Computer Programming – CS 6011 Lecture 16: Threads

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

FALL 2022

Miscellaneous

- Questions?
- Lambda Functions
  - Anonymous / unnamed / in-line functions

```
() -> { }
```

- HTML Injection
  -

Lecture 16 – Topics

- Threads
  - Executing multiple pieces of code at the same time.
  - Synchronization

Processes vs Threads

- A process is a running program
  - `% ps -def | more`
- A single program, in order to do multiple things at the same time, can use multiple threads.
- Processes cannot "talk" to each other (directly*).
  - Threads can (as we will see).
- Definition: A thread, or thread of execution, is a mechanism that allows a program (a process) to divide itself into two or more simultaneously running tasks. (~Wikipedia)

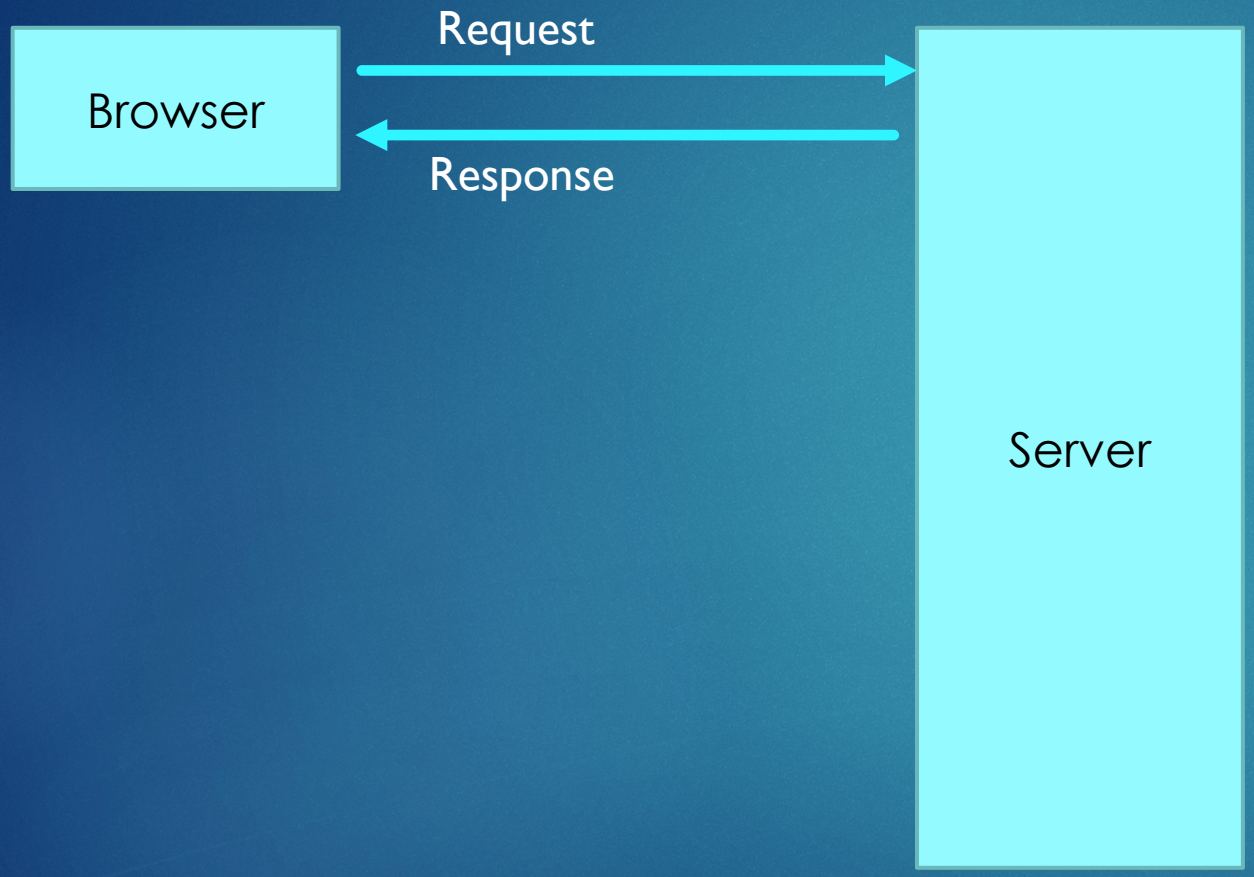
Why Threads

- (Many) Programs need to do multiple things concurrently.
- CPUs stopped getting faster* a while back… so instead of a faster CPU, the hardware people give us more CPUs (cores)…
  - Threads are one way (of several) a computer can make use of multiple cores.
  - If this laptop has 4 cores – how could I use them all at the same time?
    - I could run 4 different programs simultaneously.
    - Threads allow a single program to use multiple computer resources (cores) at the same time.
- *Concurrency* – running multiple threads at the same time. Also called *Parallelism**.
  - We see this all the time – most explicitly in User Interfaces.
  - While a program is doing something, you can still interact with the UI.
- Any questions on this concept? Where have we seen these things so far in our projects?

Thread we have "Seen" before.

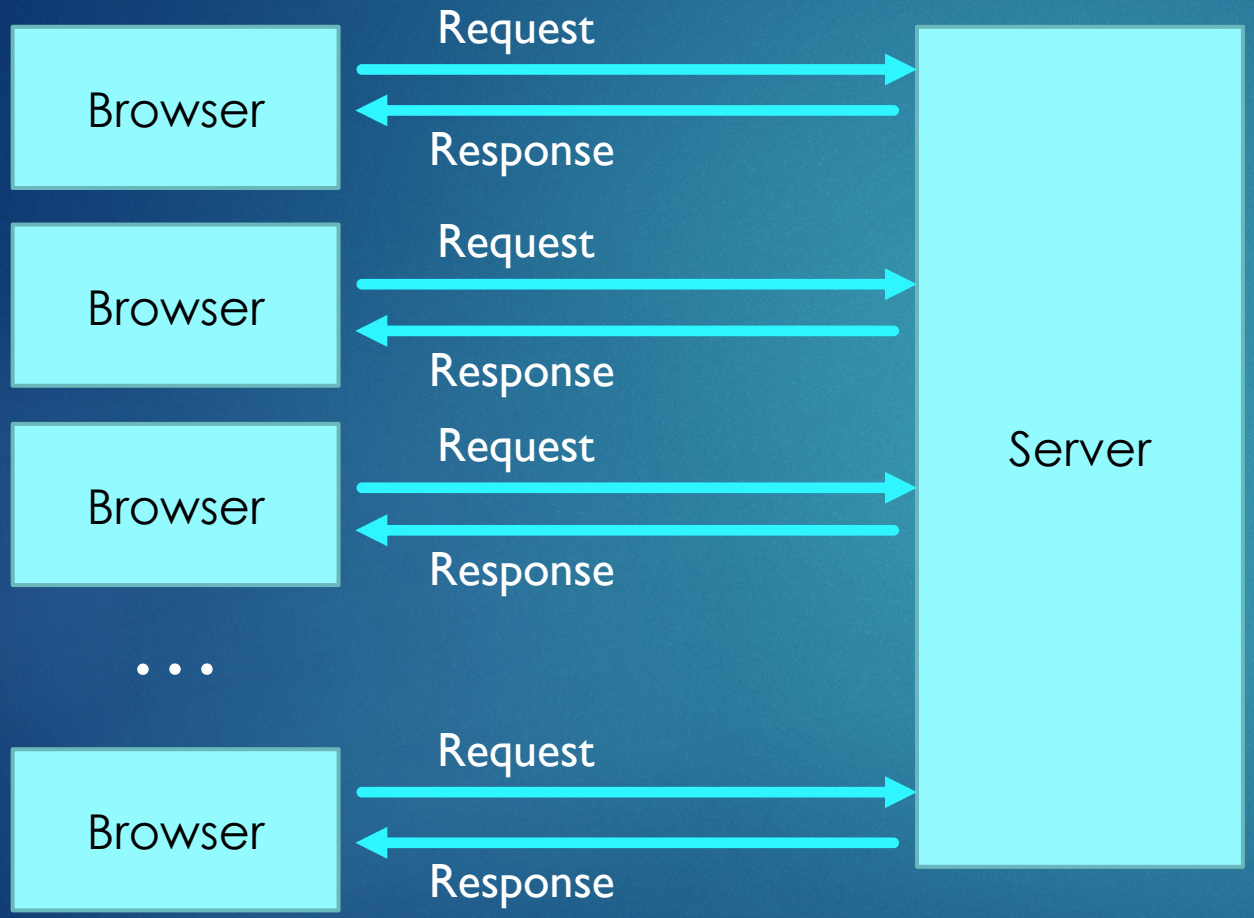
- The playing of a clip by the synthesizer… how did we "wait" until the sound was done before doing anything.
  - while (!done) { }
    - What happened to our program during this time?
      - It became frozen.
  - Added a listener…
    - The audio library actually creates its own thread and notifies the main program (thread) when it is done.
- As we work with threads, we will see why it is important to use them, and to use them correctly. For example:
  - In some of your game projects from CS 6010 you added while loops to check for events. These loops did not stop until the user pressed a button or clicked the mouse.
  - The program became unresponsive (and the spinning beachball of death appeared).

Web Server



- With a single client (browser), the
  - Request
  - Process Request
  - Response
- Happens in an apparent *synchronous* manner.
- But what would happen if there were a lot of clients making requests at the same time?

Parallel Web Server



- Multiple requests hit server simultaneously…
  - Is this possible?
  - Technically no – network card only handles one message at a time (though very fast).
- Multiple requests can hit server very quickly (before it can completely handle them).
- What does the server do?
  - Could handle them in the order they arrived, one at a time (eg: checkout line at the grocery store)… or
  - Create a thread to handle each one.
    - Threads (can) run on their own hardware (Core), and thus can respond to the requests simultaneously. Well, at least process the request simultaneously.

What is a Thread?

- Basically a running function – one that runs independently (and usually at the same time) as other code.
  - You can have many of these Threads (methods) running at the same time. Each thread doing something for you.
- This allows for (one type of) *asynchronous* programming.
- What resources does a computer need to run a function?
  - Memory – Call Stack
  - CPU Time – Either on the same core or on a separate core from other Threads / Processes.
- Threads run in the same "memory space" (heap) as each other.
  - This means that if you don't play attention, one thread can alter the same variables that another thread is using.

Threads in Java

- It is easy to make threads in Java – in fact your `main()` function is actually running in a thread.
- Threads are created using the `Thread` class.
- However, Threads use a "Runnable" object to do the actual work.
  - Runnable is an interface that contains one function: `void run()`
  - You *implement* a Runnable and provide the `run()` function.
  - Then you pass the runnable to the thread (via its constructor) for it to use.
- Important Thread methods:
  - `start();` // Start running the thread (in parallel with any other threads).
  - `join();` // Wait for a thread to finish running.
  - `threadId();` ~~`getId();`~~ // Returns a unique identifier for the Thread object.
  - `Thread.currentThread();` // Returns the Thread object the current method is running in.

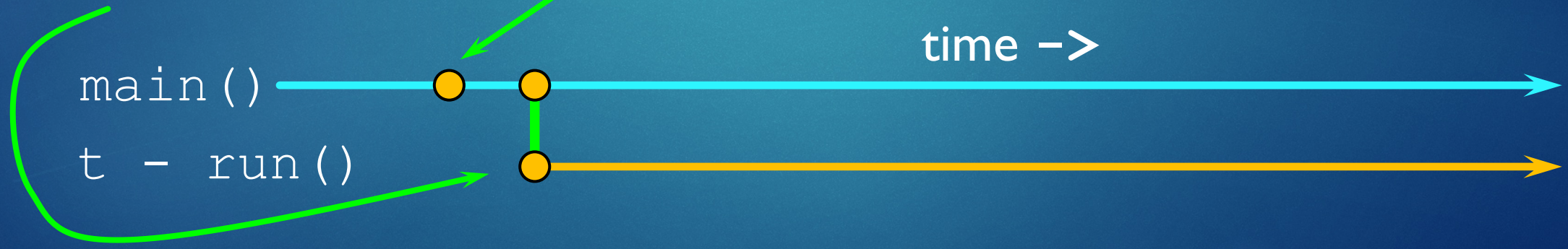
Creating a Thread (Lambda Version)

- Creating a Thread as a *lambda* function. (Ie, in-line)

```
Thread t = new Thread( () -> {
    // Code to be run in this thread.
    // Many times the code is within a while() loop so that it continues
    // doing work for us…
} );
```

- To actually get the thread running, we need to use the `start()` method.

```
t.start();
```

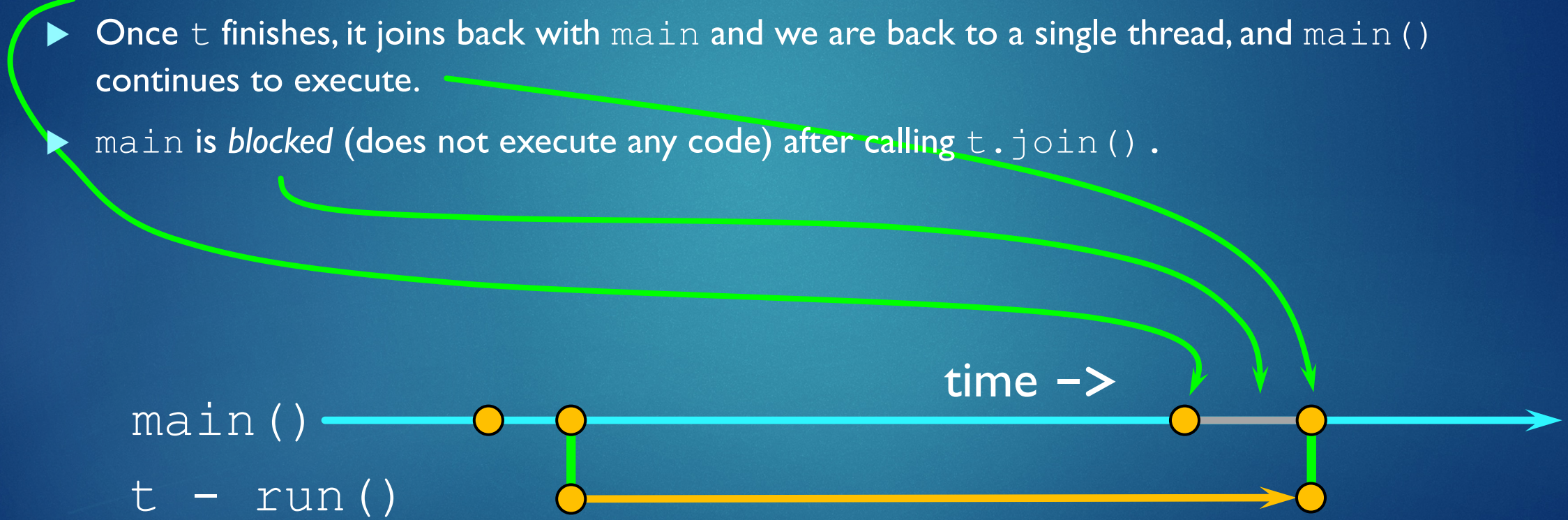


Ending a Thread…

- If `main()` can't do anything else until `t` finishes its jobs, we use:

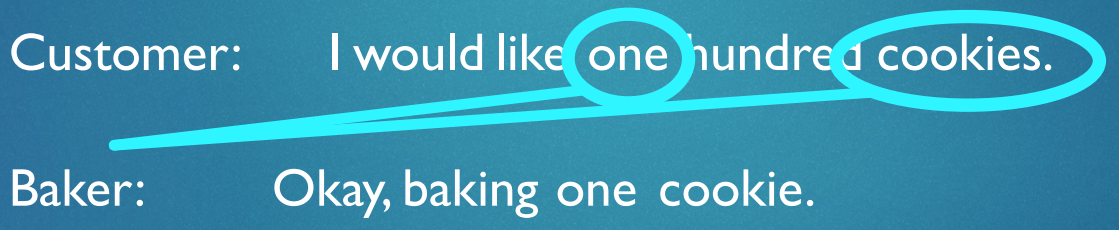
```
t.join() // Main waits for t to finish…
```

- Once `t` finishes, it joins back with `main` and we are back to a single thread, and `main()` continues to execute.
- `main` is *blocked* (does not execute any code) after calling `t.join()`.



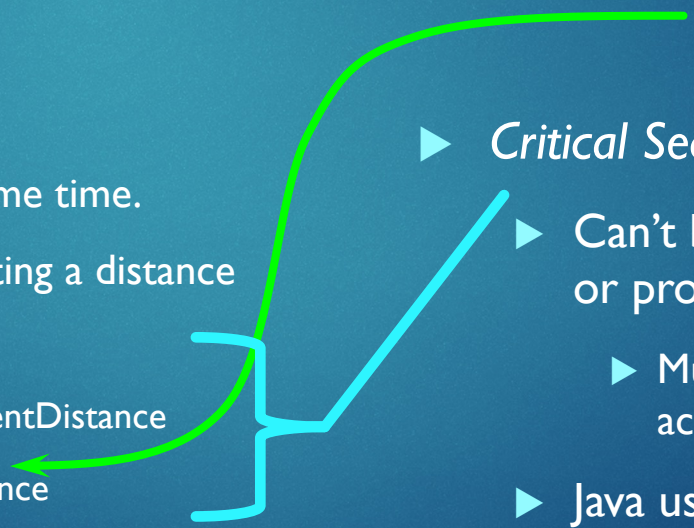
(Mis-)Synchronization

- Think about two people working together – a customer and a baker.
  - Each of these "people" is part of our program and represented by a different thread (so that they can be doing their own things at the same time).
  - The customer asks for some cookies… [I would like one hundred cookies]



How do Threads Talk to Each Other?

- Messages – We'll talk about this in a future class.
- Shared Memory
  - Common variables
    - One Whiteboard
    - Everyone is a ghost
      - No one knows when anyone else is doing something
    - Everyone has a marker
  - What can go wrong with this?
    - Two threads read / write at the same time.
    - Example, two threads using / updating a distance
      - Read currentDistance
      - Calculate the new value for currentDistance
      - Store new value for currentDistance
- Another Thread
  - Changes value of currentDistance
- *Critical Section*
  - Can't be interrupted by another thread or problems occur.
    - Mutexes / Locks are used to control access to critical section.
  - Java uses *synchronized* (keyword)



Critical Section Example (Not Protected)

- `int x = 33;`
- If we aren't careful, this happens:
- Thread 1:

```
                  33        0
int   i   =   x   +   x;
```

- What should `i`'s value be equal?
  - How was that calculated? What's the value of `x`?
- Let's do this one step at a time…
  - The computer (well, an individual thread) can only do one thing at a time, so what is the first thing it does in the line above? What happens first?
    - Computer loads the value of `x` from memory. Now what?
    - Computer is about to load the value of the 2nd variable (which just happens also to be `x`).
- But wait… there is a 2nd thread that is running:
- Thread 2:

```
x = 0;
```

- Now back to Thread 1…
  - `i` is set to 33…

Synchronization

- Because threads are running at the same time (and in a nondeterministic order), any data (variable, etc) that is read (accessed) by one thread, but was written (assigned) by another thread is a potential problem.
  - These types of bugs happen "randomly" and are very hard to reproduce.
  - The best approach to fixing them is to prevent them in the first place.
- Synchronization is an approach to prevent different threads from accessing the same data at the same time.
- We'll learn several ways to safely share data between threads.

Java Synchronized

- Synchronized Method

```
public class MyClass {
    public synchronized void doit() { … }
}
```

- Each thread that calls `doit()` – *on the same object* – checks to see if any other thread is currently running the `doit()` code, and if so, waits until the other thread is done.

```
MyClass myObj = new MyClass();
```

- What if two different threads have access to `myObj` and try to run `doit()`?
  - Thread t1

    ```
    myObj.doit();
    ```

  - Thread t2

    ```
    myObj.doit();
    ```

- Order of t1 `doit()` vs t2 `doit()`?
  - Undefined… but only **one** will execute at a time.

Java Synchronized

- Synchronized Method

```
public class MyClass {
    public synchronized void doit() { … }
}
```

- However, threads using *different* objects can access the same code simultaneously!

```
MyClass myObj1 = new MyClass();
MyClass myObj2 = new MyClass();
```

- Thread t1

```
myObj1.doit();
```

- Thread t2

```
myObj2.doit();
```

- Order of t1 `doit()` vs t2 `doit()`?
  - Undefined… but both can execute at the same time – they are using *different* objects!

General Rule

- When we have shared objects (such as `myObj` on slide 17), we can use synchronized methods to allow multiple threads to call the same function (at the "same" time) on that object.
  - This will protect shared variables (data) from being corrupted.
  - But will only allow one thread at a time to actually run…
    - In other words, the program has been *serialized* during this part of its execution and is thus not enjoying any speed up benefits.

Threading is Easy… If:

1. None of the threads share any data; or
2. All the shared data is *read-only*.

Threading Example

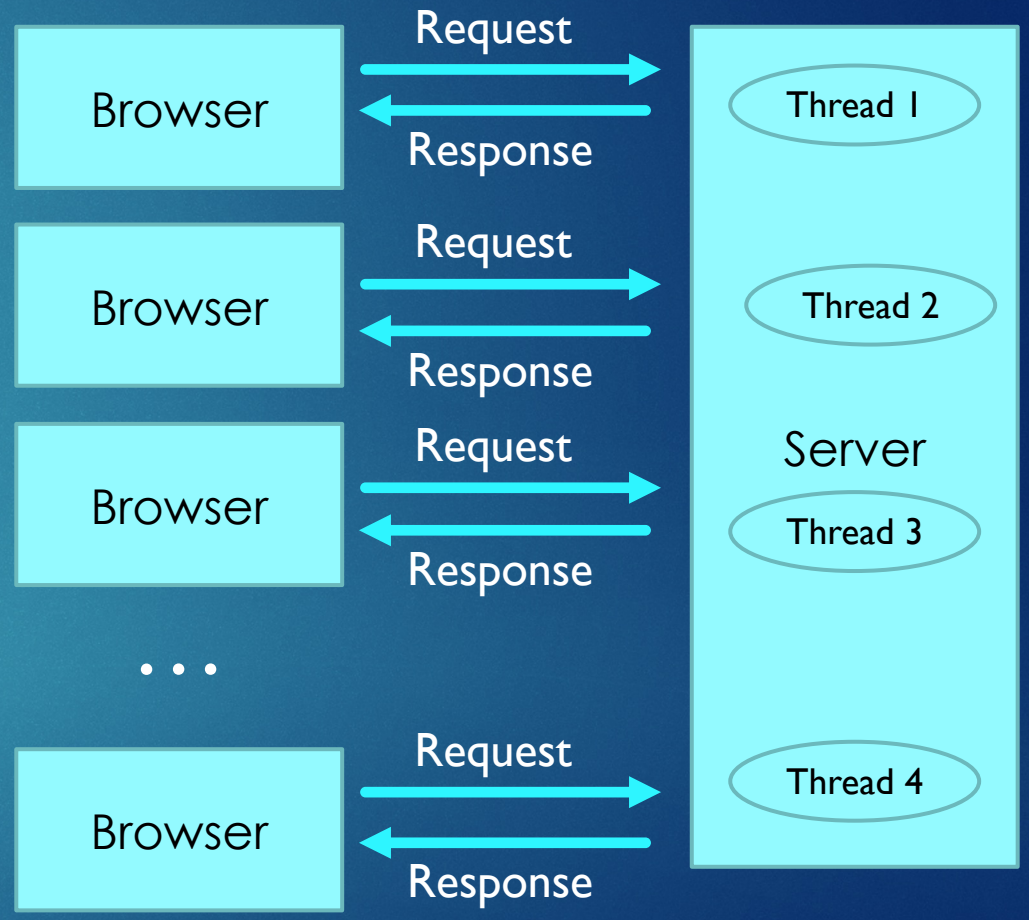
- Create a *Runnable* class.
  - Runnables only have one method: *run()*.
- Create threads to execute the code (in parallel).
  - `start()` the thread and off it goes…
    - Executing the `run()` method.
- In-class example…

Lab

- Examining a “critical region”.
- Basically x = x + number;
- What happens when a bunch of threads try to help each other to do this (quickly) at the same time?

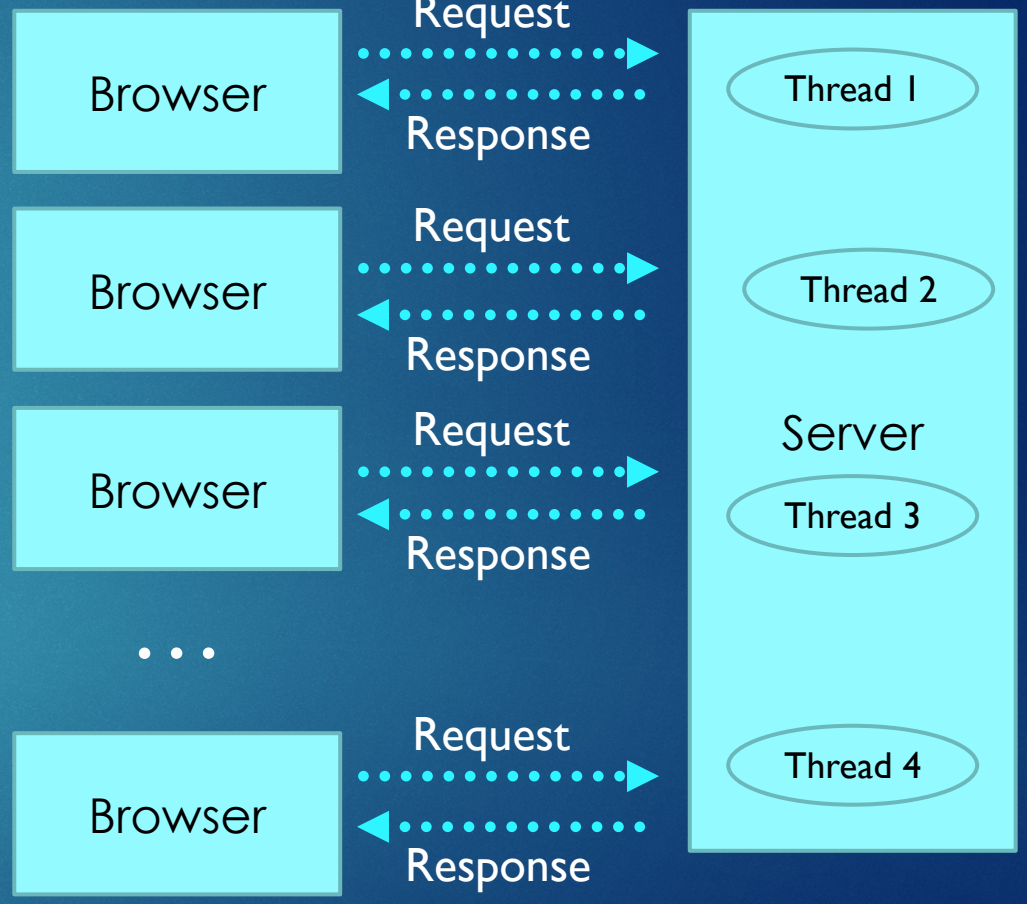
Assignment – Multithreaded Web Server

- Why would we want to do this?
  - So multiple clients can talk to the server at the same time.
- When do we want to create a thread?
  - When a new client has connected to us.
  - When does this occur?
    - `ss.accept()`
- When do we destroy a thread?
  - When the server is done responding to a client.
  - Do we have to do this ourself?
    - No – the garbage collector will take care of them for us.
- Recap – What happens now…



Assignment – Multi-Threaded Web Server

- At this point Thread 1 – 4 have finished running (ie: finished processing and responding to the requests)… The `run()` function ends and thus nothing is referencing them anymore… so:
  - The garbage collector kicks in and…
    - The Thread objects are gone.
- Considerations for threading…
- Do you have any shared data for your webserver?
  - Do you have static variables?
  - As a side note, we do have a "shared" resource. What is it?
    - Files
    - But they are "read-only"* so don't need to worry about this.



Multi-Threaded Web Server Demo

- Loading multiple pages / images at the same time.
  - I'm using an (artificial) random delay before serving each image.
- Network tab on browser tools…

Tuesday Assignments

- Code Review – Web Chat Client
- Lab – Threads
- Assignment – Multi-Threaded Web Server

~ Fin ~