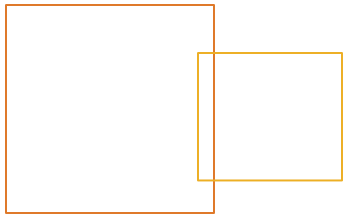
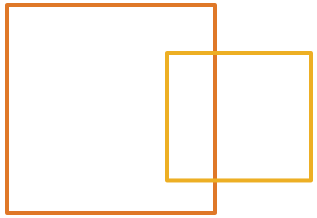
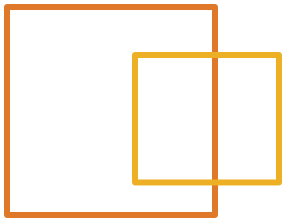
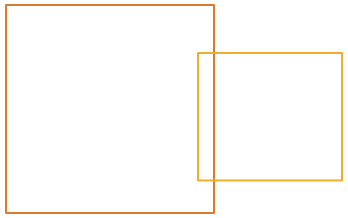
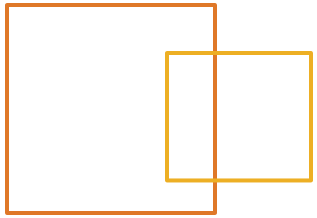
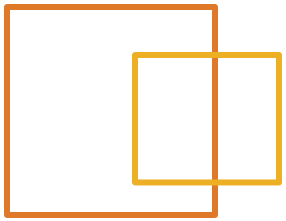





Fast Track to Java

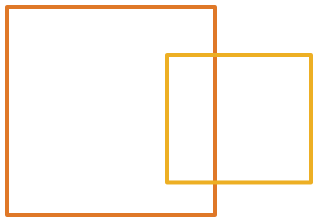
Customized for Starbucks

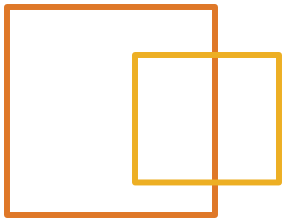
Delivered by DevelopIntelligence


Multi-Threading

Objectives

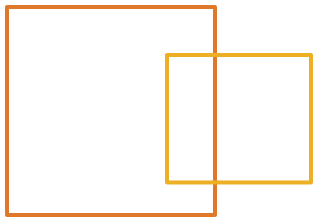


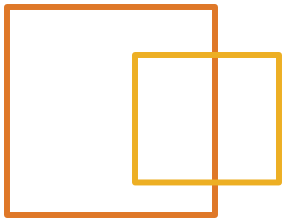


At the end of this module you should be able to:

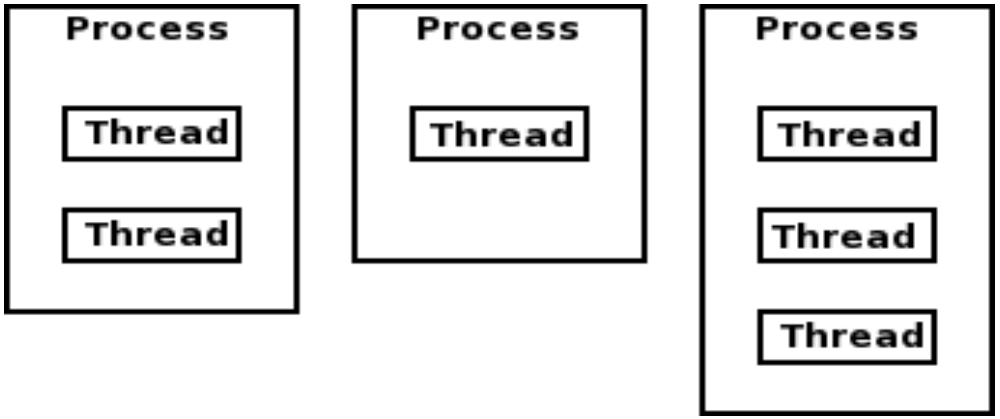
- Understand concurrency and threads, and why they are used
- Create and run a thread using the Thread class
- Understand thread priorities and scheduling
- Describe what a daemon thread is
- Use the Runnable interface
- Use thread synchronization
- Understand how threads are coordinated

Threads

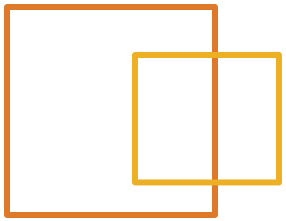




- Process: A flow of control running with its own address space, stack, etc.
- Thread: A flow of control sharing an address space with another thread, but with its own stack, registers, etc.
- Concurrency: Processes that are running at the same time



Threads in the Language

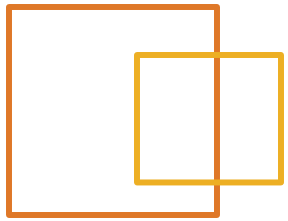


- Threading is built into JavaThere two primary classes used in threaded programming
  - `java.lang.Thread`
  - Class that represents a thread
  - Two main lifecycle methods
    - `start` – prepares the thread for execution
    - `run` – main body of execution
  - In this class, `run` has no useful implementation
  - `java.lang.Runnable`
  - Interface for creating a body of execution for a thread
  - Has one method – `run`
  - Threads delegate execution to `Runnable`'s `run`

Threads in the Language (cont.)

- There are two ways to utilizes threads
- extend `java.lang.Thread`
  - Over-ride `run` method to do something useful
  - Consider supplying application specific constructors
  - Once a `Thread` instance finishes, it can not be reused
- implement `java.lang.Runnable`
  - Any `Runnable` can the code for execution by a `Thread`
  - More flexible
  - A `Runnable` instance can be reused—even concurrently

Threads in the Platform

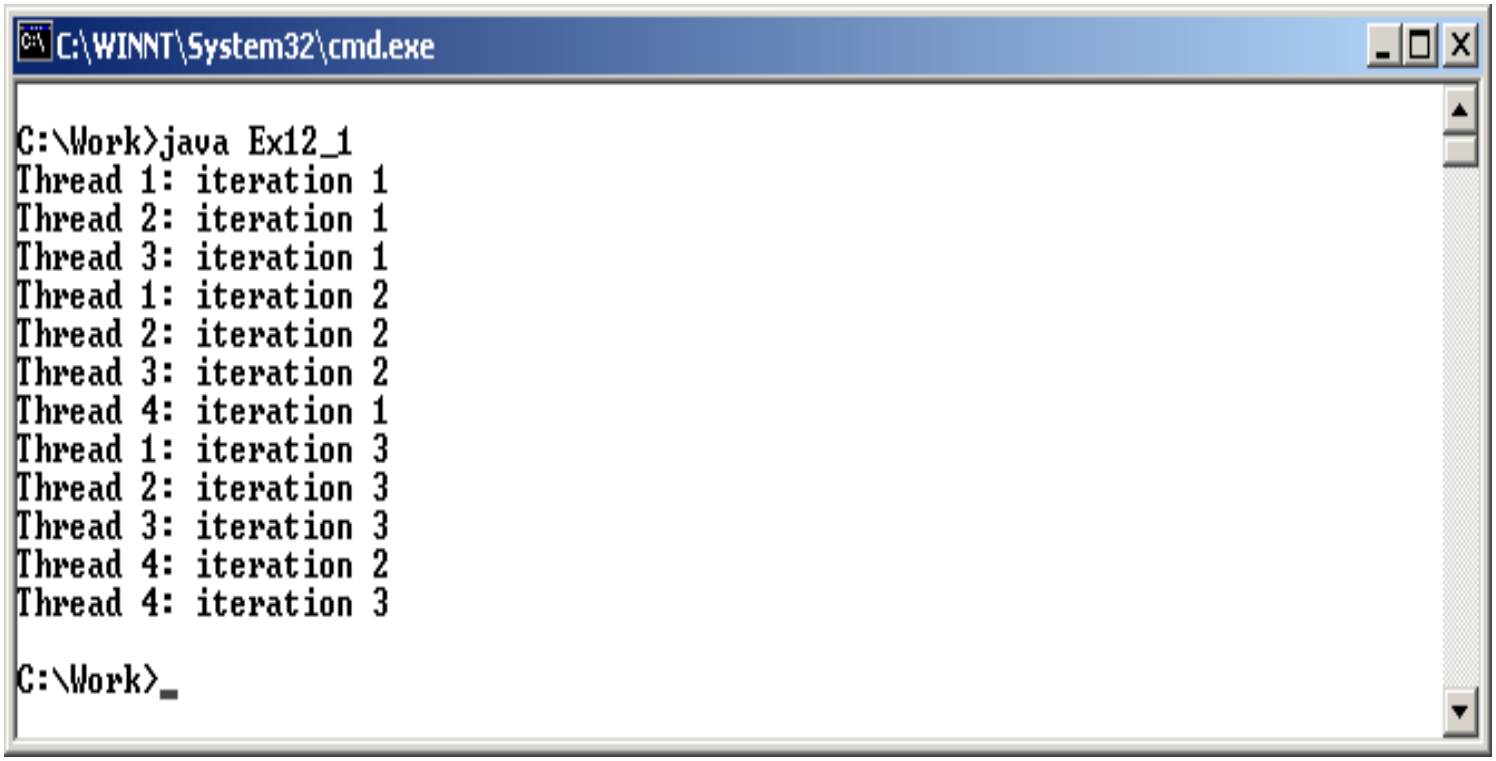


- The virtual machine has its own thread scheduler
  - Provides platform independent thread scheduling
  - Usually maps down to or interacts with underlying OS scheduler
- Scheduling is not defined
  - Pre-emption is possible
  - Time-sharing is possible
  - Combination is possible too
  - Thread priority is a hint
- RTSJ demands priority preemptive schedule

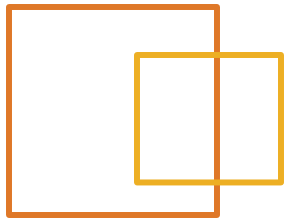
Extending Thread Example

```
public class Threads1 extends Thread {
   private int iterations = 0;  // loop counter
   private int id;              // number of thread
   private static int threadNumber = 1;
   public Ex12_1 () {
     id = threadNumber++;
     start();
   }
   public void run() {
     while (iterations++ <3) {
       try {
         sleep(5);
     } catch (InterruptedException e) {}
     System.out.println("Thread "+id+": iteration "+iterations );}
   }
   public static void main(String[] args) {
     for(int i = 0; i < 4; i++){
       new Threads1();
     }
   }
 }
```

Extending Thread Example Output

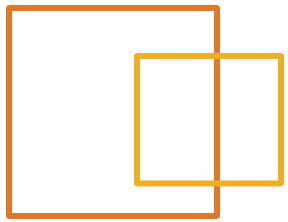


Thread Control



- Since Java might be pre-emptive, you might need to do some work to avoid thread starvation
- `java.lang.Thread` has methods that can help
  - `sleep`
  - `yield`
- To stop a thread, write code in the thread that polls a "stop flag", and have the thread shut itself down cleanly
  - Methods to stop, suspend, or resume a `Thread`'s execution have all been deprecated

Thread Control (cont.)



- Adjusting priority might also be useful to prevent starvation
  - Implementation is not guaranteed
  - Integral values that range between the values provided in the Thread class
  - Range represented as constants
  - MIN_PRIORITY
  - MAX_PRIORITY
  - Range usually falls between 1 and 10 with default value of 5
  - Numeric range of priorities is platform independent

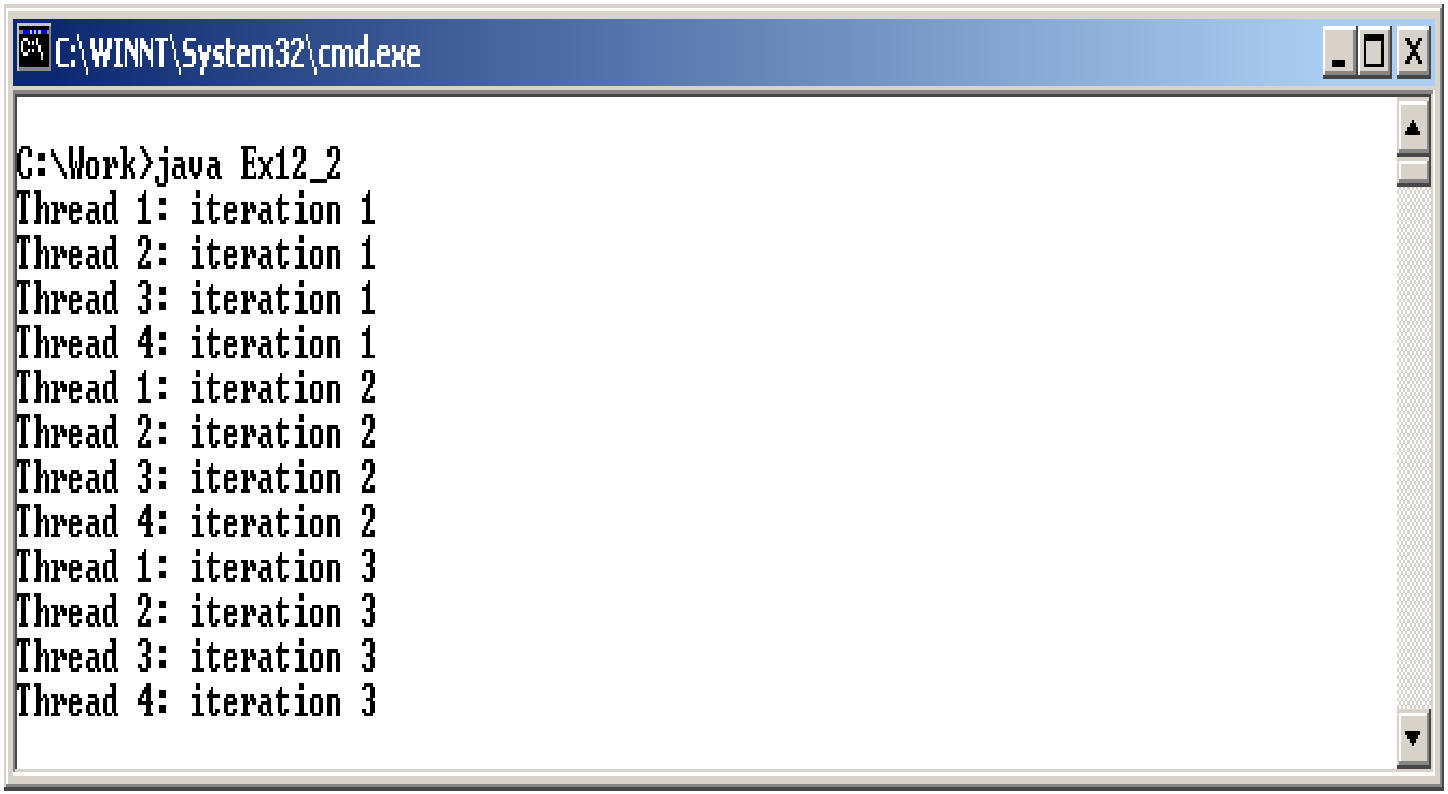
Daemon Threads

- Consider creating daemon threads
  - Like daemon processes in an operating system
  - Run continuously in the background
  - Method `setDaemon()` is called before the `start()`
  - Cannot start any non-daemon threads
  - Do not keep JVM alive—VM quits when all non-daemon threads have quit

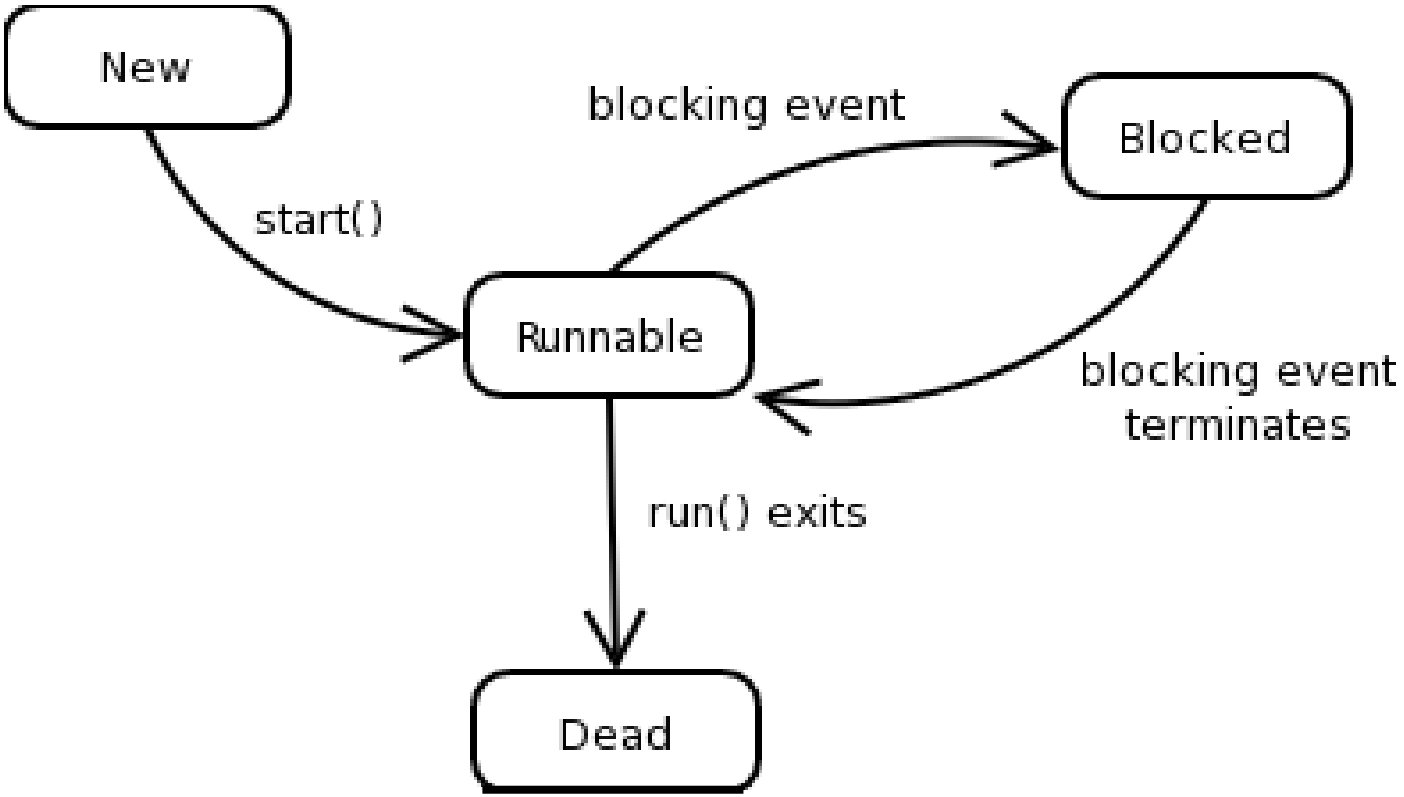
Thread Yielding Example

```
public class Yielding extends Thread {
  private int iterations = 0;  // loop counter
  public void run() {
    while (iterations++ <3) {
      System.out.println("Thread "
        + Thread.currentThread().getName()
        + ": iteration " + iterations );
        yield();
       }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 4; i++){
      new Yielding().start();
    }
  }
}
```

Thread Yielding



Thread State Transitions



Interrupting Threads Example

```
class BlockedThread extends Thread {
  public BlockedThread() { start(); }

  public void run() {
    try {
      System.out.println("BlockedTread running...");
      synchronized(this) {
        System.out.println("BlockedTread blocking...");
        wait();
      }
    } catch(InterruptedException e) {
      System.out.println("BlockedTread Interrupted");
    }
  }
}
```

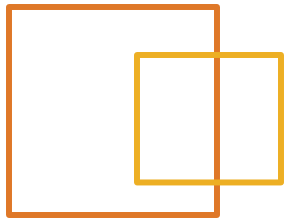
Interrupting Threads Example (cont.)

```
public class RudeThread extends Thread{
  static BlockedThread blocked = new BlockedThread();
  public static void main(String[] args) {
    RudeThread c = new RudeThread();
    c.start();
  }
  public void run() {
    try {
      sleep(5000);
    } catch (InterruptedException e) {}
    System.out.println("Preparing to interrupt ");
    blocked.interrupt();
    blocked = null;
  }
}
```

Runnable Interface Example

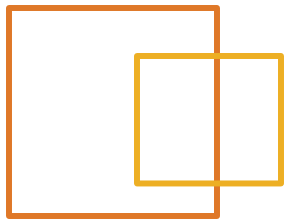
```
public class Threads2 implements Runnable {
  public void run() {
    System.out.println("In Runnable " + this);
  }
  public static void main(String[] args) {
    Thread t1 = new Thread(new Threads2());
    t1.start();
    System.out.println(" Thread Object " + t1);
  }
}
```

Thread Access Control



- Deals with synchronization of threads
  - Prevents corrupted data
  - Ensured intermediate results are ready before use
- There are two ways to create thread synchronization
  - synchronized methods
  - synchronized blocks
- Synchronization relies on obtaining and freeing object locks
- Object locks are obtained when a thread executes synchronized code

Thread Access Control



- `java.lang.Object` has built-in mechanisms for notifying other Thread about lock status
  - `wait`
  - `notify`
  - `notifyAll`

Synchronized Method Example

```
class Account {
  private PrintWriter out;
  Account(PrintWriter p) { out = p; }
  synchronized void deposit(int amount, String name){
    out.println(name + " deposit " + amount);
    int balance = getBalance();
    balance += amount;
    setBalance(balance);
  }
}
// Output of this code is
#1 trying to deposit 1000
#2 trying to deposit 1000
*** Final balance is 2000
```

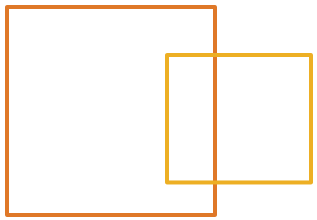
Synchronized Block Example

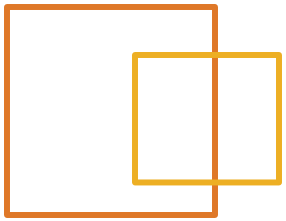
```
// Alternative deposit method in Account class
void deposit(int amount, String name) {
  int balance; // local copy of balance
  out.println(name + " deposit " + amount);
  synchronized(this) {
    balance = getBalance();
    balance += amount;
    setBalance(balance);
  }
}
// Output of this code is
#1 trying to deposit 1000
#2 trying to deposit 1000
*** Final balance is 2000
```

Thread Cooperation

```
first.start();
second.start();
// wait here until both threads complete
try {
  first.join();
  second.join();
} catch (InterruptedException e) {
  e.printStackTrace();
}
// Print the final result
out.println("Final balance is " + remoteBalance);
```

Summary





In this module, we covered:

- Concurrency and threads, and why they are used
- Creating and running a thread using the Thread class
- Thread priorities and scheduling
- What a daemon thread is
- Using the Runnable interface
- Using thread synchronization
- How threads are coordinated