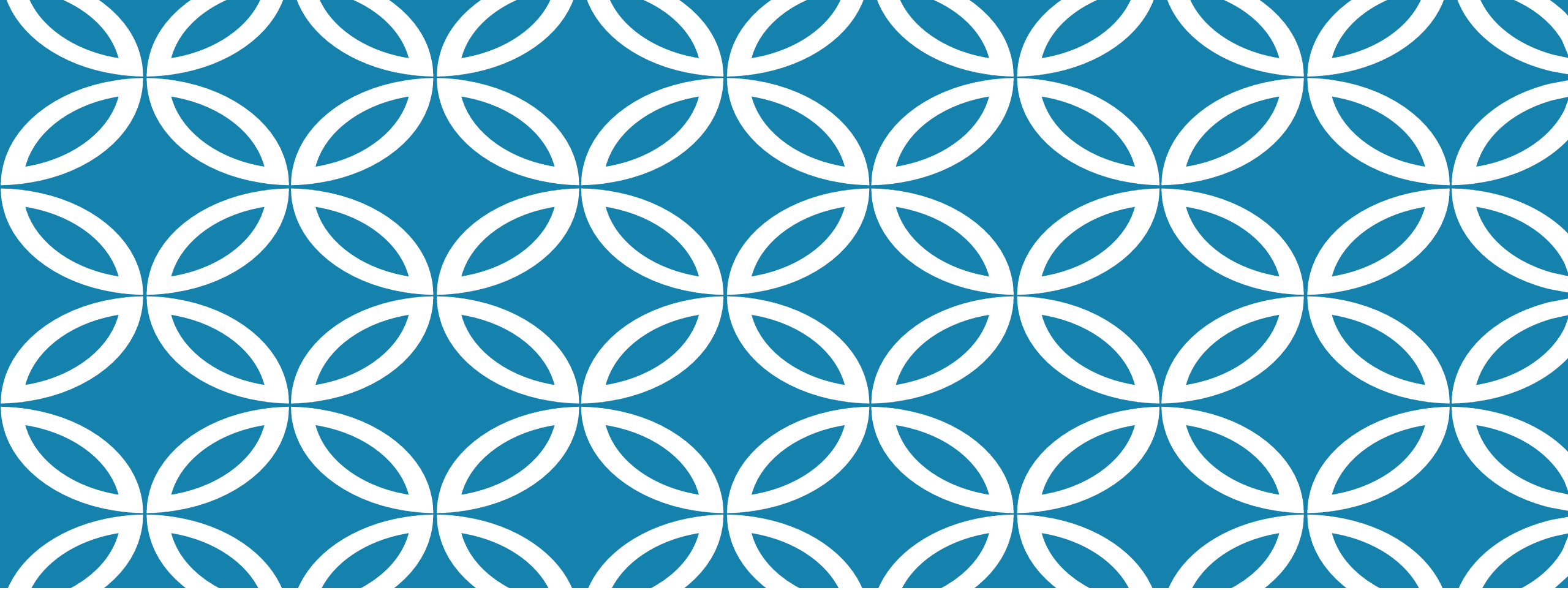

Chapter 11
Multithreaded Programming

Introduction:

- Unlike many other computer languages, Java provides built-in support for *multithreaded programming.*
- A multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a *thread,* and each thread defines a separate path of execution.
- Multithreading is a specialized form of multitasking.
- There are two distinct types of multitasking: process based and thread-based.
- ***process-based*** **multitasking** is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor
- In a ***thread-based*** **multitasking** environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, these two actions are being performed by two separate threads.

The Java Thread Model:

- Threads exist in several states.
- A thread can be *running*.
- It can be *ready to run* as soon as it gets CPU time.
- A running thread can be *suspended,* which temporarily suspends its activity.
- A suspended thread can then be *resumed,* allowing it to pick up where it left off.
- A thread can be *blocked* when waiting for a resource.
- At any time, a thread can be terminated, which halts its execution immediately.
- Once terminated, a thread cannot be resumed.

Thread Priorities:

- *A thread can voluntarily relinquish control*. This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

- *A thread can be preempted by a higher-priority thread.* In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking.*

Synchronization:

- if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the *monitor.*
- The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

Messaging:

Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

The Thread Class and the Runnable Interface:

To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

The **Thread** class defines several methods that help manage threads.

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

The Main Thread:

- When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program.
- It is the thread from which other "child" threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.
- The main thread is created automatically when your program is started.
- It can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**.
- Its general form is shown here:

  static Thread currentThread( )
- This method returns a reference to the thread in which it is called.
- Once you have a reference to the main thread, you can control it just like any other thread.

```java
// Controlling the main Thread.
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();

    System.out.println("Current thread: " + t);

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);

    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

Output:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Order: the name of the thread, its priority, and the name of its group.

Creating a Thread:

you create a thread by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:

- You can implement the **Runnable** interface.
- You can extend the **Thread** class.

Implementing Runnable:

The easiest way to create a thread is to create a class that implements the **Runnable** interface.

```
// Create a second thread.
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
```

```
class ThreadDemo {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {


     System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Output:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Extending Thread:

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class.

```
// Create a second thread by extending Thread
class NewThread extends Thread {

  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
```

```
class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Choosing an Approach:

classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of **Thread**'s other methods, it is probably best to implement **Runnable**.

Creating Multiple Threads:

your program can spawn as many threads as it needs.

```
// Create multiple threads.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;
```

```
NewThread(String threadname) {
  name = threadname;
  t = new Thread(this, name);
  System.out.println("New thread: " + t);
  t.start(); // Start the thread
}
```

```
  // This is the entry point for thread.
  public void run() {
   try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + "Interrupted");
    }
    System.out.println(name + " exiting.");
  }
}

class MultiThreadDemo {
  public static void main(String args[]) {
    new NewThread("One"); // start threads
    new NewThread("Two");
    new NewThread("Three");
    try {
      // wait for other threads to end
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    System.out.println("Main thread exiting.");
  }
}
```

output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

Using isAlive( ) and join( ):

final boolean isAlive( )

The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

final void join( ) throws InterruptedException

- This method waits until the thread on which it is called terminates.
- Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
  }
}
```

```
class DemoJoin {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");
    NewThread ob3 = new NewThread("Three");

 System.out.println("Thread One is alive: "
                    + ob1.t.isAlive());
 System.out.println("Thread Two is alive: "
                    + ob2.t.isAlive());
 System.out.println("Thread Three is alive: "
                    + ob3.t.isAlive());
 // wait for threads to finish
try {
   System.out.println("Waiting for threads to finish.");
   ob1.t.join();
   ob2.t.join();
   ob3.t.join();
 } catch (InterruptedException e) {
   System.out.println("Main thread Interrupted");
 }
                    System.out.println("Thread One is alive: "
                                       + ob1.t.isAlive());
                    System.out.println("Thread Two is alive: "
                                       + ob2.t.isAlive());
                    System.out.println("Thread Three is alive: "
                                       + ob3.t.isAlive());

                    System.out.println("Main thread exiting.");
                  }
                }
```

output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

Thread Priorities:

To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**.
This is its general form:
final void setPriority(int *level*)

- *level* specifies the new priority setting for the calling thread.
- The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**.
- Currently, these values are 1 and 10, respectively.
- To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5.
- These priorities are defined as **static final** variables within **Thread**.

You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here:
final int getPriority( )

```
// Demonstrate thread priorities.
class clicker implements Runnable {
  long click = 0;
  Thread t;
  private volatile boolean running = true;

  public clicker(int p) {
    t = new Thread(this);
    t.setPriority(p);
  }

  public void run() {
    while (running) {
      click++;
    }
  }

  public void stop() {
    running = false;
  }

  public void start() {
    t.start();
  }
}
```

```
class HiLoPri {
  public static void main(String args[]) {
    Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
    clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
    clicker lo = new clicker(Thread.NORM_PRIORITY - 2);

    lo.start();
    hi.start();
    try {
      Thread.sleep(10000);
    } catch (InterruptedException e) {
        System.out.println("Main thread interrupted.");
    }


  lo.stop();
  hi.stop();

  // Wait for child threads to terminate.
  try {
    hi.t.join();
    lo.t.join();
  } catch (InterruptedException e) {
    System.out.println("InterruptedException caught");
  }
```

```
    System.out.println("Low-priority thread: " + lo.click);
    System.out.println("High-priority thread: " + hi.click);
  }
}
```

output from this program is shown here:

```
Low-priority thread: 4408112
High-priority thread: 589626904
```

Synchronization:

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called *synchronization.*
- Java provides unique, language-level support for it.
- Key to synchronization is the concept of the monitor (also called a *semaphore*).
- A *monitor* is an object that is used as a mutually exclusive lock, or *mutex.*

- Only one thread can *own* a monitor at a given time.
- When a thread acquires a lock, it is said to have *entered* the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first
- thread *exits* the monitor. These other threads are said to be *waiting* for the monitor.
- A thread that owns a monitor can reenter the same monitor if it so desires.

You can synchronize your code in either of two ways

1) Using Synchronized Methods
2) Using synchronized Statement

Using Synchronized Methods:

```
// This program is not synchronized.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}

class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;
```

```
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }

  public void run() {
    target.call(msg);
  }
}

class Synch {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");
    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```

output from this program is shown here:

```
Hello[Synchronized[World]
]
]
```

To fix the preceding program, you must *serialize* access to **call( )**. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede **call( )**'s definition with the keyword **synchronized**,

```
class Callme {
    synchronized void call(String msg) {
    ...
```

This prevents other threads from entering **call( )** while another thread is using it. After **synchronized** has been added to **call( )**, the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

The synchronized Statement:

- Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods.

- Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class.
- How can access to an object of this class be synchronized?
- The solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

```
synchronized(object) {
  // statements to be synchronized
}
```

```
// This program uses a synchronized block.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}

class Caller implements Runnable {
  String msg;

  Callme target;
  Thread t;
```

```
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }

  // synchronize calls to call()
  public void run() {
    synchronized(target) { // synchronized block
      target.call(msg);
    }
  }
}
```

```
class Synch1 {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```

Interthread Communication:

- **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
- **notify( )** wakes up a thread that called **wait( )** on the same object.
- **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.

These methods are declared within **Object**, as shown here:
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
Additional forms of **wait( )** exist that allow you to specify a period of time to wait.

```
// An incorrect implementation of a producer and consumer.
class Q {
  int n;

  synchronized int get() {
    System.out.println("Got: " + n);
    return n;
  }

  synchronized void put(int n) {
    this.n = n;
    System.out.println("Put: " + n);
  }
}

class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }

  public void run() {
    int i = 0;

    while(true) {
      q.put(i++);
    }
  }
}

class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}
```

```
class PC {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println("Press Control-C to stop.");
  }
}
```

output from this program is shown here:

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

```
// A correct implementation of a producer and consumer.
class Q {
  int n;
  boolean valueSet = false;

  synchronized int get() {
    while(!valueSet)
      try {
        wait();

     } catch(InterruptedException e) {
        System.out.println("InterruptedException caught");
      }

      System.out.println("Got: " + n);
      valueSet = false;
      notify();
      return n;
  }

  synchronized void put(int n) {
    while(valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("InterruptedException caught");
      }
      this.n = n;
      valueSet = true;
      System.out.println("Put: " + n);
      notify();
    }
}

class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }

  public void run() {
    int i = 0;

    while(true) {
      q.put(i++);
    }
  }
}
```

```
class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}

class PCFixed {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println("Press Control-C to stop.");
  }
}
```

output from this program is shown here:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

Deadlock:

deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects

```
// An example of deadlock.
class A {
  synchronized void foo(B b) {
    String name = Thread.currentThread().getName();

    System.out.println(name + " entered A.foo");

    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("A Interrupted");
    }

    System.out.println(name + " trying to call B.last()'
    b.last();
  }

  synchronized void last() {
    System.out.println("Inside A.last");
  }
}
```

```
class B {
synchronized void bar(A a) {
  String name = Thread.currentThread().getName();
  System.out.println(name + " entered B.bar");

  try {
    Thread.sleep(1000);
  } catch(Exception e) {
    System.out.println("B Interrupted");
  }

  System.out.println(name + " trying to call A.last()");
  a.last();
}

synchronized void last() {
  System.out.println("Inside A.last");
}
}
```

```
class Deadlock implements Runnable {
  A a = new A();
  B b = new B();

  Deadlock() {
    Thread.currentThread().setName("MainThread");
    Thread t = new Thread(this, "RacingThread");
    t.start();

    a.foo(b); // get lock on a in this thread.
    System.out.println("Back in main thread");
  }

  public void run() {
    b.bar(a); // get lock on b in other thread.
    System.out.println("Back in other thread");
  }

  public static void main(String args[]) {
    new Deadlock();
  }
}
```

output from this program is shown here:

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()
```

Suspending, Resuming, and Stopping Threads:

Suspending, Resuming, and Stopping Threads Using Java 1.1 and Earlier:

Prior to Java 2, a program used **suspend( )** and **resume( )**, which are methods defined by **Thread**, to pause and restart the execution of a thread. They have the form shown below:

final void suspend( )

final void resume( )

```
// Using suspend() and resume().
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 15; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(200);
      }
    } catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
  }
}
```

```
class SuspendResume {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");

    try {
      Thread.sleep(1000);
      ob1.t.suspend();
      System.out.println("Suspending thread One");
      Thread.sleep(1000);
      ob1.t.resume();
      System.out.println("Resuming thread One");
      ob2.t.suspend();
      System.out.println("Suspending thread Two");
      Thread.sleep(1000);
      ob2.t.resume();
      System.out.println("Resuming thread Two");
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }
    // wait for threads to finish
    try {
      System.out.println("Waiting for threads to finish.");
      ob1.t.join();
      ob2.t.join();
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Sample output from this program is shown here. (Your output may differ based on processor speed and task load.)

```
New thread: Thread[One,5,main]
One: 15
New thread: Thread[Two,5,main]
Two: 15
One: 14
Two: 14
One: 13
Two: 13
One: 12
Two: 12
One: 11
Two: 11
Suspending thread One
Two: 10
Two: 9
Two: 8
Two: 7
Two: 6
Resuming thread One
Suspending thread Two
One: 10
One: 9
One: 8
One: 7
One: 6
Resuming thread Two
Waiting for threads to finish.
Two: 5
One: 5
Two: 4
One: 4
Two: 3
One: 3
Two: 2
One: 2
Two: 1
One: 1
Two exiting.
One exiting.
Main thread exiting.
```

The **Thread** class also defines a method called **stop( )** that stops a thread. Its signature is shown here:

final void stop( )

Once a thread has been stopped, it cannot be restarted using **resume( )**.

The Modern Way of Suspending, Resuming, and Stopping Threads:

While the **suspend( )**, **resume( )**, and **stop( )** methods defined by **Thread** seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs, because these methods are deprecated by Java 2.

```
// Suspending and resuming a thread the modern way.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;
  boolean suspendFlag;
```

```
NewThread(String threadname) {
  name = threadname;
  t = new Thread(this, name);
  System.out.println("New thread: " + t);
  suspendFlag = false;
  t.start(); // Start the thread
}
```

```
// This is the entry point for thread.
public void run() {
  try {
    for(int i = 15; i > 0; i--) {
      System.out.println(name + ": " + i);
      Thread.sleep(200);
      synchronized(this) {
        while(suspendFlag) {
          wait();
        }
      }
    }
  } catch (InterruptedException e) {
    System.out.println(name + " interrupted.
  }

  System.out.println(name + " exiting.
}

void mysuspend() {
  suspendFlag = true;
}

  synchronized void myresume() {
    suspendFlag = false;
    notify();
  }
}

class SuspendResume {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");

    try {
      Thread.sleep(1000);
      ob1.mysuspend();
      System.out.println("Suspending thread One");
      Thread.sleep(1000);
      ob1.myresume();
      System.out.println("Resuming thread One");
      ob2.mysuspend();
      System.out.println("Suspending thread Two");
      Thread.sleep(1000);
      ob2.myresume();
      System.out.println("Resuming thread Two");
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }
```

```
    // wait for threads to finish
    try {
      System.out.println("Waiting for threads to finish.");
      ob1.t.join();
      ob2.t.join();
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    System.out.println("Main thread exiting.");
  }
}
```

Using Multithreading:

- when you have two subsystems within a program that can execute concurrently, make them individual threads.
- If you create too many threads, you can actually degrade the performance of your program rather than enhance it.
- Some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program!

Disclaimer

- These slides are not original and have been prepared from various sources for teaching purpose.

Sources:

- Herbert Schildt, Java™: The Complete Reference

Thank You