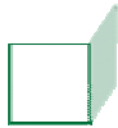




CHAPTER 2

THREADS



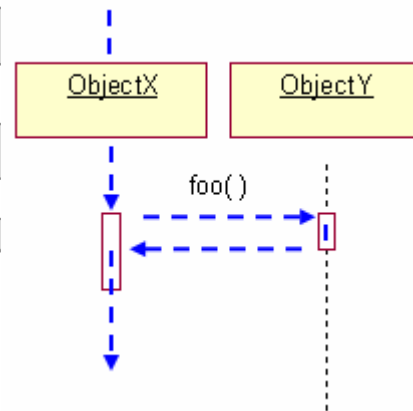
OBJECTIVES

After completing "Threads," you will be able to:

- Describe the role of threads of execution in a Java application or component.
- Use multiple threads in your Java code.
- Describe the organization of Java threads into thread-groups.
- Synchronize threads which may need to share one or more objects, so as to protect against concurrent access to and modification of those objects.
- Use **join**, **wait** and **notify** to further control the behavior of multiple threads in a JVM instance.
- Use the Concurrency API for easier task management, synchronization, and control of complex data structures.

A Virtual CPU

- A **thread** is variously described as a **unit of execution**, or perhaps most intuitively, as a **virtual CPU**.
- The thread is the abstraction by which we understand how processing is performed by software and hardware.
- A thread might best be described by enumerating its properties.
 - A thread has a starting point and a stopping point.
 - From its start it performs work as assigned; the means by which work is assigned boils down to a combination of sequences of processor instructions in memory and the behavior of a thread scheduler.
 - A thread can only be doing one thing at a time.
 - In most modern operating systems, threads can be **suspended**, which means put in a state in which they cannot do work, and later **resumed**, or put back in a runnable state.
- There can be many threads in a running **process** in most operating systems.

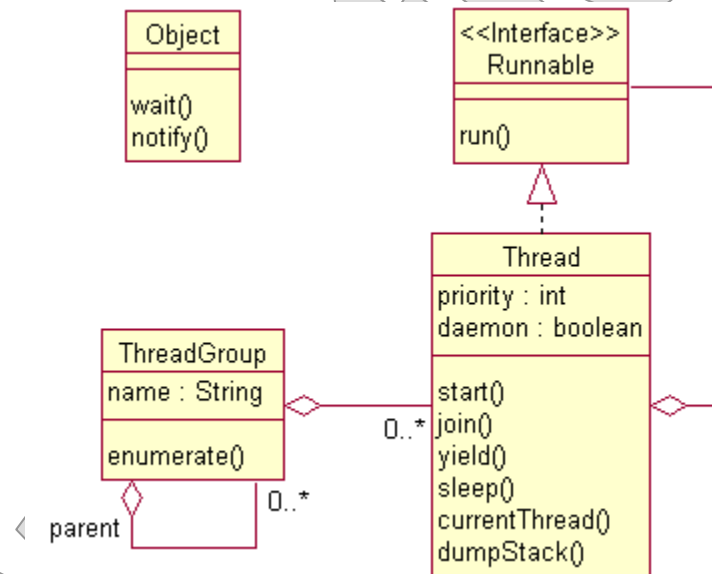


Threads – Who Needs ‘Em?

- Everything discussed in this course thus far assumes a single thread of execution.
- You never work **without** threads, but you can work with just one.
- The advantages of using multiple threads for an application can be many:
 - If there is a **user interface** that can be used to start time-consuming jobs, it is important to **avoid blocking the user** from continuing to interact with the interface; this is solved by using separate threads for the UI and the job.
 - Sometimes a more **complex task** can be optimized by using several **parallel threads**, especially when some subtasks require resources the use of which would cause a thread to block for a time.
 - In a **distributed context**, an application may well have to serve many clients, and therefore many client requests. If each request must be processed serially by a single thread **performance** will scale very poorly.

The Java Thread Model

- The Java Virtual Machine, which maps to a process in the native OS, manages threads, usually leveraging the thread model of the OS itself.
- The Core API provides a window to the JVM threads in the form of a class, **java.lang.Thread**.



- There is a one-to-one mapping between instances of the **Thread** class and actual threads modeled by the JVM.
- The JVM sets up a few threads for tasks that your code will never see, such as garbage collection.

Threads and Thread Groups

- The API also includes a **ThreadGroup** class, used to organize hierarchies of **Threads**.
 - A **ThreadGroup** can contain multiple **Threads** as well as other **ThreadGroups**.
 - Certain features of thread behavior and security attach to thread groups, making administration of a multiple-threaded application easier.
- **Threads themselves have lifecycles that run from being started through periods of activity and inactivity, and eventually they die.**
 - The **Thread** object outlives the JVM thread, and can be queried for status after thread death.
- **Threads can either be daemon threads or non-daemon threads.**
 - Mark this state with the **setDaemon** method.
 - The significance of this marker is that the JVM will terminate (as a process) when all **non-daemon** threads have terminated (unless **System.exit** is called).
- A thread also has a **priority** attribute which determines the frequency with which the thread scheduler will allow the thread to run.

Thread Priority

- The possible thread priorities vary from platform to platform, so they are captured in static fields on the **Thread** class:
 - **MIN_PRIORITY** and **MAX_PRIORITY** define the boundaries.
 - **NORM_PRIORITY** defines the default thread priority.
- When a thread of a higher priority than the current one enters a runnable state, the thread scheduler can **pre-empt** the running thread by suspending it and running the higher-priority thread.
- Threads of equal priority are treated differently on different platforms.
 - In particular, Windows platforms use a technique called **time slicing** which allows many threads of the same priority to be allotted bursts of time to run, in rotation.
 - Non-Windows systems for the most part do not do this, with the effect that one thread can keep another of the same priority waiting.
 - There is a method **Thread.yield** which you can call to assure that same-priority threads get a chance to run during a long stretch of processing.

Identifying the Current Thread

- Many threads may be used at various times to run the same span of code.
- You can get a reference to the thread on which your code is **currently** running using a static method on the **Thread** class:

```
Thread current = Thread.currentThread ();
```

- **Threads** do behave as ordinary Java objects.
 - You can reference and act upon them.
 - You can add them to collections.
 - After a **Thread** has died, you can check other attributes to determine what happened during its lifetime.
- The lifespan of the thread object exceeds that of the actual thread of execution.
 - You create a thread object, and later explicitly **start** it.
 - The object lives as long as references to it exist – normal garbage-collection rules – which means it may persist well after the thread of execution has died.

- In **Examples/ViewThreads** there is a simple application that shows the full group/thread hierarchy of the running JVM.

- It starts with the current thread and walks up to group, parent group, etc.:

```
ThreadGroup group =  
    Thread.currentThread ().getThreadGroup ();  
while (group.getParent () != null)  
    group = group.getParent ();
```

- A recursive method **showGroup** then produces an indented tree of threads and groups, using **showThread** for the leaf nodes of the tree (the threads themselves).

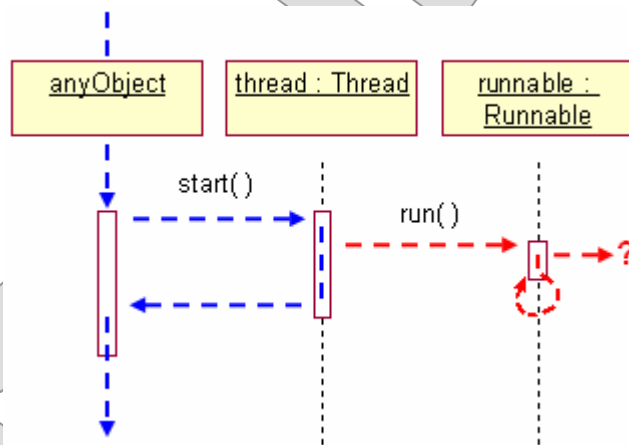
- **Build and test the application, and you will see the following output.**

```
build  
run  
  system  
    Reference Handler  
    Finalizer  
    Signal Dispatcher  
    main  
    * main
```

- Note that the current thread is marked with an asterisk; **showThread** accomplishes this by testing the passed thread object for identity with the current thread.

Spawning Threads

- To spawn a new thread in the JVM, you create a new instance of **Thread** and set it running.
- However, **Thread** itself doesn't know what to do when it's started; it needs a reference to a **Runnable** object, and we will look at how to develop one in a moment.



```
Thread worker = new
Thread (someRunnable);
worker.start ();
```

- You can manipulate the state of a thread once you've created it (or work on any thread once you have a reference to it).
 - Call **yield** to ask the thread scheduler to let other threads run.
 - Call **sleep** on the thread to suspend it for a finite time.
- Methods that forcibly **suspend**, **resume**, and **stop** another thread have been deprecated, as they are susceptible to deadlocks.
- The better practice uses flags that are observed by the thread object but made public so that outside actors can tell the thread to pause, resume, or stop.

Monitoring Thread State

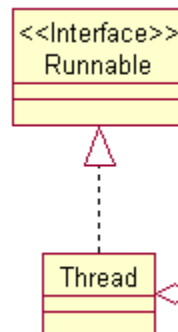
- You can check thread priority with **getPriority**.
- You can see whether or not a thread is somewhere between its **start** and **stop** points by calling **isAlive**.
 - “Alive” is not the same as “currently running.”
- The more general-purpose method is **getState**, which returns a value from the enumeration **Thread.State**:

```
enum State { NEW, RUNNABLE, BLOCKED, WAITING,  
            TIMED_WAITING, TERMINATED      };
```

- For diagnostic purposes, you can print a current stack trace for a thread using **dumpStack**.
 - Method-call stacks occur per thread of execution, and so a thread can produce a trace of this stack to a stream.
 - The exception stack traces we’ve seen here and there are examples of this. They reflect the state of the thread of execution on which the exception occurred.
 - Use **getAllStackTraces** as a way of checking the complete thread/stack status at a given moment.

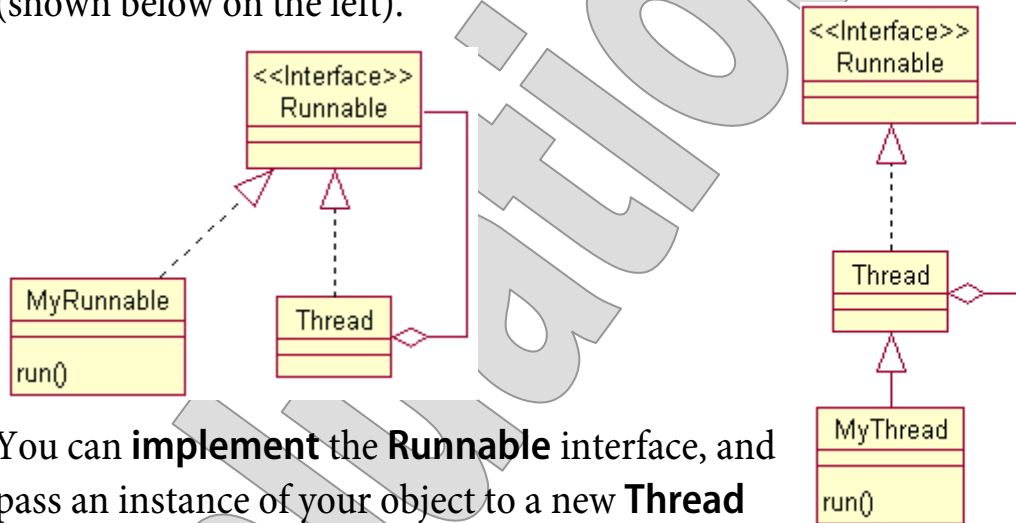
Defining Thread Behavior

- There are two means of defining what a given thread should do when it runs.
- When **Thread.start** is called, it in turn calls the **run** method on the **Runnable** object, and the code in this method determines the work of the thread.
- When (if) **run** returns, the thread dies.
- However, the **Thread** constructor has many overloads; in particular:
 - Constructors that take a **Runnable** object, as observed.
 - Also constructors that take no **Runnable** object reference – what then?
 - The trick is that **Thread** both **references** and **implements** **Runnable**, or in other words **is a Runnable** object.
 - Thus a thread can call its own **run** method instead of relying on a provided object reference.



Creating Thread Classes

- There are two approaches to defining thread behavior.
 - You can **extend** the **Thread** class to get everything in one place; then you can create a new instance of your class and call start on it (shown below on the left).

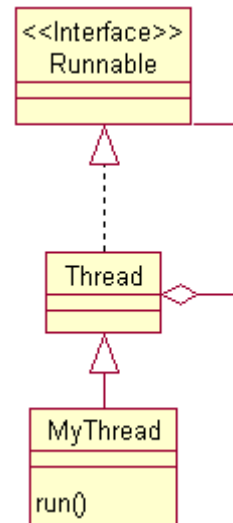


- You can **implement** the **Runnable** interface, and pass an instance of your object to a new **Thread** instance (shown at right).
- Neither approach builds in any limitations on the thread's behavior.
- However, in Java, you can only **extend** one other class, while you can **implement** as many interfaces as you like; thus in some cases extending **Thread** will not be an option, or not desirable.

Subclassing Thread

- Subclass **Thread** to create a new thread class that can function on its own.
- Here is a thread class that counts to twenty million before it dies:

```
public Counter
    extends Thread
{
    public void run ()
    {
        for (int x = 1; x <= 20000000; ++x)
        {
            System.out.println (" " + x);
            if (x % 10000 == 0)
                Thread.currentThread ().yield ();
        }
    }
}
```



- Use the thread like this:

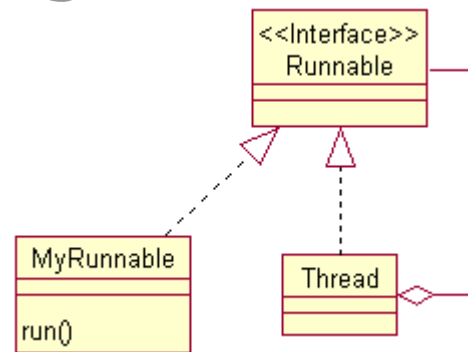
```
Thread counter = new Counter ();
counter.start ();
```

- Do not call **run**! That would be just like calling any other method on any other object – to wit, it would not cause the desired behavior to run on a different thread.
- Only the **start** method can do this. **start** calls **run**, but only after creating a new thread of execution in the JVM.

Implementing Runnable

- Implement **Runnable** to create a class that defines the work that a thread might do, but not the total behavior of a JVM thread.
- Here is a **Runnable** class that monitors the standard input stream for user interaction, perhaps while other threads are working on a lengthy chore:

```
public Monitor
    implements Runnable
{
    public void run ()
    {
        int key = System.in.read
        ();
        switch ((char) key)
        {
            case 'x':
            case 'X':
                return;
        }
    }
}
```



- Use the thread like this:

```
Runnable monitor = new Monitor ();
new Thread (monitor).start ();
```

- In **Demos/Search** we'll use a second application thread to allow the user to cancel a search for a specific filename.
 1. Review the application code in **src/cc/threads/Search.java**, which parses command line arguments and calls the static method **searchForFile**.
 2. This method recursively searches a given directory for a given filename, writing the full path of any matches to the console.
 - This example will not work perfectly in Eclipse.
 - It uses a trick to update the console output that doesn't work for the Eclipse Console view, resulting in verbose output.
 - Also the keyboard handling we're about to try will fail.
 - It will be better to code this in Eclipse, and test externally.
 3. Give it a quick try:

build

run Search.java

```
Searching for file: Search.java
```

```
From root directory:
```

```
C:\Capstone\JavaAdv\Examples\Search\Step1\.
```

```
Hit ENTER to cancel ...
```

```
Found C:\Capstone\JavaAdv\Examples\Search\Step1\  
src\cc\threads\Search.java
```

4. Now try it on a larger tree of files, and note that there is no good way to stop the process except to crash the process with **Ctrl-C**:

run Search.java c:

...

How could we give the user better control?

5. We'll define a status flag that will tell the running search code whether it should continue.

This can be set externally by a new thread that monitors user input.

6. First, we need to define a status flag that will tell the running search code whether it should continue.

Start by creating an enumerated type **Status**:

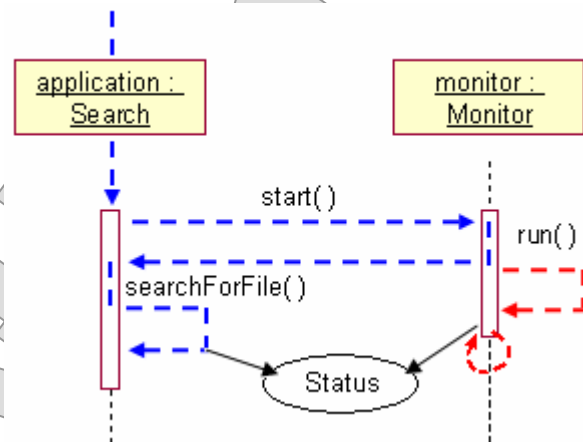
```
public enum Status { SEARCHING, CANCELED, DONE };
```

7. Define a static field of this type:

```
private static Status status;
```

8. In **searchForFile**, start each recursion by checking this flag, and if we are either **DONE** or **CANCELED**, quit:

```
public static File searchForFile  
    (String filename, File path)  
    throws IOException  
{  
    File candidate = null;  
  
    if (status != Status.SEARCHING)  
        return null;  
  
    System.out.print (CLEAR);  
    ...
```



9. In `main`, set the status to **SEARCHING** before calling `searchForFile`, and add a **finally** clause to the system that assures that the status is correctly set to **DONE**:

```
status = Status.SEARCHING;
try
{
    File found = searchForFile (filename, path);
}
catch (IOException ex)
{
    System.out.println ("IOException ... ");
}
finally
{
    status = Status.DONE;
}
```

10. This is all well and good, but how can the status change while the search is ongoing? Create a new inner class **Monitor** that **extends Thread**:

```
private static class Monitor
extends Thread
{
    public void run ()
    {
    }
}
```

11. Implement **run** to poll for keyboard input, so long as the status is still **SEARCHING**. When ENTER is hit, set the status to **CANCELED**:

```
public void run ()
{
    while (status == Status.SEARCHING)
        try
        {
            if (System.in.available () != 0)
            {
                int key = System.in.read ();
                status = Status.CANCELED;
            }
        }
        catch (IOException ex)
        {
        }
}
```

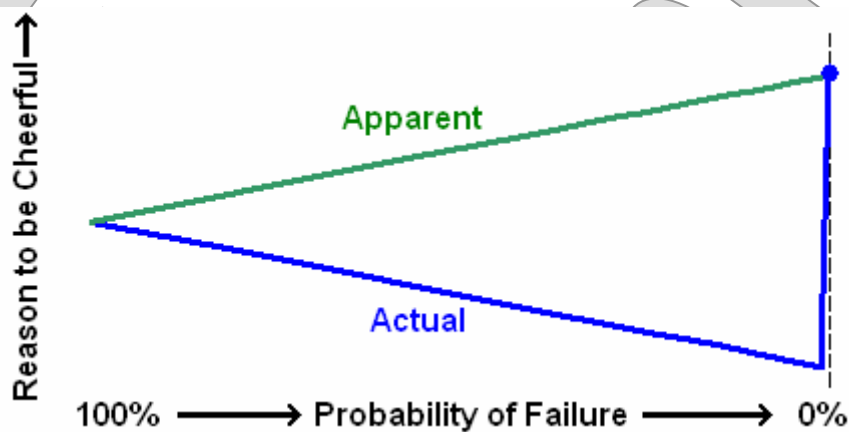
12. Create and **start** a new **Monitor** before calling **searchForFile**:

```
status = Status.SEARCHING;
new Monitor ().start ();
try
{
    File found = searchForFile (filename, path);
}
```

13. Build and test at this point. You should now be able to let the search run to completion, or to stop it by hitting ENTER.

Thread Synchronization

- It can be useful or essential to use multiple threads in a JVM to solve a programming problem.
- One they're there, however, multiple threads can ruin your life.
- If threads share data – primitive values or object references – there is a particular danger.
 - Whatever your thread-priority choices, it is possible that any given thread might be interrupted during processing.
 - This can happen at **any** point, even in the middle of what looks in source code like a single instruction.
 - If while one thread is **interrupted** another makes a state change on an object, the original thread might be resumed only to proceed with processing based on bad assumptions. This is a nasty sort of bug known as a **race condition**.
 - What's nasty about it? It is **intermittent**, and if anything the less likely it is to occur (short of zero probability), the worse the problem it poses:



Working with Single Values

- Most primitive values are managed **atomically** – that is, the operation of reading or writing their values cannot be meaningfully interrupted.
- Reads and writes of object references are also atomic.
- Still, some surprising effects can occur:
 - There is no guarantee of atomicity over **64-bit values**. Thus a read of a **long** could produce neither the starting value nor the ending value of a concurrent write!
 - The compiler can optimize method code in some ways that throw most assumptions of sequentiality out the window. It can **reorder** certain statements, seemingly arbitrarily, for the sake of performance.
 - It can **cache** a value in a local register if it sees the value as **non-volatile**; a loop boundary is a good example.
 - It can substitute a local variable that captures the results of an expression for that expression as used later in the code – this is called **forward substitution**.
- Believe it or not, even code like the following is susceptible to race conditions:

```
public foo ()
{
    int a = x;
    y = 1;
}
```

```
public bar ()
{
    int b = y;
    x = 2;
}
```

- Both `a == 2` and `b == 1` can be true in the same run!

volatile Fields

- This seemingly impossible result is enabled first by statement reordering, and then by interruption.

- The reordered code might be as follows; the compiler doesn't see a problem with this since within either method the two assignments are unrelated:

```
public foo ()           public bar ()
{
    y = 1;              {
    int a = x;          x = 2;
}                       int b = y;

```

- ... and then a simple interruption of one method by the other would give us this surprising result.

- One can avoid this and other compiler-generated surprises by declaring shared fields **volatile**:

```
private volatile int x, y;
```

- This informs the compiler that the field value may change at any time due to an external action.
- In response the compiler assures that
 - It will read the value of the field “fresh” each time it is referenced in an expression.
 - It will not reorder, substitute, or otherwise optimize based on an assumption of a stable value for the field.
- In practice this is of limited usefulness, and higher-level synchronization of code blocks makes it moot.

Working with Arrays

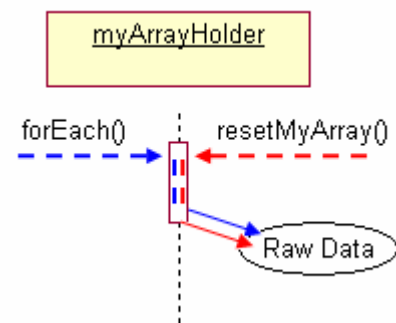
- Consider a method that iterates over the elements of an array:

```
public void forEach ()
{
    for (int i = 0; i < myArray.length; ++i)
        myArray[i].doSomething ();
}
```

- What if there is another method on the class that allows for the array field to be reset?

```
public void resetMyArray (SomeClass[] value)
{
    myArray = value;
}
```

- If one pre-empts the other, there will be trouble!
 - Thread one, for instance, might be in the midst of a call to **forEach**, maybe on the 12th array element, when interrupted.
 - Thread two, given the green light by the thread scheduler, calls **resetMyArray**, plugs in a whole new collection of only ten **SomeClass** instances, returns and is suspended.
 - Thread one resumes, attempting to **doSomething** on lucky 13, and things go very wrong.



Synchronizing Methods

- The problem described above can be solved by **synchronizing** the two methods.
- Synchronization is one of several common concurrency techniques – something like a critical section in general parlance – and relatively primitive.
- The **synchronized** keyword, used as a modifier on a method declaration, causes the JVM to enforce a locking policy on the object and method.
 - When any **synchronized** method runs on a given object, a lock is set on the object itself, on behalf of the running thread.
 - The lock is only reset when the method completes.
 - Should any other thread be used to make a call on another **synchronized** method on the **same object**, that call/thread will block and wait for the locking call/thread to complete.

```
public synchronized void forEach ()
{
    for (int i = 0; i < myArray.length; ++i)
        myArray[i].doSomething ();
}
```

```
public synchronized void resetMyArray
    (SomeClass[] value)
{
    myArray = value;
}
```

Synchronizing Code Blocks

- Using **synchronized** on a method declaration is actually a simple use of a more general capability.
- You can synchronize any block of code in Java.
- Further, when you define a block of code as **synchronized**, you can choose any object reference to be used as the host for the lock, not just the object on which a method is being called.
- This can be useful in building thread-safe code that minimizes the time in which it locks out multiple threads.
 - There is almost always a tension between thread safety and the overall performance of the system.

```
protected void fireEvent (int param)
{
    SomeEvent ev = new SomeEvent (param);

    Object[] localArray = null;
    synchronized (this) // just lock to copy array
    {
        localArray = memberVector.toArray ();
    }

    for (int i = 0; i < localArray.length; ++i)
        ((SomeListener) localArray[i]).occurred (ev);
}
```

In this lab you will enhance the threading code in the file-search application, to allow the user to suspend the search and then to choose to resume it or to cancel it altogether. This will require more than just a status flag and concurrent checks on status in the two threads: you will need to synchronize certain stretches of code to assure that they are waiting on each other.

Detailed instructions are contained in the Lab 2A write-up at the end of the chapter.

Suggested time: 30 minutes.

Evaluate Only

wait and notify

- For greater control over thread synchronization, you can use the **wait** and **notify** calls on an object.
- **wait** will suspend the calling thread until a **notify** call is made on the called object.
- Use **wait** to cause a thread to suspend until a condition is met.
 - Always call **wait** in a loop whose boundary condition is that which must be met before processing can proceed.
 - Do this because **wait** and **notify** are at bottom just optimizations; it is possible to awaken from a wait to find the condition unmet, and in that case you just **wait** again.
 - For instance you might **wait** for a count to come to zero, and **notify** every time the count were decremented.

```
while (resourcesBusy != 0)
    wait ();
```

- Always call **wait** and **notify** from a synchronized method or code block.
 - The object's **monitor** is the target of the **wait/notify** calls.
 - Running in **synchronized** code acquires ownership of the monitor, which is necessary for the **wait/notify** to work.

join and sleep

- You can make the current thread block until another completes (or dies) using the **join** call on that thread.
- If an application were to kick off two parallel threads to manage a computation, and wanted to print a report of the results, it would wait for completion:

```
Thread thread1 = new ComputingThread ();
Thread thread2 = new ComputingThread ();
thread1.start ();
thread2.start ();
try
{
    thread1.join ();
    thread2.join ();
    printReport ();
}
catch (InterruptedException ex) {}
```

- Make a thread suspend for a fixed amount of time by calling **sleep** on that Thread:

```
try
{
    Thread.currentThread.sleep (5000);
    // ~5000 milliseconds - not exact
}
catch (InterruptedException ex) {}
```

The Concurrency API

- Java 5 introduced a rich new **Concurrency API**.
- Implemented in package **java.util.concurrent** and subpackages, this API provides utilities for managing threads and the data they may share, at a higher level than the basic **java.lang** support for thread objects.
 - There are **synchronizers** such as a **Semaphore** class that facilitate coordination of complex tasks between threads.
 - **Thread-safe collections** implement the basic Collections API interfaces in ways that allow them to be safely shared.
 - **Atomic operations** over certain simple values can now be performed without complex locking logic.
 - **Thread pools** are available for parallel processing or for scalable request processing.
 - Various **queues** are implemented to facilitate sharing or piping of data between producer and consumer threads.
 - Many of the intermediate-level **locking objects** that make the above features possible are also available directly to application code, to build other, specific high-level abstractions.
- **Complete coverage of the Concurrency API is beyond our scope for this course.**
 - We will “touch down” in a few places and see some typical code that illustrates many of the above features.

Synchronizers

- Using **synchronized** blocks or methods is one straightforward way to keep multiple threads out of one another's hair.
- However it is just one of several common multi-threading techniques known to application programming in general.
 - Synchronized blocks closely mirror what the rest of the world calls **critical sections** of code.
 - Other languages or OS APIs offer other, more sophisticated tools, such as **semaphores**, **mutexes**, and **events**.
- Java 5 brought some of these tools into the Core API.
- For example, a **Semaphore** object can protect a resource or datum, and to allow some finite number of threads to access the protected thing concurrently.
 - Create the semaphore with some finite number of **permits**.
 - Code that wants to use the resource will first **acquire** the semaphore – an action which blocks while the semaphore is acquired by other threads equal to the number of permits.
 - When another thread **releases** the semaphore, a caller waiting to **acquire** it is allowed to continue.
- Don't leave it to the caller! Make sensitive resources private, and wrap access to them in code that assures the semaphore will be used appropriately.

Working with Collections

- All data sharing between threads poses concurrency issues, but collections are especially problematic:
 - They are pervasive in most complex application code.
 - They suffer many moments vulnerable to race conditions in common use – similar to what we saw for arrays earlier in this chapter.
- There are three main strategies for sharing collections of data between threads:
 - Synchronize access oneself, using techniques we've seen.
 - Derive **synchronized collections** using **Collections** utility methods such as **synchronizedList**.
 - Use **thread-safe collection** implementations provided in the Concurrency API.

Synchronized Collections

- The **Collections** class utility in **java.util** provides a set of methods for adapting an existing collection with a wrapper that assures total thread-safety.

```
static <T> Collection<T> synchronizedCollection
    (Collection<T> backingCollection);
static <T> List<T> synchronizedList
    (List<T> backingList);
static <K,V> Map<K,V> synchronizedMap
    (Map<K,V> backingMap);
static <T> Set<T> synchronizedSet
    (Set<T> backingSet);
static <K,V> SortedMap<K,V> synchronizedSortedMap
    (SortedMap<K,V> backingMap);
static <T> SortedSet<T> synchronizedSortedSet
    (SortedSet<T> backingSet);
```

- The reference derived by calling any of these methods can be safely shared between threads.
 - It is essential to provide this same “wrapped” collection to all users of the underlying collection; don’t mix and match use of the backing collection with the wrapper.
 - Also, any use of iterators derived from the synchronized collection must itself be synchronized on the collection itself, as in:

```
synchronized (mySafeCollection)
{
    for (String s : myStringList)
        buffer.append (s);
}
```

Thread-Safe Collections

- A few thread-safe collection implementations are provided; these perform well under certain assumptions, but are not recommended for the general case.
- The **CopyOnWriteArrayList<E>** and **CopyOnWriteArraySet<E>** address situations where reads are much more frequent than writes.
 - The strategy is to create a new copy of the underlying array on each write operation, and to use that as the data going forward.
 - Any threads currently reading information will complete their work on the old array of data, and therefore will be entirely safe from surprises brought on by that write operation.
 - Where writes are frequent this performs very poorly.
- The other safe collection types are all various sorts of **queues**.
 - The **first-in, first-out (FIFO) queue** is especially important in multi-threaded programming – this model supports passage of data and also task scheduling.
 - Does processing of the queue **block** under certain conditions, or can it always be processed concurrently?
 - Various implementations are available to support different strategies: examples are **ArrayBlockingQueue<E>** vs. **ConcurrentLinkedQueue<E>**.

Atomic Operations

- A common need in multi-threaded programming is to combine a few simple operations on a single value – such as a boolean or integer – into one atomic one.
- Java 5 introduced another feature long known to operating system APIs and other languages: the **atomic operation**.
- The classes in question – in package **java.util.concurrent.atomic** – encapsulate types of data to be manipulated atomically.
 - For instance **AtomicInteger** allows the caller to **get**, **set**, **compareAndSet**, **addAndGet**, **getAndIncrement**, **incrementAndGet**, and so on.
 - The available methods may seem like overkill, but remember that our idea of a minimal interface must be different here, since calling two atomic operations in sequence would not itself be an atomic operation!
 - These methods typically take advantage of similar features in the operating system, and so avoid the performance penalties of synchronization in the JVM.
 - **Booleans**, **integers**, and **long integers** are supported, as well as object references via **AtomicReference<V>**.
 - There are **atomic arrays** of integers, longs, and objects.
 - Utilities such as **AtomicIntegerFieldUpdater<V>** facilitate managing fields that are not themselves atomic objects.

Thread Pools

- For sheer performance or for scalability when implementing a server, a common programming problem is the **thread pool**.
- The Concurrency API provides high-quality thread-pool implementations that can be instantiated and put to work with a minimum of hassle.
 - Call one of the factory methods on `java.util.concurrent.Executor`: `newCachedThreadPool`, `newFixedThreadPool`, etc.
 - The returned object implements **ExecutorService**.
 - This system isolates the calling code from implementation details such as the strategy for scheduling tasks, creating and recycling threads, and so on.
 - Simply **execute** a given **Runnable** task on the service, and trust that it will be performed, efficiently, in the future.
 - You can monitor progress, wait for certain tasks to complete, shut down the pool, and wait for all tasks to complete after shutdown.
- Fine-grained management of tasks under a thread pool is achieved by use of different queue types.
- It is also possible to subclass the basic pool and queue implementations to take complete control of the scheduling and management policies.

- We'll look at three alternate implementations of our file-searching utility, and compare their threading strategies:
 - **Single-threaded**
 - Multi-threaded using a **thread pool** of recycled threads
 - Multi-threaded using **new threads** and not recycling them
- Each of these new examples has crude benchmarking code built in so we can compare their performance.
- In **Examples/Search/Step5**, the pre-existing single-threaded implementation is enhanced with just that benchmarking code.
 - To let the application run more efficiently, we've also removed the console output that shows status as what folder is being searched; now all we'll see are "hits" where the file has been found.
- Try this one on a moderately large tree, as a baseline:

```
build
run Search.java c:\Capstone
...
Found C:\Capstone\JavaAdv\Labs\Lab2A\src\cc\threads
  \Search.java
Time: 9911 msec
```

- Exact results will vary, of course – even from run to run.

- Now take a look at **Examples/Search/Step6**, and see that a thread pool has been put in place.

– See `src/cc/threads/Search.java`:

```
private static final int POOL_SIZE = 20;
private static ExecutorService threadPool =
    Executors.newFixedThreadPool (POOL_SIZE);
private static AtomicInteger workerCount =
    new AtomicInteger (0);
private static AtomicInteger pathsThatSpawned =
    new AtomicInteger (0);
private static AtomicInteger pathsThatDidNotSpawn
    = new AtomicInteger (0);
```

- Dividing a tree search up for purposes of parallel processing is not straightforward.
- But the approach shown here gives pretty good load-balancing, and trades some precision for efficiency since calculating the work per worker would cost more than we lose in the form of unequal loads.

- We take the opportunity to schedule new tasks on open threads as we drill down into child directories:

```
File[] children = path.listFiles ();
int count = children.length;
for (File child : children)
    if (child.isDirectory ())
    {
        if (--count == 0 ||
            workerCount.get () >= POOL_SIZE)
        {
            pathsThatDidNotSpawn.incrementAndGet ();
            searchForFile (filename, child);
        }
        else
        {
            pathsThatSpawned.incrementAndGet ();
            threadPool.submit
                (new Searcher (filename, child));
        }
    }
}
```

- The inner class **Searcher** is a **Runnable** that just calls **searchForFile**, continuing the recursion – but of course on a separate thread.

- But we cap that task-submission behavior, waiting for a worker to run out of sub-tree to process.
- Each worker increments a global count, works, and then decrements the count before dying. This lets a new worker tackle whatever part of the tree comes up for searching next:

```
public void run ()
{
    workerCount.incrementAndGet ();

    try
    {
        searchForFile (filename, path);
    }
    catch (IOException ex)
    {
        System.out.println
            ("IOException while searching.");
    }

    if (workerCount.decrementAndGet () == 0)
    {
        System.out.println ();
        ViewThreads.showAllThreads ();
        System.out.println ();

        threadPool.shutdown ();
    }
}
```

- When the worker count comes to zero, we know the search is completed, so we call **shutdown** on the thread pool – which otherwise would sit waiting for new tasks to run.

- The code that launches the process now does so on a thread – scheduling the first of many **Searcher** tasks – and then awaits the termination of the thread pool, as initiated by the **shutdown** call we just saw.

```
status = Status.SEARCHING;
new Monitor ().start ();
threadPool.submit
    (new Searcher (filename, path));
threadPool.awaitTermination
    (1, TimeUnit.HOURS);
status = Status.DONE;
```

- We also keep count of how many nodes in the tree submit new tasks to the thread pool vs. continue recursion on the current thread.

- Try this version out, and in most cases you'll see a significant speed advantage.
 - You'll also notice that we're dumping the thread tree – using old friend `src/cc/threadd/ViewThreads.java` – and you can see how the thread pool is organized.

build

run Search.java c:\Capstone

Using a pool of 20 threads.

...

Found C:\Capstone\JavaAdv\Labs\Lab2A\src\cc\threads
\Search.java

system

Reference Handler

Finalizer

Signal Dispatcher

Attach Listener

main

main

Thread-0

pool-1-thread-1

pool-1-thread-2

...

pool-1-thread-19

pool-1-thread-20

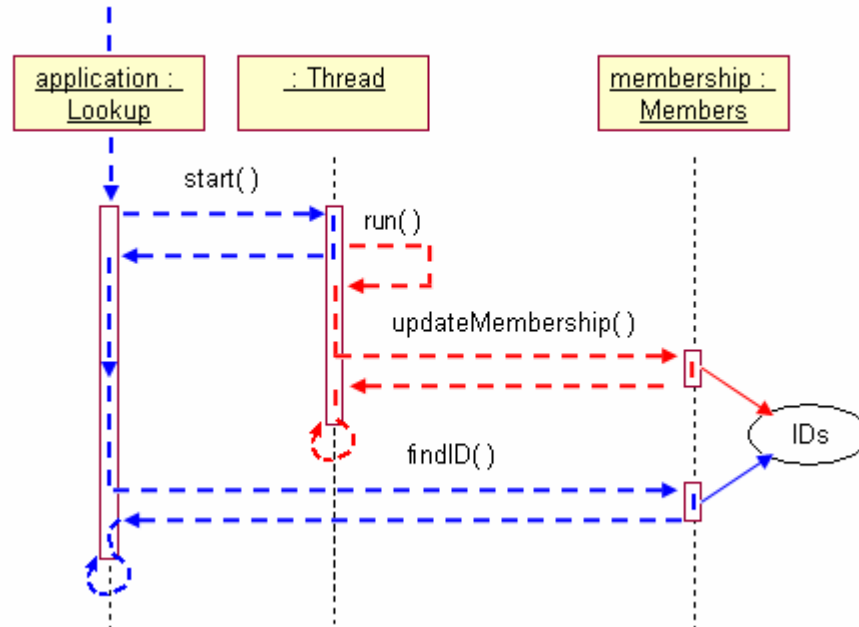
Time: 3528 msec

Paths that submitted new tasks: 195

Paths that didn't submit tasks: 1246

- We won't investigate **Examples/Search/Step7** quite so closely, but it may be worth a quick tour and a test.
 - We still keep a count and cap on the number of tasks that are active concurrently.
 - But now when we create a task it is run on its own new thread – no pooling.
 - When the work is done, the thread dies, and makes room for another to search elsewhere in the file system.
- **So the tradeoff, for performance purposes, is:**
 - No overhead costs for managing a fixed-size pool of threads and recycling them to new tasks.
 - Ongoing cost of creating new thread objects, and cleaning them up when they die.
- **If you test this version, you'll see little or no speed advantage one way or the other.**
 - Some testing with very large target trees has shown an increasing edge for a well-tuned thread pool.
 - But the real win of pools is usually in avoiding a thread-per-task policy when the task is long-running – such as managing an open-ended user or remote-client conversation.
- **You might also try tuning the size of the pool, in either version: try edge cases such as one or two threads, or much larger pools.**

In this lab you will observe race conditions present in a **Members** class. The class continuously varies the size of and values in a list of member IDs, to simulate the asynchronous activity of members joining and quitting. Another thread repeatedly asks for confirmation of a certain ID, thus setting the stage for race conditions as the list of IDs is concurrently written and read.



You will begin by testing out several subtly different implementations of the **findID** method, and observing their relative (apparent!) success. You will then fix the race condition in two different ways: using **synchronized** code blocks and wrapping the list in a **synchronizedList** provided by the **Collections** utility.

Detailed instructions are contained in the Lab 2B write-up at the end of the chapter.

Suggested time: 30 minutes.

SUMMARY

- Threads of execution play a fundamental role in a running Java Virtual Machine, whether your code explicitly uses and controls them or not.
- Threads in the Java threading model are organized into thread groups for administrative purposes.
- Threads as modeled by the Core API map to threads as modeled by the JVM, which usually means direct use of thread support in the operating system.
- Because of the need for portability, the Java thread model is pointedly simple, lacking more sophisticated synchronization primitives that are most likely found in many target platforms.
- Nevertheless, some synchronization capability exists, largely through the use of **synchronized** methods and code blocks, and of **wait**, **notify**, and **join** methods.