Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques in Java

Programming Techniques with Threads

Main bibliographic sources
- https://docs.oracle.com/javase/tutorial/essential/concurrency/
- Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea, Java Concurrency in Practice, Addison Wesley, Pearson Education
- K. Sharan, Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams 1st Edition, APRESS, 2014.

T. Cioara, V. Chifu, C. Pop
2025

Overview

Program

Algorithm written in a programming language

vs

Process

Running instance of a program having all system resources allocated by the operating system

Multitasking

Ability of an operating system to execute multiple tasks (or processes) at once

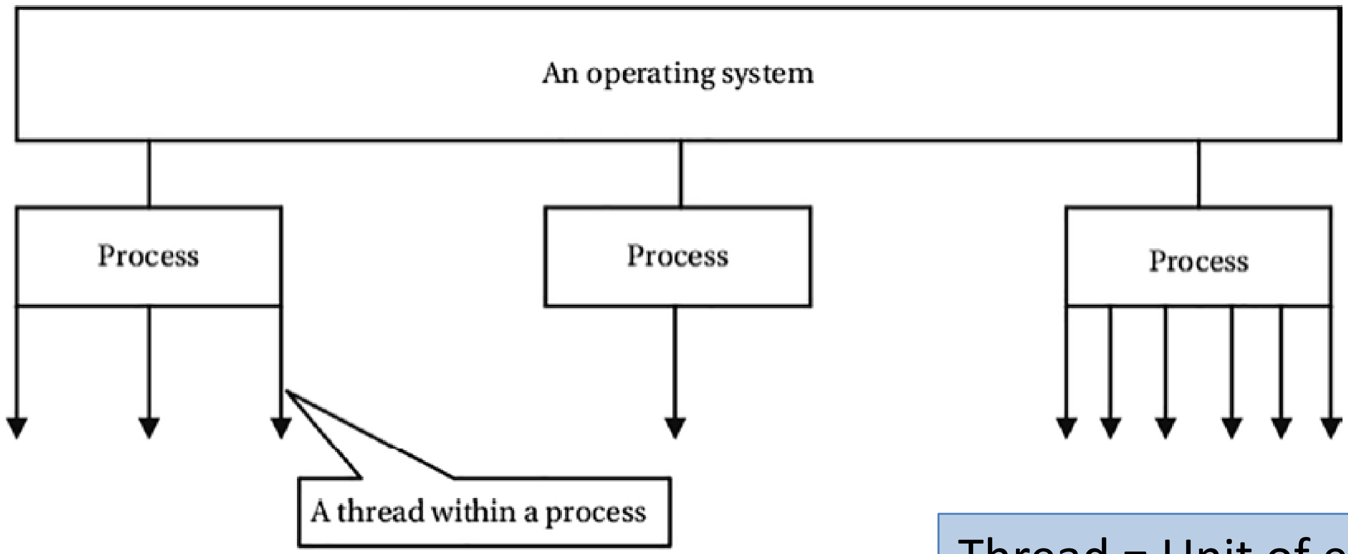
- **Cooperative Multitasking**

  The running process decides when to release the CPU so that other processes can use the CPU (e.g., ***Context switch****)*

- **Preemptive Multitasking**

  The operating system allocates a time slice to each process

Processes and Threads



Process = address space + resources + threads

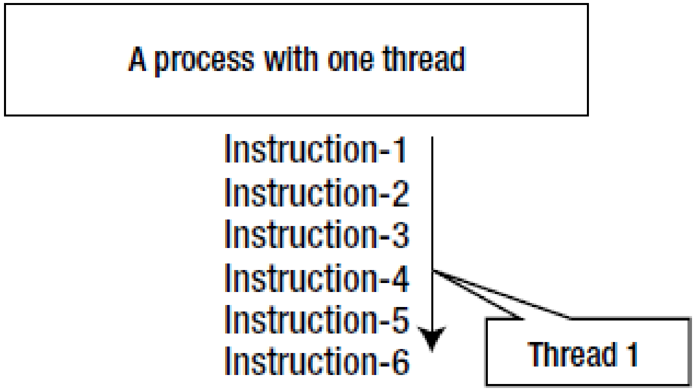
- Running instance of a program
- Communicates with other processes using Inter Process Communication (IPC) resources, such as pipes and sockets

Thread = Unit of execution within a process

- Shared access to address space and resources of the process
- Maintains a program counter, a stack and a private memory
- Communicate with each others

Threads

- Threads are scheduled on the CPU for execution, not the processes
  - Context switch occurs between the threads
- Multi-threaded program
  - Dividing the program logic to use two threads within a process



Creating a Thread in Java

Directly control thread creation and management

Instantiate Thread each time the application needs to initiate an asynchronous task

OR

Abstract Thread management

Pass the application's tasks to an executor

Creating a Thread in Java

1. Inheriting your class from the Thread Class

```
public class MyThreadClass extends Thread {
   @Override
   public void run() {
      System.out.println("Hello Java thread!");
   }
   // More code goes here
}
…
MyThreadClass myThread = new MyThreadClass();
myThread.start();
```

2. Implementing the Runnable Interface

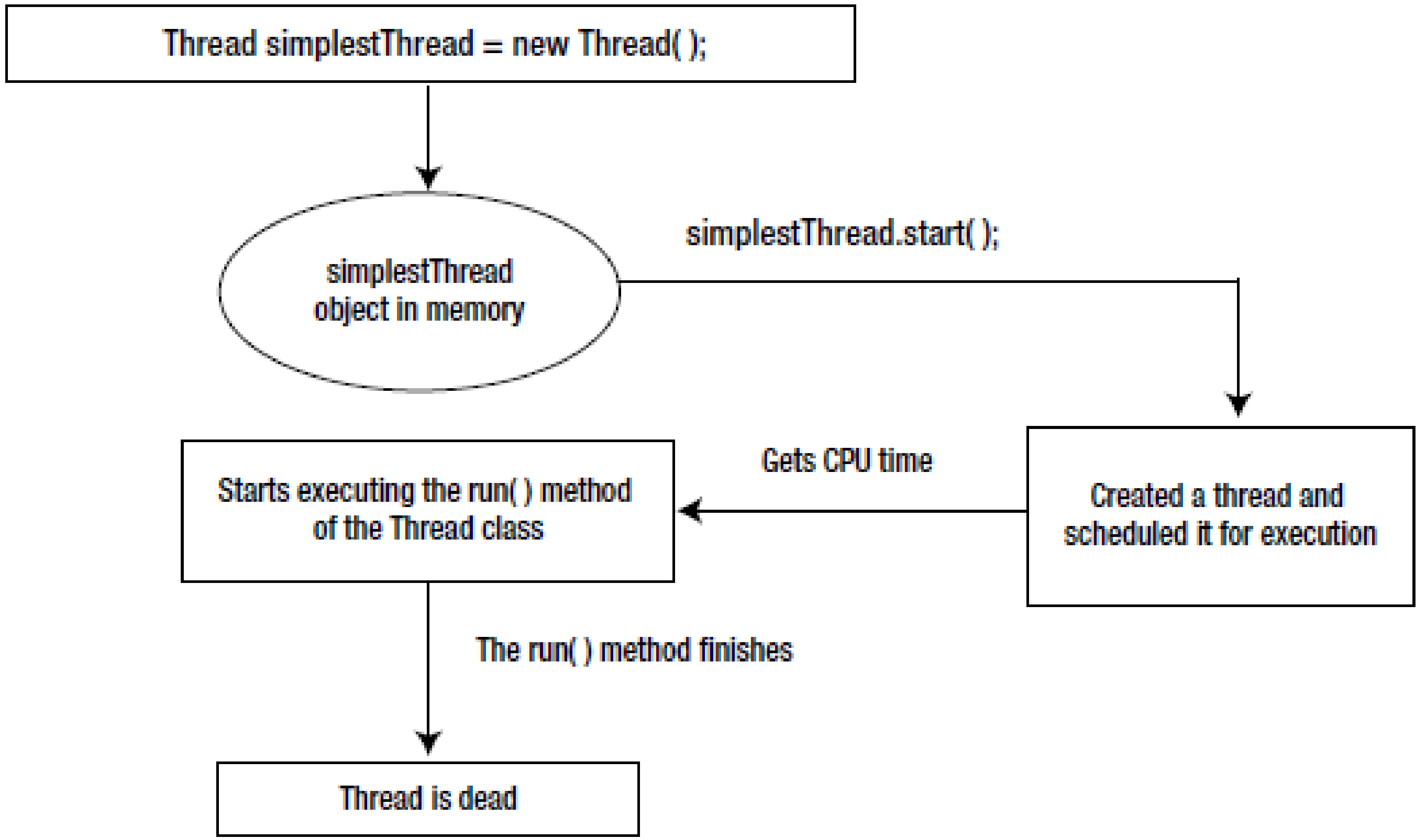
```
@FunctionalInterface
public interface Runnable
{
   void run();
}
```

```
public class RunnableClass implements Runnable{
   public void run(){
      System.out.println("class impl. Runnable");
   }
}
…
Thread myThread = new Thread(new RunnableClass());
myThread.start();
```

Thread versus Runnable

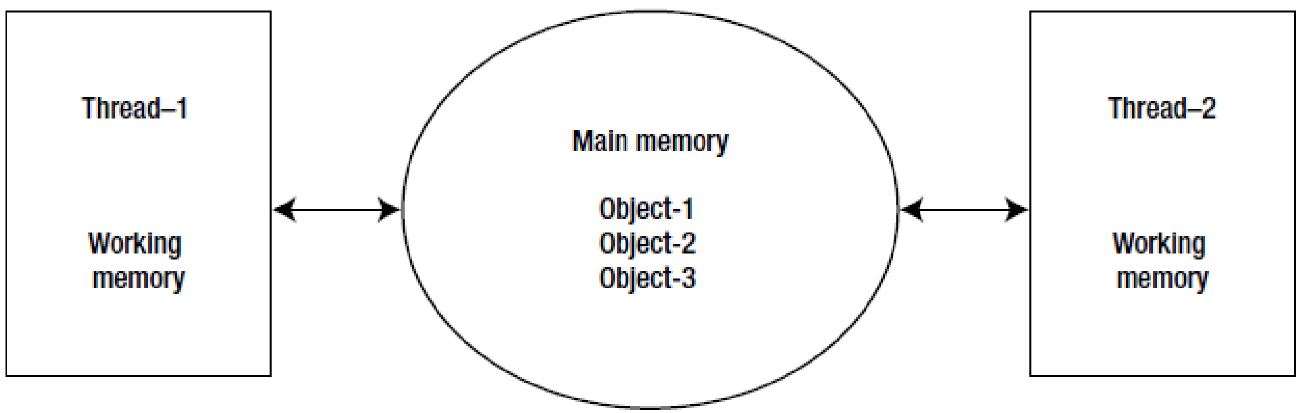
Criteria	Features	
Inheritance	Extend Thread	Cannot extend another class
	Implement Runnable	Can extend another class and can implement other interfaces
Reusability	Extend Thread	Contains both thread and job specific behavior code
	Implement Runnable	Contains only the functionality we want in the run method
Classes vs interfaces	Extend Thread	Defines the core identity of the new class
	Implement Runnable	Describes some abilities of the new class
Coupling	Extend Thread	Tight coupling
	Implement Runnable	Loose coupling – the code is split in 2 parts: behavior and thread
Overhead	Extend Thread	Additional overhead because of inheritance
	Implement Runnable	-

Creating a Thread in Java



Java Memory Model

- How, when and in what order program variables are stored to and read from the main memory
  - Each thread has a working memory



Atomicity | **Visibility** | **Ordering**

Thread Methods

- Thread sleep
  - For a specified duration
  - The current thread will be put in the wait state

Exception thrown in case the current thread is interrupted by another thread while *sleep* is active.

```
public class App{
   public static void printMessages() throws InterruptedException{
      for (int i=0; i< 10; i++){
         System.out.println( "Sending message number " + i);
         Thread.sleep(4000);
         }
     }

   public static void main( String[] args ) throws InterruptedException{
      App.printMessages();
     }
}
```

The time for suspending the execution of a running thread must be given in milliseconds and must be a positive number.

Threads Methods

- **Thread Interrupt**

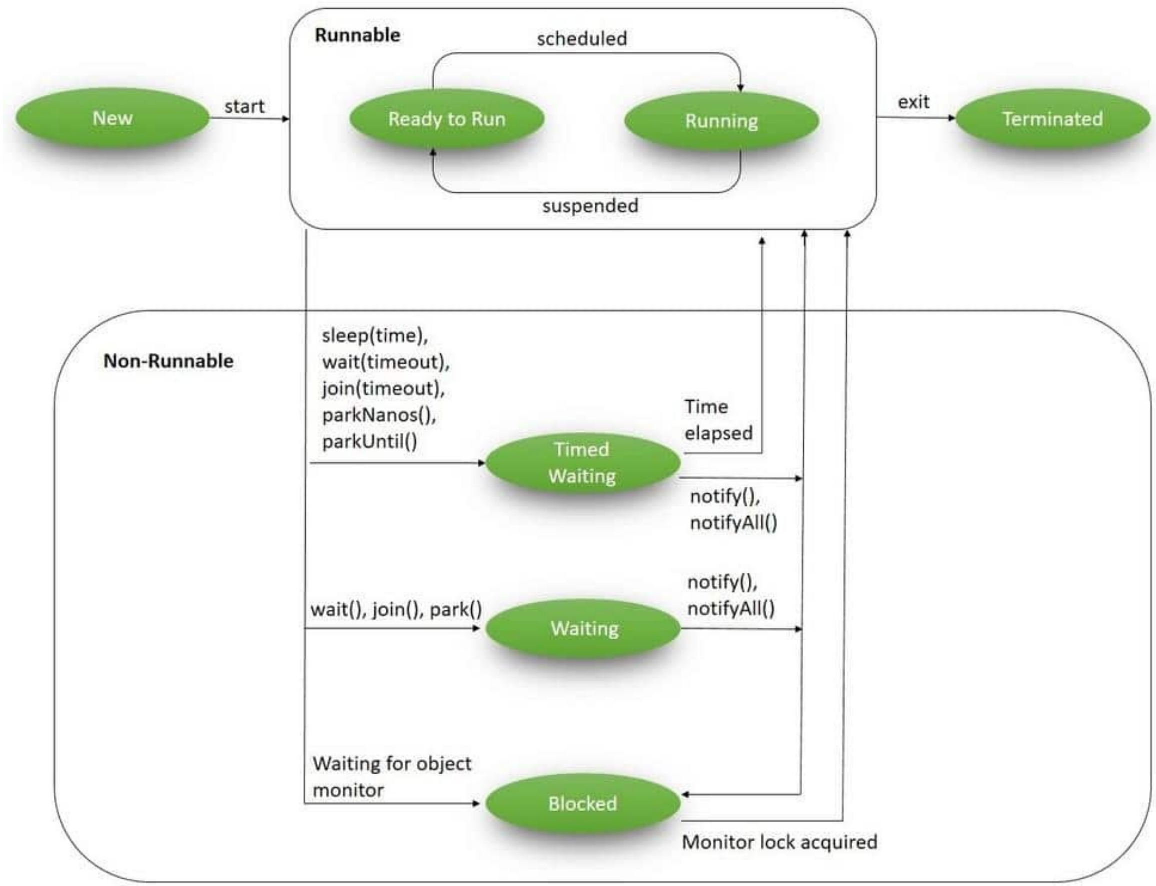
Invoking methods that throw InterruptedException

```
public class MyThread implements Runnable{
  public void run() {
    for (int i=0; i< 10; i++){
      System.out.println("Sending "+i);
      try {
        Thread.sleep(4000);
      } catch (InterruptedException e) {
       System.out.println("Interrupted!");
       return;
       }
    }
  }
}
…
Thread thread = new Thread(new MyThread());
thread.start();  thread.interrupt();
```

Not invoking methods that throw InterruptedException

```
public class MyThread implements Runnable{
  public void run() {
    for (int i=0; i< 10; i++){
       System.out.println("Sending "+i);
       if(Thread.interrupted()){
         System.out.println("Interrupted!");
         return;
       }
    }
  }
}
…
 Thread thread = new Thread(new MyThread());
 thread.start();  thread.interrupt();
```

Threads - Lifecycle



(From https://www.baeldung.com/java-thread-lifecycle)

Threads using Timers

Steps for scheduling a task using Timer

Step 1: Create a subclass of the **TimerTask** class and override the **run** method

Step 2: Create a thread using the **Timer** class.

- background thread that will execute the timer's tasks sequentially

Step 3: Create an object of the subclass created at Step 1.

Step 4: Plan the execution using the **schedule method**

```
//Step 1
public class SendingMessageTask extends TimerTask {
  private String message;
  public SendingMessageAction(String aMessage){
    this.message = aMessage;
  }

  @Override
  public void run() {
    System.out.println("Sending "+this.message);
  }
}
…
//Step 2
Timer aTimer = new Timer();
//Step 3
SendingMessageTask sendingMessageTask = new
                       SendingMessageTask("Hello");
// Step 4 –repeated fixed-delay
aTimer.schedule(sendingMessageTask, 1000, 2000);
```

Threads Issues

- Safety hazards

Shared resource

```
public class Buffer {
    private int number = -1;
    public int getNumber() { return number; }
    public void setNumber(int number) { this.number = number; }
}
```

The threads

```
public class Producer extends Thread{
    private Buffer buffer;
    public Producer(Buffer buffer){
        this.buffer = buffer;
    }
    public void run(){
     for(int i=0; i < 10; i++){
       buffer.setNumber(i);
       System.out.println("Producer set:"+i);
       try{
          sleep(1000);
       }catch (InterruptedException e) {
          e.printStackTrace();
        }
     }
    }
}
```

```
public class Consumer extends Thread{
  private Buffer buffer;
  public Consumer(Buffer buffer){
    this.buffer = buffer;
  }
  public void run(){
    int value = 0;
    for(int i=0; i<10; i++){
      value = buffer.getNumber();
      System.out.println("Consumer
received:"+value);
     }
    }
}
```

```
Producer set:0
Consumer received:0
Consumer received:0
Consumer received:0
Consumer received:0
Consumer received:0
Consumer received:0
Consumer received:0
Consumer received:0
Consumer received:0
Consumer received:0
Producer set:1
Producer set:2
Producer set:3
Producer set:4
Producer set:5
Producer set:6
Producer set:7
Producer set:8
Producer set:9
```

Not the expected result!

Source: C. Frasinaru, Curs practic de Java, Matrix ROM.

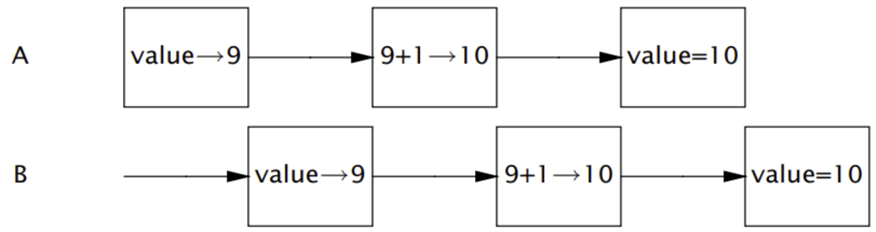
Threads Issues

- Liveness problems
  - Deadlock
  - Starvation
- Thread interference
- Memory inconsistency

```
@NotThreadSafe
public class UnsafeSequence {
    private int value;

    /** Returns a unique value. */
    public int getNext() {
        return value++;
    }
}
```

Listing 1.1. Non-thread-safe sequence generator.



Source: Java concurrency in practice

Thread Safety

- Writing thread-safe code is about managing access to **shared mutable state**
  - Mutable state variable without appropriate synchronization => **broken program**
  - Solutions
    1) Don't share the state variable across threads
    2) Make the state variable immutable,
    3) Use synchronization whenever accessing the state variable

  - Use synchronization (*volatile variables, synchronized keyword, explicit locks and atomic variables)* to coordinate access

When designing thread-safe classes, good object-oriented techniques (i.e., encapsulation, immutability and clear specification of invariants) are your best friends!

Thread Safety

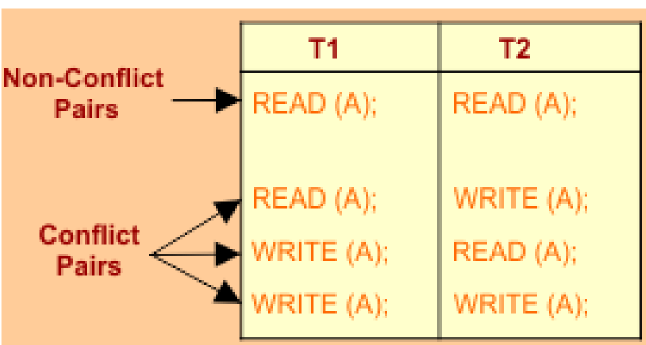
- Thread-safe class
  - It behaves correctly when accessed from multiple threads,
  - Encapsulate any needed synchronization
  - **Stateless objects** are always thread-safe!
  - Java classes that are thread safe

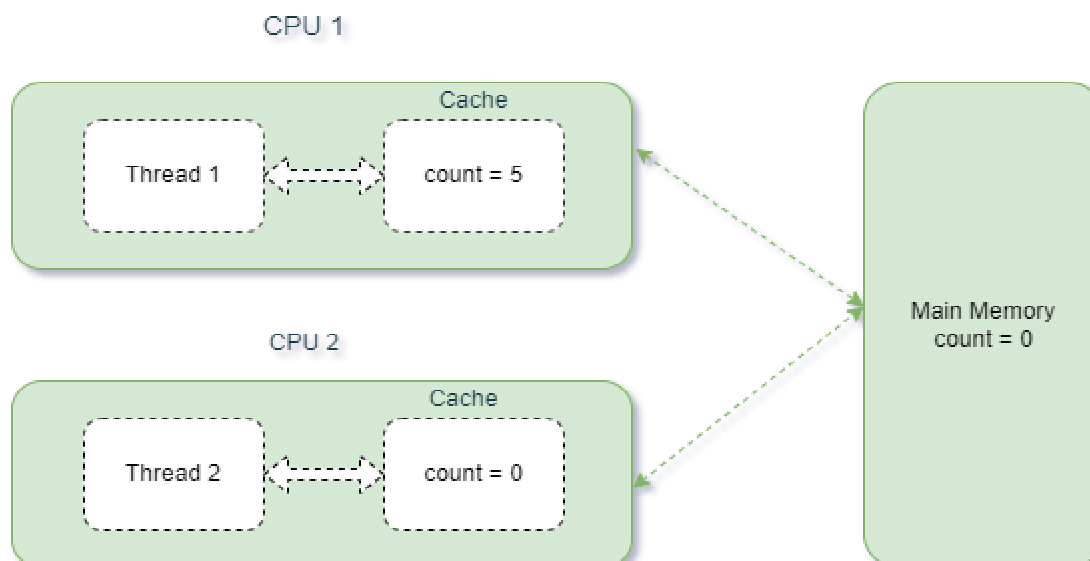
```
public class TransactionManager {
    public double executeTransaction(Transaction transaction){
        ...
    }
}
```

Thread Safety

- Atomicity
  - Atomic action cannot be interleaved => avoids thread interference
  - java.util.concurrent.atomic package
  - Reads and writes are atomic for reference variables and for most primitive variables

Reads and writes are atomic for all variables declared **volatile**





Working with memory don't happens instantly

Source

Thread Safety

- Atomicity

Compound Operations

```
int i=0;
i++; // Get I value & add one to it
/*Accessed simultaneously by both Th1 and Th2
Can lead to inconsistencies:
-  result can be 1(both threads got 0 and
   incremented to 1)
-  result can be 2(second thread got the value
   1 incremented by the first thread)
*/
```

Atomic Operations

```
AtomicInteger i= new AtomicInteger();
i.getAndIncrement ();
```

i++ is not atomic!

- *read-modify-write* operation
- not stateless and is not thread-safe due to instance variable

Race conditions

Volatile variables

- Changes are always visible to other threads
- Establishes a happens-before relationship with subsequent reads of that same variable
- Sees also the side effects of the code that led up the change

Thread Safety

- Locking - synchronized statements
  - To preserve state consistency, update state variables in a single atomic operation!

Block of code to be guarded by the lock

```
synchronized (lock) {//Reference to an object
    // Access or modify shared state guarded by lock
    …
}
```

- Every Java object can implicitly act as a lock for purposes of synchronization
  - ***intrinsic locks***
  - Automatically acquired and released
  - A happens-before relationship is established

Thread Safety

- Locking - synchronized blocks
  - User defined locks can be used to create synchronized code
  - Synchronized methods can have problems with liveness.

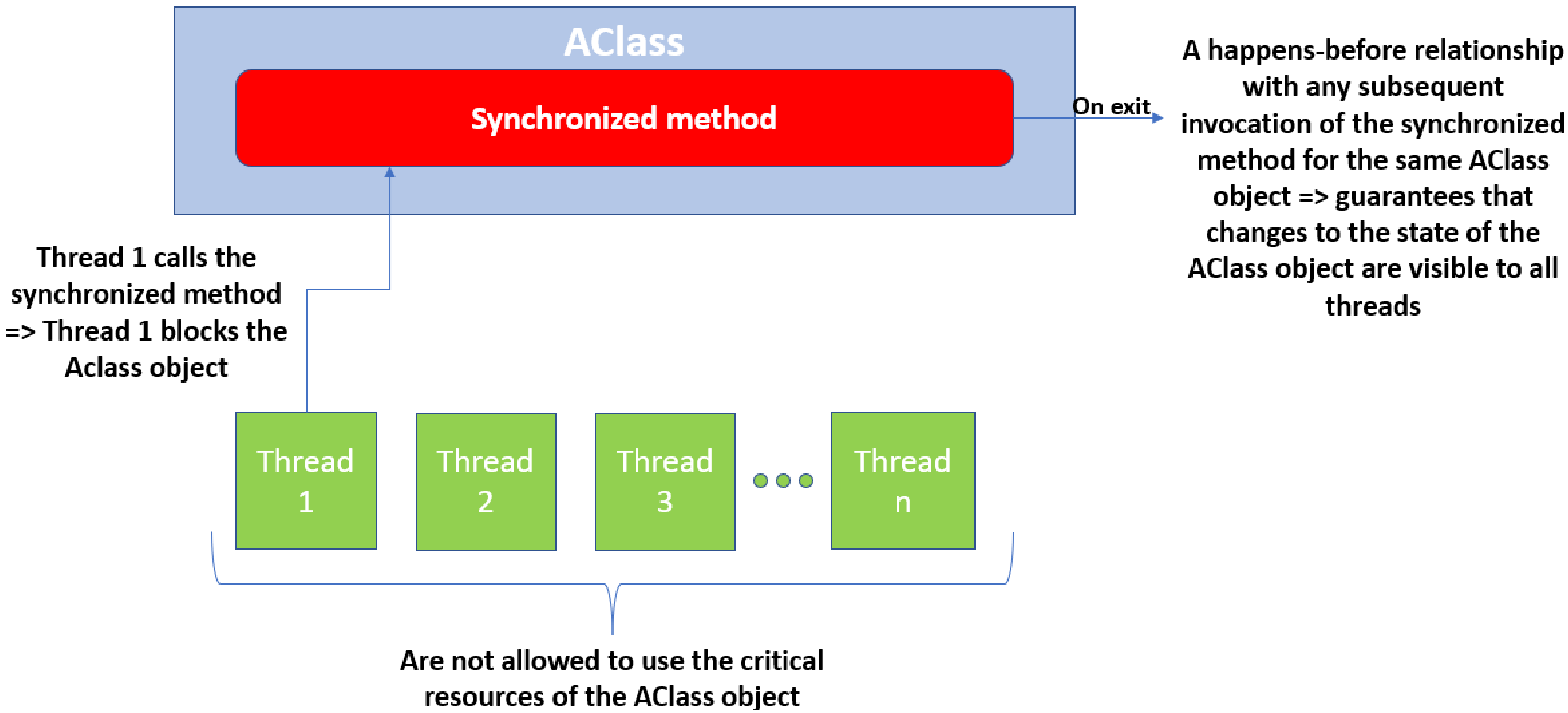
```
public void transfer(Account a,
             Account b, double sum){
  synchronized(a){// Th1 locks acc. a
     //Th2 locks account b
    synchronized(b){
    //transfer sum
    }
  }
}
…
Call function :
Th1: transfer(a,b,sum1);
Th2: transfer(b,a,sum1);
```

Deadlock situation!!!

```
synchronized(MyClass.class){
   // some code
 }
Or
synchronized(this){
   // some code
 }
```

Thread Safety

- Locking - Synchronized Methods
  - **synchronized** keyword in the methods' declaration
  - **Constructors cannot be synchronized**



Thread Safety

- Locking - Inter-Thread communication
  - A way by which synchronized threads can communicate
  - **wait()** – makes a thread to wait
    - The thread releases the lock associated to the object and wait
    - wake up -> use **notifyAll()** or **notify() or** waiting time has expired

Wait()	**Sleep()**
Wait() method belongs to Object class.	Sleep() method belongs to Thread class.
Wait() method releases lock during Synchronization.	Sleep() method does not release the lock on object during Synchronization.
Wait() should be called only from Synchronized context.	There is no need to call sleep() from Synchronized context.
Wait() is not a static method.	Sleep() is a static method.

Source

Thread Safety

- Locking - Synchronized Methods and Threads Coordination

A Solution for the Producer-Consumer Problem

Critical section

Critical section

```java
public class Buffer {
  private int number = -1;
  private boolean available = false;
  public synchronized int get(){
    while(!available){
      try{ wait(); }catch(InterruptedException ex){
        ex.printStackTrace();
      }
    }
    available = false;
    notifyAll();
    return number;
  }
  public synchronized void put(int number){
    while(available){
      try{ wait(); }catch(InterruptedException ex){
        ex.printStackTrace();
      }
    }
    this.number = number;
    available = true;
    notifyAll();
  }
}
```

Source: C. Frasinaru, Curs practic de Java, Matrix ROM.

Expected result

```
Consumer received:0
Producer set:0
Producer set:1
Consumer received:1
Producer set:2
Consumer received:2
Producer set:3
Consumer received:3
Producer set:4
Consumer received:4
Consumer received:5
Producer set:5
Producer set:6
Consumer received:6
Consumer received:7
Producer set:7
Producer set:8
Consumer received:8
Producer set:9
Consumer received:9
```

Thread Safety

- Immutable Objects
  - State cannot change after they are constructed
  - require a copy object for each distinct value
    - Make fields final and private
    - No “set” methods
    - Do not allow subclasses to override methods
    - Attention to ‘get’ methods on mutable instance fields

Immutable objects are thread safe

```
public class Employee{
   private final String employeeID;
   private final String firstName;
   private final String lastName;

   //constructor-assigns values to all
   // fields
   public Employee(String id, String first,
                   String last){
     eployeeID = id;
     firstName = first;
     lastName = last;
   }
   public int getEmployeeID() {
      return Integer.parseInt(employeeID);
   }
   // should be removed
   public void setEmployeeID(int id) {
      employeeID = Integer.toString(id);
   }
   … }
```

Thread Safety

- Thread Safe Collections – Synchronized Collections
  - Synchronization wrappers which create synchronized views of collections
    - *syncronizedCollection*, *synchronizedList*, *synchronizedMap*, etc.

```
List<String> list = Collections.synchronizedList(new ArrayList<String>());
```

For iteration, the collection needs to use external sync

  - Achieve thread-safety through intrinsic locks
  - Synchronized collections are thread safe

Must manually synchronize on the returned collection when iterating over it

```
Collection c = Collections.synchronizedCollection(myCollection);
...
synchronized (c) {
  Iterator i = c.iterator(); // Must be in the
                             // synchronized block
  while (i.hasNext()) foo(i.next());
}
```

Thread Safety

- Thread Safe Collections – Concurrent Collections
  - Designed for concurrent accesses from multiple threads
  - *java.util.concurrent package: BlockingQueue, ConcurrentHashMap, ConcurrentNavigableMap, CopyOnWriteArrayList*
  - Achieve thread-safety
    - **BlockingQueue** – provides blocking put and take methods
      - Support the producer-consumer design patterns
    - **ConcurrentHashMap** - divides its data into segments
      - Different threads can acquire locks on each segment
      - Multiple threads can access the map at the same time
    - **CopyOnWriteArrayList** - creates a separate copy of List for each write operation

Are much more performant than synchronized collections

Executor Framework

- Executors and Interfaces
  - Separates thread management and creation from the rest of application
  - Executors - objects encapsulating thread management and creation
  - *java.util.concurrent* package interfaces
    - ***Executor*** – supports launching new tasks – method **execute**
    - ***ExecutorService*** - adds method **submit** on Callable objects
    - ***ScheduledExecutorService*** - supports future and/or periodic execution of tasks

```
Executor exec= Executors.newFixedThreadPool(100);
Runnable task = new Runnable() {
      public void run() {
           //execute job
       }
};
exec.execute(task);
```

you can replace
new Thread(r)).start();
With: e.execute(r);

- A Callable does return a result and can throw a checked exception.
- Future object

Executor Framework

- Result Bearing Jobs
  - **Runnable** – suitable when we are not looking for a thread execution result
  - **Callable** – suitable when we are looking for a thread execution result

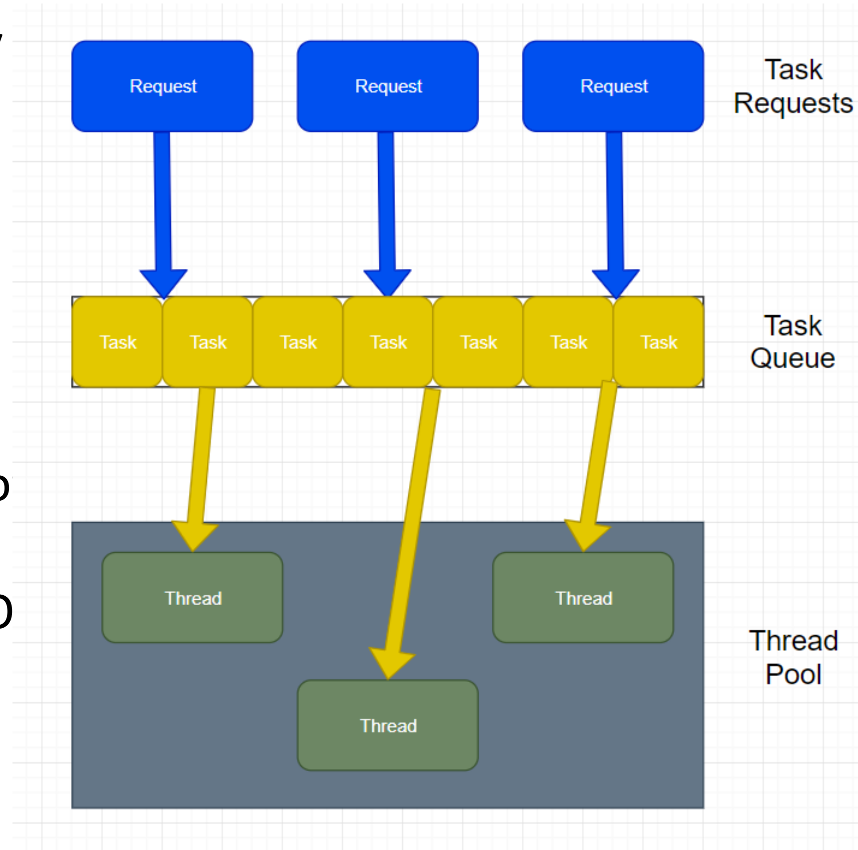
```
public class FactorialTask implements Callable<Integer>{
 int number;
 public Integer call() throws InvalidParamaterException{
   int fact = 1;
   // ...
   for(int count = number; count > 1; count--) {
     fact = fact * count;
   }
   return fact;
 } }
```

The result of submitting a Callable or a Runnable task to the Executor Service

```
public void whenTaskSubmitted_ThenFutureResultObtained(){
    FactorialTask task = new FactorialTask(5);
    Future<Integer> future = executorService.submit(task);
     assertEquals(120, future.get().intValue());
}
```

Executor Framework

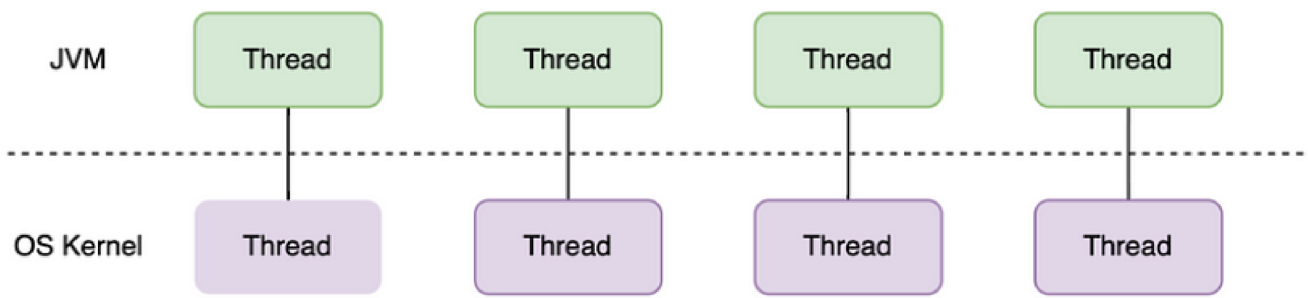
- Thread Pools
  - worker threads which exist separately from the Runnable and Callable tasks they execute
  - Minimize the overhead due to thread creation
  - Methods for creating executors that use thread pools
    - newFixedThreadPool - fixed size o threads
    - Tasks processed sequentially (FIFO LIFO, priority, etc.)
    - newScheduledThreadPool – fixed sized; supports delayed and periodic execution



Source

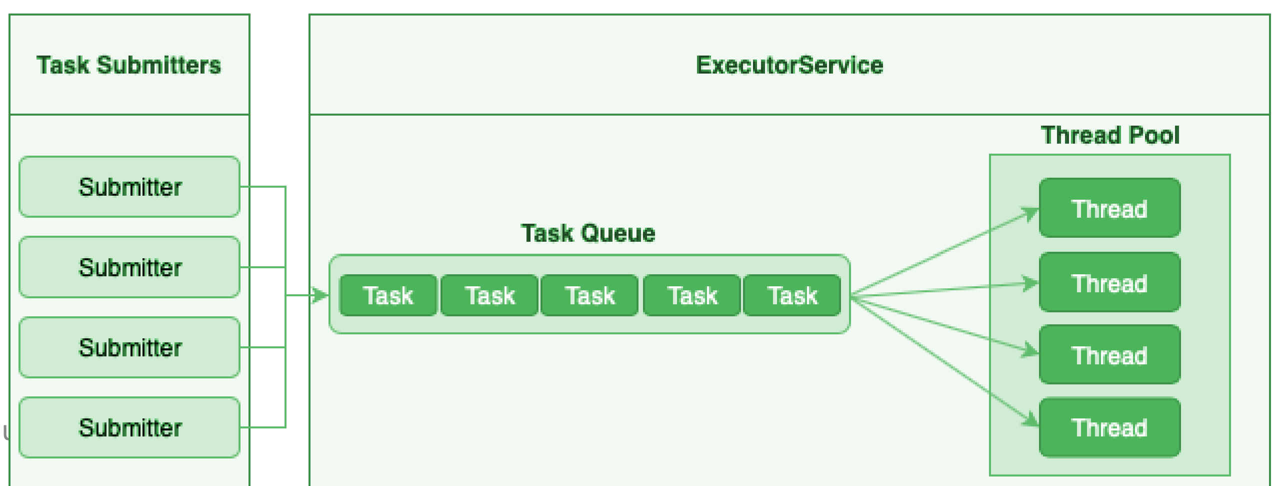
Virtual Threads In Java 19

- Java threads limitations
  - Threads of the machine are expensive, and the number is limited
    - Limiting factor long before other resources, such as CPU or network connections, are exhausted



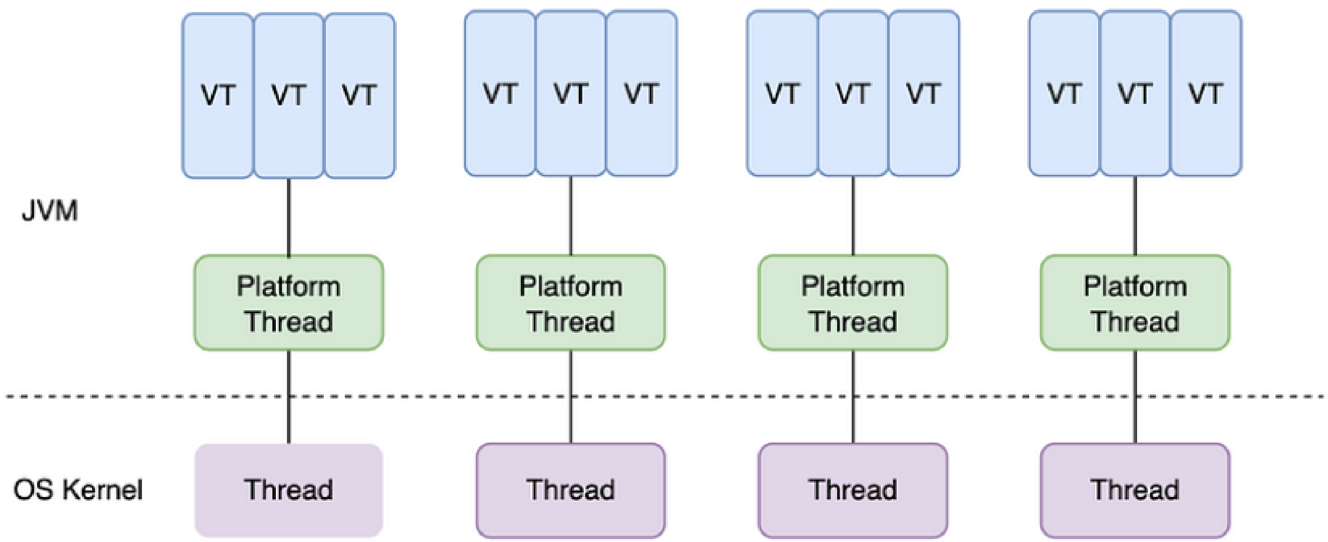
Sources:
https://medium.com/javarevisited/how-to-use-java-19-virtual-threads-c16a32bad5f7

https://stackoverflow.com/questions/65189805/java-thread-pool-task-execution-queueing



Virtual Threads In Java 19

- **Virtual thread**
  - Requires an OS thread to do CPU work, but doesn't hold the OS thread while waiting for other resources
  - Have minimal overhead, so there can be many of them.
  - Support thread-local variables, synchronized blocks, and thread interruption



Source: https://blogs.oracle.com/javamagazine/post/java-loom-virtual-threads-platform-threads

Virtual Threads In Java 19

- **Virtual thread**
  - Managed by the JVM, similarity by design
  - Free of the system's context switch
  - They don't block the carrier thread thus blocking is a much cheaper

```
Thread.startVirtualThread(Runnable r) -replacement for calling thread.start()
```

```
try {
      Future future1 = Executors.newVirtualThreadPerTaskExecutor().submit(() -> fetchURL(url1));
   } catch (ExecutionException | InterruptedException e) {
      response.fail(e);}
```

- Conventional threads - the application code is responsible for provisioning and dispensing OS resources.
- Virtual threads - JVM obtains and releases the resources from the operating system.
- The Java runtime arranges for it to run by *mounting* it on some platform thread, called a *carrier thread*.