PROGRAMMING III
OOP. JAVA LANGUAGE

COURSE 11

PREVIOUS COURSE CONTENT

- **Input/Output Streams**
  - Text Files
  - Byte Files
  - RandomAcessFile
- **Exceptions**
- **Serialization**
- **NIO**

COURSE CONTENT

- **Threads**
  - Threads lifecycle
  - Thread class]
  - Runnable interface
  - Synchronization

MULTITASKING AND MULTITHREADING

- **Multitasking refers to a computer's ability to perform multiple jobs concurrently**
  - more than one program are running concurrently, e.g., UNIX

- **A thread is a single sequence of execution within a program**

- **Multithreading refers to multiple threads of control within a single program**
  - each program can run multiple threads of control within it, e.g., Web Browser

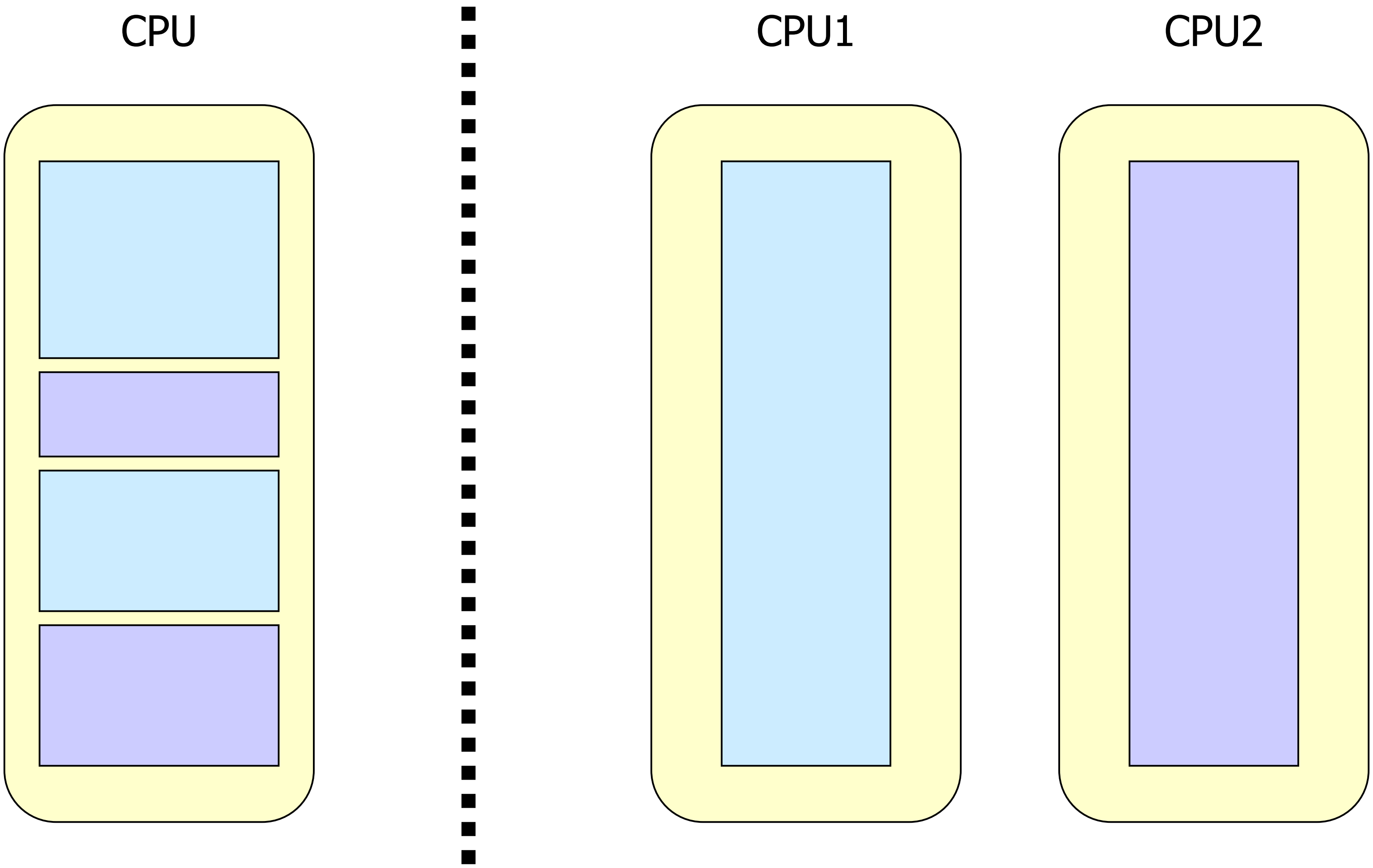
MOTIVATION FOR CONCURRENT PROGRAMMING

- **Pros**
  - Advantages even on single-processor systems
  - Efficiency
    - Downloading network data files
  - Convenience
    - A clock icon
  - Multi-client applications
    - HTTP Server, SMTP Server
  - Many computers have multiple processors
    - Find out via Runtime.getRuntime().availableProcessors()
- **Cons**
  - Significantly harder to debug and maintain than single-threaded apps

CONCURRENT VS PARALLEL PROGRAMMING

- **Concurrent**
  - Tasks that overlap in time
    - The system might run them in parallel on multiple processors, or might switch back and forth among them on the same processor
- **Parallel**
  - Tasks that run at the same time on different processors

CONCURRENT VS PARALLEL PROGRAMMING

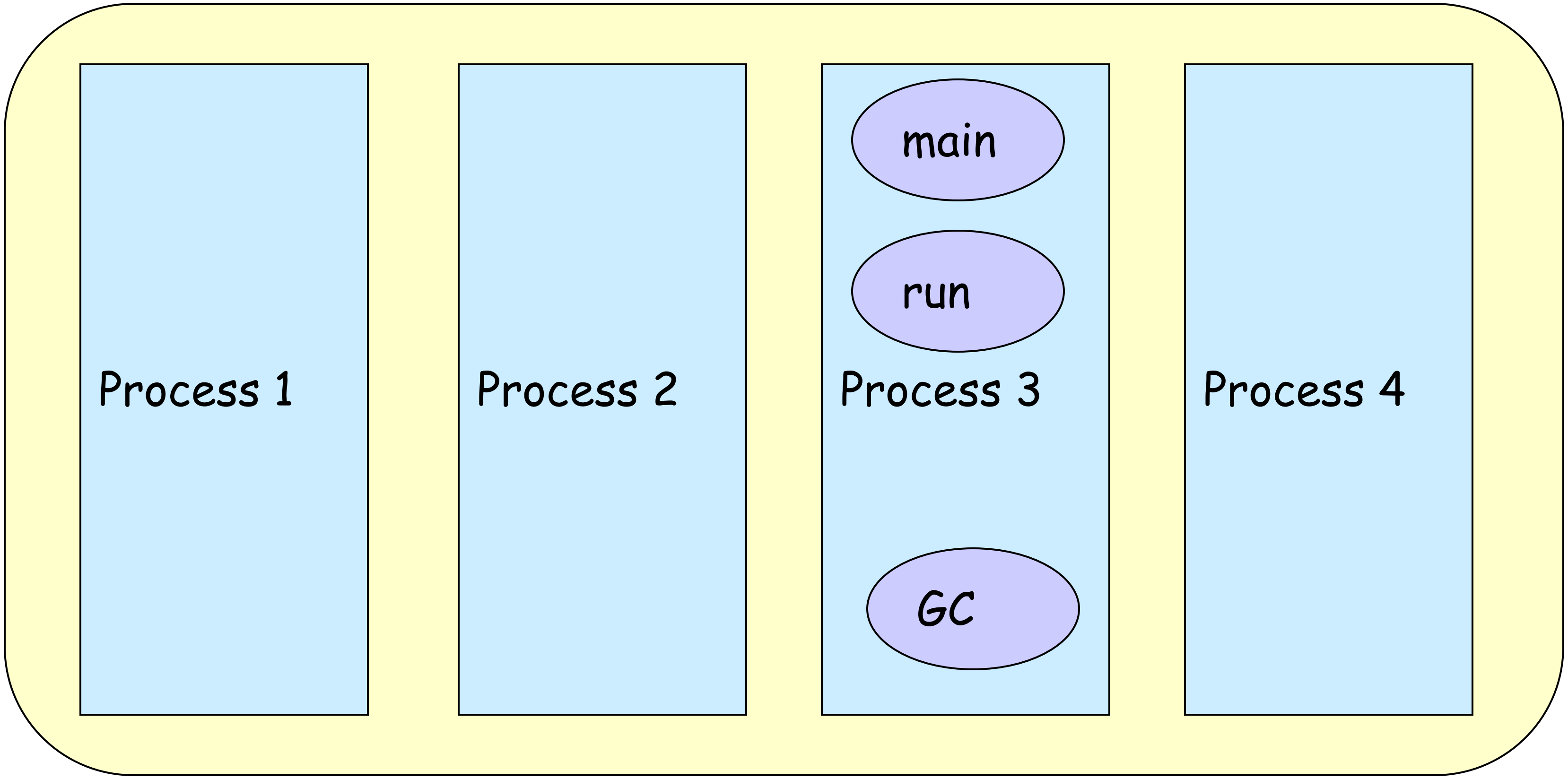


JAVA THREADS (CONCURRENT) VS. FORK/JOIN FRAMEWORK (PARALLEL)

- **Using threads**
  - When task is relatively large and self-contained
  - Usually when you are waiting for something, so would benefit even if there is only one processor
  - Needed even in Java 8 where you have parallel streams

- **Using fork/join or parallel streams**
  - When task starts large but can be broken up repeatedly into smaller pieces, then combined for final result.
  - No benefit if there is only one processor

THREADS AND PROCESSES



CREATING THREADS (METHOD 1)

- **Extending the Thread class**
  - must implement the *run()* method
  - thread ends when *run()* method finishes
  - call *.start()* to get the thread ready to run

CREATING THREADS EXAMPLE 1

```
class Output extends Thread {
        private String toSay;
        public Output(String st) {
                toSay = st;
        }
        public void run() {
                try {
                        for(;;) {
                                System.out.println(toSay);
                                sleep(1000);
                        }
                } catch(InterruptedException e) {
                        System.out.println(e);
                }
        }
}
```

CREATING THREADS EXAMPLE 1

```
class Program {
        public static void main(String [] args) {
                Output thr1 = new Output("Hello");
                Output thr2 = new Output("There");
                thr1.start();
                thr2.start();
        }
}
```

- ❑ main thread is just another thread (happens to start first)
- ❑ main thread can end before the others do
- ❑ any thread can spawn more threads

CREATING THREADS (METHOD 2)

- **Implementing Runnable interface**
  - virtually identical to extending Thread class
  - must still define the *run()*method
  - setting up the threads is slightly different

CREATING THREADS EXAMPLE 2

```
class Output implements Runnable {
        private String toSay;
        public Output(String st) {
                toSay = st;
        }
        public void run() {
                try {
                        for(;;) {
                                System.out.println(toSay);
                                Thread.sleep(1000);
                        }
                } catch(InterruptedException e) {
                        System.out.println(e);
                }
        }
}
```

CREATING THREADS EXAMPLE 2

```
class Program {
        public static void main(String [] args) {
                Output out1 = new Output(“Hello”);
                Output out2 = new Output(“There”);
                Thread thr1 = new Thread(out1);
                Thread thr2 = new Thread(out2);
                thr1.start();
                thr2.start();
        }
}
```

- **main is a bit more complex**
- **everything else identical for the most part**
- **Advantage of Using Runnable**
  - implementing runnable allows class to extend something else

CONTROLLING JAVA THREADS

- ***start()***
  - begins a thread running
- ***wait()* and *notify()***
  - for synchronization
- ***stop()***
  - kills a specific thread (deprecated)
- ***suspend()* and *resume()***
  - deprecated
- ***join()***
  - wait for specific thread to finish
- ***setPriority()***
  - 0 to 10 (MIN_PRIORITY to MAX_PRIORITY); 5 is default (NORM_PRIORITY)

CONTROLLING JAVA THREADS

- **yield()**
  - Causes the currently executing thread object to temporarily pause and allow other threads to execute
  - Allow only threads of the same priority to run

- **sleep(int *m*)/sleep(int *m*,int *n*)**
  - The thread sleeps for *m* milliseconds, plus *n* nanoseconds

JAVA THREAD SCHEDULING

- **highest priority thread runs**
  - if more than one, arbitrary

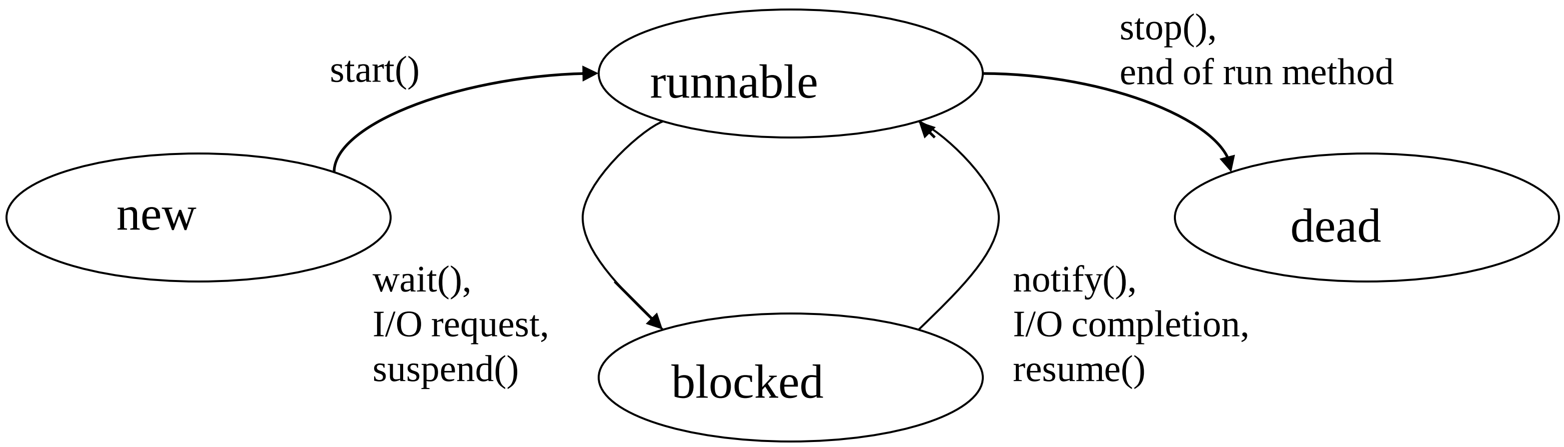
- ***yield()***
  - current thread gives up processor so another of equal priority can run
  - if none of equal priority, it runs again

- ***sleep(msec)***
  - stop executing for set time
  - lower priority thread can run

STATES OF JAVA THREADS

- **4 separate states**
  - new
    - just created but not started
  - runnable
    - created, started, and able to run
  - blocked
    - created and started but unable to run because it is waiting for some event to occur
  - dead
    - thread has finished or been stopped

STATES OF JAVA THREADS



SYNCHRONIZATION

- **Synchronization is prevent data corruption**
- **Synchronization allows only one thread to perform an operation on a object at a time.**
- **If multiple threads require an access to an object, synchronization helps in maintaining consistency.**

SYNCHRONIZATION. EXAMPLE

```
public class Counter{
        private int count = 0;
        public int getCount(){
                return count;
        }

        public setCount(int count){
                this.count = count;
        }
}
```

- In this example, the counter tells how many an access has been made.
- If a thread is accessing setCount and updating count and another thread is accessing getCount at the same time, there will be inconsistency in the value of count.

SYNCHRONIZATION. EXAMPLE. SOLUTION

```
public class Counter{
        private static int count = 0;
        public synchronized int getCount(){
                return count;
        }

        public synchoronized setCount(int count){
                this.count = count;
        }
}
```

- By adding the synchronized keyword we make sure that when one thread is in the setCount method the other threads are all in waiting state.
- The synchronized keyword places a lock on the object, and hence locks all the other methods which have the keyword synchronized. The lock does not lock the methods without the keyword synchronized and hence they are open to access by other threads.

SYNCHRONIZATION

- **Synchronizing a section of code**

```
synchronized(someObject) {
        code
}
```

- **Normal interpretation**
  - Once a thread enters that section of code, no other thread can enter until the first thread exits

- **Stronger interpretation**
  - Once a thread enters that section of code, no other thread can enter any section of code that is synchronized using the same “lock” object

- **If two pieces of code say “synchronized(blah)”, the question is if the blah’s are the same object instance**

SYNCHRONIZATION

- **Synchronized Method**
  - Pros
    - Your IDE can indicate the synchronized methods.
    - The syntax is more compact.
    - Forces to split the synchronized blocks to separate methods.
  - Cons
    - Synchronizes to this and so makes it possible to outsiders to synchronize to it too.
    - It is harder to move code outside the synchronized block.
    - Synchronized block

- **Synchronized block**
  - Pros
    - Allows using a private variable for the lock and so forcing the lock to stay inside the class.
    - Synchronized blocks can be found by searching references to the variable.
  - Cons
    - The syntax is more complicated and so makes the code harder to read.

SYNCHRONIZATION

METHOD

```
// locks the whole object
...
private synchronized void someInputRelatedWork() {
    ...
}
private synchronized void someOutputRelatedWork()
{
    ...
}
```

BLOCK

```
// Using specific locks
Object inputLock = new Object();
Object outputLock = new Object();

private void someInputRelatedWork() {
    synchronize(inputLock) {
        ...
    }
}
private void someOutputRelatedWork() {
    synchronize(outputLock) {
        ...
    }
}
```

VOLATILE VARIABLES

- **If a variable, object, or field is declared as volatile, then**
  - It can be used for reliable communication between threads.
- **Non-volatile variables, objects, and fields have unpredictable semantics, if they are read & written by more than one thread.**
  - For example, if Thread1 and Thread2 are both executing the following:

```
int x = 1;
System.out.println(name + ": " + ( x++ ) );
```

  - This is equivalent to executing:

```
int x = 1;
int t = x;
t = t + 1;
x = t;
System.out.println(name + ": " + x );
```

```
Thread1: 2
Thread2: 3
```

or

```
Thread2: 2
Thread1: 3
```

or

```
Thread2: 3
Thread1: 2
```

or

```
Thread1: 3
Thread2: 2
```

or

```
Thread1: 2
Thread2: 2
```

or

```
Thread2: 2
Thread1: 2
```

VOLATILE VARIABLES

- **If a variable, object, or field is declared as volatile, then**
  - It can be used for reliable communication between threads.
  - Semantics are predictable – if a thread reads the variable then writes it, the other thread is blocked from reading until the newly-written value is available.
    - Warning: you can cripple a multithreaded program by making all of its variables volatile.
    - The JVM must always read volatiles from memory. Frequently-used non-volatile values are retained in the CPU register file, which is *much* faster than main memory.
  - For example, if Thread1 and Thread2 are both executing the following:

```
volatile int x = 1;
System.out.println(name + ": " + ( x++ ) );
```

  - Thread1 and Thread2 always get different values!

```
Thread1: 2
Thread2: 3
```

or

```
Thread2: 2
Thread1: 3
```

```
Thread1: 3
Thread2: 2
```

or

```
Thread2: 3
Thread1: 2
```