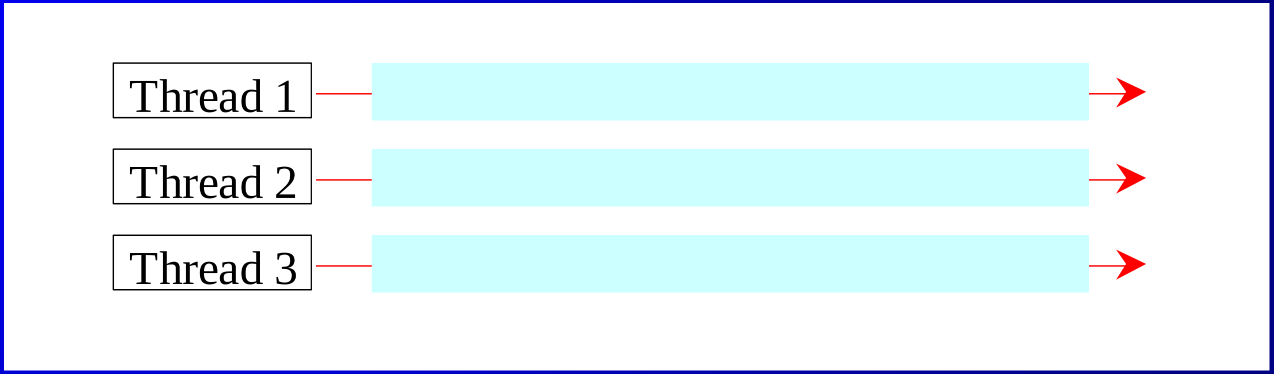
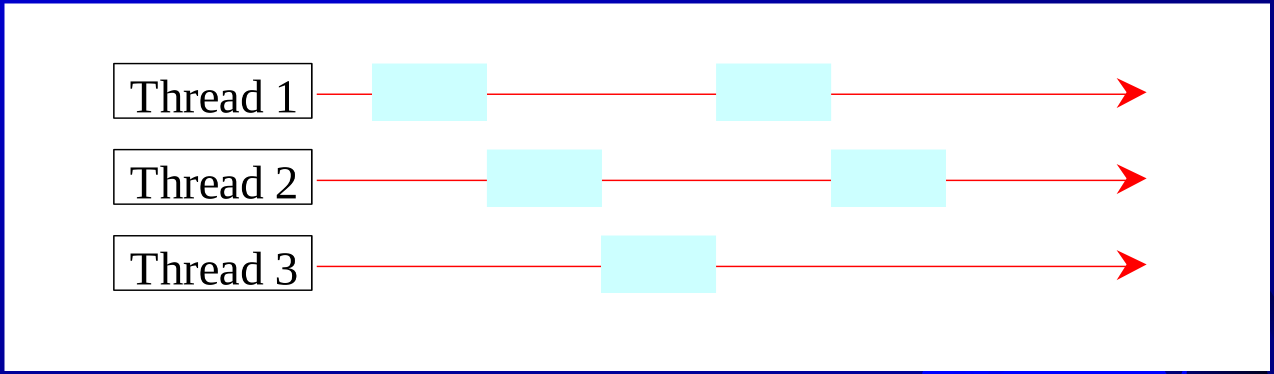
Multithreading & Networking in Java

Threads Concept

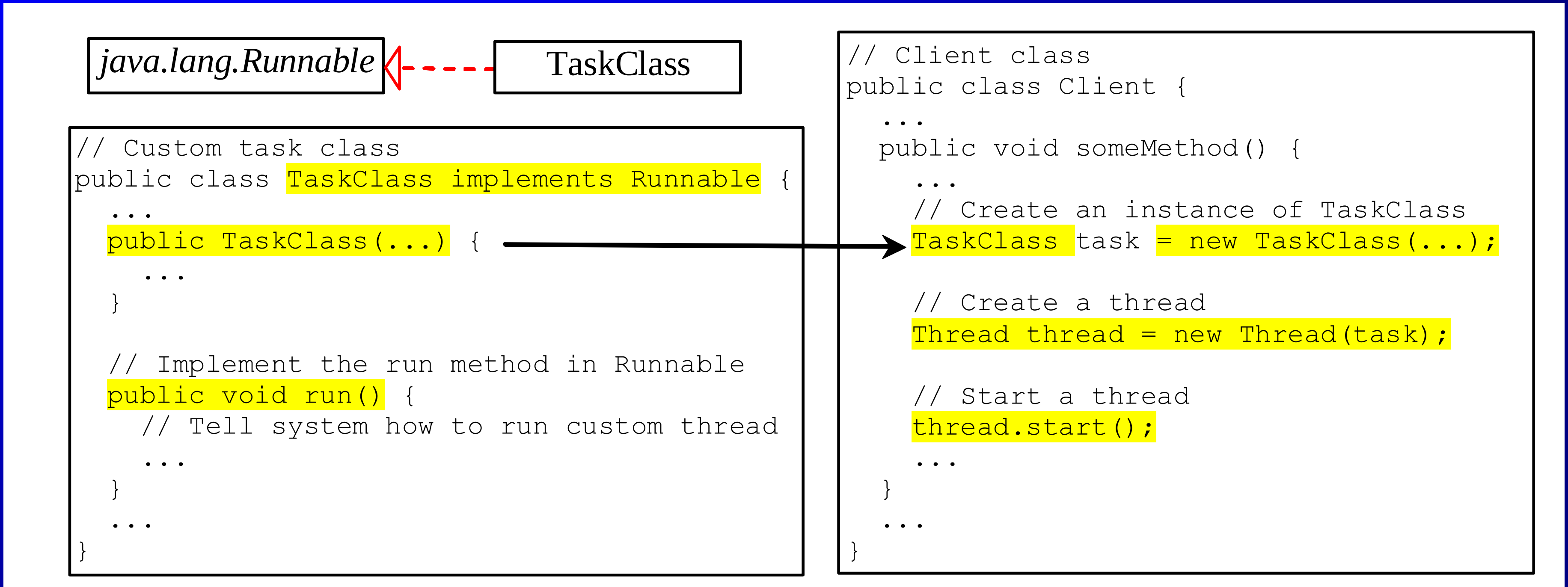
Multiple threads on multiple CPUs



Multiple threads sharing a single CPU



Creating Tasks and Threads



```
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

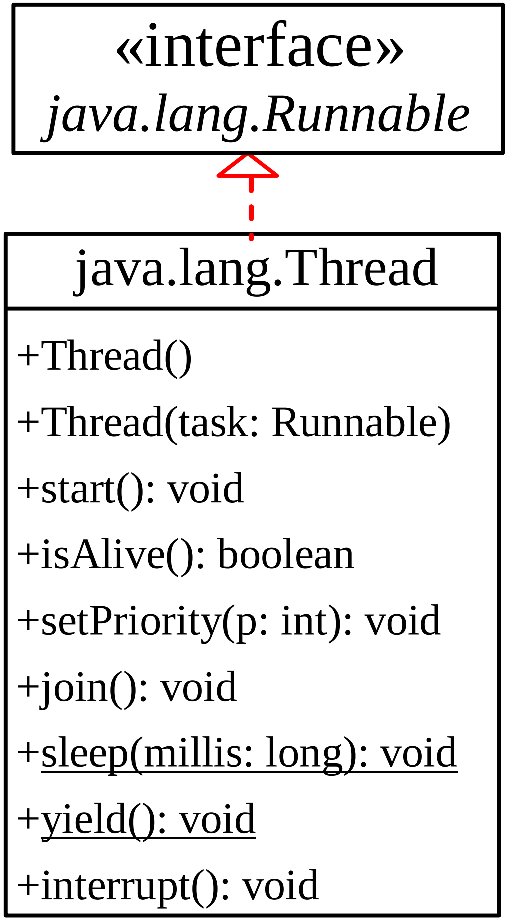
```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

Using the `Runnable` Interface to Create and Launch Threads
Example - *TaskThreadDemo*

Objective: Create and run three threads:

- – The first thread prints the letter *a* 100 times.
- – The second thread prints the letter *b* 100 times.
- – The third thread prints the integers 1 through 100.

The Thread Class



«interface»
java.lang.Runnable

java.lang.Thread

+Thread()	Creates a default thread.
+Thread(task: Runnable)	Creates a thread for a specified task.
+start(): void	Starts the thread that causes the run() method to be invoked by the JVM.
+isAlive(): boolean	Tests whether the thread is currently running.
+setPriority(p: int): void	Sets priority p (ranging from 1 to 10) for this thread.
+join(): void	Waits for this thread to finish.
+sleep(millis: long): void	Puts the runnable object to sleep for a specified time in milliseconds.
+yield(): void	Causes this thread to temporarily pause and allow other threads to execute.
+interrupt(): void	Interrupts this thread.

The Static yield() Method

You can use the yield() method to temporarily release time for other threads.

For example, suppose you modify the code in *TaskThreadDemo.java* as follows:

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    Thread.yield();
  }
}
```

Every time a number is printed, the *print100* thread is yielded. So, the numbers are printed after the characters.

The Static sleep(milliseconds) Method

The sleep(long mills) method puts the thread to sleep for the specified time in milliseconds.

For example, suppose you modify the code in *TaskThreadDemo.java* as follows:

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    try {
      if (i >= 50) Thread.sleep(1);
    }
    catch (InterruptedException ex) {
    }
  }
}
```

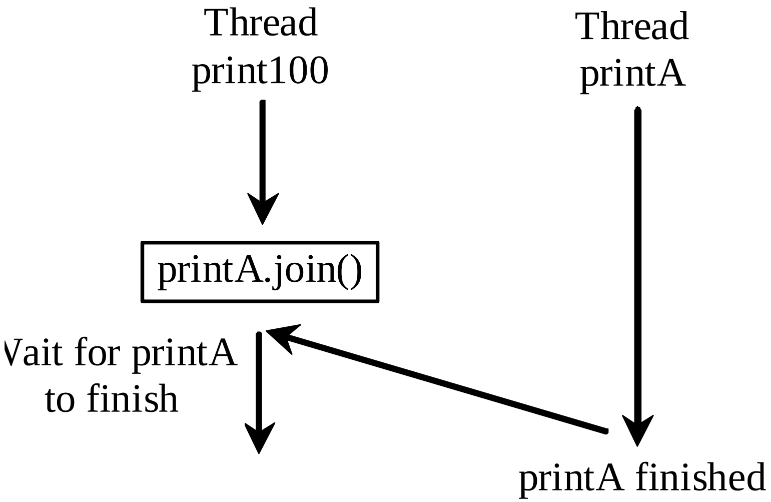
Every time a number (>= 50) is printed, the *print100* thread is put to sleep for 1 millisecond.

The join() Method

You can use the join() method to force one thread to wait for another thread to finish.

For example, suppose you modify the code in *TaskThreadDemo.java* as follows:

```java
public void run() {
  Thread thread4 = new Thread(
    new PrintChar('c', 40));
  thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {
  }
}
```



Thread
print100

Thread
printA

printA.join()

Vait for printA
to finish

printA finished

The numbers after 50 are printed after thread printA is finished.

isAlive(), interrupt(), and isInterrupted()

☞ The isAlive() method is used to find out the state of a thread.

- it returns true if a thread is in the Ready, Blocked, or Running state;

- it returns false if a thread is new and has not started or if it is finished.

☞ The interrupt() method interrupts a thread in the following way:

- if a thread is currently in the Ready or Running state, its interrupted flag is set;

- if a thread is currently blocked, it is awakened and enters the Ready state, and an java.io.InterruptedException is thrown.

☞ The isInterrupt() method tests whether the thread is interrupted.

The deprecated stop(), suspend(), and resume() Methods

The Thread class also contains the stop(), suspend(), and resume() methods.

As of Java 2, these methods are *deprecated* (or *outdated*) because they are known to be inherently unsafe.

You should assign null to a Thread variable to indicate that it is stopped rather than use the stop() method.

Thread Priority

- Each thread is assigned a default priority of `Thread.NORM_PRIORITY`. You can reset the priority using `setPriority(int priority)`.

- Some constants for priorities include
  `Thread.MIN_PRIORITY`
  `Thread.MAX_PRIORITY`
  `Thread.NORM_PRIORITY`

Thread Synchronization

A shared resource may be corrupted if it is accessed simultaneously by multiple threads.

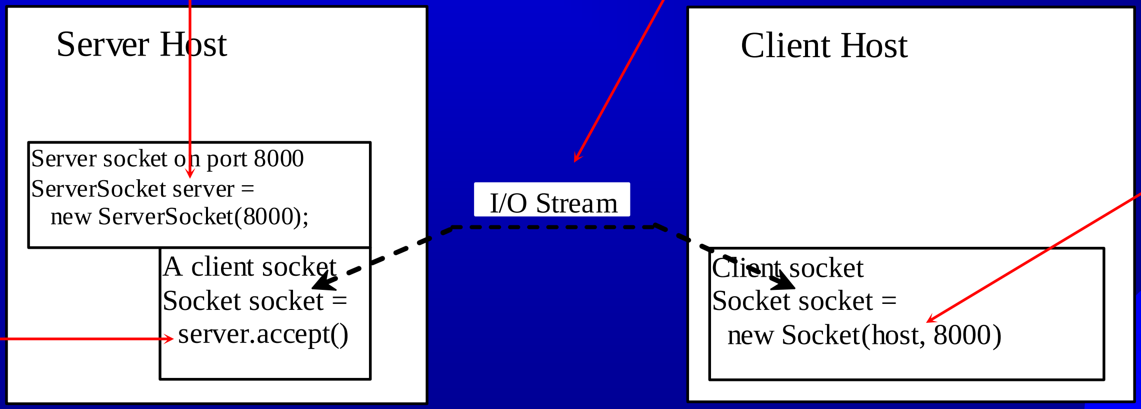
For example, two unsynchronized threads accessing the same bank account may cause conflict.

Step	balance	thread[i]	thread[j]
1	0	newBalance = bank.getBalance() + 1;	
2	0		newBalance = bank.getBalance() + 1;
3	1	bank.setBalance(newBalance);	
4	1		bank.setBalance(newBalance);

Client/Server Communications

The server must be running when a client starts. The server waits for a connection request from a client. To establish a server, you need to create a server socket and attach it to a port, which is where the server listens for connections.

After the server accepts the connection, communication between server and client is conducted the same as for I/O streams.



Server Host

Server socket on port 8000

```
ServerSocket server =
  new ServerSocket(8000);
```

A client socket

```
Socket socket =
  server.accept()
```

I/O Stream

Client Host

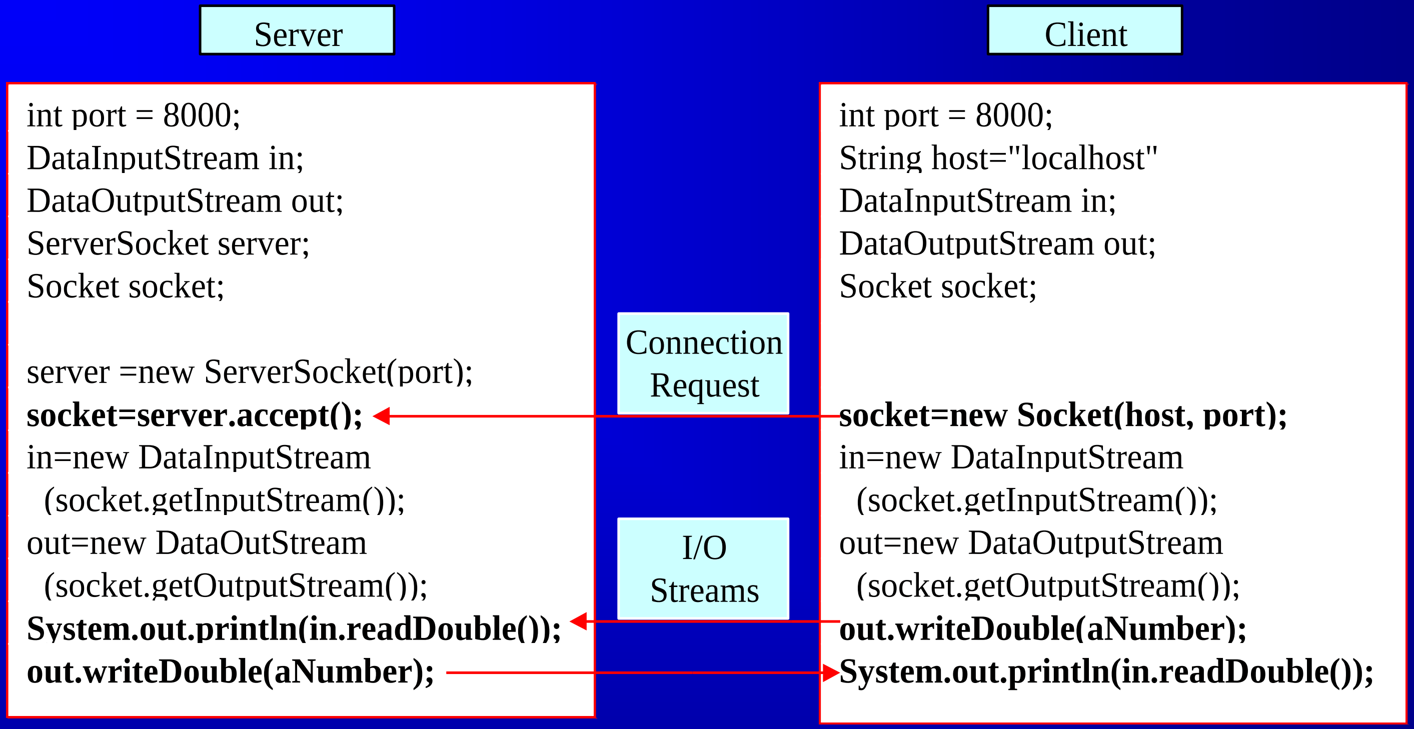
Client socket

```
Socket socket =
  new Socket(host, 8000)
```

After a server socket is created, the server can use this statement to listen for connections.

The client issues this statement to request a connection to a server.

Data Transmission through Sockets



Server

```
int port = 8000;
DataInputStream in;
DataOutputStream out;
ServerSocket server;
Socket socket;

server =new ServerSocket(port);
socket=server.accept();
in=new DataInputStream
  (socket.getInputStream());
out=new DataOutStream
  (socket.getOutputStream());
System.out.println(in.readDouble());
out.writeDouble(aNumber);
```

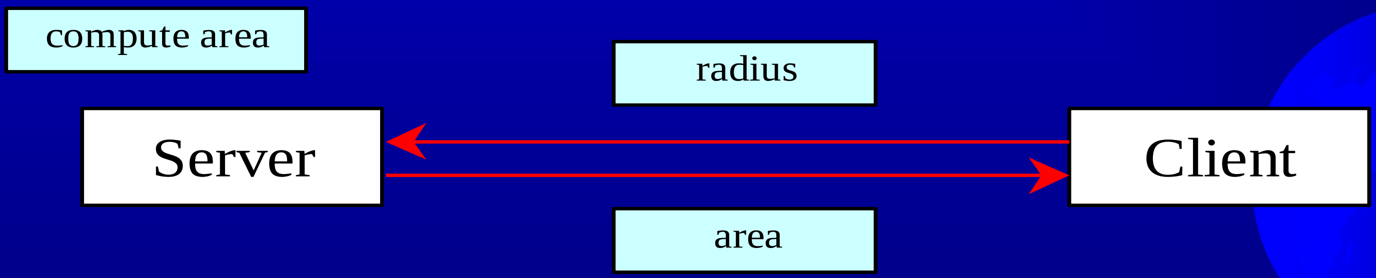
Client

```
int port = 8000;
String host="localhost"
DataInputStream in;
DataOutputStream out;
Socket socket;

socket=new Socket(host, port);
in=new DataInputStream
  (socket.getInputStream());
out=new DataOutputStream
  (socket.getOutputStream());
out.writeDouble(aNumber);
System.out.println(in.readDouble());
```

```
InputStream input = socket.getInputStream();
OutputStream output = socket.getOutputStream();
```

A Client/Server Example

- Write a client to send data to a server.
- The server receives the data, uses it to produce a result, and then sends the result back to the client.
- The client displays the result on the console.
- In this example, the data sent from the client is the radius of a circle, and the result produced by the server is the area of the circle.



compute area

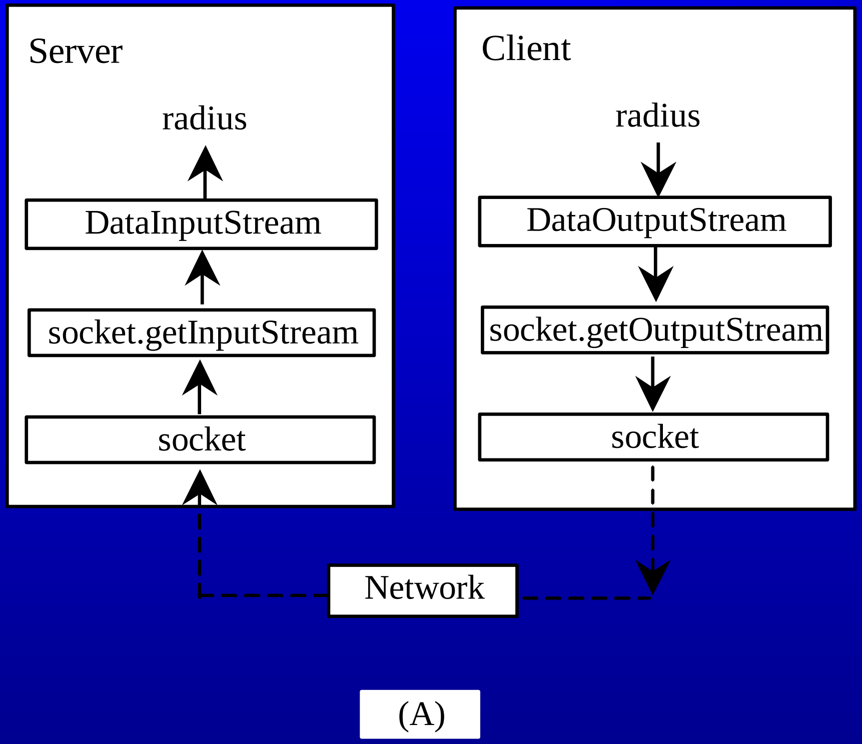
radius

Server

Client

area

A Client/Server Example, cont.



Server

radius

DataInputStream

socket.getInputStream

socket

Client

radius

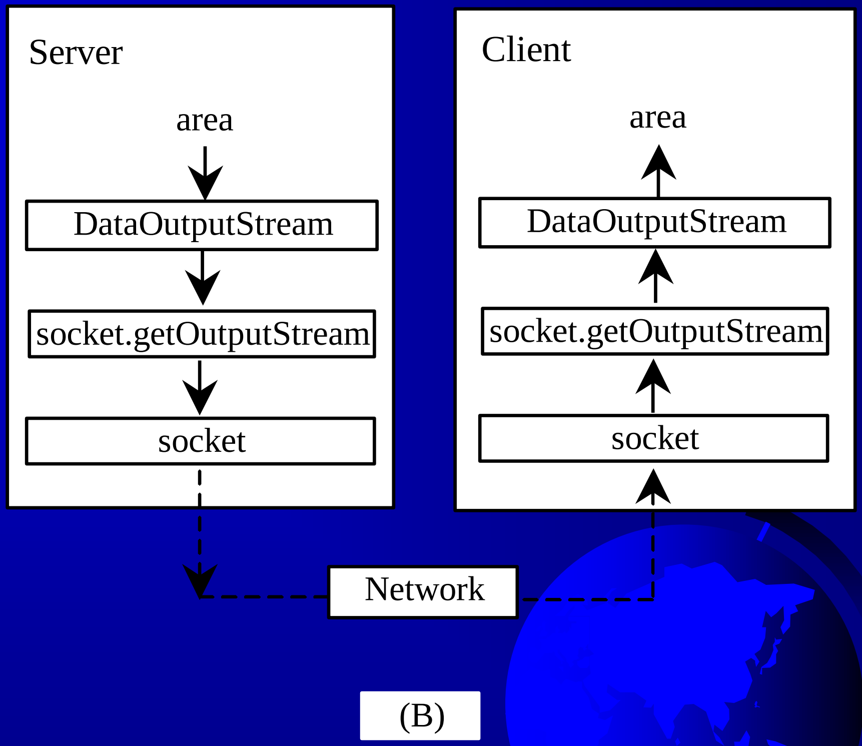
DataOutputStream

socket.getOutputStream

socket

Network

(A)



Server

area

DataOutputStream

socket.getOutputStream

socket

Client

area

DataOutputStream

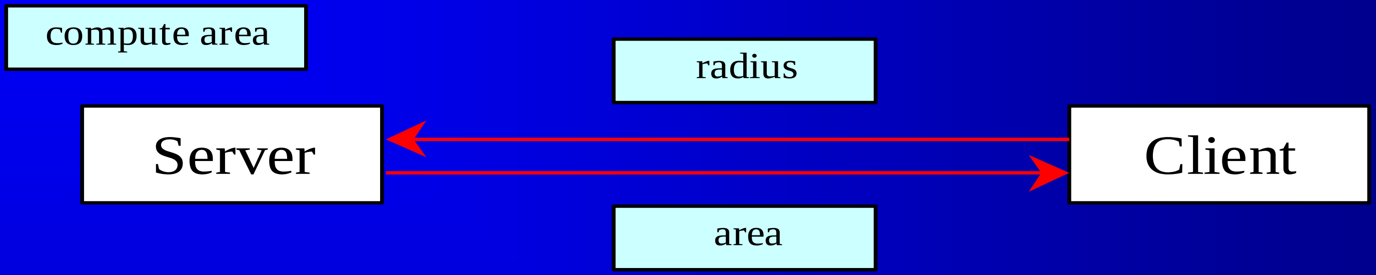
socket.getOutputStream

socket

Network

(B)

A Client/Server Example, cont. (*Server* and *Client*)



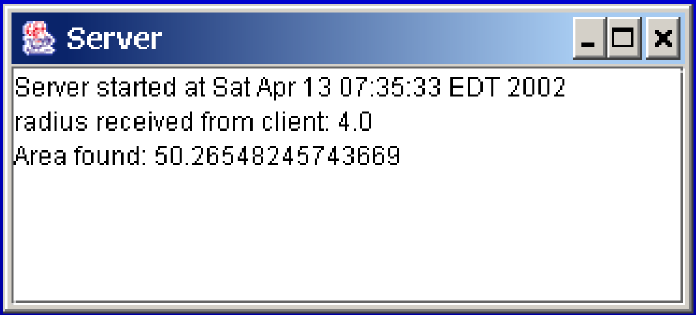
compute area

radius

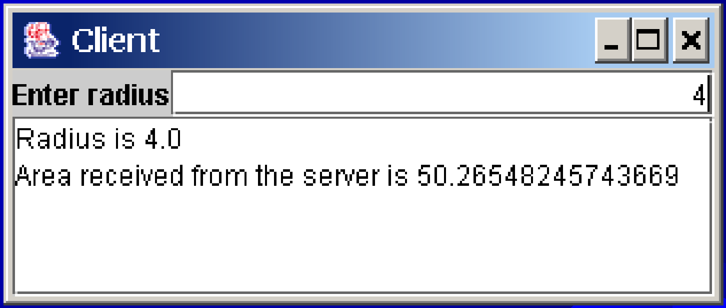
Server

Client

area



```
Server started at Sat Apr 13 07:35:33 EDT 2002
radius received from client: 4.0
Area found: 50.26548245743669
```



Enter radius 4

```
Radius is 4.0
Area received from the server is 50.26548245743669
```

Note: Start the server, then the client.

The InetAddress Class

Occasionally, you would like to know who is connecting to the server. You can use the InetAddress class to find the client's host name and IP address.

The InetAddress class models an IP address. You can use the statement shown below to create an instance of InetAddress for the client on a socket.

```
InetAddress inetAddress = socket.getInetAddress();
```

Next, you can display the client's host name and IP address, as follows:

```
System.out.println("Client's host name is " +
                  inetAddress.getHostName());

System.out.println("Client's IP Address is " +
                  inetAddress.getHostAddress());
```

Serving Multiple Clients

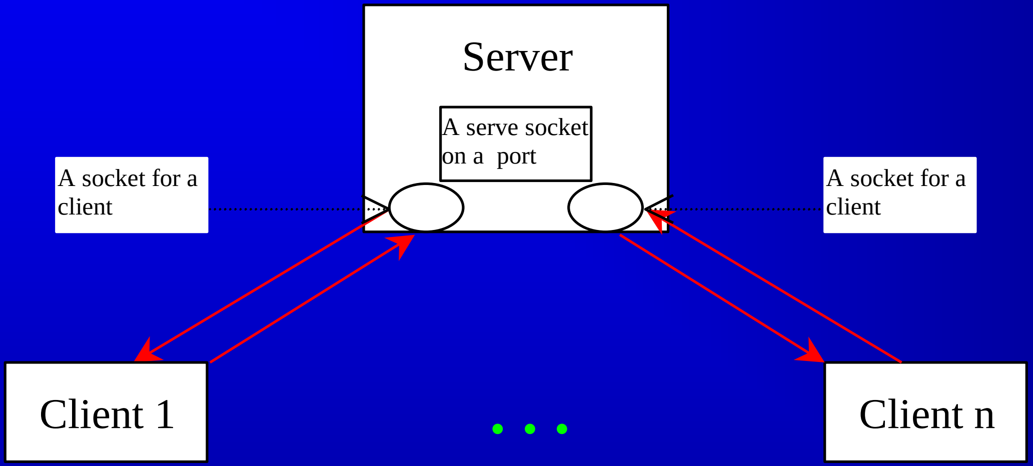
- Multiple clients are quite often connected to a single server at the same time.
- Typically, a server runs constantly on a server computer, and clients from all over the Internet may want to connect to it.
- You can use threads to handle the server's multiple clients simultaneously. Simply create a thread for each connection.

Here is how the server handles the establishment of a connection:

```
while (true) {
  Socket socket = serverSocket.accept();
  Thread thread = new ThreadClass(socket);
  thread.start();
}
```

- The server socket can have many connections.
- Each iteration of the while loop creates a new connection.
- Whenever a connection is established, a new thread is created to handle communication between the server and the new client; and this allows multiple connections to run at the same time.

Serving Multiple Clients (Example – *MultiThreadServer, Client*)

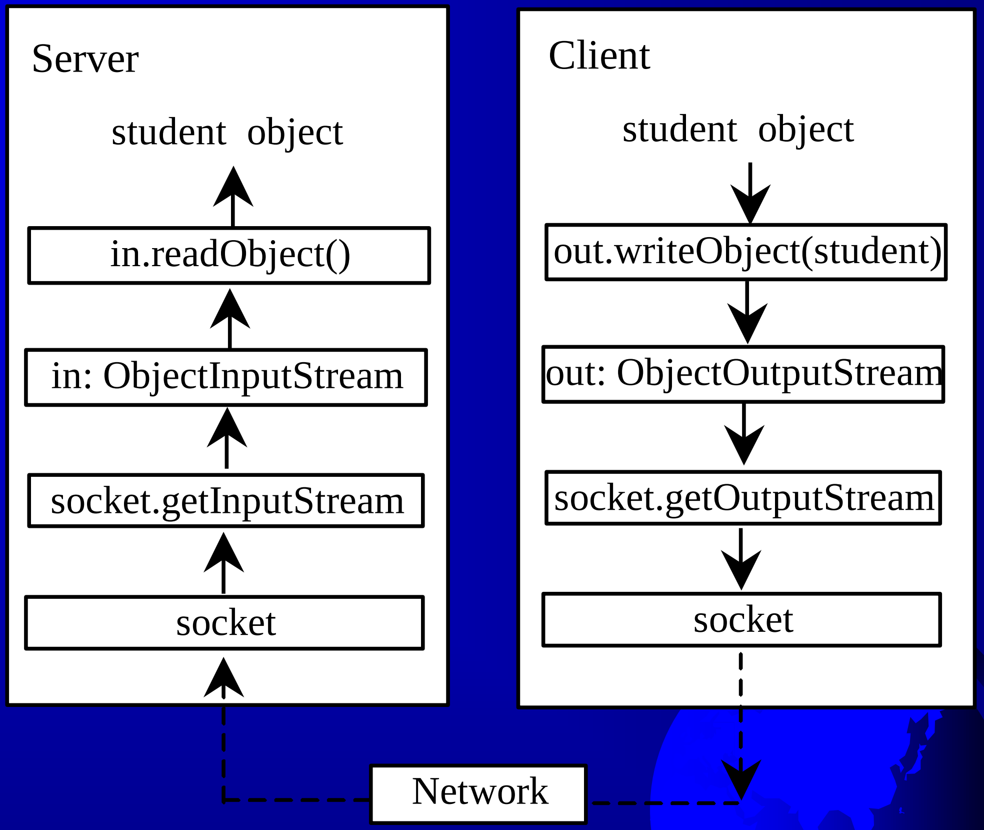


Note: Start the server first, then start multiple clients.

Passing Objects in Network Programs (Example – *Student*, *StudentServer*, *StudentClient*)

☞ Write a program that collects student information from a client and send them to a server.

☞ Passing student information in an object.



Server

student object

in.readObject()

in: ObjectInputStream

socket.getInputStream

socket

Client

student object

out.writeObject(student)

out: ObjectOutputStream

socket.getOutputStream

socket

Network

Note: Start the server first, then the client.