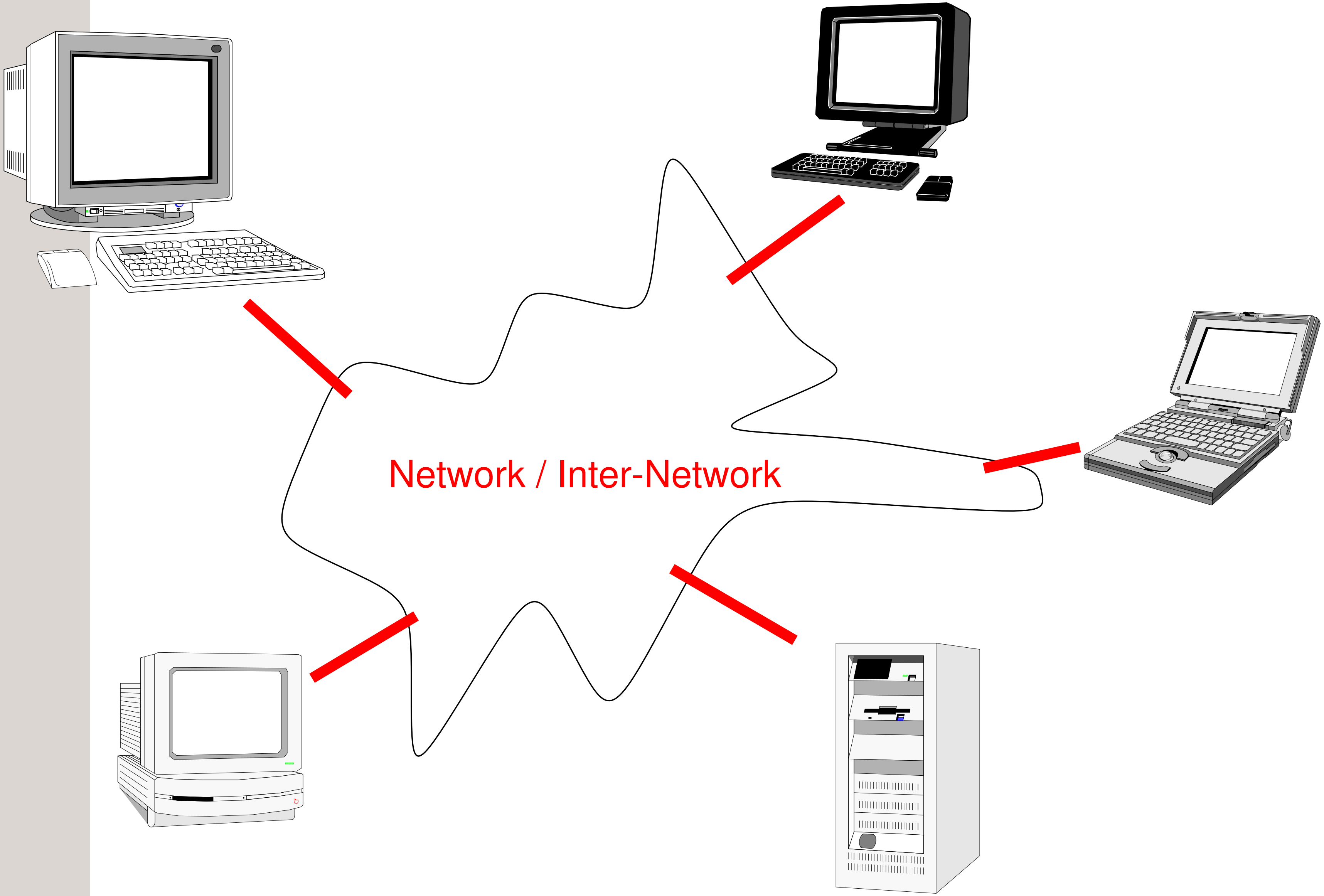
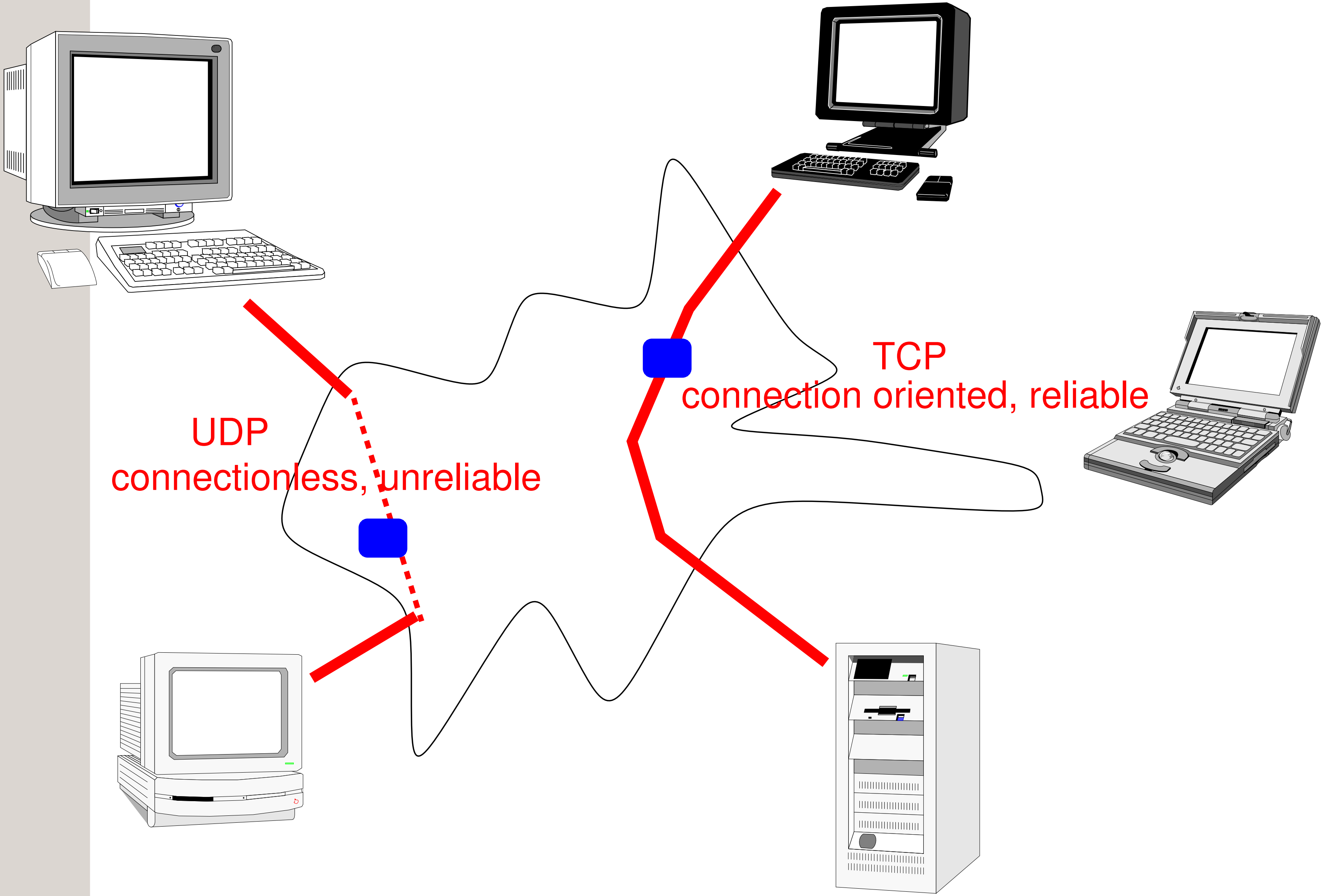
31 Network Programming



31.1 Host Addressing: InetAddress

- IP addresses:
  - DNS form: www4.informatik.uni-erlangen.de
  - "dotted quad" form: 131.188.34.42
- **`java.net.InetAddress`** contains IP address
- **`InetAddress`** has no public constructor, create instances with
  - **`getLocalHost()`**
  - **`getByName(String hostname)`**
  - **`getAllByName(String hostname)`**
- convert **`InetAddress`** instance to different format
  - **`byte[] getAddress()`**: bytes of IP address
  - **`String getHostAddress()`**: "dotted quad" IP address as String
  - **`String getHostName()`**: host name (DNS form)

31.2 Sockets


TCP
connection oriented, reliable
UDP
connectionless, unreliable

31.3 Connection-oriented Sockets

- **`java.net.Socket`**
  - TCP/IP
  - reliable
  - represents a communication endpoint at client and server
  - creating a new socket:

```
socket= new Socket("www4.informatik.uni-erlangen.de",80);
```

  - a connection endpoint is defined by *host and port* (16 bit, < 1024 privileged)
  - **`close`** closes the socket

31.3.1 TCP Client/Server

Server

```
ServerSocket serverSocket = new ServerSocket(5124);
```

Client

31.3.2 TCP Client/Server

Server

```
ServerSocket serverSocket = new ServerSocket(5124);
Socket socket = serverSocket.accept();
```

Client

31.3.3 TCP Client/Server

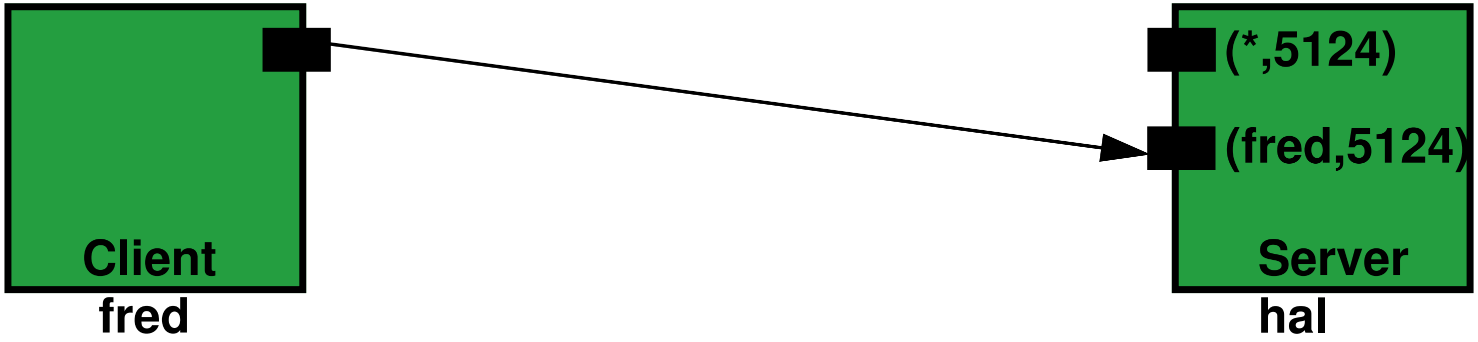
Server

```
ServerSocket serverSocket = new ServerSocket(5124);
Socket socket = serverSocket.accept();
```

Client

```
Socket socket = new Socket("hal",5124);
```

31.3.4 TCP Client/Server



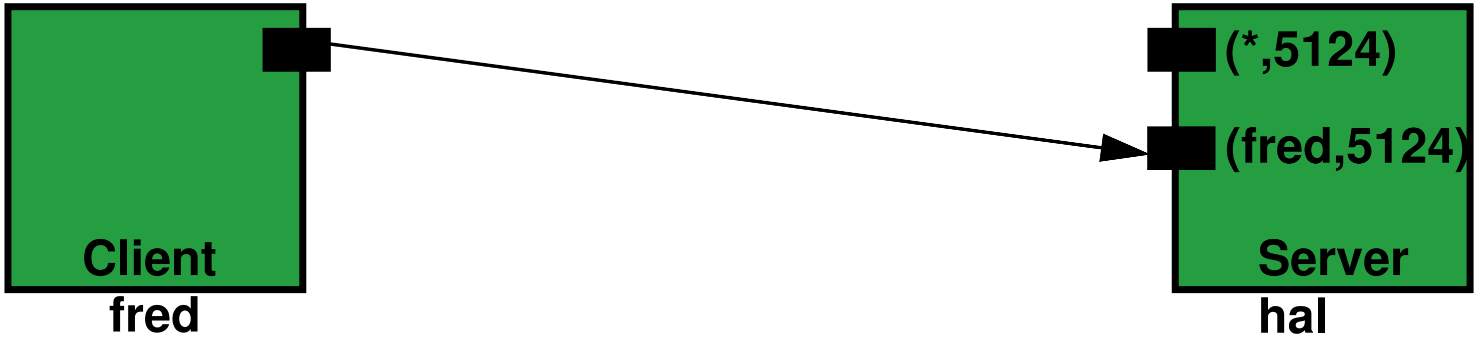
Server

```
ServerSocket serverSocket = new ServerSocket(5124);
Socket socket = serverSocket.accept(); // accept returns
```

Client

```
Socket socket = new Socket("hal",5124);
```

31.3.5 TCP Client/Server



Server

```
ServerSocket serverSocket = new ServerSocket(5124);
Socket socket = serverSocket.accept();
InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();
```

Client

```
Socket socket = new Socket("hal",5124);
InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();
```

31.4 Remarks

- **`java.net.ServerSocket`**
  - is used at server side to wait for client connection requests
  - **`accept`** waits for connection request and returns a new **`Socket`** object

```
ServerSocket serverSocket = new ServerSocket(10412);
Socket socket = serverSocket.accept();
```

  - **`close`** closes the port

- use streams to read/write from/to sockets

```
InputStream inStream = socket.getInputStream();
OutputStream outStream = socket.getOutputStream();
```

- use these streams to create more capable streams

```
DataOutputStream out =
   new DataOutputStream(new BufferedOutputStream(outStream));
```

31.5 Connectionless Sockets

- **`java.net.DatagramSocket`**
  - UDP/IP
  - unreliable: Datagrams can get lost!
  - low latency
  - Constructors:

    `DatagramSocket(int port)`
    bind to local port **`port`**

    `DatagramSocket()`
    bind to arbitrary local port
  - Methods:

    `send(DatagramPacket packet)`
    send packet, you must write receiver address in packet

    `receive(DatagramPacket packet)`
    receive packet, sender address contained in packet

31.5.1 Receiver

- receive packets at specific port

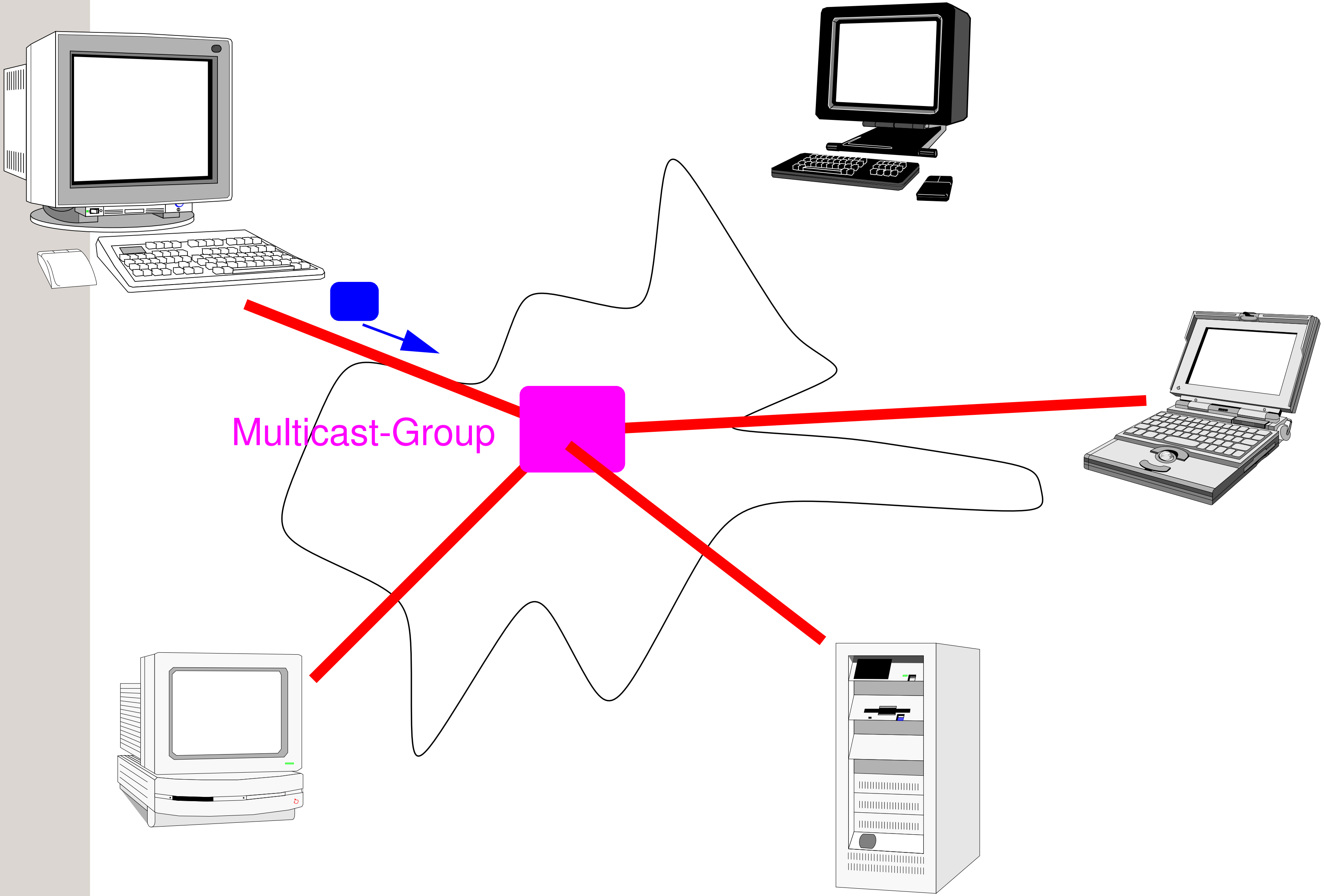
```
DatagramSocket socket = new DatagramSocket(10412);
byte[] buf = new byte[1024];
DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
InetAddress from = packet.getAddress();
int bytesReceived = packet.getLength();
```

31.5.2 Sender

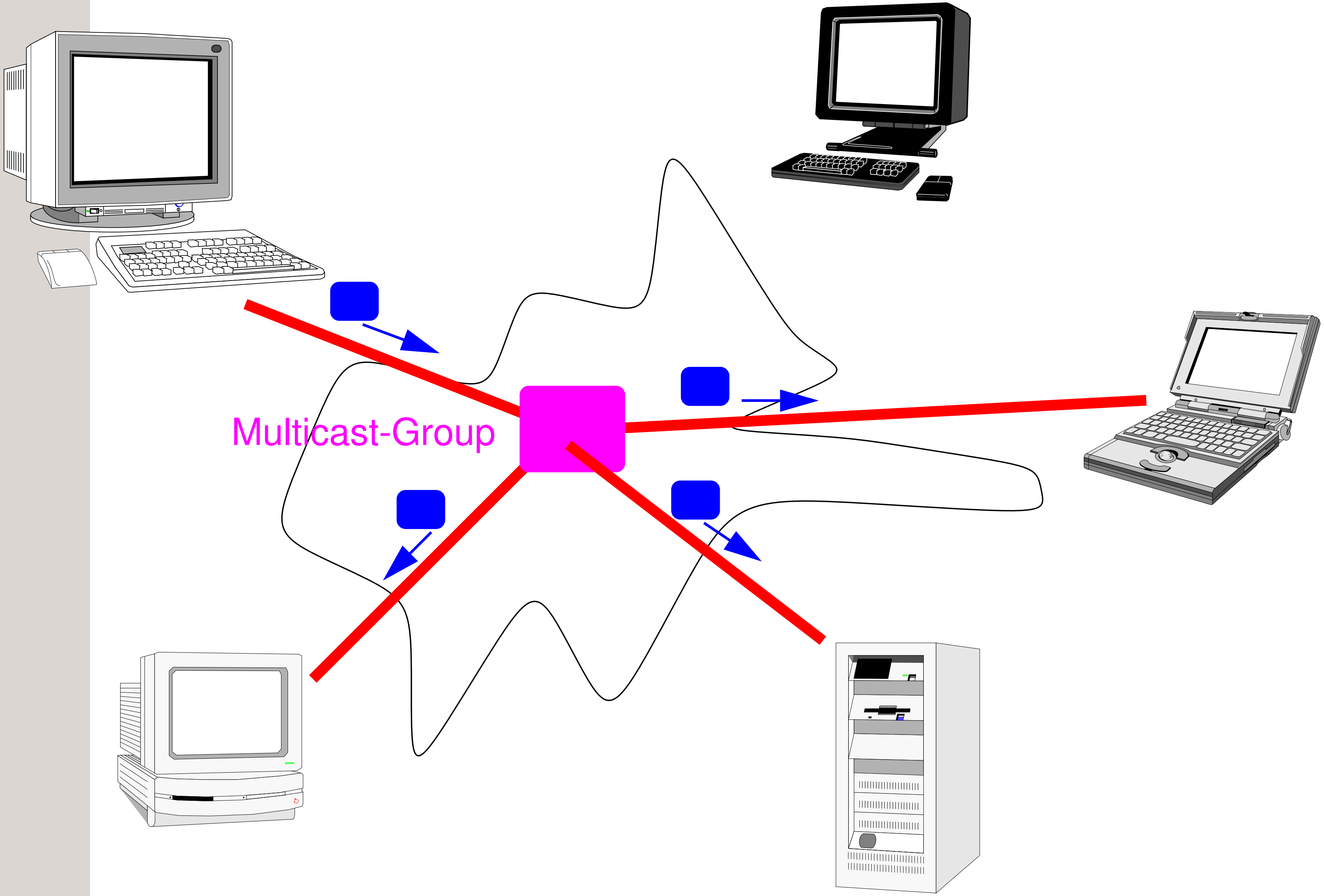
- send packets from arbitrary port

```
InetAddress addr = InetAddress.getByName("faui40");
int port = 10412;
DatagramSocket socket = new DatagramSocket();
byte[] buf = new byte[1024];
buf[0] = ...
DatagramPacket packet = new DatagramPacket(buf, buf.length,
                                           addr,port);
socket.send(packet);
```

31.6 Multicast-Sockets



31.6 Multicast-Sockets



31.6 Multicast-Sockets

- **`java.net.MulticastSocket`**
  - connection-less (subclass of **`DatagramSocket`**)
  - uses *class D* IP-addresses (**`224.0.0.1`** to **`239.255.255.255`**)
  - you can send packets after creating the socket
  - to receive packets you must join the group with **`joinGroup()`**
  - packet propagation is controlled by time-to-live parameter (TTL) (decremented when crossing network border)

```
InetAddress group = InetAddress.getByName("228.5.6.7");
MulticastSocket socket = new MulticastSocket(6789);
socket.setTTL((byte)2);
socket.joinGroup(group);
```

31.7 Applets and Sockets

- Security restriction:
  - applets can only open connections to the host they are loaded from
  - use **`getDocumentBase().getHost()`** to get the Web servers hostname

31.8 Summary

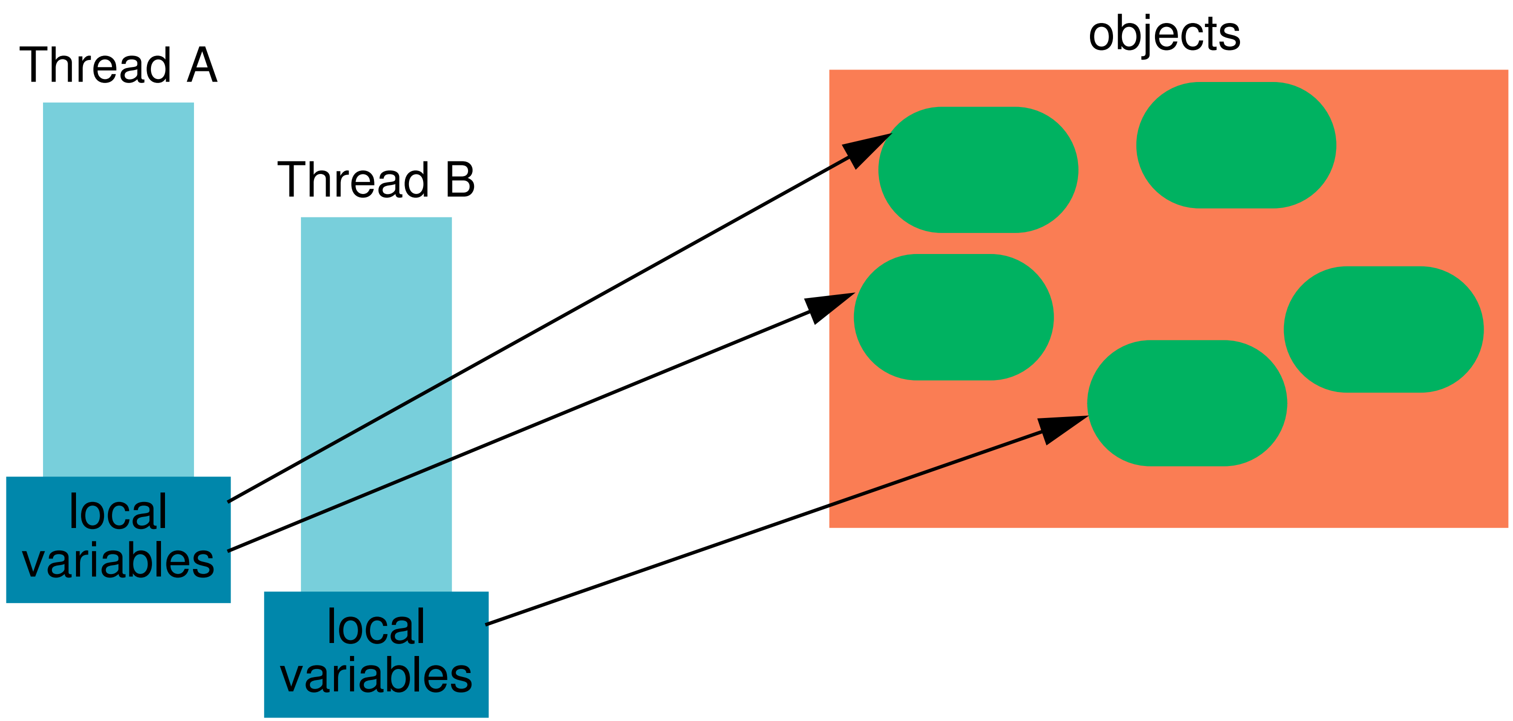
- **`Socket`**: one end of a two-way TCP communication (server or client)
  - containes target address/port and local port
  - use streams for reading/writing
- **`SocketServer`**: a server end point, creates **`Socket`** instances
- **`DatagramSocket`**: UDP communication
  - use **`send()`**/**`receive()`** for communication
  - target address is contained in **`DatagramPacket`** instance
- **`MulticastSocket`**: Multicast UDP communication
  - reserved range of IP-Addresses
  - use **`joinGroup()`** before receiving from a multicast goup

32 Threads

- Reference
  - D. Lea. *Concurrent Programming in Java - Design Principles and Patterns*. Second Edition. The Java Series. Addison-Wesley 1999.

32.1 Threads

- What is a thread?
  - ◆ unit of activity with own program counter, own registers, and own stack
  - ◆ all threads use the same address space



32.2 Advantages / Disadvantages

- Advantages
  - execute parallel algorithms on a multiprocessor
  - waiting for slow devices (e.g. network, user) does not block whole program
- Disadvantages
  - complex semantics
  - difficult to debug
  - John Ousterhout. *Why Threads Are A Bad Idea (for most purposes)*. (available from the OODS web page)

32.3 Thread Creation (1)

1. Subclass **`java.lang.Thread`** and override the **`run()`** method.
2. Create an instance of this class and call the **`start()`** method at this instance.

```
class Test extends Thread {
    public void run() {
        System.out.println("Test");
    }
}

Test test = new Test();
test.start();
```

32.4 Thread Creation (2)

1. Implement the interface **`Runnable`**, this requires implementing a **`run()`** method
2. Create a new **`Thread`** instance by passing the **`Thread`** constructor your **`Runnable`** object.
3. Call the **`start()`** method at the **`Thread`** object.

```
class Test implements Runnable {
    public void run() {
        System.out.println("Test");
    }
}

Test test = new Test();
Thread thread = new Thread(test);
thread.start();
```

32.5 Multithreading Problems

```
public class Test implements Runnable {
  public int a=0;
  public void run() {
    for(int i=0; i<100000; i++) {
      a = a + 1;
    }
  }
```

What is the result of this program?

```
  public static void main(String[] args) {
    Test value = new Test();
    Thread t1 = new Thread(value);
    Thread t2 = new Thread(value);
```

create two threads that use the same object

```
    t1.start();
    t2.start();
```

start the two threads

```
    try {
      t1.join();
      t2.join();
```

wait for the threads to finish

```
    } catch(Exception e) {
      System.out.println("Exception");
    }
    System.out.println("Expected a=200000; a="+value.a);
  }
}
```

Result

- Results of several runs: 173274, 137807, 150683
- What happens when **`a = a + 1`** is executed?

```
LOAD a into Register
ADD 1 to Register
STORE Register into a
```

- 2 possible sequences of actions when two threads are involved (initial a=0):

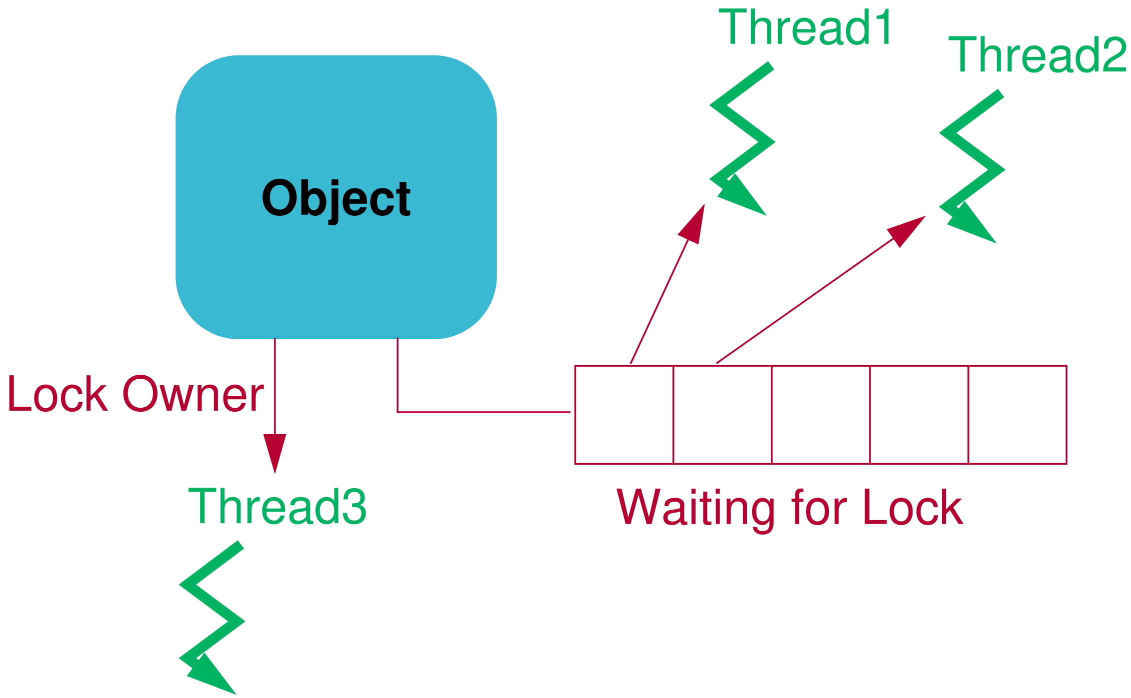
1. T1-load: a=0, Reg1=0
2. T1-add: a=0, Reg1=1
3. T1-store: a=1, Reg1=1
4. T2-load: a=1, Reg2=1
5. T2-add: a=1, Reg2=2
6. T2-store: **a=2**, Reg2=2

1. T1-load: a=0, Reg1=0
2. T2-load: a=0, Reg2=0
3. T1-add: a=0, Reg1=1
4. T1-store: a=1, Reg1=1
5. T2-add: a=1, Reg2=1
6. T2-store: **a=1**, Reg2=1

- The three operations must be executed in one step (atomically)!!

32.5.2 Synchronization

- every object can be used as a lock (monitor)



32.5.3 synchronized

- use **`synchronized`** to acquire and release locks
- declare method or block as **`synchronized`**

```
class Test {
    public synchronized void m() { ... }
    public void n() { ...
            synchronized(this) {
                ...
            }
     }
}
```

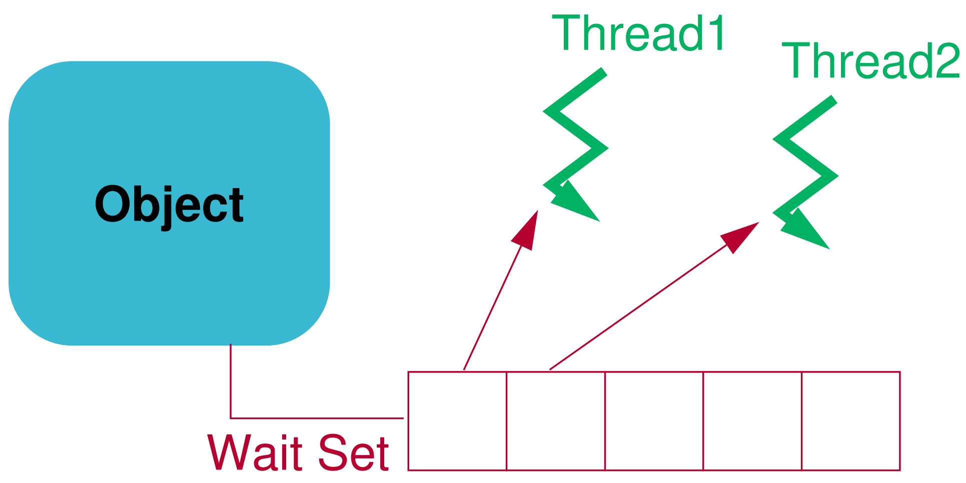
- a thread can acquire one lock multiple times (recursive lock)
- improved example: **`synchronized(this) { a = a + 1; }`**

32.5.4 Use of synchronized Methods

- no **`synchronized`** necessary
  - ◆ if code always runs in single-threaded context
  - ◆ for simple getter methods (see exception below)
- use **`synchronized`**
  - ◆ if object is written
  - ◆ if computations with the object are done (even if they *only read* the state)
  - ◆ for getter methods that read **`long`** or **`double`** types
  - ◆ for simple getter methods that must be blocked if a state update happens

32.6 Condition Variables

- threads must wait for a condition to become true
- two ways for waiting
  - active (polling)
  - passive (condition variables)
- every object can be used as a condition variable (every object is associated with a "wait set" = list of waiting threads)

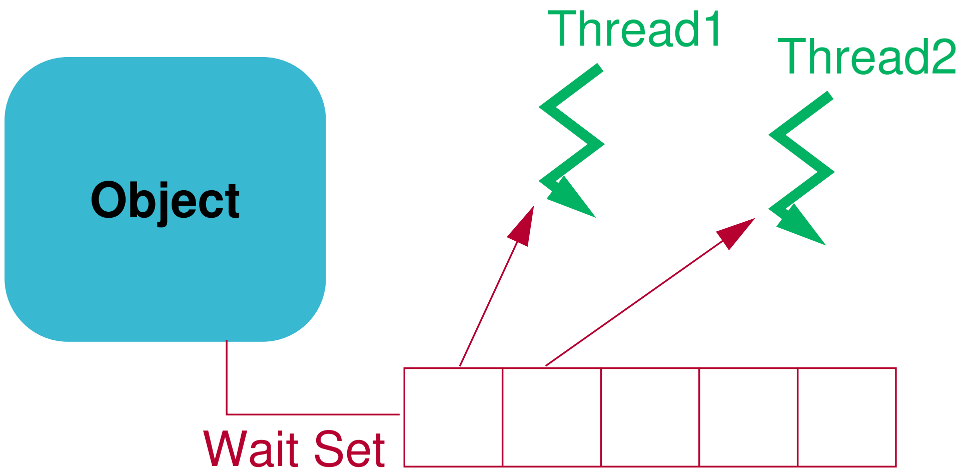


32.6.1 wait and notify

- `Object` contains methods to use the object as a condition variable
  - **`wait`**: wait for condition (inserts thread into the wait set)

```
while(! condition) { wait(); }
```

  - **`notify`**: state changed, condition could be true, inform one waiting thread (removes one thread from the wait set)
  - **`notifyAll`**: wake up all waiting threads (expensive!) (removes all threads from the wait set)



32.6.2 Condition Variables and Locks

- **`wait`**, **`notify`**, **`notifyAll`** can only be executed if calling thread holds object lock
- **`wait`** releases the lock before blocking (atomically)
  - another thread can then change the object state (making the condition true)
- acquiring the lock and unblocking happens atomically
- **`wait`** can be called with a timeout

32.6.3 Condition Variables Example

- PV system: condition is *count > 0*

```
class Semaphore {
    private int count;
    public Semaphore(int count) { this.count = count; }
    public synchronized void P() throws InterruptedException{
        while (count <= 0) {
            wait();
        }
        count--;
    }
    public synchronized void V() {
        count++;
        notify();
    }
}
```

32.6.4 Condition Variables Example

- Order system: one thread accepts customer requests (**`SecretaryThread`**) and another thread processes them (**`WorkerThread`**)

```
class SecretaryThread implements Runnable {
   public void run() {
      for(;;) {
         Customer customer = customerLine.nextCustomer();
         WorkerThread worker = classify(customer);
         worker.insertCustomer(customer);
      }
   }
}

interface WorkerThread {
   public void insertCustomer(Customer c);
}
```

32.6.4 Condition Variables Example

- Worker

```
class SpecificWorker implements Runnable, WorkerThread {
   public void run() {
      for(;;) {
        Customer customer;
        synchronized (this) {
           while(queue.empty()) wait();
           customer = queue.next();
        }
         // do something nice with customer
         // ...
      }
   }
   public synchronized void insertCustomer(Customer c) {
      queue.insert(c);
      notify();
   }
}
```

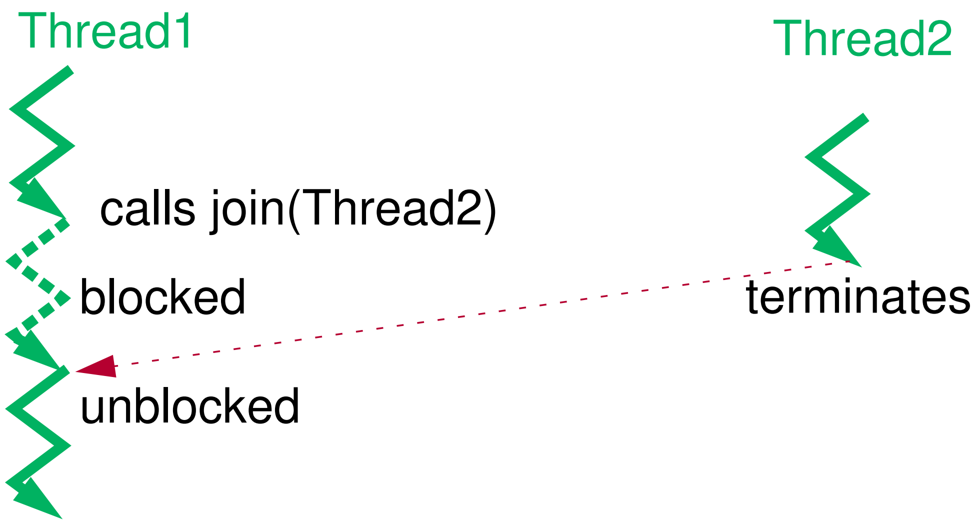
32.7 sleep

- thread has **`sleep(long n)`** method to go to sleep for n milliseconds
- thread can be suspended after return from **`sleep()`**
- sleep method is a static method of Thread; called like
  `Thread.sleep(...);`

32.8 join

- a thread can wait for another thread to die

```
workerThread = new Thread(worker);
...
workerThread.join();
worker.result();
```



32.9 Daemon Threads

- daemon threads are used for background tasks
- should not be used for main task of program
- if all *non-daemon* threads are dead the program is finished
- How can you tell if a thread should be daemon thread?
  - You cannot state a termination condition for the thread.
- Important methods of the **`Thread`** class:
  - **`setDaemon(boolean switch)`**: turns daemon property on/off
  - **`boolean isDaemon()`**: tests if thread is a daemon

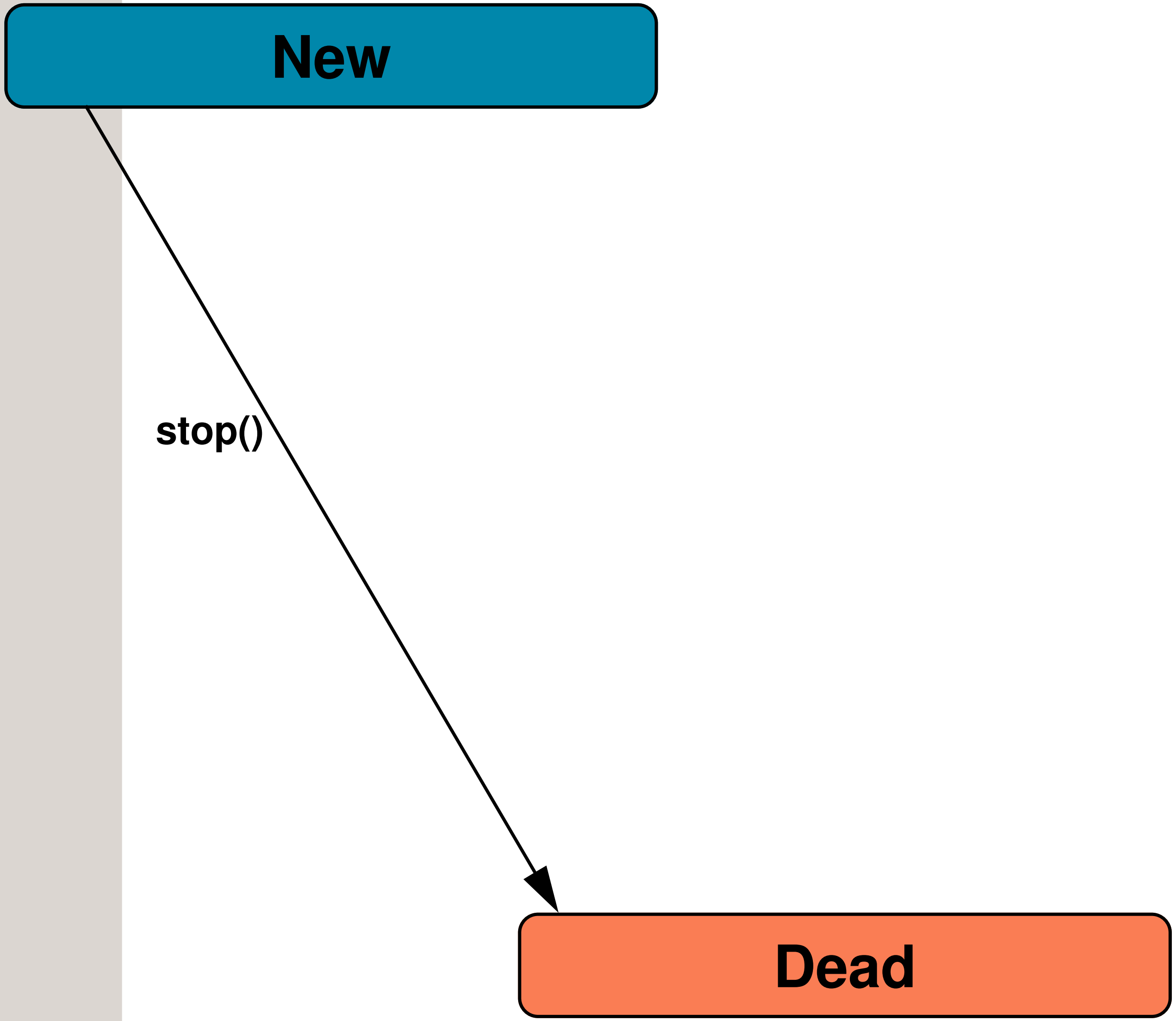
32.10 ThreadGroup

- Group of related threads (**`ThreadGroup`**):
  - Can contain threads and other thread groups.
  - Every thread can only influence threads in its own ThreadGroup
- Methods that can only used with threads of own thread group:
  - **`list()`**
  - **`stop()`**
  - **`suspend()`**
  - **`resume()`**

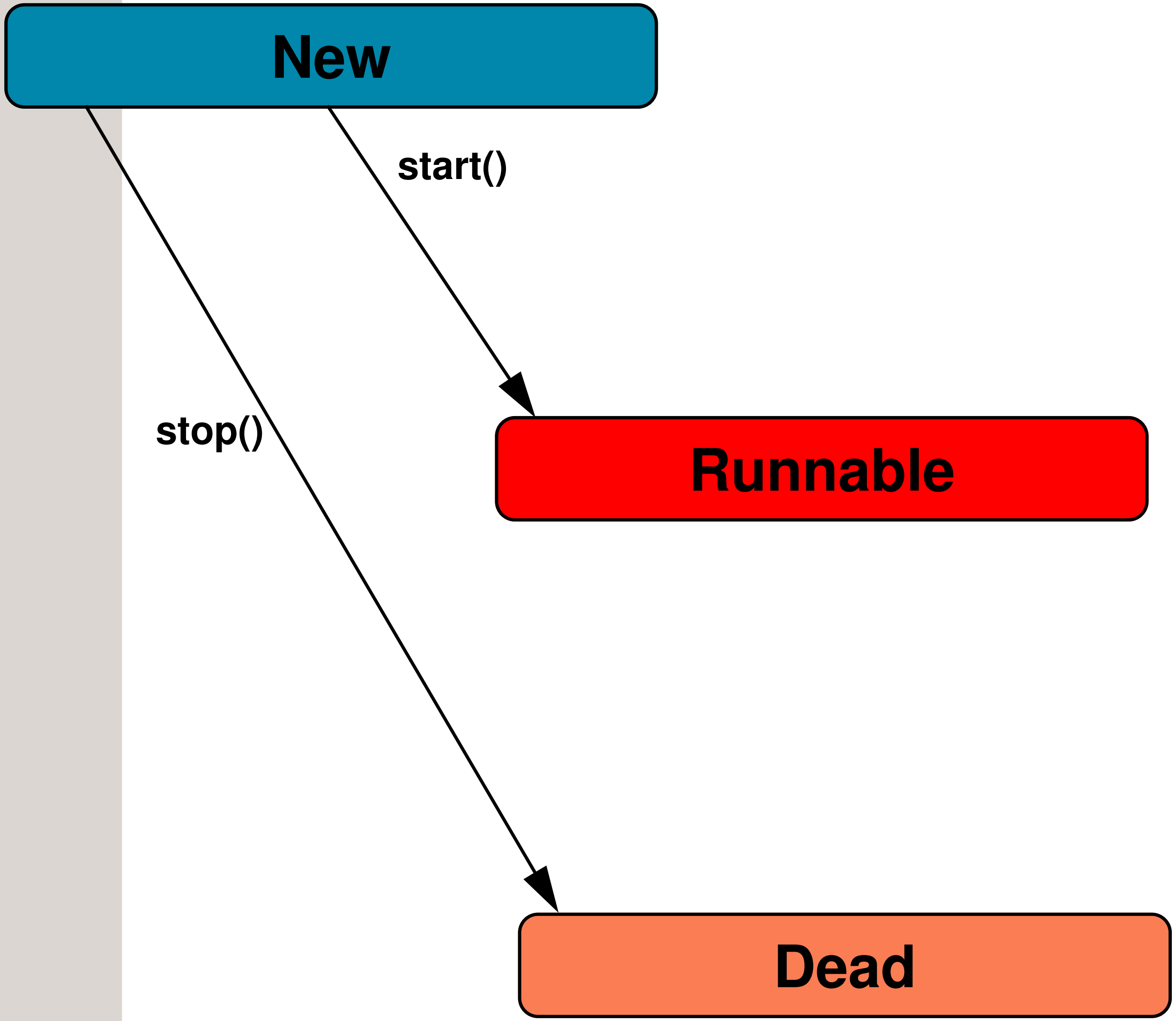
32.11 Thread States

New

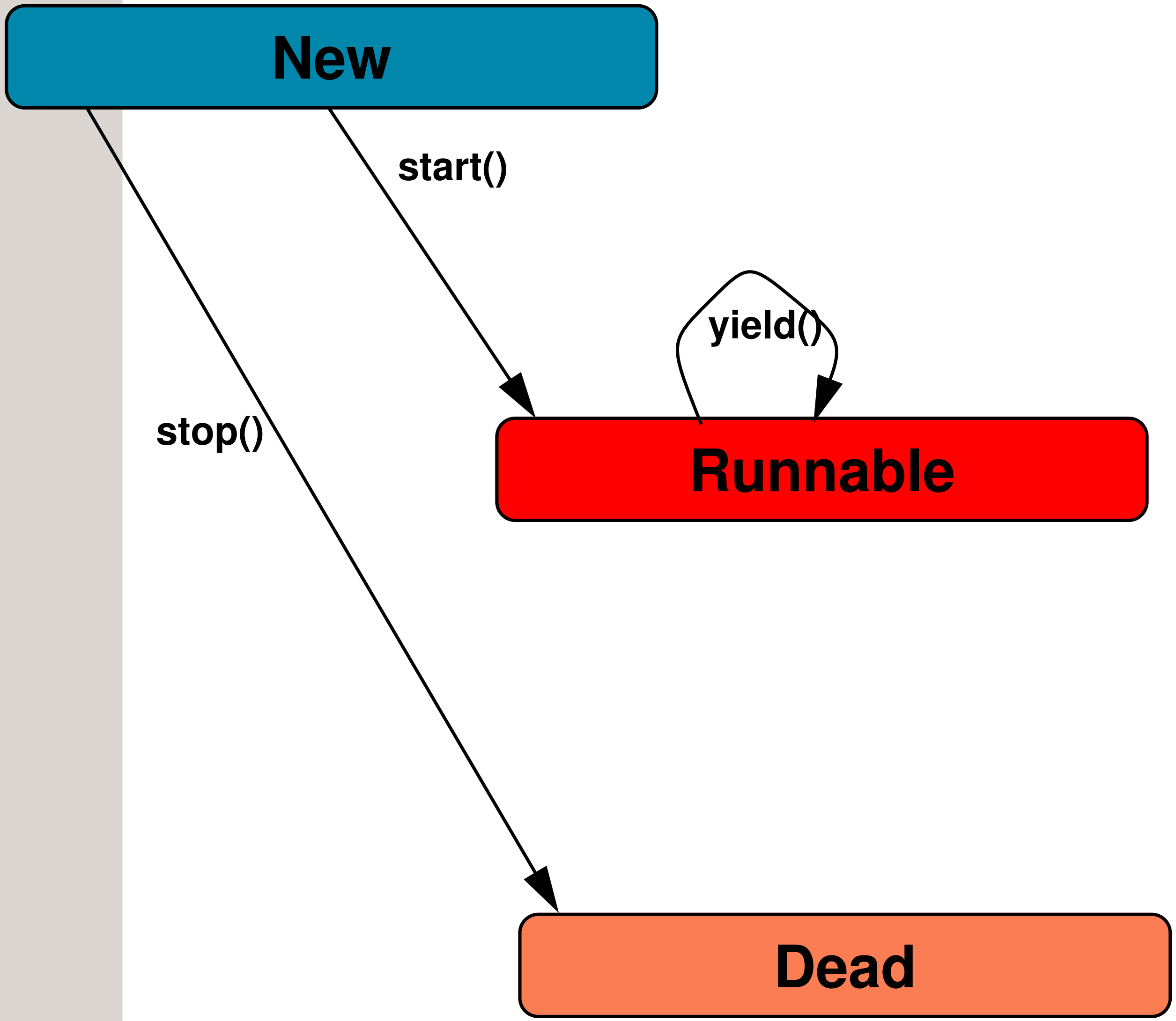
32.11 Thread States



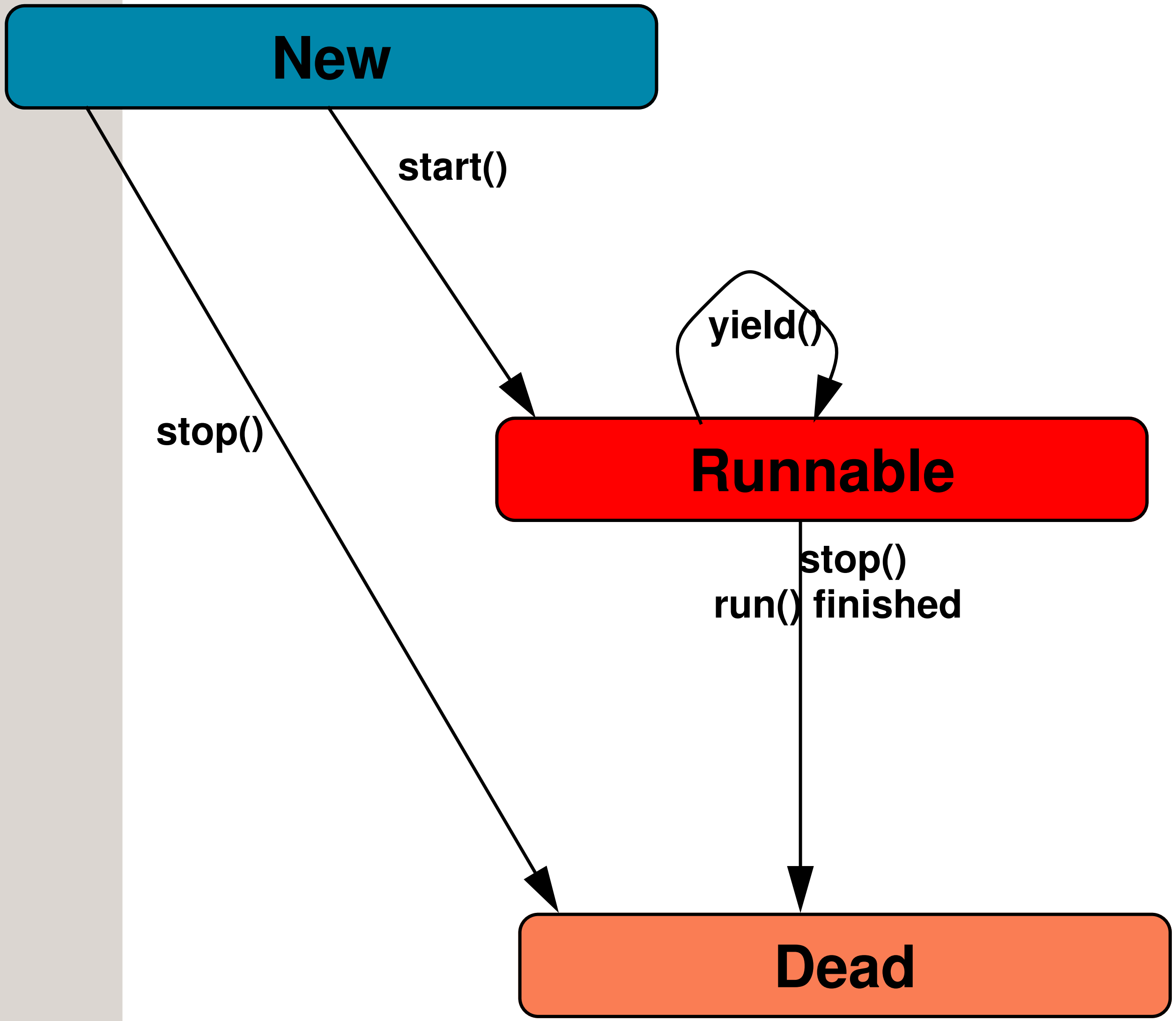
32.11 Thread States



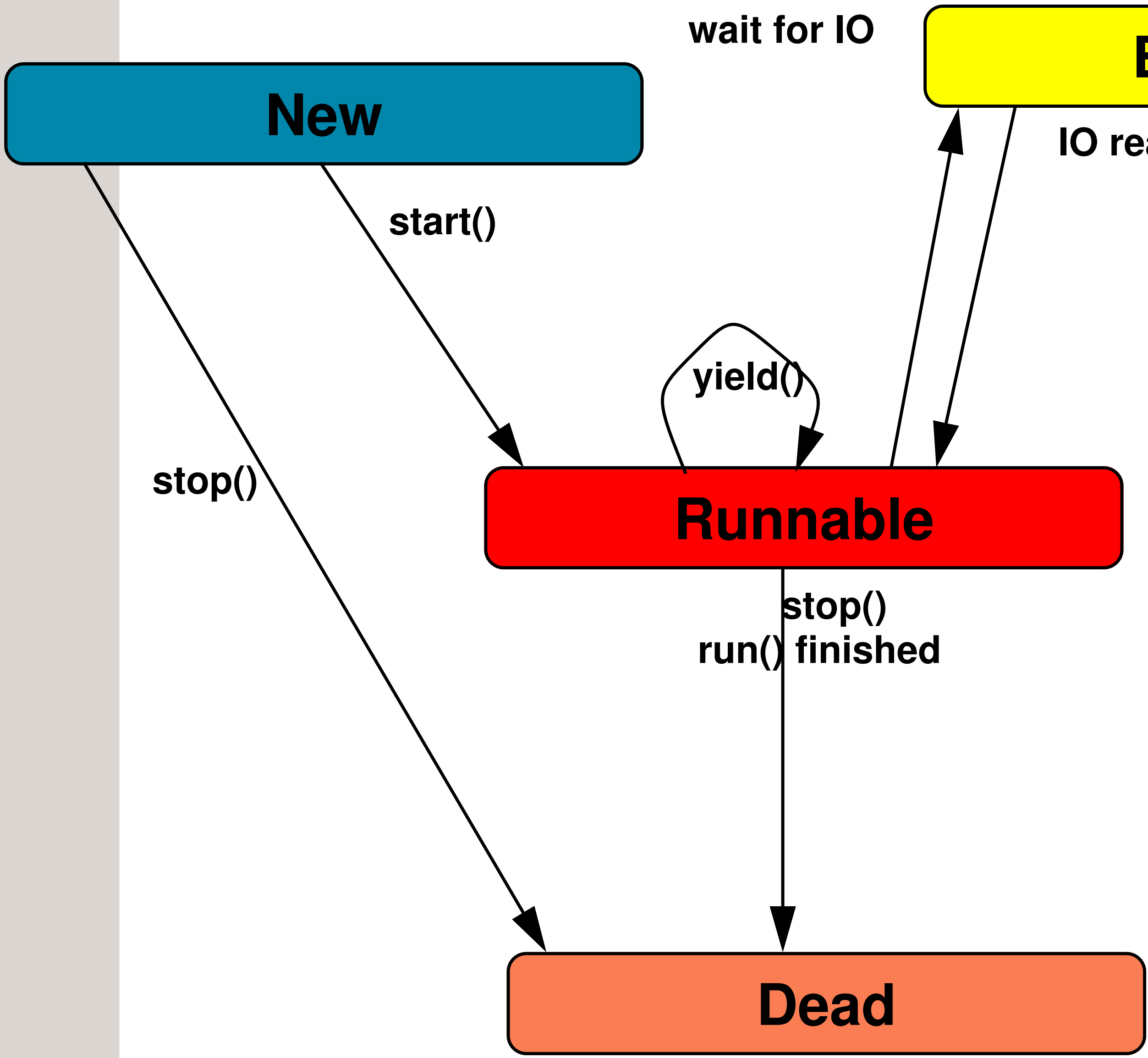
32.11 Thread States



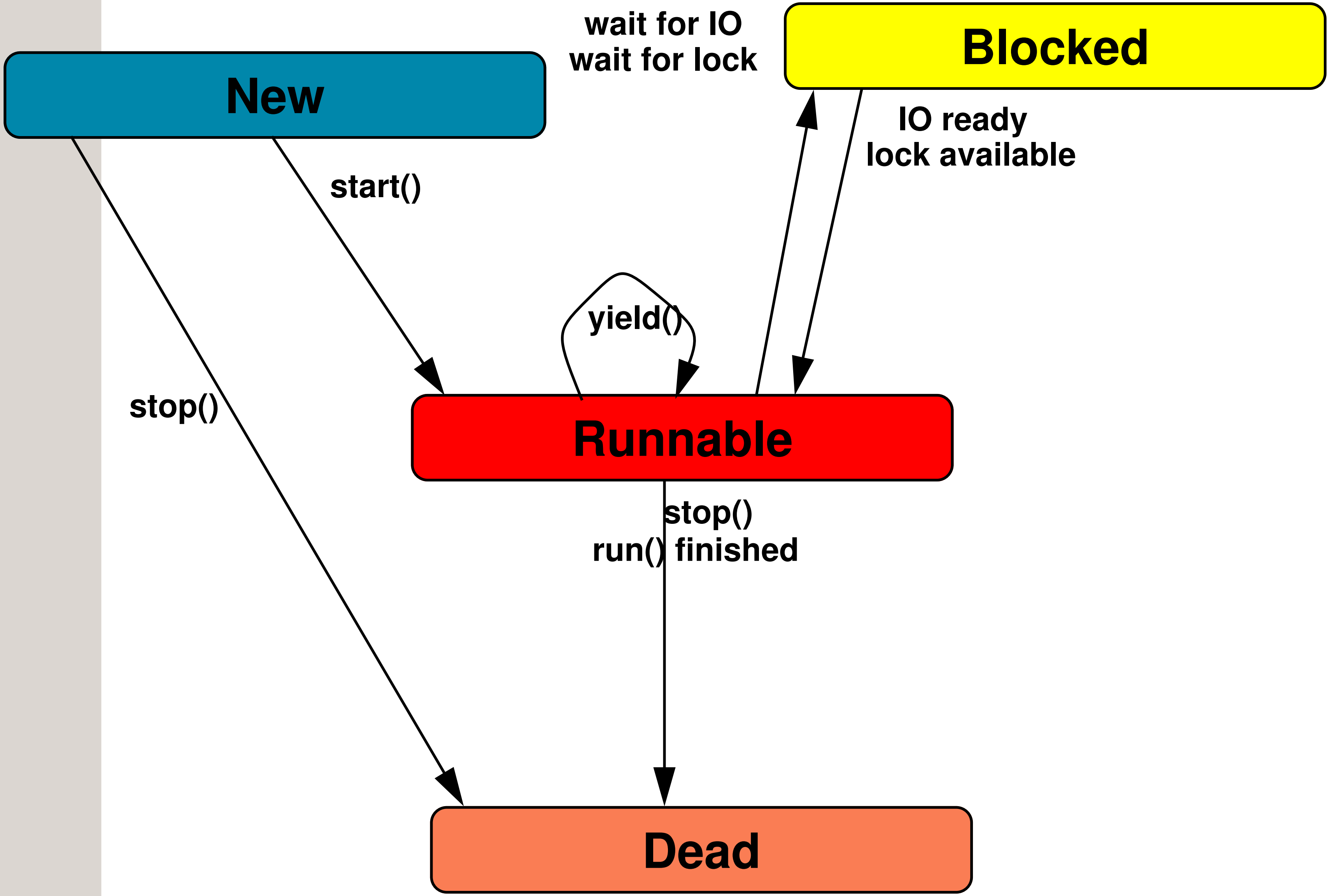
32.11 Thread States



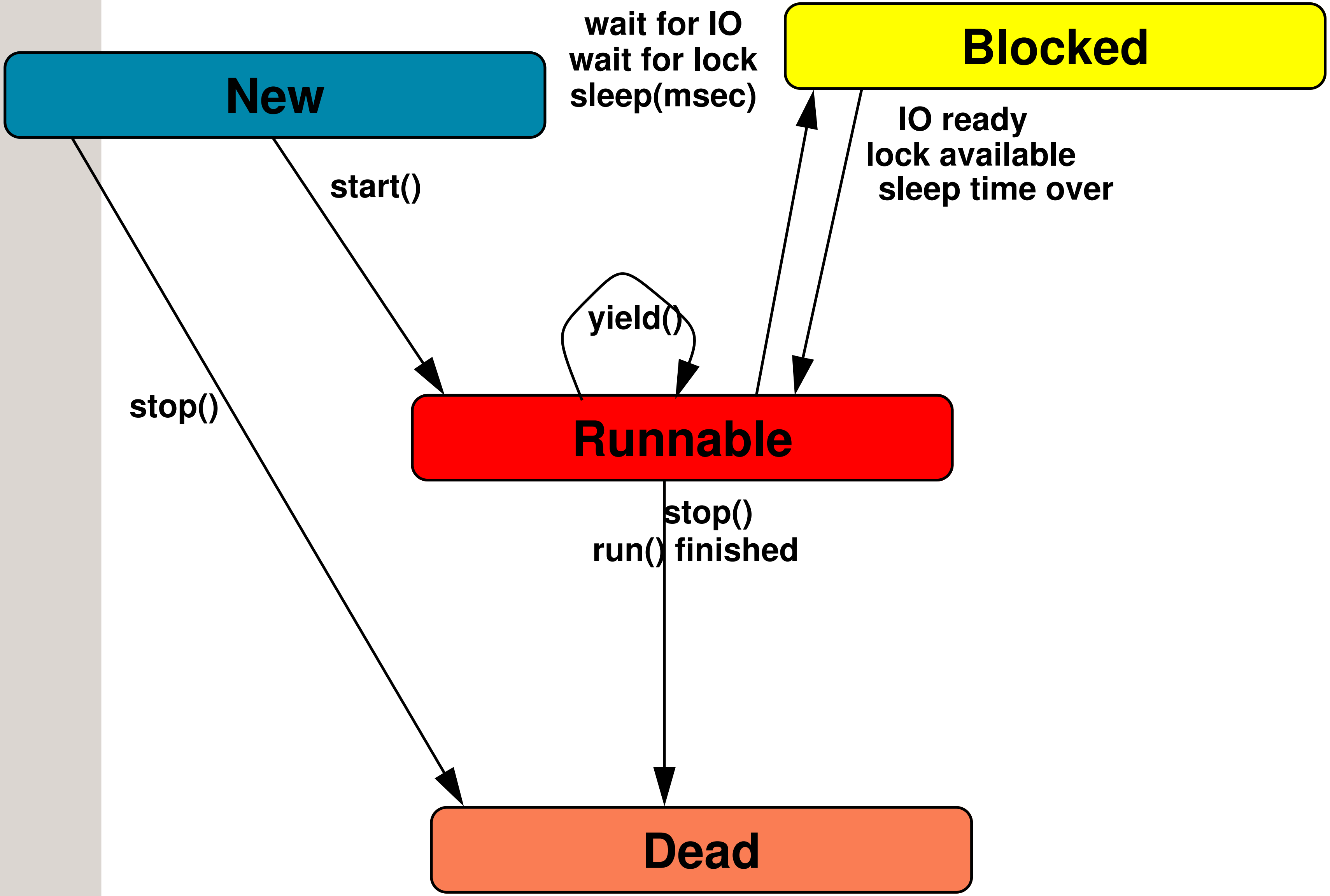
32.11 Thread States



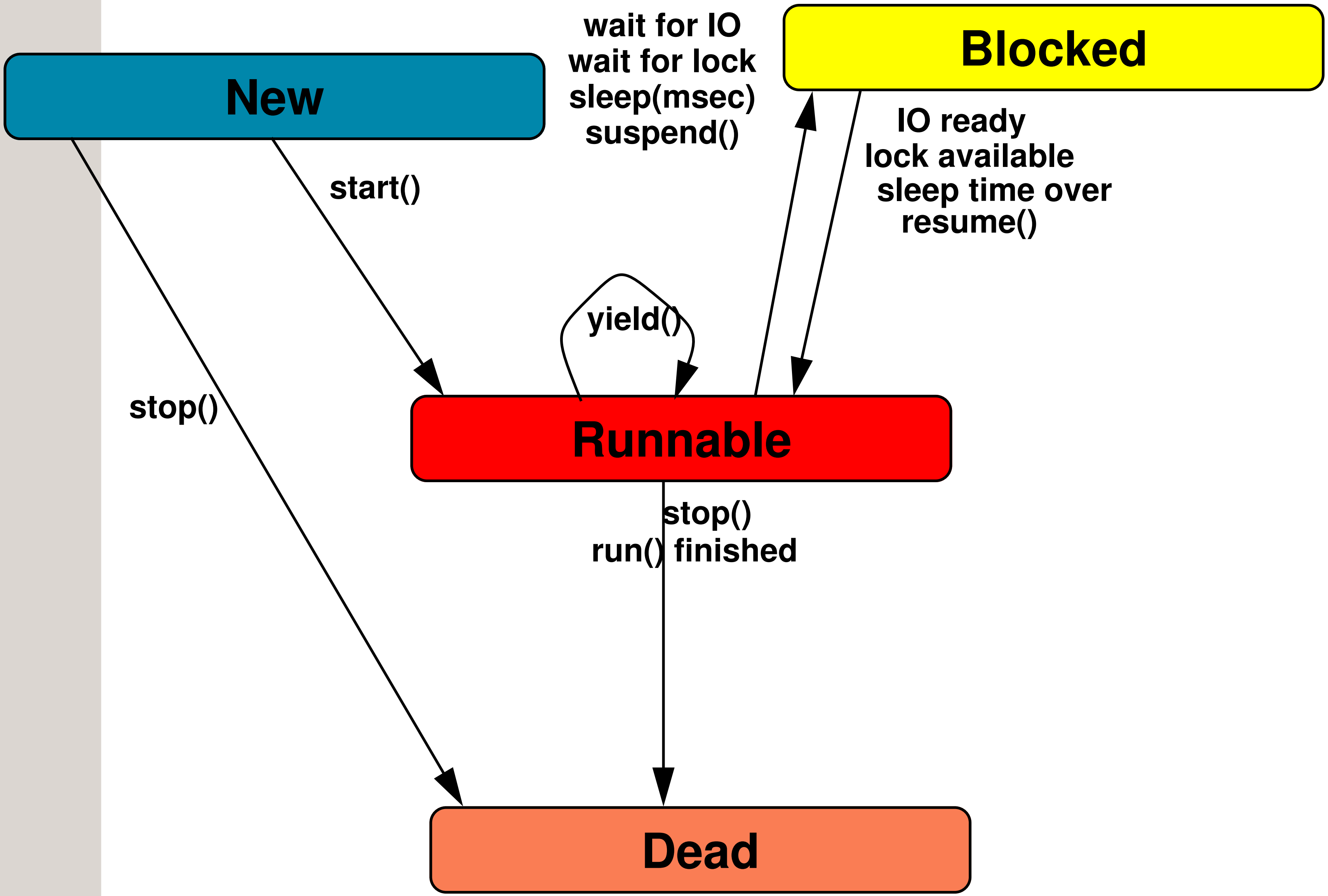
32.11 Thread States



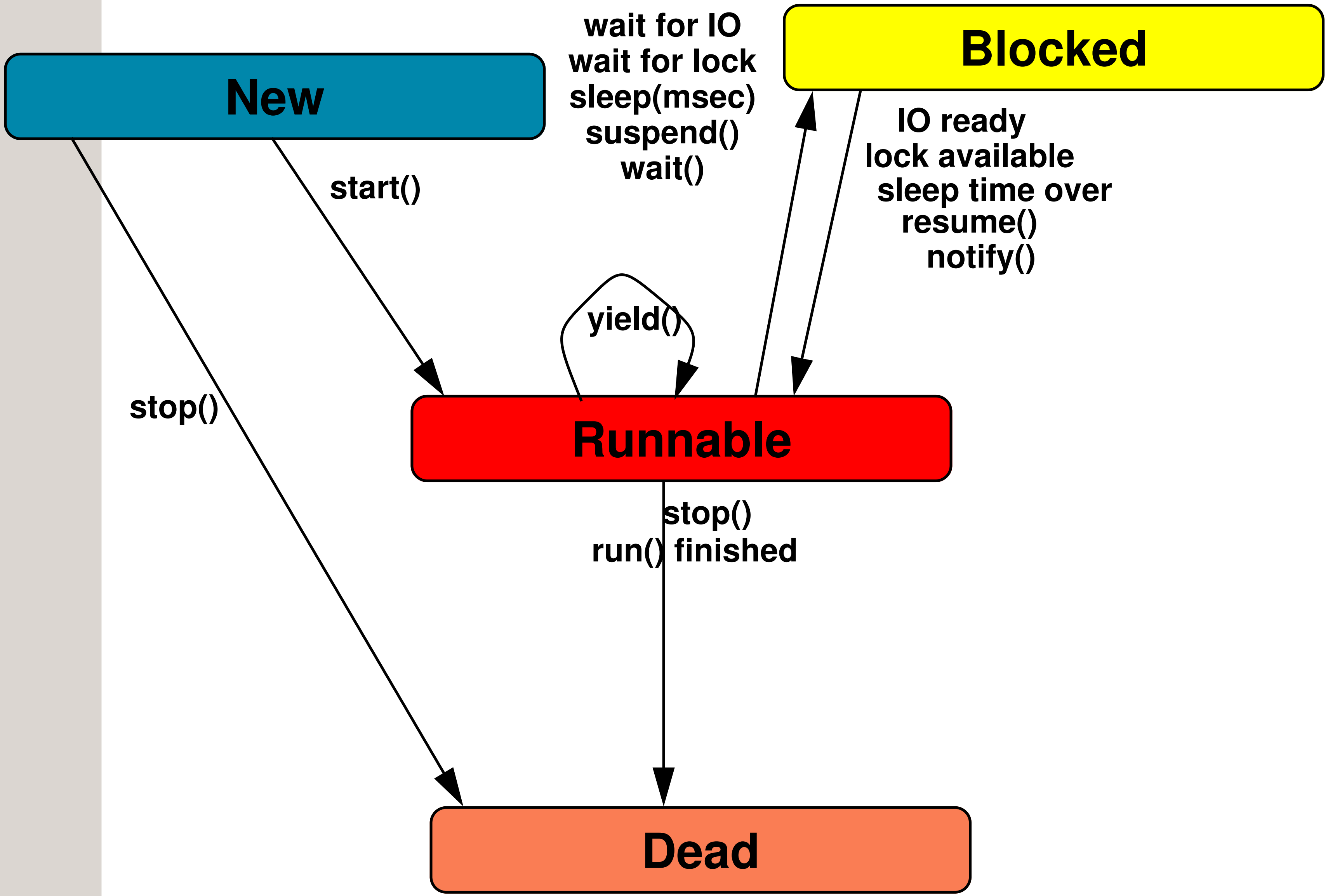
32.11 Thread States



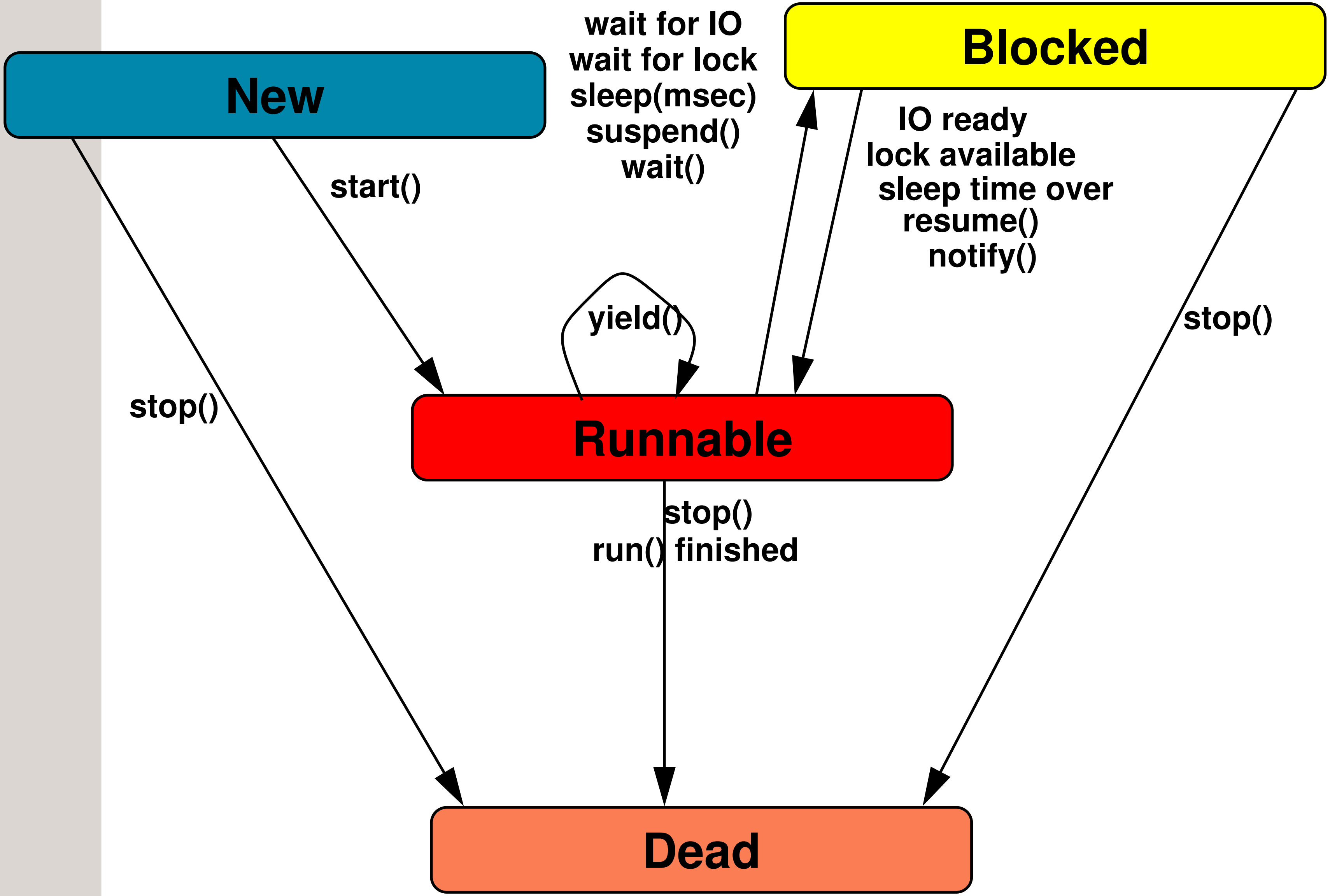
32.11 Thread States



32.11 Thread States



32.11 Thread States

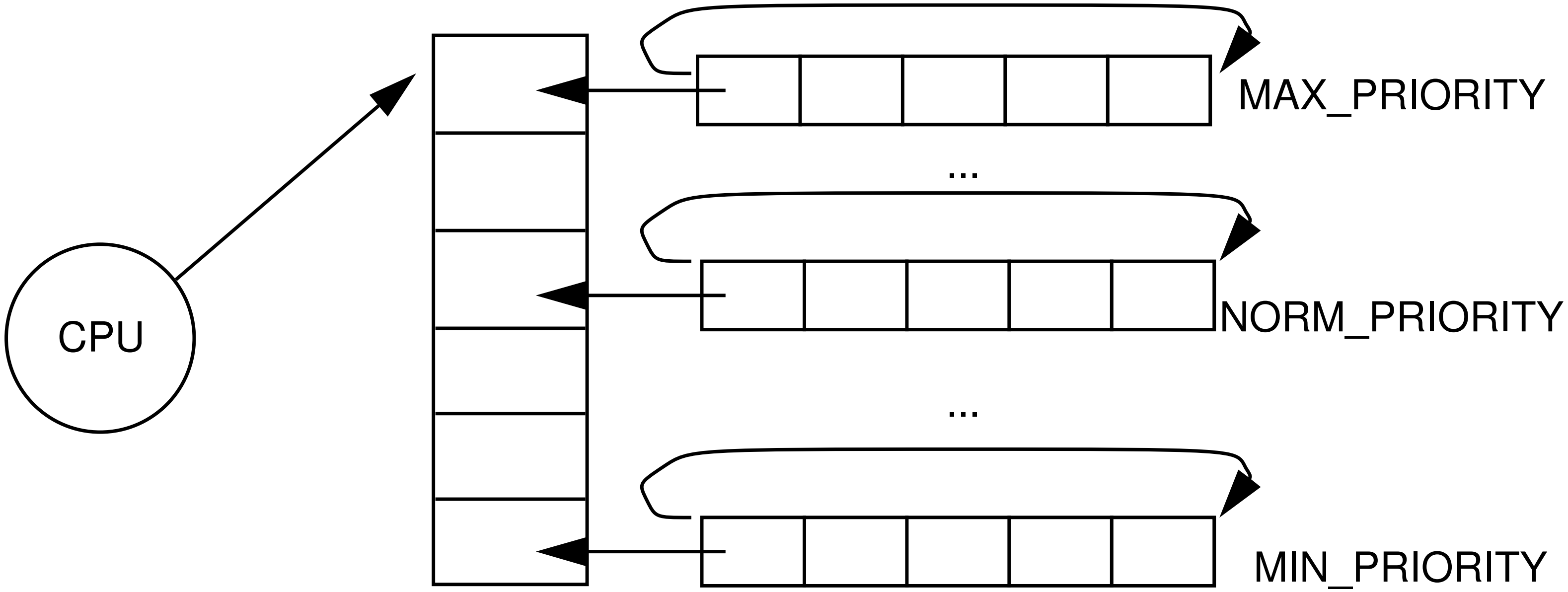


32.12 Scheduling

- default scheduling in jdk 1.1 on Solaris: preemptive, without *timeslicing*
  - a thread with higher priority preempts a thread with lower one
  - threads with equal priority are scheduled non-preemptive (FIFO)
  - if a thread blocks the next runnable thread with equal priority gets the CPU
- Java 1.1 and 1.2 on Solaris:
  - default or **`THREADS_FLAG=green`**: preemptive without timeslicing
  - **`THREADS_FLAG=native`**: preemptive with timeslicing
- Java 1.1 on WinNT/Win95
  - preemptive with timeslicing
- Correctness of a program must not depend on the scheduling strategy!!!

32.13 Priority based Round-Robin Scheduling

- Run queues of a timesliced, round-robin scheduler with static priorities



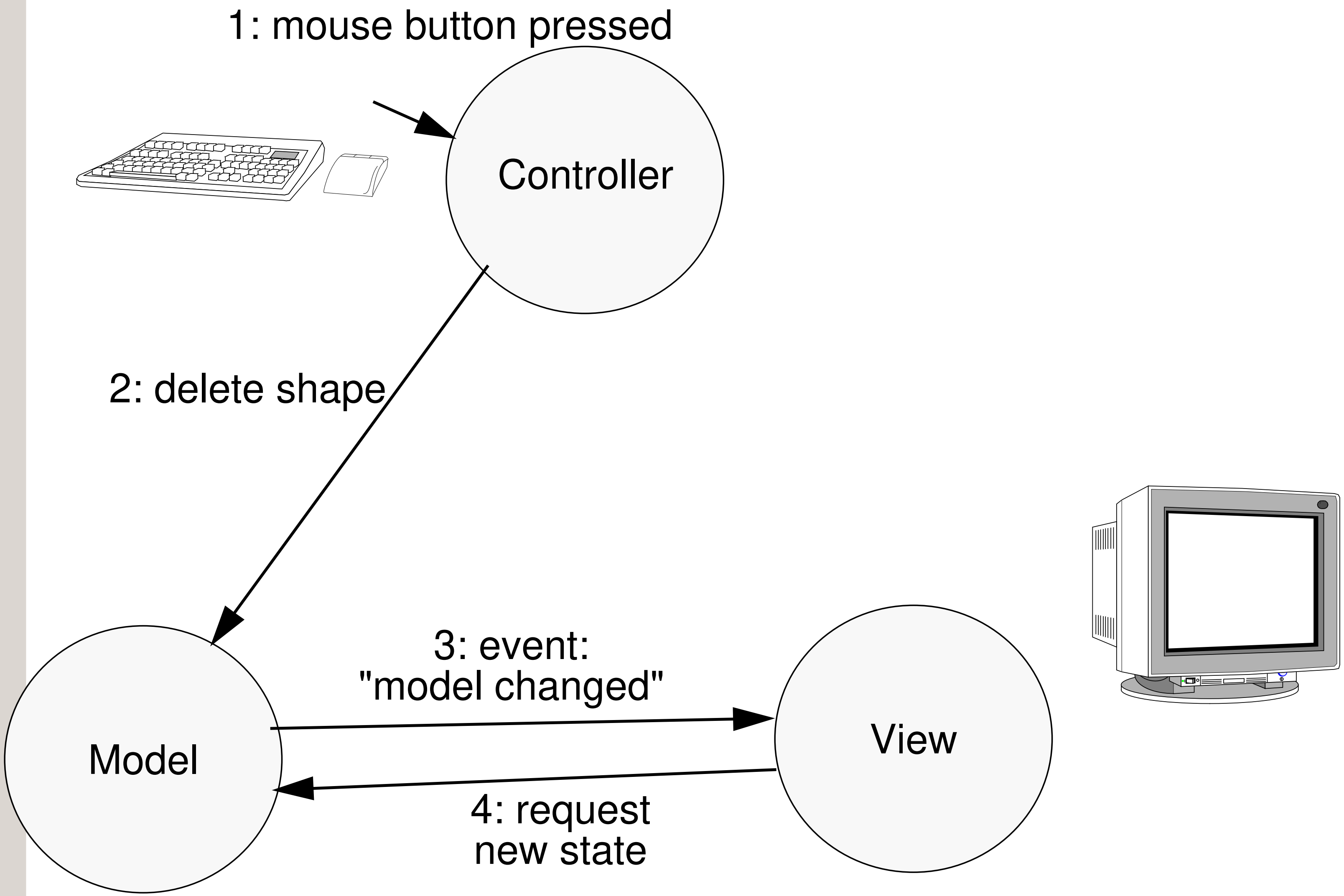
32.14 Deprecated Methods of Thread

- **`stop()`**, **`suspend()`**, **`resume()`** are deprecated in Java 1.2
- **`stop()`** releases all locks the thread holds - this is unsafe
- **`suspend()`** and **`resume()`** could lead to deadlock - stopped thread holds locks

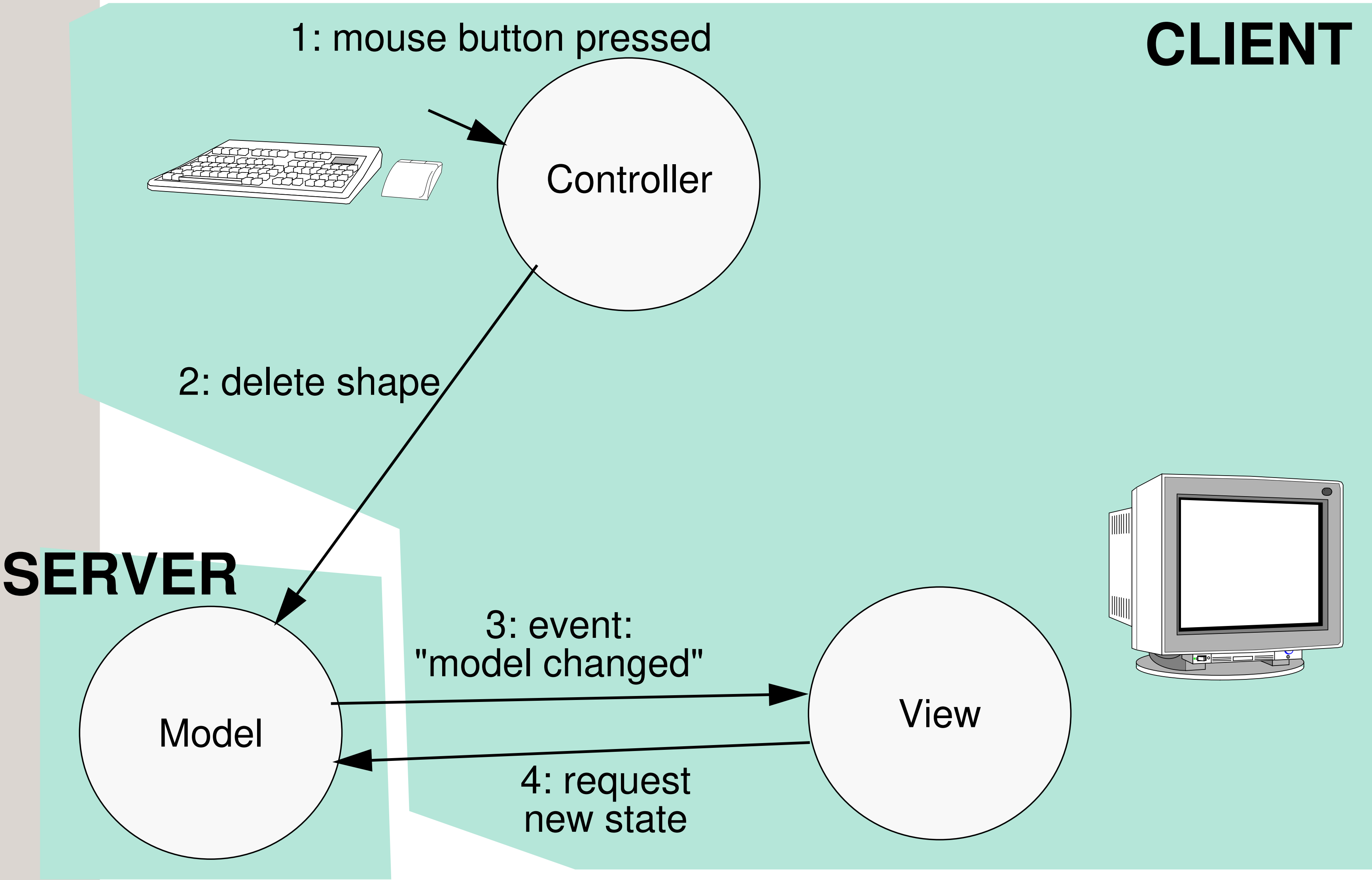
33 Design Patterns for the C/S Assignment

- "elements of reusable design"
- Examples
  - Model-View-Controller
  - Observer
  - Iterator
  - Command
  - Proxy

33.1 Model-View-Controller

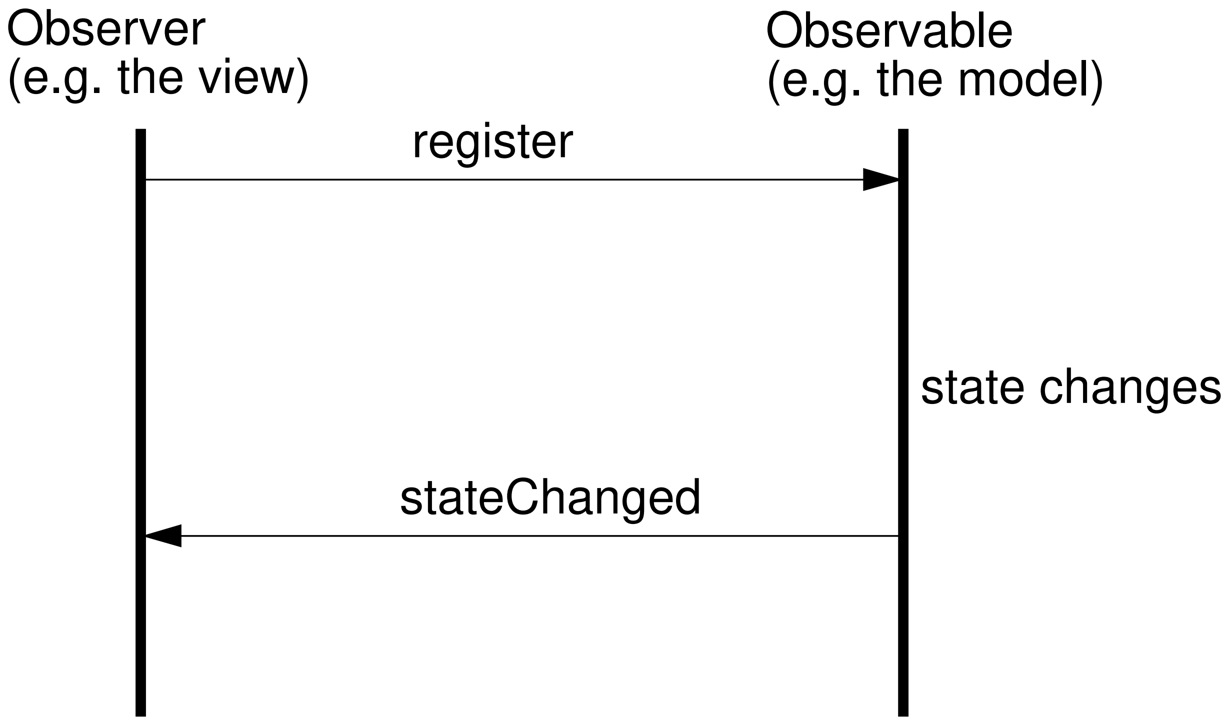


Model-View-Controller



33.2 Observer

- used in MVC by the view to observe model changes



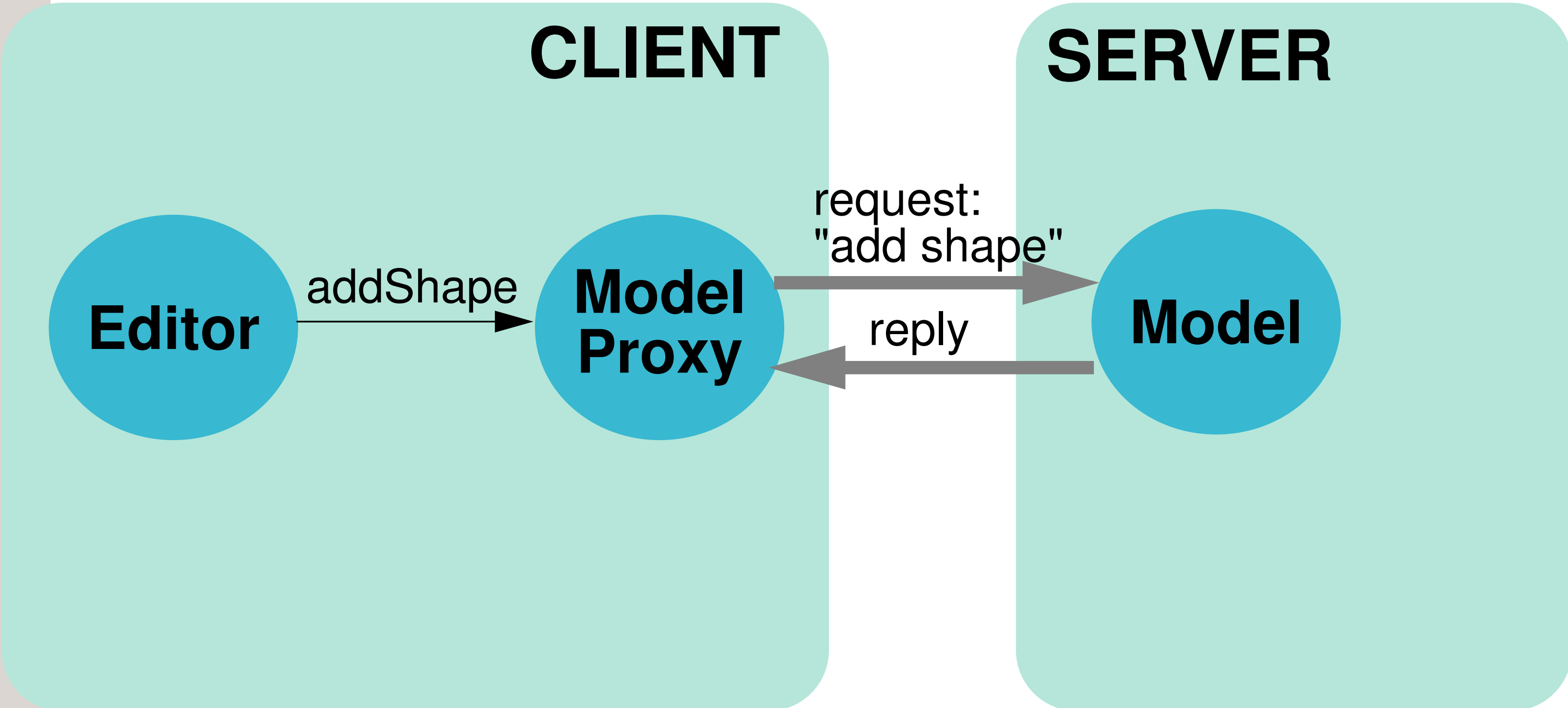
33.3 Iterator

- used to "walk through" a set of objects
- iterator is responsible for maintaining the current position

```
class Iter implements java.util.Enumeration {
  int cursor;
  Shape[] shapes;

  public Iter(Shape[] shapes) {
    this.shapes = new Shape[shapes.length];
    System.arraycopy(shapes, 0, this.shapes, 0, shapes.length);
  }

  public boolean hasMoreElements() {
   while (cursor<shapes.length && shapes[cursor]==null) cursor++;
    return cursor < shapes.length; }

  public Object nextElement() { return shapes[cursor++]; }
}
```

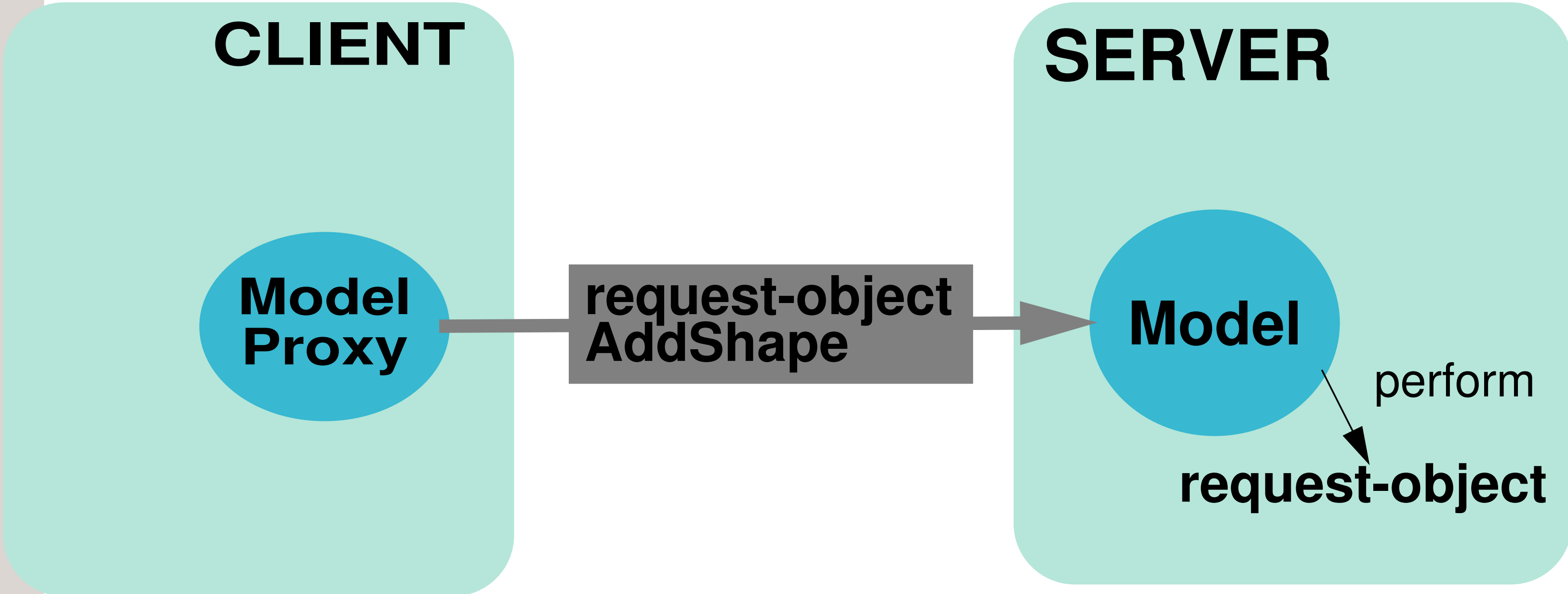
33.4 Proxy

- can be used to have a local representative of a remote object
- implements the same interface as the "real" object
- DrawingEditor with remote model:



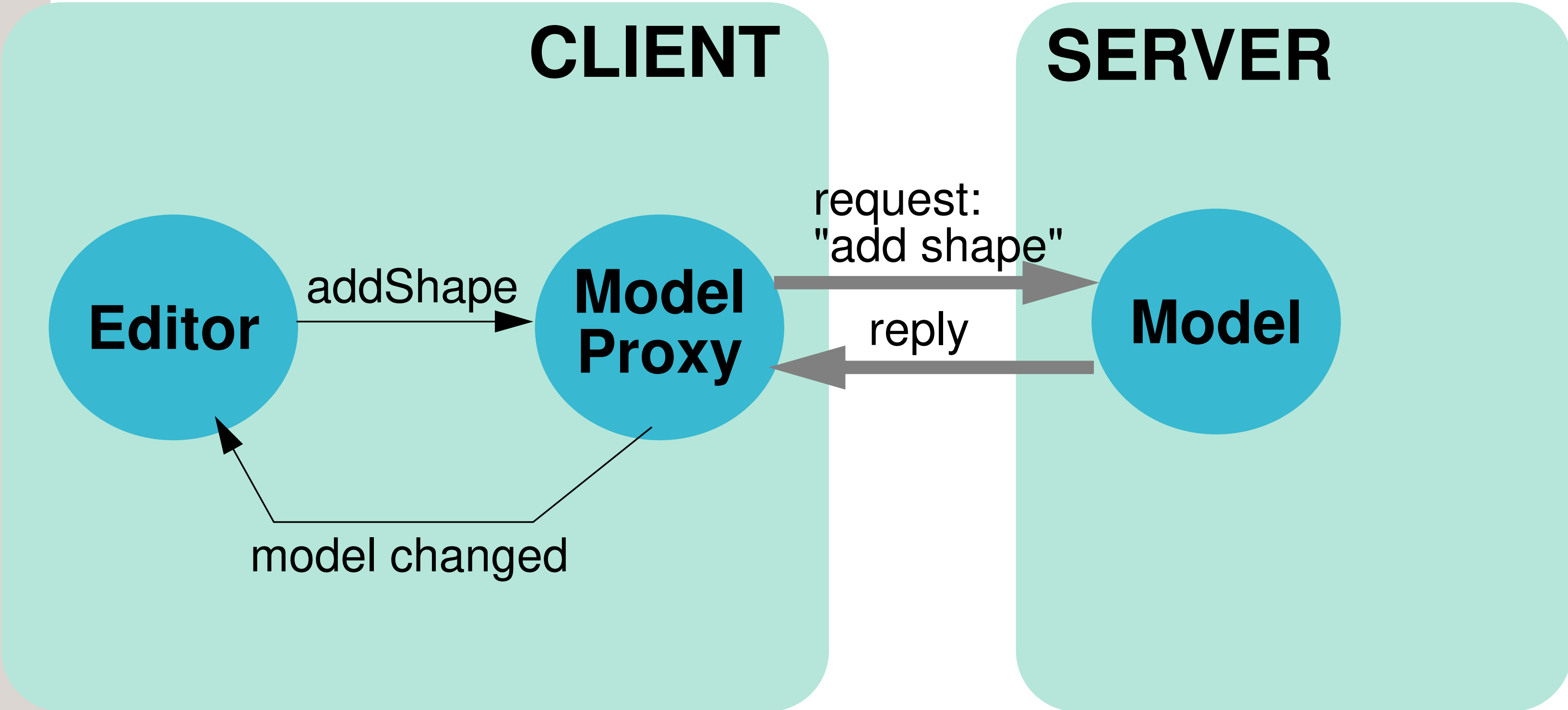
33.5 Command

- used to transfer a request to the server
- state contains information from client
- method perform() is called by server
- parameters contain information from server



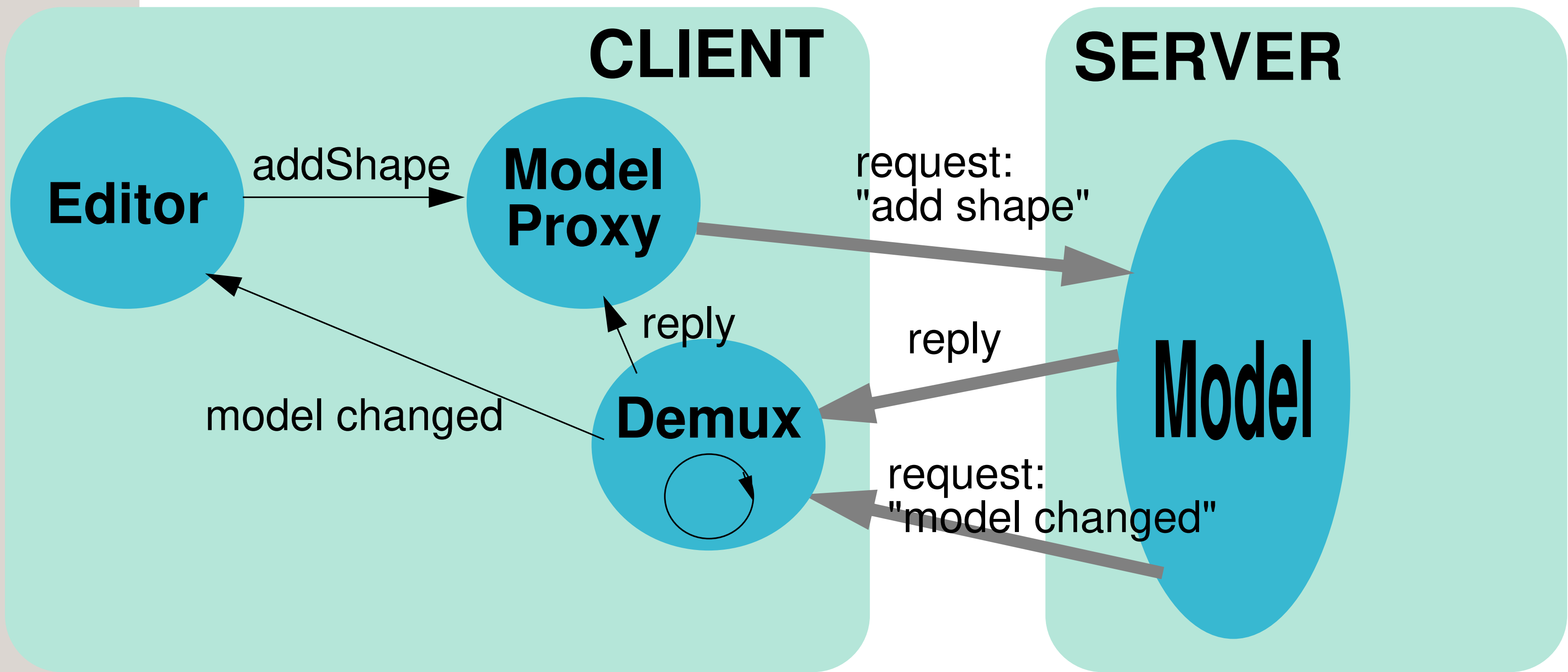
34 Repaint handling (local)

- "Quick hack": Model proxy send model changed requests



34.1 Repaint handling (Remote Observable)

- Better: "real" model sends model changed requests over the wire
- multiplex: send replies and requests on the same stream
- demultiplex the stream: separate replies from requests



35 Object identity in distributed programs

- when objects transfered through an ObjectStream they loose their identity
- you can no longer use references to check for identity
- Solution: use an object ID
  - ◆ does not change between hosts and different runs of the program