Frédéric Boisguérin
fboisgue@insa-toulouse.fr

Object-Oriented Programming

Disclaimer

High density course

Outline

1. Introduction
2. Pragmatic programming
3. User interaction
4. Concurrent programming
5. Network programming

Processes and threads

Process *vs.* Thread

Process :

- Has its own execution environment
- Has its own memory space
- Can communicate with another process through *pipes* (e.g. C language) or *sockets*

Java Virtual Machine = one single process

Thread :

- Threads exist within a process
  → every process has at least one (main)
- Can share resources with each other (memory, open files, env. variables)

✔ Efficient communication

✘ Issues : concurrent access, deadlock...

The class `Thread`

Each thread is associated with an instance of the class **`Thread`**.

There are **two ways** for using threads :

- Directly instantiate them to control creation and management :
  `(new Thread(runnable)).start()`
- Use an **`ExecutorService`** that will create/manage threads for you :
  `Executors.`*`newSingleThreadExecutor`*`().submit(runnable)`,
  `Executors.`*`newFixedThreadPool`*`(4).submit(runnable)`...

The interface `Runnable`

```java
package java.lang;

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

```java
public class MyUsefulRunnable implements Runnable {
    @Override
    public void run() {
        while (true) {
            try {
                System.out.println(LocalDateTime.now());
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // Something wrong happened... But what?
                return;
            }
        }
    }
}
```

Interrupts

```java
public class ConsoleClock implements Runnable {
    @Override
    public void run() {
        while (true) {
            try {
                print(LocalDateTime.now());
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                print("I've been interrupted !");
            }
        }
    }
}
```

```java
public static void main(String[] args) throws Exception {
    Thread myclock = new Thread(new ConsoleClock());
    myclock.start();
    Thread.sleep(5000);
    myclock.interrupt();
}
```

How many instances of `Thread` ?

What could we read on the standard output ?

Synchronization

Concurrent accesses

```java
public class UnsafeBankAccount {
    private int balance = 0;
    public void deposit(int amount) {
        balance += amount;
    }
    public void withdraw(int amount) {
        balance -= amount;
    }
    public int balance() {
        return balance;
    }
}
```

```java
public static void main(String[] args) throws Exception {
    UnsafeBankAccount bankAccount = new UnsafeBankAccount();
    Thread jacky = new Thread(() -> bankAccount.deposit(100));
    Thread michel = new Thread(() -> bankAccount.withdraw(20));
    jacky.start();
    michel.start();
    Thread.sleep(5000); // Wait a bit
    System.out.println(bankAccount.balance());
}
```

How many instances of `Thread` ?
What could we read on the standard output ?

synchronized methods

```java
public class SynchronizedBankAccount {
    private int balance = 0;
    public synchronized void deposit(int amount) {
        balance += amount;
    }
    public synchronized void withdraw(int amount) {
        balance -= amount;
    }
    public synchronized int balance() {
        return balance;
    }
}
```

```java
public static void main(String[] args) throws Exception {
    UnsafeBankAccount bankAccount = new UnsafeBankAccount();
    Thread jacky = new Thread(() -> bankAccount.deposit(100));
    Thread michel = new Thread(() -> bankAccount.withdraw(20));
    jacky.start();
    michel.start();
    Thread.sleep(5000); // Wait a bit
    System.out.println(bankAccount.balance());
}
```

What could we read on the standard output ?

Deadlock

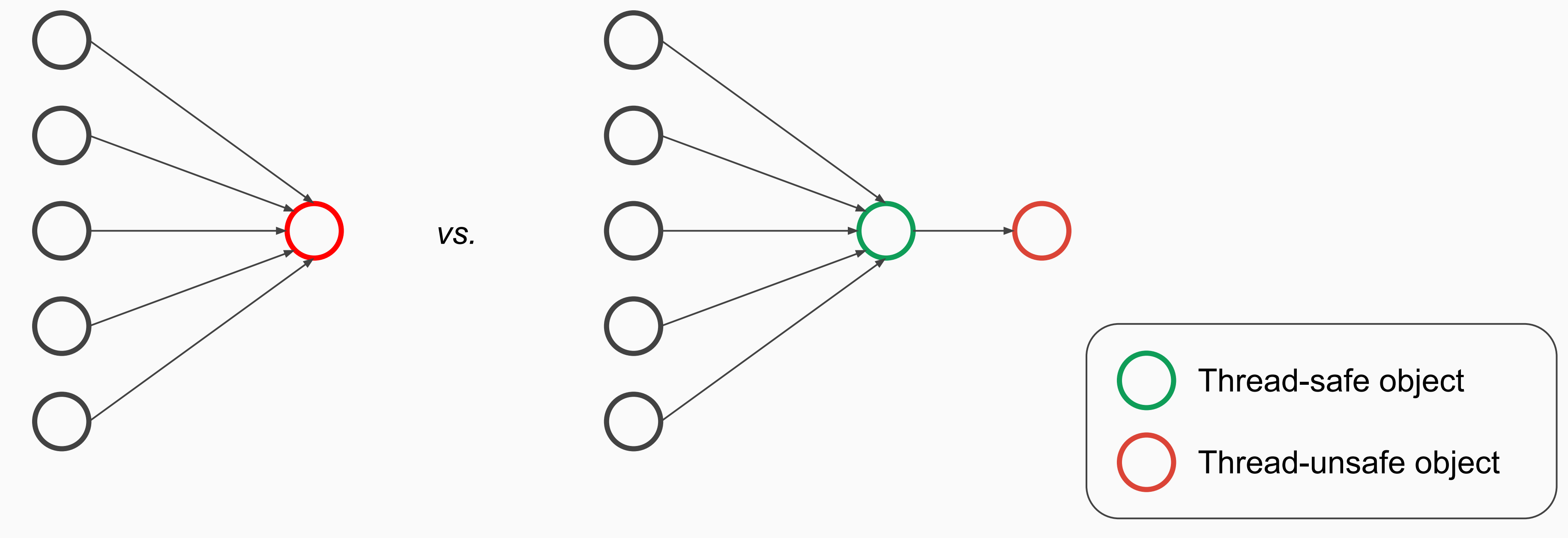
```java
public class Friend {
    private final String name;
    public Friend(String name) {
        this.name = name;
    }
    public synchronized void ping(Friend friend) {
        print(this.name + " pings " + friend.name);
        friend.pingBack(this);
    }
    public synchronized void pingBack(Friend friend) {
        print(this.name + " pings back " + friend.name);
    }
}
```

```java
public static void main(String[] args) {
    Friend alphonse = new Friend("Alphonse");
    Friend gaston = new Friend("Gaston");
    new Thread(() -> alphonse.ping(gaston)).start();
    new Thread(() -> gaston.ping(alphonse)).start();
}
```

What could happen here ?

Tips to deal with concurrency

Do you really need to deal with concurrency?



Immutable objects

- Don't provide "setter" methods
- Make all fields `final` and `private`
- Don't share references to the mutable objects between different threads

Concurrent Collections

- `BlockingQueue` defines a FIFO data structure that blocks when you attempt to add to a full queue, or retrieve from an empty queue
- `ConcurrentHashMap` (implements `Map`) makes atomic `put()` and `get()` operations to avoid synchronization
- ...

→ See package `java.util.concurrent`

Atomic variables

```
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicBankAccount {
    private AtomicInteger balance = new AtomicInteger();
    public void deposit(int amount) {
        balance.addAndGet(amount);
    }
    public void withdraw(int amount) {
        balance.addAndGet(-amount);
    }
    public int value() {
        return balance.get();
    }
}
```

Outline

1. Introduction
2. Pragmatic programming
3. User interaction
4. Concurrent programming
5. Network programming

Talk to the outside world with `java.net`

To read data from a `Socket`, use a `BufferedReader`

1. Make a socket connection to the server

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

2. Make an InputStreamReader chained to the Socket's low-level (connection) input stream

```
InputStreamReader stream = new InputStreamReader(chatSocket.getInputStream());
```

Low-level byte stream

3. Make a BufferedReader and read !

```
BufferedReader reader = new BufferedReader(stream);
String message = reader.readLine();
```

To write data to a `Socket`, use a `PrintWriter`

1. Make a socket connection to the server

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

2. Make a PrintWriter chained to the Socket's low-level (connection) output stream

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

Low-level byte stream

3. Write something

```
writer.println("Hi folks!");
```

TCP connections

Writing a simple TCP server

Local listening port

```
ServerSocket serverSocket = new ServerSocket(5000);
Socket chatSocket = serverSocket.accept();
PrintWriter writer = new PrintWriter(chatSocket.getOutputStream());
writer.println("Hi folks! You are connected to the server!");
writer.close();
```

Writing a simple TCP client

Server port

```
Socket chatSocket = new Socket("127.0.0.1", 5000);
InputStreamReader stream = new InputStreamReader(chatSocket.getInputStream());
BufferedReader reader = new BufferedReader(stream);
String message = reader.readLine();
System.out.println("Server said: " + message);
```

Serialization

Serializable objects

```
public class Message implements Serializable {
    private final String title;
    private final String content;
    public Message(String title, String content) {
        this.title = title;
        this.content = content;
    }
    public String getTitle() { return title; }
    public String getContent() { return content; }
}
```

Server with object serialization

```
ServerSocket serverSocket = new ServerSocket(5000);
Socket chatSocket = serverSocket.accept();

ObjectOutputStream stream = new ObjectOutputStream(chatSocket.getOutputStream());
Message messageToTransfer = new Message("Hello", "Some content");
stream.writeObject(messageToTransfer);

stream.close();
```

Client with object serialization

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

ObjectInputStream stream = new ObjectInputStream(chatSocket.getInputStream());
Message message = (Message) stream.readObject();

String logFormat = "Server send a message with title [%s] and content [%s]";
System.out.println(format(logFormat, message.getTitle(), message.getContent()));

stream.close();
```

Datagram sockets (UDP)

Send a DatagramPacket

```
DatagramSocket senderSocket = new DatagramSocket();
byte[] data = buildMessageAsBytes();
DatagramPacket datagramPacket = new DatagramPacket(data, data.length);
datagramPacket.setAddress(InetAddress.getByName("255.255.255.255"));
datagramPacket.setPort(RECEIVER_PORT);
senderSocket.send(datagramPacket);
senderSocket.close();
```

Receive a `DatagramPacket`

```
DatagramSocket receiverSocket = new DatagramSocket(RECEIVER_PORT);
DatagramPacket receivedPacket = new DatagramPacket(new byte[BUFFER_SIZE], BUFFER_SIZE);
receiverSocket.receive(receivedPacket);
byte[] data = receivedPacket.getData();
```

Thank you!

Questions ?

Next course :

***You** decide what you wanna know !*

Resources

Java Tutorials

- Concurrency : https://docs.oracle.com/javase/tutorial/essential/concurrency
- Network : https://docs.oracle.com/javase/tutorial/networking/index.html