



# *Threads and Synchronization*

Unit 10



# ***Unit Objectives***

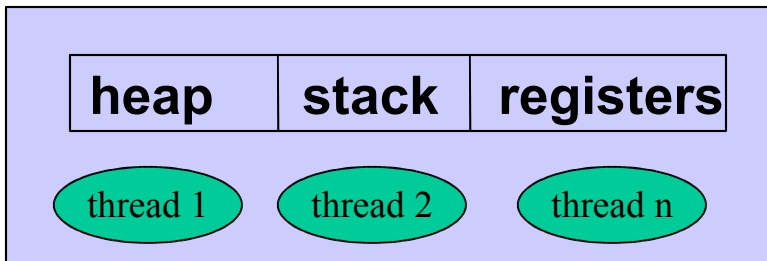
---

- Understand processes and threads
- How to implement threads
- The thread life-cycle
- Synchronization

# Processes and Threads

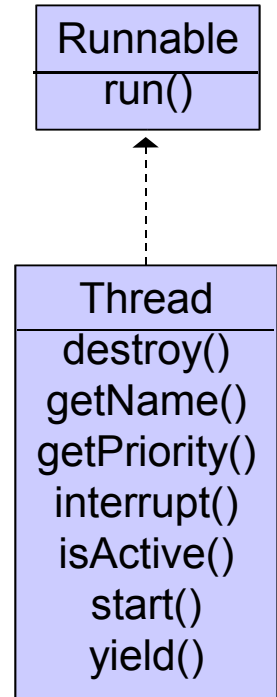
---

- On most computer systems, simple programs are run as processes
- A **process** encapsulates many things including:
  - Address space
  - Current state of the computation
    - Machine registers, call stack, etc
- **Threads** are the set of steps or actions a program executes
  - They allow multiple actions to occur at once
  - Threads do not carry all the process information data



# Threads in Java

- The `java.lang.Thread` class is the base for all objects that can behave as threads
- A Thread needs a handle to an object (Runnable object), whose `run()` method it will "execute"
- There are two ways to use the **Thread** class:
  - Subclassing Thread and overriding the `run()` method
    - The Thread itself is a "Runnable"
  - Implementing **Runnable** and starting new activities by wrapping them inside threads created with the `Thread` (Runnable) constructor



## Using the Thread Class: Subclassing

- The class `TrafficLight` extends the `Thread` class and overrides the method `run()`:

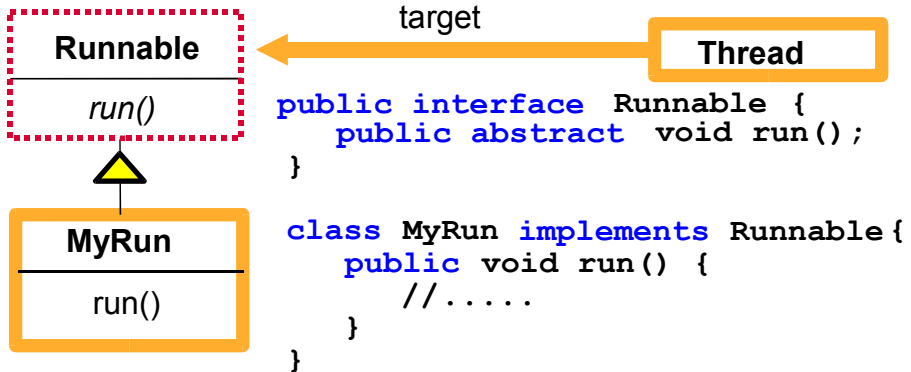
```
class TrafficLight extends Thread {  
    public void run() {  
        // loop, change color & sleep  
    }  
}
```

- Run the thread using the inherited `start()` method:

```
...  
TrafficLight t1 = new TrafficLight();  
t1.start();  
...
```

# Using the Thread Class and Runnable

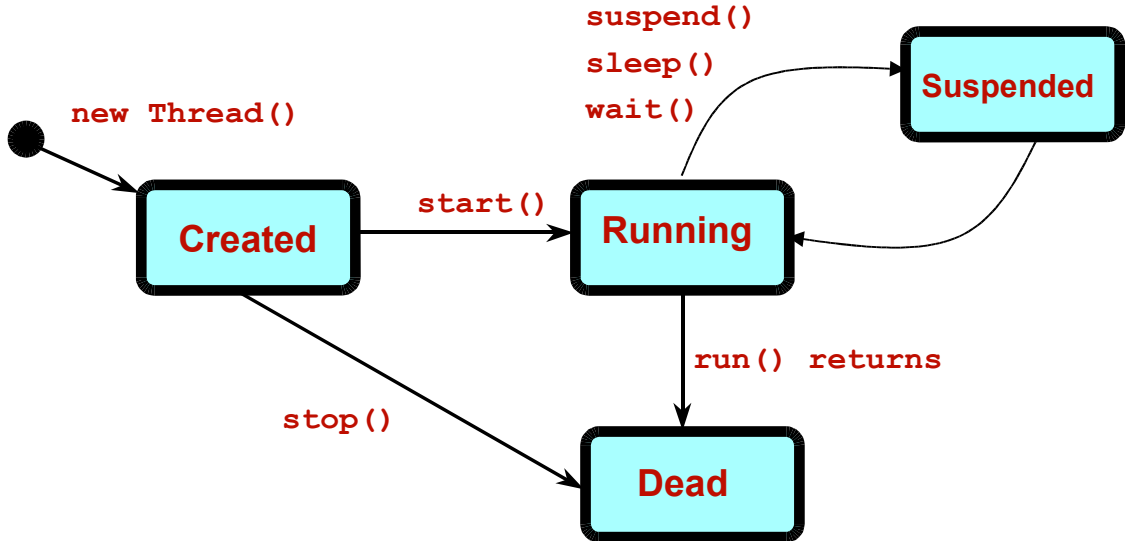
- The class MyRun must provide its implementation for the method `run()`



- Create a new Thread with the Runnable class as a parameter:

```
Thread t = new Thread(new MyRun())  
t.start();
```

# Life-cycle of a Thread



- A thread that is running or suspended is said to be alive
- Once dead, the thread cannot be restarted
  - It can be examined

# Controlling Activities

---

- Threads are meant to be controlled
  - e.g., when to run, when to pause
  - `start()` - starts the thread's `run()` method
  - `sleep()` - halts for a given amount of time
  - `stop()` - deprecated since it was inherently unsafe
  - `destroy()` - kills a thread without any cleanup
  - `resume()`  
and `suspend()` - deprecated for the same reason as `stop()`
  - `yield()` - causes the currently executing thread object to pause temporarily, and allow other threads to execute
  - `interrupt()` - causes any kind of wait to be aborted

# Stopping Threads

---

- When the `run()` method returns, the thread is dead
  - A dead thread cannot be restarted
- A dead Thread object is not destroyed (its data can be accessed)
- Set a field to indicate stop condition and poll it often

```
public void run() {
    stopFlag = false;
    try {
        while (!stopFlag) {.....}
    }
    catch (InterruptedException e) {...}
}
public void finish() {
    stopFlag = true;
    .....
}
```

# Daemon Threads

---

- Daemon Threads are
  - Service threads running in the background
  - Loops waiting on event or input
  - Common ways to implement servers polling sockets
  - Services to other threads
- JVM terminates program when only daemons remain
- Applicable control methods on Thread
  - `isDaemon()`
  - `setDaemon()`

# Multi-threading

---

- In Java, an object can be operated on by multiple threads
- It is the responsibility of the object to protect itself from any possible interference, therefore the `synchronized` keyword is used to provide limited access.
- Objects are locked while their synchronized methods execute
  - No other threads are able to access the object while it is locked

```
public synchronized boolean deduct(float t) {  
    // code block associated  
    // with critical section...  
}
```

# Synchronization

---

- When two threads are accessing the same object simultaneously, changes made by one will not be seen by the other.
- To prevent this from happening, certain methods are `synchronized`, allowing only one thread to access the object at a time.
- Example:
  - A bank account is accessed by a teller to withdraw \$100. If the account is synchronized, no other teller can access it until the first is finished and the account has been updated

# Synchronization

---

- If the synchronized method is static, a lock is obtained at the class level instead of the object level.
  - Only one thread at a time may use such a method
- It is also possible to lock objects for an arbitrary block of code

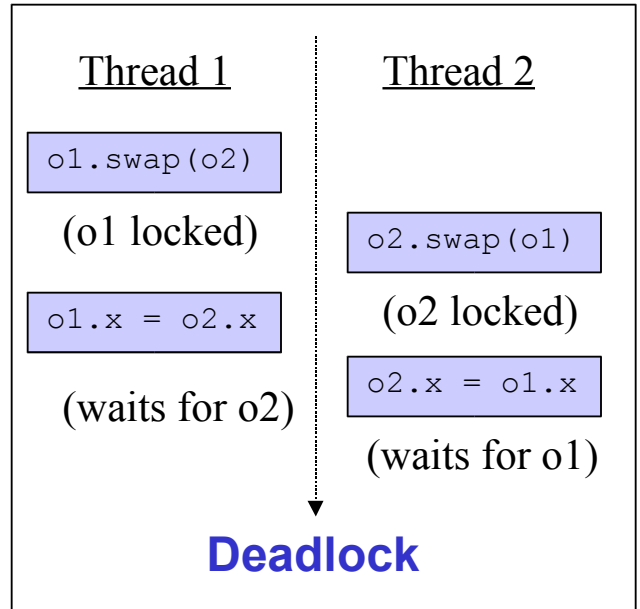
```
// non-critical section  
synchronized (object) {  
    // critical section...  
}  
  
// more non-critical stuff
```

# Synchronization Issues

- Use synchronization sparingly because it can
  - Slow performance by reducing concurrency
  - Sometimes lead to fatal conditions such as *deadlock*

• Other techniques should be used in tandem with synchronization to assure optimal performance

- For example,  
`notifyAll()`  
and `wait()`



# ***What You Have Learned***

---

- How processes and threads work
- That concurrent activity is created in Java through the construction of a Thread
- That a Thread's life begins when it is started, and ends when it:
  - Exits its run method (or its **Runnable**)
  - Is stopped, or
  - Is destroyed
- How to use the `synchronized` keyword