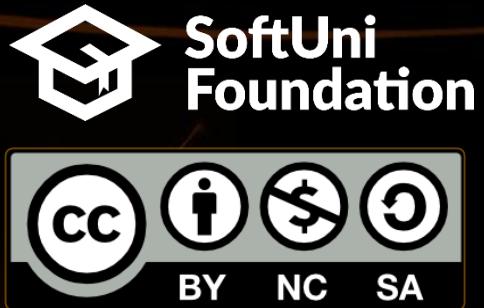
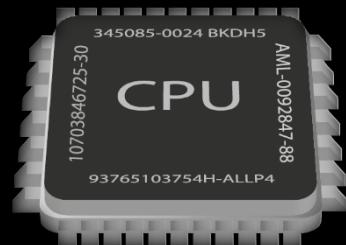


# Asynchronous Programming

## Writing Asynchronous Code in Java



**SoftUni Team**  
Technical Trainers  
Software University  
<http://softuni.bg>



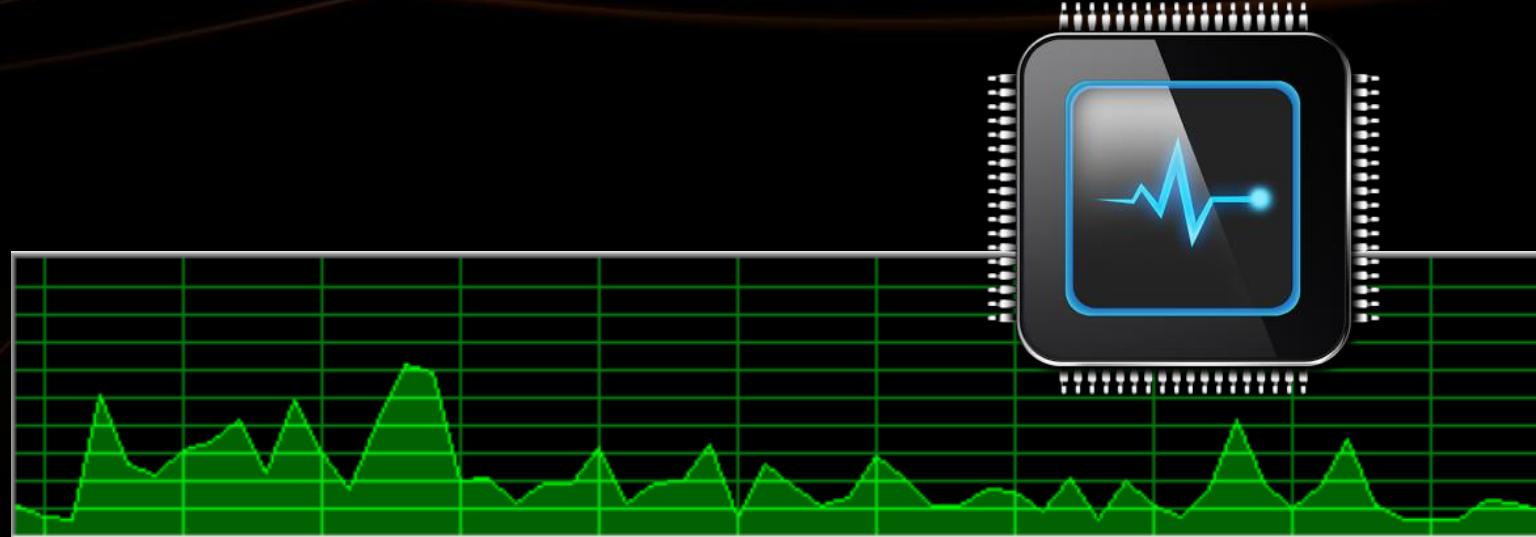
# Table of Contents

1. Processes, Threads, Tasks
2. **Sync** and **Async** Programming
3. Java High Level Threading
4. Race Conditions
5. Atomicity
6. Volatile



sli.do

#JavaFundamentals



# Single and Multi Threading

Executing Tasks Sequentially or Concurrently

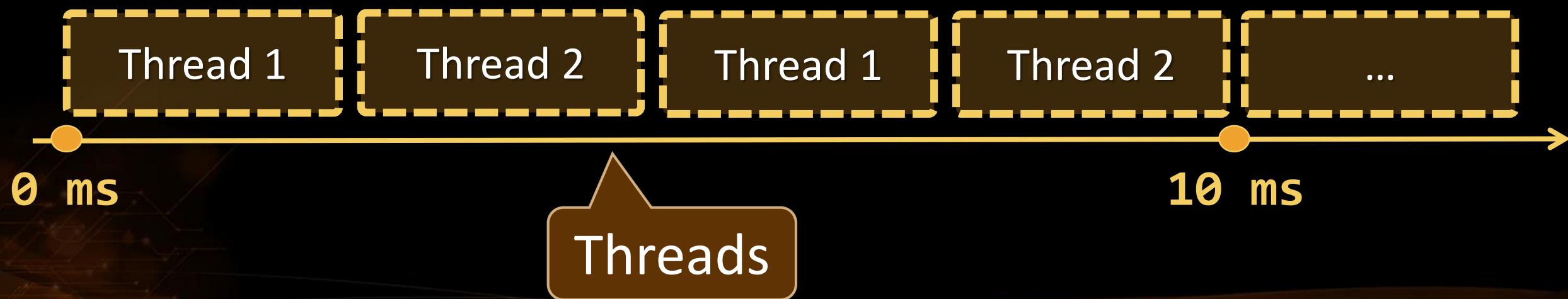
# Time Slicing

- A computer can **run many processes** (applications) at once
  - But single core CPU can execute one instruction at a time
  - **Parallelism** is achieved by the operating system's **scheduler**
  - Grants each **process** a small interval of time to run



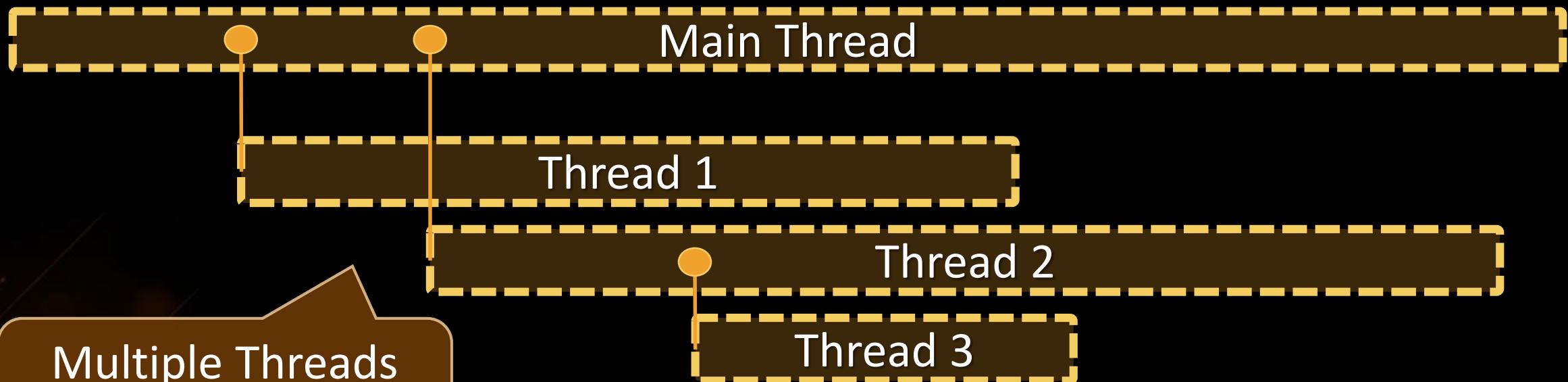
# Multi-Threading

- Processes have **threads** (at least a main thread)
- Similar to OS Multi-Tasking
- By **switching between threads**, a process can **do multiple tasks "at the same time"**



# Threads

- A **thread** executes a task
- A thread can **start other threads**



Multiple Threads  
"At the same time"

- A task is a **block of code** that is **executed by a Thread**
- A **Task in Java** is represented by the **Runnable** class

```
Runnable task = () -> {
    for (int i = 0; i < 10; i++) {
        System.out.printf("[%s] ", i);
    }
};
```

- A single thread is represented by the **Thread** class

```
Runnable task = () -> {
    for (int i = 0; i < 10; i++) {
        System.out.printf("[%s] ", i);
    }
};
```

```
Thread thread = new Thread(task);
thread.start();
```

Starts the given task

# Joining Threads

- **Join** == waiting for a thread to finish

```
Thread thread = new Thread(() -> {  
    while (true) { }  
});
```

```
thread.start();  
System.out.println("Executes.");  
thread.join();  
System.out.println("Can't be reached.");
```



Blocks the calling thread

# Problem: Single Thread

- Create a task that prints the numbers from 1 to 10
- Start a thread executing the task
- Add **System.exit(1)** at the end of your program
- Experiment with **thread.join()**

Exits the program



```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
1 2 3 4 5 6 7 8 9 10
Process finished with exit code 1
```

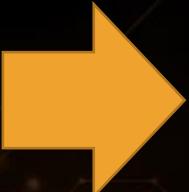
# Solution: Single Thread

```
Thread thread = new Thread(() -> {  
    for (int i = 1; i <= 10; i++) {  
        System.out.print(i + " ");  
    }  
});  
  
thread.start();  
thread.join();  
System.exit(1);
```

Try to remove  
this line

# Problem: Multi-Thread

- **Create a task** that prints the numbers from 1 to 10
- **Start 5 threads** executing the same task
- After each printing, add **Thread.yield()** statement
- **Join** all threads



```
"C:\Program Files\Java\jdk1.8.0_91\bin>java -Xms10m -Xmx10m -Djava.awt.headless=true com.alexander.ivanov.threads.ThreadsTest
[0] [0] [0] [0] [0] [1] [1] [2] [3] [-] [1] [1] [2]
Process finished with exit code 0
```

Signals CPU that another thread can be processed

# Solution: Multi-Thread

```
Runnable task = () -> {  
    for (int i = 0; i < 10; i++) {  
        System.out.printf("[%s] ", i);  
        Thread.yield();  
    }  
};  
  
// continues...
```

Try to comment  
this line

# Solution: Multi-Thread (2)

```
// Create the task
```

```
Thread[] threads = new Thread[5];
for (int i = 0; i < 5; i++)
    threads[i] = new Thread(task);
threads[i].start();

for (Thread thread : threads)
    thread.join();
```

- **interrupt()** – notifies the thread to interrupt its execution

```
Thread thread = new Thread(task);  
thread.start();  
thread.interrupt();
```

```
Runnable task = () -> {  
    if (Thread.currentThread().isInterrupted())  
        // Safely break the task  
}
```

# Problem: Responsive UI

- Create a program that prints the **primes from 0 to N**
- Implement a **responsive UI**, e.g. user can **stop the program**
- If stopped, show appropriate message



```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
n = 9999999
stop
Interrupted...
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]...
183682 primes calculated.
```

# Solution: Responsive UI

```
// Create task and thread

while (true) {
    String command = scanner.nextLine();
    if (command.equals("stop")) {
        thread.interrupt();
        break;
    } else
        System.out.println("unknown command");
}

thread.join();
```

# Solution: Responsive UI (2)

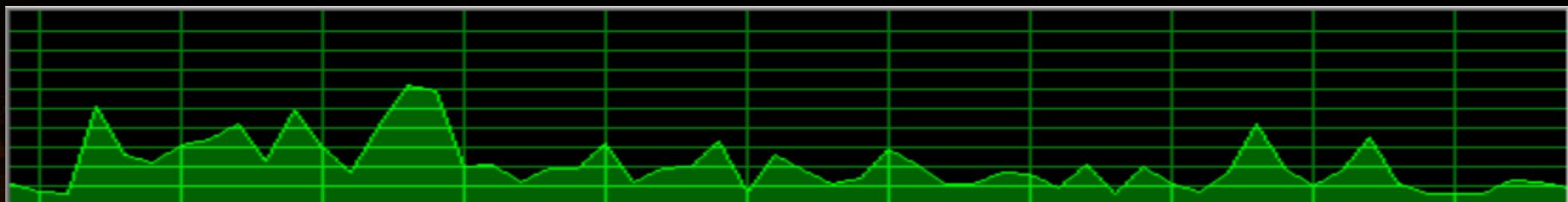
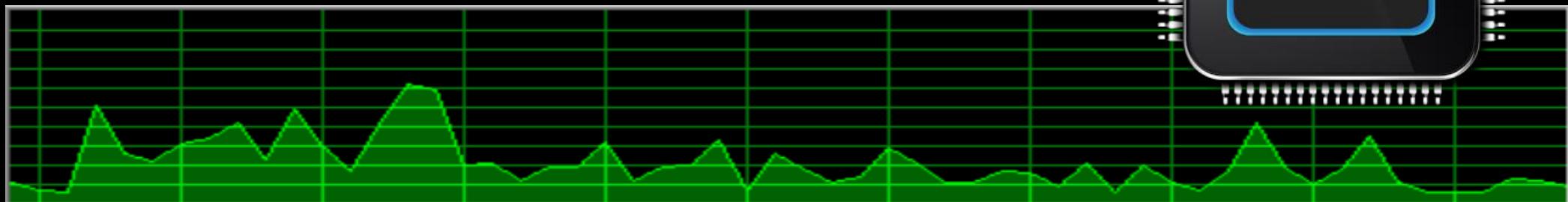
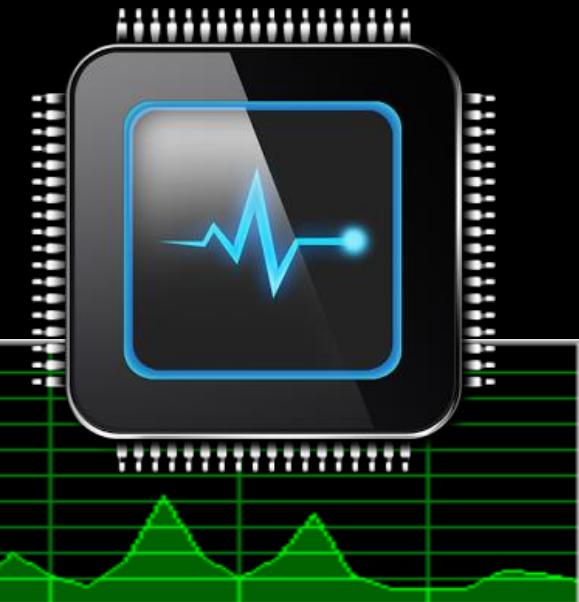
```
// Task

List<Integer> primes = new ArrayList<>();
for (int number = 0; number < to; number++)
    if (isPrime(number))
        primes.add(number);

if (Thread.currentThread().isInterrupted()) {
    System.out.println("Interrupted...");
    break;
}
```

# Multi-Threaded Code

- Two main **benefits**:
  - **Responsive User Interface (UI)**
  - Better **CPU Utilization**



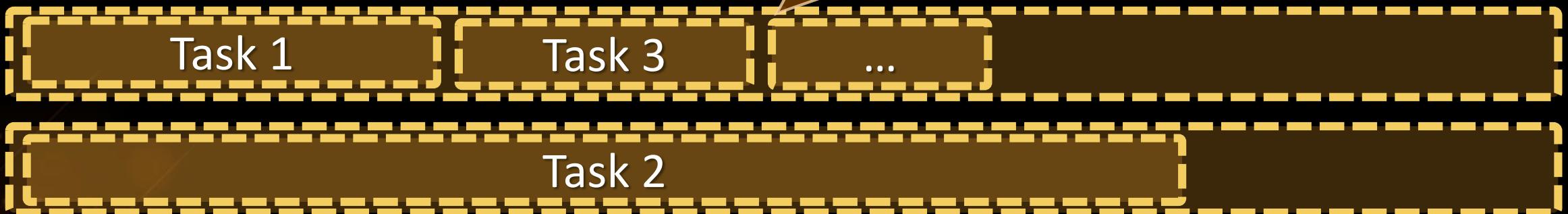
# Multi-Threaded CPU Utilization

- Single-Threaded



Heavy Operation

- Multi-Threaded



Percieved time is less

# High Level Threading

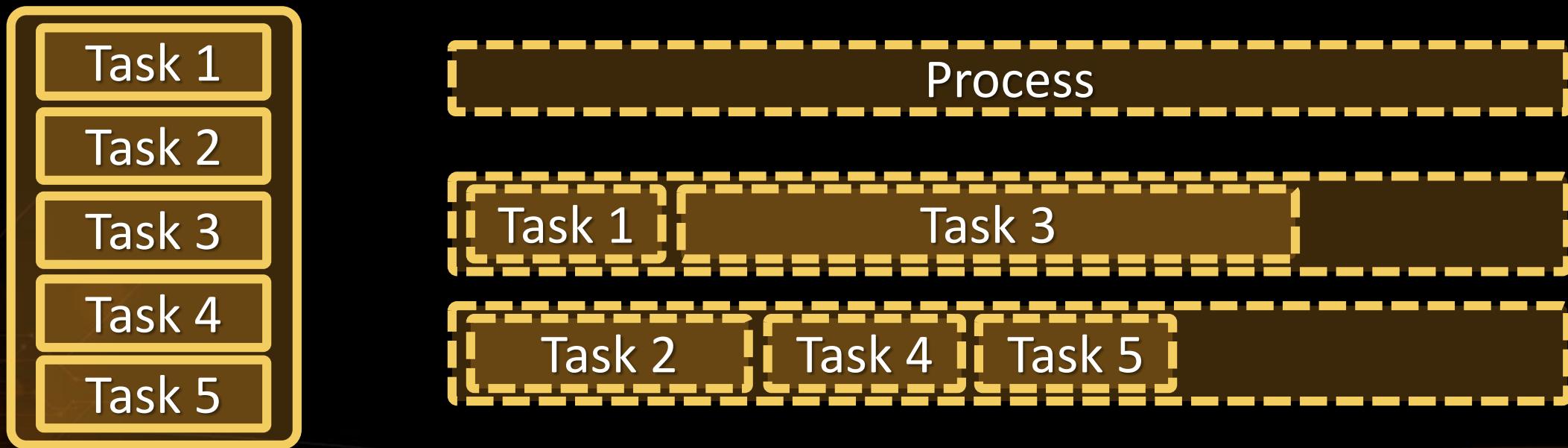
- **ExecutorService** class provides easier thread management

```
ExecutorService es =  
    Executors.newFixedThreadPool(2);
```

Several thread pool  
types are available

```
Runnable task = () -> isPrime(number);  
es.submit(task);
```

```
ExecutorService es =  
    Executors.newFixedThreadPool(2);
```



# Returning Value from a Task

- **Future<T>** - defines a result from a **Callable**:

```
ExecutorService es =  
    Executors.newFixedThreadPool(4);
```

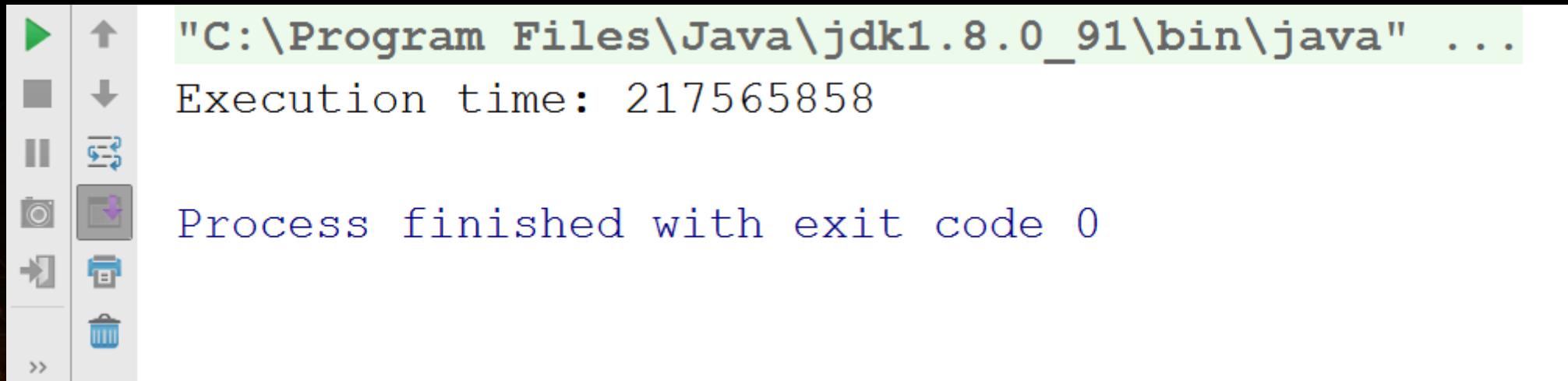
```
Future<Boolean> future =  
    es.submit(() -> isPrime(number));
```

```
if (future.isDone())  
    System.out.println(future.get());
```

blocks until done

# Problem: Benchmarking

- Test every number in the range [0...N] if it is prime or not
- Spread the calculation **over 2 or 4 threads**
- **Benchmark** and compare the difference over one thread
- Benchmark both efficient and inefficient **isPrime()**



```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
Execution time: 217565858
Process finished with exit code 0
```

The screenshot shows a terminal window with a dark background and light-colored text. On the left is a vertical toolbar with icons for running, stopping, pausing, and other operations. The main area displays the command "java" being run from a specific directory, followed by the execution time (217565858), and finally the message "Process finished with exit code 0".

# Solution: Benchmarking

- Create a `List<Integer>` for all numbers in range [0..N]
- Start timer (`System.nanoTime()`)
- Submit a task for each number, returning a `Future<Boolean>`
- Await `termination` and `shutdown` ES
- Stop timer (`System.nanoTime()`)



# Single and Multi-Threading

## Exercises in Class



# Resource Sharing

## Tasks Interfering with Each Other

- Atomic action is one that **happens all at once**
- Java – **reads** and **writes** on primitives (except double and long)

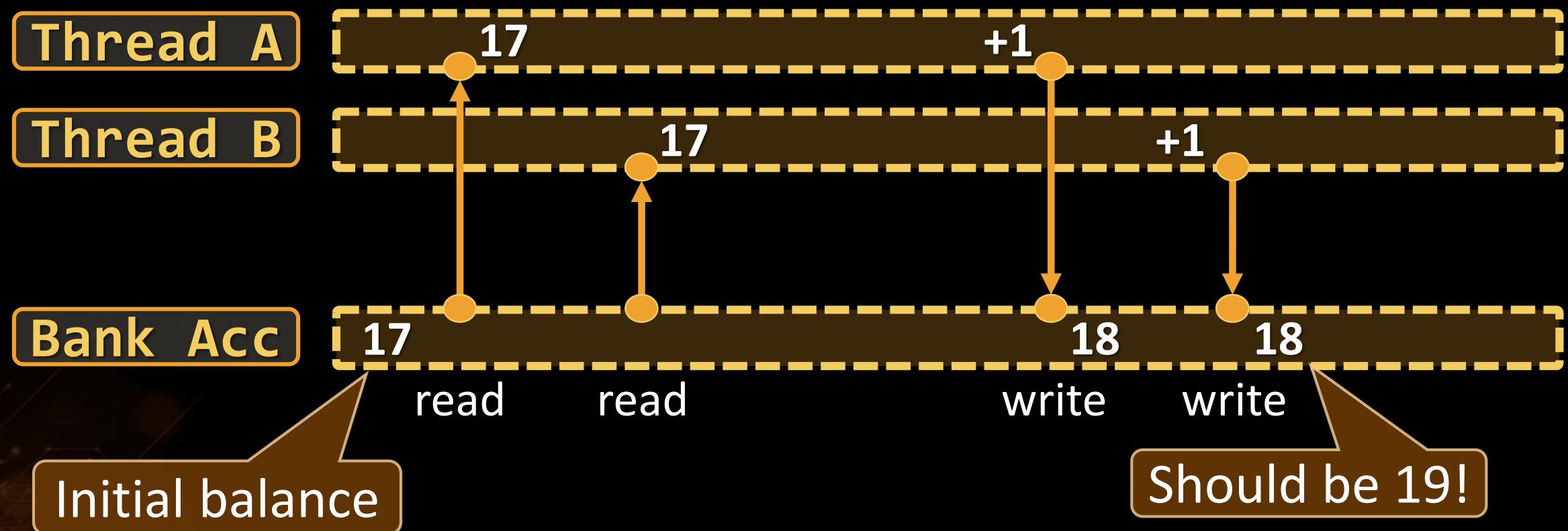
```
int a = 5; // atomic
int b = 6;

a++; // non-atomic
a = a + b;
a = b;
```



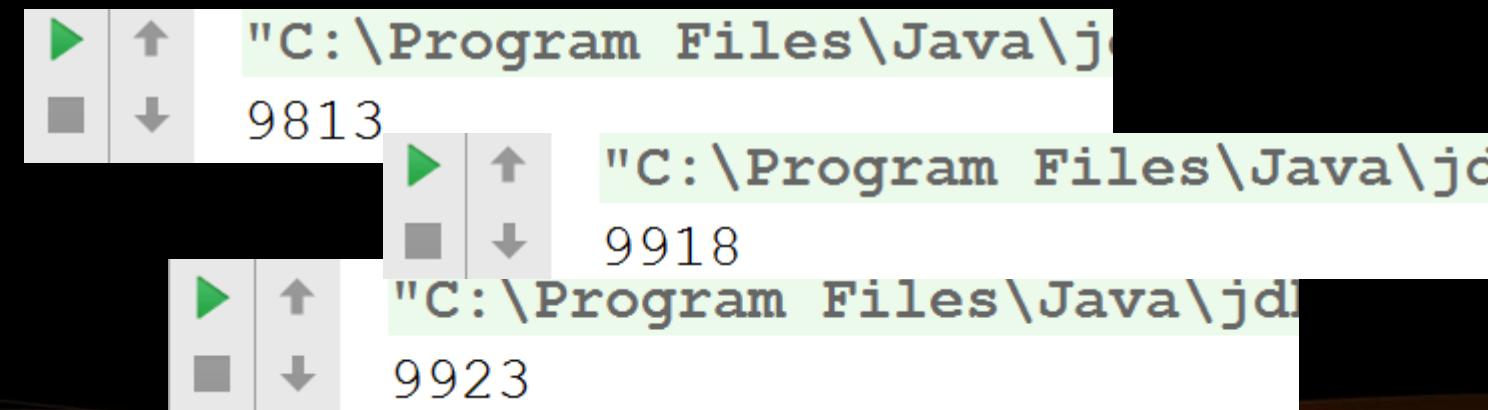
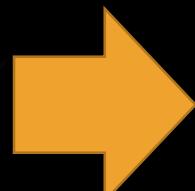
# Race Conditions

- Two different threads try to **read** and **write** data at the same time



# Problem: Transactions

- Create a **simple class** BankAccount:
  - Property: **int balance**
  - Method: **void deposit(int sum)**
- Create a **multi-threaded program** that simulates 100 transactions, each depositing 100 times 1 to the balance



```
"C:\Program Files\Java\jd... 9813
"C:\Program Files\Java\jd... 9918
"C:\Program Files\Java\jd... 9923
```

# Solution: Transactions (Unsafe)

```
class Account {  
    int balance;  
  
    void deposit(int amount) {  
        balance = balance + amount;  
    }  
}
```

Unsafe: Read +  
Write Operation

# Solution: Transactions (Unsafe) (2)

```
Runnable task = () -> {  
    for (int i = 0; i < 100; i++) {  
        account.deposit(1); }  
};
```

Unsafe: This  
may produce  
incorrect result

```
Thread[] threads = new Thread[transactions];  
for (int i = 0; i < 100; i++)  
    threads[i] = new Thread(task);  
threads[i].start();
```

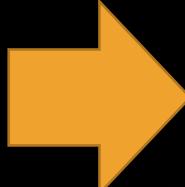
# Synchronized Keyword

- **synchronized**
  - Grants access to **only one thread** at a time
  - **Blocks** other threads until the **resource is released**
  - In other words, makes an operation **atomic**

```
synchronized (Object) {  
    // Thread safe code  
}
```

# Problem: Thread Safe Transactions

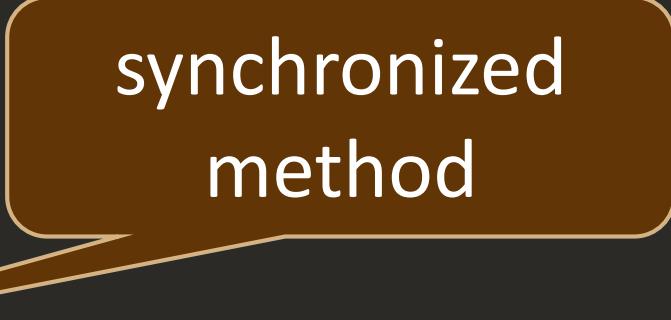
- Modify previous problem to **get same correct result every time**
- 100 transactions, each depositing 100 times 1 to the balance



```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
10000
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
10000
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
10000
```

# Solution: Transactions

```
class Account {  
    int balance;  
  
    synchronized void add (int amount) {  
        balance = balance + amount;  
    }  
}
```



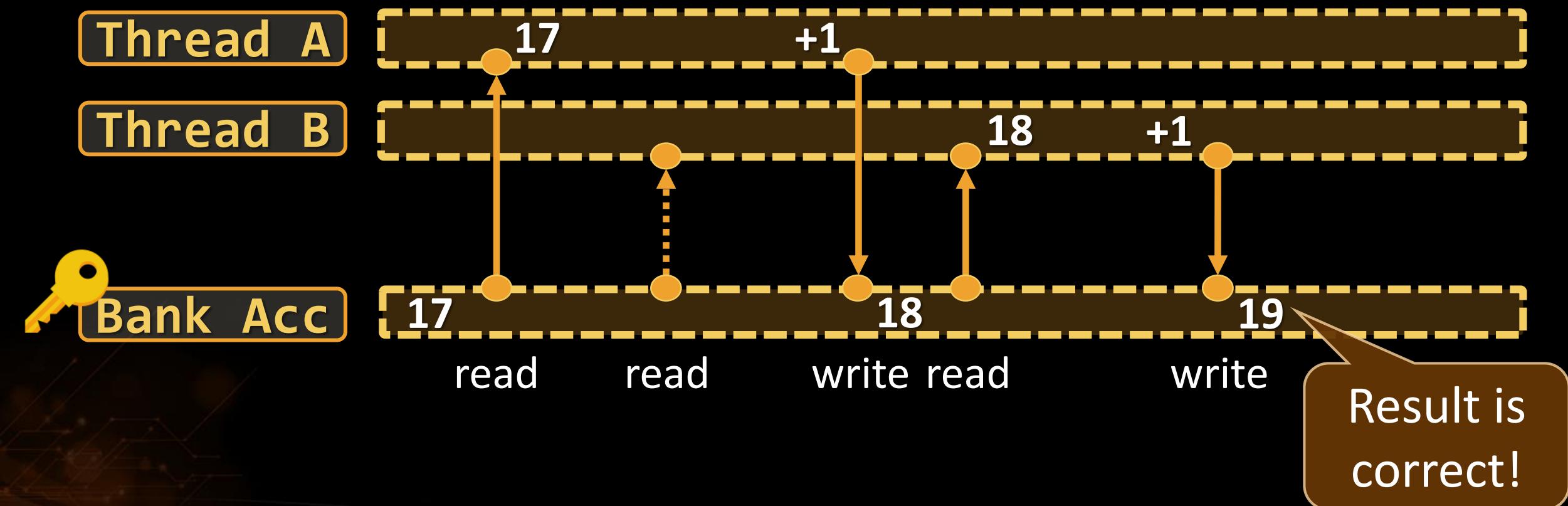
# Solution: Transactions (2)

```
class Account {  
    int balance;  
  
    void deposit (int amount) {  
        synchronized (this) {  
            balance = balance + amount;  
        }  
    }  
}
```

synchronized block

# Synchronized - Locks

- Synchronized works by **taking an object's Key**



- Every java object can be a key
- For static methods – Key is the **class itself**

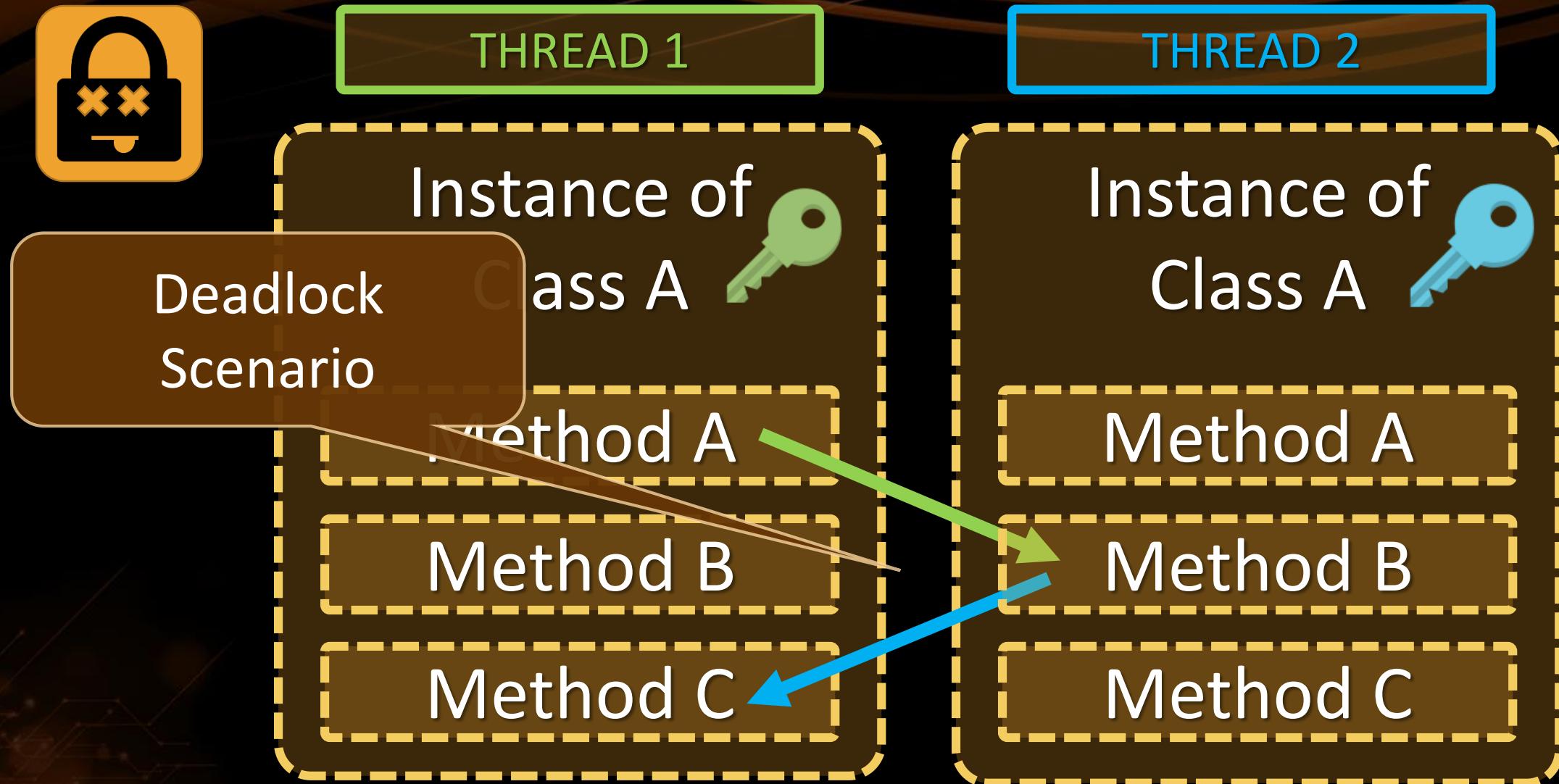
```
class Account {  
    int balance;  
    synchronized void add (int amount) {  
        balance = balance + amount;  
    }  
}
```



The object is the key

```
Account johnsAccount = new Account();
```

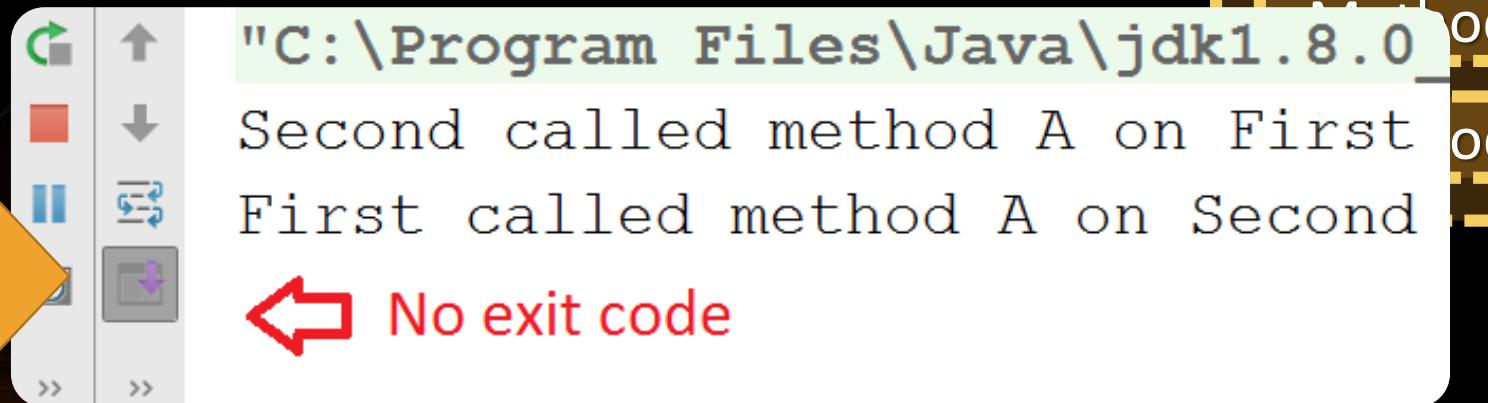
# Deadlocks



# Problem: Deadlock

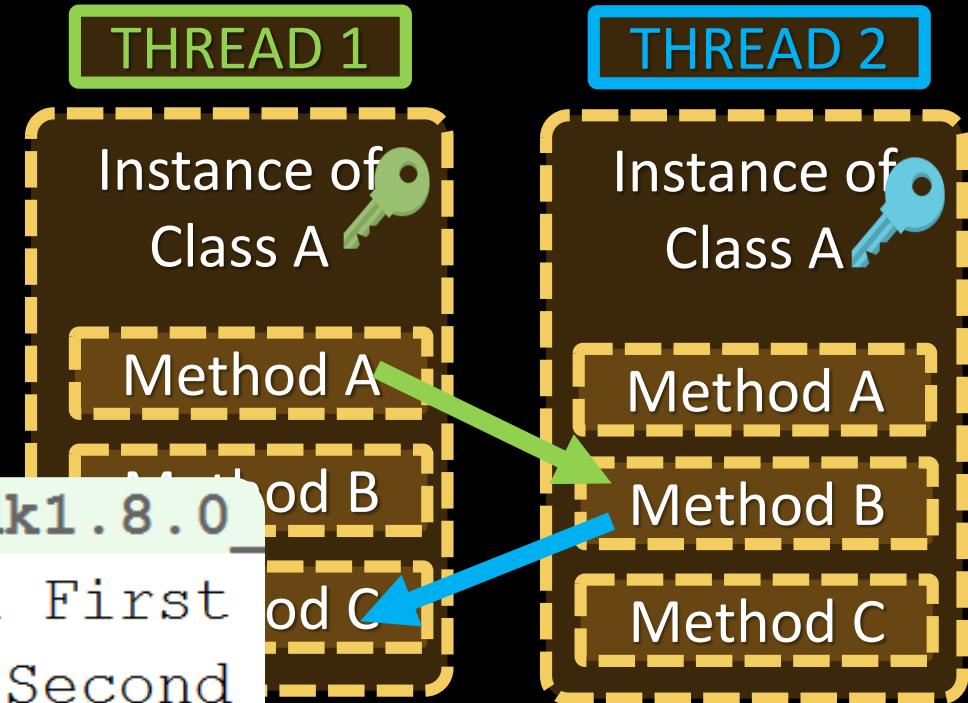
- Reproduce the deadlock scenario from the previous slide
- Use **Thread.sleep()**

To make sure  
methods execute  
at the same time



"C:\Program Files\Java\jdk1.8.0

```
Second called method A on First Thread
First called method A on Second Thread
No exit code
```



# Solution: Deadlock

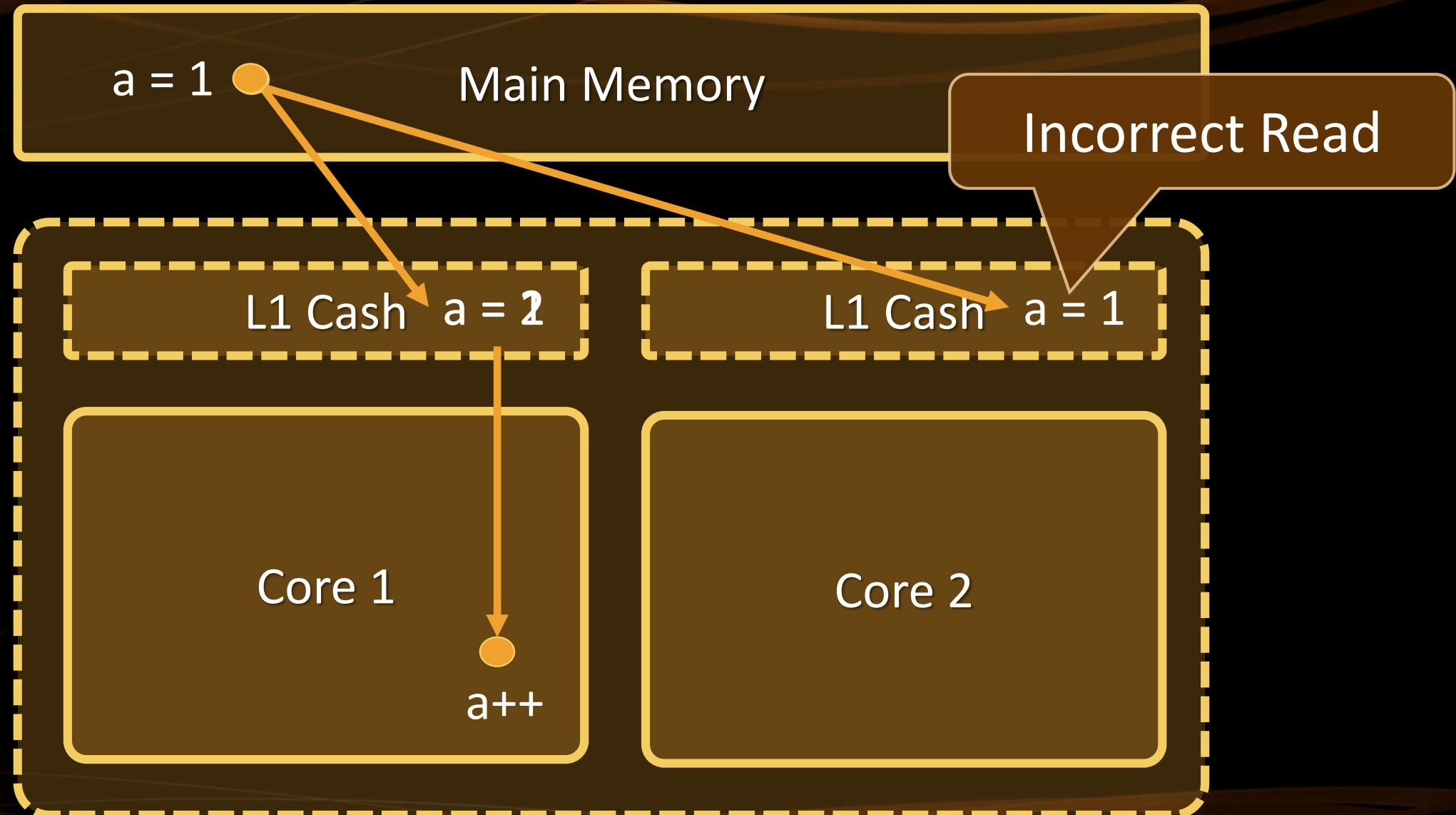
```
static class MyClass {  
    String id;  
    public MyClass(String id) {}  
    synchronized void a(MyClass other) {}  
    synchronized void b(MyClass other) {}  
    synchronized private void c() {}  
}
```

# Solution: Deadlock (2)

```
MyClass first = new MyClass("First");
MyClass second = new MyClass("Second");
Thread tFirst = new Thread(() ->
    first.a(second));
Thread tSecond = new Thread(() ->
    second.a(first));

tFirst.start();
tSecond.start();
```

# Visibility

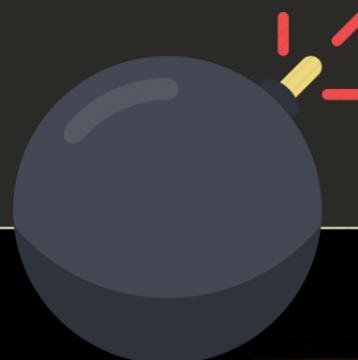


# Visibility (2)

- Every write inside a synchronized block is **guaranteed to be visible**
- Use **volatile** keyword

Every write is flushed  
to main memory

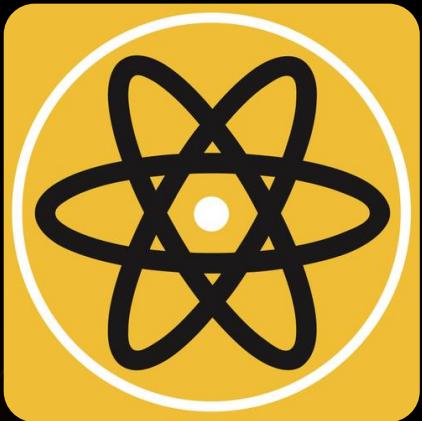
```
class Account {  
    volatile int balance;  
    synchronized void add (int amount) {  
        balance = balance + amount;  
    }  
}
```



# Concurrent Classes

- Java **java.util.concurrent** package provides thread-safe collection classes
- Some notable concurrent collections:
  - **ConcurrentLinkedQueue**
  - **ConcurrentLinkedDeque**
  - **ConcurrentHashMap**





# Race Conditions

## Exercises in Class

# Summary

- A **thread** is a unit of code execution
- **Multithreading** means a program can do several operations in parallel by using many threads
  - Used to offload CPU-demanding work so the main thread does not block
  - Can lead to synchronization issues and unexpected results
  - Java has many useful tools for asynchronous programming
    - **synchronized** and **volatile** keywords
    - **java.util.concurrent**



# Stream API



# Questions?



## SUPERHOSTING.BG

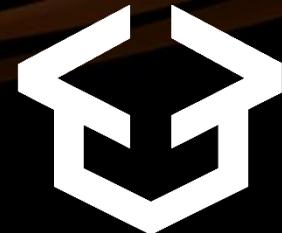
## INDEAVR

Serving the high achievers

The logo for Infragistics, featuring a blue square icon with a white 'i' shape inside, followed by the word "INFRASTICS™" and the tagline "DESIGN / DEVELOP / EXPERIENCE".

# Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg)
- Software University Foundation
  - <http://softuni.foundation/>
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University

