Secure OOP with Java

Lecture Unit - 10

Claudia Maderthaner <claudia.maderthaner@fh-hagenberg.at>

Concurrency

Process

- A process refers to a program in execution.
- A process is a self-contained execution environment.
- Each process has a complete and private set of basic runtime resources.
- Each process has its own memory space.
- A process may have many threads running.
- The JVM runs as a process.

Thread

- A thread may be described as a "lightweight" process.
- Every thread exists within a process.
- Threads require fewer resources.
- Threads can share the process' resources like memory and open files.
- Thread creation, termination and switching take less time.
- Communication between threads is more efficient.

Multithreading

- Concurrent execution of two or more parts of a program
- Each part can handle a different task at the same time
- Maximize utilization of CPU
- The Java runtime is inherently multithreaded (e.g. the garbage collector runs in a thread)

Concurrency in Java

> In fact, the subject of concurrency is a very large one, and good multithreaded development is difficult and continues to cause problems for even the best developers with years of experience under their belts.

— The Well-Grounded Java Developer

Java’s Threading Model

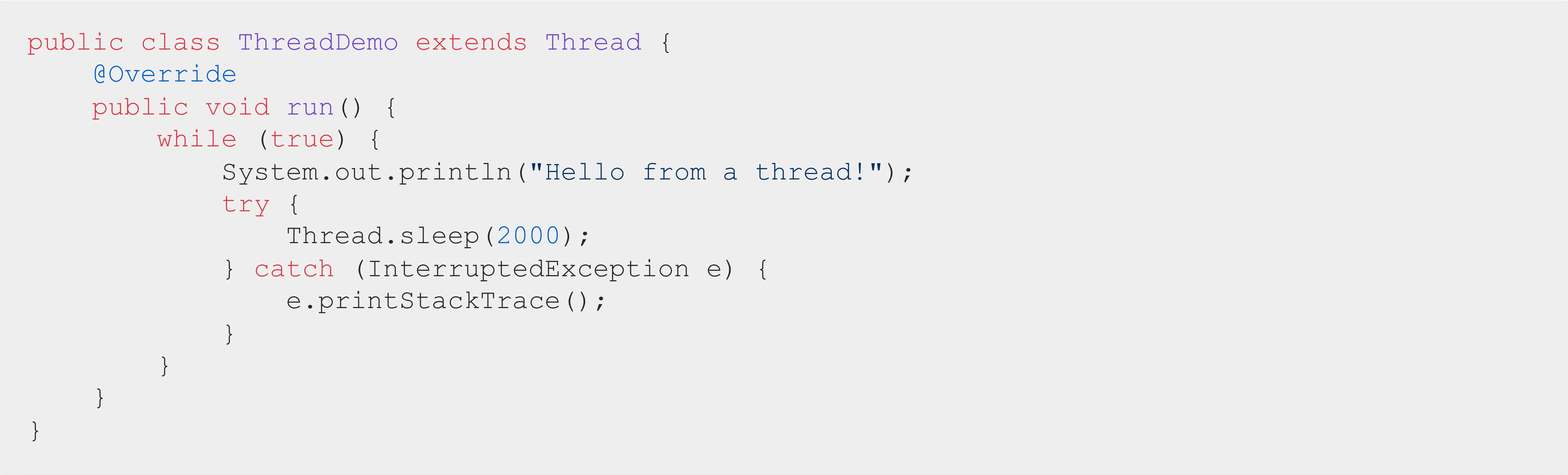
- Shared, visible-by-default mutable state
- Pre-emptive thread scheduling by the operating system

Use Cases for Concurrency

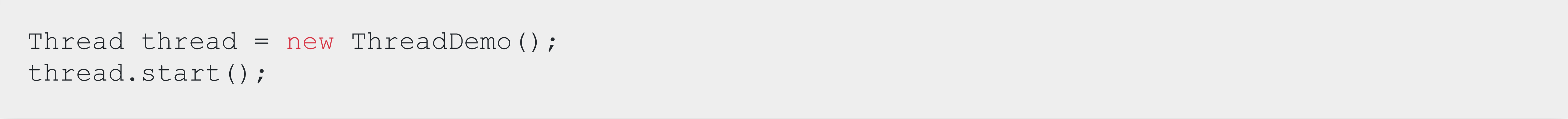
- Nonblocking I/O
- Alarms and timers
- Independent tasks
- Parallelizable algorithms

java.lang.Thread

- Base class for all threads in Java



```java
public class ThreadDemo extends Thread {
    @Override
    public void run() {
        while (true) {
            System.out.println("Hello from a thread!");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```



```java
Thread thread = new ThreadDemo();
thread.start();
```

Thread Properties

Name

the thread name may be set with the method `setName`; the thread name is useful for thread dumps

Priorities

- if the priority is not explicitly set the priority of the creating thread is used
- whenever the thread scheduler picks a new thread, it prefers threads with higher priority (but this feature is highly system dependent)

Group

represents a set of threads and/or other thread groups

```
ThreadGroup workers = new ThreadGroud("Miracle-Workers");
Thread worker1 = new Thread(workers, new Task());
```

java.lang.Runnable

```java
public class RunnableDemo implements Runnable {
    @Override
    public void run() {
        System.out.println("Hello from a thread!");
    }
}
```

```java
Runnable runnable = new RunnableDemo();
Thread thread = new Thread(runnable);
thread.start();
```

Block-Structured Concurrency

Using Threads

start()

```java
Thread thread = new ThreadDemo();
thread.start();
```

```java
Runnable runnable = new RunnableDemo();
Thread thread = new Thread(runnable);
thread.start();
```

Do not call the `run` method of the `Thread` or `Runnable` instance.

A direct call merely executes the task in the **same** thread.

Thread.sleep()

- Causes the currently executing thread to temporarily cease execution for the specified number of milliseconds.
- The thread does not lose ownership of any monitors.

```
try {
    Thread.sleep(10_000);
} catch (InterruptedException e) {
    // something happend while I was asleep
}
```

Thread.yield()

- A hint to the scheduler that the current thread is willing to yield its current use of the processor.
- The scheduler may ignore the hint.

Stopping Threads

```java
public void run() {
    System.out.println("Hello from a thread!");
}
```

```java
public void run() {
    while(true) {
        System.out.println("Hello from a thread!");
    }
}
```

stop()

- The use of `stop()` is inherently unsafe.
- All monitors that the thread has locked are unlocked.
- This may lead to an inconsistent state.
- The damaged object becomes visible to other threads.

> ⚠ **Do not use `stop()`.** It is deprecated and must not be used in any new code.

Suspending and Resuming

- `suspend()` and `resume()` suffer from the same problems as the `stop()` method.
- They are also deprecated and not to be used anymore.

Termination Flag

- Some thread internal flag that signals the thread should stop
- The `run()` method checks the flag state periodically (e. g. on every loop)

Interrupts

- Indication to a thread that it should stop
- Does complete blocking methods immediately

`interrupt()`

  - causes any blocked method to throw an `InterruptedException`
  - sets the interrupt flag on a thread object

`isInterrupted()`

returns the interrupted status of the thread

`Thread.interrupted()`

returns the interrupted status of the current thread; also clears the interrupted status

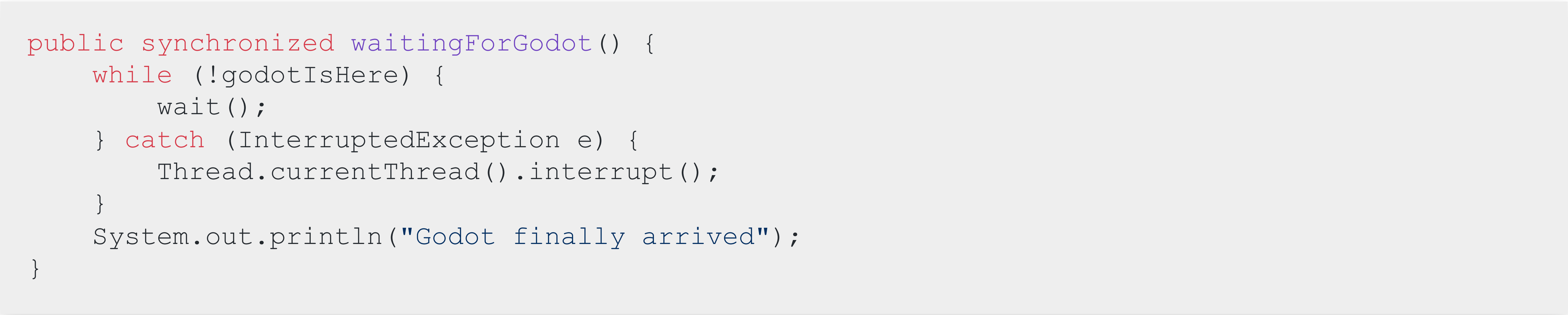
Threads and Exceptions

- Uncaught exceptions thrown in a thread will terminate this thread
- An `UncaughtExceptionHandler` may be used to add logging or some other handling.

Co-operating Threads

wait()

- The current thread must own the objects monitor.
- `wait()` releases the lock on the objects monitor.
- Forces the current thread to wait until some other thread calls `notify()` or `notifyAll()`.



```java
public synchronized waitingForGodot() {
    while (!godotIsHere) {
        wait();
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    System.out.println("Godot finally arrived");
}
```

notify()/notifyAll()

- Wakes up threads waiting for access to this object’s monitor.
- `notify()` wakes up a single arbitrary thread
- `notifyAll()` wakes up all threads waiting for this object’s monitor.

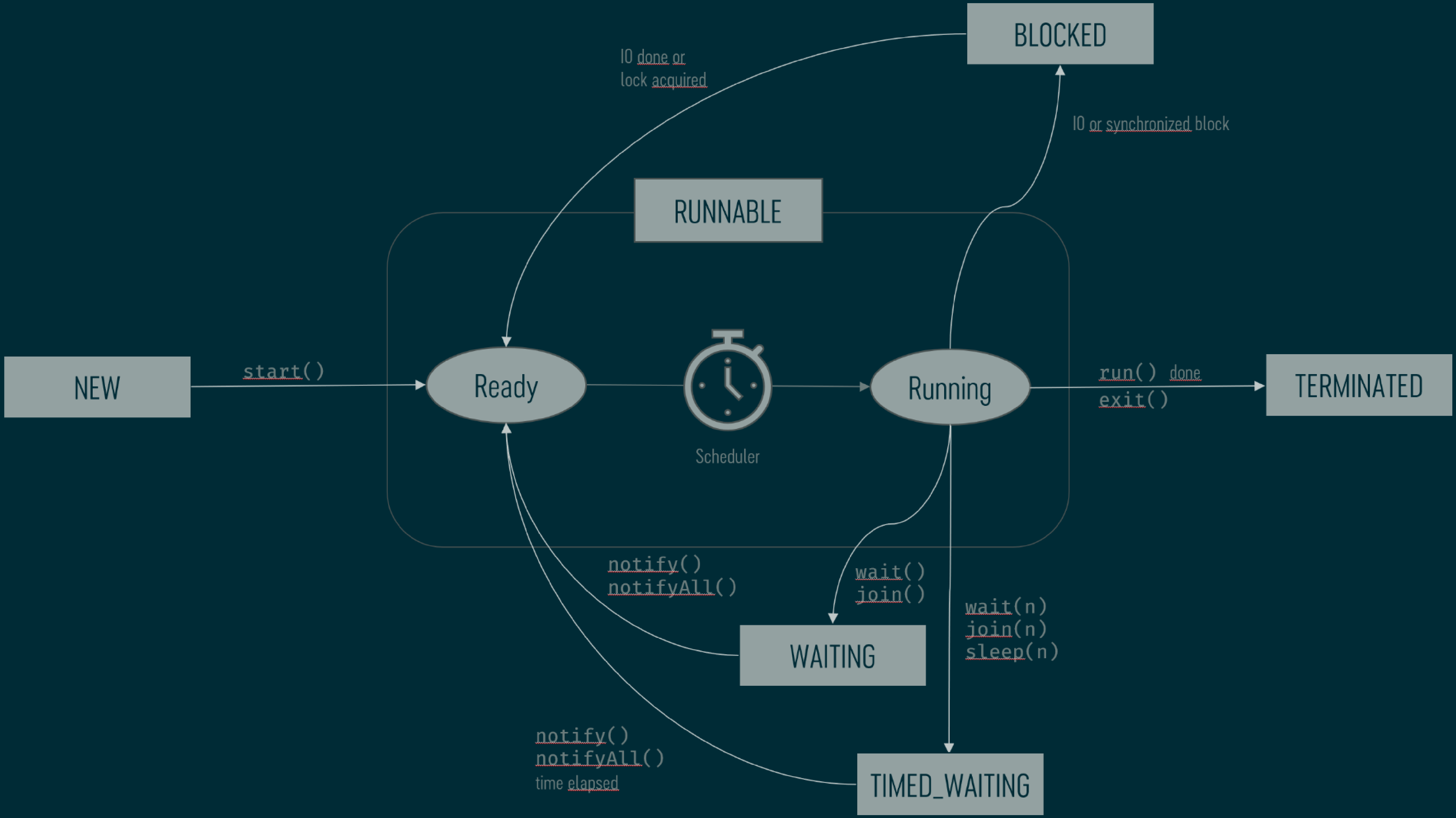
```
public synchronized godotsArrival() {
    godotIsHere = true;
    notifyAll();
}
```

join()

- Waits for this thread to finish.

Thread States

NEW	A thread that has not yet started.
RUNNABLE	A thread executing in the Java Virtual Machine.
BLOCKED	A thread that is blocked waiting for a monitor lock.
WAITING	A thread that is waiting indefinitely for another thread to perform a particular action.
TIMED_WAITING	A thread that is waiting for a another thread to perform an action for up to a specified waiting time.
TERMINATED	A thread that has exited.


BLOCKED
IO done or
lock acquired
IO or synchronized block
RUNNABLE
NEW
start()
Ready
Scheduler
Running
run() done
exit()
TERMINATED
notify()
notifyAll()
wait()
join()
WAITING
wait(n)
join(n)
sleep(n)
notify()
notifyAll()
time elapsed
TIMED_WAITING

main Thread

- When the JVM starts, a thread with the name `main` is created.
- This thread executes the `main()` method.

Daemon Threads

- Background process, not under direct control of the user.
- There daemon threads exit immediately afte the last non-daemon thread exits

Synchronization

> **If multiple threads access the same mutable state variable without appropriate synchronization, your program is broken.**

2006

— Java Concurrency in Practice

Problem Definition

- Thread interference
- Memory consistency

Interference

Transfer of money

Thread A	**Thread B**
read balance 100	
	read balance 100
add 10	
	remove 10
save 110	
	save 90

```java
public class Counter {
    private int counter = 0;

    public void increment() {
        counter++;
    }

    public void decrement() {
        counter--;
    }

    public int value() {
        return counter;
    }
}
```

Memory Consistency

- Threads my hold memory values in local caches.
- Compilers may reorder instructions for maximum throughput.
- When one thread changes a property value, other threads may not see the change immediately.
- The `volatile` keyword guarantees its read/write visibility between threads

Concurrency Issues

- Deadlock
- Race Condition
- Starvation
- Livelock

Monitor

aka Lock Object

- Every object in Java may be used as a monitor.
- This monitors combine a lock (mutual exclusion) with the ability to wait for a certain condition.
- A lock
  - protects sections of code, allowing only one thread to execute the code at a time
  - manages threads that are trying to enter a protected code segment

Synchronized Methods

```
public synchronized increase() {
    counter++;
}
```

- Uses the current object as lock object.
- A thread may only proceed when it gets the lock.

Synchronized Static Methods

```
public static synchronized increase() {
    counter++;
}
```

- Uses the class object of the class as lock object.

Synchronized Blocks

```
private Object lock = new Object();

public increase() {
    if (counter < MAXIMUM) {
        synchronized (lock) {
            counter++;
        }
    }
}
```

Reentrant Synchronization

- A lock object may only belong to one thread.
- But thread may request a lock object multiple times.
- Therefore, the thread does not block itself.

Atomic Operations

- Read/write of
  - reference type variables
  - primitive data types (except `long` and `double`)
  - fields marked as `volatile`

Immutability

- It is safe to access shared fields when they are declared `final`.

```
final String name = "James Gosling";
```

- If the referenced object is mutable it is still necessary to synchronize the operations.

```
final Person person = new Person("James Gosling");
```

Contact

Moodle Discussion Board

claudia.maderthaner@fh-hagenberg.at