

G520SC

OPERATING SYSTEMS AND

CONCURRENCY

Concurrency in Java

Dr Jason Atkin

This Lecture

- Java
 - No longer having to worry about platform dependent behaviour
 - See some of how concurrency facilities are embedded in a (semi) object oriented language
- Understanding what Java does should be easy now that we understand the basics of concurrency in general
- By seeing things again in another context, we should better understand the principles

A simple Java Application (1)

```
public class SimpleTest1
```

```
{
```

```
    public static void main( String[] args )
```

```
    {
```

```
        SimpleTest1 mainObject = new SimpleTest1();
```

```
        mainObject.go();
```

```
    }
```

We create an object rather than just using static functions ☺

```
public void go()
```

```
{
```

```
    procedure1();
```

```
    procedure2();
```

```
    System.out.println( "Variable = " + iVariable + "\n" );
```

```
}
```

```
    ... other things go here too ...
```

```
}
```

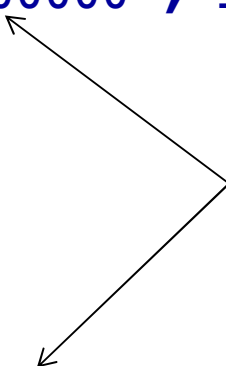
Non-static member, to do things (meaningfully named ☺)

A simple Java Application (2)

```
int iVariable = 0;
```

```
void procedure1()  
{  
    for ( int i = 0 ; i < 1000000 ; i++ )  
        ++iVariable;  
}
```

```
void procedure2()  
{  
    for ( int i = 0 ; i < 1000000 ; i++ )  
        ++iVariable;  
}
```



Each function increments
variable 1,000,000 times

Functions executed one after another and we get the result 2,000,000 as expected

Processes and Threads

Much like native applications (e.g. in C/windows API) :

- Java has both processes and threads
- Process: a self-contained environment with its own resources and one or more threads
- Thread: a single thread of operation within a process
- Java processes run within the JVM
- Within the JVM, the *threads* comprising a Java program are represented by instances of the **Thread** class

Convert to threaded implementation

- Java wraps the thread starting and thread management operations in a single **Thread** class
- You **either** create objects of a sub-class of this class and implement the **run()** method:

```
class MyThread extends Thread
{
    @Override
    public void run()
    {
        procedure1(); // Do whatever work you need
    }
}
```

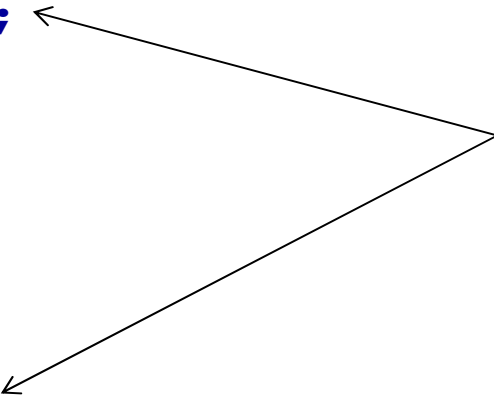
- **Or** you create objects which implement the runnable interface (i.e. the **run()** method) and pass the object reference to the constructor of a new **Thread** object

A Threaded Java Application

```
class MyThread extends Thread
{
    @Override
    public void run()
    {
        procedure1();
    }
}
```

```
public void go()
{
    // procedure1(); // Removed
    new MyThread().start(); // NOT RUN!
}
```

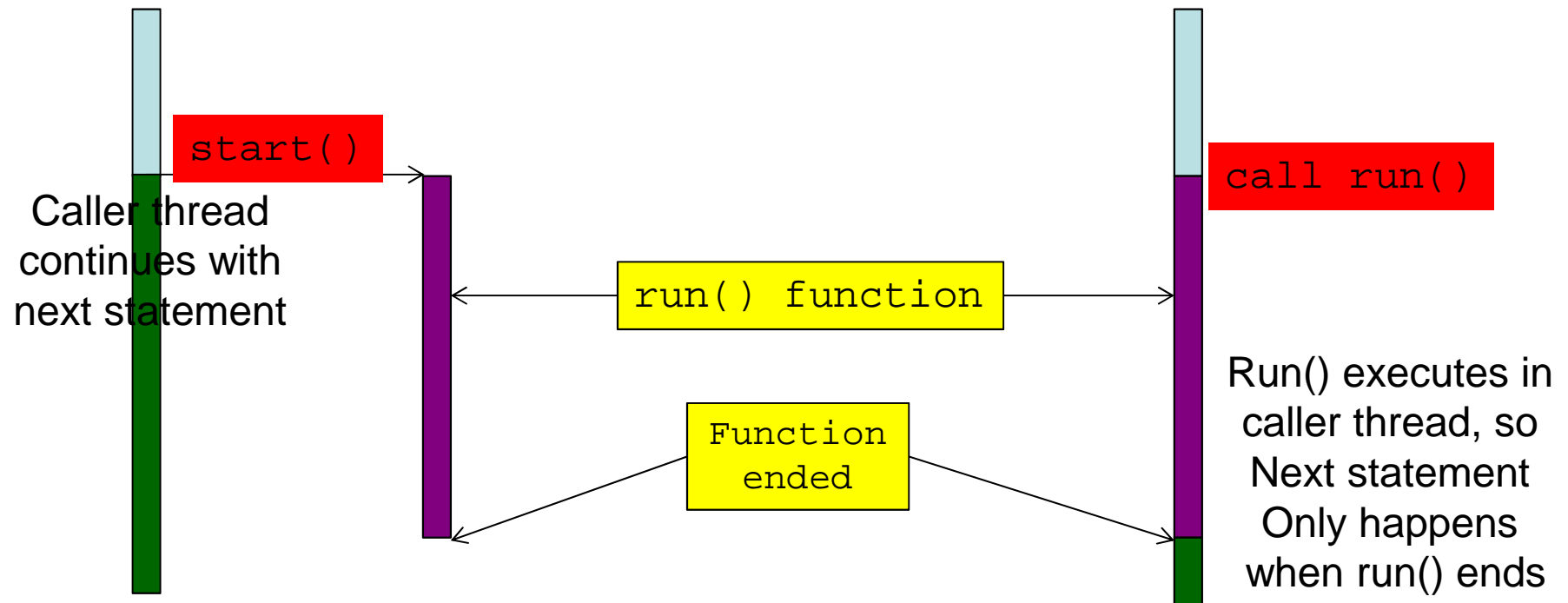
procedure1() is now
called by the new thread



Create a new Thread (sub-class) object and call start()

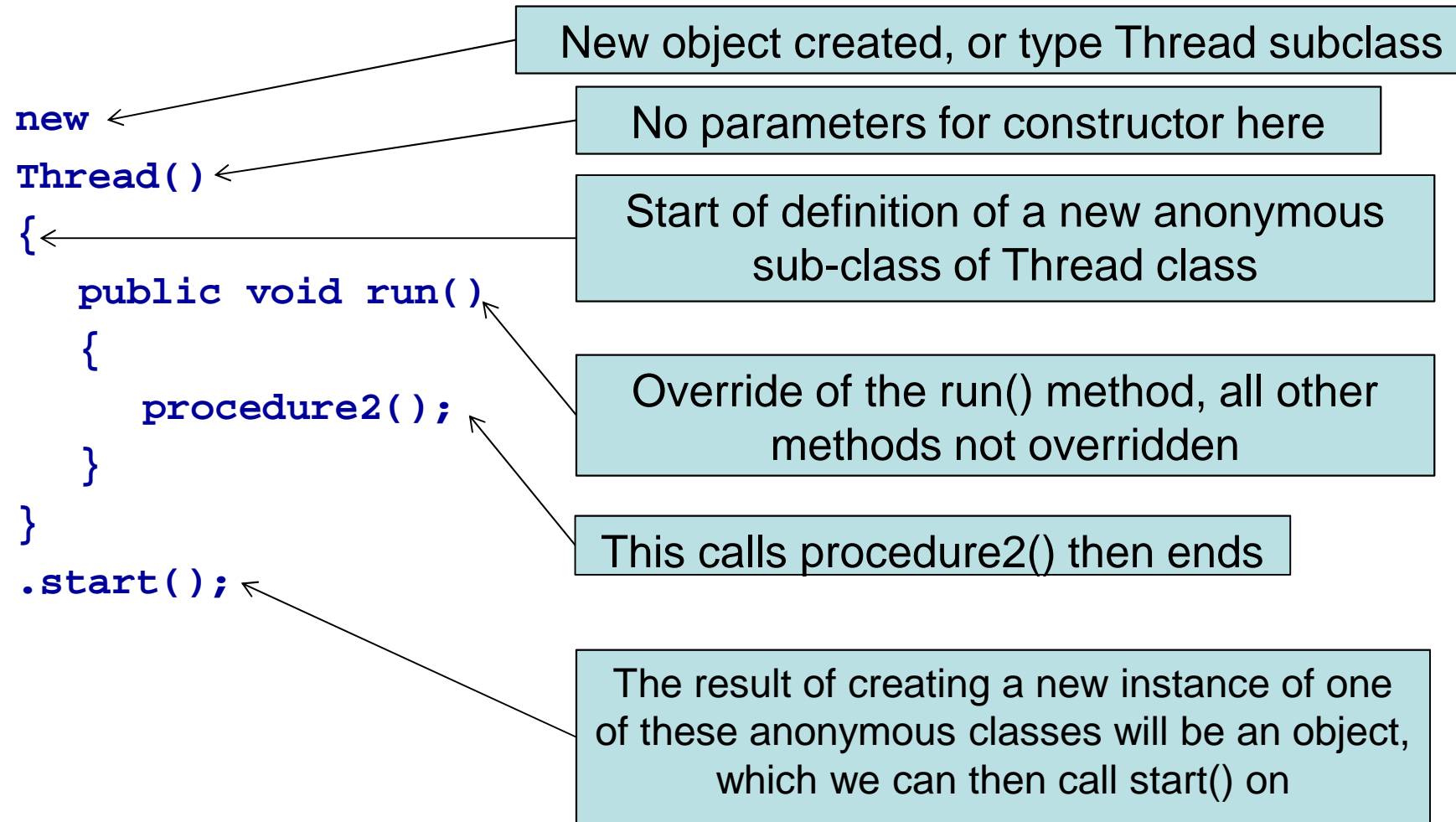
Use start() not run() to start thread

- Call **start()** to tell the JVM to start a new thread and call your **run()** function from it
- If you call **run()** directly you will not start a new thread!



We could also use an anonymous class

```
new Thread() { public void run() { procedure2(); } }.start();
```



Implementing Runnable

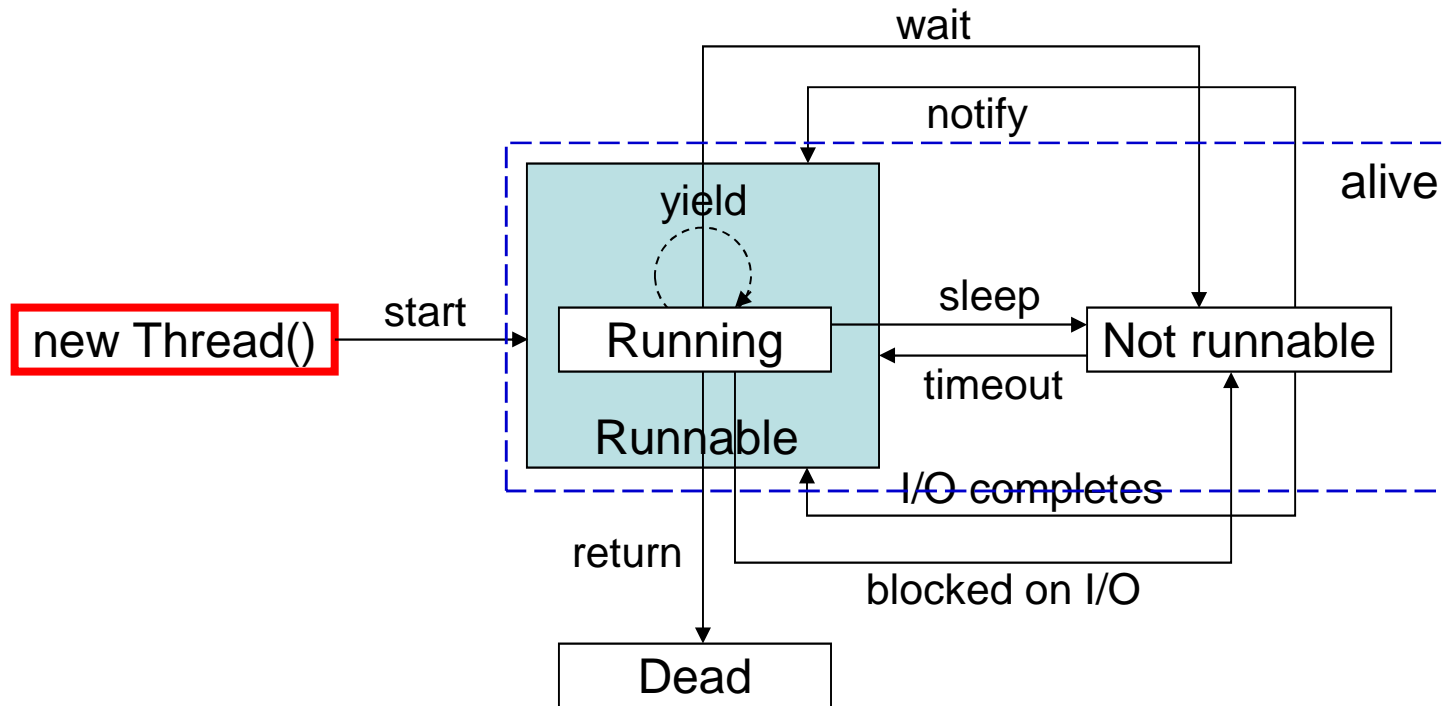
```
class MyClass
    extends SomeBaseClass
    implements Runnable
{
    // constructor etc ...

    public void run()
    {
        // Do something
    }

    // other methods ...
}
```

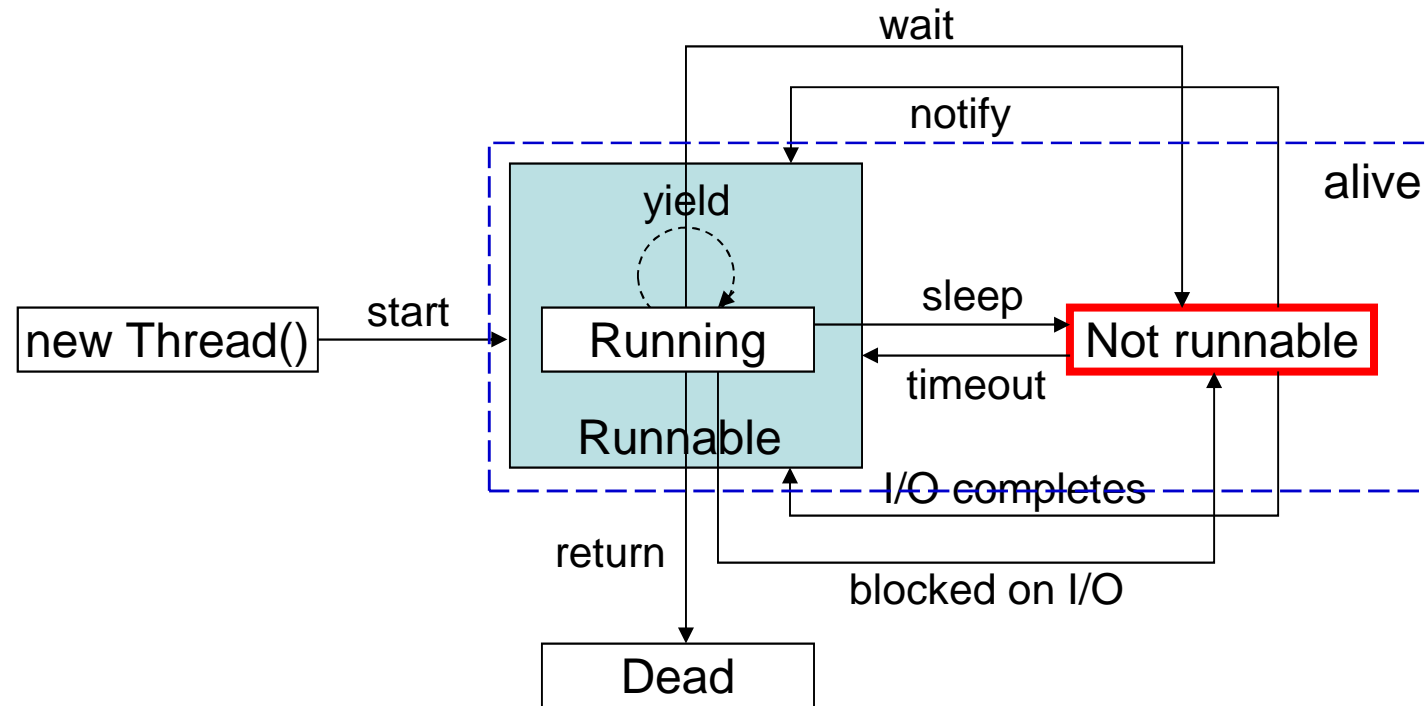
```
new Thread( this ).start(); // NOT run()
```

Thread lifecycle: creation



- A running **Thread** will execute whatever is in its **run()** method, until the **run()** method ends
 - End of function, a return statement, or an uncaught exception
- Sometimes it may enter a 'waiting' state (not runnable)

Thread lifecycle: not runnable



- A running **Thread** becomes **not runnable** when:
 - It calls `sleep()` to delay for a duration
 - It blocks for I/O
 - It blocks in `wait()` for condition synchronisation

Thread termination

- A thread terminates when its `run()` method completes:
 - either by returning normally; or
 - by throwing an unchecked exception (`RuntimeException`, `Error` or one of their subclasses)
- **Threads are not restartable**
 - Invoking `start()` more than once results in an `InvalidThreadStateException`
 - You could run the same code using a **new** Thread object
- There are several ways to get a thread to stop:
 - when the thread's `run()` method returns;
 - call `Thread.stop()`
 - Usually a **bad** idea: doesn't allow thread to clean up before it dies
 - `interrupt()` the thread (next slide)
- The process will continue until last thread terminates

Interrupting a Thread

- Each **Thread** object has an associated **boolean** interruption status:
 - **interrupt()**: sets a running thread's interrupted status to *true*
 - **Thread.interrupted()**: returns *true* if the current thread has been interrupted by **interrupt()** (and clears the flag)

- A thread can periodically check (and clear) its interrupted status, and if it is *true*, clean up and exit – but it is up to the thread to do so

```
if ( Thread.interrupted() )  
    System.out.println( "Interrupted thread" );
```

- Threads which are blocked in calls **wait()** and **sleep()** are not checking the value of the interrupted flag, so Interrupting a thread which is waiting/sleeping throws an **InterruptedException**

```
try {  
    wait() or sleep()  
} catch (InterruptedException e) {  
    // clean up and return (interrupted status not set)  
}
```

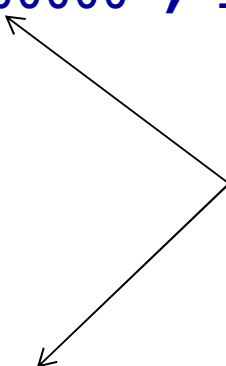
Back to the increment a
number sample

Single threaded code....

```
int iVariable = 0;
```

```
void procedure1()  
{  
    for ( int i = 0 ; i < 1000000 ; i++ )  
        ++iVariable;  
}
```

```
void procedure2()  
{  
    for ( int i = 0 ; i < 1000000 ; i++ )  
        ++iVariable;  
}
```



Each function increments
variable 1,000,000 times

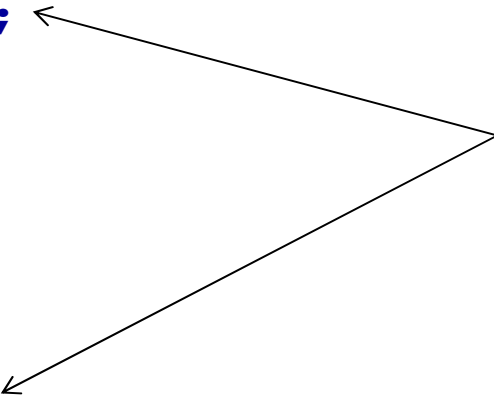
Functions executed one after another and we get the result 2,000,000 as expected

A Threaded Java Application

```
class MyThread extends Thread
{
    @Override
    public void run()
    {
        procedure1();
    }
}
```

```
public void go()
{
    // procedure1(); // Removed
    new MyThread().start(); // NOT RUN!
}
```

procedure1() is now
called by the new thread

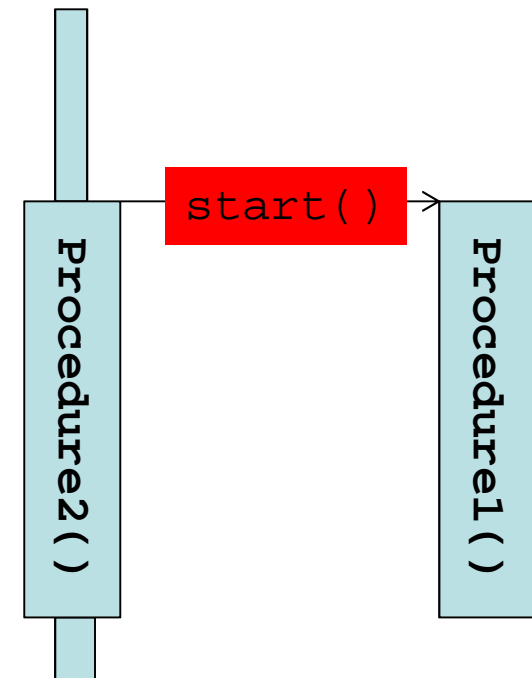


Create a new Thread (sub-class) object and call start()

Multi-threaded counting program

```
SimpleThreadTest2 mainObject = new SimpleThreadTest2();  
mainObject.go();
```

```
class MyThread extends Thread  
{  
    public void run()  
    { procedure1(); }  
}  
  
public void go()  
{  
    //procedure1();  
    new MyThread().start(); // NOT RUN!  
  
    procedure2();  
    System.out.println( "Value is " + iVariable + "\n" );  
}
```



Issues: wait for it to finish (1)

- Main thread needs to wait for the other thread to finish before it prints the result

```
Thread t = new MyThread(); // Subclass IS A base class obj  
t.start(); // Start is virtual fn so will call MyThread's
```

```
procedure2();
```

```
// Wait for thread to finish  
while ( t.isAlive() )  
{  
    try { Thread.sleep(1); } // Give up some CPU time  
    catch (InterruptedException e)  
    { e.printStackTrace(); }  
}
```

Issues: wait for it to finish (2)

- Main thread needs to wait for the other thread to finish before it

```
Thread t = new MyThread(); // Subclass IS A base class obj  
t.start(); // Start is virtual fn so will call MyThread's
```

```
procedure2();
```

```
// Wait for thread to finish
```

```
try
```

```
{
```

```
    t.join();
```

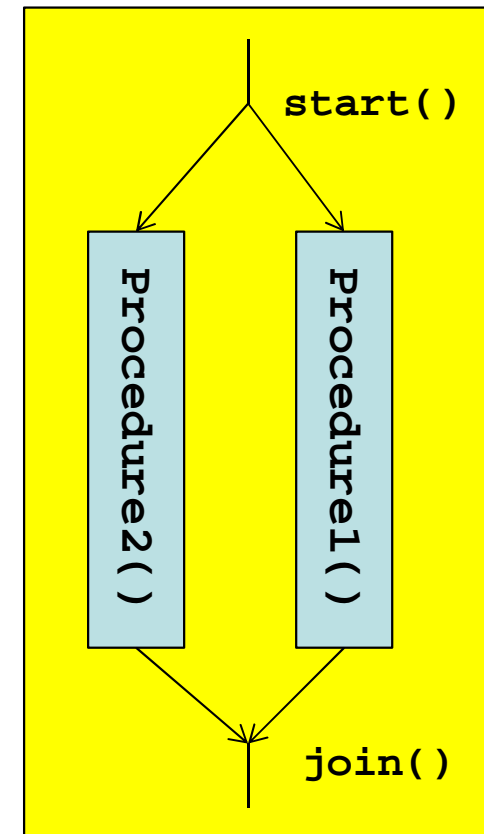
```
}
```

```
catch (InterruptedException e)
```

```
{
```

```
    e.printStackTrace();
```

```
}
```



The result is still not 2,000,000

- When you run this you will probably not get 2,000,000

- Same reasons as in C/Windows API

++X is not an atomic operation

Atomic operations can be interleaved in any order

- You could use atomic variables (objects), e.g.:

```
AtomicInteger c = new AtomicInteger(0);  
c.incrementAndGet(); c.decrementAndGet();
```

- See Atomic Variables, Oracle's tutorial:

<http://docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html>

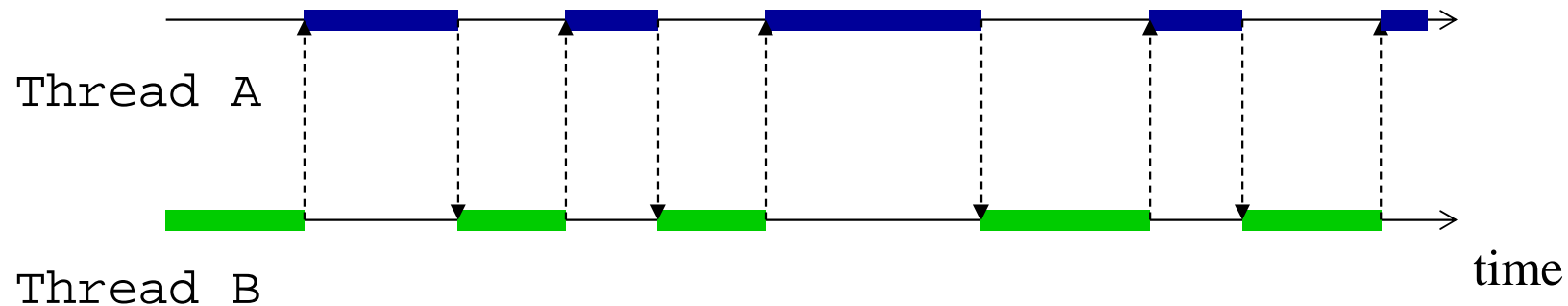
- Various types of atomic variables:

```
AtomicBoolean, AtomicInteger, AtomicLong,  
AtomicIntegerArray, AtomicLongArray  
AtomicReference
```

- You do not need to make Atomic variables volatile
 - Making a normal variable volatile is not enough

Java execution

- Consider a Java program consisting of two threads:

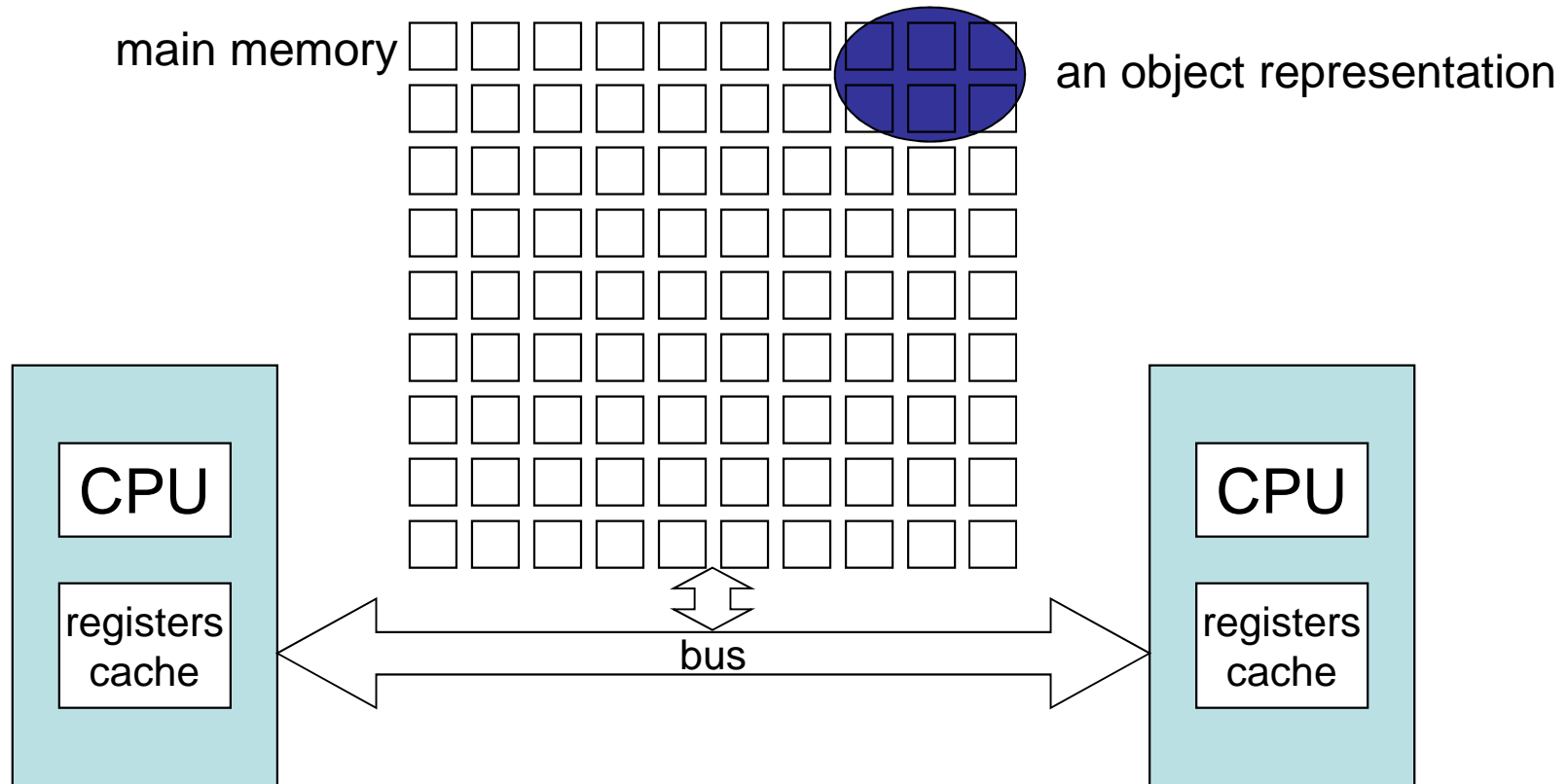


Given a single processor, the JVM executes a sequence of instructions which is an *interleaving* of the instruction sequences for each thread.

Java code optimisation

- Not only may concurrent executions be interleaved, they may also be reordered and otherwise manipulated to increase execution speed:
 - the compiler may rearrange the order of the statements
 - the processor may rearrange the execution order of the machine instructions
 - the memory system may rearrange the order in which writes are committed to memory
 - the compiler, processor and/or memory system may maintain variable values in, e.g., CPU registers, rather than writing them to memory so long as the code has (appears to have) the “intended” effect

Java Memory Model



Working memory

- Java allows threads that access shared variables to keep private ‘working copies’ of variables:
 - Each thread is defined to have a *working memory* (an abstraction of caches and registers) in which to store values
 - This allows a more efficient implementation of multiple threads
- The Java Memory Model specifies when values must be transferred between main memory and per-thread *working memory*

Model properties

- The Java Memory Model:
 - **atomicity**: which instructions must have indivisible effects
 - **visibility**: under what conditions are the effects of one thread visible to another
 - **ordering**: under what conditions the effects of operations can appear out of order to any given thread
- Original model was (perceived to be) broken
 - Updated for Java 5
 - Many new concurrency facilities added in Java 7

Example: Peterson's algorithm

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2);
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// shared variables
bool c1 = c2 = false;
integer turn = 1;
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1);
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

To work the algorithm depends upon variables being set in this order and no reordering occurring

Assumptions: Peterson's algorithm

- Peterson's algorithm implicitly relies on:
 - **atomicity**: variable reads and writes being atomic
 - **visibility**: the values written to the variables being immediately propagated to the other thread
 - **ordering**: the ordering of the instructions being preserved
 - that the **scheduling policy** is at least *weakly fair* (if you wait long enough then you get a go), otherwise eventual entry is not guaranteed
 - We will come back to this one tomorrow

Atomicity

- Reads and writes to memory cells corresponding to fields of any type **except** `long` or `double` are guaranteed to be atomic:
 - when a field (other than `long` or `double`) is used in an expression, you will get **either** its **initial** value or **some value that was written** by **some** thread
 - however you are **nOt** guaranteed to get the value most recently written by **any** thread

Visibility

- Without synchronization, changes to fields made by one thread are not guaranteed to be visible to other threads:
 - The first time a thread accesses a field of an object, it sees either
 - the initial value of the field, **or**
 - a value since written by some other thread
 - When a thread terminates, all written variables are flushed to main memory

Ordering

- The **apparent** order in which the instructions in a method are executed can differ:
 - From the point of view of the thread executing the method, instructions *appear* to be executed in the proper order (*as-if-serial* semantics)
 - From the point of view of other threads executing unsynchronised methods almost anything can happen

volatile fields

- If a field is declared **volatile**, a thread must reconcile its working copy of the field with the master copy **every** time it accesses the variable
 - reads and writes to a **volatile** field are guaranteed to be atomic (even for **longs** and **doubles**)
 - new values are immediately propagated to other threads
 - from the point of view of other threads (as well), the relative ordering of operations on **volatile** fields are preserved
- However the ordering and visibility effects surround only the single read or write to the **volatile** field itself
 - e.g, '++' on a **volatile** field is not atomic

Spin locks with `volatile`

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2);
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// shared variables
bool c1 = c2 = false;
integer turn = 1;
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1);
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

Try the sample and you should find that it worked, without having to add any explicit memory barriers

Memory Barriers

- If you tried Peterson's algorithm using volatile variables in C, on a modern operating system, you probably found that it did not work
 - I added a sample which fixed the problem using memory barriers (before every variable access)
 - Memory barriers are code **inserted into the program** to tell the **processor** not to optimise across the barrier
 - Read (load), Write (store) or both (Full)
 - Also called Memory Fences
 - Java inserts memory barriers automatically for volatile variables (**most C compilers do not**)
 - Most mutex/semaphore lock implementations have to add these (Posix does and Visual Studio does for the API calls)

Next Lecture

- Fairness in Java
- Monitors