# Concurrency in Java

## OOP week 10 lectures

Dr. **Madasar** Shah

November 2021

# Course Overview

This Semester:

- Object Oriented Programming (Prof Reddy, Dr Chetty)
- **Programming Concurrency in Java**

This week:

- Motivating concurrency, Processes vs Threads, common pitfalls
- Thread programming, more pitfalls, concurrency patterns

Question: in all our programs so far statements have run sequentially. How do we run two or more statements in parallel to each other?

# Motivation 0: a simple server in Java

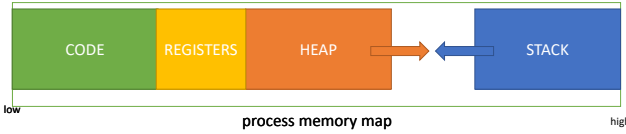
- Example server without concurrency : `socketserver.SimpleSocketServer`
- `Usage: SimpleSocketServer <port>`
- Set a port number in intelliJ 'Edit configurations', near the ▷ button

Question: can we connect more than once to this server?

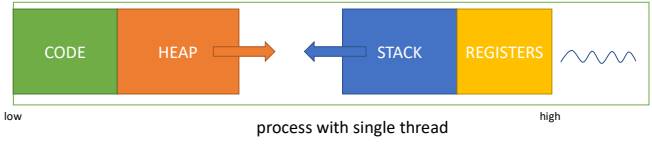
Question: can more than one person connect to the server simultaneously?

# Processes vs Threads: Motivation I

Our programs **can access the entire memory space** allocated to the process.

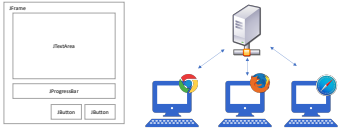



Before: **un threaded, non-concurrent programs:** Follow the sequence of statements we write in-order, **one statement at a time, to modify memory**.

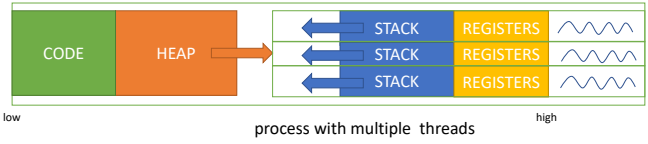


# Processes vs Threads: Motivation II

**Problem**: in certain situations, we **want more than one statement to be active at any one time** within our process. e.g. GUI app or web server apps.




Solution: **multi threaded, concurrent programs:** follow a few specific design patterns to **allow concurrent statements to execute in parallel** in one program.




Caveat: concurrent programming **requires more design and development skill** than non-concurrent programming.

# Concurrent Programming: Threads

Question: What are threads?

## Concurrent Programming: Threads

What are threads?

- One means to **achieve concurrent processing**
- A standard **library** in Java (`java.lang.Thread`)
- An OO **design pattern**: *Active Object*
- A **conventional API** in the operating-system kernel, also: **architectural pattern**
- A **feature** supported by many CPU designs
- Take **extra effort to develop**, test and debug (non-deterministic)
- **Difficult to reason** about mechanically
- In order to write threaded programs we must **discuss the compilation of our non-threaded code**

# Concurrent Programming: Key Challenge

**Always consistent modification of shared memory** using **synchronised access** to variables and objects.

# Shared data and concurrent processing

- **Threads** used to achieve parallel execution
- Threads can be **interrupted**
- **Non-atomic operations on shared data become interleaved when using multiple threads**
- Result: data can become corrupted

# Non-atomic operations

`balance++` is a **non-atomic**, **stack-based** operation.

- `balance++` compiled as:
  - **load** `balance` into cpu register r
  - **increment** register r
  - **store** r into `balance`

Inherent problem with concurrent processing that cannot be solved in hardware. Software-based **synchronisation** of resource updates needed.

Example: iplusplus

# Case Study: Concurrent Cashier System

Let's start with an example of buggy, concurrent code:

- Account.java (one object, shared by cashiers)
- Cashier.java (zero or more *active* objects)
- Manager.java (one object)

Shared Data Structure: `Account.java`

```
1  class Account {
2
3    private int balance = 0;
4
5    public void increment() {
6      balance++;
7    }
8    ...
9  }
```

*try it on codesnip!*

# Non-solution: `volatile` keyword

- `volatile` keyword: variable reads and variable writes become atomic
- declare `int balance` as `volatile`

```
1  class Account {
2
3    private volatile int balance = 0;
4    ...
5  }
```

*try it on codesnip!*

Does not fix the issue in this code- why?

# Possible solution 2a: `synchronized` keyword

- synchronized keyword: marks whole method as a *critical section*
- declare `increment()` and `decrement()` as synchronized

```
1  class Account {
2      ...
3      public synchronized void increment()
4      ...
5      public synchronized void decrement()
6      ...
7  }                                                    try it on codesnip!
```

Fixed!

Problem: overkill for all but the most simple methods.

## Possible solution 2b: `synchronized` block

- `synchronized` block: marks selected statements as a *critical section*
- mark `balance++` and `balance--` as synchronized

```
1  class Account {
2      ...
3      public void increment(){
4          synchronized(this){
5              balance++;
6          }
7      }
8      ...
9  }                                                    try it on codesnip!
```

Question: what is `this` and how is it an implicit *lock*?

Problem: `synchronized` not suitable with multiple exclusive variables.

# Possible solution 3: Lock object

- Create an **explicit** `Lock` object for each shared resource
- `lock()` and `unlock()` around `balance++` and `balance--`

```
1  class Account {
2
3      private Lock balanceLock = new ...
4
5      public void increment(){
6          balanceLock.lock();
7              balance++;
8          balanceLock.unlock();
9      }
10     ...
11 }
```

*try it on codesnip!*

result: fine-grained access control, standard Java syntax, lock per exclusive variables

# Next time: more on concurrency patterns in Java

- Writing threaded programs in Java
- Anti-patterns: spinlock, deadlock, starvation, race conditions, live lock
- Patterns: Active Object, Producer-Consumer, ThreadPool patterns and libraries

# End of Section

- Questions?

With thanks to Prof. Martín Escardó.