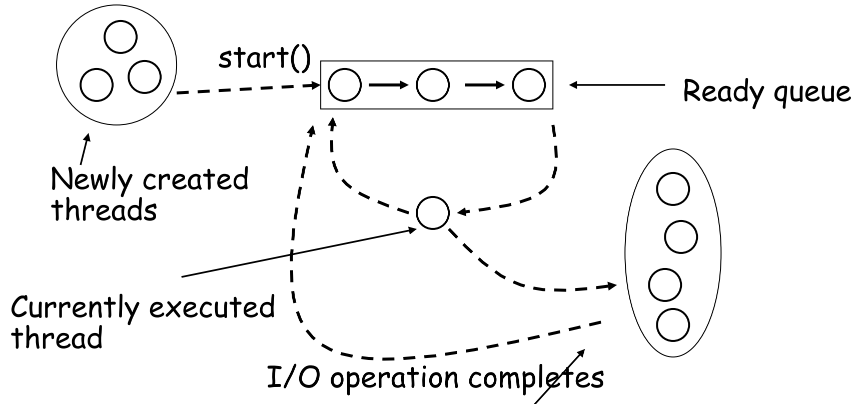
A Runnable Object

- When running the Runnable object, a Thread object is created from the Runnable object
- The Thread object’s **run()** method calls the Runnable object’s **run()** method
- Allows threads to run inside any object, regardless of inheritance

Starting the Threads

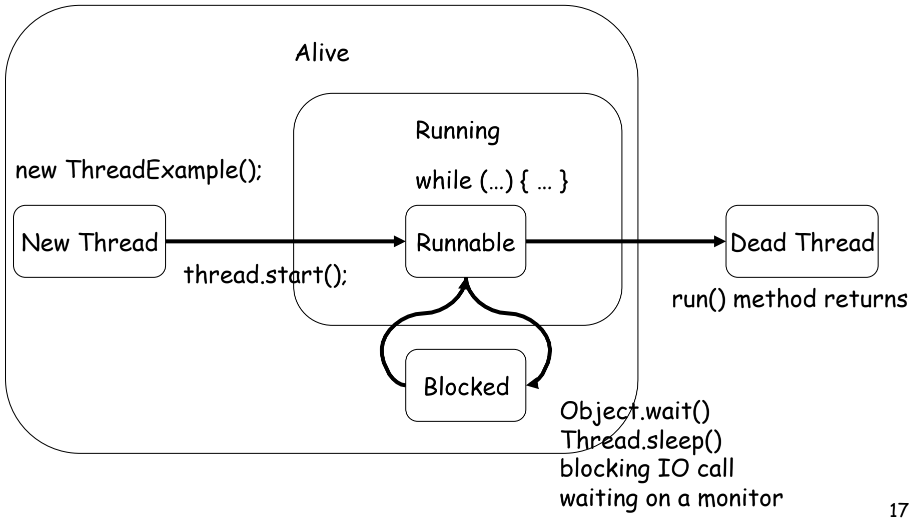
```
public class ThreadsStartExample {
       public static void main (String argv[]) {
         new ThreadExample ().start ();
         new Thread(new RunnableExample ()).start ();
       }
}
```

Scheduling Threads



- ·Waiting for I/O operation to be completed
- ·Waiting to be notified
- ·Sleeping
- ·Waiting to enter a synchronized object

Thread State Diagram



Alive

Running

while (…) { … }

new ThreadExample();

New Thread

thread.start();

Runnable

Dead Thread

run() method returns

Blocked

Object.wait()
Thread.sleep()
blocking IO call
waiting on a monitor

Example

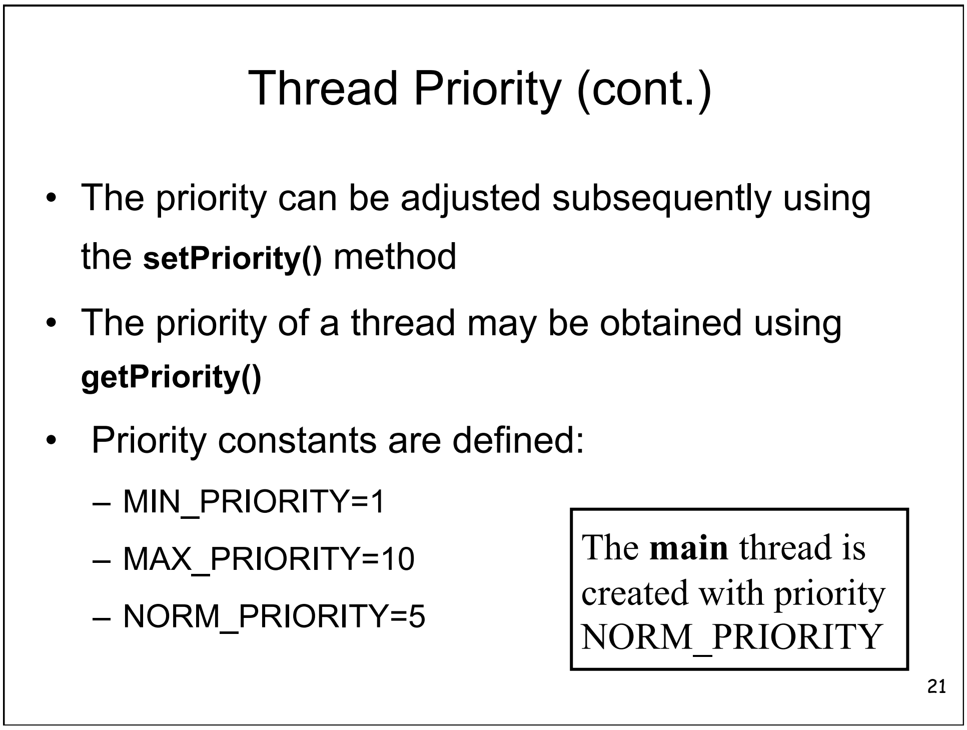
```
public class PrintThread1 extends Thread {
    String name;
    public PrintThread1(String name) {
        this.name = name;
    }
    public void run() {
        for (int i=1; i<100 ; i++) {
            try {
                sleep((long)(Math.random() * 100));
            } catch (InterruptedException ie) { }
            System.out.print(name);
        }
}
```

Example (cont)

```
  public static void main(String args[]) {
        PrintThread1 a = new PrintThread1("*");
        PrintThread1 b = new PrintThread1("-");

        a.start();
        b.start();
  }
}
```

Thread Priority

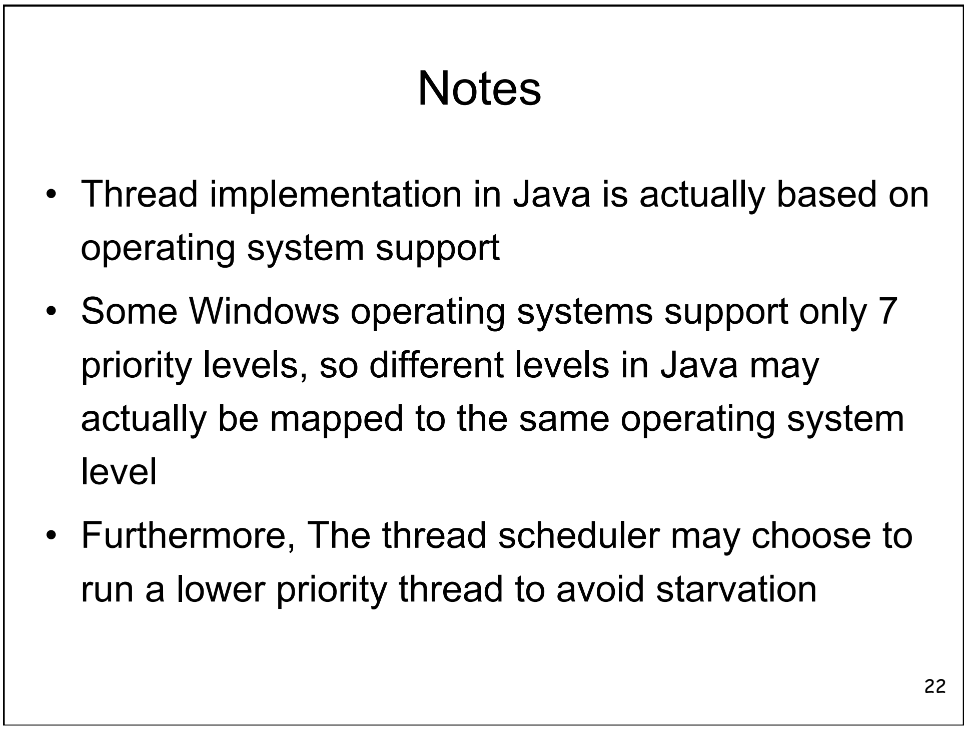
- Every thread has a priority
- When a thread is created, it inherits the priority of the thread that created it
- The priority values range from 1 to 10, in increasing priority



Thread Priority (cont.)

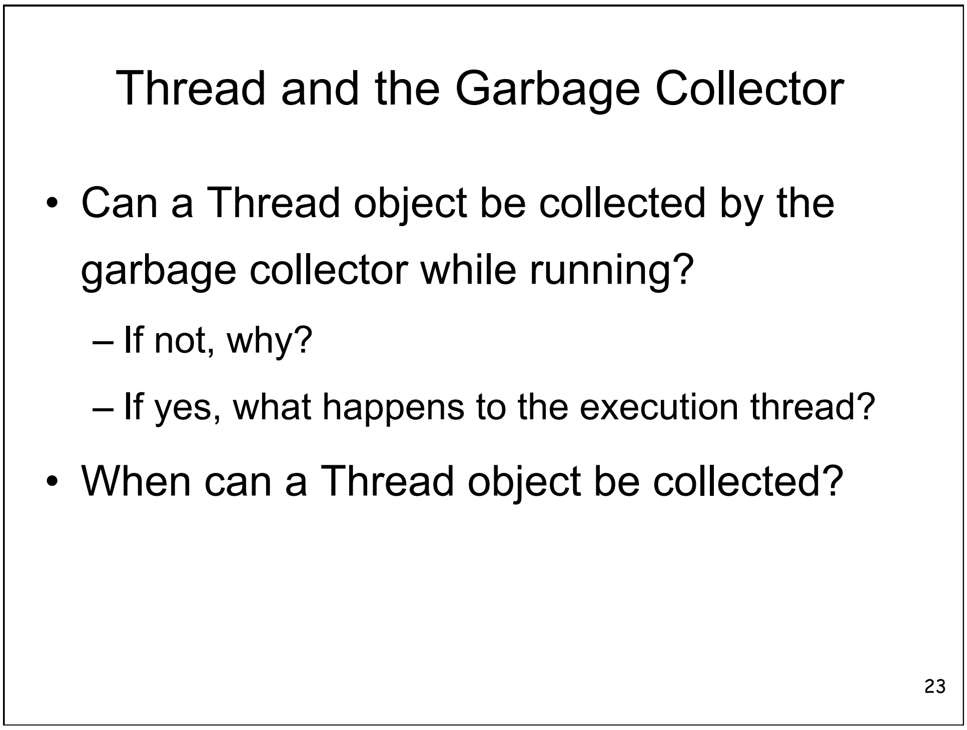
- The priority can be adjusted subsequently using the **setPriority()** method
- The priority of a thread may be obtained using **getPriority()**
- Priority constants are defined:
  - MIN_PRIORITY=1
  - MAX_PRIORITY=10
  - NORM_PRIORITY=5

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



Notes

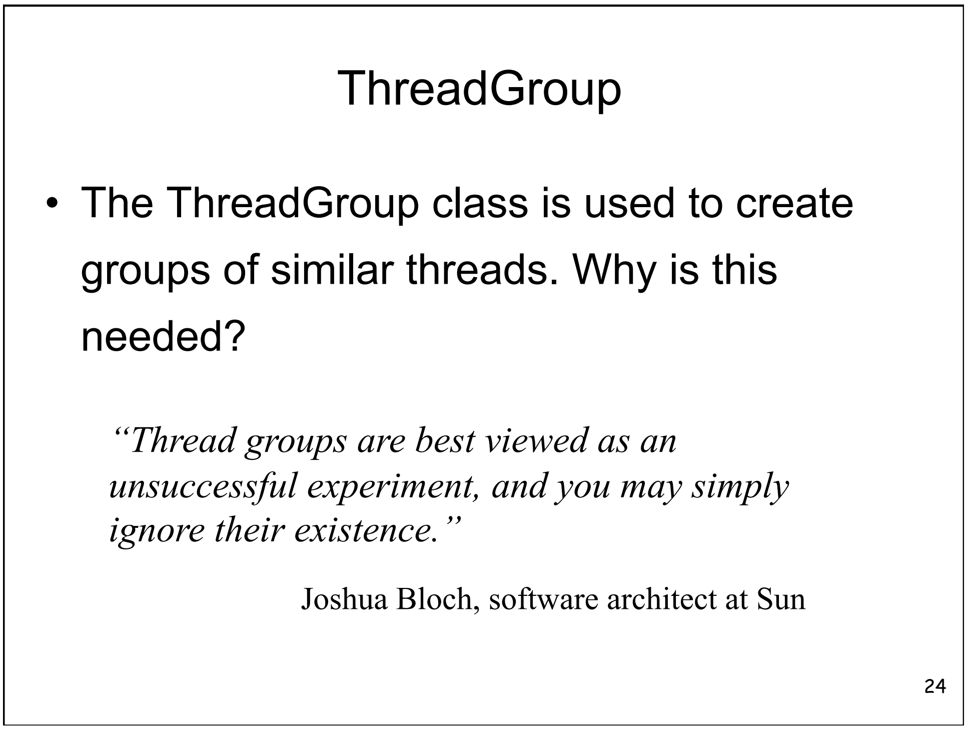
- Thread implementation in Java is actually based on operating system support
- Some Windows operating systems support only 7 priority levels, so different levels in Java may actually be mapped to the same operating system level
- Furthermore, The thread scheduler may choose to run a lower priority thread to avoid starvation



Thread and the Garbage Collector

- Can a Thread object be collected by the garbage collector while running?
  - If not, why?
  - If yes, what happens to the execution thread?
- When can a Thread object be collected?

23



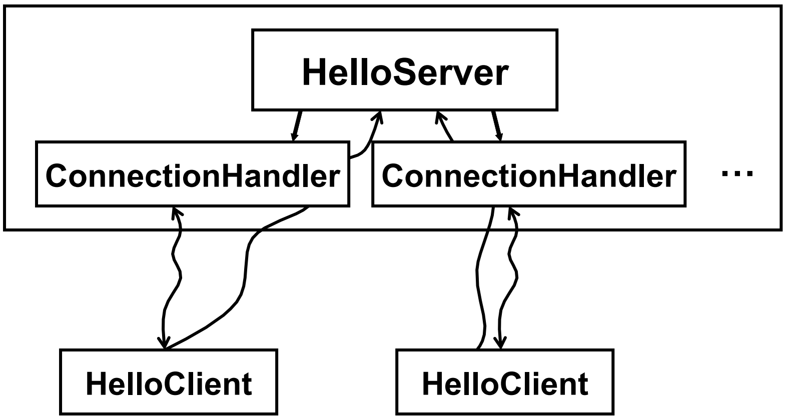
ThreadGroup

- The ThreadGroup class is used to create groups of similar threads. Why is this needed?

> *"Thread groups are best viewed as an unsuccessful experiment, and you may simply ignore their existence."*
>
> Joshua Bloch, software architect at Sun

24

Multithreading Client-Server



HelloServer

ConnectionHandler

ConnectionHandler

…

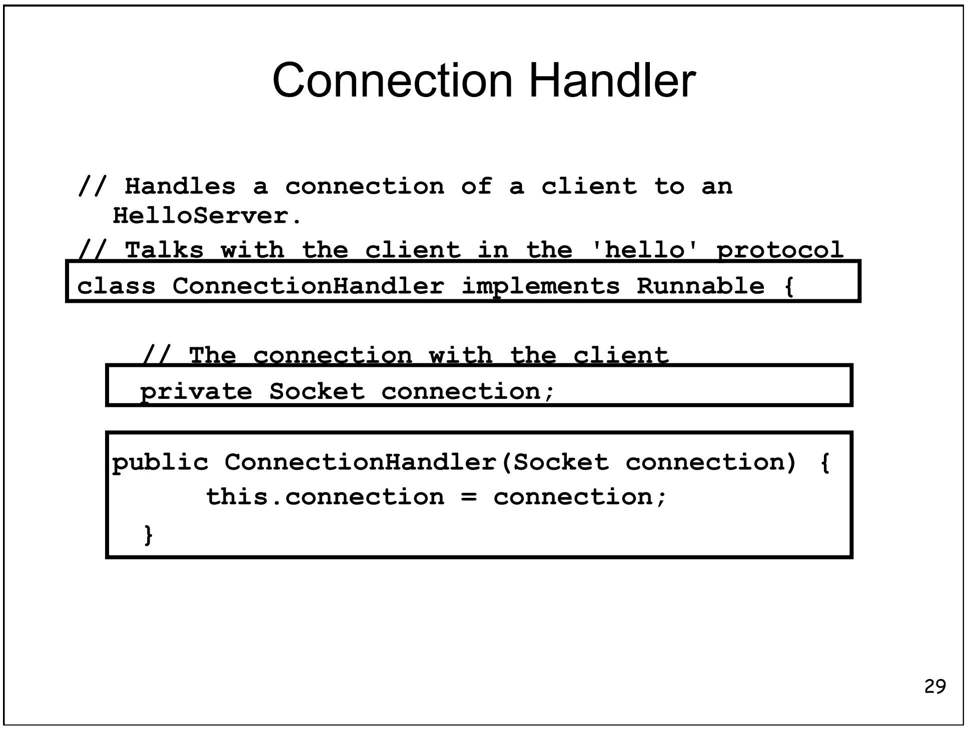
HelloClient

HelloClient

Server

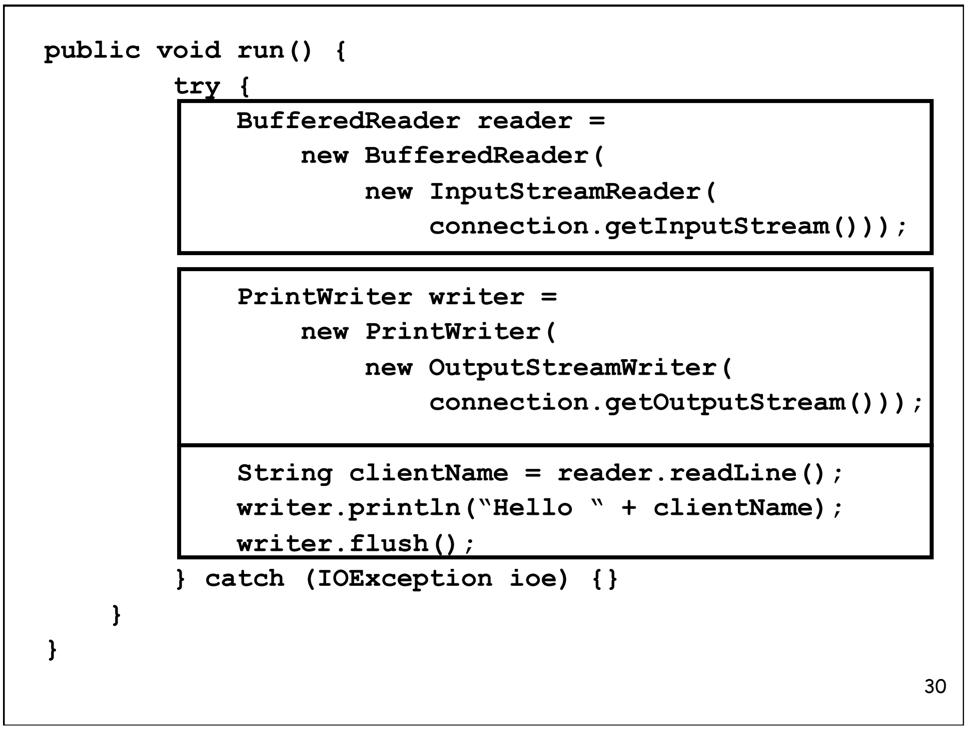
```
import java.net.*;import java.io.*;
class HelloServer {

   public static void main(String[] args) {
  int port = Integer.parseInt(args[0]);
     try {
     ServerSocket server =
           new ServerSocket(port);
       } catch (IOException ioe) {
           System.err.println(“Couldn't run “ +
               “server on port “ + port);
           return;
       }
```

```
   while(true) {
         try {
             Socket connection = server.accept();
             ConnectionHandler handler =
                 new ConnectionHandler(connection);
             new Thread(handler).start();
      } catch (IOException ioe1) {
      }
  }
```

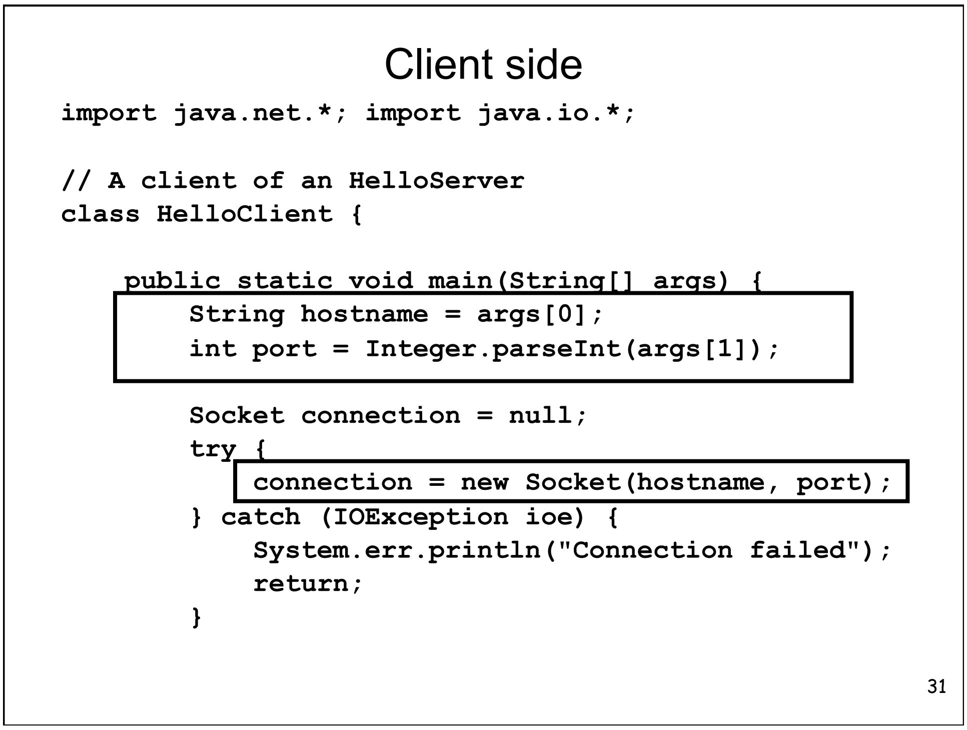


Connection Handler

```
// Handles a connection of a client to an
   HelloServer.
// Talks with the client in the 'hello' protocol
class ConnectionHandler implements Runnable {

    // The connection with the client
    private Socket connection;

   public ConnectionHandler(Socket connection) {
         this.connection = connection;
     }
```

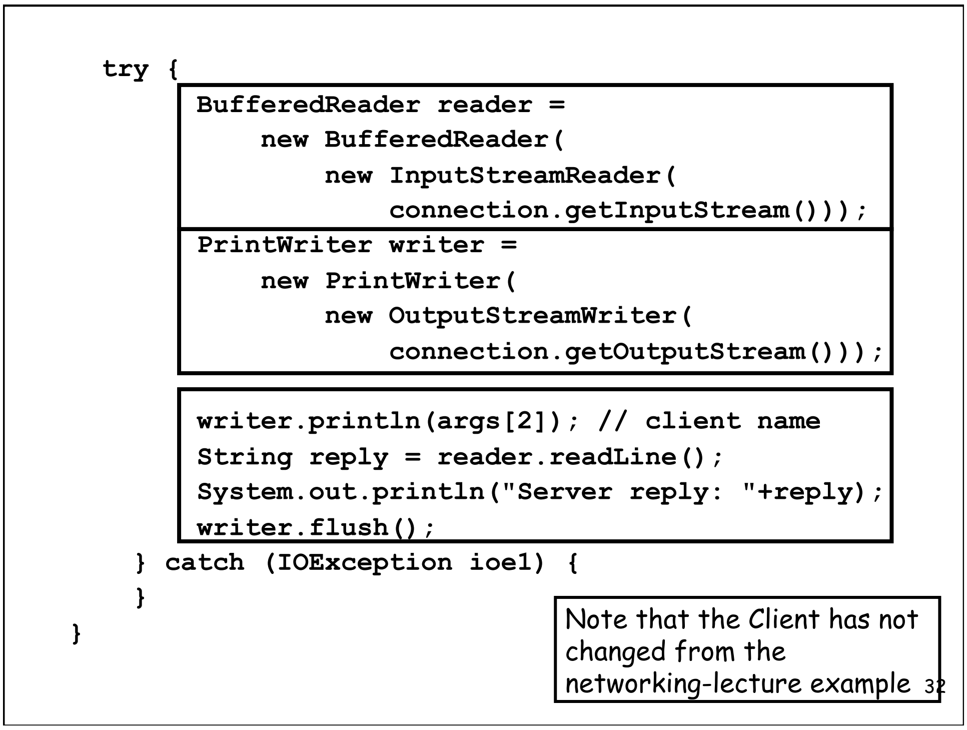


```
public void run() {
        try {
            BufferedReader reader =
                new BufferedReader(
                    new InputStreamReader(
                        connection.getInputStream()));

            PrintWriter writer =
                new PrintWriter(
                    new OutputStreamWriter(
                        connection.getOutputStream()));

            String clientName = reader.readLine();
            writer.println(“Hello “ + clientName);
            writer.flush();
        } catch (IOException ioe) {}
    }
}
```



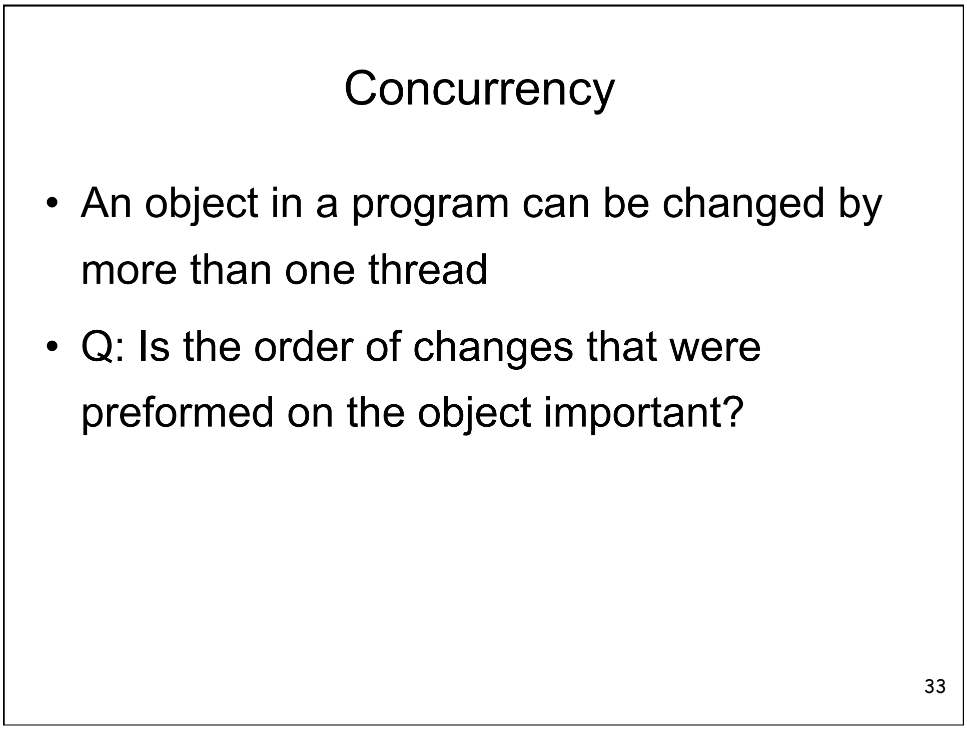
Client side

```
import java.net.*; import java.io.*;

// A client of an HelloServer
class HelloClient {

    public static void main(String[] args) {
        String hostname = args[0];
        int port = Integer.parseInt(args[1]);

        Socket connection = null;
        try {
            connection = new Socket(hostname, port);
        } catch (IOException ioe) {
            System.err.println("Connection failed");
            return;
        }
```



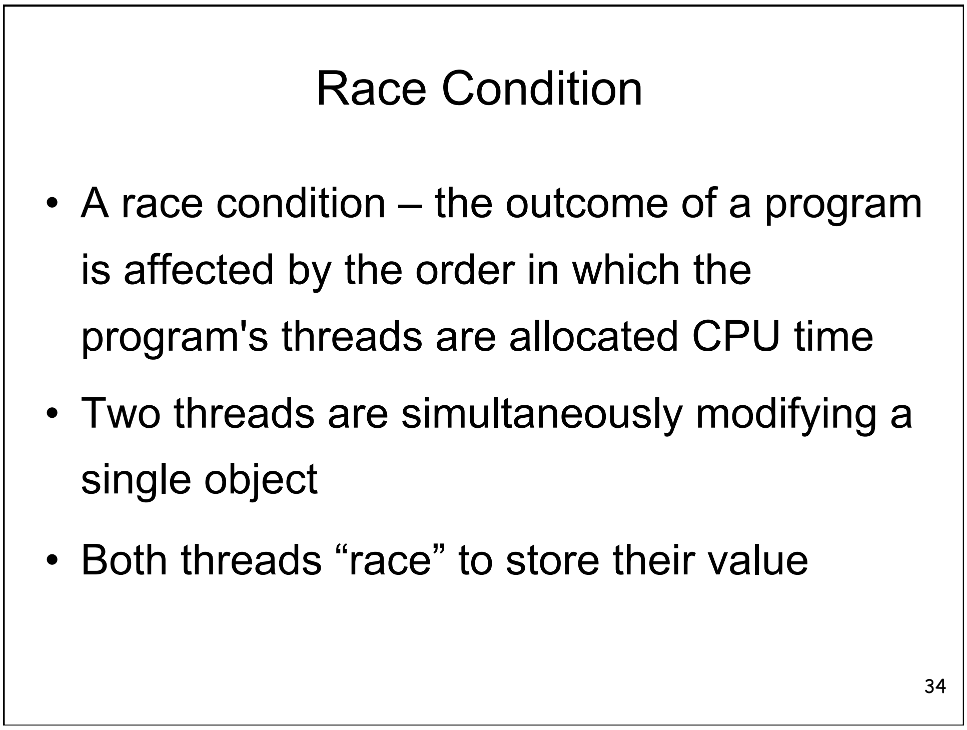
```
 try {
    BufferedReader reader =
        new BufferedReader(
            new InputStreamReader(
                connection.getInputStream()));
    PrintWriter writer =
        new PrintWriter(
            new OutputStreamWriter(
                connection.getOutputStream()));

    writer.println(args[2]); // client name
    String reply = reader.readLine();
    System.out.println("Server reply: "+reply);
    writer.flush();
  } catch (IOException ioe1) {
  }
}
```

Note that the Client has not changed from the networking-lecture example



Concurrency

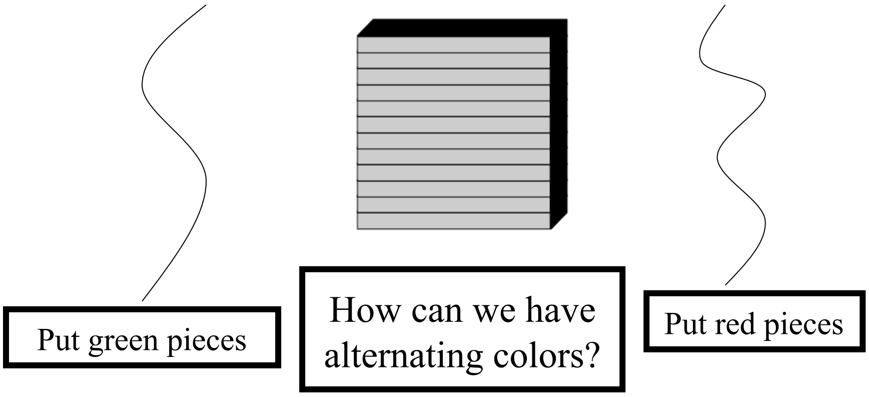
- An object in a program can be changed by more than one thread
- Q: Is the order of changes that were preformed on the object important?



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

Race Condition Example



Put green pieces

How can we have alternating colors?

Put red pieces

Monitors

- Each object has a “monitor” that is a token used to determine which application thread has control of a particular object instance
- In execution of a synchronized method (or block), access to the object monitor must be gained before the execution
- Access to the object monitor is queued

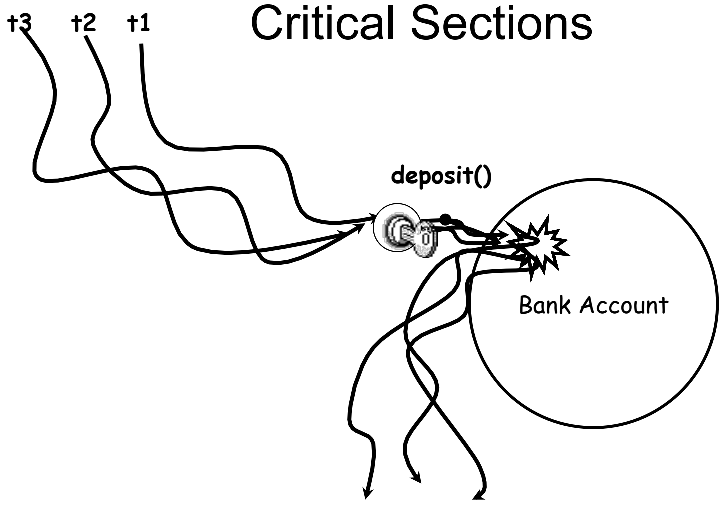
Monitor (cont.)

- Entering a monitor is also referred to as locking the monitor, or acquiring ownership of the monitor
- If a thread *A* tries to acquire ownership of a monitor and a different thread has already entered the monitor, the current thread (*A*) must wait until the other thread leaves the monitor

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



Static Synchronized Methods

- Marking a static method as synchronized, associates a monitor with the class itself
- The execution of synchronized static methods of the same class is mutually exclusive.

Synchronized Statements

- A monitor can be assigned to a block:

```
synchronized(object) { some-code }
```

- It can also be used to monitor access to a data element that is not an object, e.g., array:

```
void arrayShift(byte[] array, int count) {
   synchronized(array) {
     System.arraycopy (array, count, array,
                                    0, array.size -
  count);
   }
}
```

The Followings are Equivalent

```
public synchronized void a() {

     //… some code …
  }
```

```
public void a() {

     synchronized (this) {

          //… some code …
     }
  }
```

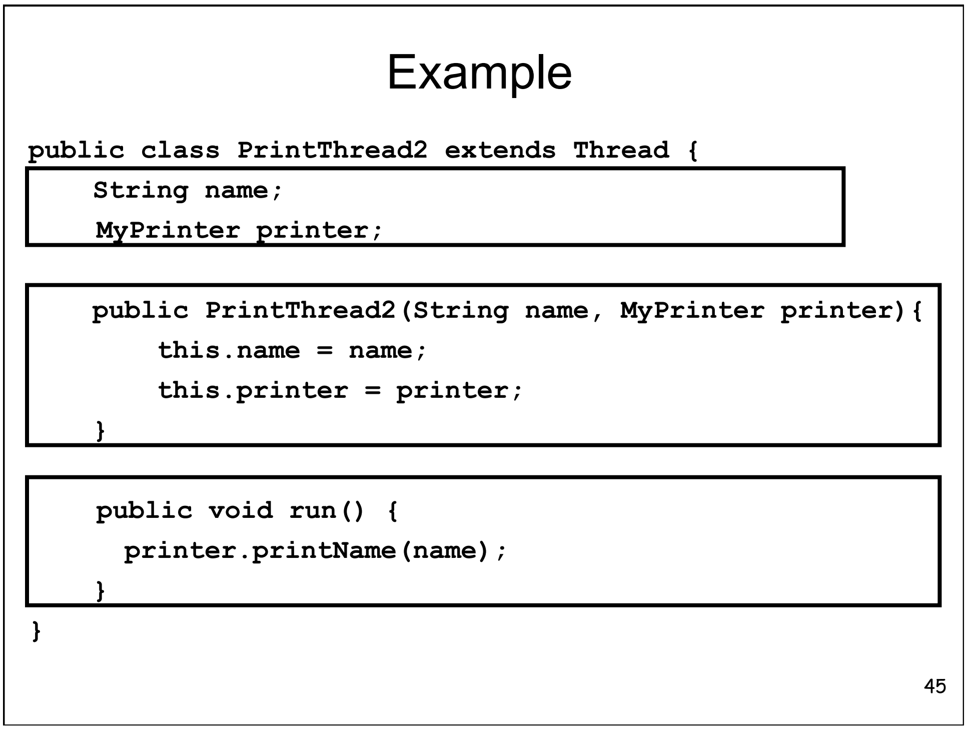
The Followings are Equivalent

```
public static synchronized void a() {

     //… some code …
  }
```

```
public void a() {

     synchronized (this.getClass()) {

          //… some code …
     }
  }
```

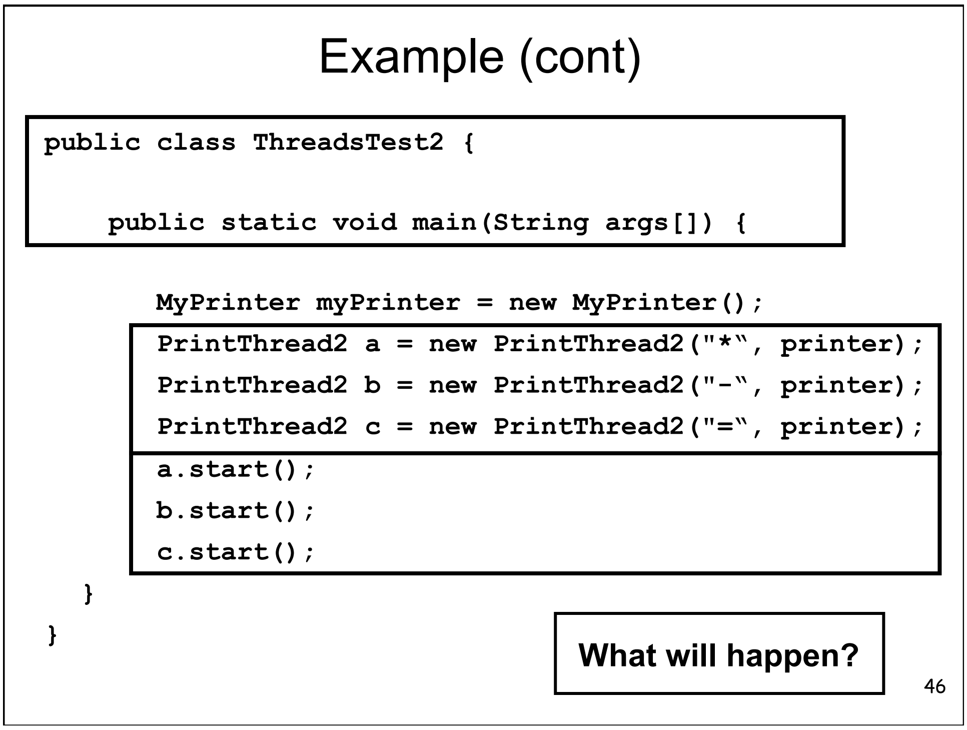
Example

```
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);
      }
    }
}
```



Example

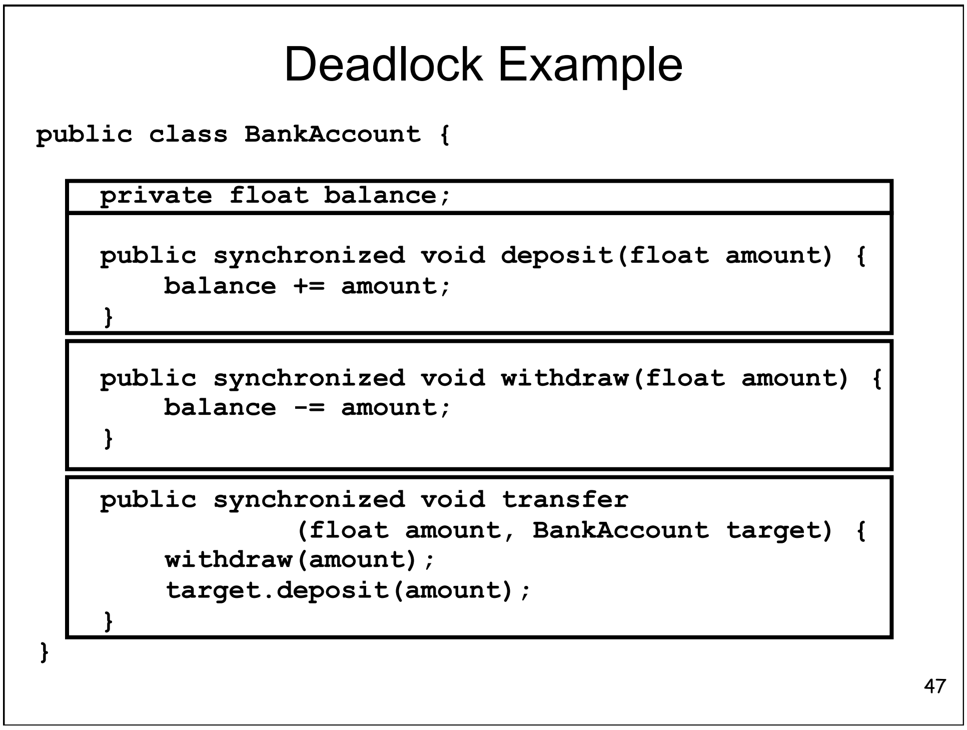
```
public class PrintThread2 extends Thread {
    String name;
    MyPrinter printer;

    public PrintThread2(String name, MyPrinter printer){
        this.name = name;
        this.printer = printer;
    }

    public void run() {
      printer.printName(name);
    }
}
```



Example (cont)

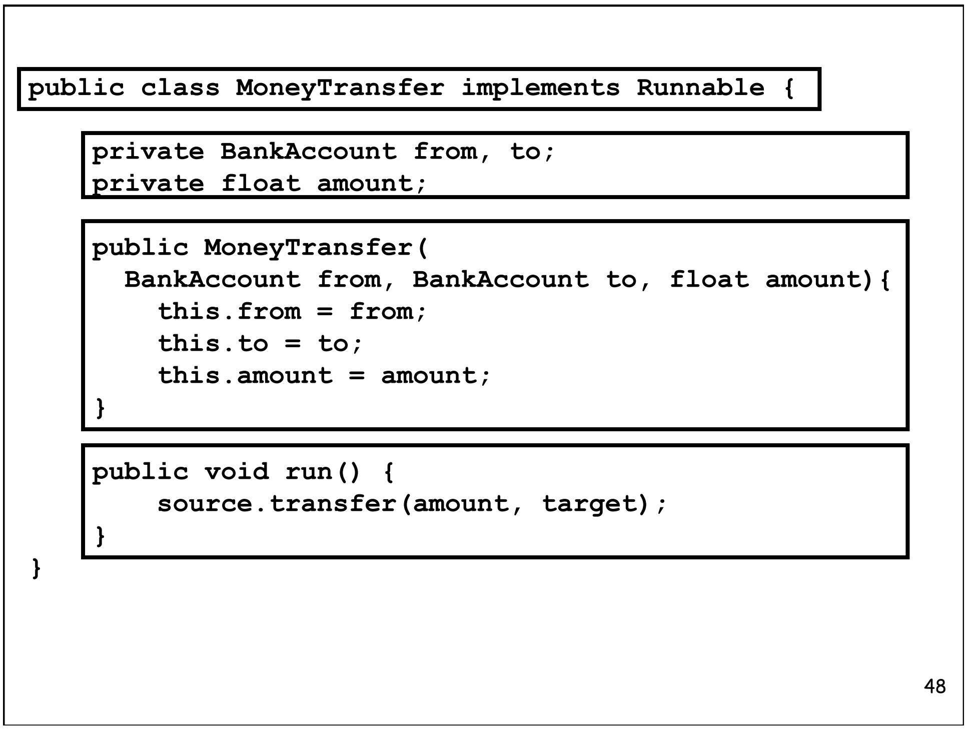
```
public class ThreadsTest2 {

    public static void main(String args[]) {

        MyPrinter myPrinter = new MyPrinter();
        PrintThread2 a = new PrintThread2("*", printer);
        PrintThread2 b = new PrintThread2("-", printer);
        PrintThread2 c = new PrintThread2("=", printer);
        a.start();
        b.start();
        c.start();
    }
 }
```

What will happen?



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);
    }
}
```



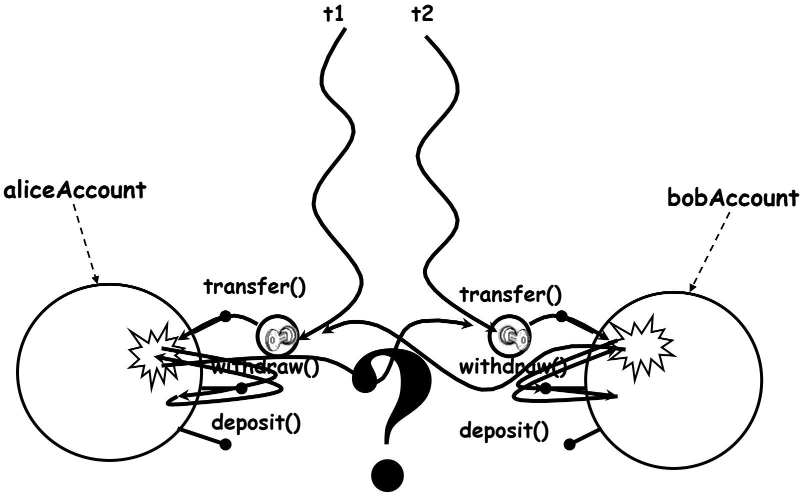
```
public class MoneyTransfer implements Runnable {

    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);
    }
}
```

```
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();
```

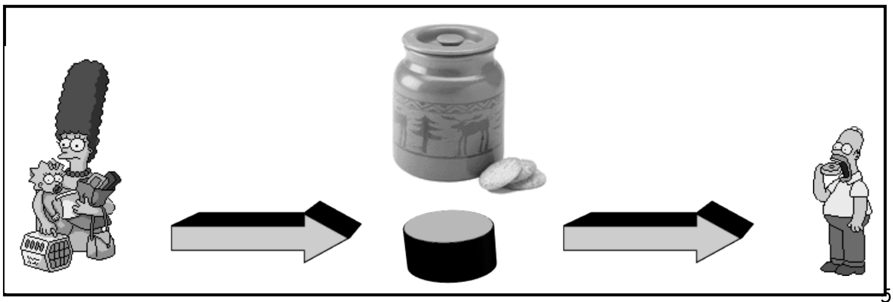
```
// At another place
Runnable transaction2 =
    new MoneyTransfer(bobAccount, aliceAccount, 700);
Thread t2 = new Thread(transaction2);
t2.start();
```

Deadlocks



Thread Synchronization

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



51

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

52

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

- **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

Consumer

- Consumer:

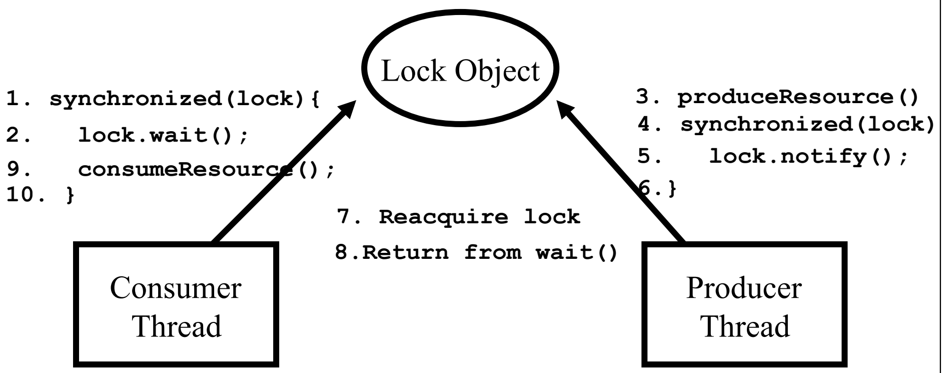
```
synchronized (lock) {
  while (!resourceAvailable()) {
    lock.wait();
  }
  consumeResource();
}
```

Producer

- Producer:

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

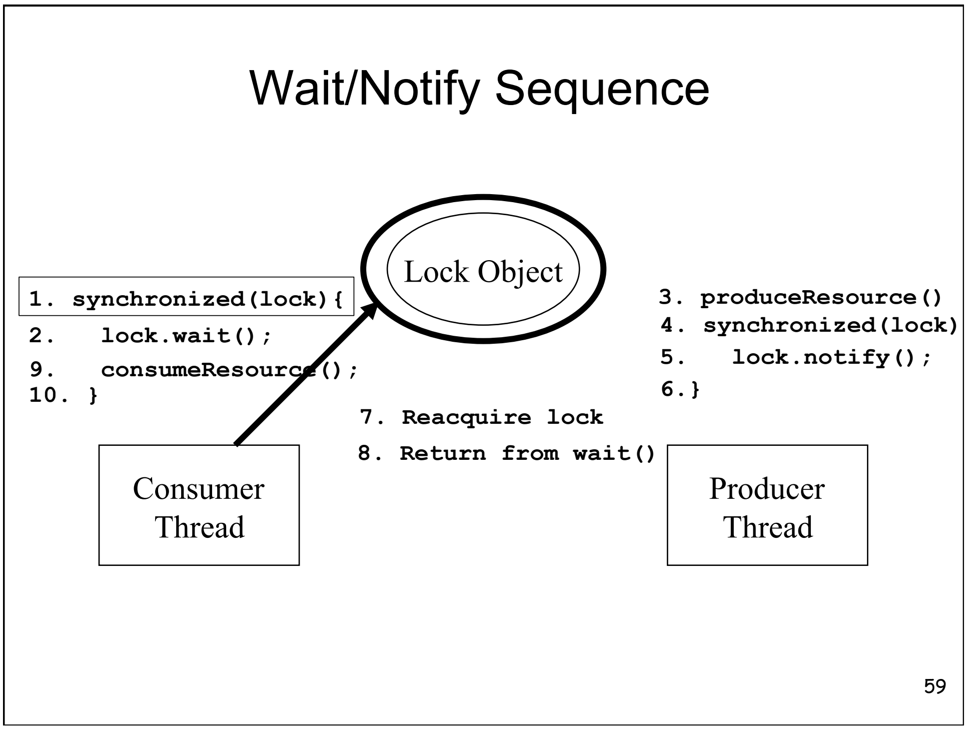
Wait/Notify Sequence



```
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. }
```

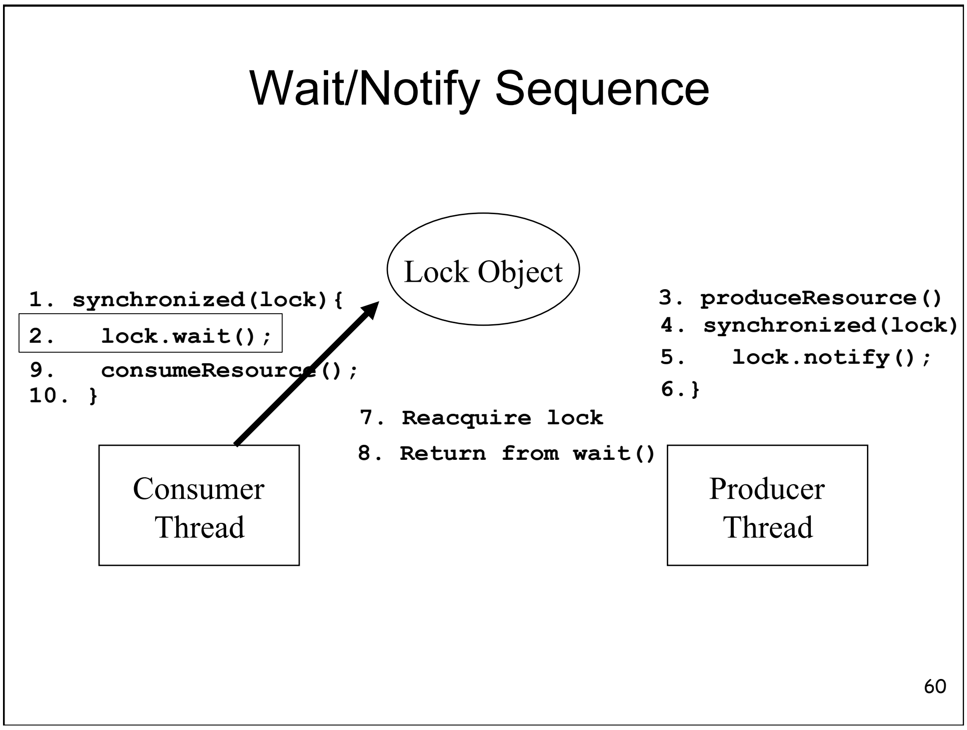
Lock Object

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

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

Consumer Thread

Producer Thread



Wait/Notify Sequence

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

Lock Object

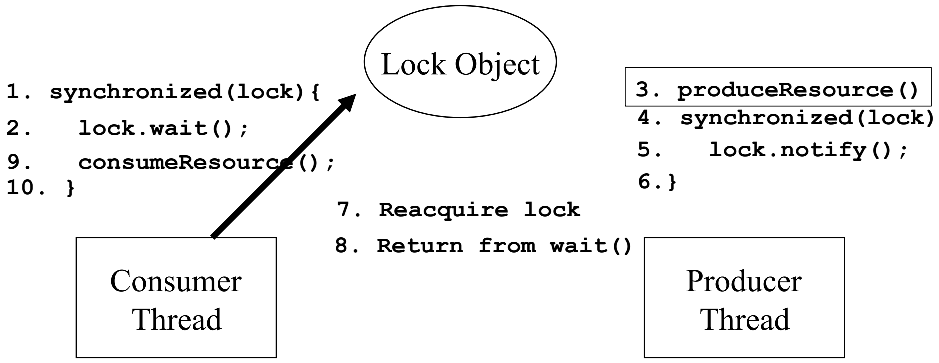
```
3. produceResource()
4. synchronized(lock)
5.   lock.notify();
6.}
```

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

Consumer Thread

Producer Thread

Wait/Notify Sequence



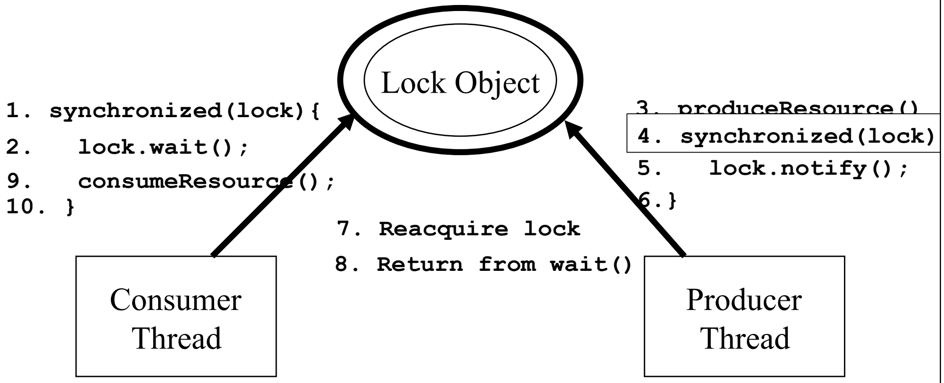
```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

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

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

61

Wait/Notify Sequence

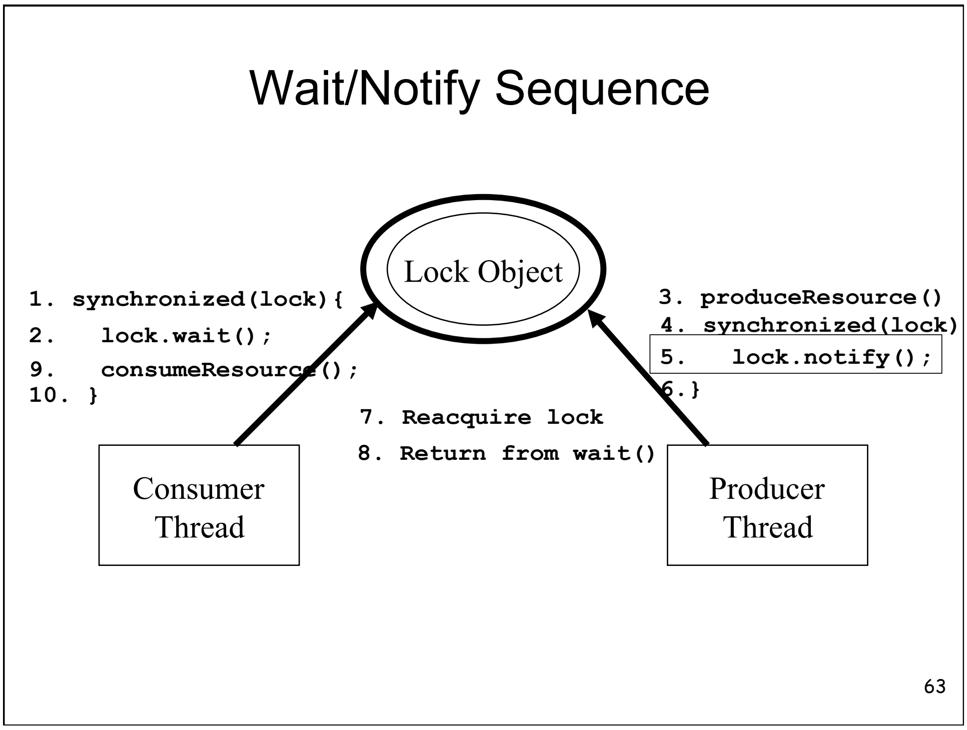


```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

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

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

62



Wait/Notify Sequence

Lock Object

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

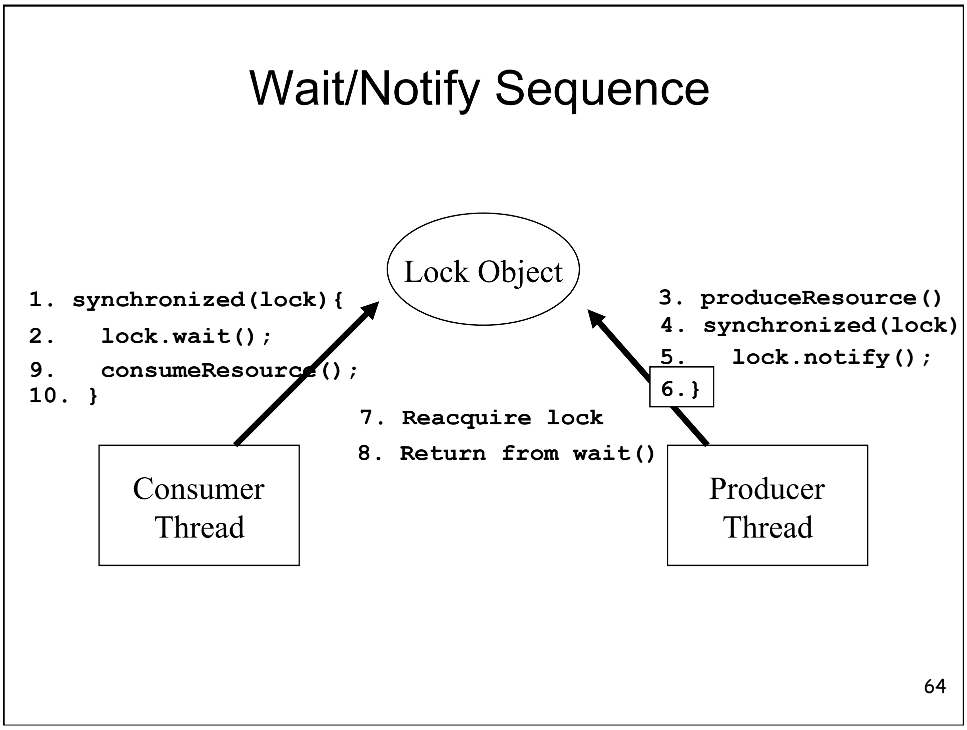
```
3. produceResource()
4. synchronized(lock)
5.   lock.notify();
6.}
```

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.}
```

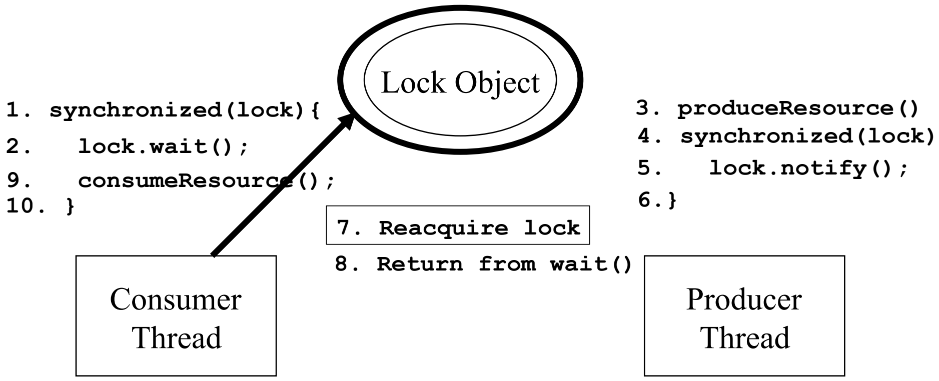
7. Reacquire lock

8. Return from wait()

Consumer Thread

Producer Thread

Wait/Notify Sequence



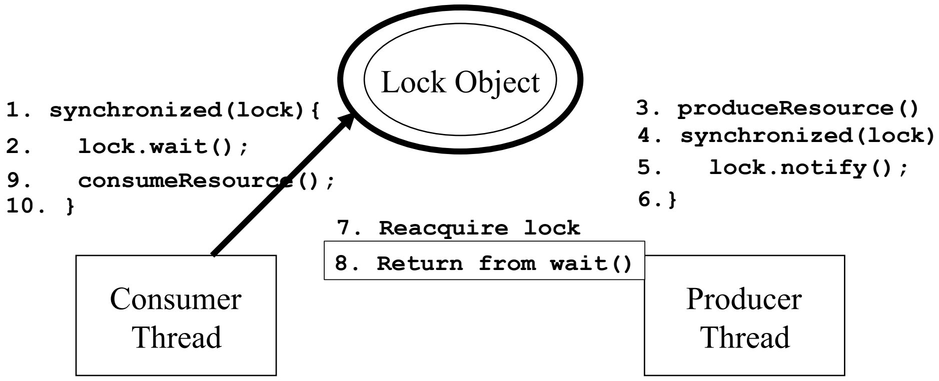
```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

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

7. Reacquire lock

8. Return from wait()

Wait/Notify Sequence

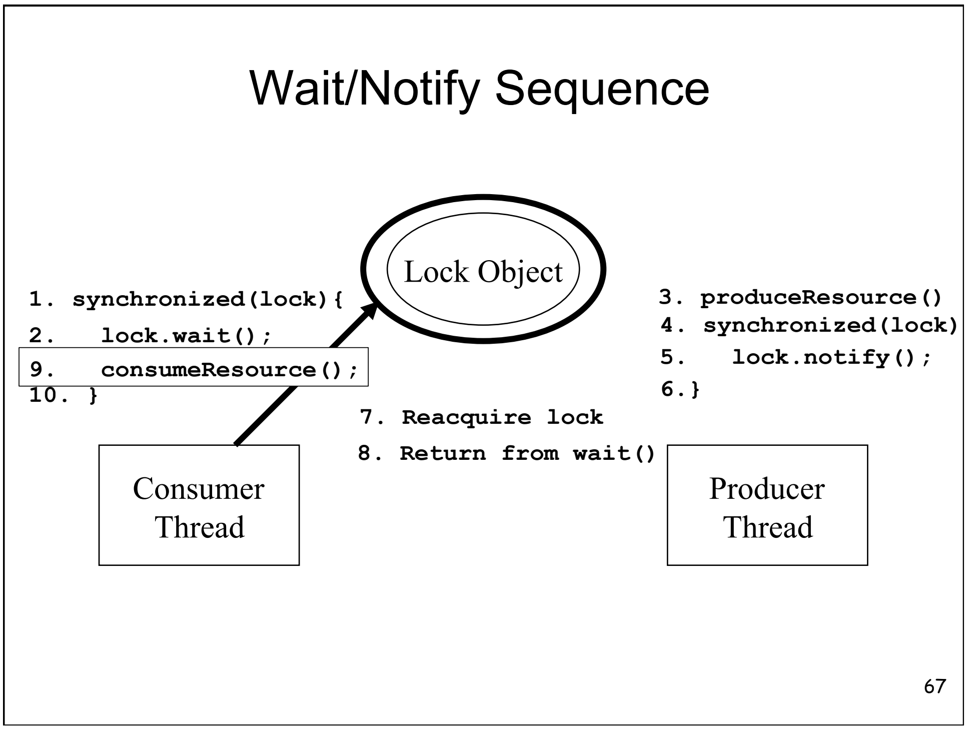


```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

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

7. Reacquire lock

8. Return from wait()



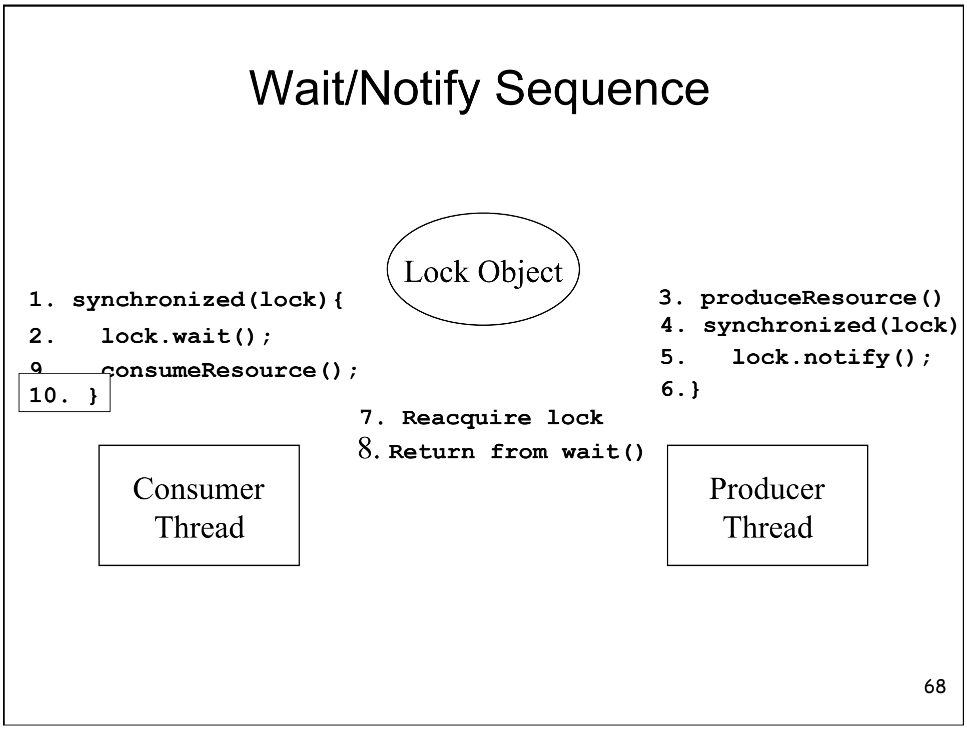
Wait/Notify Sequence

```
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()
```

67



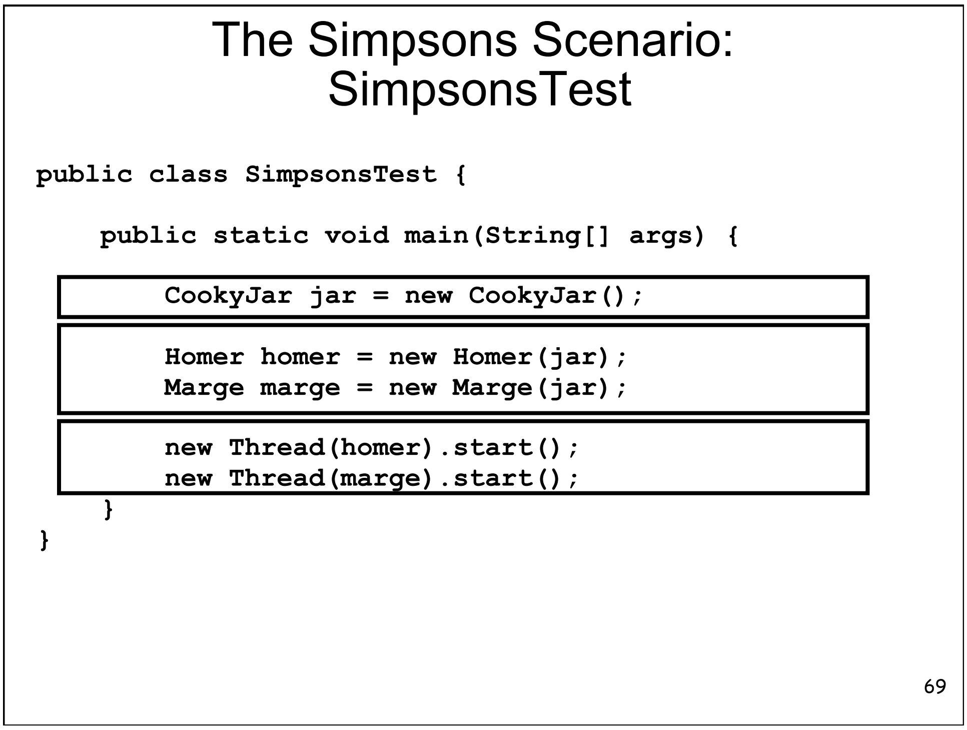
Wait/Notify Sequence

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

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

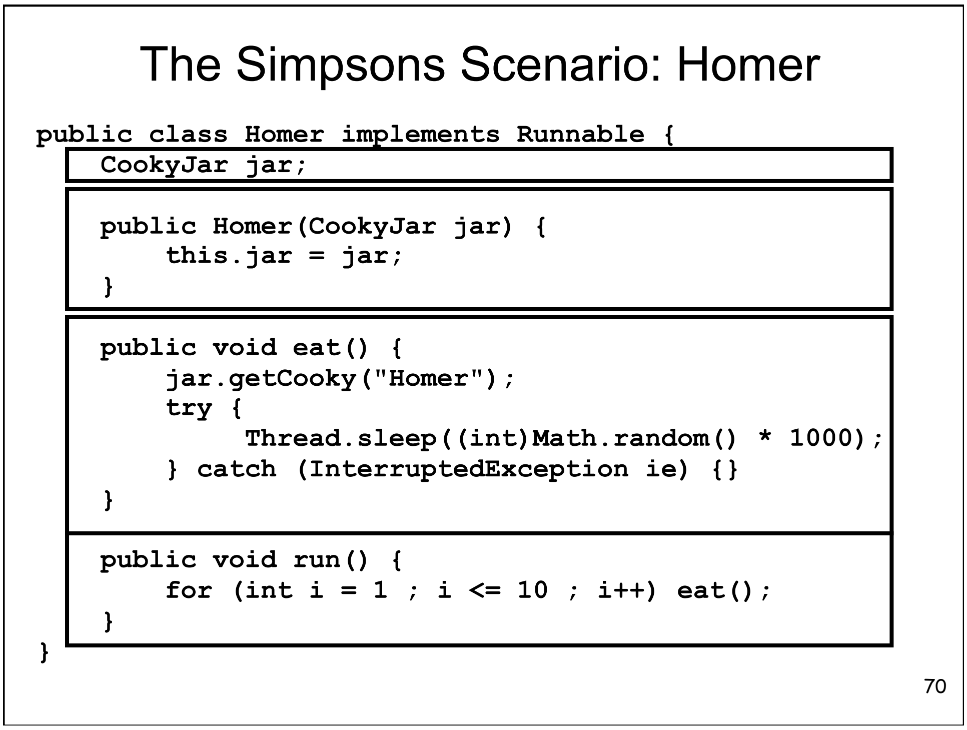
```
7. Reacquire lock
8. Return from wait()
```

68



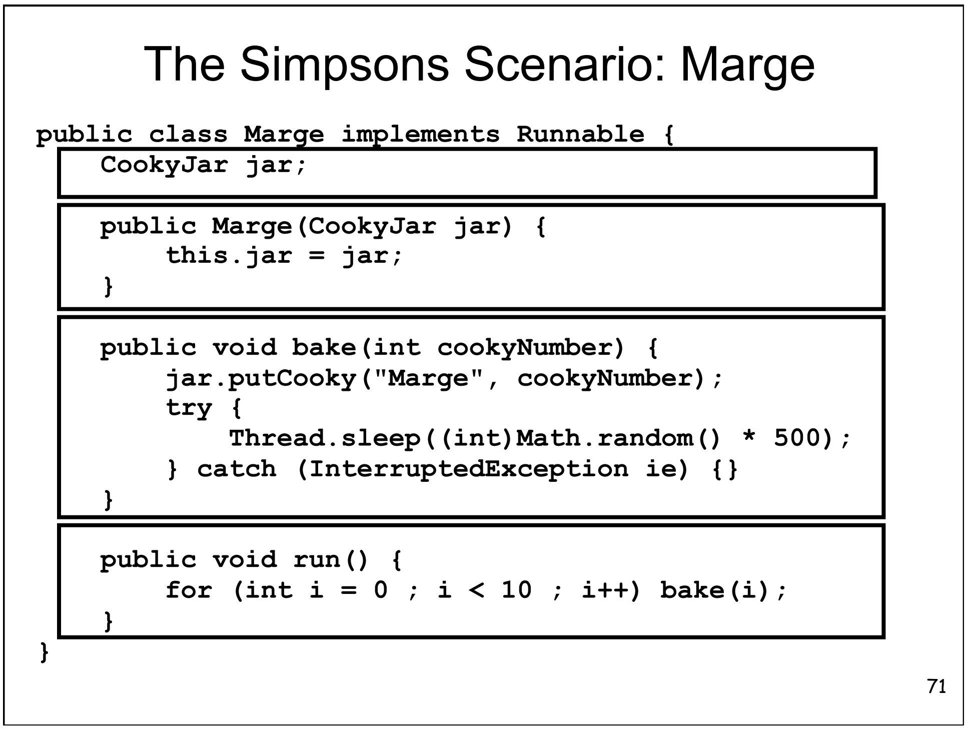
The Simpsons Scenario: SimpsonsTest

```
public class SimpsonsTest {

    public static void main(String[] args) {

        CookyJar jar = new CookyJar();

        Homer homer = new Homer(jar);
        Marge marge = new Marge(jar);

        new Thread(homer).start();
        new Thread(marge).start();
    }
}
```



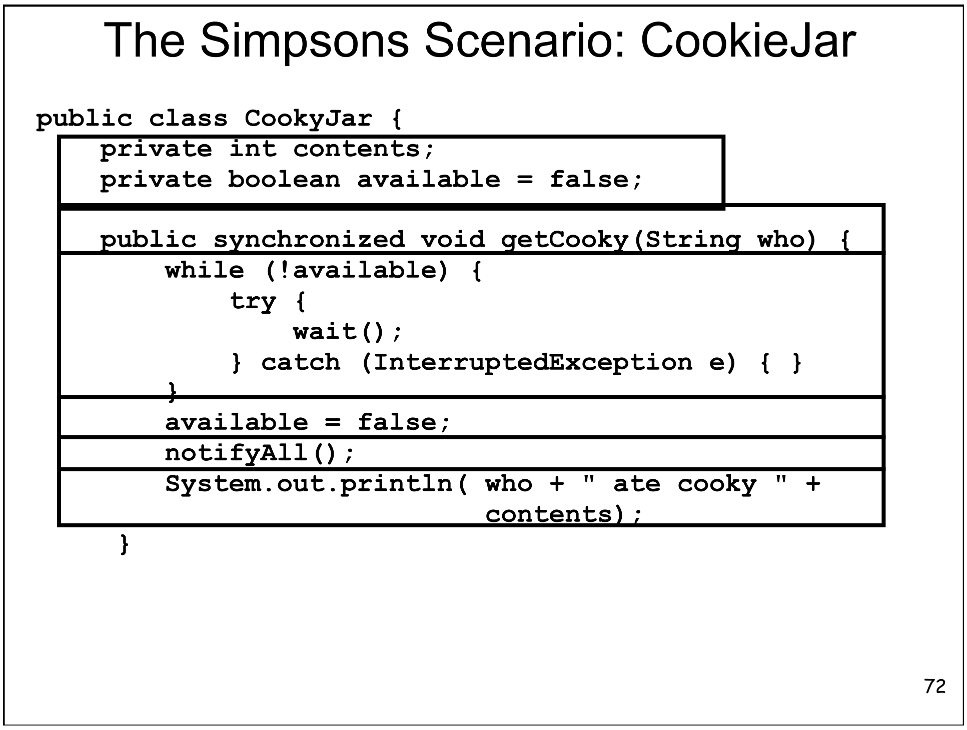
The Simpsons Scenario: Homer

```
public class Homer implements Runnable {
    CookyJar jar;

    public Homer(CookyJar jar) {
        this.jar = jar;
    }

    public void eat() {
        jar.getCooky("Homer");
        try {
             Thread.sleep((int)Math.random() * 1000);
        } catch (InterruptedException ie) {}
    }

    public void run() {
        for (int i = 1 ; i <= 10 ; i++) eat();
    }
}
```



The Simpsons Scenario: Marge

```
public class Marge implements Runnable {
    CookyJar jar;

    public Marge(CookyJar jar) {
        this.jar = jar;
    }

    public void bake(int cookyNumber) {
        jar.putCooky("Marge", cookyNumber);
        try {
            Thread.sleep((int)Math.random() * 500);
        } catch (InterruptedException ie) {}
    }

    public void run() {
        for (int i = 0 ; i < 10 ; i++) bake(i);
    }
}
```



The Simpsons Scenario: CookieJar

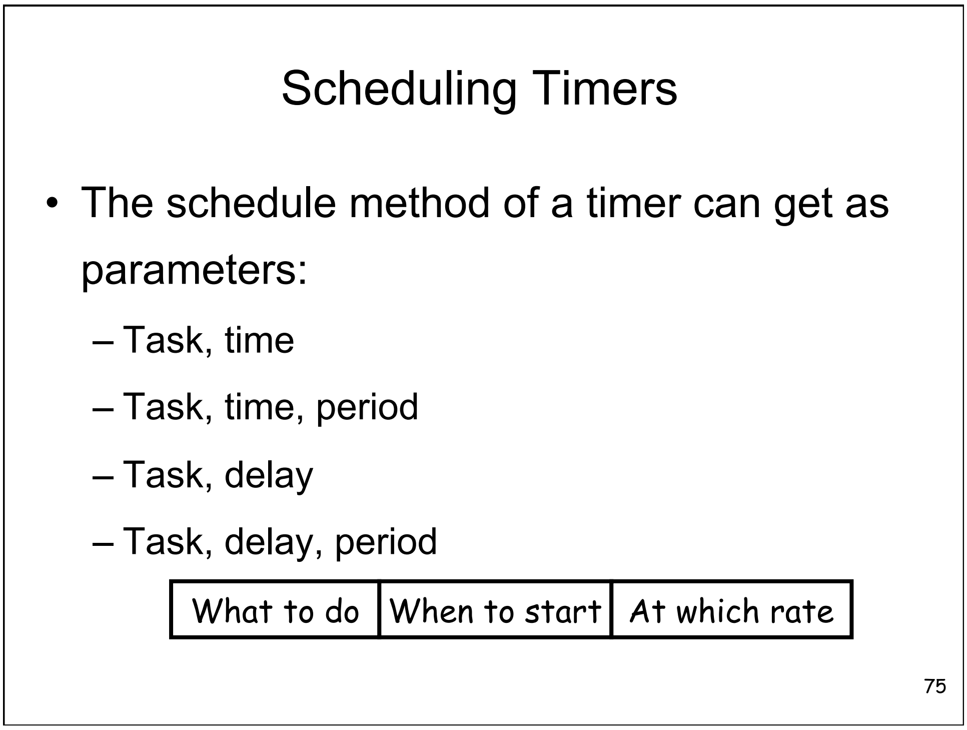
```
public class CookyJar {
    private int contents;
    private boolean available = false;

    public synchronized void getCooky(String who) {
        while (!available) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        notifyAll();
        System.out.println( who + " ate cooky " +
                                    contents);
     }
```

The Simpsons Scenario: CookieJar

```
public synchronized void putCooky(String who,
                                  int value) {
        while (available) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        contents = value;
        available = true;
        System.out.println(who + " put cooky " +
                           contents + " in the jar");
        notifyAll();
    }
}
```

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

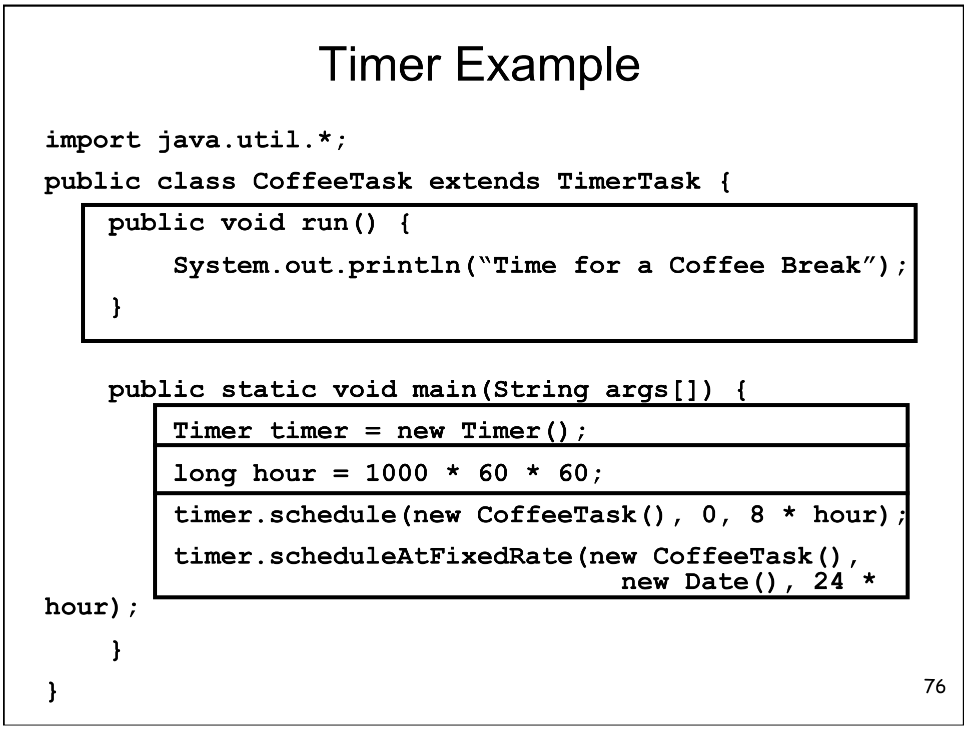


Scheduling Timers

- 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

75



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);
    }
}
```

76

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()