Fundamentals of Concurrency

By Warren Mansur; edited by Eric Braude

BOSTON UNIVERSITY
METROPOLITAN COLLEGE

Concurrency Agenda

1. **Introduction**
2. Creating Threads in Java
3. Visualizing & defining thread outcomes
4. Coordinating threads
5. Thread pools
6. Concurrent reading and writing

Background Concepts

Application

- Software Package
- Supports completing a set of related tasks
- e.g. Google Chrome or Microsoft Word

Process

- Instance of app executing on device
- Each process has its own memory, data, code, and operating system resources

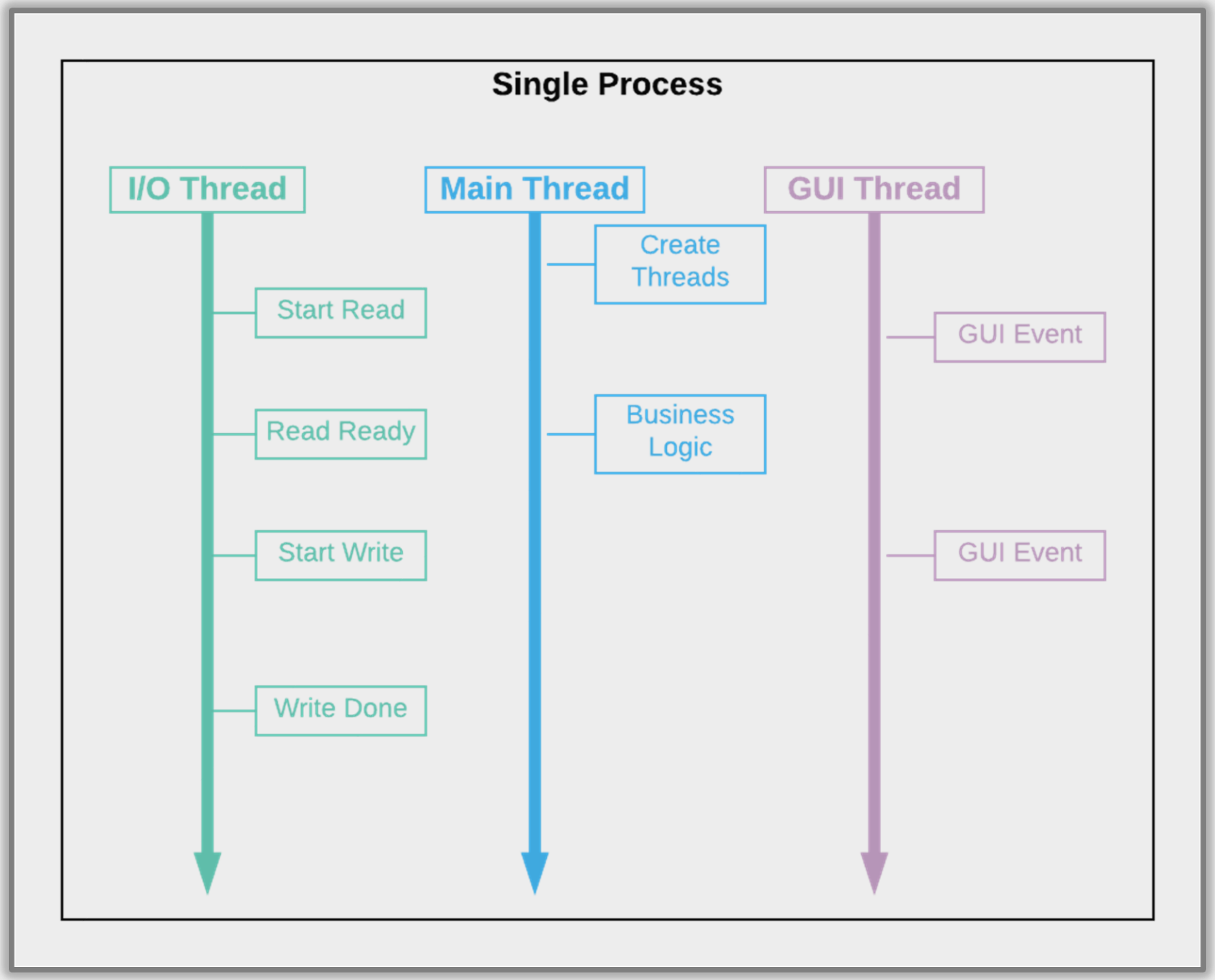
Thread

- Lightweight execution stream within process
- Each thread shares memory, data, and code within a process, but has its own execution stream (PC counter, registers, and stack)

Operating System Concurrency

- Modern hardware has sufficient resources to run many applications and threads simultaneously.
- Users have multiple processes open simultaneously, (browser, Word, messaging …)
- Processes typically have multiple threads running simultaneously for better performance and responsiveness:
  - E.g., UI responding on one thread; process performing other tasks
  - Tasks broken into subtasks and run in parallel
  - may utilize multiple processors

Multi-threading Visual



Types of Schedulers

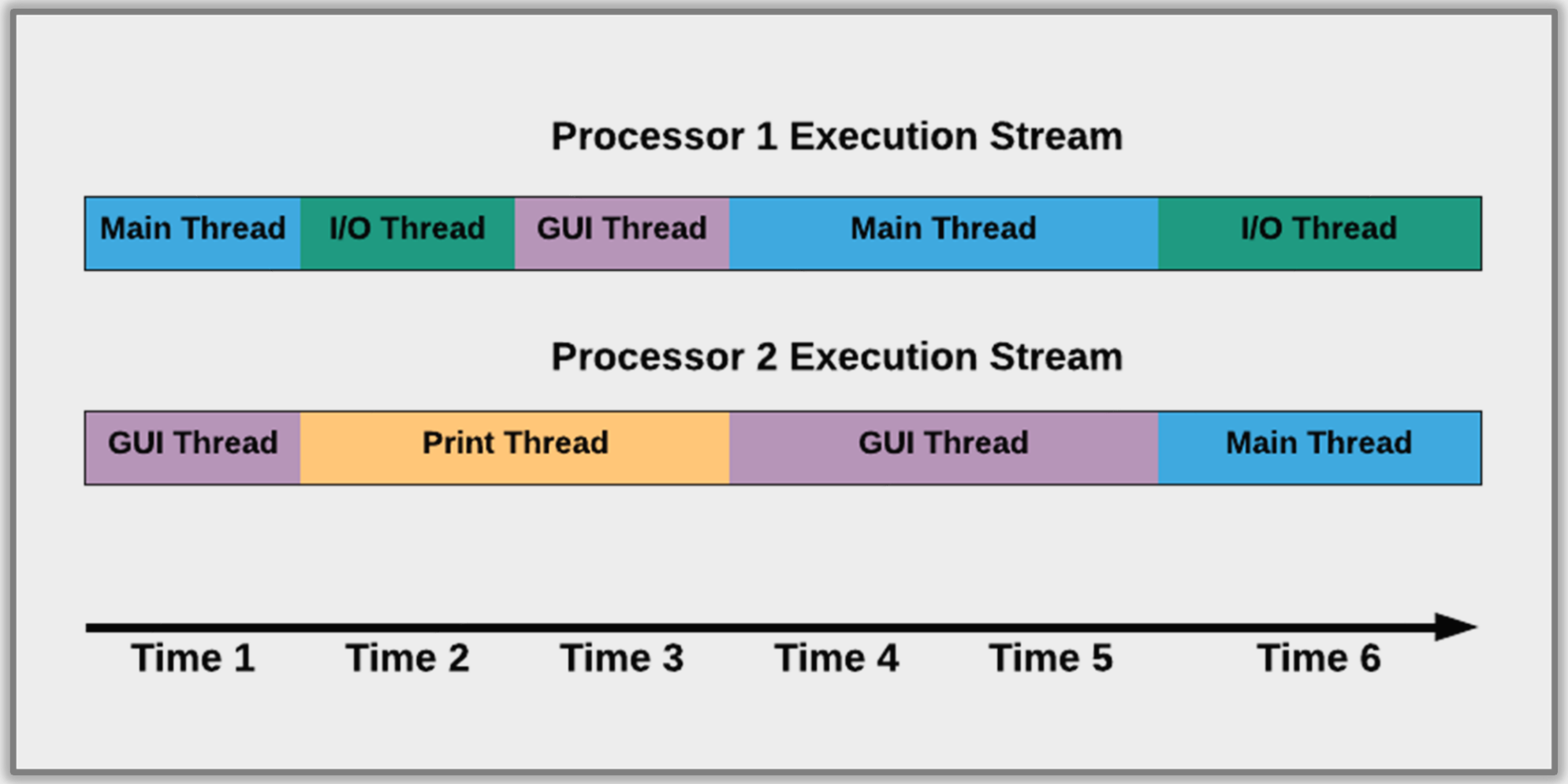
Native

- Operating system schedules threads
- Threads in one process can run on multiple processors

User

- Process (JVM) schedules threads
- Process restricted to single processor

Scheduling Visual



Thread Implementation in Java

Runnable Interface

- Multiple ways to create
- Suitable for Long-Running

Callable Interface

- Can return value when done
- Suitable for Shorter Tasks

Callable/Runnable Notes

- Classes that implement either interface can be used to create a separate stream of execution.
- Both are functional interfaces with a single public method, so are suitable for lambdas.
- An object implementing either interface is *not* a stream of execution in itself; the stream of execution happens behind the scenes; the object is a placeholder for starting and interacting with the stream of execution.
- Concurrency is not an object-oriented concept per se

Types of Threads in Java

- When the JVM starts, it creates a thread named “Main” that executes the public static void main(...) method.
- The main thread may create child threads to perform various tasks as needed.
- The JVM designates each thread as either “daemon” or “user” (the default is “user”).
- The JVM will not exit until all *user* threads have terminated (not so with daemon threads).

Thread States in Java

State	Explanation
New	just created but not yet runnable.
Runnable	ready to be executed--may or may not be currently running, depending on scheduler.
Blocked/ Waiting	temporarily inactive because waiting on a lock, resource, or notification.
Timed Wait	temporarily inactive for a specific amount of time (which can be preempted).
Terminated	has finished executing (successfully or unsuccessfully) and no longer scheduled to run.

Basic Thread Control in Java

Operation	Explanation
Thread.start()	Creates a new thread for a new stream of execution.
Thread.join()	The current thread waits for another thread to complete (optionally within a maximum wait time).
Thread.sleep()	The current thread becomes temporarily inactive for a specified period of time.
Thread.yield()	Gives a hint to the scheduler that another thread can be executed instead of the current thread.

BOSTON UNIVERSITY
METROPOLITAN COLLEGE

Concurrency Agenda

1. Introduction
2. **Creating Threads in Java**
3. Visualizing & defining thread outcomes
4. Coordinating threads
5. Thread pools
6. Concurrent reading and writing

Create by Extending *Thread*

```
public class DemoExtendThread {
  public static void main(String[] args) {
    ExtendThread ThreadA = new ExtendThread('A');
    ExtendThread ThreadB = new ExtendThread('B');
    ThreadA.start();
    ThreadB.start();
  }
}
class ExtendThread extends Thread {
  private char ThreadID;
  public ExtendThread(char ThreadID) {
    this.ThreadID = ThreadID;
  }
  public void run() {
    for (int i = 0; i < 15; i++)
    {
      System.out.print(" " + ThreadID + i);
    }
  }
}
```

Sample Run 1

```
 B0 A0 A1 A2 A3 B1 A4 A5 A6 A7 A8 A9 A10 A11
B2 A12 B3 A13 A14 B4 B5 B6 B7 B8 B9 B10 B11
B12 B13 B14
```

Sample Run 2

```
 A0 B0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12
A13 A14 B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11
B12 B13 B14
```

```java
public class DemoImplementRunnable {
  public static void main(String[] args) {
    Thread ThreadA = new Thread(new ImplementRunnable('A'));
    Thread ThreadB = new Thread(new ImplementRunnable('B'));
    ThreadA.start();
    ThreadB.start();
  }
}
class ImplementRunnable implements Runnable {
  private char ThreadID;
  public ImplementRunnable(char ThreadID) {
    this.ThreadID = ThreadID;
  }
  public void run() {
    for (int i = 0; i < 15; i++)
    {
      System.out.print(" " + ThreadID + i);
    }
  }
}
```

Create by Implementing *Runnable*

Sample Run 1

```
 B0 A0 B1 A1 A2 B2 A3 A4 A5 B3 B4 B5 B6 B7 B8
B9 B10 B11 B12 B13 B14 A6 A7 A8 A9 A10 A11 A12
A13 A14
```

Sample Run 2

```
 B0 B1 B2 B3 B4 A0 A1 A2 A3 A4 A5 A6 A7 A8 A9
A10 A11 A12 A13 A14 B5 B6 B7 B8 B9 B10 B11 B12
B13 B14
```

```
public class DemoCallable {
  public static void main(String[] args) {
    try {
      FutureTask<Integer> future = new FutureTask<Integer>
                (new ChildCallableThread());
      Thread childThread = new Thread(future);
      childThread.start();
      int result = future.get();
      System.out.println("The child thread's result is " + result);
    }
    catch (ExecutionException | InterruptedException e) {
      e.printStackTrace();
    }
  }

  private static class ChildCallableThread implements Callable<Integer> {
    public Integer call() {
      int retVal = 0;
      for (int i = 0; i < 15; i++)
      {
        retVal += i;
      }
      return retVal;
    }
  }
}
```

Create as Callable with *Future*

Output

```
The child thread's result is 105
```

BOSTON UNIVERSITY
METROPOLITAN COLLEGE

Concurrency Agenda

1. Introduction
2. Creating Threads in Java
3. **Visualizing & defining thread outcomes**
4. Coordinating threads
5. Thread pools
6. Concurrent reading and writing

In General, Without Parallelism:

OUTCOME 1: P1 (predicate #1)

OUTCOME 2: P2 (e.g., P2 is “x == y”)

OUTCOME 3: P3

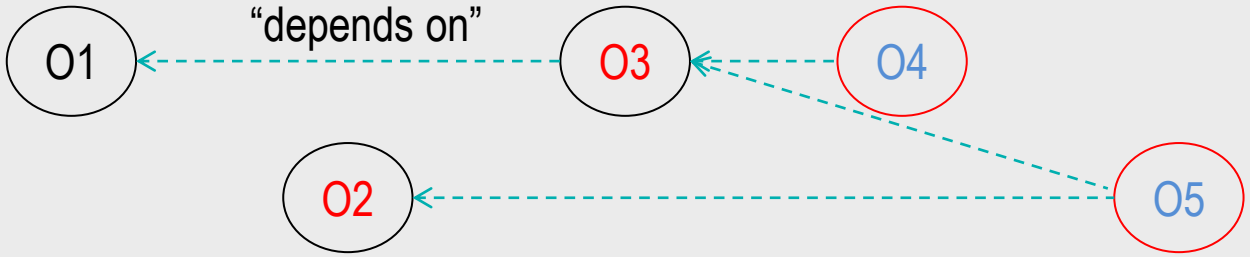
OUTCOME 4: P4

OUTCOME 5: P5

For *serial* attainment:

O_i attained before O_{i+1} attainment begins

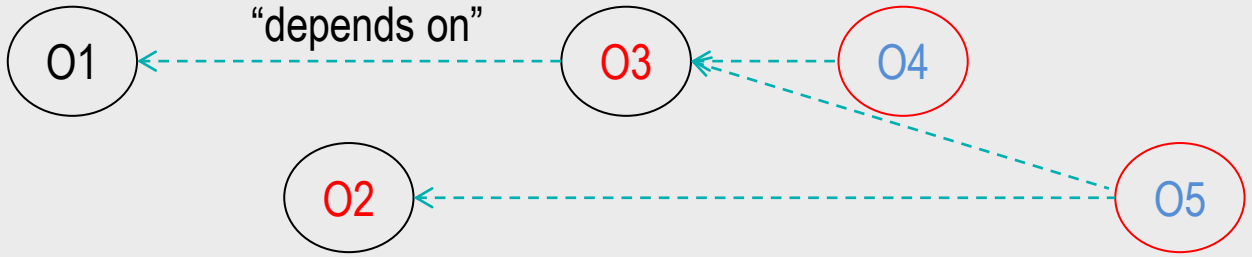
Exploiting Parallelism



Identify dependencies as in above example.

Specify starts & completions

Replace With ...



O1: P1

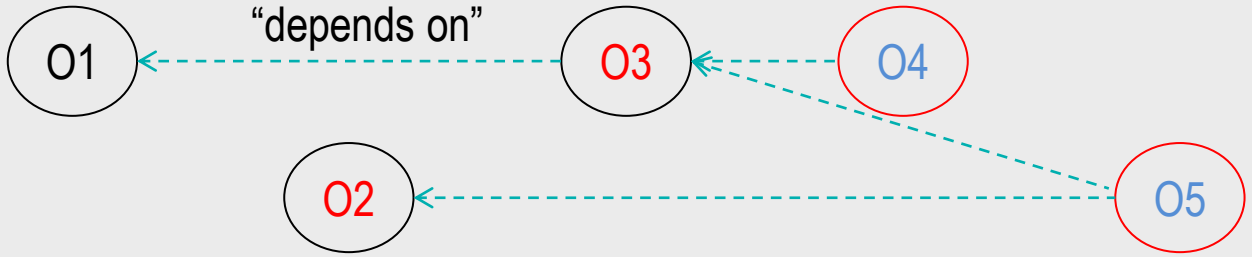
O2: Task t2 started, which implements P2

O3: Task t3 started, which implements P3

O3.5: t3 ended

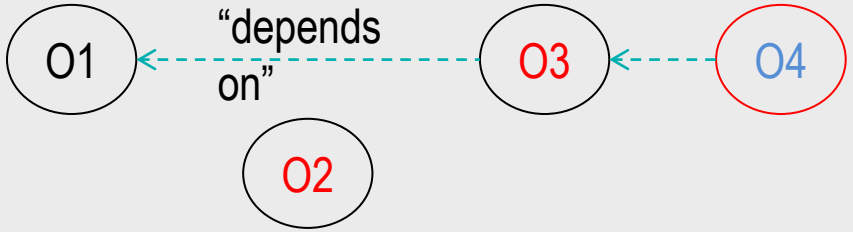
O4: …

Replace With ...



O1: P1
O2: Task t2 started, which implements P2
O3: Task t3 started, which implements P3
O3.5: t3 ended
O4: Task t4 started, which implements P4
O4.5: t2 ended
O5: Task t5 started, which implements P5
O5.5: t4 and t5 ended

Java Implementation



...

O2: Task t2 started, which implements P2

```
t2.start();
```

O3: Task t3 started, which implements P3

```
t3.start();
```

O3.5: t3 ended

```
while(t3.isAlive()){}   // wait
```

O4: ...

Divide-and-Conquer

```
def satisfy_desired_property_on(a_space):

    # Precondition: a_space can be decomposed into two* subspaces
    # of comparable magnitude,each either trivial, or else
    # of the same type as a_space

    # Postcondition: desired_property holds on a_space

     #---Oa (Solvable Immediately?):
     # EITHER desired_property holds on a_space AND this returned
     # OR a_space consists of similar non-empty space_1 and space_2

     #---Ob1: desired_property holds on space_1

     #---Ob2: desired_property holds on space_2

     #---Oc = Postcondition
```

Parallel Divide-and-Conquer

```
def satisfy_desired_property_on(a_space):

    # Precondition: ..

    # Postcondition: desired_property holds on a_space

     #---Oa (Solvable Immediately?):
     # EITHER desired_property holds on a_space AND this returned
     # OR a_space consists of similar non-empty space_1 and space_2

     #---Ob1p: Task t1 started, which implements
     #          “desired_property holds on space_1”

     #---Ob2p: Task t2 started, which implements
     #          “desired_property holds on space_2”

     #---Oc: t1 and t2 ended AND Postcondition
```

BOSTON UNIVERSITY
METROPOLITAN COLLEGE

Concurrency Agenda

1. Introduction
2. Creating Threads in Java
3. Visualizing & defining thread outcomes
4. **Coordinating threads**
5. Thread pools
6. Concurrent reading and writing

Joining

```
public class DemoJoin {
  public static void main(String[] args) {
    try {
      ThreadToJoin threadToJoin = new ThreadToJoin();
      threadToJoin.start();
      threadToJoin.join();
      for (int i = 0; i < 15; i++)
      {
        System.out.print(" M" + i);
      }
    }
    catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  private static class ThreadToJoin extends Thread {
    public void run() {
      for (int i = 0; i < 15; i++)
      {
        System.out.print(" C" + i);
      }
    }
  }
}
```

Output

```
 C0 C1 C2 C3 C4 C5 C6 C7 C8
C9 C10 C11 C12 C13 C14 M0 M1
M2 M3 M4 M5 M6 M7 M8 M9 M10
M11 M12 M13 M14
```

yield()

```
public class DemoYield {
  public static void main(String[] args) {
    ChildThread childThread = new ChildThread();
    childThread.start();
    for (int i = 0; i < 15; i++)
    {
      Thread.yield();
      System.out.print(" M" + i);
    }
  }

  private static class ChildThread extends Thread {
    public void run() {
      for (int i = 0; i < 15; i++)
      {
        System.out.print(" C" + i);
      }
    }
  }
}
```

Sample Run 1

```
 M0 C0 C1 C2 C3 M1 C4 C5 C6 C7 C8 C9
C10 C11 M2 C12 C13 M3 C14 M4 M5 M6 M7
M8 M9 M10 M11 M12 M13 M14
```

Sample Run 2

```
 M0 C0 M1 C1 M2 C2 M3 C3 M4 C4 C5 C6 C7
C8 C9 C10 C11 C12 C13 C14 M5 M6 M7 M8
M9 M10 M11 M12 M13 M14
```

Sleeping

```
public class DemoSleep {
  public static void main(String[] args) {
    try {
      long firstMilli = System.currentTimeMillis();
      Thread.sleep(1500);
      long secondMilli = System.currentTimeMillis();
      System.out.println((secondMilli-firstMilli) +
            " milliseconds have elapsed.");
    }
    catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Sample Run 1

```
1500 milliseconds have elapsed.
```

Sample Run 2

```
1504 milliseconds have elapsed.
```

BOSTON UNIVERSITY
METROPOLITAN COLLEGE

Concurrency Agenda

1. Introduction
2. Creating Threads in Java
3. Visualizing & defining thread outcomes
4. Coordinating threads
5. **Thread pools**
6. Concurrent reading and writing

The Need for Threadpools

- An application may need many tasks running in parallel
  - E.g., application servers, middleware servers, aggressive clients
  - thousands + threads
- Not efficient to continually decommission old threads and create new threads every time a new task is needed
- A threadpool reuses threads to avoid this

Executors and ExecutorService

- *Executors* class creates different types of threadpools:
  - Statically sized thread pool for a fixed number of threads.
  - Dynamically sized thread pool that reuses threads if available, and creates new ones as needed
  - Single threaded pool

- *ExecutorService* subclasses act as both a threadpool and a means of creating and interacting with threads.

```java
public class DemoThreadpool {
  public static void main(String[] args) {
    ExecutorService threadpool = Executors.newFixedThreadPool(2);
    threadpool.execute(new ChildThread("A"));
    threadpool.execute(new ChildThread("B"));
    threadpool.execute(new ChildThread("C"));
  }
  private static class ChildThread implements Runnable
    private String prefix;
    public ChildThread(String prefix) {
      this.prefix = prefix;
    }
    public void run() {
      for (int i = 0; i < 10; i++)
      {
        System.out.print(" " + prefix + i);
      }
    }
  }
}
```

Fixed Size Threadpool Demo

Output

```
 A0 B0 A1 B1 A2 B2
A3 A4 A5 A6 A7 B3
A8 B4 A9 B5 B6 B7
B8 B9 C0 C1 C2 C3
C4 C5 C6 C7 C8 C9
```

```java
public class DemoDynamicThreadpool {
  public static void main(String[] args) throws InterruptedException {
    ExecutorService threadpool = Executors.newCachedThreadPool();
    threadpool.execute(new ChildThread("A"));
    threadpool.execute(new ChildThread("B"));
    threadpool.execute(new ChildThread("C"));
    threadpool.shutdown();
    threadpool.awaitTermination(10000, TimeUnit.MILLISECOND
    System.out.println("\nAll threads are done.");
  }
  private static class ChildThread implements Runnable {
    private String prefix;
    public ChildThread(String prefix) {
      this.prefix = prefix;
    }
    public void run() {
      for (int i = 0; i < 10; i++)
      {
        System.out.print(" " + prefix + i);
      }
    }
  }
}
```

Dynamic Threadpool Demo

Output

```
 B0 C0 A0 C1 B1 C2 C3 C4 C5 C6
A1 A2 A3 A4 A5 A6 C7 B2 C8 A7
A8 A9 C9 B3 B4 B5 B6 B7 B8 B9
All threads are done.
```

BOSTON UNIVERSITY
METROPOLITAN COLLEGE

Concurrency Agenda

1. Introduction
2. Creating Threads in Java
3. Visualizing & defining thread outcomes
4. Coordinating threads
5. Thread pools
6. **Concurrent reading and writing**

Concurrent Reading and Writing Data

- **Threads reading data concurrently generally no problem**

- **Writing is problematical**
  - **because a different thread could read or write while writing is happening**
  - **result indeterminate.**

Solution

Prohibit other code from running while all or part of a designated method executes—using synchronize:

```
public synchronized void incrementCount() {
      // write to a variable
}

public void incrementCount() {
      // code
      synchronized(this) {
            // write to a variable
      }
      // code
}
```

For more flexibility ...

... see Lock interface and atomic classes.

Concurrency in the Real World

Most Apps

- Concurrency is used in most significant apps.

Two Primary Uses

- A primary use of concurrency is performance.
- A second is user-interface continuity and usability.

Multiple Kinds

- Apps often use multiple kinds of concurrency for different situations.

Debugging is Hard

- It can be hard to debug multi-threaded apps.
- Some strategies include:
  - temporarily allowing only one thread to run its code.
  - debug statements that can be turned on or off as needed.
  - conceptual diagrams or walkthroughs.