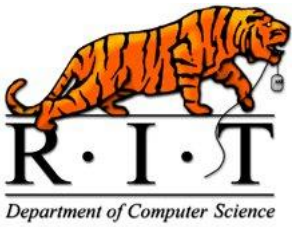
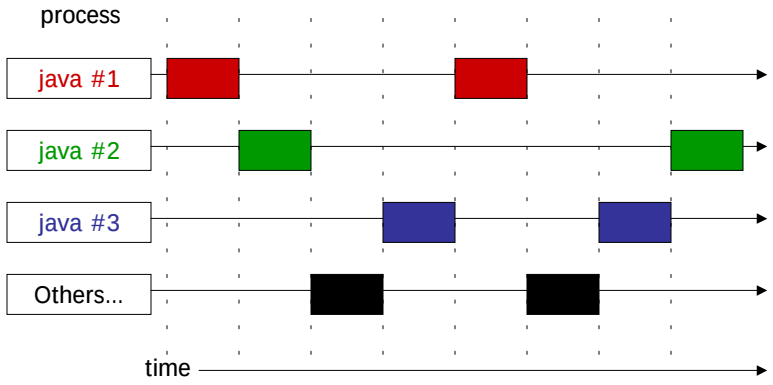


Multithreading

Sean P. Strout (sps@cs.rit.edu)
Rob Duncan (rwd@cs.rit.edu)

What is a Process?

- What happens when you run a Java program?
  - Launch 3 instances of MyProgram.java while monitoring with `top`
    - `See appendix A`
- On a single CPU architecture, the operating system manages how processes share CPU time

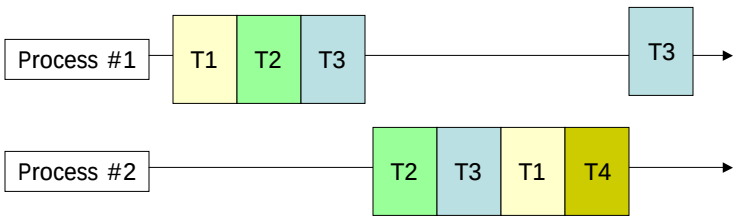


What is a Process?

- A **process** is a running instance of a program including all variables and the state of the system
  - The Java interpreter is the process which runs your program
- Besides running your program, the JVM must also do other tasks, like managing the memory your code uses (garbage collection)
- How does the JVM manage multiple tasks within a single process?
  - threads

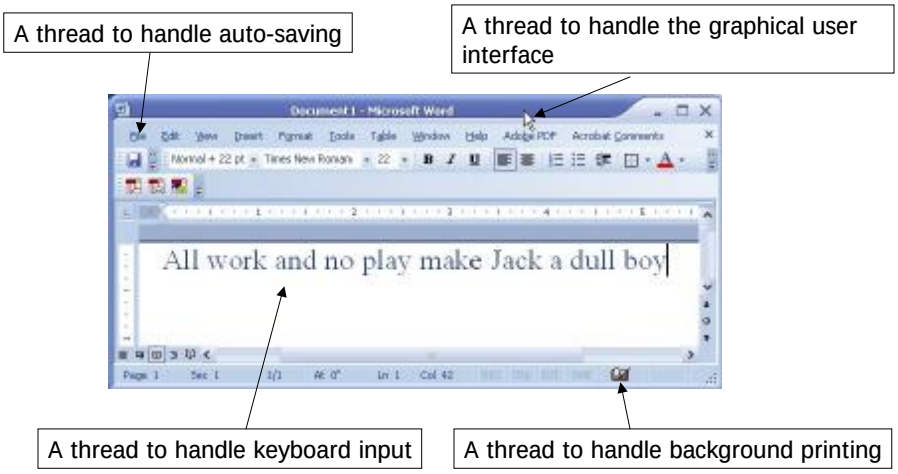
What is a Thread

- A **thread** is a flow of execution of a task in a program
- Java has built-in support for **multithreading**
  - Multiple tasks running concurrently within a single process
- Multiple processes/threads sharing CPU time:



Why Use Threads?

- Multithreading can make your program more responsive and interactive, and run faster than a non-threaded version



Java Threads

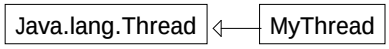
- When your program executes as an application, the JVM starts a thread for the `main()` method
- When your program runs as an applet, the web browser starts a thread to run the applet
- Each new thread is an object of a class that:
  1. Extends the `Thread` class –OR-
  2. Implements the `Runnable` interface
- The new object is referred to as a **runnable object**

When Execution Ends

- The Java Virtual Machine continues to execute threads until either of the following occurs:
  - The exit method of class `Runtime` has been called
  - All threads that are not *daemon* threads have died, either by returning from the call to the `run()` or by throwing an exception that propagates beyond `run()`.
- You cannot restart a dead thread, but you can access its state and behavior.

Creating Threads by Extending Thread

- A template for developing a custom thread class which extends the `Thread` class




```
// Custom thread class
public class MyThread
  extends Thread {
    public MyThread(…) {
        …
    }

    // Override the run method
    // in Thread
    public void run() {
        // Run the thread
        …
    }
}
```

```
// Client class
public class Client {
  public void method(…) {
      // Create thread1
      MyThread t1 = new
          MyThread(…);

      // Start thread1
      t1.start();

      // Create thread2
      MyThread t2 = new
          MyThread(…);

      // Start thread2
      t2.start();
  }
}
```

Creating Threads by Extending Thread

- Write a program that creates and runs three threads:
  - The first thread prints the letter a 5000 times
  - The second thread prints the letter b 5000 times
  - The third thread prints the integers 1 through 5000
- We'll make one thread class to handle the first two threads, `PrintChar`
- The third thread will be implemented by the `PrintNum` class

TestThread.java

```
/* Three independent threads:
 *
 * 1. Thread one prints the letter 'a' 5000 times.
 * 2. Thread two prints the letter 'b' 5000 times.
 * 3. Thread three prints the integers 1 through 5000.
 *
 */
public class TestThread {
	// main method
	public static void main(String args[]) {
		// Create threads
		PrintChar printA = new PrintChar('a', 5000);
		PrintChar printB = new PrintChar('b', 5000);
		PrintNum print100 = new PrintNum(5000);

		// Start threads
		printA.start();
		printB.start();
		print100.start();
	} // main
} // TestThread
```

TestThread.java

```
class PrintChar extends Thread {
	private char charToPrint;	// the character to print
	private int times;		// The times to repeat

	// Construct a thread with specified character and number
	// of times to print the character
	public PrintChar(char c, int t) {
		charToPrint = c;
		times = t;
	} // printChar

	// Override the run() method to tell the system
	// what the thread will do
	public void run() {
		for (int i=0; i<=times; i++) {
			System.out.print(" " + charToPrint);
		}
	} // run
} // PrintChar
```


TestThread.java

```
class PrintNum extends Thread {
	private int lastNum;		// the last number to print

	// Construct a thread with the last number
	public PrintNum(int n) {
		lastNum = n;
	} // PrintNum

	// Override the run() method to tell the system
	// what the thread will do
	public void run() {
		for (int i=0; i<=lastNum; i++) {
			System.out.print(" " + i);
		}
	} // run
} // PrintNum
```

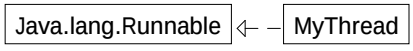
TestThread Output

Running TestThread.java reveals that each thread gets to "run" for a period of time before it gets interrupted – to allow another thread to run.

Eventually, all of the threads finish running (all output is done).

Creating Threads by Implementing Runnable

- A template for developing a custom thread class which implements the `Runnable` interface




```
// Custom thread class
public class MyThread
  implements Runnable {
    public MyThread(…) {
        …
    }

    // Override the run method
    // in Thread
    public void run() {
        // Run the thread
        …
    }
}
```

```
// Client class
public class Client {
  public void method(…) {
      // Create thread1
      Thread t1 = new Thread (
          new MyThread(…));

      // Start thread1
      t1.start();

      // Create thread2
      Thread t2 = new Thread (
          new MyThread(…));

      // Start thread2
      t2.start();
  }
}
```

Creating Threads by Implementing `Runnable`

- If your custom thread subclass already inherits from another superclass, you must implement the `Runnable` interface
  - The interface only requires the `run()` method to be defined
- The custom thread class gets wrapped into a `Thread` object when created

```
Thread t1 = new Thread(new PrintChar('a', 5000) );
Thread t2 = new Thread(new PrintChar('b', 5000) );
Thread t3 = new Thread ( new PrintInt(5000) );
```


TestRunnable.java

```
/* Three independent threads:
 *
 * 1. Thread one prints the letter 'a' 5000 times.
 * 2. Thread two prints the letter 'b' 5000 times.
 * 3. Thread three prints the integers 1 through 5000.
 *
 */

public class TestRunnable {

      // main method
      public static void main(String args[]) {
            // Create threads
            Thread printA = new Thread (new PrintChar('a', 5000));
            Thread printB = new Thread (new PrintChar('b', 5000));
            Thread print100 = new Thread (new PrintNum(5000));

            printA.start();
            printB.start();
            print100.start();

      } // main
} // TestRunnable
```

TestRunnable.java

```
class PrintChar implements Runnable {
	private char charToPrint;	// the character to print
	private int times;		// The times to repeat

	// Construct a thread with specified character and number
	// of times to print the character
	public PrintChar(char c, int t) {
		charToPrint = c;
		times = t;
	} // printChar

	// Override the run() method to tell the system
	// what the thread will do
	public void run() {
		for (int i=0; i<=times; i++) {
			System.out.print(" " + charToPrint);
		}
	} // run
} // PrintChar
```

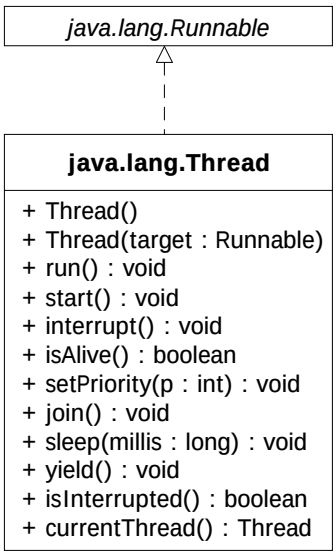

TestRunnable.java

```
class PrintNum implements Runnable {
	private int lastNum;		// the last number to print

	// Construct a thread with the last number
	public PrintNum(int n) {
		lastNum = n;
	} // PrintNum

	// Override the run() method to tell the system
	// what the thread will do
	public void run() {
		for (int i=0; i<=lastNum; i++) {
			System.out.print(" " + i);
		}
	} // run
} // PrintNum
```

TestRunnable Output

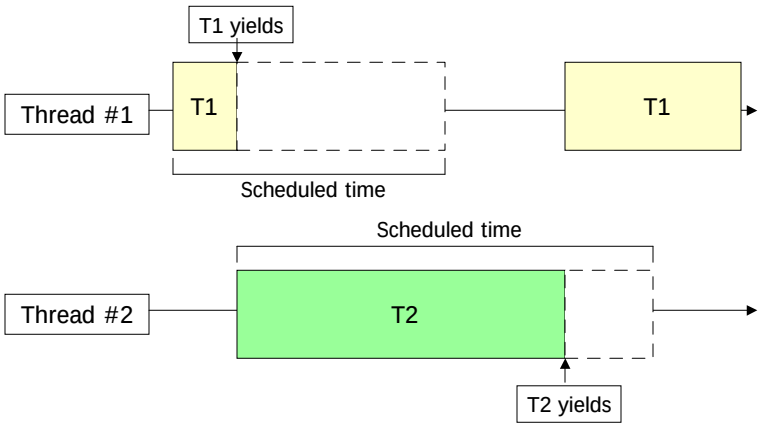
The output will be the same as TestThread.java. The only difference is the mechanism used to get the thread started. Once they are started, they are treated the same by the JVM.

Thread class



Thread Control - `yield`

- Use `yield()` to temporarily release CPU time for other threads



TestYield.java

```
/* Three independent threads which yield after each output statement
 *
 * 1. Thread one prints the letter 'a' 10 times.
 * 2. Thread two prints the letter 'b' 10 times.
 * 3. Thread three prints the integers 1 through 10.
 */
public class TestYield {
      // main method
      public static void main(String args[]) {

            // If there are cmd line args, the threads won't yield
            boolean yield = args.length > 0 ? false : true;

            // Create threads
            PrintChar printA = new PrintChar('a', 10, yield);
            PrintChar printB = new PrintChar('b', 10, yield);
            PrintNum print100 = new PrintNum(10, yield);

            // Start threads
            printA.start();
            printB.start();
            print100.start();
      } // main
} // TestYield
```

Department of Computer Science

TestYield.java

```
class PrintChar extends Thread {
	private char charToPrint;	// the character to print
	private int times;		// The times to repeat
	private boolean yield;		// Do I yield?
	// Construct a thread with specified character and number
	// of times to print the character
	public PrintChar(char c, int t, boolean y) {
		charToPrint = c;
		times = t;
		yield = y;
	} // printChar
	// Override the run() method to tell the system
	// what the thread will do
	public void run() {
		for (int i=0; i<=times; i++) {
			System.out.print(" " + charToPrint);

			// Let other threads run if told to
			if (yield) {
				Thread.yield();
			}
		}
	} // run
} // PrintChar
```


Department of Computer Science

TestYield.java

```
class PrintNum extends Thread {
	private int lastNum;		// the last number to print
	private boolean yield;		// Do I yield?

	// Construct a thread with the last number
	public PrintNum(int n, boolean y) {
		lastNum = n;
		yield = y;
	} // PrintNum

	// Override the run() method to tell the system
	// what the thread will do
	public void run() {
		for (int i=0; i<=lastNum; i++) {
			System.out.print(" " + i);

			// Let other threads run if told to
			if (yield) {
				Thread.yield();
			}
		}
	} // run
} // PrintNum
```

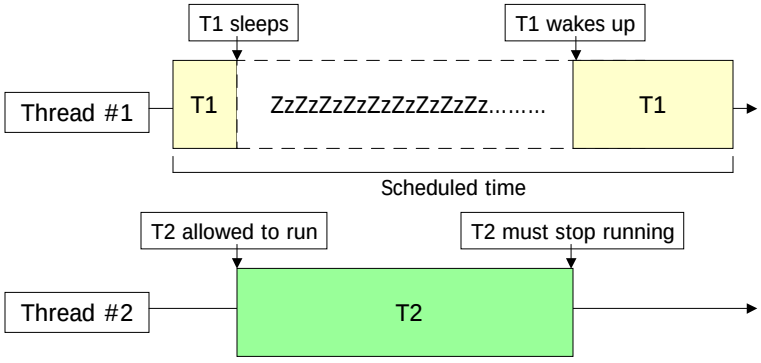
TestYield Output

OUTPUT:

```
% java TestYield false
 a a a a a a a a a a a b b b b b b b b b b b 0 1 2 3 4 5 6 7 8 9 10
% java TestYield
 a b b b b b b b b b b 0 1 2 3 4 a a a a a a a a a a 5 6 7 8 9 10
```


Thread Control - `sleep`

- `sleep()` causes the thread to cease execution for a specified number of milliseconds



Department of Computer Science

TestSleep.java

```
/* Threads 1 and 2 will sleep for a specified amount of time.
 *
 * 1. Thread one prints the letter 'a' 10 times.
 * 2. Thread two prints the letter 'b' 10 times.
 * 3. Thread three prints the integers 1 through 10.
 *
 */

public class TestSleep {

	// main method
	public static void main(String args[]) {

		// If there are cmd line args, the threads won't sleep
		int sleep = args.length == 0 ? 0 : Integer.parseInt(args[0]);

		// Create threads
		PrintChar printA = new PrintChar('a', 10, sleep);
		PrintChar printB = new PrintChar('b', 10, sleep);
		PrintNum print100 = new PrintNum(10);

		// Start threads
		printA.start();
		printB.start();
		print100.start();
	} // main
} // TestSleep
```


Department of Computer Science

TestSleep.java

```
class PrintChar extends Thread {
	private char charToPrint;	// the character to print
	private int times;		// The times to repeat
	private int sleep;		// amount to sleep
	// Construct a thread with specified character and number
	// of times to print the character
	public PrintChar(char c, int t, int s) {
		charToPrint = c;
		times = t;
		sleep = s;
	} // printChar
	// Override the run() method to tell the system what the thread will do
	public void run() {
		for (int i=0; i<=times; i++) {
			System.out.print(" " + charToPrint);

			// sleep for specified amount of time
			try {
				Thread.sleep(sleep);
			}
			catch (InterruptedException ex) {
			}
		}
	} // run
} // PrintChar
```

Department of Computer Science

TestSleep.java

```
class PrintNum extends Thread {
        private int lastNum;            // the last number to print

        // Construct a thread with the last number
        public PrintNum(int n) {
                lastNum = n;
        } // PrintNum

        // Override the run() method to tell the system
        // what the thread will do
        public void run() {
                for (int i=0; i<=lastNum; i++) {
                        System.out.print(" " + i);

                }
        } // run
} // PrintNum
```


Department of Computer Science

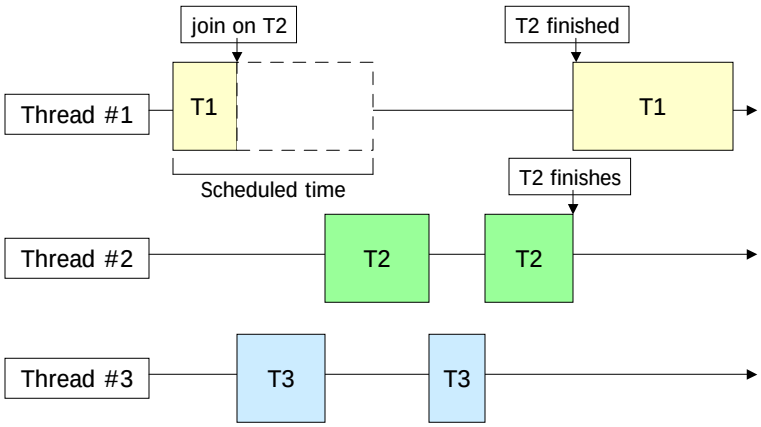
TestSleep Output

OUTPUT:

```
% java TestSleep
a b 0 1 b a a b 2 3 b a a b 4 5 b a a b 6 7 b a a b 8 9 b a a b
% java TestSleep 1
 a b 0 1 2 3 4 5 6 7 8 9 10 b a a b b a a b b a a b b a a b b a a b
% java TestSleep 10
 a b 0 1 2 3 4 5 6 7 8 9 10 a b a b a b a b a b a b a b a b a b a b
% java TestSleep 20
 a b 0 1 2 3 4 5 6 7 8 9 10 a b a b a b a b a b a b a b a b a b a b
```

Thread Control - `join`

- Use `join()` to force one thread to wait for another thread to finish



TestJoin.java

```
/* Three independent threads.  The third thread prints out half
 * its numbers and then waits for thread 2 to finish.
 * 1. Thread one prints the letter 'a' 200 times.
 * 2. Thread two prints the letter 'b' 200 times.
 * 3. Thread three prints the integers 1 through 200.
 */
public class TestJoin {
	private Thread printA;
	private Thread printB;
	private Thread printC;
	// Constructor makes the 3 threads and starts them
	public TestJoin() {
		// Create threads
		Thread printA = new Thread (new PrintChar('a', 200));
		Thread printB = new Thread (new PrintChar('b', 200));
		// pass in a reference to printB
		Thread print100 = new Thread (new PrintNum(200, printB));
		// start the threads
		printA.start();		printB.start();		print100.start();
	} // TestJoin
	// main method
	public static void main(String args[]) {
		TestJoin test = new TestJoin();
	} // main
} // TestJoin
```

Department of Computer Science

TestJoin.java

```
class PrintChar implements Runnable {
        private char charToPrint;       // the character to print
        private int times;              // The times to repeat

        // Construct a thread with specified character and number
        // of times to print the character
        public PrintChar(char c, int t) {
                charToPrint = c;
                times = t;
        } // printChar

        // Override the run() method to tell the system
        // what the thread will do
        public void run() {
                for (int i=0; i<=times; i++) {
                        System.out.print(" " + charToPrint);
                }
        } // run
} // PrintChar
```


Department of Computer Science

TestJoin.java

```
class PrintNum implements Runnable {
        private int lastNum;            // the last number to print
        private Thread waitOnThread;    // the thread to wait on

        // Construct a thread with the last number
        public PrintNum(int n, Thread  t) {
                lastNum = n;
                waitOnThread = t;
        } // PrintNum

        // Override the run() method to tell the system
        // what the thread will do
        public void run() {
                for (int i=0; i<=lastNum; i++) {
                        System.out.print(" " + i);
                        try {
                                if (i==lastNum/2) {
                                        waitOnThread.join();
                                }
                        } catch (InterruptedException ex) {
                        }
                }
        } // run
} // PrintNum
```

TestJoin Output

```
OUTPUT:
------

a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
   a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
   a a a a a a a a a a a a a a a a a a a a a a a a a a a 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
   15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
   42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
   69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
   96 97 98 99 100 b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b
   b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b
   b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b
   b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b
   b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b a a a a a a a a a a
   a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
   a a a a a a a a a a a a a a a a a a a a a a a a a a a 101 102 103 104 105 106 107 108
   109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
   128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146
   147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165
   166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184
   185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200
```

Thread Control – `wait()`, `notify()` and `notifyAll()`

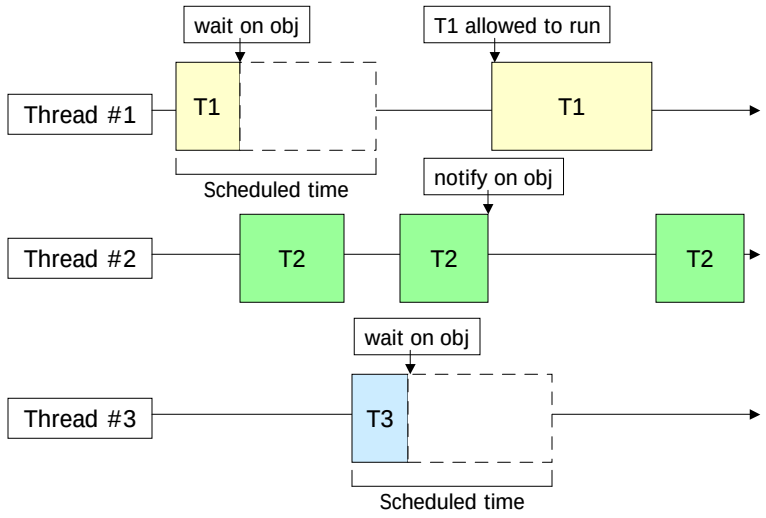
- There are three more methods defined in the `Object` class which are used to facilitate communication among active threads

- `public final void wait() throws InterruptedException`
  - Forces the thread to wait until the `notify` or `notifyAll` method is called for the object to which `wait` is called

- `public final void notify()`
  - Awakens one of the threads that are waiting on this object. Which one is notified depends on system implementation

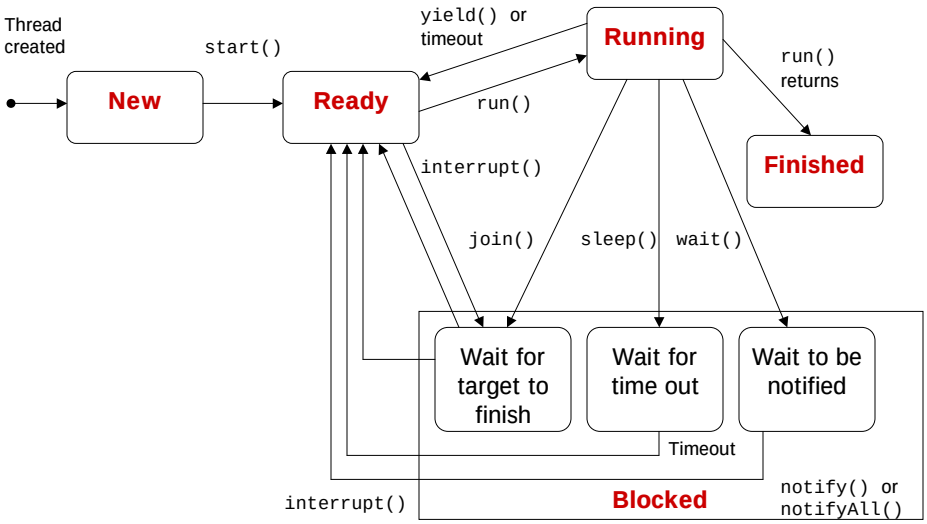
- `public final void notifyAll()`
  - Awakens all the threads that are waiting on this object

Thread Control – `wait()`, `notify()` and `notifyAll()`



Thread States

- Threads can be in one of five states



Thread States

- The Running state is the only time the CPU is executing the thread's code
- `isAlive()` returns `true` if the thread is in the Ready, Blocked or Running state
- `interrupt()` interrupts a thread in the following way:
  - If the thread is in the Ready or Running state, its interrupted flag is set and it's moved to the Blocked state
  - If the thread is currently Blocked, it is awakened and enters the Ready state, and a `java.lang.InterruptedException` is thrown

isAlive()

```
public class WorkerThread extends Thread {
  private int result = 0;

  public void run() {
    // Perform a complicated time consuming calculation
    // and store the answer in the variable result
  }

  public static void main(String args[]) {
    WorkerThread t = new WorkerThread();
    t.start();

    while ( t.isAlive() );

    System.out.println( result );
  }
}
```

What happens if this statement is left out?

- This solution works, but is there a better method?

Thread Priorities

- Java assigns every thread a priority. By default, a thread inherits the priority of the thread that spawned it
  - Mutator is `setPriority()` and accessor is `getPriority()`
- Priorities range from `1` to `10`
  - `Thread.MIN_PRIORITY = 1`
  - `Thread.NORM_PRIORITY = 5`
  - `Thread.MAX_PRIORITY = 10`
- A situation known as **contention** or **starving** occurs when higher priority threads don't yield and lower priority threads never get a chance to run

TestPriority.java

```
/* Three independent threads.  The third thread has the
 * highest priority and should finish first.
 *
 * 1. Thread one prints the letter 'a' 5000 times.
 * 2. Thread two prints the letter 'b' 5000 times.
 * 3. Thread three prints the integers 1 through 5000.
 */
public class TestPriority {
      public static void main(String args[]) {
            // Create threads
            PrintChar printA = new PrintChar('a', 5000);
            PrintChar printB = new PrintChar('b', 5000);
            PrintNum print100 = new PrintNum(5000);

            // Set priority on printA and printB to min
            printA.setPriority(Thread.MIN_PRIORITY);
            printB.setPriority(Thread.MIN_PRIORITY);

            // Set priority on print100 to the max
            print100.setPriority(Thread.MAX_PRIORITY);

            // Start threads.  Even though print100 is started
            // last it will finish first because of priorities.
            printA.start();
            printB.start();
            print100.start();
      } // main
} // TestPriority
```

Department of Computer Science

TestPriority.java

```
class PrintChar extends Thread {
	private char charToPrint;	// the character to print
	private int times;		// The times to repeat

	// Construct a thread with specified character and number
	// of times to print the character
	public PrintChar(char c, int t) {
		charToPrint = c;
		times = t;
	} // printChar

	// Override the run() method to tell the system
	// what the thread will do
	public void run() {
		for (int i=0; i<=times; i++) {
			System.out.print(" " + charToPrint);
		}
	} // run
} // PrintChar
```


Department of Computer Science

TestPriority.java

```
class PrintNum extends Thread {
	private int lastNum;		// the last number to print

	// Construct a thread with the last number
	public PrintNum(int n) {
		lastNum = n;
	} // PrintNum

	// Override the run() method to tell the system
	// what the thread will do
	public void run() {
		for (int i=0; i<=lastNum; i++) {
			System.out.print(" " + i);
		}
	} // run
} // PrintNum
```

TestPriority Output

The third thread completes before the other two.

Thread Groups

- A **thread group** is a set of threads with similar functionality on which you can perform the same operations
- Construct a thread group with a unique name:

```
ThreadGroup grp = new ThreadGroup("thread group");
```

- Using the Thread constructor, place it in the group:

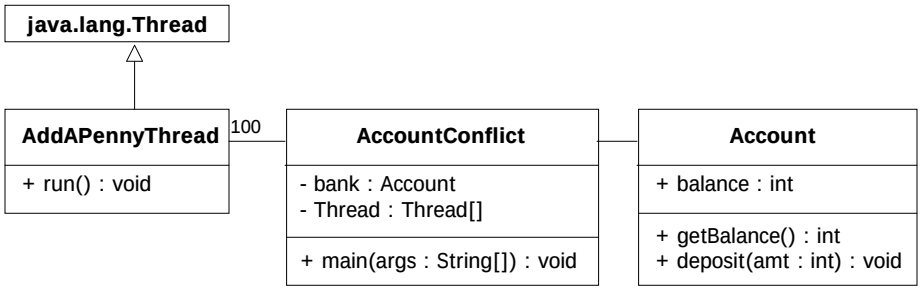
```
Thread t = new Thread(grp, new MyThread(), "label");
```

- Each thread must be started individually
- To find out how many threads in the group are running:

```
grp.activeCount();
```

Resource Conflict

- Write a program which launches 100 threads, each of which adds a penny to an account. Assume the account is initially empty.



AccountConflict.java

```
public class AccountConflict {
    private Account account = new Account();
    private Thread thread[] = new Thread[100];

    // Start the program and print out the balance at the end
    public static void main(String args[]) {
        AccountConflict test = new AccountSync();
        System.out.println("The balance is: " + test.account.getBalance());
    } // main

    // Constructor does the work of creating and launching the threads
    public AccountSync() {
        ThreadGroup grp = new ThreadGroup("account group");
        boolean done = false;

        // Create an launch 100 threads
        for (int i=0; i<100; i++) {
            thread[i] = new Thread(grp, new AddAPennyThread(),
                "thread" + i);
            thread[i].start();
        }
        // Wait for all the threads in the group to finish
        while (!done) {
            if (grp.activeCount() == 0) {
                done = true;
            }
        }
    } // AccountConflict
```

Department of Computer Science

AccountConflict.java

```
// Nested class for the threads - contains the run method
      class AddAPennyThread extends Thread {
            public void run() {
                  account.deposit(1);
            } // run
      } // AddAPennyThread

      // Nested class for the "resource".
      class Account {
            private int balance = 0;        // current balance
            public int getBalance() {
                  return balance;
            } // getBalance
            public void deposit(int amount) {
                  int newBalance = balance + amount;

                   balance = newBalance;
            } // deposit
      } // Account
}
```


Department of Computer Science

AccountConflict Output

OUTPUT:

The balance is: 42

Resource Conflict

- Consider this scenario with the previous example:

Step	balance	thread[0]	thread[1]
1	0	newBalance = balance+1	
2	0		newBalance = balance+1
3	1	balance = newBalance	
4	1		balance = newBalance

- Multiple threads can store the value of the common balance before it is updated individually
- This is known as a **race condition**
- A class is **thread-safe** if it does not cause a race condition in the presence of multiple threads

Critical Region

- To avoid race conditions, you must prevent more than one thread from accessing the **critical region** of a program

class Account:

```
public void deposit(int amount) {
     int newBalance = balance + amount;
     balance = newBalance;
}
```

- Only one thread should be allowed to enter the `deposit` method at a time

Synchronizing Instance Methods

- To **synchronize** an instance methods means to require a thread to obtain a lock on the object for which the method was invoked

class AddAPennyThread:

Synchronize on the `account` object

```
class AddAPennyThread extends Thread {
      public void run() {
            account.deposit(1);
      }
}
```

- To make `deposit` thread-safe in `Account`:

```
public synchronized void deposit(int amount) { … }
```


AccountSync.java

```
public class AccountSync {
      private Account account = new Account();
      private Thread thread[] = new Thread[100];

      // Start the program and print out the balance at the end
      public static void main(String args[]) {
            AccountSync test = new AccountSync();
            System.out.println("The balance is: " + test.account.getBalance());
      } // main

      // Constructor does the work of creating and launching the threads
      public AccountSync() {
            ThreadGroup grp = new ThreadGroup("account group");
            boolean done = false;

            // Create an launch 100 threads
            for (int i=0; i<100; i++) {
                  thread[i] = new Thread(grp, new AddAPennyThread(),
                        "thread" + i);
                  thread[i].start();
            }
            // Wait for all the threads in the group to finish
            while (!done) {
                  if (grp.activeCount() == 0) {
                        done = true;
                  }
            }
      } // AccountSync
```

Department of Computer Science

AccountSync.java

```
// Nested class for the threads - contains the run method
      class AddAPennyThread extends Thread {
            public void run() {
                  account.deposit(1);
            } // run
      } // AddAPennyThread

      // Nested class for the "resource".
      class Account {
            private int balance = 0;        // current balance
            public int getBalance() {
                  return balance;
            } // getBalance
            public synchronized void deposit(int amount) {
                  int newBalance = balance + amount;

                   balance = newBalance;
            } // deposit
      } // Account
}
```


Department of Computer Science

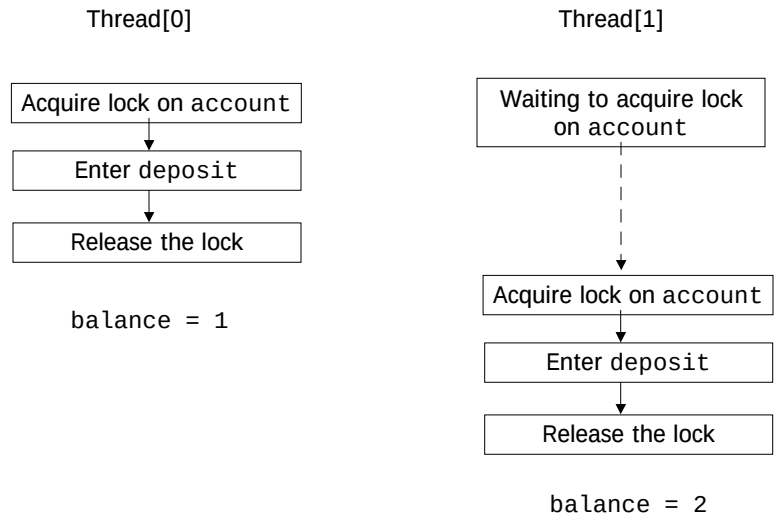
AccountSync Output

OUTPUT:

The balance is: 100

Synchronizing Instance Methods

- The preceding scenario with synchronization:



Synchronized Statements

- You could lock on a portion of a method using **synchronized statements**

class AddAPennyThread:

```
class AddAPennyThread extends Thread {
      public void run() {
            synchronized (account) {
                  account.deposit(1);
            }
      }
}
```

- This can increase concurrency and improve performance

Thread Cooperation

- Synchronization avoids race conditions by ensuring mutual exclusion to critical regions
- Threads also need a way to cooperate without requiring threads to finish
- The `Object` methods `wait()`, `notify()` and `notifyAll()` must be called in a synchronized method/block or else an `IllegalMonitorStateException` will occur

Thread Cooperation

- The template for coordinating threads:

Thread 1

```
synchronized (obj) {
   try {
      // Wait for condition
      while (!condition)
         obj.wait();

      // Do something…
   }
   catch InterruptedException ex) {
      ex.printStackTrace();
   }
}
```

Thread 2

```
synchronized (obj) {
   // Do stuff…

   // When condition is true
   obj.notify();
}
```

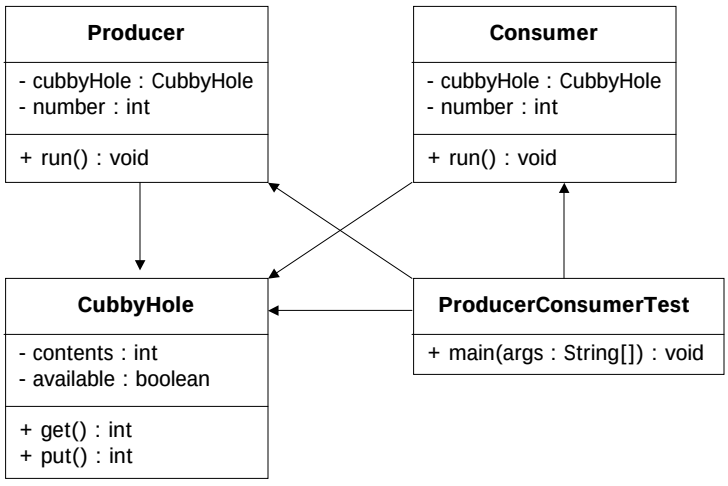
resume

Or `obj.notifyAll()` to wake up all threads

Producer/Consumer

- The **producer/consumer** problem is a classic operating system issue involving threads
- The idea is you have a producer which creates new resources and adds them to some collection
- The consumer takes elements out of the same collection, when available, and uses them
- We must be careful to design a solution which avoids:
  - Producer generating resources which the consumer misses or gets out of order
  - Consumer getting the same element more than once

Producer/Consumer



Department of Computer Science

Producer

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number
                               + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```


Department of Computer Science

Consumer

```
public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number
                               + " got: " + value);
        }
    }
}
```

Department of Computer Science

CubbyHole 1

```
public class CubbyHole {
    private int contents;
    private boolean available = false;

    public int get() {
        available = false;
        return contents;
    }

    public synchronized void put(int value) {
        contents = value;
        available = true;
    }
}
```


Department of Computer Science

Producer/Consumer Test

```
public class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
}
```

Producer/Consumer Output

```
OUTPUT:
Producer #1 put: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Producer #1 put: 1
Producer #1 put: 2
Producer #1 put: 3
Producer #1 put: 4
Producer #1 put: 5
Producer #1 put: 6
Producer #1 put: 7
Producer #1 put: 8
Producer #1 put: 9
```


CubbyHole 2

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

Deadlock

- Sometimes two or more threads need to acquire the locks on several shared objects
- **Deadlock** occurs when each thread has the lock on one of the objects and is waiting for the lock on the other

Thread 1

```
synchronized (obj1) {
   // do stuff…
   synchronized (obj2) {
      // do stuff…
   }
}
```

Waiting for thread 2 to release lock on obj2

Thread 2

```
synchronized (obj2) {
   // do stuff…
   synchronized (obj1) {
      // do stuff…
   }
}
```

Waiting for thread 1 to release lock on obj1

DeadLock.java

```
/* This is a demonstration of how NOT to write multi-threaded programs.
 * It is a program that purposely causes deadlock between two threads that
 * are both trying to acquire locks for the same two resources.
 * To avoid this sort of deadlock when locking multiple resources, all threads
 * should always acquire their locks in the same order.
 **/
public class Deadlock {
  public static void main(String[] args) {
    // These are the two resource objects we'll try to get locks for
    final Object resource1 = "resource1";
    final Object resource2 = "resource2";
    // Here's the first thread.  It tries to lock resource1 then resource2
    Thread t1 = new Thread() {
      public void run() {
        // Lock resource 1
        synchronized(resource1) {
          System.out.println("Thread 1: locked resource 1");
          // Pause for a bit, simulating some file I/O or something.
          // Basically, we just want to give the other thread a chance to
          // run.  Threads and deadlock are asynchronous things, but we're
          // trying to force deadlock to happen here...
          try { Thread.sleep(50); } catch (InterruptedException e) {}
          // Now wait 'till we can get a lock on resource 2
          synchronized(resource2) {
            System.out.println("Thread 1: locked resource 2");
          }
        }
      }
    };
```

Department of Computer Science

Deadlock.java

```
// Here's the second thread.  It tries to lock resource2 then resource1
   Thread t2 = new Thread() {
     public void run() {
       // This thread locks resource 2 right away
       synchronized(resource2) {
         System.out.println("Thread 2: locked resource 2");

         // Then it pauses, for the same reason as the first thread does
         try { Thread.sleep(50); } catch (InterruptedException e) {}

         // Then it tries to lock resource1.  But wait!  Thread 1 locked
         // resource1, and won't release it 'till it gets a lock on
         // resource2.  This thread holds the lock on resource2, and won't
         // release it 'till it gets resource1.  We're at an impasse. Neither
         // thread can run, and the program freezes up.
         synchronized(resource1) {
           System.out.println("Thread 2: locked resource 1");
         }
       }
     }
   };

   // Start the two threads. If all goes as planned, deadlock will occur,
   // and the program will never exit.
   t1.start();
   t2.start();
 }
}
```


Department of Computer Science

Deadlock

- To fix deadlock, use the technique known as **resource ordering**
- Assign an order on all the objects whose locks must be acquired and assure each thread acquires the lock in that order

Thread 1

```
synchronized (obj1) {
   // do stuff…
   synchronized (obj2) {
      // do stuff…
   }
}
```

Thread 2

```
synchronized (obj1) {
   // do stuff…
   synchronized (obj2) {
      // do stuff…
   }
}
```

Review - Java's Monitor

- The mechanism that Java uses to support synchronization is often called the **monitor**
- The two types of thread synchronization are:
  - Mutual exclusion using `synchronized` keeps threads from interfering with one another when sharing data
  - Coooperation via `wait` and `notify` allows threads to safely share data to achieve a common goal
- A **monitor region** is like a room with some data that only one thread is allowed into at a time

Scheduling Implementations

- Scheduling is typically either:
  - non-preemptive
  - preemptive
- Most Java implementations use preemptive scheduling.
  - the type of scheduler will depend on the JVM that you use.
  - In a non-preemptive scheduler a thread leaves the running state only when it is ready to do so.
- What does this mean for your applications?

Appendix A

```
public class MyProgram {
        public static void main(String args[]) {
                int i=0;
                while (true) {
                        i = i + 1;
                }
        }
}

% java MyProgram &
% java MyProgram &
% java MyProgram &
% top
```