Threads

Topics

- Synchronous versus asynchronous
- Creating threads option 1
- Creating threads option 2
- Stopping threads
- Thread groups
- Daemon threads
- Thread safety
- Yielding and notifying threads
- Thread priorities
- Loose threads

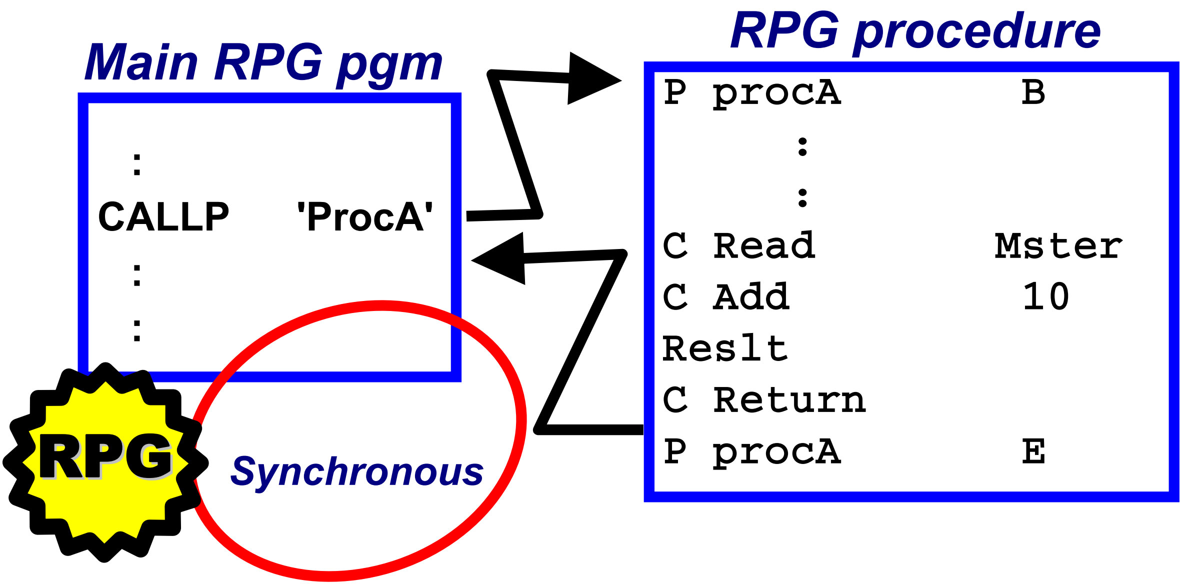
Unit objectives

After completing this unit, you should be able to:

- Describe Java’s support of threads
- Write Java code that creates, starts, stops, and manages threads
- Synchronize access to variables by multiple threads

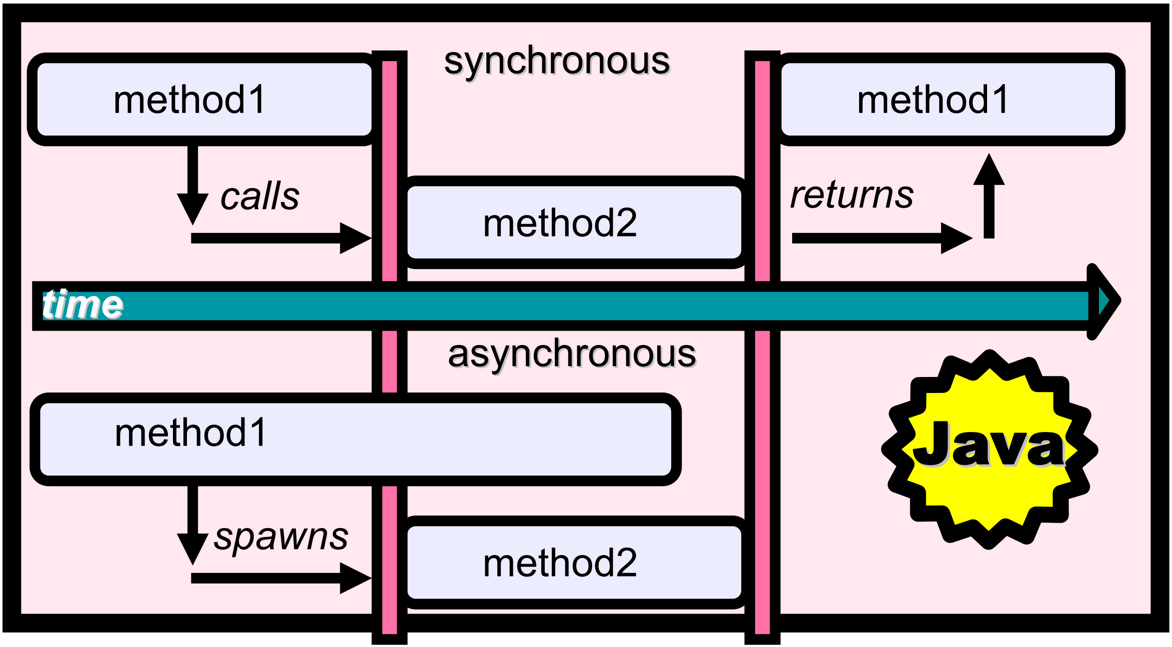
An introduction (1 of 2)

- Threads are foreign to an RPG programmer.
  - Why? Because RPG does not support them! Well maybe!!!
    - RPG "calls" are all synchronous, that is, control returns after the procedure, subroutine, or program has run:



An introduction (2 of 2)

- Threads are "spawned" asynchronously
  - Calling code and the called code run concurrently
  - Not supported in RPG, but it is in Java



Threads versus jobs

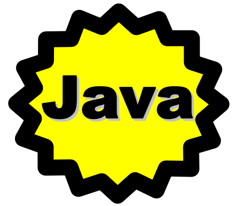
- Asynchronous execution is not totally foreign to AS/400
  - Submitting to batch is asynchronous (SBMJOB)
  - Jobs on AS/400 == processes on other platforms

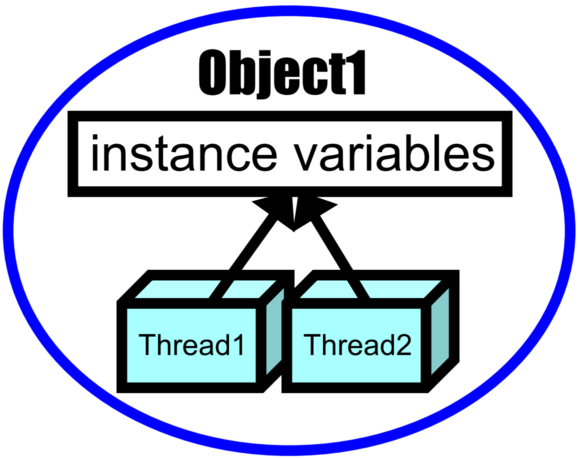
- Jobs are different from threads mainly in “overhead”!
  - With jobs you have to:
    - Allocate storage, set up library list, set up job attributes, resolve objects, load system resources, and so forth

- Threads have very little overhead
  - They share everything with the other threads in that job or process, including instance variables.

Threads and data

- A single method in a given object can be running multiple times concurrently.
  - Each in its own thread
  - Each thread gets its own local variables
    - But all threads share instance variables!





Thread questions

- Thread questions that will be covered:
  - How do you call a method asynchronously (that is, spawn a thread)?
  - How do you stop a thread?
  - How do you get back information from that thread?
  - If necessary, how do you wait for that thread to end?
  - How do you temporarily suspend a thread?
  - How do you change a thread's priority?
  - How do threads exchange information with each other?

Thread example

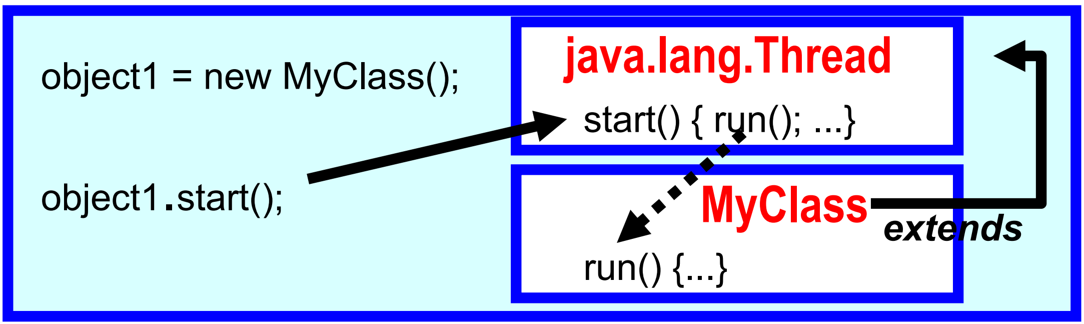
- Let's take the following print report method and convert it to run asynchronously...

```
public void printReport()
{
    // note: Printable objects[]; is an instance variable
    System.out.println("Printing objects...");
    for (int index = 0;
           index < objects.length;
           index = index + 1)
      objects[index].print();
    System.out.println(objects.length + " objects printed");
}
```

- Two ways to run a method asynchronously:
  – Extending `class Thread`
  – Implement `interface Runnable`

extend Thread (1 of 2)

- If the method to run asynchronously is part of a class *not* extending another class, use this:
- Add `extends Thread` to the class definition
  - Define method public void run()
    - Only this method runs asynchronously
    - Define it to call your method
- To run the code:
  - Create an instance of the class
  - Invoke the inherited start() method in it as follows:



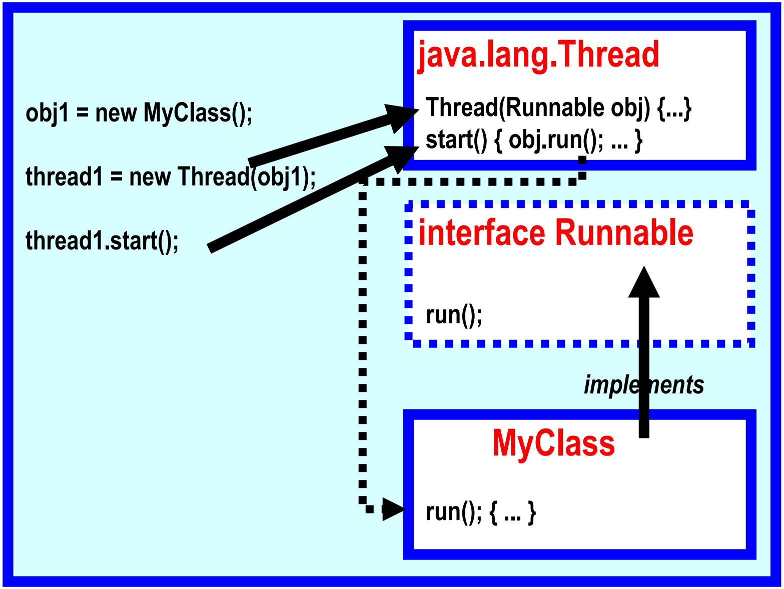
extend Thread (2 of 2)

```
public class TestBillableThread extends Thread
{
  ...
  public static void main(String[] args)
  {
    Printable[] objlist = new Printable[2];
    objlist[0] = new Presentation();
    objlist[1] = new Widget();
    // create instance of this class, run it
    TestBillableThread object1 =
       new TestBillableThread(objlist);
    object1.start();
  }
  public void printReport()
  {
    *** SAME ***
  }
  public void run()
  {
     printReport();
  }
}
```

Keep run() simple!
The method printReport() could
have been renamed to run()

implements Runnable (1 of 2)

- If you cannot extend your class, then you need to implement the Java supplied interface Runnable:
  - Add `implements Runnable` to the class definition
  - Define method public void run()



- To run the code:
  - Create an instance of the class
  - Invoke the start() method on it

implements Runnable (2 of 2)

```
public class TestBillableRunnable implements Runnable
{
    ...
    public static void main(String[] args) // cmdline entry
    {
      Printable[] objlist = new Printable[2];
      objlist[0] = new Presentation();
      objlist[1] = new Widget();
      TestBillableRunnable object1 =
            new TestBillableRunnable(objlist);
      Thread thread1 = new Thread(object1); // create thread object
      thread1.start(); // implicitly invoke run method of object1
    }

    public void printReport()
       *** SAME ***

    public void run()
    {
        printReport();
    }
}
```

run() method

Stopping a thread (1 of 3)

- Declare boolean instance variable in threaded class:
  - `boolean stop = false; // should thread stop now?`
- Constantly check variable in threaded method:
  - ```
    if (stop)
       ... // do cleanup, return from method
    ```
- In main thread, set stop to true when appropriate (for example, when user decides to stop it):

```
// start the thread running...
thisObject.start();
System.out.println("... press <Enter> to stop");
try {
 System.in.read(); // wait for <Enter> key
}
catch (java.io.IOException exc) {}
// <Enter> pressed - turn on "stop" flag
thisObject.stop = true; // better: thisObject.setStop(true);
```
```

Stopping a thread (2 of 3)

```
// Potentially long running method
public void run()
{
   for (int secs=0;
          (secs < seconds) && !stop;
          secs++)
     {
        try
        {
          sleep(1000); // sleep for one second
          System.out.println(secs + " seconds");
        }
        catch (InterruptedException exc) { }
     } // end for loop
   if (stop)
     System.out.println("... thread stopped");
} // end run method
```

```
>java TestThreads 30
Running... ... press <Enter> to stop
0 seconds
1 seconds
2 seconds
3 seconds
... thread stopped
```

Stopping a thread (3 of 3)

- Another alternative:
  - The stop() method found in the Thread class
  - Called by another thread, usually the main thread
- In the example, change...
  - `thisObject.stop = true;`

to...
  - `thisObject.stop();`
- More responsive than first alternative

Note: stop() is "deprecated" in JDK 1.2.0 or higher

```
>java TestThreads 30
Running... ... press <Enter> to stop
0 seconds
1 seconds
2 seconds
```

Cleaning up after stop

- Want to do clean up after terminating a thread?
  - stop() sends ThreadDeath exception you can monitor for:

```
public void runLong()
{
    try
    {
      for (int secs=0; (secs < seconds); secs++)
      {
         try
         {
            sleep(1000L); // sleep for one second
            System.out.println(secs + " seconds");
         } catch (InterruptedException exc) {}
      } // end for loop
    } catch (ThreadDeath exc)
    {
      System.out.println("... thread stopped");
      throw(exc);
    }
} // end runLong method
```

Put entire body in a try - catch

You re-throw ThreadDeath so thread continues to die as expected.

Thread groups

- In Internet and GUI programming, multiple simultaneous threads is common.
  - Java designers added support for *thread groups*.
- Create instance of ThreadGroup class:
  - `ThreadGroup groupObject = new ThreadGroup("longRunning");`
- To include a thread in a ThreadGroup, pass the thread group object as a parameter to the constructor:
  - `Thread threadObject = new Thread(groupObject, thisObject);`
  - `Thread threadObject2= new Thread(groupObject, thisObject);`
- To stop all threads in a thread group, use stop():
  - `groupObject.stop(); // stops all threads in this group`

Ending threads

- What happens when your main method ends?
  - Java queues its exit until all active threads end
  - Control does not return to console command line
  - You see your process still running in job log
- To force an exit … try---> `System.exit(0);`
  - Kills all still-running threads
  - Equivalent to SETON LR!
  - Convention: pass 0 for OK, pass 1 for error

```
public static void main(String args[])
{
    MyApp application = new MyApp();
    application.run();
    System.exit(0);
}
```

Daemon threads

- Some threads are intended to run forever until the application ends.
  - For example, polling for input on a queue
  - Java's garbage collector thread
  - Java's User Interface event listener thread

- These are true "background" threads
  - Java has special term: a daemon thread
    - Daemon threads are killed automatically when program ends
    - Do not require use of System.exit(0);
  - You can make a thread a daemon by using the setDaemon(true); method from the Thread class.

Thread safety (1 of 5)

- Consider class Inventory that manages inventory of stock
  - It has an instance variable called *onHand* that represents how much inventory is in stock.
  - It has a method called takeOrder(int howMany) that places an order, ensuring onHand does not go below zero, and adding to the inventory if it does.

- Imagine takeOrder is called as thread
  - Imagine taking one thousand orders, each spawning takeOrder as a thread
    - All one thousand orders may run simultaneously
    - All will be using the single instance of the object
    - They will "step on" each other! One thread might be looking at the *onHand* variable while the other is decrementing it.

Thread safety (2 of 5)

```
public class Inventory
{
  private static final int AMOUNT_INCREMENT = 2000;

  private int onHand = 5000; // amount in inventory
  public boolean stop = false; // stop whole thing
```

```
 // method to fulfill an order
public void takeOrder(int howMany)
{
    int old = onHand;
    String error = "";
    if (stop) // have we been stopped?
      return; //  exit now
    if (howMany > onHand)
    { // increase inventory
       addToInventory(howMany);
       error = "Order: "+howMany+", old: "+old+", new: "+onHand;
    }
    onHand = onHand - howMany; // actually take order
    if (onHand < 0) // inventory should never be negative...
    {
       System.out.println("Error - onHand less than zero! "+onHand);
       System.out.println(error);
       stop = true;
    }
} // end takeOrder method
```

Code 'exposed' by multiple concurrent threads ... need to be synchronized!

Thread safety (3 of 5)

```
    // method to increase inventory,
    // accounting for size of current order
    private void addToInventory(int howMany)
    {
        if (howMany > AMOUNT_INCREMENT)
          onHand +=(howMany-onHand)+1;
        else
          onHand += AMOUNT_INCREMENT;
    } // end addToInventory method

} // end Inventory class
```

Thread safety (4 of 5)

OrderThread

```
public class OrderThread implements Runnable
{
  Inventory inventoryObject; // passed in to us
  int       howMany; // how many items to order
  // constructor
  public OrderThread(Inventory inventoryObject, int howMany)
  {
      this.inventoryObject = inventoryObject;
      this.howMany = howMany;
  }
  // "run" method, called by using Start()
  // This method places the order for the given amount
  public void run()
  {
      inventoryObject.takeOrder(howMany);
  }
} // end OrderThread class
```

Thread safety (5 of 5)

main

```
public class TestInventory
{
  public static void main(String[] args)
  {
    Inventory inv = new Inventory();
    java.util.Random random = new java.util.Random();
    int idx;
    System.out.println("Running... ");
    for (idx = 0; (idx <= 1000) && !inv.stop; idx++)
    {
       int nextRandom = java.lang.Math.abs(random.nextInt());
       nextRandom = (nextRandom % 10000) + 1;
       OrderThread newOrder = new OrderThread(inv, nextRandom);
       Thread newThread = new Thread(newOrder);
       newThread.start();
    }
    if (inv.stop)
      System.out.println("...stopped at: " + idx);
    else
      System.out.println("...all orders placed.");
  }
} // end TestInventory class
```

Result

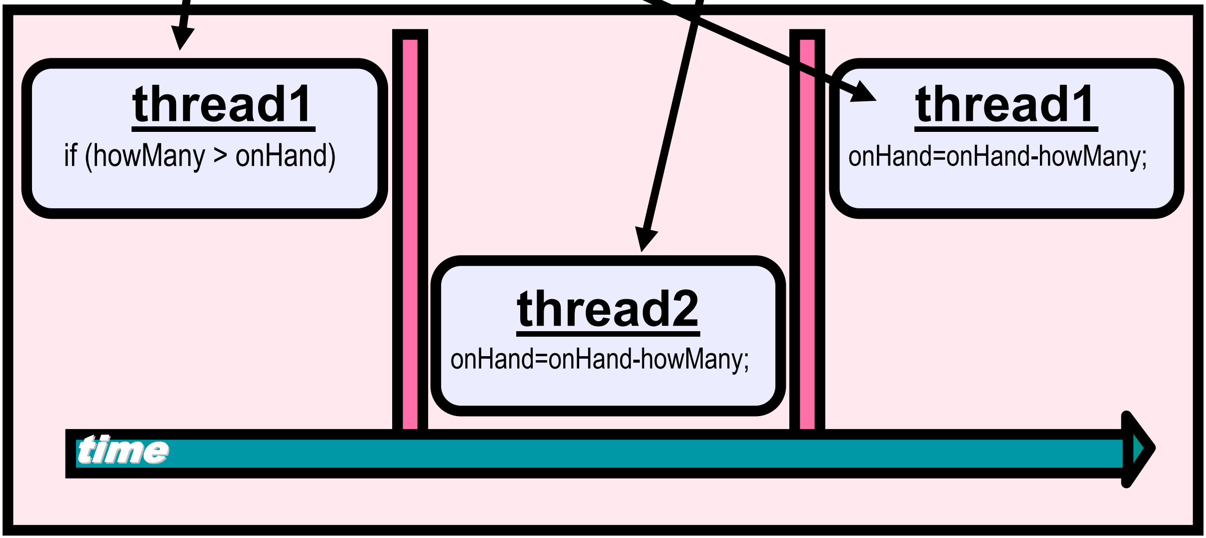
```
>java TestInventory
Running...
Error-onHand less than zero! -921
Order: 5820, old: 2001, new: 4899
Error-onHand less than zero! -1695
Error-onHand less than zero! -7353
Order: 5658, old: -921, new: -1695
Error-onHand less than zero! -7582
Order: 229, old: 1, new: 2001
...stopped at: 104
Order: 7354, old: 5821, new: 5659
```

Inventory fell below zero

Some threads still ran after "stop" set to true (past line of code that checks it)

Why?

- Between *checking* onHand and *changing* onHand, another thread is *changing* onHand



Thread safety

Solution

```
public synchronized void takeOrder(int howMany)
```

synchronize entire method

or

synchronize only sensitive code

```
synchronized(this)
{
  if (howMany > onHand)
    { // increase inventory
     addToInventory(howMany);
     error = "Order: "+howMany+", old: "+old+", new: "+onHand;
    }
  // actually take order
  onHand = onHand - howMany;
} // end synchronized(this)
```

Synchronization summary (1 of 2)

- Multiple threads running against the same object need to be *thread safe*:
  - The object needs to be "locked" for each thread to ensure that only one thread at a time accesses common variables.
- This is done using the `synchronized` modifier keyword on the threaded method:
  - `public synchronized void takeOrder(int onHand)`
- This locks the object containing the method while a thread is executing that method
  - Other threads are put in a "lock queue" to await their turn to run
- You can also synchronize individual lines of code by using a `synchronized(this) { ... }` block

Synchronization summary (2 of 2)

- Under V5R3 and RPG IV, program can be called in synchronized thread-safe manner

- Now under IBM i 6.1, multithread support is extended to RPG IV
  – Ability to run concurrently in multiple threads
    - Each thread has its own static storage for fields.
    - STATIC(ALLTHREAD) for fields is used by all threads.
    - Procedures can be serialized individually.

Yield and notify

- Sychronized methods can call yield() to *wait* for some event.
- Other methods or threads call notify() or notifyAll() to restart yielded threads.
  - Yield, notify, and notifyAll are in java.lang.Object class, so are available to all classes
  - notify lets the first yielded thread run.
  - notifyAll lets all of them potentially run.
- Useful in producer and consumer roles, such as DataQueues:
  - A consumer thread yields, waiting for an entry on the queue.
  - A producer thread puts an entry on the queue and notifies the consumer thread.

Thread priority

- Threads run simultaneously by using "time splicing"
  – Each thread is given a bit of time to run, then is put back on the queue
  – Which thread runs next is determined by how long it has been waiting, and its "priority"
  – The higher the priority, the more time a thread is given

- Set priority with setPriority() in Thread class

- Options are MIN_PRIORITY, NORM_PRIORITY and MAX_PRIORITY (constants in Thread class)
  – Default is NORM_PRIORITY
  – Give background threads MIN_PRIORITY
  – Give user interface threads MAX_PRIORITY

- Can also set priority for thread groups

Loose threads

- Other useful methods in Thread class:
  - yield() – gives up time splice so next thread can run
  - suspend() – temporarily stops this thread
  - resume() – restarts this suspended thread
  - join() – does not return until the thread has run to completion
  - isAlive() – returns true if method has not yet run to completion

Topics covered

- Synchronous versus asynchronous
- Creating threads option 1
- Creating threads option 2
- Stopping threads
- Thread groups
- Daemon threads
- Thread safety
- Yielding and notifying threads
- Thread priorities
- Loose threads

Unit summary

Having completed this unit, you should be able to:

- Understand Java's support of threads
- Write Java code that creates, starts, stops, and manages threads
- Synchronize access to variables by multiple threads