Chapter 11.

Networking and Multi-threading

2020-2021
COMP2396 Object-Oriented Programming and Java
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong

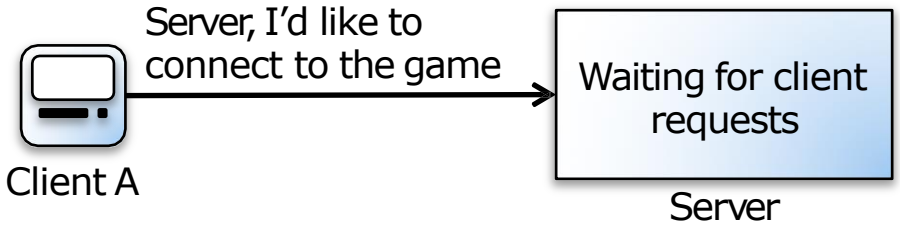
Connecting to Another Program

- Ocean would like to extend his fantasy adventure game to a multi-player online game
- He adopted the client-server model where each player will launch a client on his own machine that will be connected to a server over the network
- A client can then communicate with other clients by sending and receiving messages to and from the server
- In Java, sending and receiving data over the network is just I/O with a slightly different connection stream
- All the low-level networking details are taken care of by classes in the java.net package

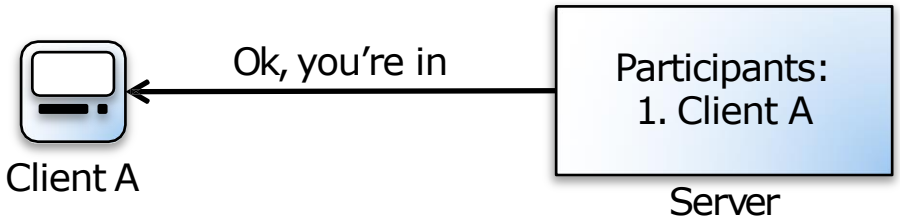
Client-Server Model

How it works:

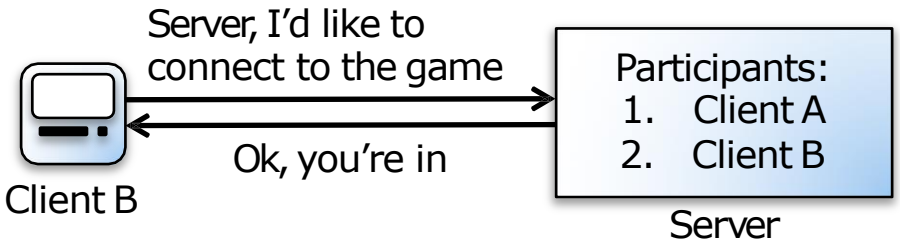
1. A client connects to the server



2. The server makes a connection and adds the client to the list of participants



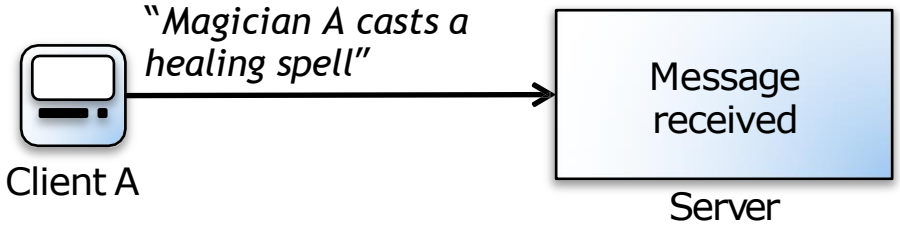
3. Another client connects



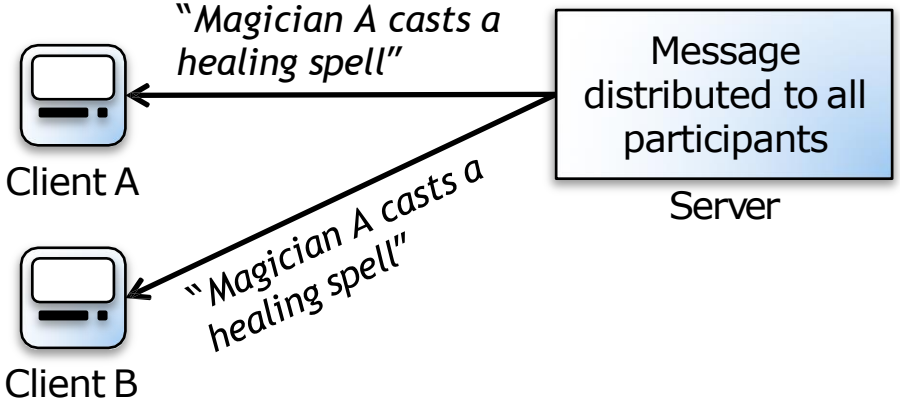
Client-Server Model

How it works:

4. Client A sends a message to the game server

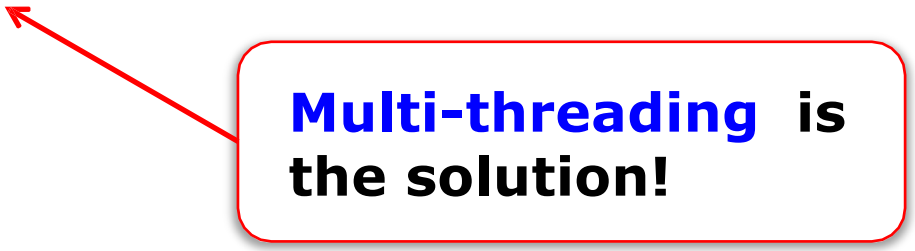


5. The server distributes the message to ALL participants (including the original sender)



Connecting, Sending, and Receiving

- In order to get his game client working, Ocean needs to learn
  1. How to establish the initial connection between the client and server
  2. How to send messages to the server
  3. How to receive messages from the server
  4. How to send outgoing messages to and simultaneously receive incoming messages from other participants via the server



Socket Connection

- Client and server applications communicate over a Socket connection
- A Socket (java.net.Socket class) is an object that represents a connection between 2 applications which may (or may not) be running on 2 different physical machines
- To create a Socket connection, a client must know the IP address and TCP port number of the server application

```
Socket socket = new Socket("192.168.1.103", 5000);
```

IP address for the server

TCP port number

TCP Port

- A TCP port is a 16-bit unsigned number assigned to a specific server application
- TCP port numbers allow different clients to connect to the same machine but communicate with different applications running on that machine
- TCP port numbers from 0 to 1023 are reserved for well known services (e.g., 80 for HTTP, 23 for Telnet, 20 for FTP, 25 for SMTP, and 110 for POP3 mail server, etc.)
- When writing a server program, you might use any TCP port number between 1024 to 65535
- Only 1 program can be running on a single TCP port. If you try to bind a program to a port that is already in use, you will get a java.net.BindException

Reading Data from a Socket

- The 4 simple steps in reading data from a socket

1. Make a Socket connection to the server

```
Socket sock = new Socket("127.0.0.1", 5000);
```

127.0.0.1 is the IP address for "localhost" (i.e., the one this code is running on)

2. Make an InputStreamReader

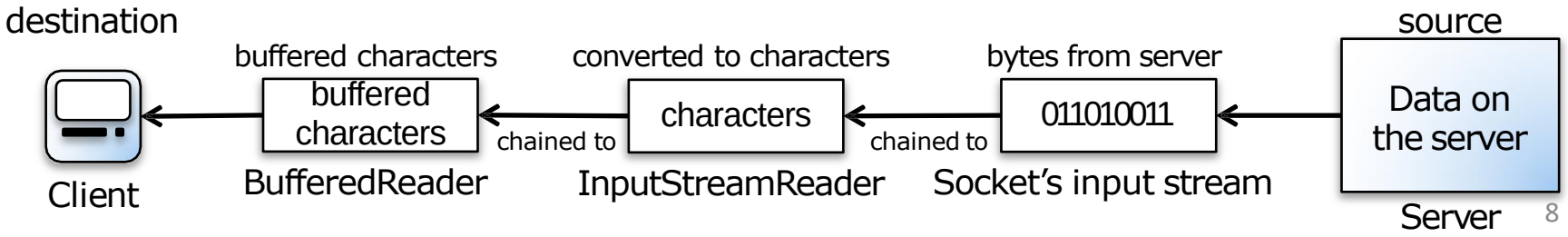
```
InputStreamReader streamReader = new InputStreamReader(sock.getInputStream());
```

3. Make a BufferedReader

```
BufferedReader reader = new BufferedReader(streamReader);
```

4. Read data

```
String line = reader.readLine();
```



Writing Data to a Socket

- The 3 simple steps in writing data to a socket

1. Make a Socket connection to the server

```
Socket sock = new Socket("127.0.0.1", 5000);
```

2. Make a PrintWriter

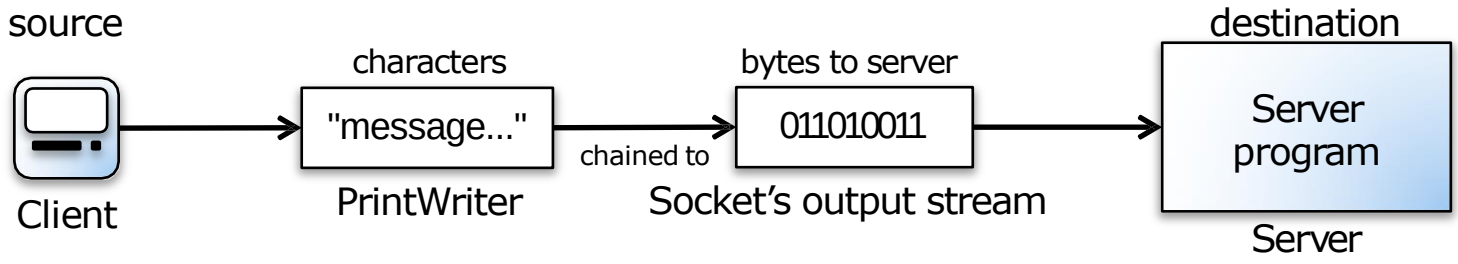
```
PrintWriter writer = new PrintWriter(sock.getOutputStream());
```

3. Write data

```
writer.println("message to send");
writer.print("another message");
```

println() adds a new line at the end of what it sends

print() doesn't adds the newline



Example: Daily Advice Client

Example

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

public class DailyAdviceClient {
   Socket sock;

   public void go() {
      try {
         sock = new Socket("127.0.0.1", 5000);
         InputStreamReader streamReader =
                     new InputStreamReader(sock.getInputStream());
         BufferedReader reader = new BufferedReader(streamReader);
         String advice = reader.readLine();
         System.out.println("Today's advice: "  + advice);
         reader.close();
      }  catch (Exception ex) {  ex.printStackTrace(); }
   }
   //  code for main()...
}
```

I/O operations can throw exceptions

Make a Socket connection

Writing a Simple Server

How it works:

1. A server application creates a ServerSocket on a specific port

```
ServerSocket serverSock = new ServerSocket(5000);
```

2. The server application waits for a new client

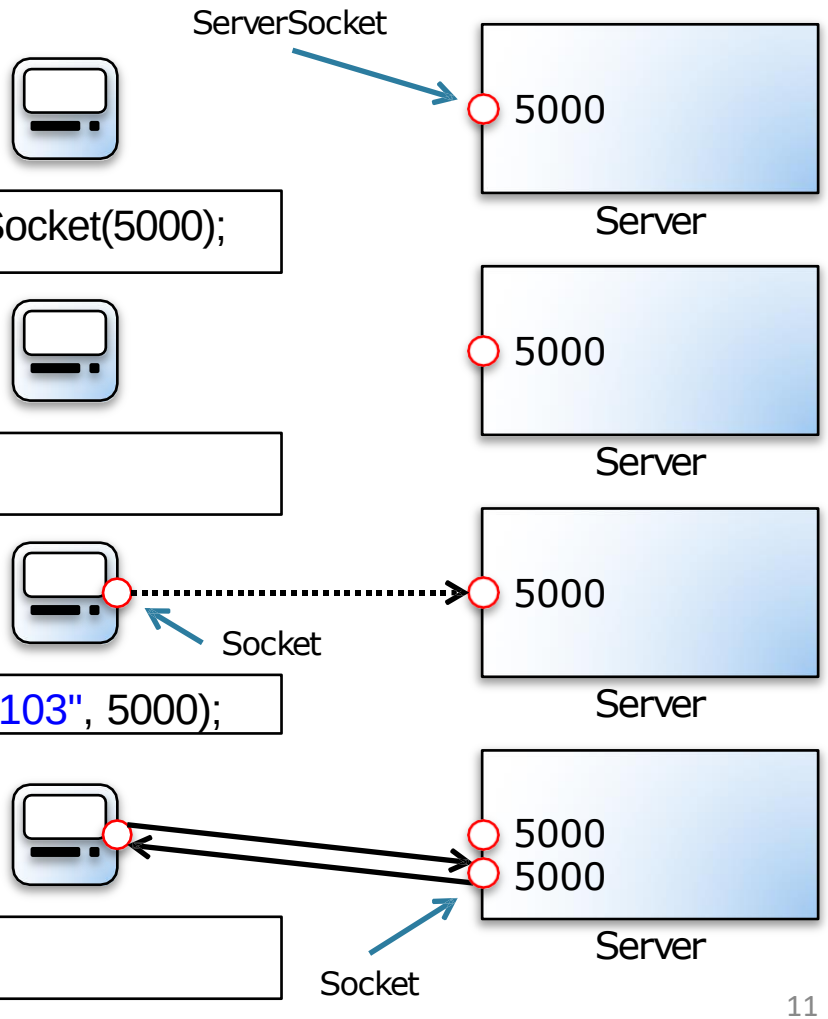
```
Socket sock = serverSock.accept();
```

3. A client makes a Socket connection to the server application

```
Socket sock = new Socket("192.168.1.103", 5000);
```

4. The server creates a new Socket to communicate with this client

```
Socket sock = serverSock.accept();
```



Writing a Simple Server

- The accept() method of a ServerSocket blocks (just sits there) while it is waiting for a client Socket connection
- When a client finally tries to connect, the method returns a plain Socket on the same port
- This Socket knows how to communicate with the client (i.e., it knows the client's IP address and TCP port number)

Example: Daily Advice Server

Example

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

public class DailyAdviceServer {
   String[] adviceList = {"Practice makes perfect", "Never give up",
         "Focus on the task at hand", "Don't look back", "Be yourself",
         "Believe in your own work"};
   ServerSocket serverSock;

   public String getAdvice() {
      int random = (int) (Math.random() * adviceList.length);
      return adviceList[random];
   }

   public static void main(String[] args) {
      DailyAdviceServer server = new DailyAdviceServer();
      server.go();
   }
```

Example: Daily Advice Server

Example

```
public void go() {
  try {
    serverSock = new ServerSocket(5000);

    while (true) {
      Socket sock = serverSock.accept();
      PrintWriter writer = new PrintWriter(sock.getOutputStream());
      String advice = getAdvice();
      writer.println(advice);
      writer.close();
      System.out.println(advice);
    }
  } catch (Exception ex) {
    ex.printStackTrace();
  }
} // close go
}
```

This server application listens for client requests on port 5000 on the machine this code is running on

The server goes into an infinite loop, waiting for and serving client requests

Multi-threading

- The daily advice server code in the previous example has a serious limitation that it can only handle 1 client at a time
- It cannot accept a request from another client until it has finished with the current client and started the next iteration of the infinite loop
- In order to make the server capable of handling multiple clients concurrently, separate threads are needed and each new client Socket should be assigned to a new thread
- Similarly, if a client wants to send and receive messages to and from the server simultaneously, a separate thread should be created for receiving messages from the server

Multi-threading

- A thread can be considered as a line of execution. It has its own call stack for storing method invocations and local variables
- Every application has at least 1 thread running when it is started (i.e., the main thread)
- The main thread can create additional threads (i.e., worker threads) to handle different tasks in parallel (e.g., reading from a file, printing to a printer)
- Multi-threading refers to the 'concurrent' execution of multiple threads in a single application
- Multi-threading often makes an application more responsive by moving long-running tasks to worker threads

Multi-threading in Java

- Java has multi-threading built right into the fabric of the language
- A Thread (java.lang.Thread class) is an object that represents a thread of execution
- The 3 simple steps in launching a new thread
  1. Make a Runnable object (the thread's job)

```
Runnable threadJob = new MyRunnable();
```

  2. Make a Thread object and give it a Runnable (the job)

```
Thread myThread = new Thread(threadJob);
```

  3. Start the Thread

```
myThread.start();
```

Runnable Interface

- A Runnable object is to a Thread object what a job is to a worker. It is an instance of a class that implements the Runnable interface
- The Runnable interface defines only 1 method

```
public interface Runnable {
   void run();
}
```

- By passing a Runnable object to the Thread constructor, it tells the new Thread object which job to run
- When the Thread object's `start()` method is called, a new thread of execution starts and the Runnable object's `run()` method is put on the bottom of the new thread's stack

A Multi-threading Example

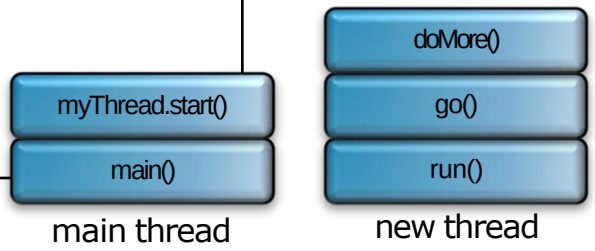
Example

```
public class MyRunnable implements Runnable {
   public void run() { go(); }
   public void go() { doMore(); }

   public void doMore() {
      System.out.println("top of the stack");
   }

   public static void main(String[] args) {
      Runnable threadJob = new MyRunnable();
      Thread myThread = new Thread(threadJob);
      myThread.start();
      System.out.println("back in main");
   }
}
```

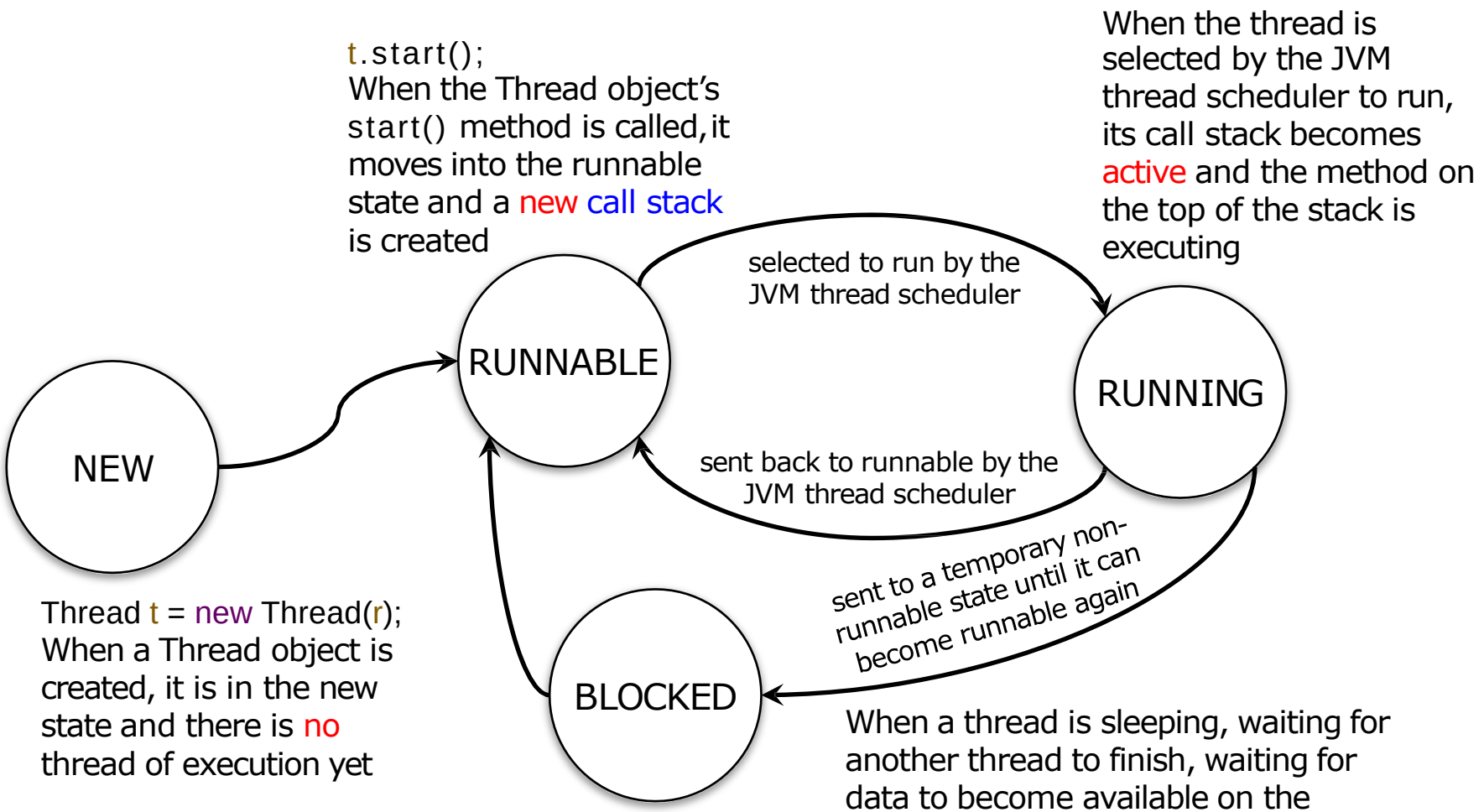
What will be the output?

A
top of the stack
back in main

B
back in main
top of the stack



States of a Thread

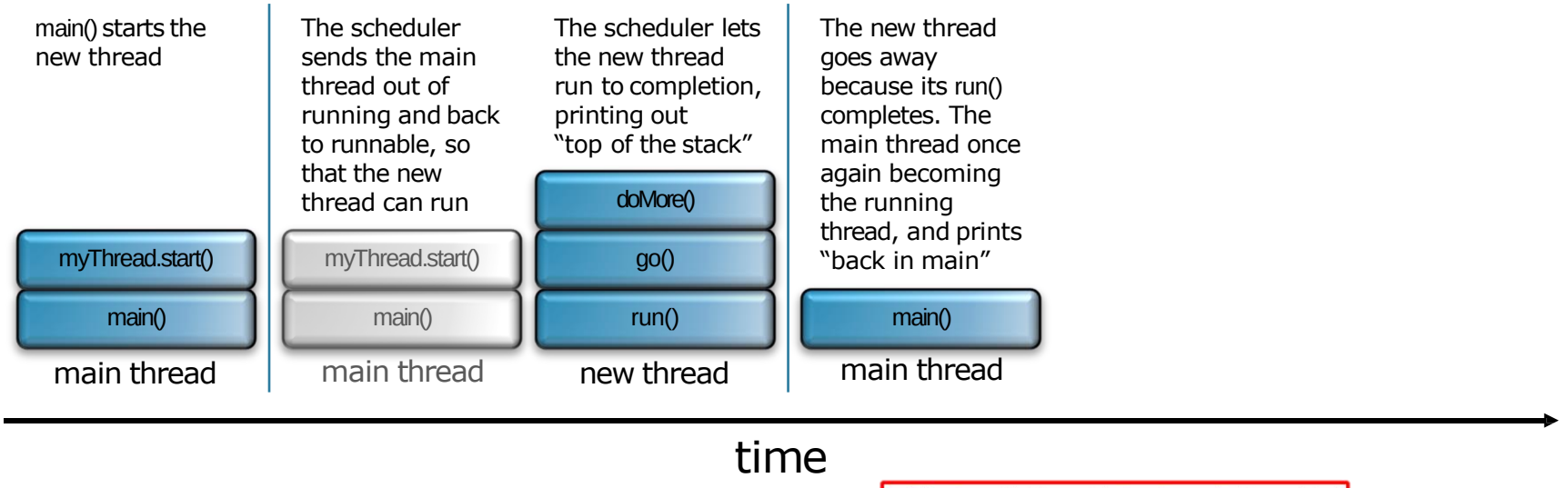


Thread Scheduler

- The thread scheduler makes all the decisions about
  - Which thread moves from runnable to running state
  - When and under what circumstances a thread leaves the running state
  - Where a thread goes when it is kicked out of the running state
- There is no API for calling methods on the scheduler (i.e., no way to control the scheduler)
- There are no guarantees about scheduling!
- Never base your program's correctness on the scheduler working in a particular way!

A Multi-threading Example

Scenario 1:



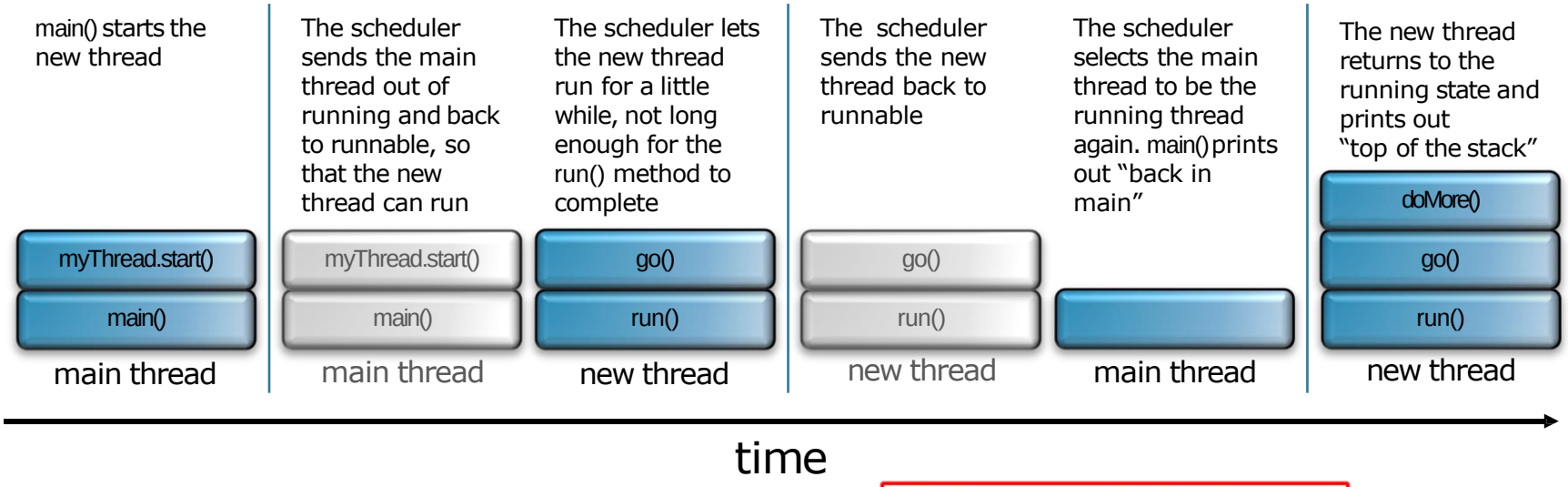
Sample output

```
top of the stack
back in main
```

```
public class MyRunnable implements Runnable {
   public void run() { go(); }
   public void go() { doMore(); }

   public void doMore() {
      System.out.println("top of the stack");
   }

   public static void main(String[] args) {
      Runnable threadJob = new MyRunnable();
      Thread myThread = new Thread(threadJob);
      myThread.start();
      System.out.println("back in main");
   }
}
```

A Multi-threading Example

Scenario 2:



Sample output

```
back in main
top of the stack
```

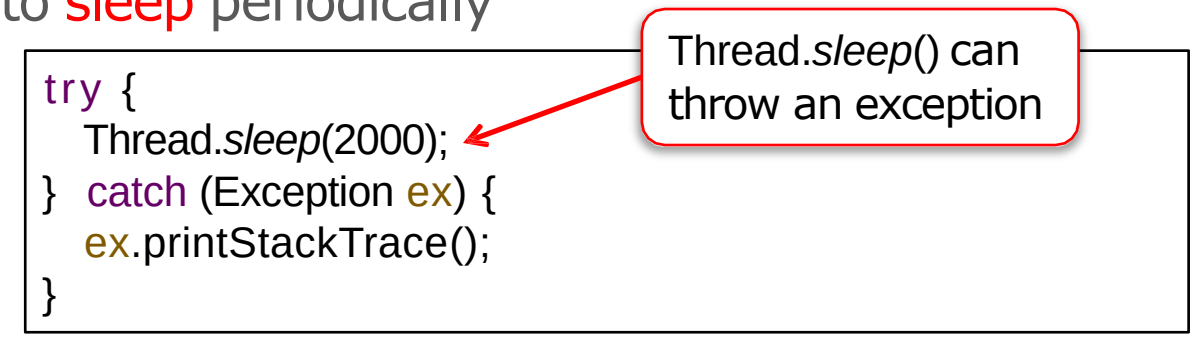
```
public class MyRunnable implements Runnable {
   public void run() { go(); }
   public void go() { doMore(); }

   public void doMore() {
      System.out.println("top of the stack");
   }

   public static void main(String[] args) {
      Runnable threadJob = new MyRunnable();
      Thread myThread = new Thread(threadJob);
      myThread.start();
      System.out.println("back in main");
   }
}
```

An Analogy of Threads (for Easier Understanding)

- Mrs. Chan likes investing in the stock market. She always uses her mobile phone to check the stock prices.
- Mr. Chan: "You are so busy monitoring the stock market. How can you handle the housework?“
- Mrs. Chan: “I will hire a domestic house assistant to handle the housework.“ (domestic house assistant ⇔ worker thread)
- In reality, Mrs. Chan doesn't hire any domestic house assistant. She just schedules herself to handle both tasks. She does some housework, monitors the stock market for a while and then goes back to her housework.
- In view of Mr. Chan, it seems like there are two workers at home (main thread + 1 worker thread) doing two different tasks but in fact, there is only one worker (1 processor) doing both tasks by proper scheduling.

Putting a Thread to Sleep

- One of the best ways to help your threads take turns is to put them to sleep periodically



- Putting a currently-running thread to sleep will force it to leave the running state, thus giving another thread a chance to run
- A sleeping thread will not wake up before the specified duration (in milliseconds)
- When a thread wake up, it always goes back to the runnable state

Concurrency Problem

- Concurrency problem may occur when 2 or more threads have access to the same object on the heap
- Having 2 or more threads accessing the same object at approximately the same time will result in a race condition, and may cause data corruption
- Example
  - Mr. and Mrs. Smith share a bank account
  - They always check the balance before making a withdrawal to ensure their account will not be overdrawn
  - The problem is they always fall asleep in between checking the balance and making the withdrawal
  - When they wake up, they make the withdrawal without checking the balance again

Mr. & Mrs. Smith Example

Example

```
public class BankAccount {
   private int balance = 100;
   public int getBalance() { return balance; }
   public void withdraw(int amount) {
      balance = balance – amount;
   }
}
```

```
public class SmithJob implements Runnable {
   private BankAccount account = new BankAccount();

   public static void main(String[] args) {
      SmithJob theJob = new SmithJob();
      Thread mrSmith = new Thread(theJob);
      Thread mrsSmith = new Thread(theJob);
      mrSmith.setName("Mr. Smith");
      mrsSmith.setName("Mrs. Smith");
```

One single shared bank account

2 threads having the same job that accesses the same bank account

Mr. & Mrs. Smith Example

Example

```
    mrSmith.start();
    mrsSmith.start();
}

public void run() {
    for (int x = 0; x < 2; x++) {
        makeWithdrawal(60);
    }
}

private void makeWithdrawal(int amount) {
    if (account.getBalance() >= amount) {
        System.out.println(getName() + " is about to withdraw");
        try {
            System.out.println(getName() + " is going to sleep");
            Thread.sleep(500);
        } catch (Exception ex) { ex.printStackTrace(); }
```

Making a withdrawal of $60 twice

Checking the balance before withdrawal

Falling asleep

Mr. & Mrs. Smith Example

- Example

```
            System.out.println(getName() + " wakes up");
            account.withdraw(amount);
            System.out.println(getName() + " completes the withdrawal");
            if (account.getBalance() < 0) {
               System.out.println("Overdrawn!");
            }
         }
         else {
            System.out.println("Not enough money for " + getName());
         }
      }

      private String getName() {
         return Thread.currentThread().getName();
      }
   }
```

Waking up and completing the withdrawal

Checking for overdrawn

Mr. & Mrs. Smith Example

- Sample output

```
Mr. Smith is about to withdraw
Mr. Smith is going to sleep
Mrs. Smith is about to withdraw
Mrs. Smith is going to sleep
Mr. Smith wakes up
Mrs. Smith wakes up
Mr. Smith completes the withdrawal
Mrs. Smith completes the withdrawal
Overdrawn!
Overdrawn!
Not enough money for Mr. Smith
Not enough money for Mrs. Smith
```

Synchronization

- To solve the concurrency problem in the previous example, we need to make sure that once a thread has checked the account balance, it has a guarantee that it can wake up and finish the withdrawal before any other thread can check the account balance
- This can be achieved by making the makeWithdrawal() method atomic
- Use the keyword synchronized to modify a method so that only 1 thread at a time can access it

```
private synchronized void makeWithdrawal(int amount) {
   // ...
}
```

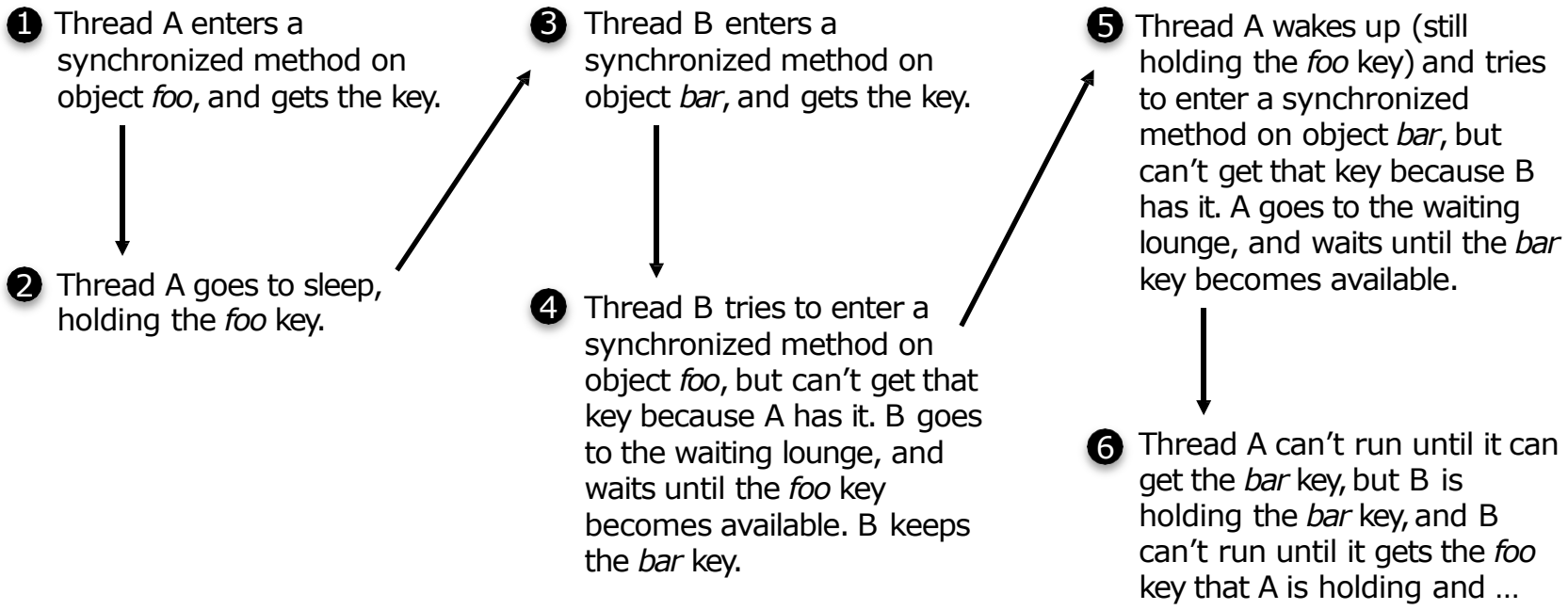
- To protect your data (like the bank account), synchronize the methods that act on that data

Using an Object's Lock

- Every Java object has a lock with a single key
- Most of time, the lock is unlocked
- Object locks come into play only when there are synchronized methods
- A thread can enter one of the synchronized methods only if it can get hold of the object's key
- Even if an object has more than 1 synchronized method, there is still only 1 key
- Once a thread has entered a synchronized method on an object, no other threads can enter any synchronized methods on the same object

Deadlock Problem

- A thread deadlock happens when you have two threads, both of which are holding a key the other thread wants



Deadlock Problem

- There is no way out of this scenario and the two threads will simply sit and wait forever!
- Java has no mechanism to handle deadlock, and it won't even know a deadlock occurred
- It is up to you to design carefully!

Chapter 11.

End

2020-2021
COMP2396 Object-Oriented Programming and Java
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong