Threads: Concurrency

Dr Gokop Goteng

READ http://tutorials.jenkov.com/java-concurrency

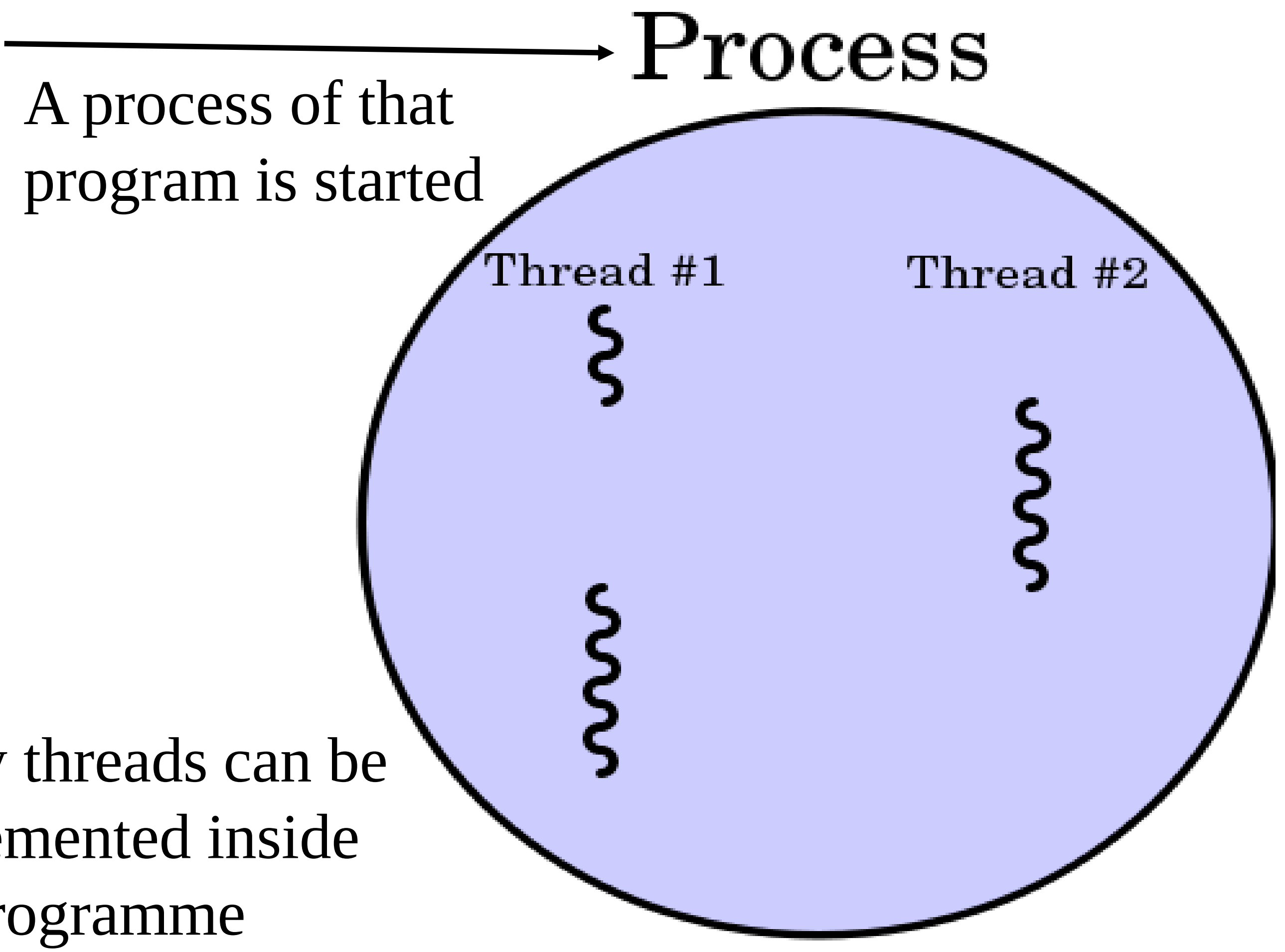
Computer Process and Thread

- Process: This is an instance of a computer programme that is executed and is running
  - If you open a computer program such as Java Eclipse or MS Word and you are working with them, the programmes (in this case Java Eclipse and MS Word) are running as a process
- Thread: This is the smallest part of a programme instruction that runs and is managed independently by a scheduler of an operating system or a Java Virtual Machine (JVM) in case of Java programmes
- A process can consist of one or more threads
- So threads are part of a process

Computer Process and Thread

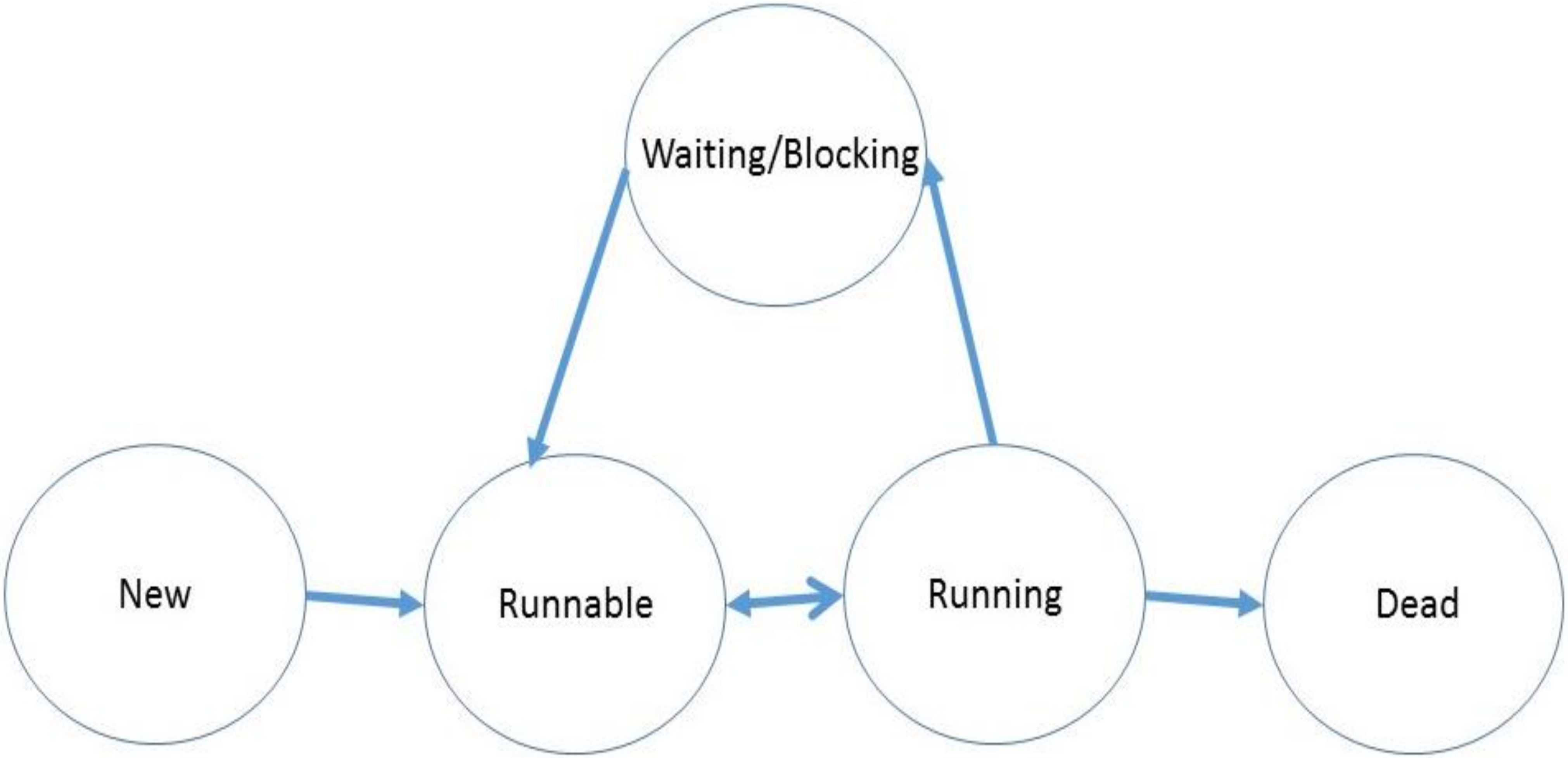
Start a computer programme

A process of that program is started



Many threads can be implemented inside the programme

Life Cycle of a Thread



Life Cycle of a Thread

- ***New:*** This is when the Thread instance is created using the "new Thread()" command. The thread is still not alive yet
- ***Runnable:*** This is when the Thread instance variable invokes the "start()" method using the "t.start()" command, where t is the thread instance variable. The thread is now alive and has joined the queue of threads about to be executed
- ***Running:*** This is when the Thread is now executing
- ***Blocked/Waiting/Sleeping:*** This is when the Thread goes out of the execution or running mode when the wait, interrupt or sleep commands are executed

Life Cycle of a Thread

- ***Dead:*** This is when the Thread completes its task terminates execution calling “t.destroy()” command for garbage collection.

Threads are similar to the operating system concept of a *process*

- **Except that processes run in separate address spaces**, while all the threads of one process run in the same address space.
- This means that two threads could make use of the same object concurrently.
  - » This can lead to unpredictable results, so it is necessary to ensure that threads cooperate with each other on shared objects.

JVM is a process

- A Java application runs by default in one **process**.
  - It executes your byte code.
- Within a Java application you work with several **threads** to achieve pseudo parallel processing or asynchronous behaviour.

Processes and Threads

- A *process* runs independently and isolated from other processes. It cannot directly access shared data in other processes. **The resources of the process, e.g. memory and CPU time, are allocated to it via the operating system.**
- A *thread* is a so called lightweight process. Each has its **own call stack** but can access shared data of other threads in the same process. **Every thread has its own memory cache**. If a thread reads shared data it stores this data in its own memory cache. A thread can re-read the shared data

There are three very important concepts when doing concurrent programming

- Atomicity :
  - An operation is said atomic when it cannot be interrupted.
  - There are almost no atomic operations in Java
  - one is the assignment a = 5, (note: integer)
    - but a = b++ is not atomic.
  - In some cases, you'll have to make some actions atomic with synchronization
- Visibility : This occurs when a thread must watch the actions of another thread
  - E.g. for the termination of the thread; for a value being set.
  - This also implies some kind of synchronization.

- Order of execution :
  - When you have normal program, all you lines of code run in the same order every time you launch the application.
  - This is not the case with concurrent programming
  - **The order of execution is not guaranteed !**

The Thread class

- The Thread class is responsible for executing *your stuff* in a thread.
- *your stuff* is encapsulated in a run method.
- And the Thread class manages the running of the thread
- Two ways of giving the Thread the run method
  - » 1 **Passing a runnable object to the Thread constructor**
  - » 2 **Creating an instance of a new class that inherits the Thread class, explicitly specifying the run method in the class**

Creating a thread METHOD 1

- Declare a Runnable

```
public class MyFirstRunnable implements Runnable
{public void run() {
          System.out.println("In a thread");   }
```

- Create the new Thread instance and start execution

```
Thread thread = new Thread(new MyFirstRunnable());
thread.start();
```

Method 1 : What is a Runnable?

- Runnable is an Interface that requires you to implement a run() method
- A Runnable is used to create a type of class that can be put into a thread, describing what the thread is supposed to do.
- If you did not have the Runnable interface, the Thread class, which is responsible for executing your stuff in the other thread, would not have the promise to find a run() method in your class,
    - so you could get errors.

Predict the outcome?

```
Thread thread = new Thread(new
MyFirstRunnable());
thread.start();
System.out.println("In the main Thread");
```

In a thread
In the main Thread
or
In the main Thread
In a thread

```
public void run() {
System.out.println("In a thread");
}
```

You can use the runnable several times

```
Runnable runnable = new
MyFirstRunnable();

for(int i = 0; i <25; i++){
 new Thread(runnable).start();
}
```

How many threads will be running?

You can also give names to a Thread using the setName() method

```
public class MySecondRunnable implements Runnable{
public void run() {
System.out.printf("I'm running in thread %s \n",
Thread.currentThread().getName());
}}

…………………………………………………………..

Runnable runnable = new MySecondRunnable();
for(int i = 0; i < 25; i++){
  Thread thread = new Thread(runnable);
  thread.setName("Thread "+ i);
  thread.start();
}
Order different each time
```

We could be more cryptic: and use an Anonymous Type

- You do not need to define a class
  - » you can do all of that inline:

```
Thread t = new Thread(new
Runnable()
{ public void run() { // stuff here
} });

t.start();
```

What is going on here?

Let’s see

Example of anonymous inner class implementing an interface

```
interface ProgrammerInterview { public void read(); }

class Website {
ProgrammerInterview p = new ProgrammerInterview () {
public void read() { System.out.println("interface
ProgrammerInterview anonymous class implementer"); }
};
}
```

NB We are NOT creating an instance of an interface

We are simultaneously **creating an anonymous class that implements the ProgrammerInterview interface** and also **creating an instance of that anonymous class**

What is happening?

- Inside the Website class, the code in red is actually creating an instance of an **anonymous inner class** that implements the ProgrammerInterview interface.
- The class that is being instantiated (where “p” is the instance variable) is anonymous **because it has no name**
  - » **Note cannot write constructor of such a class as it has no name**
- We are simultaneously creating an anonymous class that implements the ProgrammerInterview interface and also creating an instance of that anonymous class.

Another way of creating a thread METHOD 2

```
public class MyThread extends
Thread {
public void run(){
    System.out.println("MyThread
running");
     }
   }
myThread=new MyThread();
myThread.start();
```

Is one better than the other?

- Because it is only possible to extend one class, using

```
public class MyThread extends Thread
```

  - » no other classes can be inherited from `MyThread`
- Using the Runnable interface allows a subclass of Thread to be used if required.
- Note: The **default run method** of the class Thread is

```
public void run(){if(target!=null)
                  {target.run();}
```

Making a thread sleep

- Can make a thread sleep for a certain number of milliseconds.
- The Thread class has a method sleep(long millis).
- This method is static, hence you can ONLY make the current Thread sleep.
  - » Thread.sleep(1000);
- You cannot choose the thread you want to sleep
  - » your only choice is the current Thread

Making a thread sleep

- if you want to sleep 1000 milliseconds and 1000 nanoseconds (1 microsecond)

```
Thread.sleep(1000, 1000);
```

What is an interrupt?

- An *interrupt* is an **indication** to a thread that it should stop what it is doing and do something else.
  - » It does not stop the thread
- The programmer decides how a thread responds to an interrupt, but it is common for the thread to terminate.
- A thread sends an interrupt (to another thread t) by invoking interrupt on the Thread object to be interrupted.
  - » t.interrupt()
- **The interrupted thread must support its own interruption**.

Thread.interrupt()

- if the **thread is sleeping or joining or waiting on another Thread**,
  - an InterruptedException is thrown.
  - and then the interrupted status of the Thread will be cleared (immediately)
- You can now do things to tidy up and finish the thread

More completely also thrown for :

- Process.waitFor(), which lets us wait for an external process (started from our Java application) to terminate;
- various methods in the Java 5 concurrency libraries, such as the tryLock() method of the Java 5 ReentrantLock class.

Example: A sleeping thread can be interrupted, i.e. the interrupt flag is set

- t.interrupt() sets the interrupted flag of t
- E.g. if t is sleeping inside the try block below we can catch the thrown exception:

```
try{
 Thread.sleep(1000);
} catch(InterruptedException e){
 e.printStackTrace();
}
```

In the run() method of t

- They respond to interruption by clearing the interrupted status and throwing InterruptedException.
- *Interrupting a thread that is not alive need not have any effect.*

NB the above code is **not a good way** to manage the InterruptedException for sleep
…. Later

join: Waiting for another thread to die

- E.g., create five threads to compute intermediate results and wait for these 5 threads to finish to compute the final results
- To do this, you can use the join() method of the Thread class.
- This method is not static, so you can use it on any thread to wait for it to die. (or sleep)
- Like sleep() this method throws InterruptedException when the thread is interrupted during waiting for an other thread.

Example

- So to wait on thread2, you just have to:

```
try{
    thread2.join();
    } catch(InterruptedException
e){
     e.printStackTrace();
    }
```

Wait for thread2 to die

Will make the current Thread wait for thread2 to die.

Reasons a thread may be blocked

- A thread that is prevented from executing is said to be blocked.

A thread may be **blocked** because

1. It has been put to sleep for a set amount of time
2. It is suspended with a call to **suspend**() and will be blocked until a **resume**() message
   - » Stop , suspend, resume deprecated
3. The thread is suspended by call to **wait()**, and will become runnable on a ***notify*** or ***notifyAll*** message.

- *sleep*(), *join()* and *wait/notify* are probably the most important of the situations where a thread can be blocked.

Recap

http://www.ibm.com/developerworks/library/j-jtp05236/

- Every thread has a Boolean property associated with it that represents its *interrupted status*.
  - The interrupted status (aka flag) is initially false;
- when a thread is interrupted by some other thread through a call to Thread.interrupt(), one of two things happens:

What happens when interrupted

1. **If that thread is executing a low-level interruptible blocking method**
   like Thread.sleep(), Thread.join(),
   or Object.wait(),
   - it unblocks,resets and throws InterruptedException.
2. **Otherwise**, interrupt() merely sets the thread interruption status.

Dealing with the interrupt

- Code running in the interrupted thread can later poll the interrupted status to see **if** it has been requested to stop what it is doing;
- the interrupted status can
  - » be **read** with Thread.isInterrupted() or
  - » can be **read and cleared** in a single operation with Thread.interrupted()

Interrupt Mechanism - recap

- The interrupt mechanism is implemented using an internal flag known as the interrupt status.
- Invoking interrupt on a target thread sets this flag.
- A thread may check for an interrupt (of itself) by invoking the **static** method **Thread.interrupted**, and in this case **the interrupt status is cleared**.
- one thread can query the interrupt status of another by using the **non-static isInterrupted**. This does not change the interrupt status flag.
  - » Only looking!

And again!!

- interrupted() is **static** and checks the current thread. isInterrupted() is an **instance method** which checks the Thread object that it is called on.
- **A common error** is to call a static method on an instance.

```
Thread myThread = ...;
if (myThread.interrupted()) {}
// WRONG! This might not be checking myThread.
// Need Thread.interrupted()
if (myThread.isInterrupted()) {} //Right!
```

Another difference

- is that interrupted() also clears the status of the current thread.

In other words, if you call it twice in a row and the thread is not interrupted between the two calls, the **second call will return false even if the first call returned true**

Summary: Thread.interrupt Java API 1 (For reference only)

- First the **checkAccess** method of this thread is invoked, which may cause a SecurityException to be thrown.
- If **this thread is blocked** in an invocation of the wait(), wait(long), or wait(long, int) methods, or of the join(), join(long), join(long, int), sleep(long), or sleep(long, int), methods of this class, **then the interrupted status will be set BUT then interrupt status will be cleared and it will throw an InterruptedException**.

NB continued

Interrupt – example 1

```
class A extends Thread{
public void run(){
for (int i=1; i<=5;i++)
            System.out.println (i);}

}
```

Interrupt – example 1

```
public static void main(String args[]){
A t1=new A();
t1.start();
t1.interrupt();
}
}
```

Output?

Note that t1 has no mechanism to handle an interrupt!

Example 2
Interrupting a thread that stops working

```
class A extends Thread{
public void run(){
try{
Thread.sleep(1000);
System.out.println("task");
}catch(InterruptedException e){
throw new RuntimeException("Thread 
interrupted..."+e);
}
}
```

After interrupting we propagate InterruptedException, throw RuntimeException
and thread A will stop and program as well

RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine

continued

```
public static void main(String args[]){
A t1=new A();
t1.start();
try{
t1.interrupt();
}catch(Exception
e){System.out.println("Exception handled
"+e);}
}
}
```

Handling the exception

```
class A extends Thread{
public void run(){
try{
Thread.sleep(1000);
System.out.println("task");
}catch(InterruptedException e){
System.out.println("Exception handled "+e);
}
System.out.println("thread is running...");
}
```

Handling continued

```
public static void main(String
args[]){
A t1=new A();
t1.start();

t1.interrupt();

}
}
```

Answer: Handling the exception

```
class A extends Thread{
public void run(){
try{
Thread.sleep(1000);
System.out.println("task");
}catch(InterruptedException e){
System.out.println("Exception handled "+e);
}
System.out.println("thread is running...");
}
```

```
Exception handled java.lang.InterruptedException: sleep interrupted
thread is running...
```

RECAP: Interrupting a Thread: Thread.interrupt()

- sets the interrupted status flag of the target thread.
  - Does not “stop” it
- code MAY poll the interrupted status and handle it appropriately.
- If the target thread **does not poll** the interrupted status the interrupt is effectively ignored.
- Polling occurs via the Thread.interrupted() and t.isInterrupted() methods .

There are 3 ways to terminate a thread

- The thread has finished the work it is designed to do, and exits the run() method naturally.
- The thread has received an interruption signal, while doing its work.
  - » It decides to not continue with work and exits the run() method.
  - » It may also decide to ignore the signal and continue to work.
- The thread has been marked as **daemon thread**. When the parent thread who created this thread terminates, this thread will be forcefully terminated by the JVM.

You have no command to force a Thread to stop (in Java)

- if the Thread is not well-written, it can continue its execution infinitely.
- Typically
  - » use a shared boolean and when the boolean becomes true to reach the **end of run** (and hence the thread stops)
  - » **Or** can also interrupt a thread with the interrupt() method.
    - This can give a quicker exit from the thread , e.g. use return
    - A thread rarely interrupts itself , but see later

Example: Interrupt while displaying important information

- it simply returns from the run method after it catches that exception.

```
for (int i = 0; i < importantInfo.length; i++) {
                    // Pause for 4 seconds
          try { Thread.sleep(4000); } catch
               (InterruptedException e) {
           // We've been interrupted: no more messages.
                         return; }
//Print a message System.out.println(importantInfo[i]); }
```

REPEAT: Thread.interrupted() and Thread.isInterruptedflag.

- isInterrupted() is an instance method which checks the Thread object that it is called on.
  - Is thread you call it on currently interrupted?
  - Looks at the thread interrupted status
- interrupted() is static and checks the current thread.
  - "Have I been interrupted since the last time I asked?"
  - **interrupted()** clears the status of the current thread.
  - In other words, if you call it twice in a row and the thread is not interrupted between the two calls, the second call will return false even if the first call returned true.

One should not ignore InterruptedException

```
try {
doSomething();
} catch( InterruptedException swallowed) {
// BAD BAD PRACTICE, TO IGNORE THIS EXCEPTION
// Also just logging is also not a useful
// option....
}
```

What you should sometimes do…

```
try {
doSomething();
} catch(InterruptedException e) {
// Restore the interrupted status
Thread.currentThread().interrupt();


}
```

Example: After one second, the second thread is interrupted

```
public class InterruptThread {
public static void main(String[] args) {
Thread thread1 = new Thread(new WaitRunnable());
thread1.start();
try { Thread.sleep(1000); }   // wait a second to make sure
                             // thread1  started
catch (InterruptedException e) { e.printStackTrace(); }

thread1.interrupt(); }   // now interrupt thread1
```

and the interrupted status has been set to false.

```
private static class WaitRunnable implements Runnable {
@Override
public void run() {
System.out.println("Current time millis : " + System.currentTimeMillis());
try { Thread.sleep(5000); }   // will be asleep when interrupt comes
catch (InterruptedException e) {
                    System.out.println("The thread has been interrupted");
                    System.out.println("The thread is interrupted : " +
                    Thread.interrupted(); }
System.out.println("Current time millis : “ + System.currentTimeMillis()); }
} }
```

and the interrupted status has been set to false.

```
private static class WaitRunnable implements Runnable {
@Override
public void run() {
System.out.println("Current time millis : " + System.currentTimeMillis());
try { Thread.sleep(5000); }   // will be asleep when interrupt comes
catch (InterruptedException e) {
                    System.out.println("The thread has been interrupted");
                    System.out.println("The thread is interrupted : " +
                    Thread.currentThread().isInterrupted()); }
System.out.println("Current time millis : " + System.currentTimeMillis()); }
} }
```

Recap

- Many methods that throw InterruptedException, such as sleep, are designed to cancel their current operation and return immediately when an interrupt is received.
- By convention, any method that exits by throwing an InterruptedException clears interrupt status when it does so.
  - » As sleep did
  - » However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt.
  - » Also generally interrupt does not stop the method
  - » It sets a flag
    - – The programmer has to stop it

@Override: Use it every time you override a method for two benefits.

- Do it so that you can take advantage of the compiler checking to make sure you actually are overriding a method when you think you are.
  - » This way, if you make a common mistake of misspelling a method name or not correctly matching the parameters, you will be warned that you method does not actually override as you think it does.
  - » Secondly, it makes your code easier to understand because it is more obvious when methods are overwritten.
- Additionally, in Java 1.6 you can use it to mark when a method implements an interface for the same benefits.
  - » Might be better to have a separate annotation @Implements

Examples of management of interrupt : If not sleeping you can just test for interrupt

```
public class InterruptableRunnable
implements Runnable {
@Override
public void run() {
while(!Thread.currentThread().isInt
errupted()){ //Heavy operation
} } }
```

Check before entering

Example: Simply catching an InterruptedException does not make your thread "interrupt safe"

```
public class UglyRunnable implements Runnable {
@Override
public void run() {
while(!Thread.currentThread().isInterrupted()){
     //Heavy operation
     try { Thread.sleep(5000); }
     catch(InterruptedException e) {
     e.printStackTrace(); }
     //Other operation } } }
```

If another thread interrupts your thread while your thread is sleeping. The sleep will be interrupted, but the interrupted status will be cleared (sleep does this) so the loop will continue.

Making thread safe: interrupt again as may need to do something rather than just exit

```
public class BetterRunnable implements Runnable {
@Override
public void run() {
while(!Thread.currentThread().isInterrupted()){
        //Heavy operation
        try { Thread.sleep(5000); }
        catch   (InterruptedException e) {
                Thread.currentThread().interrupt(); }
        //Other operations }
} }
```

Restoring the interrupted status

- Now the interrupted status is restored and the loop that tests for interrupt can be stopped after interrupt.
- Depending on your code, you can also add a continue statement after the interrupt() to prevent operations after interrupt.
  - continue: The unlabelled form skips to the end of the innermost loop's body **and evaluates the boolean expression that controls the loop.**
  - **Hence exit loop**
- In some cases, you’ll also need to make several if statements to test the interrupted status to do or not to do some operations.

```
catch (InterruptedException e) {
Thread.currentThread().interrupt(); continue;}
```

t. isAlive()
e.g. printing the time for a certain period

```
 public static void main(String[] a) {
TimerThread normal_clock = new TimerThread(NORMAL_CLOCK);
TimerThread count_down = new TimerThread(COUNT_DOWN);
normal_clock.start(); // gets the current time
count_down.start(); // starts from a value, e.g. 5 mins, and counts down every second
while (count_down.isAlive()) {  // count_down just terminates after the fixed period
        System.out.print(  …the times from our normal clock….)
       try {
          sleep(1);
        } catch (InterruptedException e) {
          System.out.println("Interrupted.");
        }
```

isAlive( ) returns **true** if the thread upon which it is called is still running - **false** otherwise. While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( )**

Daemon Threads

```
public class DaemonTest {
public static void main(String[] args) {
new WorkerThread().start();

try { Thread.sleep(7500); } catch
(InterruptedException e) {}
System.out.println("Main Thread ending") ; }
}
```

```
class WorkerThread extends Thread {
public WorkerThread() { setDaemon(true) ;}

public void run() { int count=0 ;
while (true) { System.out.println("Hello from
Worker "+count++) ;
try { sleep(5000); } catch (InterruptedException e)
{} } //end of while
} }
```

Interpretation

setDaemon(false)

(i.e. when it's a user thread),

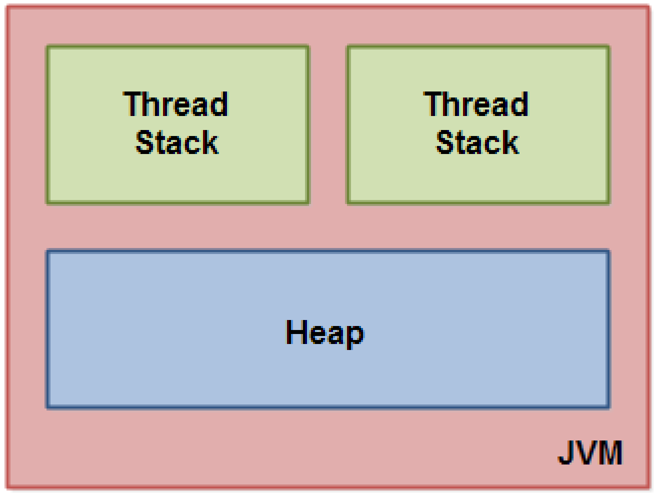
the Worker thread continues to run. After main has finished.

setDaemon(true)

(i.e. when it's a daemon thread)

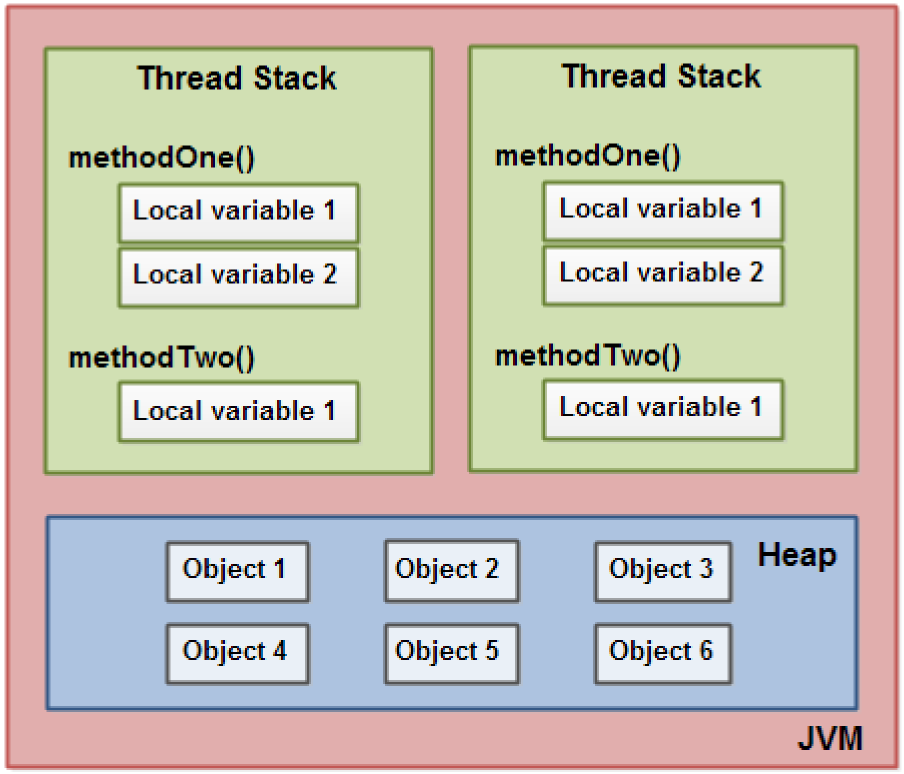
the Worker thread terminates when the main thread terminates.

Java Memory model



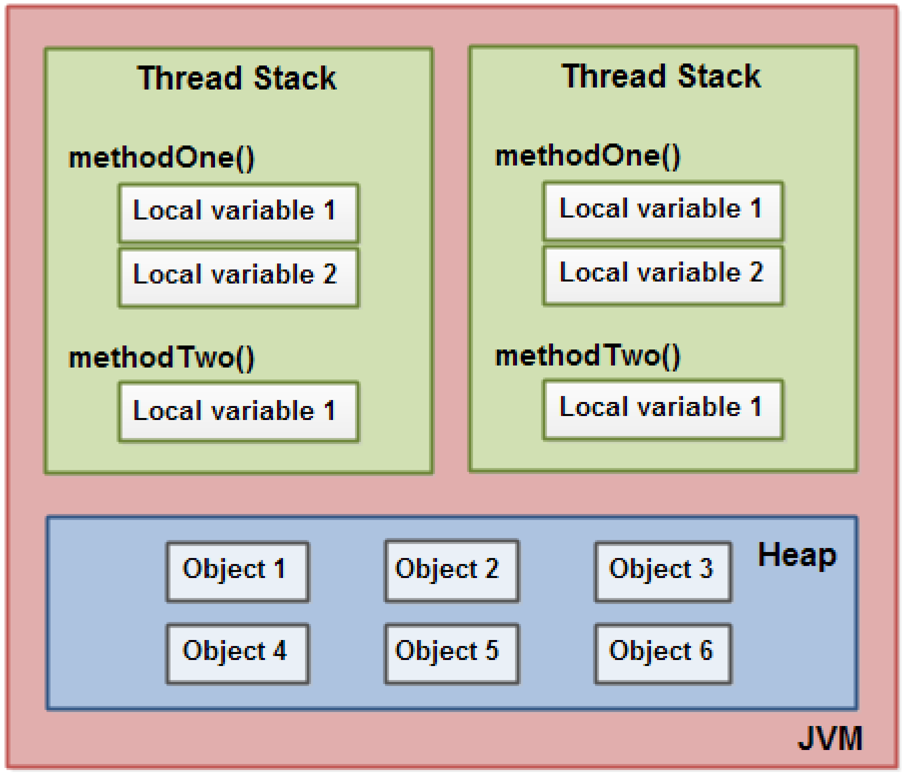
- Each thread running in the Java virtual machine has its own thread stack
- Share the heap

primitive types are stored as local variables in the stack



- primitive types :-
  boolean, *byte*, *short*, *char*,
  int, *long*, *float*, *double*
- Hence not visible to other threads.
- One thread **may pass a copy** of a primitive variable to another thread,
  - » but it cannot share the primitive local variable itself.

A local variable may also be a reference to an object



- the object itself if stored on the heap
- An object may contain methods and these methods may contain local variables.
  - These local variables are also stored on the thread stack, even though the object the method belongs to is stored on the heap.
  - when the method is called

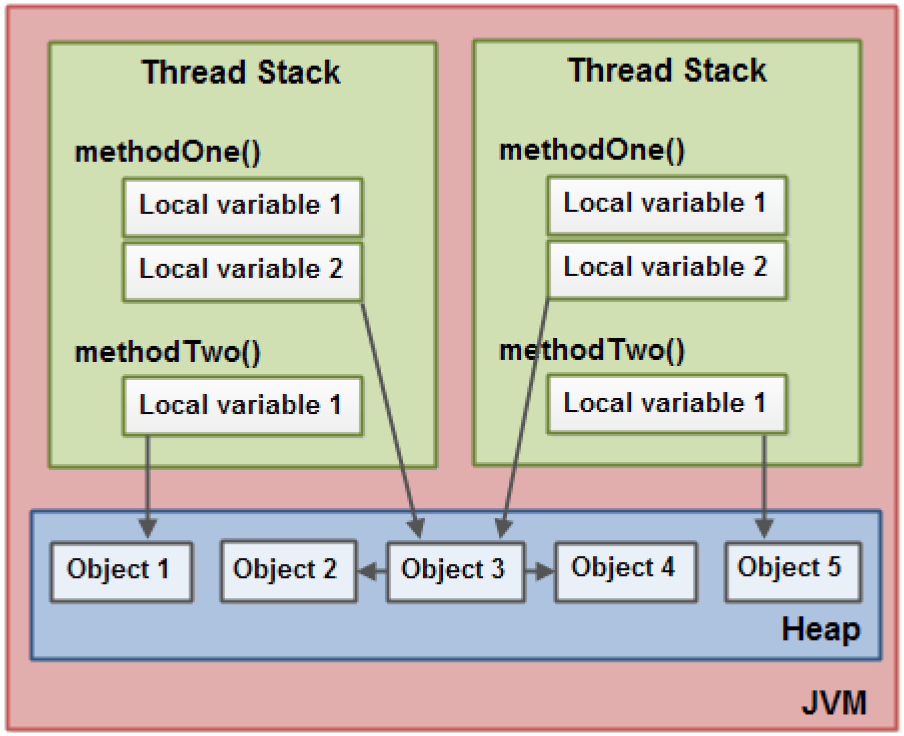
A **member variable** of an object is stored on the heap with the object

- **Whether**
  - » the member variable is of a primitive type
- **Or**
  - » it is a reference to an object.

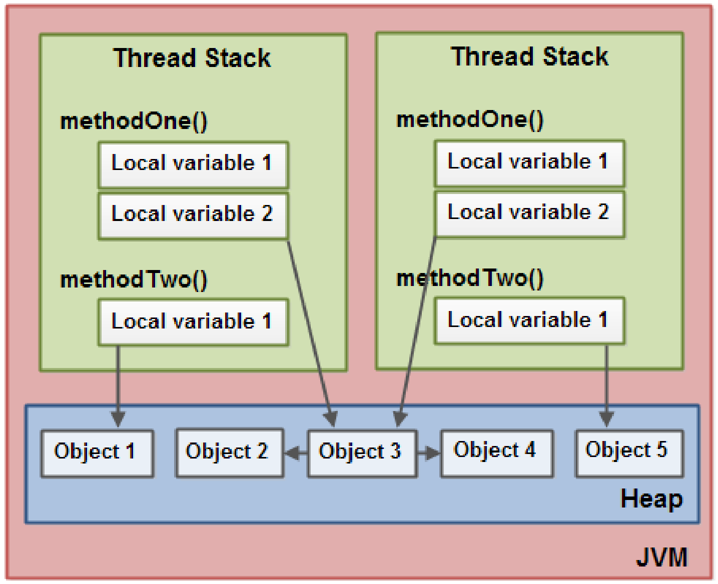
- Objects on the heap can be accessed by all threads that have a reference to the object.
- When a thread has access to an object, it can also get access to that object's member variables.

Example:

- If two threads call a method on the same object at the same time, they will both have access to the object **member** variables, but each thread will have its own copy of the **local** variables.



Many objects can be shared through references



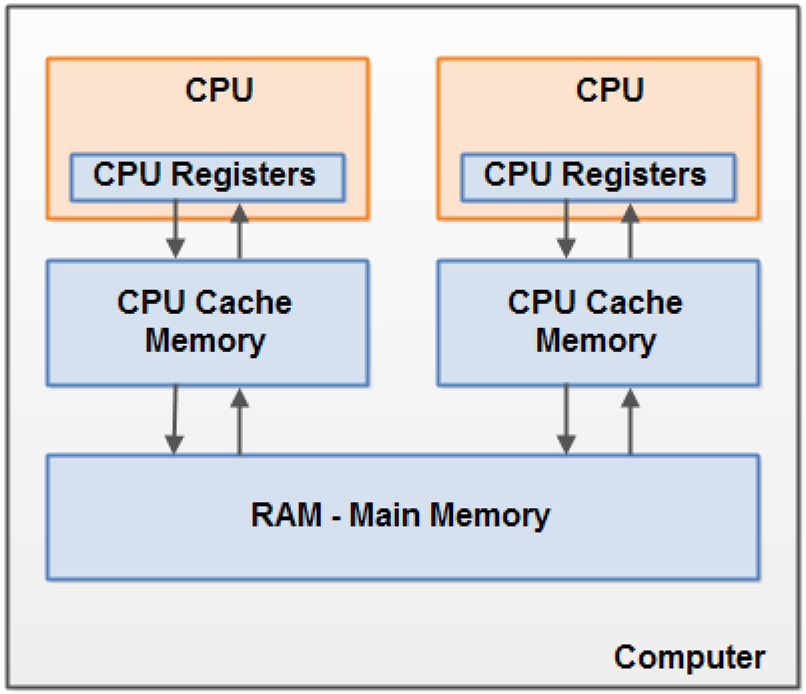
See http://tutorials.jenkov.com/java-concurrency/java-memory-model.html

- Suppose shared Object 3 has a reference to Object 2 and Object 4 as member variables
  - » See arrows
- Both threads can access Object 2 and Object 4 via the member variable references in Object 3

Static class variables

- Static class variables are also stored on the heap along with the class definition.

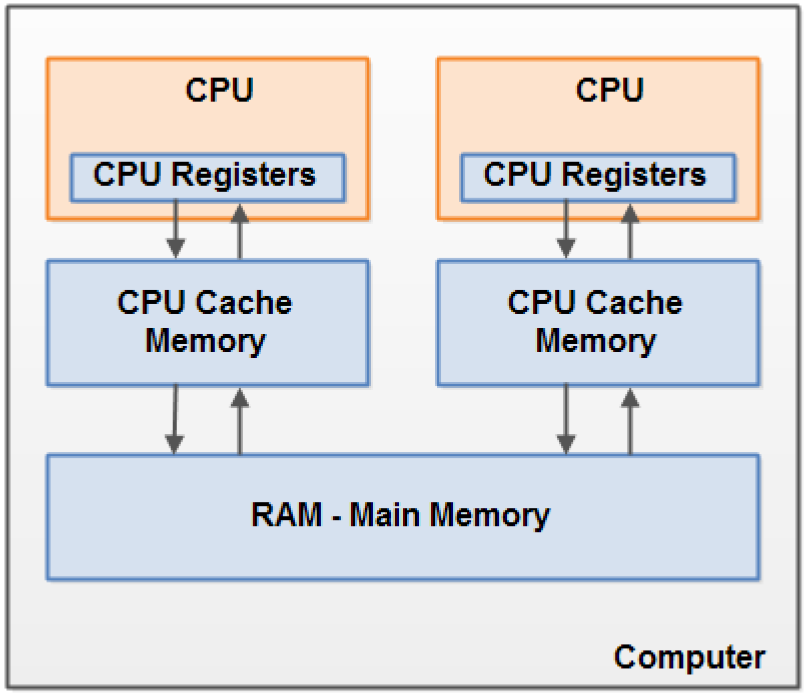
How the Java memory model works with a hardware memory architecture.



- Each CPU is capable of running one thread at any given time.
- Hence if application is multithreaded, one thread per CPU may be running truly concurrently

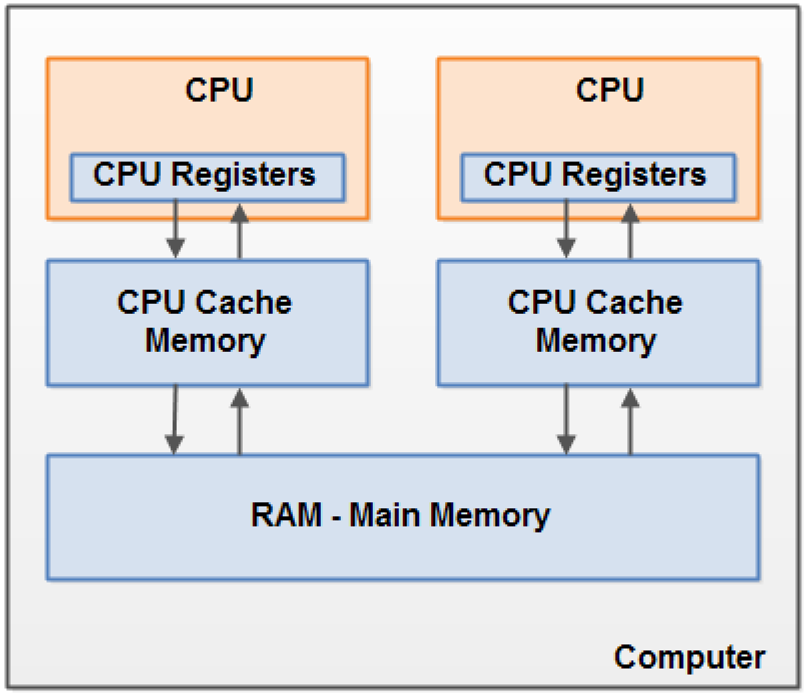
Each CPU typically has a CPU cache memory layer (s)

- The CPU can access its cache memory much faster than main memory
- Typically not as fast as it can access its internal registers.
- Typically cache memory speed is
  - » < internal register speed and
  - » > main memory speed



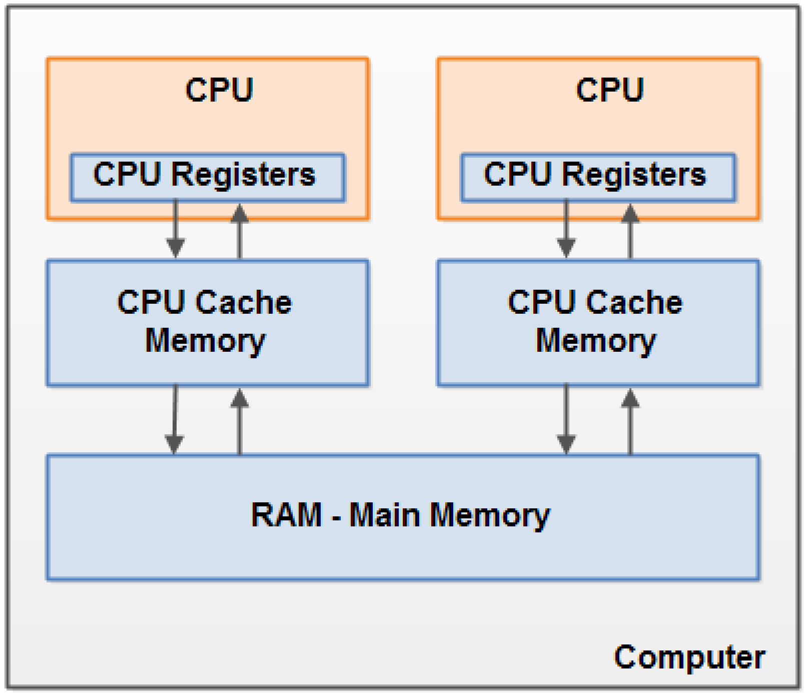
CPUs can access the bigger main memory

- When a CPU needs to access main memory it will read part of main memory into its CPU cache.
  - It may even read part of the cache into its internal registers and then perform operations on it.



Writing back to main memory

- When the CPU needs to write the result back to main memory it
  - » will flush the value from its internal register to the cache memory, and
  - » at some point flush the value back to main memory.
- Typically flushed back to main memory **when the CPU needs to store something else in the cache memory**.

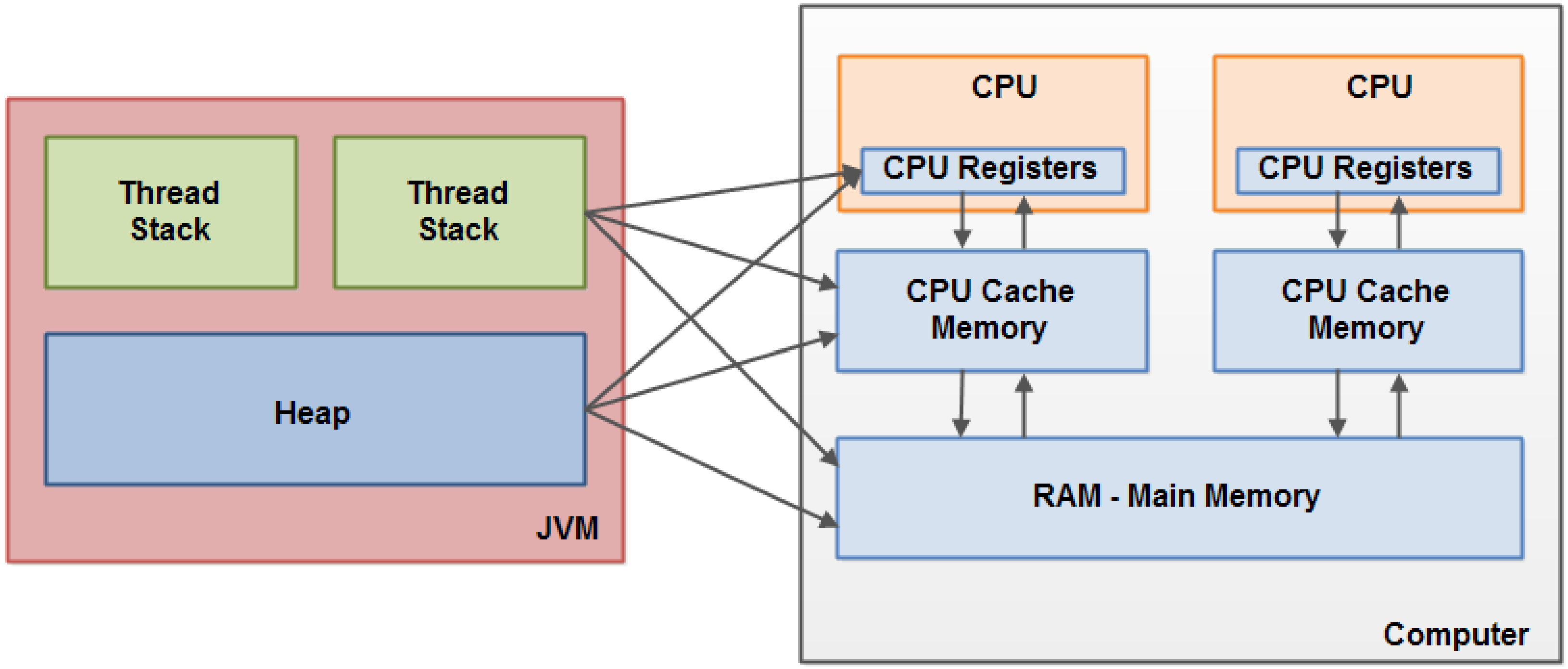


It does not necessarily read / write the full cache each time

The java memory model and hardware model are different

- The hardware memory architecture does not distinguish between thread stacks and heap.
  - It is just memory
- On the hardware, **both** the thread stack and the heap are located in main memory.
- Parts of the thread stacks and heap may sometimes be present in CPU caches and in internal CPU registers.

Parts of the thread stacks and heap may be held in CPU caches and registers.

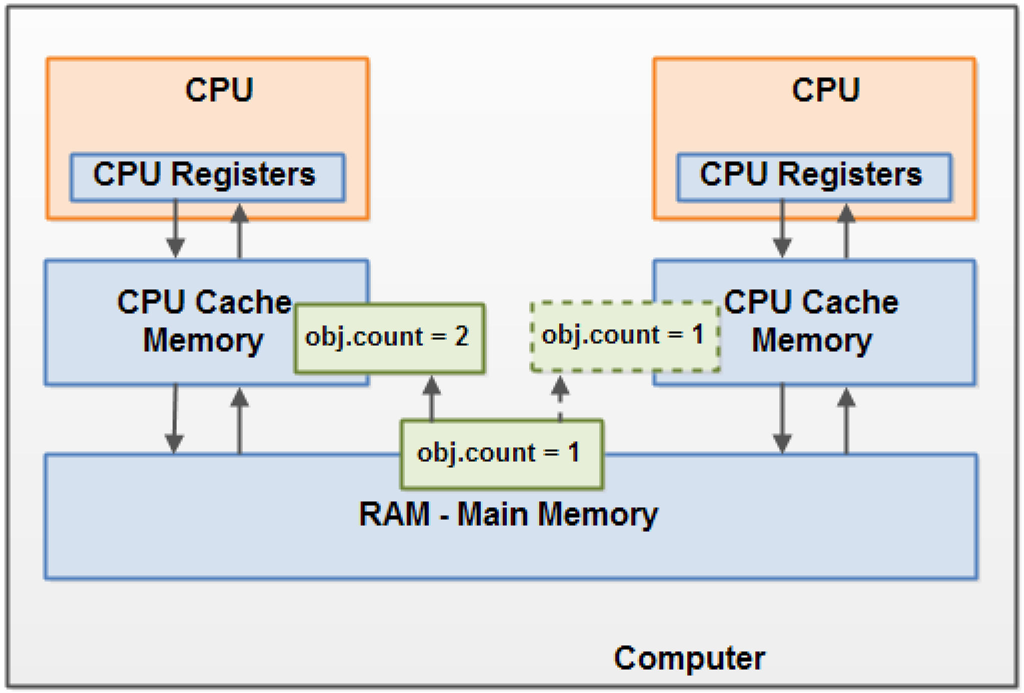


Storage in different memory areas may cause problems

- **Visibility** of thread updates (writes) to shared variables.
- **Race conditions** when reading, checking and writing shared variables.

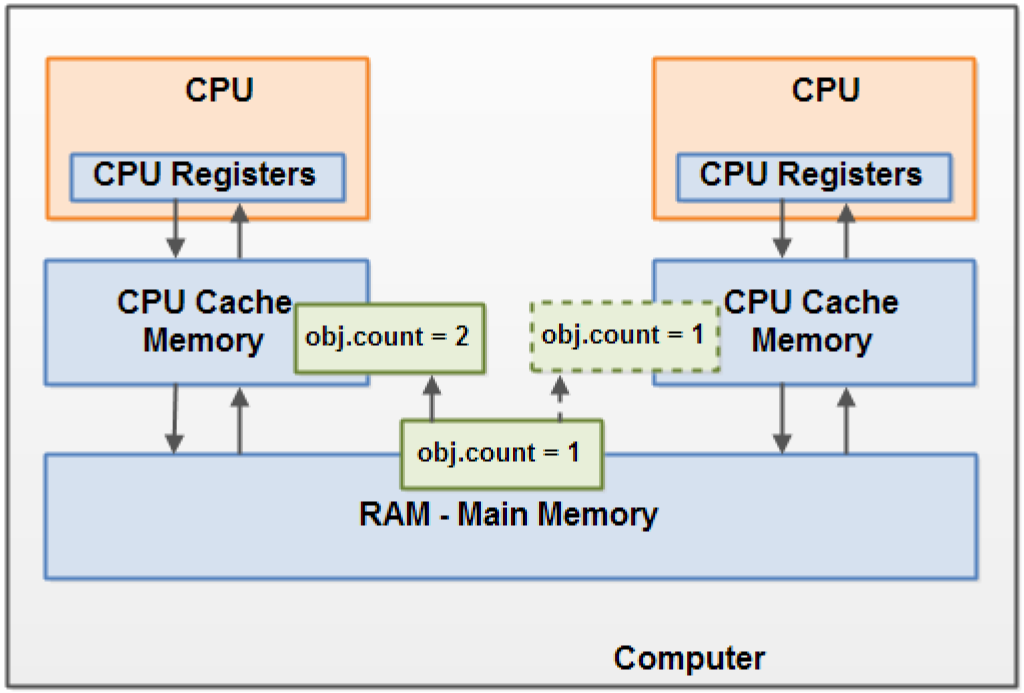
Visibility of shared objects

- Imagine that the shared object is initially stored in main memory.
  - obj.count
- A thread running on CPU one then reads the shared object into its CPU cache.
- There it makes a change to the shared object.



Visibility of shared objects

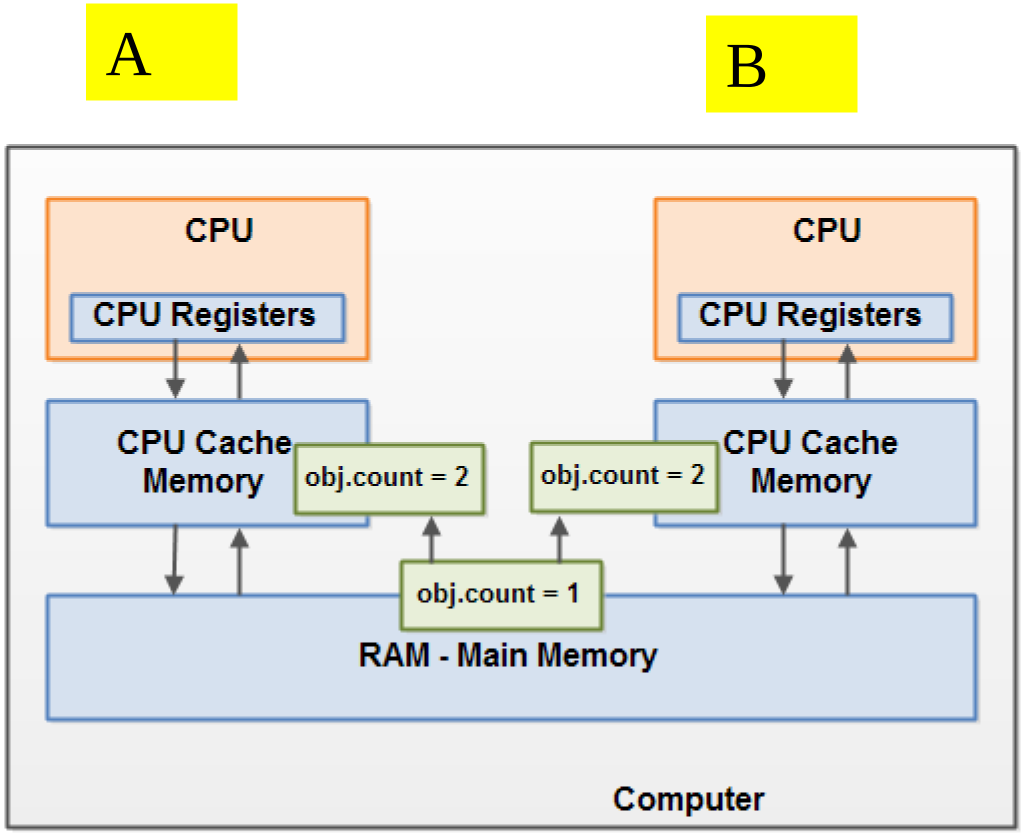
- As long as the CPU cache has not been flushed back to main memory, the changed version of the shared object is not visible to threads running on other CPUs.
- Hence each thread may end up with its own copy of the shared object,
  - » each copy sitting in a different CPU cache.



Solved using volatile

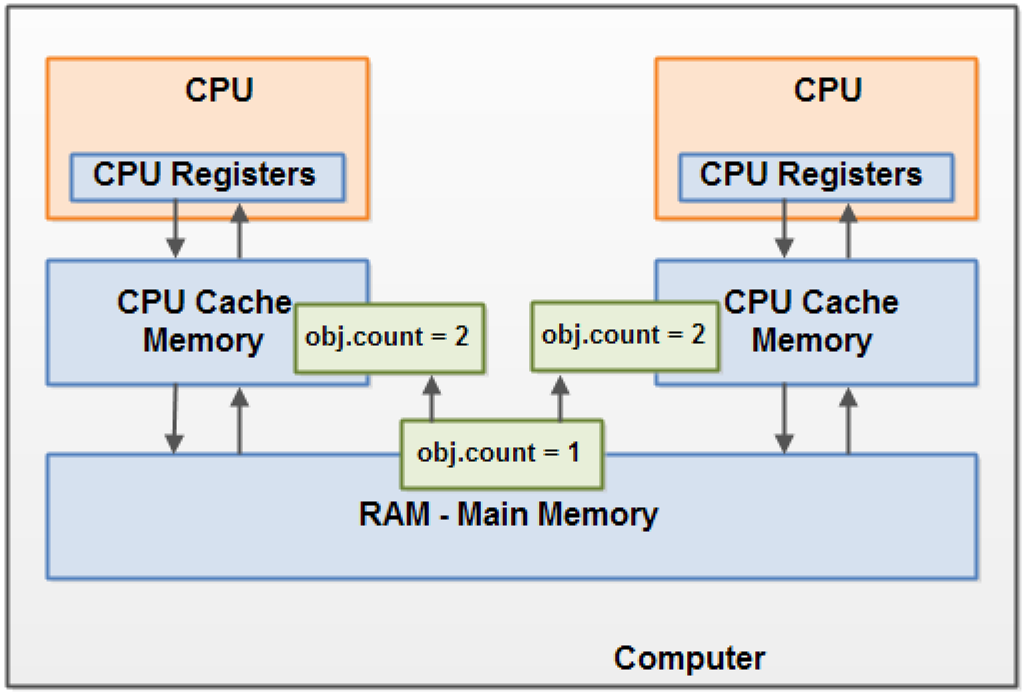
Race condition

- thread A reads the variable count of a shared object into its CPU cache
- thread B does the same, but into a different CPU cache.
- Now thread A adds one to count, and thread B does the same.



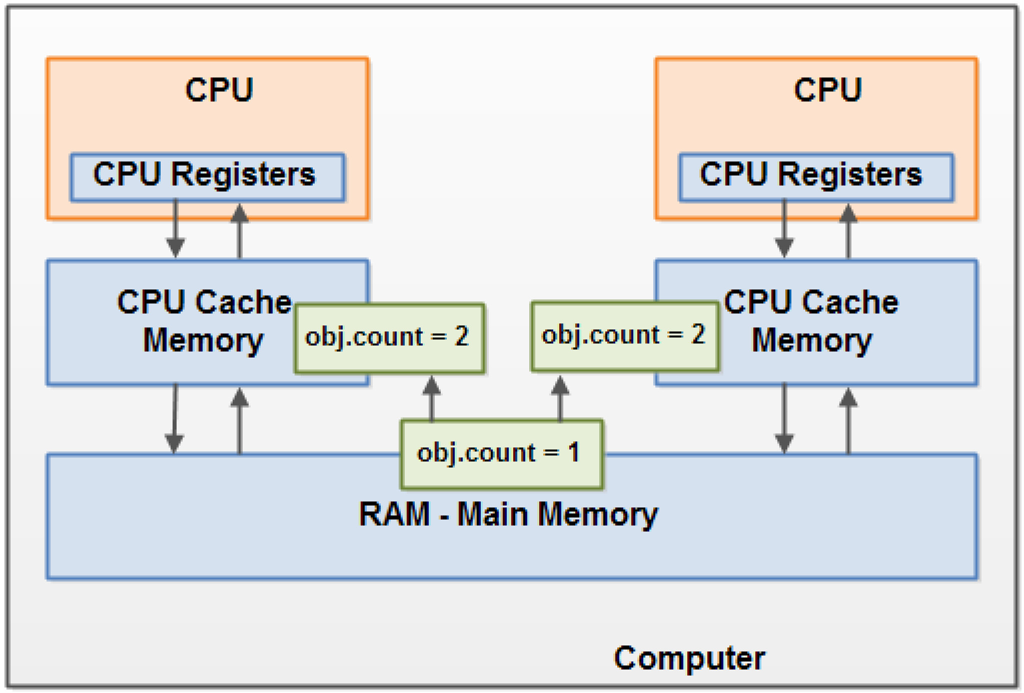
If sequential count=3

- The variable has been incremented two times, once in each CPU cache.
- *If these increments had been carried out sequentially*, the variable count would be been incremented twice and had the original value + 2 written back to main memory.



However, the two increments have been carried out concurrently *without proper synchronization*

- Regardless which of thread A and B that writes its updated version of count back to main memory,
  - the updated value will only be 1 higher than the original value,
  - despite the two increments.



This is solved by synchronization

Volatile

- Volatile is used to indicate that a **variable's value will be modified by different threads**
- The value of this variable will **never be cached thread-locally**: all reads and writes will go straight to "main memory";
- Access to the variable **acts as though it is enclosed in a synchronized block**, synchronized on itself.
  - » We say "acts as though", because there is no actual lock object involved (at least to the programmer).
  - » The *load*, *store*, *read*, and *write* actions on volatile variables are atomic, even if the type of the variable is double or long.

Common traps with volatile

- If you declare arr[] as volatile, that means that the *reference* to the array is volatile; but individual field accesses (e.g. arr[5]) are **not thread-safe**
- **Unary operations (++, --) *aren't atomic***
  - A danger of "raw" volatile variables is that they can misleadingly make a non-atomic operation look atomic. For example:

```
volatile int i; ... i += 5;
```

NOT SAFE

Java 5

- *From Java 5 truly atomic get-and-set operations are permitted with volatile*
  - *With special atomic wrapper classes (not covered)*
  - *Also an efficient means of accessing the nth element of a volatile array (and performing atomic get-and-set on the element) is provided.*

Why volatile?: to stop caching

```
public class StoppableTask extends Thread {
private volatile boolean pleaseStop;
public void run() {
while (!pleaseStop) { // do some stuff... } }
public void tellMeToStop() { pleaseStop = true; } }
```

read

write

e.g. does not alter pleaseStop

If the variable were not declared volatile (and without other synchronization), then it would be legal for the thread running the loop to cache the value of the variable at the start of the loop and never read it again.

If you don't like infinite loops, this is undesirable!

NB pleaseStop is effectively accessed by two threads

When not to use volatile

- volatile is **not** necessary– or in fact possible– for fields that are **immutable** (declared **final**);
- volatile is **not** necessary for variables that are accessed by **only one thread** (make sure this is true!);
- volatile is **not** suitable for **complex operations** where you need to **prevent access** to a variable for the duration of an operation
  - Even things like x=x+5
  - in such cases, you should use object synchronization or one of Java 5's explicit lock classes.

When to use volatile

- We write a variable, such as a flag, in one thread;
  - » Such as calling tellMeToStop
- We check that variable in another thread;
- crucially, the **value to write *doesn't* depend** on the current value...
  - » We are just setting it because of other reasons
  - » We do not need to check and then set

Back to stopping a thread using a stop flag

- In order for a thread to respond to the stop flag the thread has to be running
  - » So checking on stop flag not always OK
- If thread is in a non-runnable state, setting a stop flag variable will have no effect!
  - – A thread is in a non-runnable state if
    - Its *sleep* method is invoked.
    - The thread calls the *wait* method to wait for a specific condition to be satisfied.
    - The thread is blocking on I/O.
  - » To get the thread to stop **promptly** you need to break out of the sleep or wait or join using **interrupt()**

On another point: Thread.yield() may be better than Thread.sleep()

- *yield* stops the thread running continuously and gives some time back to the OS.
- It just gives up the thread's turn, and regains it in the next round.
- *sleep*() has a slightly larger overhead because it creates a system that includes some kind of timer that will wake the process. (Depends on implementation basically)
- Hence yield sometimes better

if the thread is not in a sleep() or wait() or… , then interrupt() does not throw an exception.

```
 Thread.yield();
// let another thread have some time
perhaps to interrupt this one.
if
(Thread.currentThread().isInterrupted())
{ //checking if there has been an
  // interrupt and do something, e.g.
throw new InterruptedException ("Stopped
after yield”}
```

The main advantage to Thread.interrupt()

- **is that you can immediately break out of interruptable calls,**
  - » **which you can't do with the flag approach.**
- You should routinely check at strategic points where you can safely stop and cleanup

```
if (Thread.currentThread.isInterrupted()) {
// cleanup and stop execution }
```

Code that can be accessed by multiple threads must be made thread safe.

```
public class Example {
private int value = 0;
public int getNextValue(){ return value++; }
}
```

- An increment like this is not a simple action, but three actions :
  - Read the current value of “value”
  - Add one to the current value
  - Write that new value to “value”
- Multiple threads may call *getNextValue*()
- Volatile (on its own) does not help as value++ is not atomic

http://www.baptiste-wicht.com/2010/08/java-concurrrency-synchronization-locks/

Critical sections

- The code segments within a program that access the same data from within separate, concurrent threads are known as *critical sections*
- In the Java language, you mark critical sections in your program with the synchronized keyword.
- Each object in Java is associated with a **monitor**, which a thread can *lock* or *unlock*
  - *See Entry Set (which looks after the locking) and Wait Set explained later*
- In the Java language, the unique **monitor** comes into play with
  - every object that has a synchronized method or
  - if the object is the target of a synchronized section. (later)
- Will first look at this intrinsic lock or monitor lock

Each Java **object** contains an **intrinsic lock**

```
public class Example {
private int value = 0;
public synchronized int getNextValue(){ return value++; }
}
……
e=new Example();
e.getNextValue();
```

- You can add the synchronized keyword to a method to acquire the lock before invoking the method and release it after the method execution.
- You have the guarantee that only one thread can execute the method at the same time.
- The used lock is the intrinsic lock of the instance

Intrinsic lock

- Makes methods (or statements – later) atomic.
- When a thread has a lock, no other thread can acquire it
  - » It must wait for the first thread to release the lock.
- If you have two methods with the synchronized keyword,
  - » **only one method of the two will be executed at the same time**
  - » because the **same lock** is used for the two methods.

Alternative: explicit use of the intrinsic lock - on an Example instance

```
public class Example {
private int value = 0;
public int getNextValue() { synchronized
(this) { return value++; } } }
```

- Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:
- Generally, critical sections in Java programs are methods.
- **You can mark smaller code segments as synchronized.**
- However, this violates object-oriented paradigms and leads to confusing code that is difficult to debug and maintain.
- For the **majority of your Java programming purposes, it's best to use synchronized only at the method level.**

Synchronized methods

```
protected synchronized int
getNextAvailableItem() {… return items; }
```

=

```
protected int getNextAvailableItem() {
synchronized (this){… return items; }
}
```

Can be less efficient but from a design perspective better

If the method is static

- the used lock is the Class object.

Example: Instead of using synchronized methods we create two objects solely to provide locks.

```
public class MsLunch {
private long c1 = 0;
private long c2 = 0;
private Object lock1 = new Object();
private Object lock2 = new Object();
public void inc1() { synchronized(lock1) { c1++; }
}
public void inc2() { synchronized(lock2) { c2++; }
} }
```

class MsLunch has two instance fields, c1 and c2, that are never used together.
All updates of these fields must be synchronized, ……BUT
No reason to prevent an update of c1 from being interleaved with update of c2

You can **choose the lock to block on**

- E.g. if you don’t want to use the intrinsic lock of the current object but an other object, you can use an other object just as a lock :

```
public class Example {
private int value = 0;
private final Object lock = new Object();
public int getNextValue() {
     synchronized (lock) { return value++; } } }
```

The result is the same but has the differences:

- » We are not locking on e (the instance of Example)
- » The new lock is internal to the object (and private) so no other code can use the lock.

With complex classes, it not rare to use several locks to provide thread safety on the class.

Fully Synchronizing objects

- a class in which every method is synchronised (that of course has no public instance variables) *guarantees locally sequential behaviour*
- they only do one thing at a time. They are either:
  - ready (idle - not having the monitor),
  - active (processing a method), or
  - waiting(for a reply)

the main differences between synchronized and volatile

- a primitive variable may be declared volatile
  - » you can't synchronize on a primitive with synchronized
- an access to a volatile variable **never has the potential to block**: we're only ever doing a simple read or write,
  - » so unlike a synchronized block we will never hold on to any lock;
  - » because accessing a volatile variable **never holds a lock**, it is **not suitable** for cases where we want to ***read-update-write*** as an atomic operation (unless we're prepared to "miss an update");

A fully synchronised class

```
public class ExpandableArray{
private Object[] data;
private int size;
public ExpandableArray(int cap){}
public synchronized int size(){return size;}
public synchronized Object at(int i) throws
    NoSuchElementException {...else return data[i];}
public synchronized void append (Object x){}
public synchronized void removeLast() throws
    NoSuchElementException{}
}
```

Creating an ExpandableArray

```
public class ExpandableArray{
private Object[] data;
private int size;        //the size actually used

public ExpandableArray(int cap){
data=new Object[cap];
size=0;}

public synchronized int size(){ return size;}
                              }
```

Creating an ExpandableArray

```
public synchronized Object at(int i) throws
   NoSuchElementException {
if (i<0 || i>= size) throw NoSuchElementException;
else return data[i];
}
public synchronized void removeLast() throws
   NoSuchElementException{
if(size==0) throw NoSuchElementException();
else data[--size]=null;    }
```

Creating an ExpandableArray

```
public synchronized void append (Object x){
    if(size >= data.length){
    Object[] olddata=data;
    data=new  Object[3*(size+1)/2];
    for(int i=0;i<size; i++) data[i]=olddata[i];
    }
    data[size++]=x;    // the next time size may be too large, hence
                       //check above
}
```

Questions

- If a synchronized method contains a call to a non-synchronized, can another method still access the non-synchronized method at the same time?
  - » Yes. Other methods can access non-synchronized methods.
- **Java synchronized keyword is re-entrant in nature** it means if a synchronized method calls another synchronized method which requires same lock then the current thread that is holding lock can enter into that method without acquiring lock.
  - » Counter is used

Questions

- One more **limitation of java synchronized keyword** is that it can only be used to control access of shared object within the same JVM. If you have more than one JVM and need to synchronized access to a shared file system or database, the Java synchronized keyword is not at all sufficient. You need to implement a kind of global lock for that.

A read write lock is a common requirement

- Imagine you have an application that reads and writes some resource,
  - but writing it is not done as much as reading is.
- Two threads reading the same resource does not cause problems for each other, so multiple threads that want to read the resource are granted access at the same time, overlapping.
- But, **if even a single thread** wants to write to the resource, no other reads nor writes must be in progress at the same time.

Java 5 comes with read / write lock

- Synchronizing on read and write would work, but slow.
- To solve this problem of allowing multiple readers but only one writer, you will need a “read / write lock.”
  - » implementations are in the java.util.concurrent package.
  - » BUT you could implement yourself. Need *wait* and *notify* which we will do soon

Partial Synchronization

```
public class LinkedCell{
protected double value;
protected LinkedCell next;// once set is assumed not to change
public LinkedCell (double v, LinkedCell t){….}
public synchronized double value(){ return value;}
public synchronized double setValue(){ value=v;}
public LinkedCell next(){ return next;}
public double sum() {……}
public boolean includes(double x) {…..}              }
```

Partial Synchronization

```
public class LinkedCell{
protected double value;
protected LinkedCell next;

public LinkedCell (double v, LinkedCell t){
value=v;
next=t;}
```

Partial Synchronization

```
public LinkedCell next(){ return next;}

public synchronized double value(){ return value;}
public synchronized double setValue(){ value=v;}
```

Each LinkedCell instance will have a monitor associated with it

Partial Synchronization

Not synchronized so many threads can call this at same time

```
public double sum(){
double v= value();
```

get the value by the synchronized accessor

*so value of **this** cell cannot change while being read*

```
if (next() != null) v = v + next().sum() ;
return v;
}
```

Values in cells not currently being accessed could be changed during calculation of sum()

Scope of a lock

- The time between the when the lock is taken and when the lock is released
  - so far everything has been *method scope*
- Lock scopes can be determined by segments of code within a method
- Remember, as always, **locks apply to objects** not methods

Scope of a lock

public boolean includes(double x){

synchronized(this)

{ if(value==x) return true;}

dealing with values

*& the **cell denies access** to all **synchronised** functions or **synchronized** code segments*

if(next()==null) return false;

else return next().includes(x);}

dealing with linkage so no need for synchronization

}

Full synchronization

```
synchronized double badsum()
{double  v= value();
  return  v + nextSum();
}
double nextSum(){
return (next()==null) ? 0: next().sum();
}
```

LinkedCell receiving the badsum message remains locked for the duration of nextSum

Synchronization: Summary

- Synchronization is implemented by exclusively accessing the underlying internal lock associated with an object
  - » including the class object for static methods
- Each lock acts as a counter
  - » if counter is not zero on entry to a synchronized method or block because another thread holds the lock, the current thread is blocked until the count is 0
  - » on entry the counter is incremented
  - » on return or exit by an exception the counter is decremented

Synchronization: Summary

- Individual code blocks within a method can be synchronized using
  - » synchronized(someObject)

    { code wanting exclusive access to someObject}
- someObject is often *this*
- a synchronized method is equivalent to a non-synchronized method with all its code enclosed by synchronized(this)
  - » e.g. void methodname(){ synchronized(this){…. } }

Synchronization: Summary

- Any method or code block marked as synchronized is executed in its entirety (unless explicitly suspended by a *wait* - later) before the receiving object is allowed to perform any other synchronized method called from any other thread.
- If a method is not marked as synchronized then it can be executed immediately, even when another method, even a synchronized method, of the object is executing

Synchronization: Summary

- the synchronized qualifier is not automatically inherited
  - » you must qualify in the subclass when overriding else it is unsynchronized
- A non-static method can lock static data using a code block
  - » synchronized (getClass()) {…}

    is locking the class object and hence the static data

Reminder: Limits of volatile

volatile int i; ...

i += 5;

So another thread *could* sneak in the middle of the overall operation of i += 5.

Therefore volatile is not enough if we want i += 5; to be thread safe

Note that we can't literally synchronize on an int primitive

```
int temp;
synchronized (i) { temp = i; } temp += 5;
synchronized (i) { i = temp; }
```

Not possible. Need to synchronize on an object

Prior to Java 5

- Pretty much the only solution is to introduce an explicit lock object and
- synchronize on it around all accesses to the variable.
- Now more classes that provide atomic behaviour
  - E.g. BlockingQueue classes provide a more convenient means of implementing job queues.
- Still common

Monitors

- Just as every Java object has a **lock**, every Java object has a **wait set** that is manipulated by *wait*(), *notify*() and *notifyAll*() and *Thread.interrupt*()
- **Entities that possess both locks and wait sets are usually called monitors**
  - » Details different in different languages
- Any object can serve as a monitor but need to have a synchronized lock on an object for it to do so
  - » Typically `synchronized(O) { }`

for some object O to act as a monitor or just by creating O which has synchronised methods. Other threads might use O in their constructors and hence have access.

Java's **monitor** supports two kinds of thread synchronization: *mutual exclusion* and *cooperation*

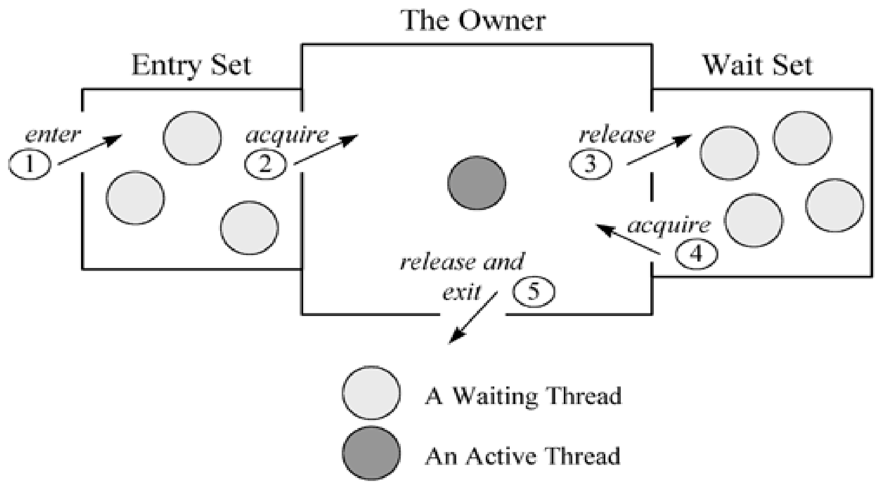
- Mutual exclusion: supported by object locks, enables multiple threads to independently work on shared data without interfering with each other.
  - Been looking at this
- Cooperation: supported by the wait and notify methods of class Object, enables threads to work together towards a common goal.
  - This has been very limited until now!

The monitor object is like a building

- that contains one special room that can be occupied by **only one thread** at a time.
  - This room usually contains some data.
  - From the time a thread enters this room to the time it leaves, it has exclusive access to any data in the room.
- Entering the monitor building is called "entering the monitor."
- *Entering the special room* inside the building is called "*acquiring* the monitor."
- **Occupying** the room is called "**owning** the monitor," and leaving the room is called "**releasing** the monitor." Leaving the entire building is called "exiting the monitor."
- Note we could use the word lock but monitor is better as we are extending the concept here

Monitor has an entry set, a wait set and the owner

- In the centre, a large rectangle contains a **single** thread, the monitor's owner.
- On the left, a small rectangle contains the entry set.
- On the right, another small rectangle contains the wait set.
- Active threads are shown as dark grey circles.
- Suspended (sleeping) threads are shown as light grey circles.



When a thread arrives at the beginning of a monitor region

- it is placed into an *entry set* for the associated monitor.
- If no other thread is waiting in the entry set and no other thread currently owns the monitor, the thread acquires the monitor and continues executing the monitor region.
- When the thread finishes executing the monitor region, it exits (and releases) the monitor.

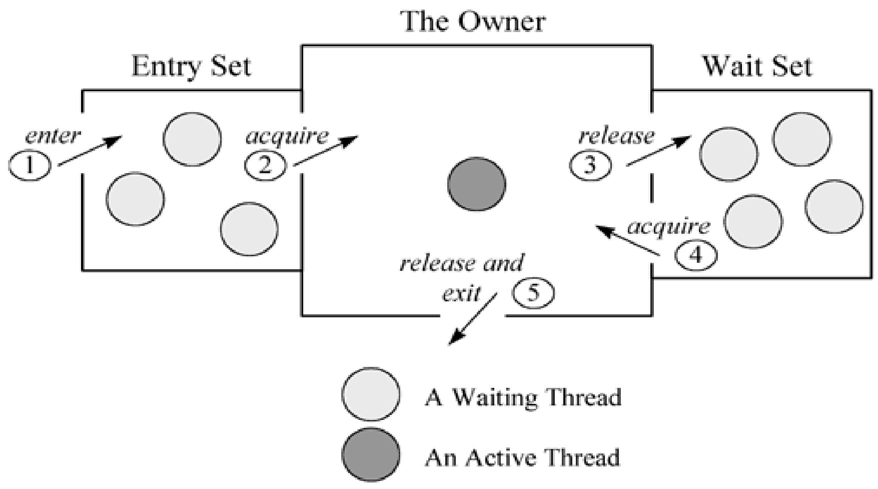


Figure 20-1. A Java monitor.

Recap

- A monitor/critical region is code that needs to be executed as one indivisible operation with respect to a particular monitor.
- In other words, one thread must be able to execute a monitor region from beginning to end without another thread concurrently executing a monitor region of the same monitor.
  - A monitor enforces this one-thread-at-a-time execution of its monitor regions.
- The only way a thread can enter a monitor is by arriving at the beginning of one of the monitor regions associated with that monitor.
- The only way a thread can move forward and execute the monitor region is by acquiring the monitor

Cooperation is…

- important when one thread needs some data to be in a particular state and another thread is responsible for getting the data into that state.
- E.g. one thread, a "read thread," may be reading data from a buffer that another thread, a "write thread," is filling the buffer.
  - » The write thread is responsible for filling the buffer with data.
  - » Once the write thread has done some more writing, the read thread can do some more reading.
  - » The **read thread needs the buffer to be in a "not empty" state** to read out of the buffer.
  - » i.e. if **the buffer is empty, it must wait**.

Arriving at the monitor region (again)

- When a thread arrives at the start of a monitor region, it enters the monitor via the leftmost door, door number one, and finds itself in the rectangle that houses the entry set.
- If no thread currently owns the monitor and no other threads are waiting in the entry set, the thread passes immediately through the next door, door number two, and becomes the owner of the monitor.
- As the monitor owner, the thread continues executing the monitor region.
- **If**, on the other hand, **there is another thread currently claiming ownership of the monitor, the newly arrived thread must wait in the entry set**, possibly along with other threads already waiting there.
- The newly arrived thread is blocked and therefore doesn't execute any further into the monitor region.

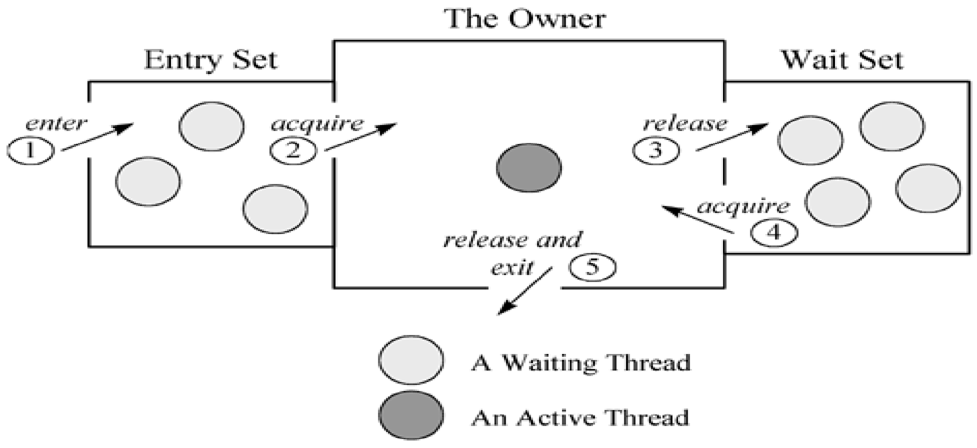


Figure 20-1. A Java monitor.

Releasing the monitor

- E.g. three threads are suspended in the entry set and four threads suspended in the wait set.
- These threads will remain where they are until the current owner of the monitor--the active thread--**releases the monitor**.
- The active thread **can release the monitor in either of two ways:**
  - » it can complete the monitor region it is executing. If it completes the monitor region, it exits the monitor via the door five.
  - » If it executes a wait command, it
    - – **releases the monitor and goes through door number three**

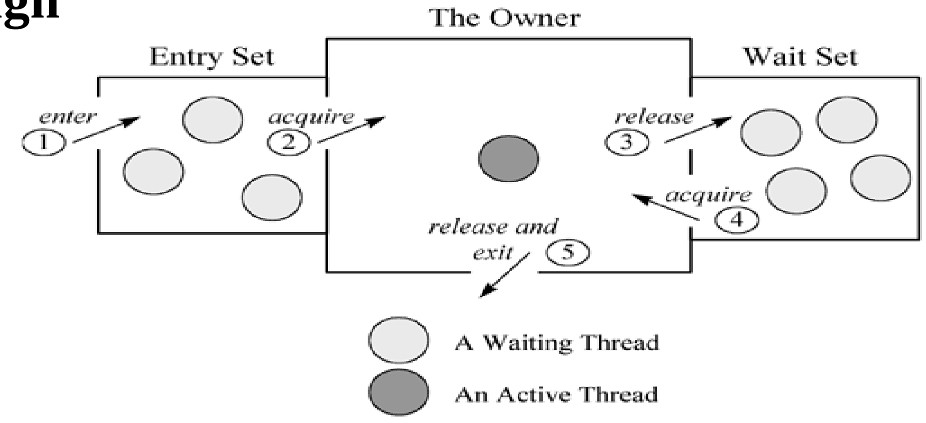
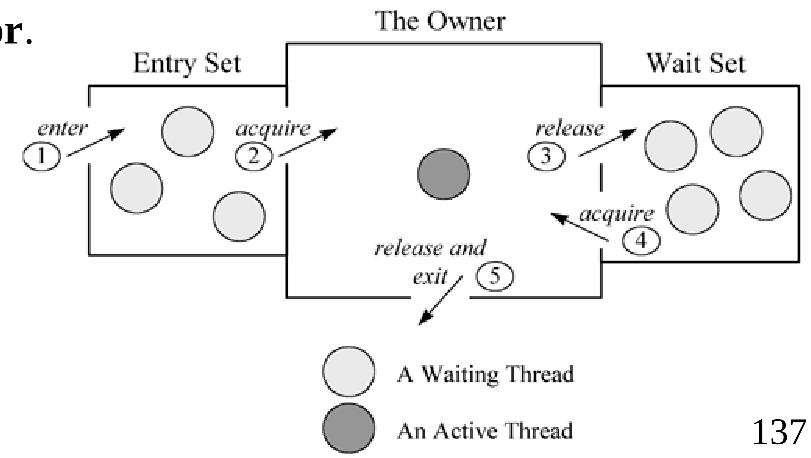


Figure 20-1. A Java monitor.

Notify says who can compete

- If the (former) monitor owner **did not execute a notify before it released the monitor** (and **none of the waiting threads were previously notified** and were waiting to be resurrected), then **only** the threads in the entry set will compete to acquire the monitor.
- If the former owner **did** execute a notify, then the entry set threads will have to compete with one or more threads from the wait set.
- **If a thread from the entry set wins** the competition, it passes through door number two and becomes the new owner of the monitor.
- **If a thread from the wait set wins** the competition, it exits the wait set and reacquires the monitor as it passes through door number four.
  - » Note that doors three and four are the only ways a thread can enter or exit the wait set.
- A thread **can only execute a wait command if it currently owns the monitor**, and it **can't leave the wait set without automatically becoming again the owner of the monitor**.



The JVM monitor is a "Wait and Notify" monitor (aka "Signal and Continue")

- In this kind of monitor, a thread that currently owns the monitor can **suspend itself inside the monitor** by executing a *wait command.*
- When a thread executes a wait, it releases the monitor and enters a *wait set.*
  - » The thread will stay suspended in the wait set until some time after **another** thread executes a ***notify*** *command* inside the monitor.
- When a thread executes a notify, it continues to own the monitor until it releases the monitor of its own accord, either by executing a wait or by completing the monitor region.
- After the notifying thread has released the monitor, the waiting thread will be resurrected and will reacquire the monitor.

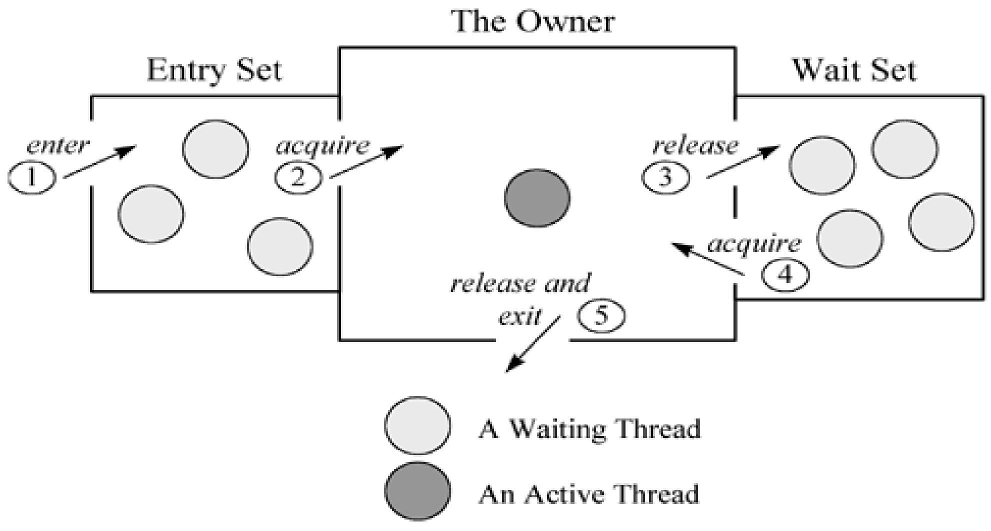


Figure 20-1. A Java monitor.

Why Signal and Continue ?

- because after a thread does a **notify (the signal) it retains ownership of the monitor and continues executing the monitor region (the continue).**
- *At some later time*, the notifying thread releases the monitor and **a** waiting thread is resurrected.
  - » Presumably, the waiting thread suspended itself because the data protected by the monitor wasn't in a state that would allow the thread to continue doing useful work.
  - » Also, the notifying thread presumably executed the notify command after it had placed the data protected by the monitor into the state desired by the waiting thread.
  - » **But because the notifying thread continued, it may have altered the state after the notify such that the waiting thread still can't do useful work.**
  - » Alternatively, a third thread may have acquired the monitor after the notifying thread released it but before the waiting thread acquired it, and the third thread may have changed the state of the protected data.

Why Signal and Continue ?

- **As a result, a notify must often be considered by waiting threads merely as a hint that the desired state *may* exist.**
- Each time a waiting thread is resurrected,
  - » it may need to check the state again to determine whether it can move forward and do useful work.
  - » If it finds the data still isn't in the desired state, the thread could execute another wait or give up and exit the monitor.

The JVM offers two kinds of notify commands: "notify" and "notify all."

- A notify command selects one thread **arbitrarily** from **the wait set and marks it for eventual resurrection**.
- A notifyAll command **marks all threads currently in the wait set for eventual resurrection**.
- Note: The acquisition and release of a monitor is done automatically and atomically by the Java runtime system.

How a JVM selects the a thread from the wait or entry sets is a design decision.

JVM Implementation designers can decide how to select:

- a thread from the wait set given a notify command
- the order to resurrect threads from the wait set given a notify all command
- the order to allow threads from the entry set to acquire the monitor
- how to choose between threads suspended in the wait set versus the entry set after a notify command

e.g. implement entry set and wait sets as first-in-first-out (FIFO) queues, so that the thread that waits the longest will be the first chosen to acquire the monitor.

e.g. have ten FIFO queues, one for each priority a thread can have inside the Java virtual machine. The virtual machine could then choose the thread that has been waiting the longest in the highest priority queue that contains any waiting threads.

Implementations are free to select threads in an arbitrary manner.

Note the following:

- ***wait*** **and** ***notify*** **should be placed within synchronized code to ensure that the current code owns the monitor**
- **You suspend yourself (sleep) with a wait when cannot continue**
- **You let waiting threads know that data has changed by notify**
  - » **NB saying change occurred but does not stop**
- http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html

Note the following:

- A call to *wait* from within synchronized code causes the thread to give up its lock and go to sleep. (and adds it to the wait set)
- This normally happens to allow another thread to obtain the lock and continue some processing.
- The *wait* method is meaningless without the use of *notify* or *notifyAll* which allows code that is waiting to be notified that it can wake up and continue executing.

Example: a buffer, a read thread, and a write thread.

- Assume the buffer is protected by a monitor.
- When a read thread enters the monitor that protects the buffer, it checks to see if the buffer is empty.
  - If the buffer is not empty, the read thread reads (and removes) some data from the buffer. Satisfied, it exits the monitor.
  - On the other hand, if the buffer is empty, the read thread executes a wait command. As soon as it executes the wait, the read thread is suspended and placed into the monitor's wait set.
- In the process, the read thread releases the monitor, which becomes available to other threads. At some later time, the write thread enters the monitor, writes some data into the buffer, executes a notify, and exits the monitor.

A buffer, a read thread, and a write thread.

- When the write thread executes the notify, **the** read thread is marked for eventual resurrection. After the write thread has exited the monitor, the read thread is resurrected as the owner of the monitor.
- **If there is any chance that some other thread** has come along and consumed the data left by the write thread, the **read thread must explicitly check to make sure the buffer is not empty**.
  - » If there is no chance that any other thread has consumed the data, then the read thread can just assume the data exists.
- The read thread reads some data from the buffer and exits the monitor.

example of using the *wait/notify:* **Producing code**

```
while(true){
try{ wait(); }catch (InterruptedException e) {} }
//some producing action goes here
notifyAll();
```

At first glance like it will just loop forever.

The wait method however effectively **means *give up the lock* on the object and wait until the notify or *notifyAll* method tells you to wake up.**

Example: many threads setting & accessing contents through public methods

```
class CubbyHole {
private int contents;
private boolean available = false;
public synchronized int get() {
while (available == false) {
   try { wait(); }
   catch (InterruptedException e)
      { }
   } //end while
available = false;
notifyAll();
return contents; }
}
```

```
public synchronized void put (int value)
{

while (available == true) { try { wait();
} catch (InterruptedException e) { } }
contents = value;
available = true;
notifyAll(); }
```

Notify does not release the monitor; only wait and } do

while

- Note the while loop……….Not an if
- Because the notify in the other method will inform all threads that are waiting, but only one will succeed.
  - Hence the others must wait
- **Spurious wakeups:** Even if one producer and one consumer we need loop
  - A waiting thread can sometimes be re-activated without notify() being called!
  - Putting this check in a while loop will ensure that if a spurious wake-up occurs, the condition will be re-checked, and the thread will call wait() again.

Common usage pattern: A Wait Condition

- A condition is a logical statement that must hold true for the thread to proceed
- If it is not true the thread must wait for the condition to become true

while(**!** *the condition which should be true*)

{ wait(); }

Example: The Producer and Consumer Threads (1/4)

```
public class Producer implements Runnable {
  private Buffer buffer;

  public Producer(Buffer b) { buffer = b; }

  public void run() {
    for (int i=0; i<250; i++) {
      buffer.put((char)('A' + (i%26)));
    }
  }
}
```

Method **put()** is **synchronized**, so it has the lock on the buffer **b** when running.

```
public class Consumer implements Runnable {
  private Buffer buffer;

  public Consumer(Buffer b) { buffer = b; }

  public void run() {
    for (int i=0; i<250; i++) {
      System.out.println(buffer.get());
    }
  }
}
```

Example: The Producer and Consumer Threads (2/4)

- The buffer has two constraints and two predicates to test them:

```
public class Buffer {
  private char[] buf;    // buffer storage
  private int last;      // last occupied position

  public Buffer(int sz) {
    buf = new char[sz];
    last = 0;
  }

  public boolean isFull() { return(last == buf.length); }

  public boolean isEmpty() { return (last == 0); }

```

Somewhere else, e.g. main

Buffer b= new Buffer(10);

Producer p= new Producer(b); b.start();

Consumer c=new Consumer(b);c.start();

Example: The Producer and Consumer Threads (3/4)

- The method that can put the **Producer** thread on the condition queue:

```
  public synchronized void put(char c) {
    while (isFull()) {
      try {
        wait();
      }
      catch(InterruptedException e) { }
    }
    buf[last++] = c; // We get here only if buffer not full.
                     // Have lock on buffer object if get
                     // here.
    notify(); // Like wait(), this is a message to the buffer
              // instance.
             //because a get may now be possible
  }
```

Example: The `Producer` and `Consumer` Threads (4/4)

- The method that can put the **Consumer** thread on the condition queue:

```
  public synchronized char get(){
    while (isEmpty()) {
      try {
        wait();
      }
      catch(InterruptedException e) { }
    }
    char c = buf[0];
    System.arraycopy(buf, 1, buf, 0, --last);
    notify();
    return c;
  }
}
```

Why Synchronized?

- Take the case where there is no synchronization and
  - *put()* is called when the buffer happens to be full and
  - a *get()* occurs **immediately** after the *put()* method has checked `isFull()`
- This consumer thread then get()'s an element from the queue, and notifies the waiting threads that the queue is no longer full.
  - Because the first thread has already checked the condition however, it will simply call wait() after being re-scheduled, even though it could make progress.

Writing a read write lock

- Summarizing the conditions for getting read and write access to the resource (for this example)
- Read access
  - » if no threads are writing, and no threads have requested write access
- Write access
  - » if no threads are reading or writing have requested write access
- Want an effective way to control access to the resource

Rationale

- A thread wants to read the resource, it is OK as long as no threads are writing to it, and no threads have requested write access to the resource.
  - By up-prioritizing write-access requests we **assume** that write requests are more important than read-requests.
  - Besides, if reads are what happens most often, and we did not up-prioritize writes, starvation could occur.
  - Threads requesting write access would be blocked until all readers had unlocked theReadWriteLock. If new threads were constantly granted read access the thread waiting for write access would remain blocked indefinitely, resulting in starvation.

Rationale

- **Therefore a thread can only be granted read access if no thread has currently locked** the ReadWriteLock for writing, **or requested it locked** for writing
- A thread that wants write access to the resource can be granted when no threads are reading nor writing to the resource.
  - » It doesn't matter how many threads have requested write access or in what sequence
    - – unless you want to guarantee fairness between threads requesting write access

A simplified read write lock

```
public class **ReadWriteLock**{
private int readers = 0;
private int writers = 0;
private int writeRequests = 0;

public synchronized void lockRead() throws
InterruptedException{ while(writers > 0 ||
writeRequests > 0){ wait(); } readers++; }
```

A simplified read write lock

```
public synchronized void unlockRead(){ readers--;
notifyAll(); }


public synchronized void lockWrite() throws
InterruptedException{ writeRequests++; while(readers
> 0 || writers > 0){ wait(); } writeRequests--;
writers++; }


 public synchronized void unlockWrite() throws
InterruptedException{ writers--; notifyAll(); } }
```

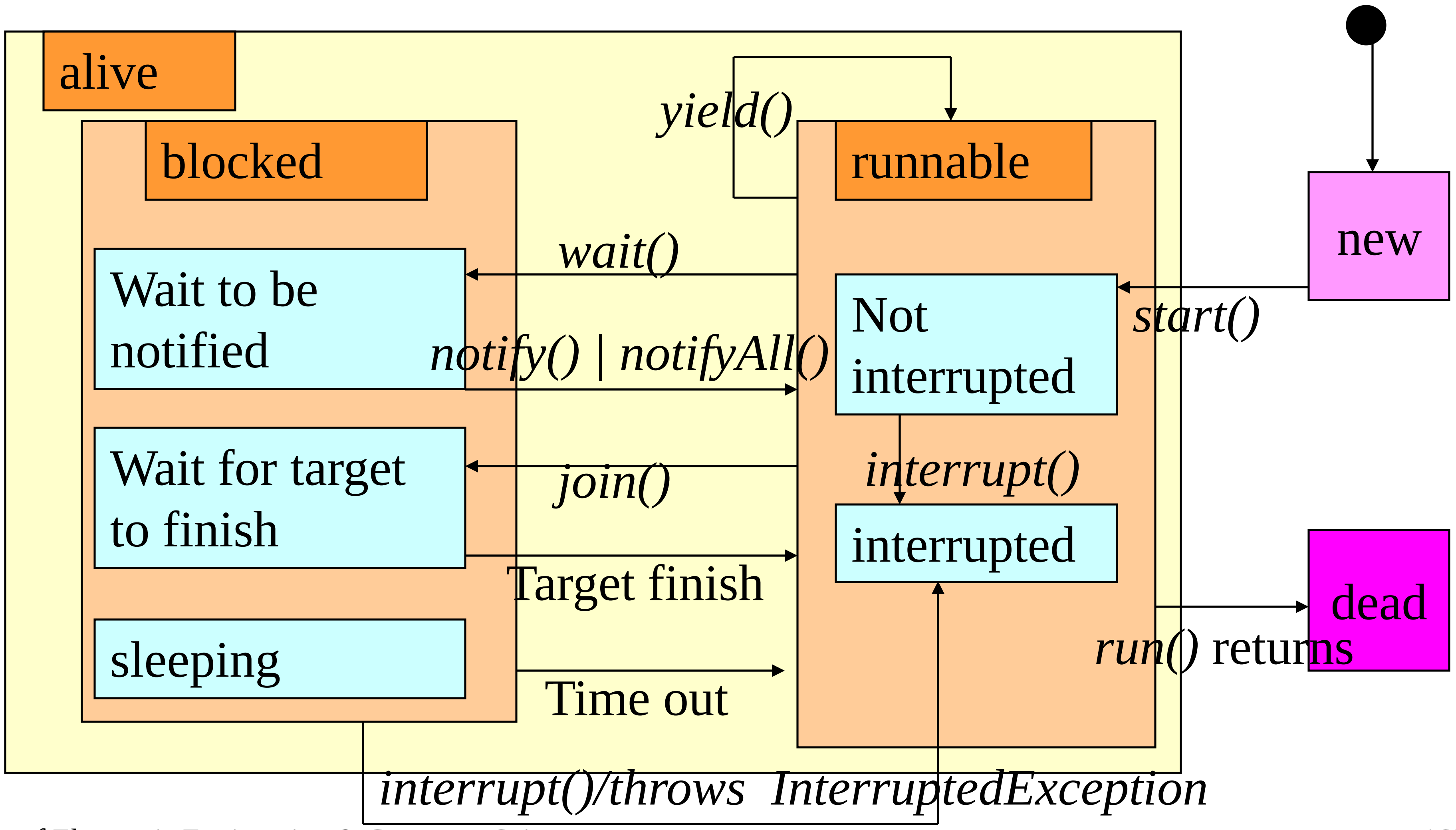
A simplified read write lock

Previous example taken from

http://tutorials.jenkov.com/java-concurrency/read-write-locks.html

They take the example further, bringing in more issues. A very good example.

Another Version of Life Cycle of a Thread



wait(long milliseconds [, int nanoseconds])

- For when you do not want to wait indefinitely for an event:
  - » if timed out, **`notify()`** is called automatically.
- Not told if **`wait()`** ended because condition became **`true`** or timed out (unfortunately!):
  - » need to recheck the wait condition and the system time to pinpoint the case.

Deadlock

- Two or more threads waiting for two or more locks to be freed, and the circumstances in the program is such that the locks will never be freed
- The synchronization primitives let you create deadlock
- Java provides no mechanisms to support deadlock prevention
- There are several design techniques (or patterns) to help you avoid them

Another example: Taking Turns

- We want each participant in a group to take turns in a fixed order
  - » Suitable for k player games

```
class Participant extends Thread{
  protected Participant next;
   // a reference to the Participant whose turn it is
  //  next fixed at set up for
  //  each participant instance
  protected Participant turn;
   // a reference to the Participant whose turn it is
   //  equals this if it is this participants turn
  public synchronized void run(){......}       }
```

We use each (whole) instance of a Participant as the lock.
Our run method needs to acquire the lock before it can run again (it can be notified after a wait). The run method only runs once, but is suspended and restarted by wait and notify.
There is another method *hasTurn* that needs exclusive access. Hence synchronization

Another example: Taking Turns

- Suppose there are n participants (instances) and they take turns in the order p_0, p_1, ….. $p_{(n-1)}$
- Create the n threads and initialise the next field of each participant as

  p_0.next= p_1

  p_1.next= p_2

  …………………..

  p_{n-1}.next= p_0,

To start the game we start all the participants

p_0.start();

p_1.start();

……………………..

p_{n-1}.start();

p_0.hasTurn() synchronized and makes turn=this

The whole run method is using the player instance as the lock (actually monitor)

```
public synchronized void run(){
while(!isDone()){
    //if it is not my turn wait till it is
    while(turn!=this){ try {wait();}
                       catch (InterruptedException ex){return;}
                 }
// it must be my turn now
// now perform an action…….      then
turn=null;  //it is not my turn now
next.hasTurn(); //tell the next participant it is its turn
}}
```

Initially for p_0 turn=this

Initially for p_1 !turn=this so waits

Next is the lock & it is available

next.hasTurn();

```
public synchronized void hasTurn(){
turn=this;
notify();
}
//other fields and methods e.g. isDone()
} // end of class definition
```

notifies one of the threads waiting on the lock of **next**
There is only one. So no problem