SWEN 601 Software Construction

Threads & Concurrency

Activity: Getting Started

1. Begin by accepting the GitHub Classroom invitation for today's homework.
   a. *The project **may** already contain some code!*
2. Create a `session` package. This is where you will write your solutions to today's activities.
3. Create a `homework` package. This is where you will implement your solution to the homework.

When you submit your homework, you will include your activities. You may earn up to a 10% bonus on the homework if you have completed all of the activities.

Do not submit code that ***does not compile***. Comment it out if necessary.

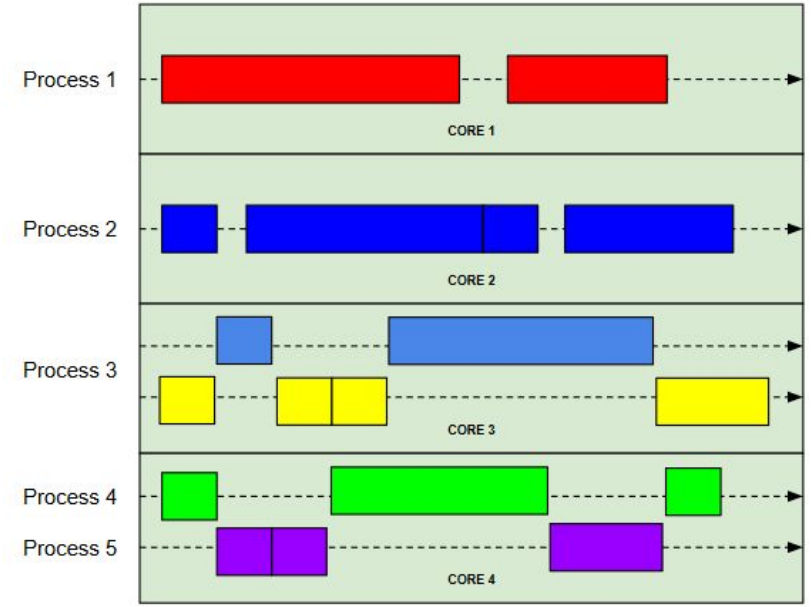
Next Two Weeks

WEEK 13	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #18		Quiz #19		
LECTURE			Exceptions & IO		Threads & Concurrency		
HOMEWORK	Hwk 18 Due (**11:30PM**)		*Hwk 19 Assigned*		*Hwk 20 Assigned*	Hwk 19 Due (**11:30pm**)	

WEEK 14	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #20				
LECTURE			Thread Cooperation		**No Class (Thanksgiving)**		
HOMEWORK	Hwk 20 Due (**11:30pm**)		*Hwk 21 Assigned*				

Processes

- In computing, a ***process*** is a program that is ***currently*** being executed by the operating system.
- A process may have more than one ***thread of execution***; each of which executes ***separate*** code to perform some task.
  - This allows the process to perform ***more than one task*** a time, e.g. playing music and displaying animation at the same time.
- Virtually all computers have ***processors*** with multiple ***cores***, but each core may only execute ***one*** process at a time.
- There are usually many ***more*** processes than cores, and so the cores must be ***shared*** between programs.



Each core in a computer processor can only execute one process at a time.

There are usually many more processes running than there are available cores, and so some processes must share time on the same core.

The Thread Scheduler

- So who or what decides whether or not a thread is given time on one of the available processors?
- A mysterious entity known only as **"The Thread Scheduler"**
- The ***thread scheduler*** works ***differently*** on ***different*** architectures (operating systems, processors, etc.).
- Its behavior is a ***black box***, and somewhat ***unpredictable***.
  - Your threads are in ***competition*** with all of the other threads running on the computer.
  - There is ***no way*** to guarantee that one of your threads gets time on the processor.

The thread scheduler is like a bouncer that gets to decide who is allowed onto the processor, and, once there, when it's time for them to get back in line.

Its behavior cannot be predicted or relied upon, but most thread schedulers try to be fair.

Extend java.lang.Thread

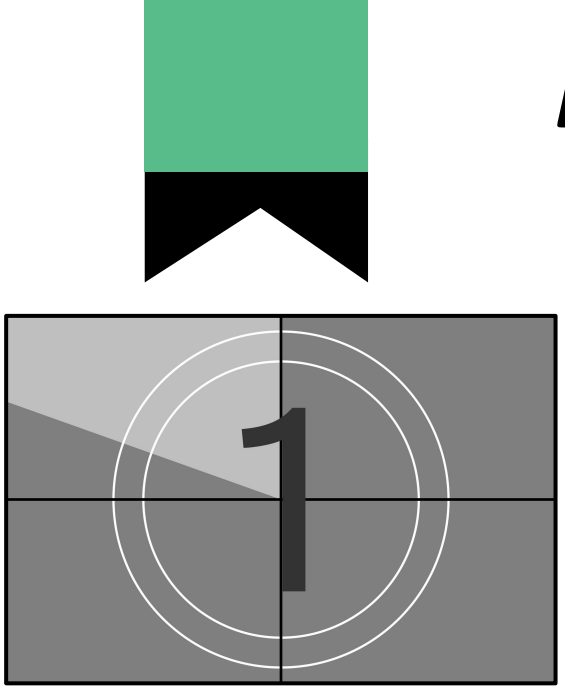
- Java provides several ways to create a program that runs in its own thread of execution; one option is to ***extend*** the `java.lang.Thread` class.
- The child class ***overrides*** the `run()` method to include any ***code*** that should be ***executed*** in the thread.
- Once an instance of the class is ***created***, the `start()` method is called to ***execute*** the code in a ***separate*** thread of execution.
- Each instance of the class that is ***created*** and ***started*** executes its code in a ***separate*** thread.

```
public class MyThread
                    extends Thread {

  @Override
  public void run() {
    System.out.println("Hi!");
  }

}
```

The run method includes any code that should be executed in the thread.

```
MyThread thread = new MyThread();
thread.start();
```

An instance of the thread class must be ***created*** and ***started*** to begin executing the code.



Activity: Extending Thread

1. Create a class named `CounterThread` that `extends Thread`.
2. Add a field to hold a `number` as well as an initializing constructor.
3. ***Override*** the `run()` method so that it counts from `1` to `100` (it should print its `number` on every line).
4. Add a `main` method that ***creates*** and ***starts*** one of your threads.

```
thread 1: 1
thread 1: 2
thread 1: 3
...
thread 1: 99
thread 1: 100
```

Implement java.lang.Runnable

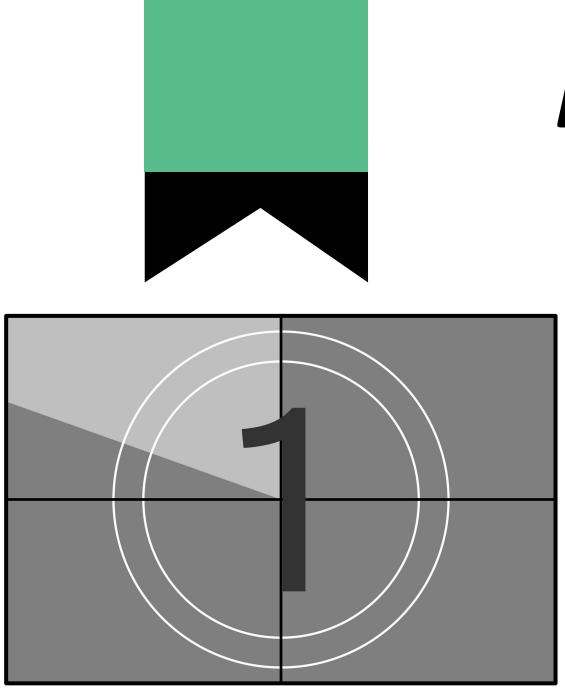
- Java provides several ways to create a program that runs in its own thread of execution; one option is to ***implement*** the `java.lang.Runnable` `interface`.
- The class ***implements*** the `run()` method to include any ***code*** that should be ***executed*** in the thread.
- Once the class is created it must be ***passed*** into the ***constructor*** of a ***new*** `Thread`.
- The new `Thread` must be ***started*** in order to execute the code in the `run()` method.

```
public class MyRunnable
             implements Runnable {

  @Override
  public void run() {
    System.out.println("Hi!");
  }

}
```

As with extending Thread, the run method includes any code that should be executed in the thread.

```
MyRunnable runner = new MyRunnable();
Thread thread = new Thread(runner);
thread.start();
```

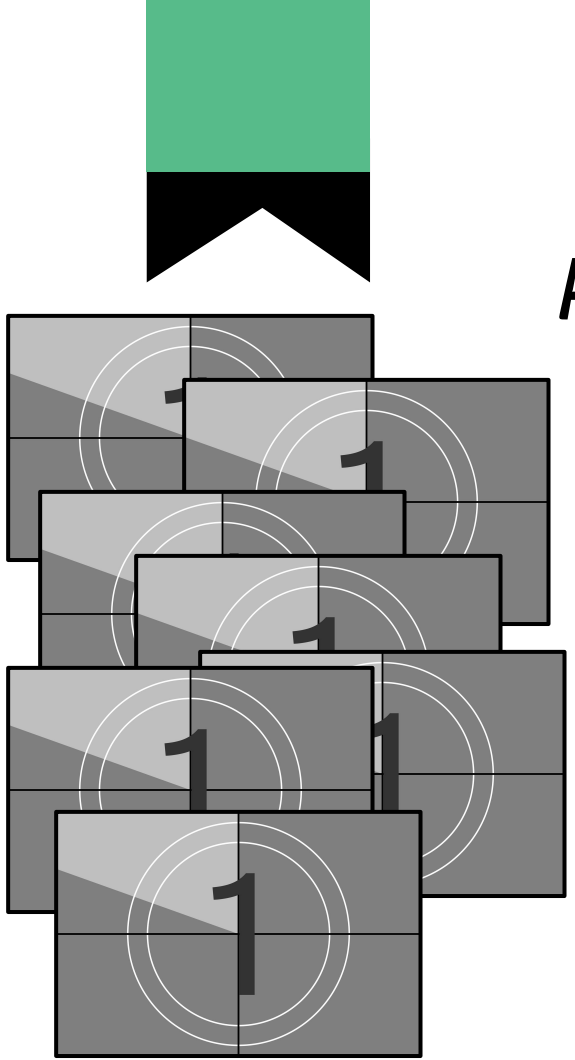
An instance of the `Thread` class must be created with the `Runnable` and started.



Activity: Implementing Runnable

1. Create a class named `RunnableCounter` that `implements Runnable`.
2. Add a field to hold a `number` as well as an initializing constructor.
3. ***Implement*** the `run()` method so that it counts from `1` to `100` (it should print its `number` on every line).
4. Add a `main` method that ***creates*** and ***starts*** one of your threads.

```
runnable 1: 1
runnable 1: 2
runnable 1: 3
...
runnable 1: 99
runnable 1: 100
```

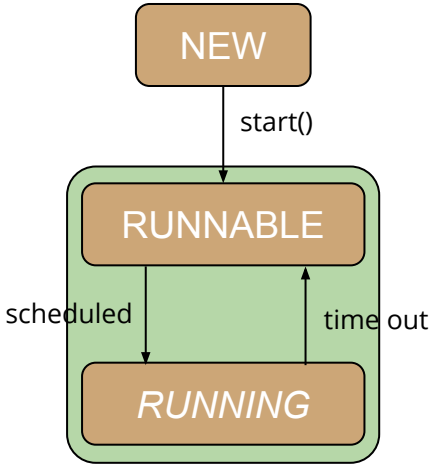


Activity: So Many Counters

1. Create a class named `ManyCounters`.
2. Add a `main` method that ***creates*** and ***starts*** at least 10 of your threads (you may pick which implementation).
3. Take a close look at the output. What happened?

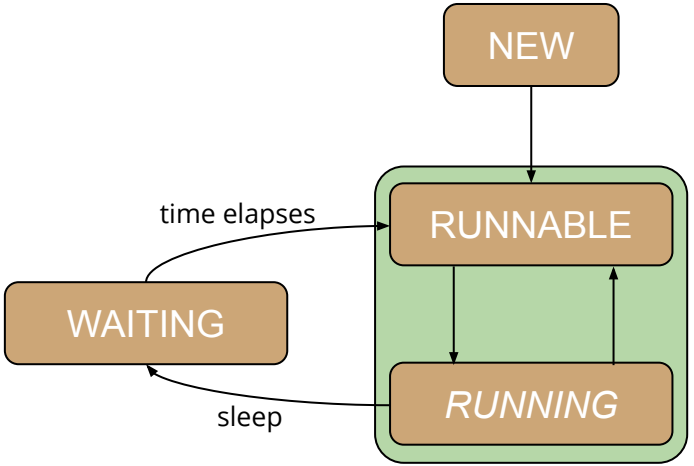
Thread States

- A `Thread` may exist in one of many ***different states***.
- ***NEW*** - indicates that the `Thread` has been ***created*** but not ***started***.
- ***RUNNABLE*** - indicates that the `Thread` has started and is ***eligible*** to be ***scheduled*** on the processor.
- ***RUNNING*** - indicates that the `Thread` is ***actively running*** on the processor.
- The ***thread scheduler*** may choose to give some other Thread ***time*** on the processor, and move a Thread from ***running*** to ***runnable*** at any time.



Sleeping

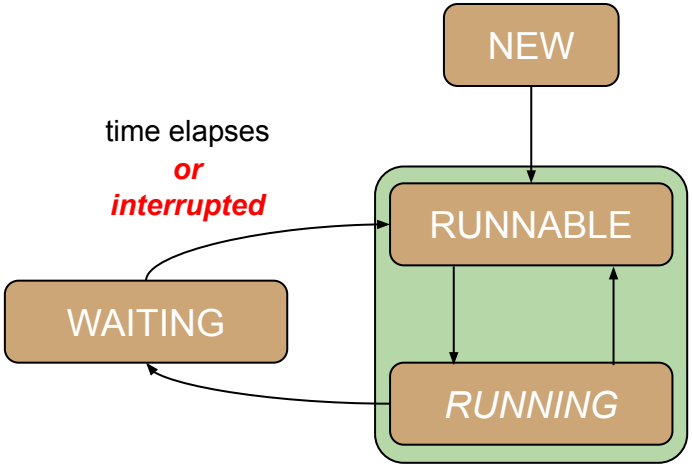
- A `Thread` may ***pause execution*** temporarily by calling the `static` `Thread.sleep(ms)` method.
  - A duration in ***milliseconds*** must be specified to determine how long execution will be ***paused***.
- The *`sleep`* method is a `static` method on the `Thread` ***class*** that will always pause the `Thread` ***from which it is called***.
- Calling the *`sleep`* method will cause the thread to transition into a new state: ***WAITING***.
- When the duration specified by the parameter to the *`sleep`* method ***elapses***, the `Thread` transitions back to the ***RUNNABLE*** state.



Note that a `Thread` in the ***WAITING*** state is ***not eligible*** to be scheduled on the processor.

Interruption

- Whenever a `Thread` enters the **WAITING** state, its normal execution is ***suspended***.
- In many cases the thread is waiting for a ***specific condition***, e.g. ***time to elapse***.
- A thread that is **WAITING** for one of these conditions may sometimes be ***interrupted prematurely***.
  - This is done by calling the `interrupt()` method on the `Thread` that is **WAITING**.
- This will cause the `Thread` to transition back to the **RUNNABLE** state where an `InterruptedException` will be ***thrown*** as soon as the `Thread` is running on the processor again.
  - `InterruptedException` is a ***checked exception***, and for this reason calls to `Thread.sleep(ms)` must be handled, e.g. with a `try`/`catch`.




```
public void waitASec() {
  try {
    Thread.sleep(1000);
  } catch(InterruptedException e) {
    System.err.println("Interrupted!");
  }
}
```

Activity: Blast Off!

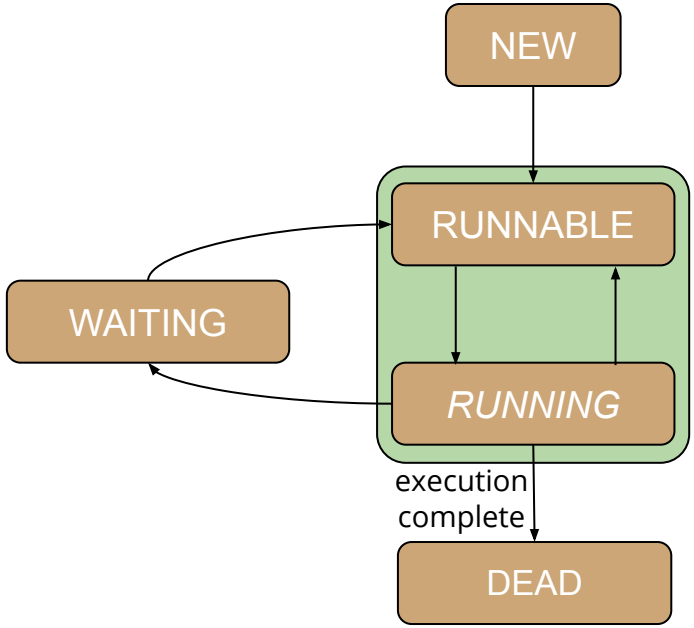
```
T-10
T-9
T-8
...
```

Create a class named `BlastOff` that simulates a timer counting the seconds to a rocket launch.

1. You can choose whether to extend `Thread` or implement `Runnable`.
2. Begin the countdown with `T-10`.
3. Pause for `1` second between each count. You will need to use `Thread.sleep(ms)` to do this.
4. Once you pass `0`, count up from there, e.g. `T+1`, `T+2`, and so on.

Thread Death

- So in what state is a `Thread` that has ***finished executing***?
  - ***DEAD***
- The `isAlive()` method can be called on a Thread to determine whether it is ***DEAD*** or alive.
  - It returns `true` if the `Thread` is in any state other than ***DEAD***, and `false` otherwise.




```
if(myThread.isAlive()) {
  System.out.println("It's aliiiiiiiive!");
}
```

A ***DEAD*** `Thread` cannot be restarted. To run the same code again, a new `Thread` must be created.

Activity: Night of the Living Thread

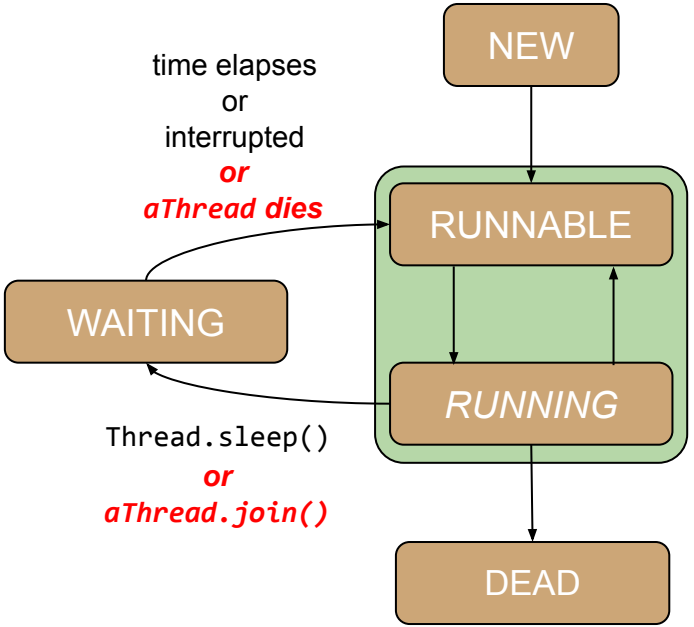
Change the `main` method in your `CounterThread` class:

1. Call `isAlive()` and print it immediately before and after the thread is started.
2. Sleep for 1 second and call `isAlive()` on the thread again.

What happens if you try to call `start()` on the thread once it is dead?

Join

- One `Thread` may pause execution temporarily and wait for a second thread to ***die*** by calling the `join()` method on the second thread.
  - Optionally, a ***duration in milliseconds*** may be used to specify the maximum amount of time the first thread will wait for the second thread to ***die***.
- Like the `Thread.sleep(ms)` method, calling the `join()` method will cause the thread to transition into the ***WAITING*** state.
- When the second thread ***dies*** or the specified maximum duration ***elapses***, the `Thread` transitions back to the ***RUNNABLE*** state.



Activity: Joining

Change the `main` method in your `ManyCounters` class so that it prints the message "All done!" after all of the threads have finished printing.

1. You will need to add all of the threads to a list.
2. Once all of the threads are started, iterate over the list and call `join()` on each thread.