

# Multithreading and Concurrency

---



**Jim Wilson**

MOBILE SOLUTIONS DEVELOPER & ARCHITECT

@hedgehogjim blog.jwhh.com



# Overview



Single threading vs. multithreading

Threading foundation types

Thread pools

Concurrency issues

Coordinating method access

Manual thread synchronization

Concurrency related types & packages

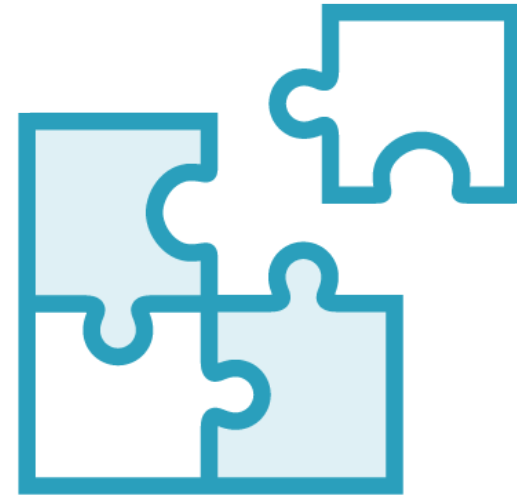


# Threading and Concurrency Coverage



## New to threading

Provide the building blocks you need to begin building your understanding of threading and concurrency



## Experienced with threading

Provide the necessary understanding of Java threading and concurrency to enable you apply your existing knowledge in Java



# A Quick Look at the Basics

## Process

- Instance of a program/application
- Has resources such as memory, etc.
- Has at least one thread

## Thread

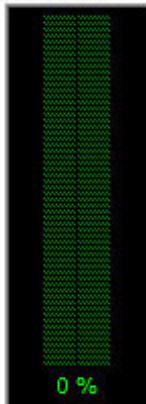
- Sequence of programmed instructions
- The thing that executes a program's code
- Utilizes process resources



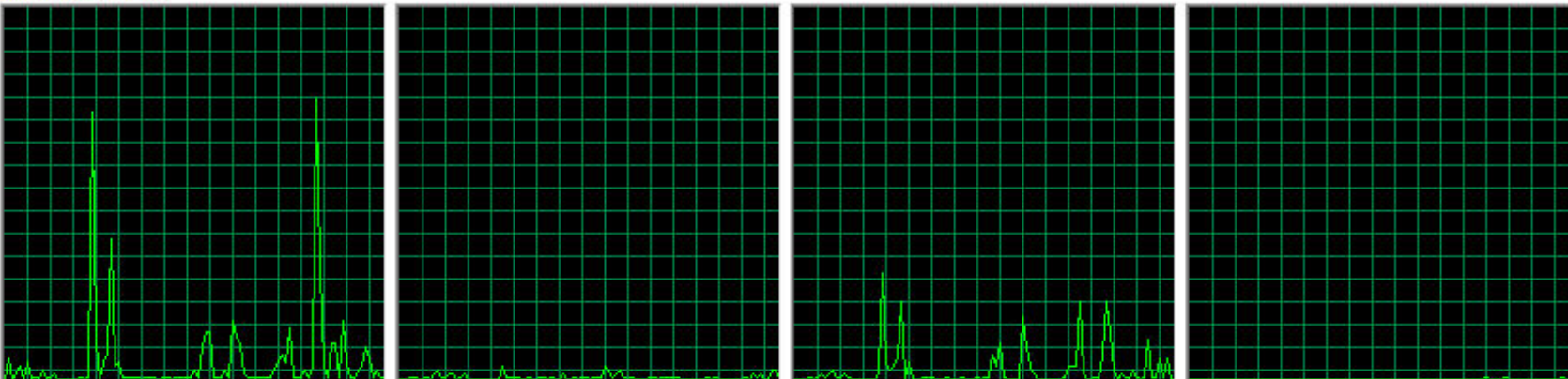
File Options View Help

Applications Processes Services Performance Networking Users

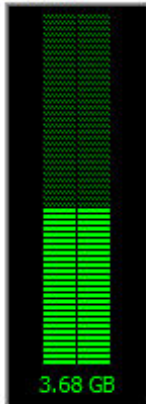
CPU Usage



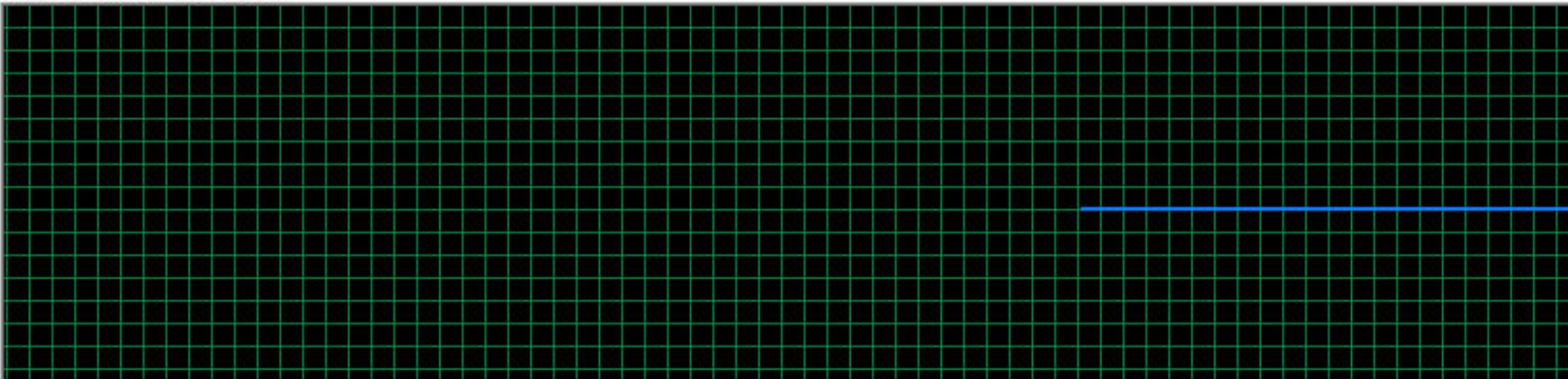
CPU Usage History



Memory



Physical Memory Usage History



Physical Memory (MB)

Total	8148
Cached	2676
Available	4372
Free	1882

Kernel Memory (MB)

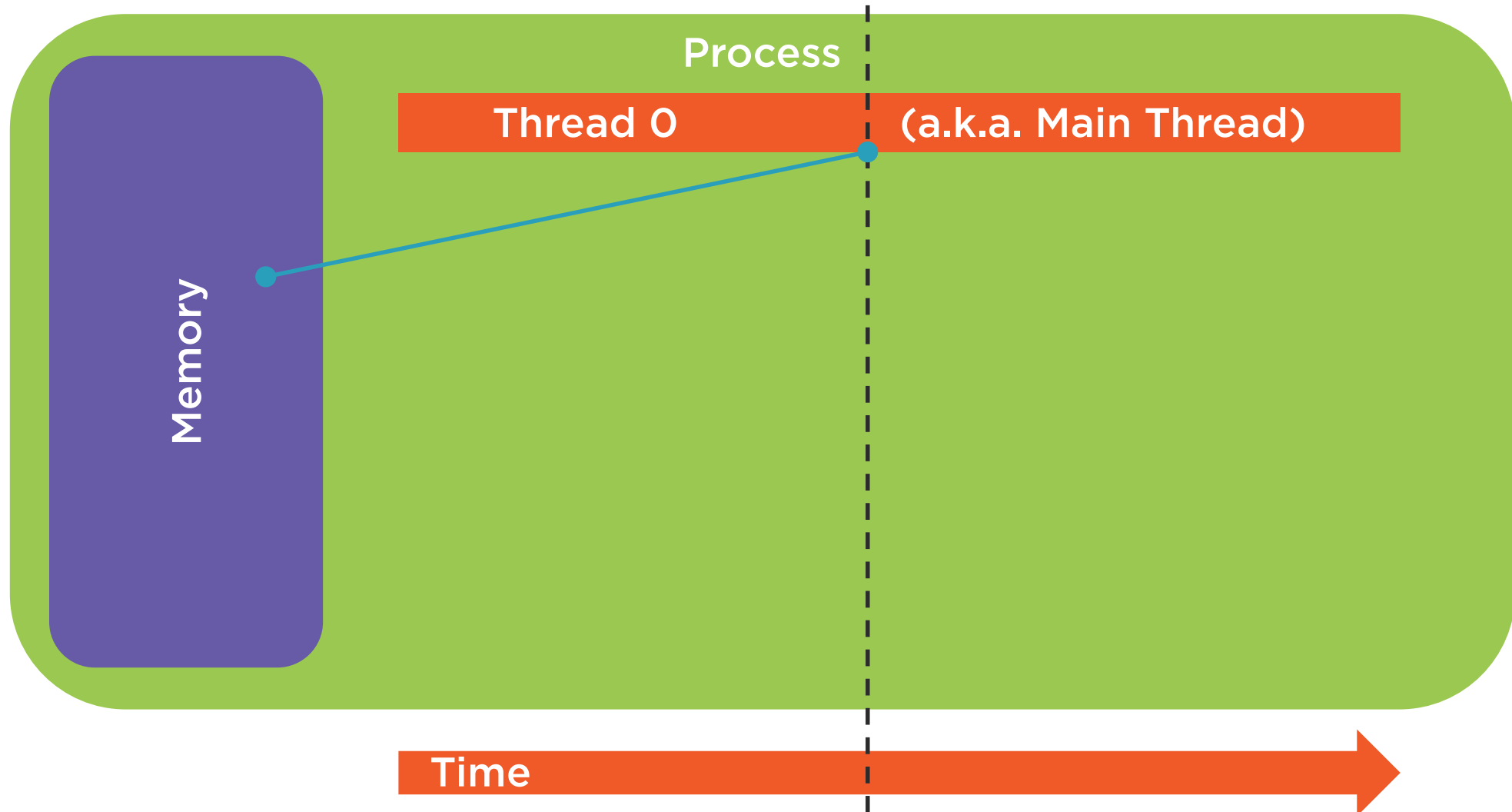
Paged	265
Nonpaged	122

System

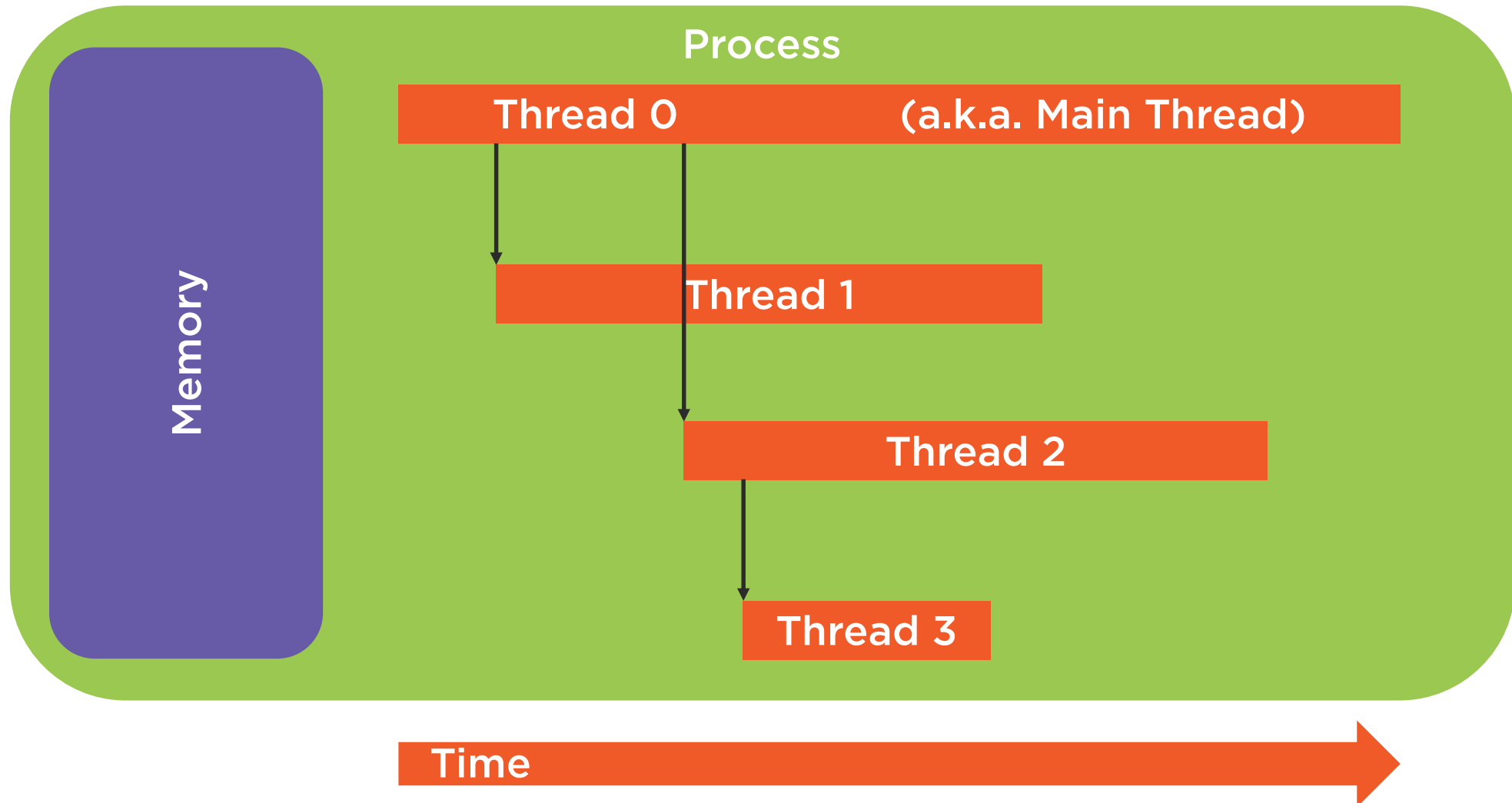
Handles	38644
Threads	1394
Processes	108
Up Time	0:01:22:24
Commit (GB)	5 / 15

Resource Monitor...

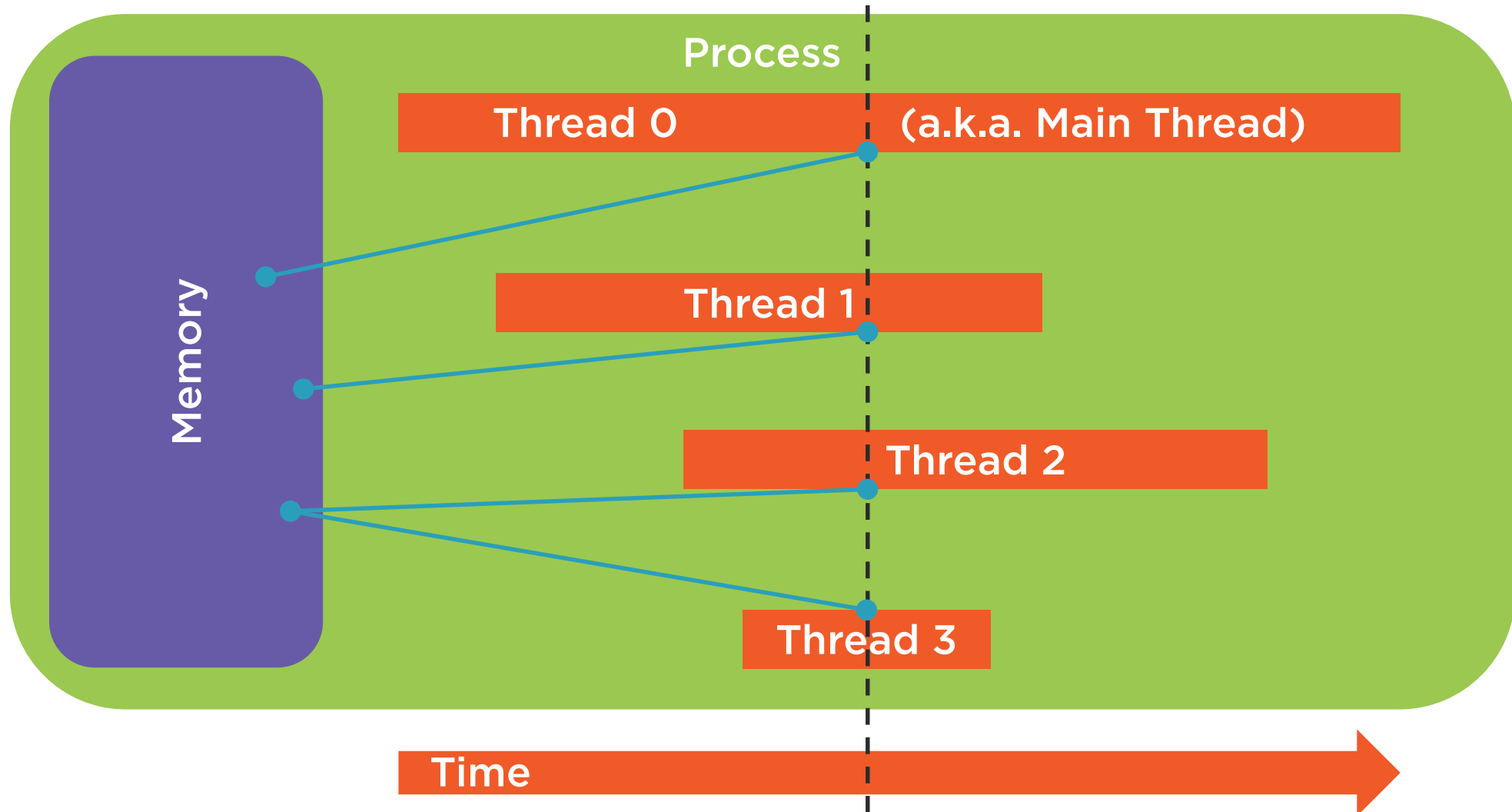
# Single Threaded Process



# Multithreading



# Concurrency





# The Case for Multithreading

## Can enable more complete CPU use

- Threads often wait on non-CPU tasks
  - Interacting with storage, networks, etc.
- Most computers have multiple CPU cores
  - Allows things to run in parallel

## Why does any of this matter?

- Can reduce perceived execution times
  - Less wall-clock time passes



# A Simple Adder Class

```
class Adder {  
    private String inFile, outFile;  
    public Adder(String inFile, String outFile) { /* assign filenames to member fields */ }  
    public void doAdd()                throws IOException {  
        int total = 0;  
        String line = null;  
  
        try (BufferedReader reader = Files.newBufferedReader(Paths.get(inFile))) {  
            while ((line = reader.readLine()) != null)  
                total += Integer.parseInt(line);  
        }  
  
        try (BufferedWriter writer = Files.newBufferedWriter(Paths.get(outFile))) {  
            writer.write("Total: " + total);  
        }  
    }  
}
```



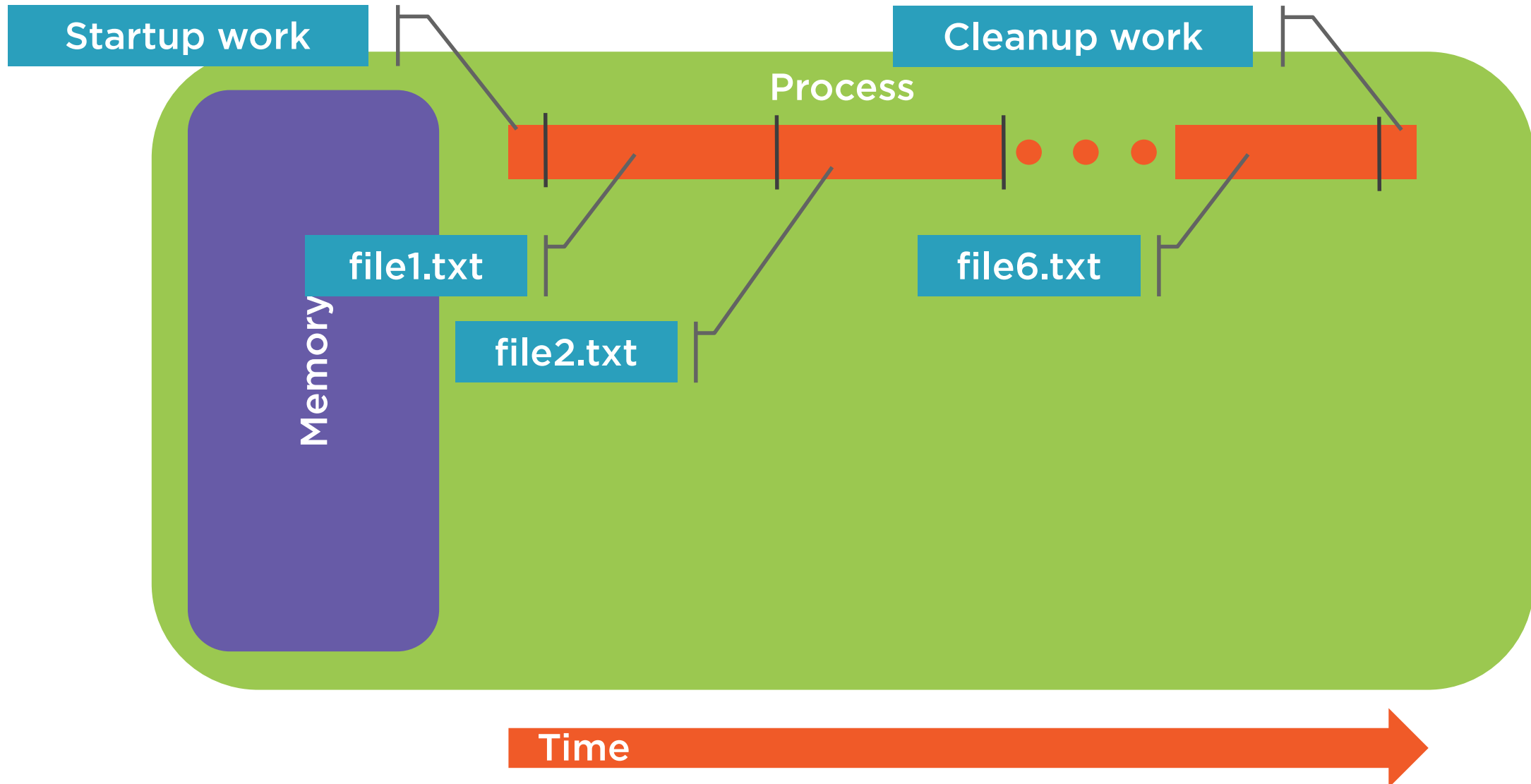
# Using Simple Adder Class

```
String[] inFiles = {"/file1.txt", ... "/file6.txt"};
String[] outFiles = {"/file1.out.txt", ... "/file6.out.txt"};

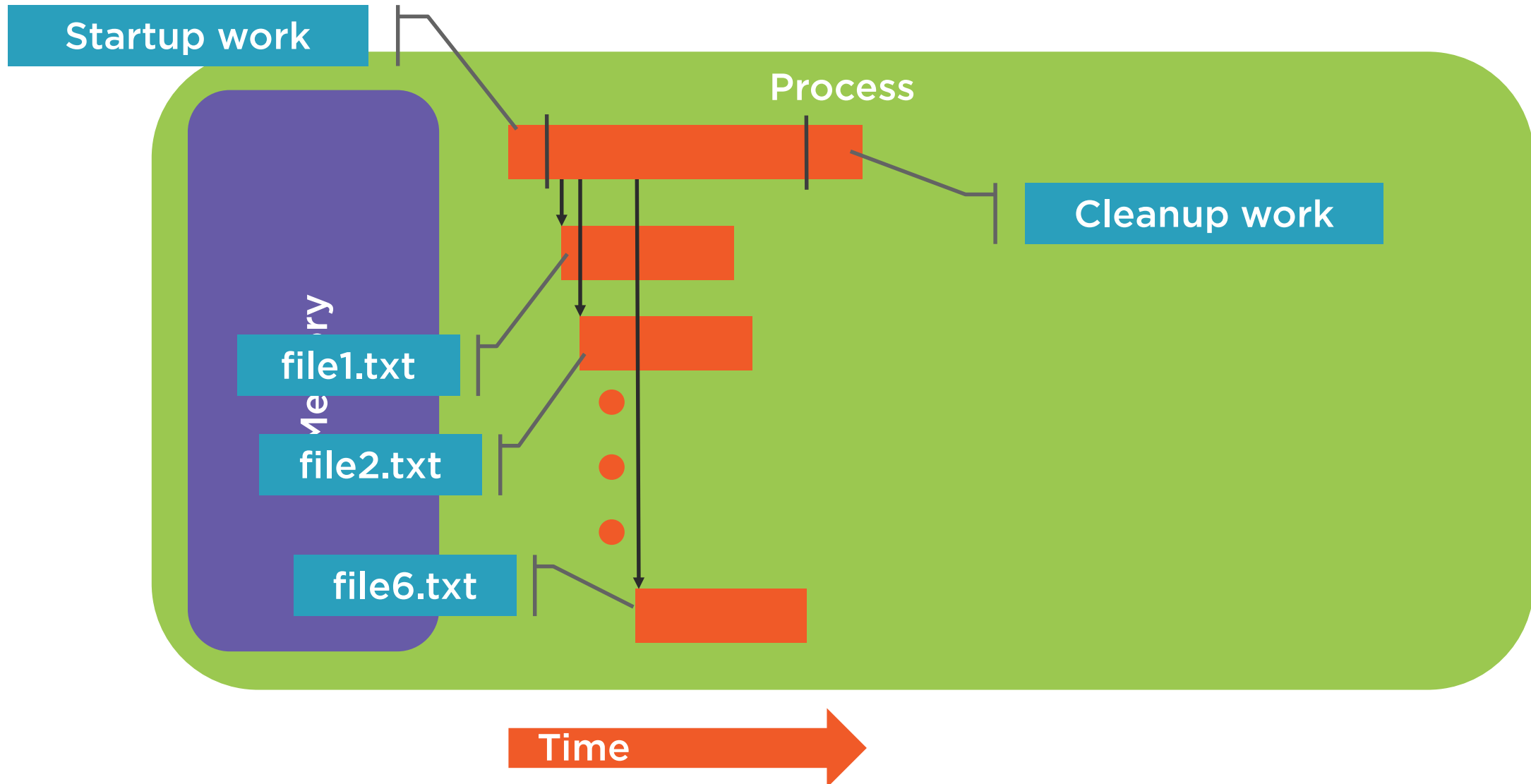
try {
    for(int i=0; i < inFiles.length; i++) {
        Adder adder = new Adder(inFiles[i], outFiles[i]);
        adder.doAdd();
    }
} catch(IOException e) {
    // do something
}
```



# Processing on a Single Thread



# Processing on Multiple Threads



# The Move to Multithreading

## **Multithreading is an explicit choice**

- Must break the problem into parts
- Must handoff the parts for processing

## **Java provides differing levels of abstraction**

- Supports very direct handling
  - Manual creation & coordination
- Supports higher level handling
  - Simplified creation & coordination



# Java Threading Foundation

## Limited threading abstraction

- Very close to the standard OS behavior
- Each thread started for a specific task
  - Terminates at end of task

## Requires explicit management

- Responsible to manage coordination

## Exceptions tied to thread

- Each thread must handle own exceptions



# Threading Foundation Types

## Runnable interface

- Represents a task to be run on a thread
- Only member is the run method

## Thread class

- Represents a thread of execution
- Can interact with and effect thread state
- Begin execution with start method





# Adder with Threading Support

```
class Adder    implements Runnable {  
    private String inFile, outFile;  
    public Adder(String inFile, String outFile) { ... }  
    public void doAdd() throws IOException { ... }  
    public void run() {  
        try {  
            doAdd();  
        } catch(IOException e) { ... }  
    }  
}
```



# Running Adder on Separate Threads

```
String[] inFiles = {"/file1.txt", ... "/file6.txt"};
String[] outFiles = {"/file1.out.txt", ... "/file6.out.txt"};

try {
    for(int i=0; i < inFiles.length; i++) {
        Adder adder = new Adder(inFiles[i], outFiles[i]);
        adder.doAdd();
    }
} catch(IOException e) {
    // do something
}
```



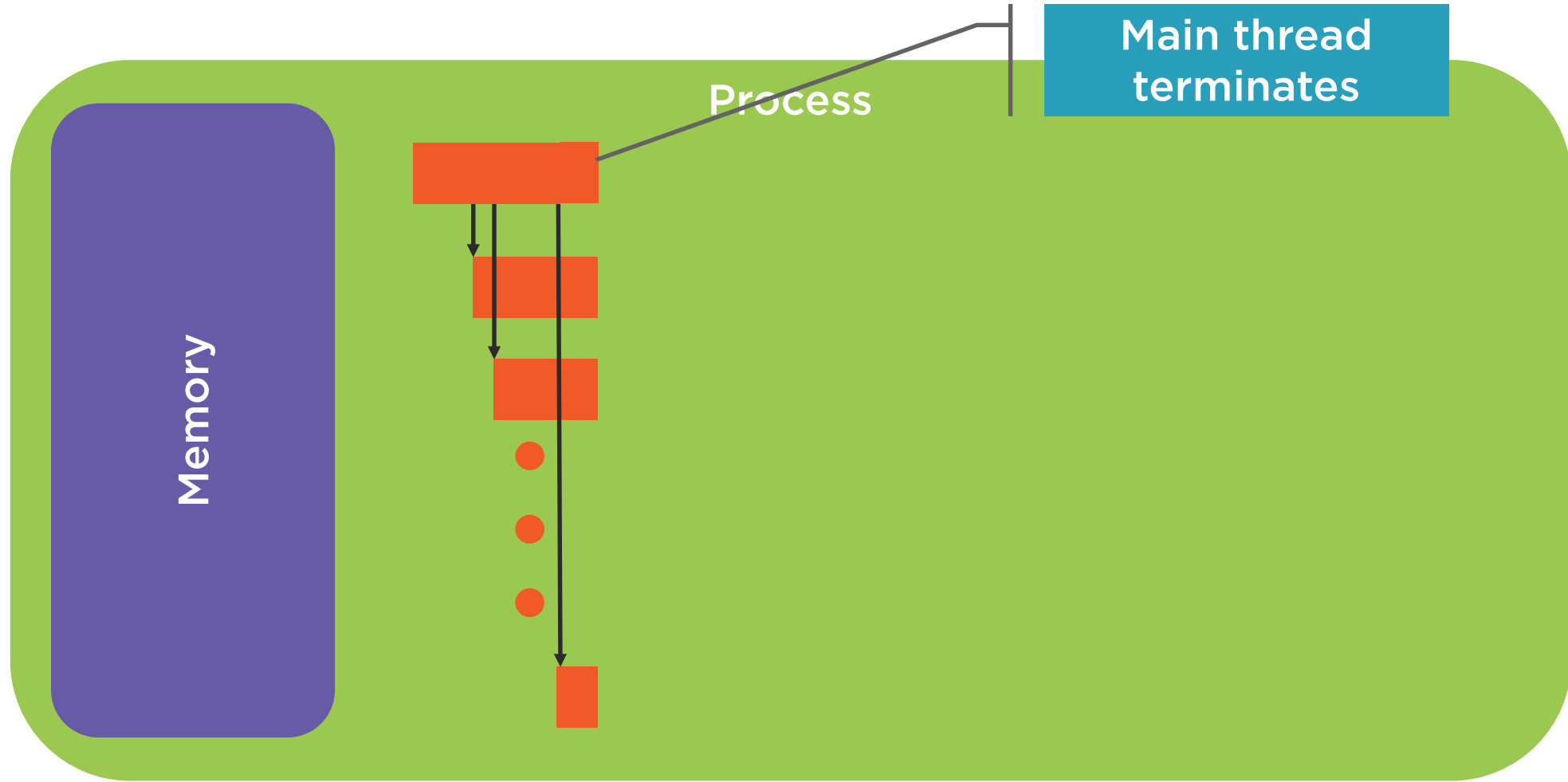
# Running Adder on Separate Threads

```
String[] inFiles = {"/file1.txt", ... "/file6.txt"};
String[] outFiles = {"/file1.out.txt", ... "/file6.out.txt"};

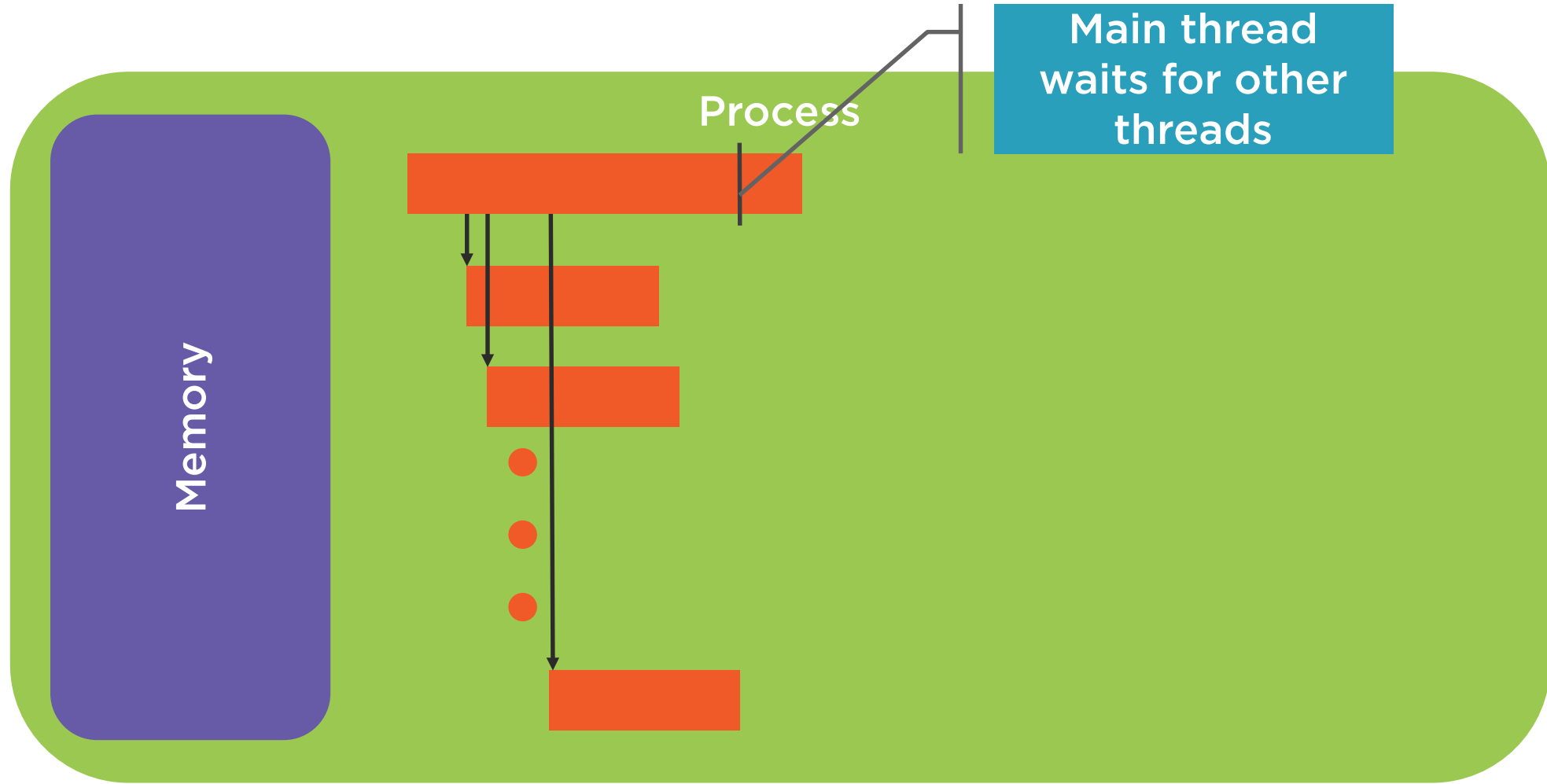
try {
    for(int i=0; i < inFiles.length; i++) {
        Adder adder = new Adder(inFiles[i], outFiles[i]);
        Thread thread = new Thread(adder);
        thread.start();
    }
} catch(IOException e) {
    // do something
}
```



# Processing on Multiple Threads



# Processing on Multiple Threads



# Running Adder on Separate Threads

```
String[] inFiles = {"/file1.txt", ... "/file6.txt"};
String[] outFiles = {"/file1.out.txt", ... "/file6.out.txt"};

Thread[] threads = new Thread[inFiles.length];

for(int i=0; i < inFiles.length; i++) {
    Adder adder = new Adder(inFiles[i], outFiles[i]);
    Thread thread = new Thread(adder);
    thread.start();
}
```



# Running Adder on Separate Threads

```
String[] inFiles = {"/file1.txt", ... "/file6.txt"};
String[] outFiles = {"/file1.out.txt", ... "/file6.out.txt"};

Thread[] threads = new Thread[inFiles.length];

for(int i=0; i < inFiles.length; i++) {
    Adder adder = new Adder(inFiles[i], outFiles[i]);
    threads[i] = new Thread(adder);
    threads[i].start();
}

for(Thread thread:threads)
    thread.join(); // Blocks waiting for thread completion
```



# Thread Management Details

## Value of the Thread class

- Allows direct control over thread startup, shutdown, & coordination

## Challenge of the Thread class

- Responsible to efficiently manage thread startup, shutdown & coordination
- Easily misused





# Abstracting Thread Management with Thread Pools

## Java offers thread pools

- Creates a queue for tasks
- Assigns tasks into a pool of threads
- Handles details of managing threads



# Thread Pool Types

## **ExecutorService interface**

- Models thread pool behavior
- Can submit tasks
- Request and wait for pool shutdown

## **Executors class**

- Methods for creating thread pools
  - Dynamically sized pools
  - Size limited pools
  - Pools that schedule tasks for later



# Running Adder on Separate Threads

```
String[] inFiles = {"/file1.txt", ... "/file6.txt"};
String[] outFiles = {"/file1.out.txt", ... "/file6.out.txt"};

Thread[] threads = new Thread[inFiles.length];

for(int i=0; i < inFiles.length; i++) {
    Adder adder = new Adder(inFiles[i], outFiles[i]);
    threads[i] = new Thread(adder);
    threads[i].start();
}

for(Thread thread:threads)
    thread.join(); // Blocks waiting for thread completion
```



# Running Adder in a Thread Pool

```
String[] inFiles = {"/file1.txt", ... "/file6.txt"};
String[] outFiles = {"/file1.out.txt", ... "/file6.out.txt"};

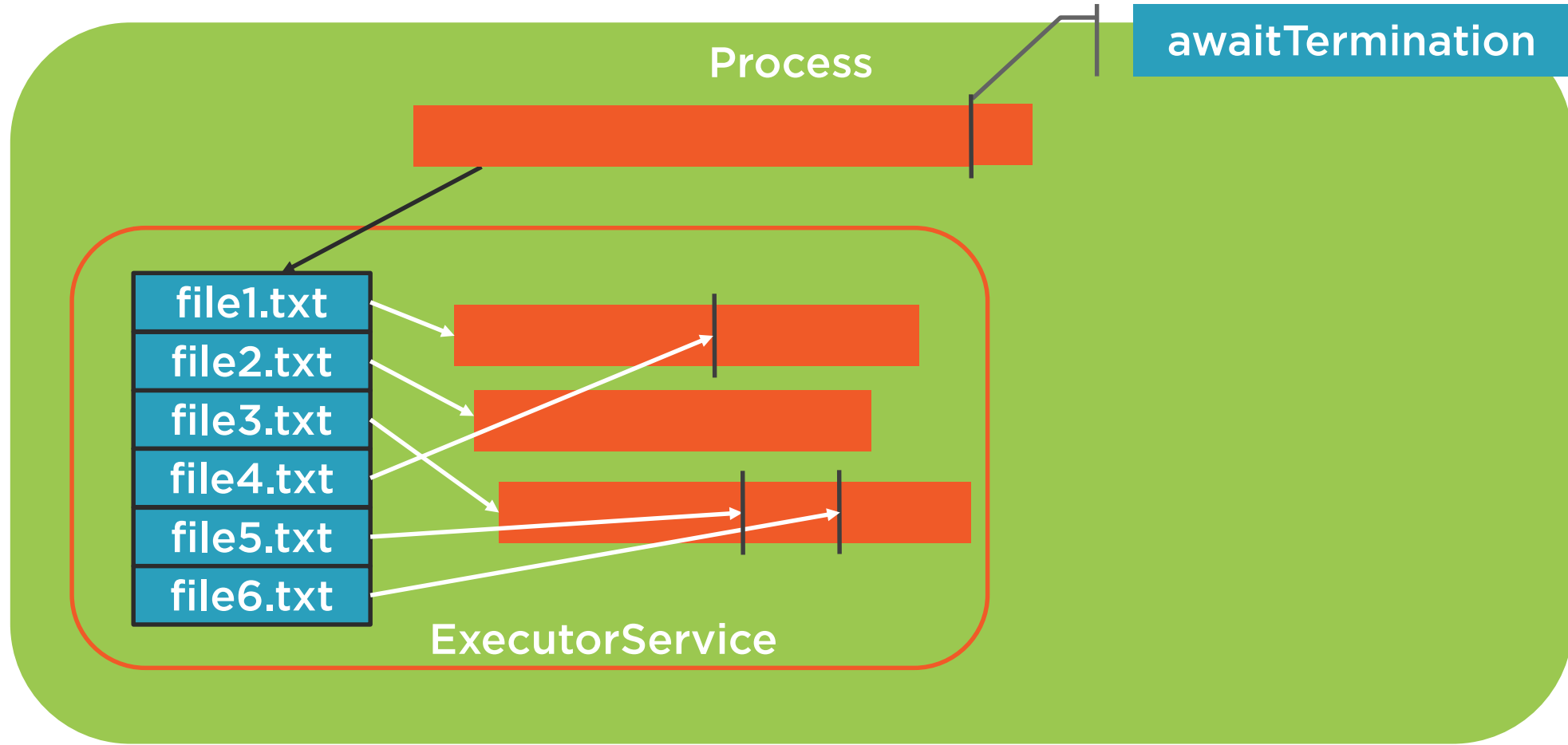
ExecutorService es = Executors.newFixedThreadPool(10);

for(int i=0; i < inFiles.length; i++) {
    Adder adder = new Adder(inFiles[i], outFiles[i]);
    es.submit(adder);
}

try {
    es.shutdown();
    es.awaitTermination(60, TimeUnit.SECONDS);
} catch (Exception e) { ... }
```



# Processing in a Thread Pool



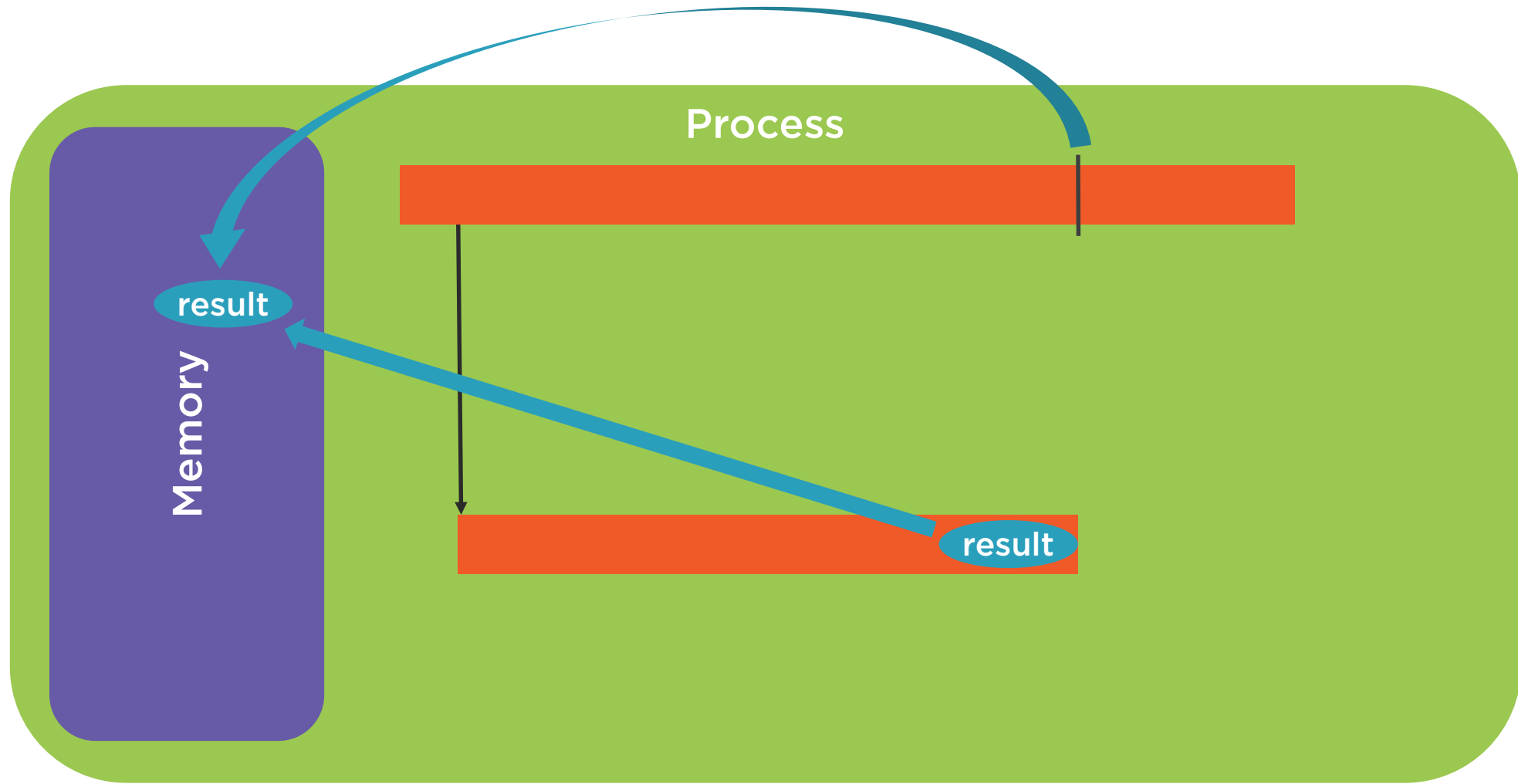
# Creating a Closer Relationship Between Thread Tasks

## **Multithreading not always loosely coupled**

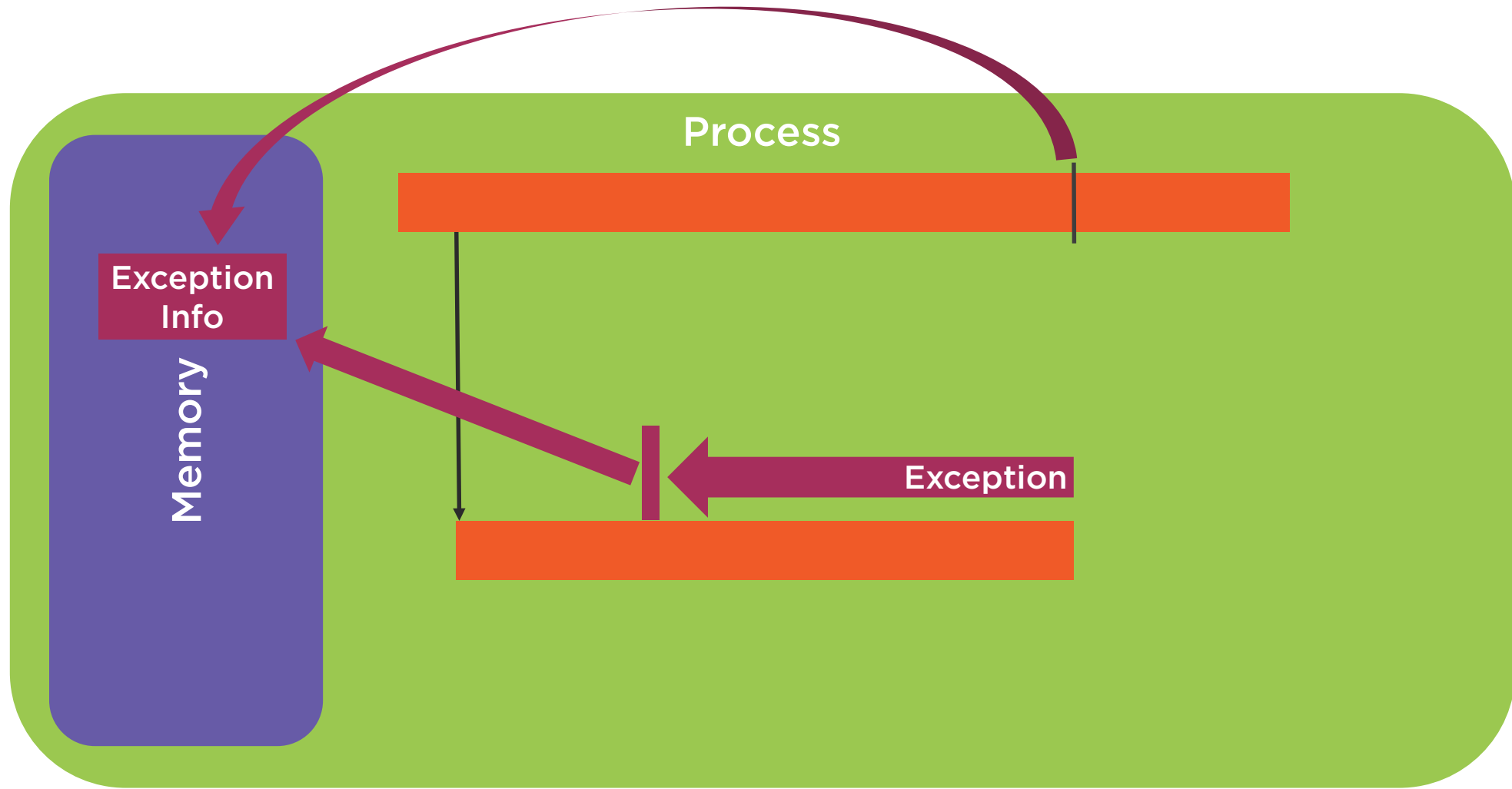
- Caller may need results from worker
- May need to know if task succeeded



# Thread Result Manual Handling

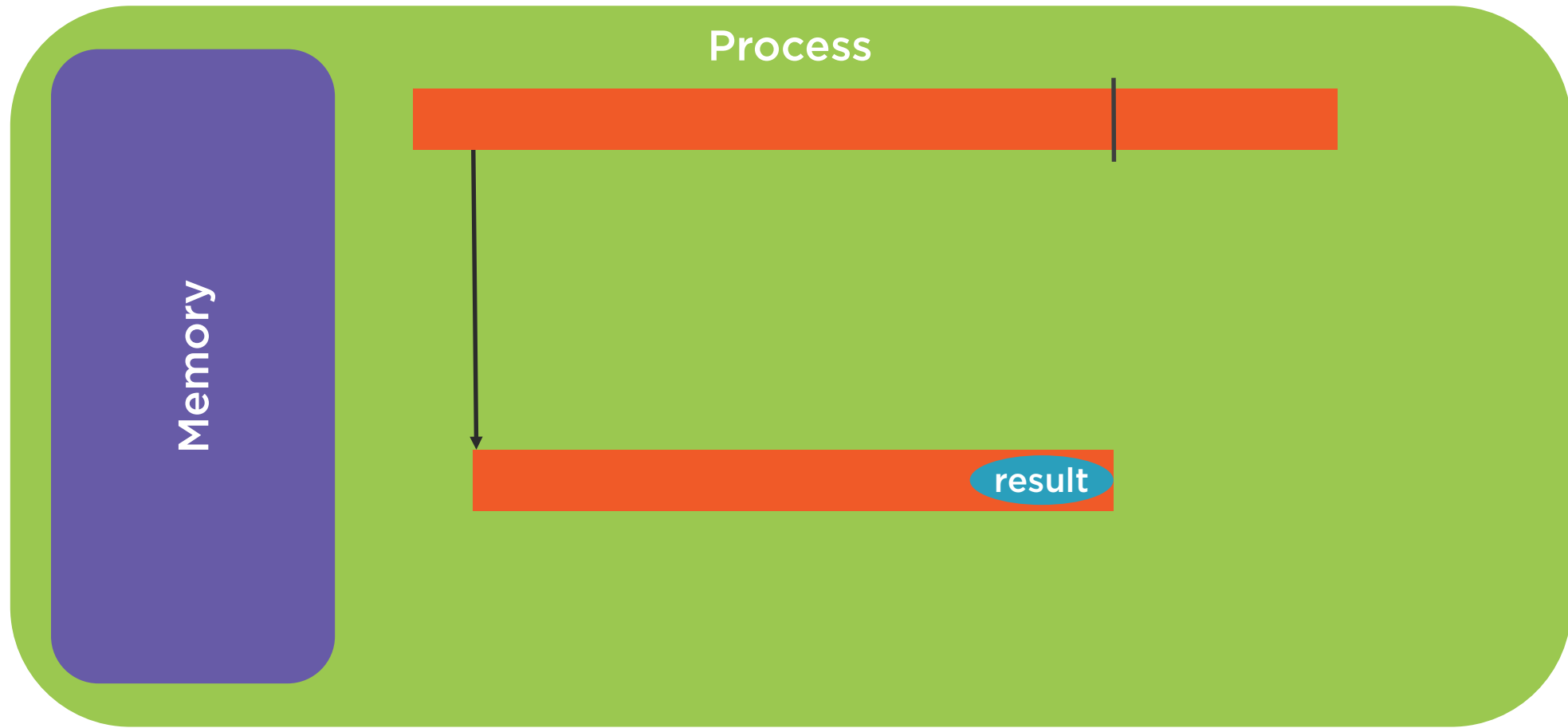


# Thread Exception Manual Handling

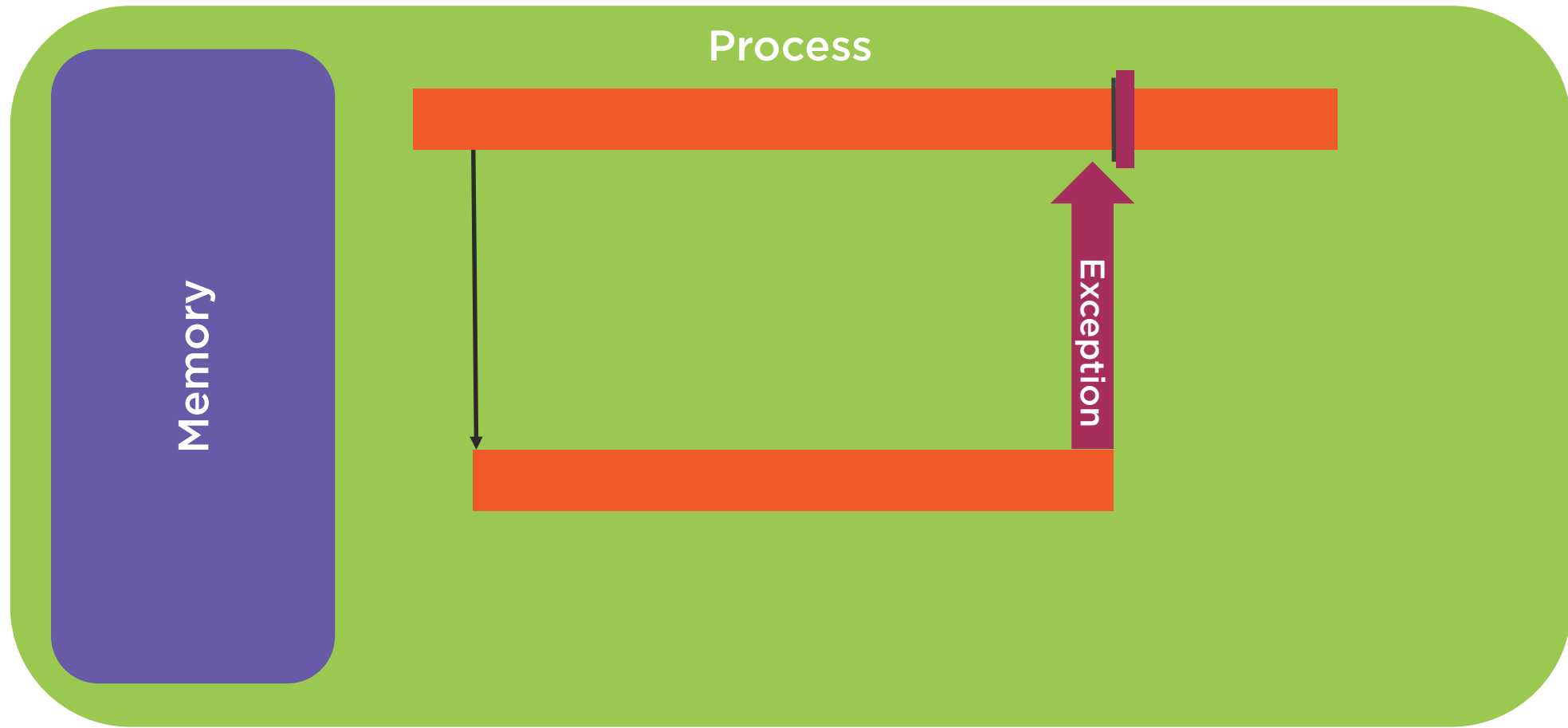




# Thread Result Handling Desired



# Thread Exception Handling Desired



# Threading Relationship Types

## Callable interface

- Represents a task to be run on a thread
  - Can return results
  - Can throw exceptions
- Only member is the call method

## Future interface

- Represents results a thread task
  - Returned by `ExecutorService.submit`
- The key method is `get`
  - Blocks until task completes
  - Returns Callable interface result
  - Throws Callable interface exception



# Adder Method Returning a Value

```
public void doAdd() throws IOException
{
    int total = 0;
    String line = null;

    try (BufferedReader reader = Files.newBufferedReader(Paths.get(inFile))) {
        while ((line = reader.readLine()) != null)
            total += Integer.parseInt(line);
    }

    try (BufferedWriter writer = Files.newBufferedWriter(Paths.get(outFile))) {
        writer.write("Total: " + total);
    }
}
```



# Adder Method Returning a Value

```
public int l() throws IOException
{
    int total = 0;
    String line = null;

    try (BufferedReader reader = Files.newBufferedReader(Paths.get(inFile))) {
        while ((line = reader.readLine()) != null)
            total += Integer.parseInt(line);
    }

    try (BufferedWriter writer = Files.newBufferedWriter(Paths.get(outFile))) {
        writer.write("Total: " + total);
    }
}
```



# Adder Method Returning a Value

```
public int l() throws IOException
{
    int total = 0;
    String line = null;

    try (BufferedReader reader = Files.newBufferedReader(Paths.get(inFile))) {
        while ((line = reader.readLine()) != null)
            total += Integer.parseInt(line);
    }

    return total;
}
```



# Adder Implementing Callable

```
class Adder implements Runnable {  
    private String inFile;  
    public Adder(String inFile) { ... }  
    public int doAdd() throws IOException { ... }  
    public void run() {  
        try {  
            doAdd();  
        } catch(IOException e) { ... }  
    }  
}
```



# Adder Implementing Callable

```
class Adder implements      Runnable { <Integer> {  
    private String inFile;  
    public Adder(String inFile) { ... }  
  
    public int doAdd() throws IOException { ... }  
  
    public      Integer      call()      throws IOException {  
  
        return doAdd();  
  
    }  
}
```





# Start Adder Processing

```
String[] inFiles = {"/file1.txt", ... "/file6.txt"};

ExecutorService es = Executors.newFixedThreadPool(3);

for(int i=0; i < inFiles.length; i++) {
    Adder adder = new Adder(inFiles[i]);
    es.submit(adder);
}
```



# Start Adder Processing

```
String[] inFiles = {"/file1.txt", ... "/file6.txt"};
ExecutorService es = Executors.newFixedThreadPool(3);
Future<Integer>[] results = new Future[inFiles.length];
for(int i=0; i < inFiles.length; i++) {
    Adder adder = new Adder(inFiles[i]);
    results[i] = es.submit(adder);
}
```



# Retrieving Adder Class Results

```
for(Future<Integer> result:results) {  
    try {  
        int value = result.get(); // blocks until return value available  
        System.out.println("Total: " + value);  
    } catch(ExecutionException e) { // Exception raised in Adder  
        Throwable adderEx = e.getCause(); // Get the Adder exception  
        // Do something with adderEx  
    } catch(Exception e) { . . . } // Non-Adder exceptions  
}  
es.shutdown();
```



# Concurrency Issues

## The challenge of concurrency

- Threads sometimes share resources
- No problem if resources only read
- Changes must be coordinated

## Failure to coordinate can create problems

- Receive wrong results
- Crash the program



# A Simple Bank Account Class

```
public class BankAccount {  
    private int balance;  
    public BankAccount(int startBalance) {  
        balance = startBalance;  
    }  
  
    public int getBalance() {  
        return balance;  
    }  
  
    public void deposit(int amount) {  
        balance += amount;  
    }  
}
```



# A Class to Update the Bank Account

```
public class Worker implements Runnable {  
    private BankAccount account;  
    public Worker(BankAccount account) {  
        this.account = account;  
    }  
    public void run() {  
        for(int i=0; i < 10; i++) {  
            int startBalance = account.getBalance();  
            account.deposit(10);  
            int endBalance = account.getBalance();  
        }  
    }  
}
```



# Running on a Single Thread

```
ExecutorService es = Executors.newFixedThreadPool(5);
```

```
BankAccount account = new BankAccount(100);
```

```
Worker worker = new Worker(account);
```

```
es.submit(worker);
```

```
// Shutdown es and wait
```

```
End Balance: 110 StartBalance: 100
```

```
End Balance: 120 StartBalance: 110
```

```
End Balance: 130 StartBalance: 120
```

```
End Balance: 140 StartBalance: 130
```

```
End Balance: 150 StartBalance: 140
```

```
End Balance: 160 StartBalance: 150
```

```
End Balance: 170 StartBalance: 160
```

```
End Balance: 180 StartBalance: 170
```

```
End Balance: 190 StartBalance: 180
```

```
End Balance: 200 StartBalance: 190
```



# Running on Multiple Threads


```
ExecutorService es = Executors.newFixedThreadPool(5);  
BankAccount account = new BankAccount(100);  
for(int i = 0; i < 5; i++) {  
    Worker worker = new Worker(account);  
    es.submit(worker);  
}  
// Shutdown es and wait
```





# 1 Thread vs. 5 Threads

100    +10   +10   +10   +10   +10   +10   +10   +10   +10   +10   = 200



1 Thread

100    +10   +10   +10   +10   +10   +10   +10   +10   +10   +10

+10   +10   +10   +10   +10   +10   +10   +10   +10   +10

+10   +10   +10   +10   +10   +10   +10   +10   +10   +10

+10   +10   +10   +10   +10   +10   +10   +10   +10   +10

+10   +10   +10   +10   +10   +10   +10   +10   +10   +10   = 600 500 500 500



5 Threads



# What Happened on the 5 Threads

End Balance: 110 Start Balance: 100 Worker: 1

End Balance: 120 Start Balance: 110 Worker: 2

End Balance: 130 Start Balance: 120 Worker: 3

End Balance: 140 Start Balance: 130 Worker: 4

End Balance: 150 Start Balance: 140 Worker: 5

End Balance: 160 Start Balance: 150 Worker: 5

End Balance: 170 Start Balance: 160 Worker: 3

End Balance: 170 Start Balance: 160 Worker: 2

•  
•  
•

End Balance: 510 Start Balance: 500 Worker: 4

End Balance: 520 Start Balance: 510 Worker: 5

End Balance: 520 Start Balance: 510 Worker: 1

End Balance: 530 Start Balance: 520 Worker: 2

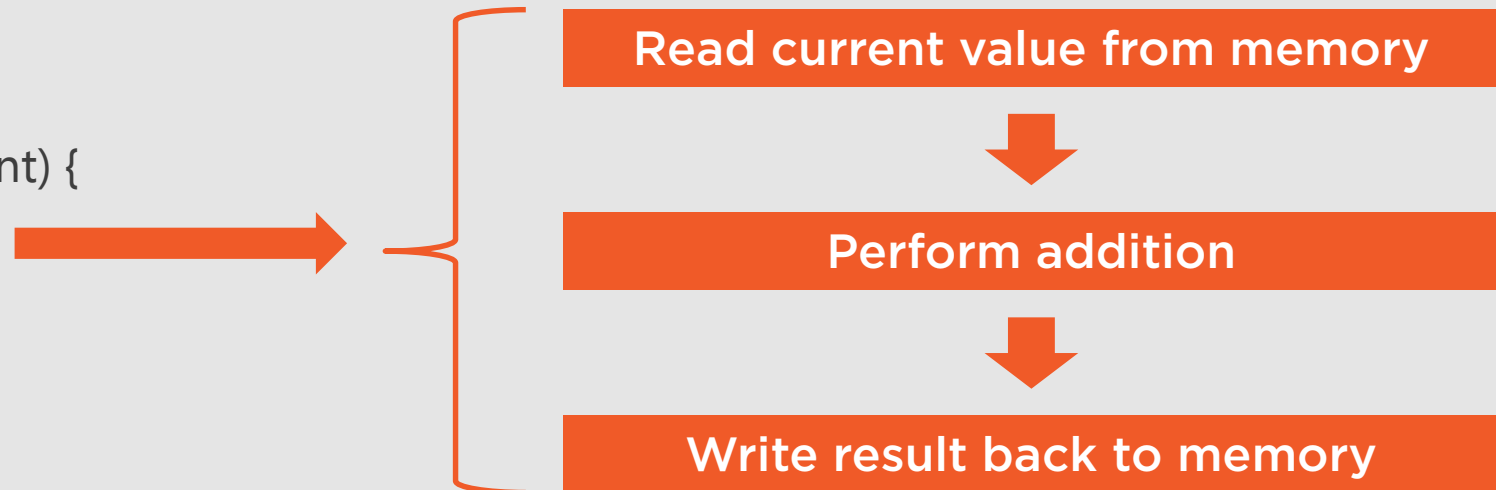
End Balance: 540 Start Balance: 530 Worker: 3

End Balance: 550 Start Balance: 540 Worker: 3

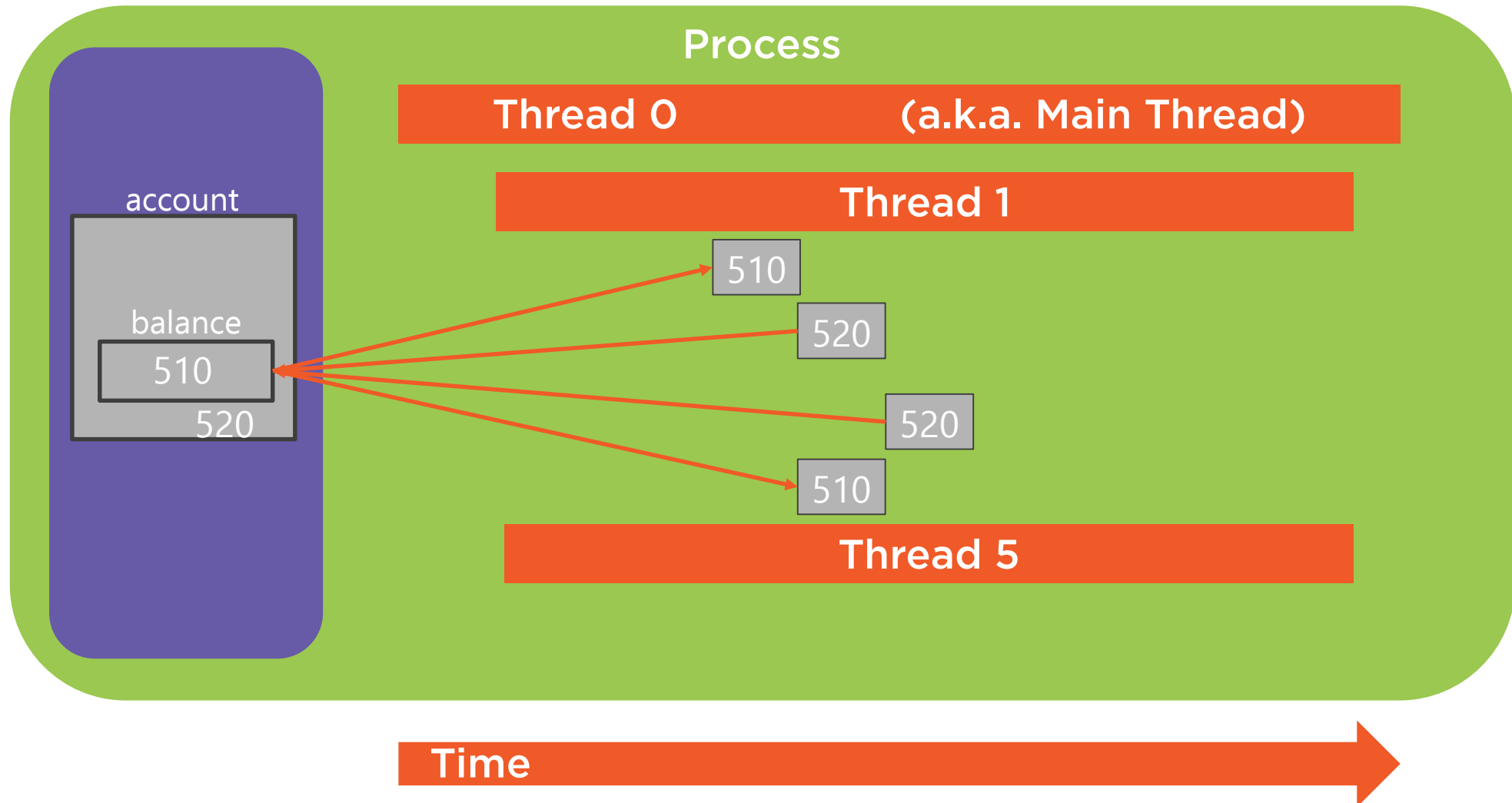


# There's More Than Meets the Eye

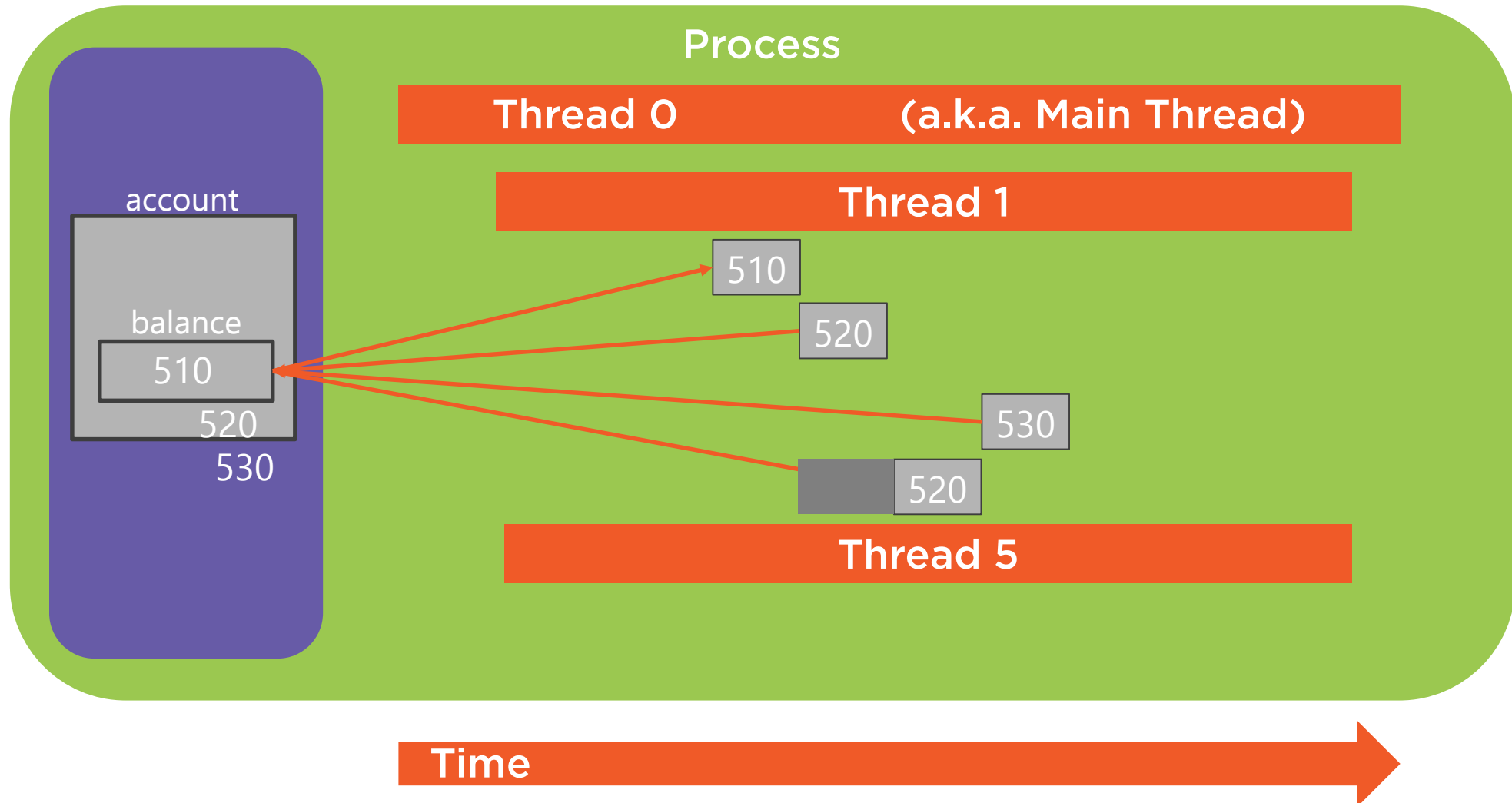
```
public class BankAccount {  
    private int balance;  
    public BankAccount(int startBalance) {  
        balance = startBalance;  
    }  
  
    public int getBalance() {  
        return balance;  
    }  
  
    public void deposit(int amount) {  
        balance += amount;  
    }  
}
```



# Unprotected Concurrency



# Coordinated Concurrency



# Coordinating Method Access

## Synchronized methods

- Coordinate thread access to methods
- Use synchronized method modifier
  - Class can have as many as needed

## Synchronization managed per instance

- No more than one thread can be in any synchronized method at a time



# Using Synchronized Methods

## When to use synchronized

- Protect modification by multiple threads
- Reading value that might be modified by another thread

## Why not always synchronize methods

- Has significant overhead
- Use only in multithreading scenarios

## Constructors are never synchronized

- A given object instance always created on exactly one thread



# Synchronized Methods on Bank Account Class

```
public class BankAccount {  
    private int balance;  
    public BankAccount(int startBalance) {  
        balance = startBalance;  
    }  
  
    public    synchronized    int getBalance() {  
        return balance;  
    }  
  
    public    synchronized    void deposit(int amount) {  
        balance += amount;  
    }  
}
```





# 5 Threads Running Correctly


100    +10   +10   +10   +10   +10   +10   +10   +10   +10   +10

         +10   +10   +10   +10   +10   +10   +10   +10   +10   +10

         +10   +10   +10   +10   +10   +10   +10   +10   +10   +10

         +10   +10   +10   +10   +10   +10   +10   +10   +10   +10

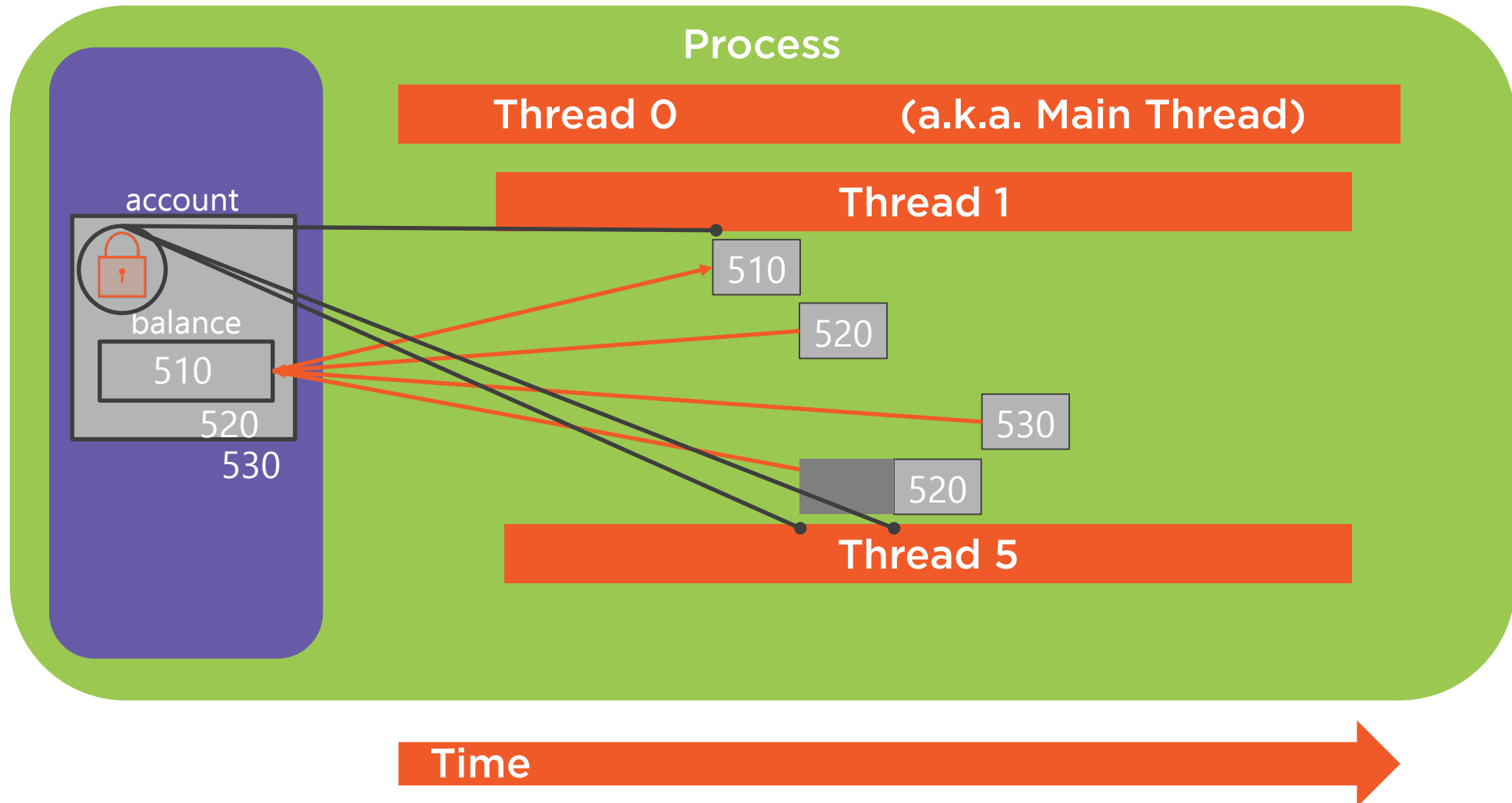
         +10   +10   +10   +10   +10   +10   +10   +10   +10   +10 = 600



5 Threads



# Behavior of Synchronized Methods



# Manual Synchronization

## Synchronized methods

- Automated concurrency management
- Used lock of current object instance

## All Java objects have a lock

- Can manually acquire that lock
  - Use synchronized statement block
  - Available to any code with a reference



# Synchronized Method

```
class BankAccount {  
    private int balance;  
    // other members elided for clarity  
    public synchronized void deposit(int amount) {  
        balance += amount;  
    }  
}
```

```
class Worker implements Runnable {  
    private BankAccount account;  
    // other members elided for clarity  
    public void run() {  
        for(int i=0; i<10; i++) {  
            account.deposit(10);  
        }  
    }  
}
```



# Synchronized Statement Block

```
class BankAccount {  
    private int balance;  
    // other members elided for clarity  
    public void deposit(int amount) {  
        balance += amount;  
    }  
}
```

```
class Worker implements Runnable {  
    private BankAccount account;  
    // other members elided for clarity  
    public void run() {  
        for(int i=0; i<10; i++) {  
            synchronized(account) {  
                account.deposit(10);  
            }  
        }  
    }  
}
```



# Why Use Synchronized Statement Blocks

## **Synchronized blocks provide flexibility**

- Enables use of non-thread safe classes
- Can protect complex blocks of code
- Sometimes synchronized methods just aren't enough



# Bank Account Class Revisited

```
public class BankAccount {  
    private int balance;  
    public BankAccount(int startBalance) { balance = startBalance; }  
    public synchronized int getBalance() {  
        return balance;  
    }  
    public synchronized void deposit(int amount) {  
        balance += amount;  
    }  
    public synchronized void withdrawal(int amount) {  
        balance -= amount;  
    }  
}
```



# Transaction Worker

```
public class TxWorker implements Runnable {  
    protected BankAccount account;  
    protected char txType; // 'w' -> withdrawal, 'd' -> deposit  
    protected int amt;  
    public TxWorker(BankAccount account, char txType, int amt) { . . . }  
    public void run() {  
        if (txType == 'w')  
            account.withdrawal(amt);  
        else if (txType == 'd')  
            account.deposit(amt);  
    }  
}
```





# Dispatching Transactions

```
ExecutorService es = Executors.newFixedThreadPool(5);  
TxWorker[] workers = // Retrieve TxWorker instances  
  
for(TxWorker worker:workers)  
    es.submit(worker);  
  
// Shutdown es and wait
```



# Transaction Worker

```
public class TxWorker implements Runnable {  
    protected BankAccount account;  
    protected char txType; // 'w' -> withdrawal, 'd' -> deposit  
    protected int amt;  
    public TxWorker(BankAccount account, char txType, int amt) { . . . }  
    public void run() {  
        if (txType == 'w')  
            account.withdrawal(amt);  
        else if (txType == 'd')  
            account.deposit(amt);  
    }  
}
```



# Transaction Promo Worker

```
public class TxPromoWorker extends TxWorker {  
    public TxPromoWorker(BankAccount account, char txType, int amt) { super(. . .) }  
  
    public void run() {  
        if (txType == 'w')  
            account.withdrawal(amt);  
  
        else if (txType == 'd') {  
            account.deposit(amt);  
            if(account.getBalance() > 500) {  
                int bonus = (int)((account.getBalance() - 500) * 0.1);  
                account.deposit(bonus);  
            }  
        }  
    }  
}
```

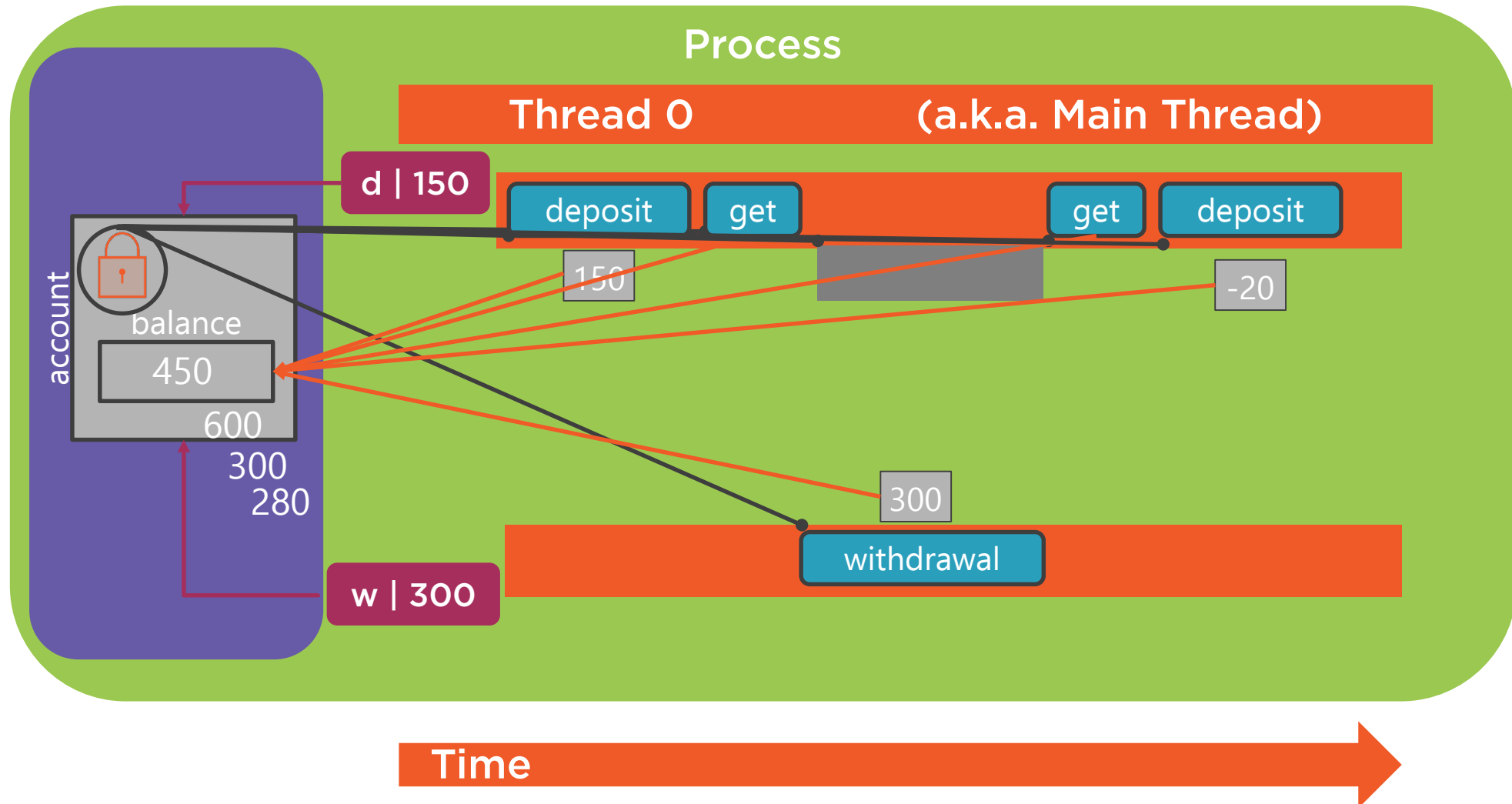


# Dispatching Transactions

```
ExecutorService es = Executors.newFixedThreadPool(5);  
TxWorker[] workers =          // Retrieve TxPromoWorker instances  
  
for(TxWorker worker:workers)  
    es.submit(worker);  
  
// Shutdown es and wait
```



# Behavior of Synchronized Methods

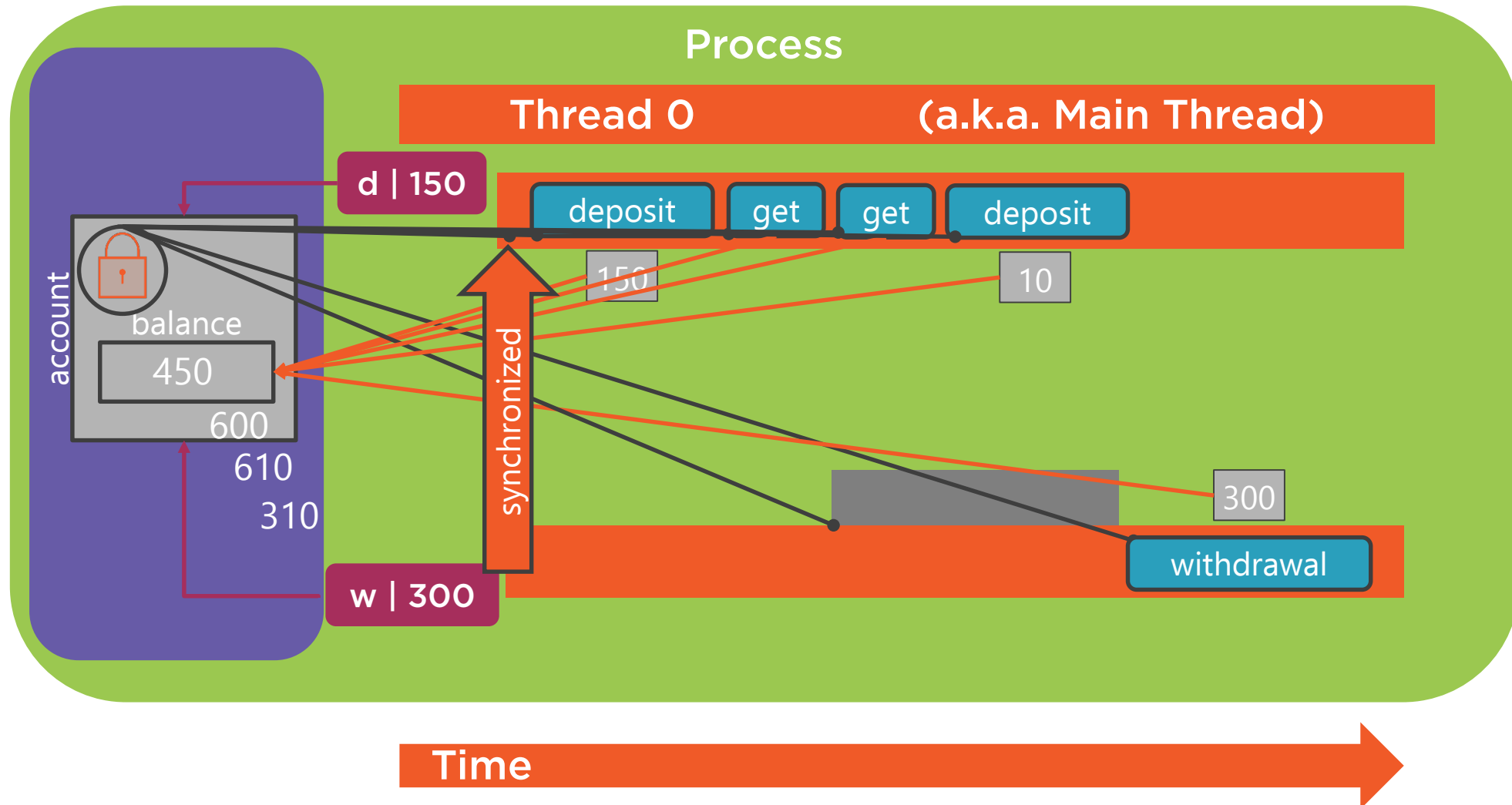


# Thread Safe Transaction Promo Worker

```
public void run() {  
    if (txType == 'w')  
        account.withdrawal(amt);  
    else if (txType == 'd') {  
        synchronized(account) {  
            account.deposit(amt);  
            if(account.getBalance() > 500) {  
                int bonus = (int)((account.getBalance() - 500) * 0.1);  
                account.deposit(bonus);  
            }  
        }  
    }  
}
```



# Behavior of Synchronized Methods



# Collections and Concurrency

## Concurrency and collections

- Concurrency safe collection access
- Blocking collections





# Concurrency Safe Collection Access

## Synchronized collection wrappers

- Most collections are not thread safe
- Can create thread safe wrapper
  - Use Collection class static methods
    - `synchronizedList`
    - `synchronizedMap`
    - Etc.
  - Wrapper is a thread safe proxy
    - Actual work occurs in original object



# Blocking Collections

## Coordinating producers and consumers

- One or more threads produce content
- One or more other threads consume
  - Must wait for content if not available

## Java provides blocking queues

- Attempt to read blocks if empty
  - Wakes up when content available
- Examples
  - `LinkedBlockingQueue`
  - `PriorityBlockingQueue`
  - Etc.



# Java Provides Still More

## **java.util.concurrent**

- Types for managing concurrency
  - Has much of what we've talked about
- Semaphores
  - Coordinate access to multiple resources
- Lots more

## **java.util.concurrent.atomic**

- Types providing atomic operations
  - set, get, getAndAdd, compareAndSet



# Summary



## Thread class

- Represents a thread of execution
- Similar to most OS thread representations
- Responsible to handle most details

## Runnable interface

- Represents a task to run on a thread
- Simply override run method
- Can't return results
- Exceptions responsibility of thread



# Summary



## ExecutorService

- Abstracts thread management details
- Can interact with thread pools

## Callable interface

- Represents a task to be run on a thread
- Can return results
- Can throw exceptions

## Future interface

- Represents results of a thread task
- Can access results from task
- Can throw exceptions from thread task



# Summary



## All Java objects have a lock

- Can access with synchronized methods
  - Acquires lock of target instance of call
  - Only one active at a time on an object
- Can manually acquire lock
  - Use synchronized statement block
  - Available to any code referencing object

