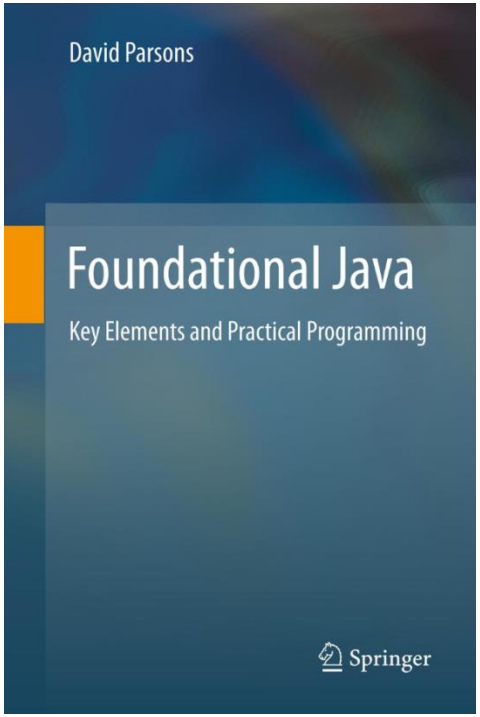


Chapter 16
Multithreading

Foundational Java

Key Elements and Practical Programming

Threads

- Each thread is a separate computation unit
- It is not a process
  - Thread based multitasking allows parts of the same program to run concurrently
- Shares data and code with other threads in the same program
  - They are *lightweight*
- All Java programs have a main user thread
  - But other child threads can be spawned from it
  - Can be user or daemon threads

Implementing Multithreading

- Multithreaded programs can be written using inheritance from class Thread
- Or can implement the Runnable interface
- Java supports thread synchronisation through methods of the Object class

Creating and Running a Thread

- Extending the Thread class
- The parameterised Thread constructor
- Methods of the Thread class
  - getName
  - run
  - Thread.sleep (static)
  - start
- InterruptedException
  - Checked exception of the ‘sleep’ method

The Tortoise

- Subclass of Thread

```
public class Tortoise extends Thread
```

- Can use the constructor to set the thread name
  - default name will be similar to ‘Thread-0’

```
public Tortoise(String name)
{
    super(name);
}
```

The Thread's 'run' Method

- The 'run' method defines the thread's actions

```
public void run()
{
    int sleepTime;
    for (int i = 0; i < 10; i++)
    {
            …
    }
    System.out.println(getName() + " has finished!");
}
```

Sleeping Thread

- The (static) 'sleep' method puts the thread to sleep for the specified number of milliseconds
- Has a checked exception that must be handled
  - InterruptedException
  - May be triggered by the interrupt() method

```
sleepTime = (int) (Math.random() * 1000);
try
{
   Thread.sleep(sleepTime);
}
catch (InterruptedException e)
{
   e.printStackTrace();
}
```

‘run’ Method

```
@Override
 public void run()
 {
   int sleepTime;
   for (int i = 0; i < 10; i++)
   {
       System.out.println(getName() + " has gone " + i + " metres");
       sleepTime = (int) (Math.random() * 1000);
       try {
           Thread.sleep(sleepTime);
       }
       catch (InterruptedException e) {
          System.out.println("Interrupted");
        }
      }
      System.out.println(getName() + " has finished!");
}
```

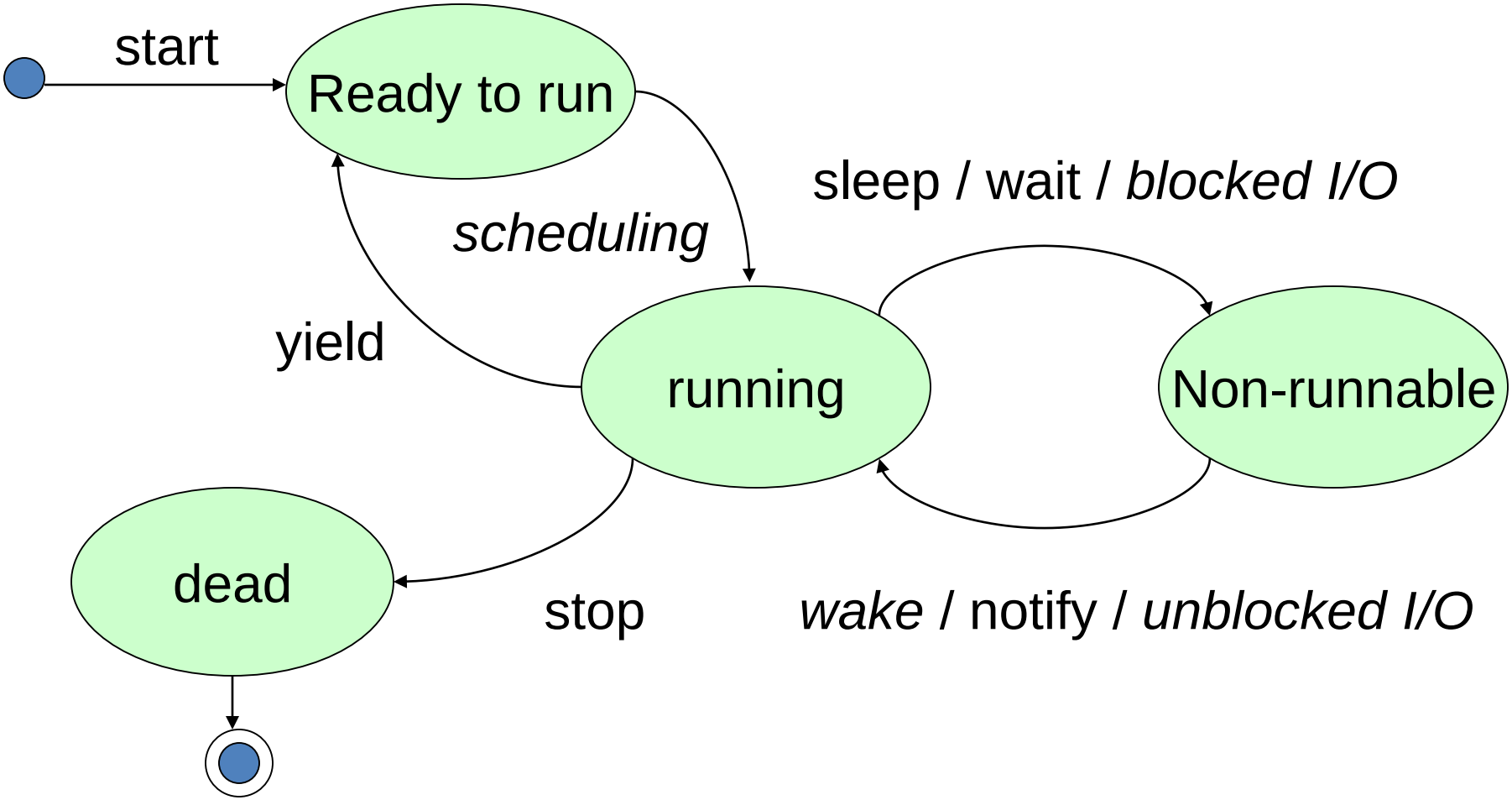
Starting a Thread

- Create a new instance of the Thread subclass
- Call the ‘start’ method
  - This invokes the ‘run’ method of the Thread

```
Tortoise racingTortoise = new Tortoise("Tortoise");
racingTortoise.start();
```

  - Note that we don’t call ‘run’ directly

Thread States



Running Multiple Threads

- Multiple threaded objects can be started

```
// create two separate thread objects (a tortoise and a hare)
    Hare racingHare = new Hare("Hare");
    RacingTortoise racingTortoise = new RacingTortoise("Tortoise");
// start them both racing
    racingHare.start();
    racingTortoise.start();
```

Exercise 16.1

- Write a class called RaceHorse that inherits from Thread and jumps fences between randomly generated sleeps
- After a horse has jumped five fences it passes the finishing line
- Use the constructor to set the name of the horse
- Create a Steeplechase class that starts a number of horses running (but don't bet real money on the winner)

Thread Priority

- We can set the relative priorities of different threads using the ‘setPriority’ method
- This takes an integer parameter in the range 1 (low) to 10 (high)
- There are 3 priority constants in the Thread class
  - MIN_PRIORITY (1)
  - NORM_PRIORITY (5)
  - MAX_PRIORITY (10)

‘Bees’ are a Selfish Thread

- The ‘run’ method contains a tight loop
- It carries on without giving other threads a chance
- i.e. There is no ‘sleep’

```
public class Bees extends Thread
{
   public void run()
    {
        for (int i = 0; i < 20000; i++)
        {
            System.out.print("z");
        }
    }
}
```

The Bear

- The bear wants to get at the honey when it wakes up

```
public class Bear extends Thread
{
    public void run()
    {
        try
        {
            Thread.sleep(50);
        } catch (InterruptedException e)
        { … }
      System.out.println("Mmmmmm, honey, yum yum!");
    }
}
```

Relative Priorities

- One way of the bear getting access to the honey is by setting its priority to be much higher

```
Bees honeyBees = new Bees("Honey bees");
honeyBees.setPriority(Thread.MIN_PRIORITY);
Bear hungryBear = new Bear("Hungry bear");
hungryBear.setPriority(Thread.MAX_PRIORITY);
honeyBees.start();
hungryBear.start();
```

- This is dependent on the operating system being pre-emptive to give the bear a time slice

Exercise 16.2

- Modify the Steeplechase class so that one horse has maximum priority, and the others have minimum priority
- Is this enough to ensure that the maximum priority horse always win the race?

Yielding

- Threads can be less selfish by being prepared to 'yield' to other threads
- A running thread can call 'yield' during an activity
- This gives other threads the chance to run

```
public void run()  {
    for (int i = 0; i < 10000; i++)  {
        System.out.print("z");
        if(i%500 == 0)
        {
           Thread.yield();
        }
    }
}
```

Implementing the Runnable Interface

- Inheriting from 'Thread' prevents us from subclassing anything else
- The alternative is to use the 'Runnable' interface

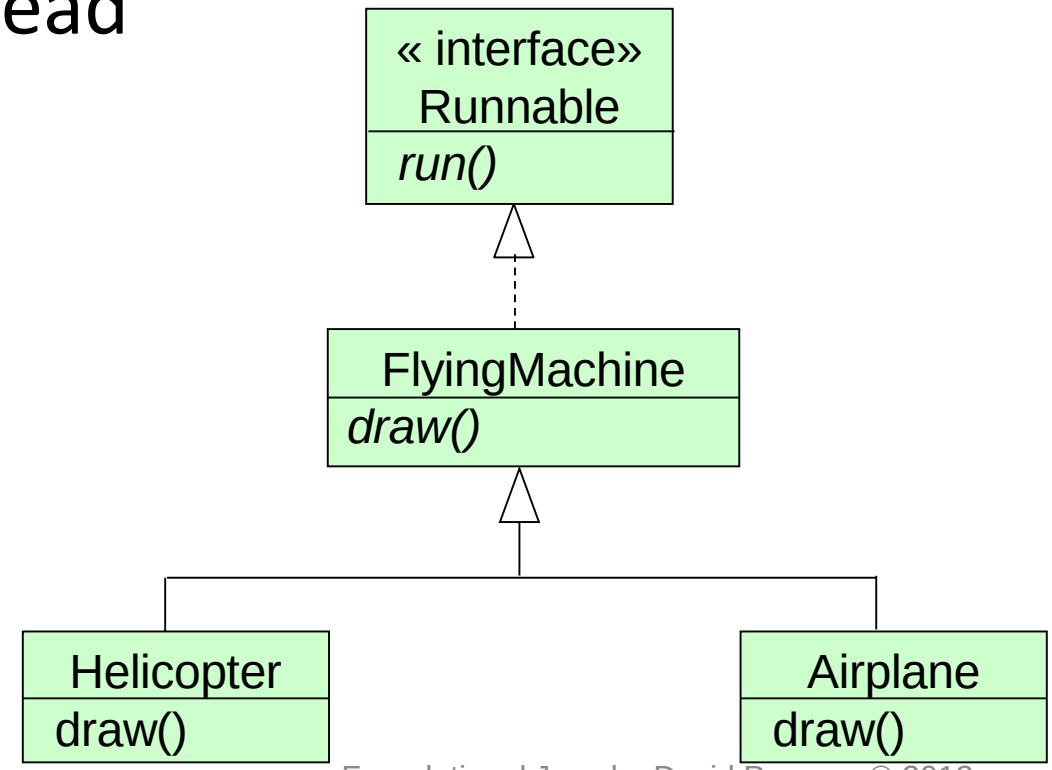
```
public class abstract FlyingMachine implements Runnable
```

- We could then extend another class, if required

```
public class Airplane extends FlyingMachine implements Runnable
```

FlyingMachine Hierarchy

- In this hierarchy, the Helicopter and Airplane need to inherit from FlyingMachine, not Thread



Thread Instances

- We can declare Thread objects

```
private Thread thread;
```

- Then pass the Runnable object (this) as the parameter to the thread constructor

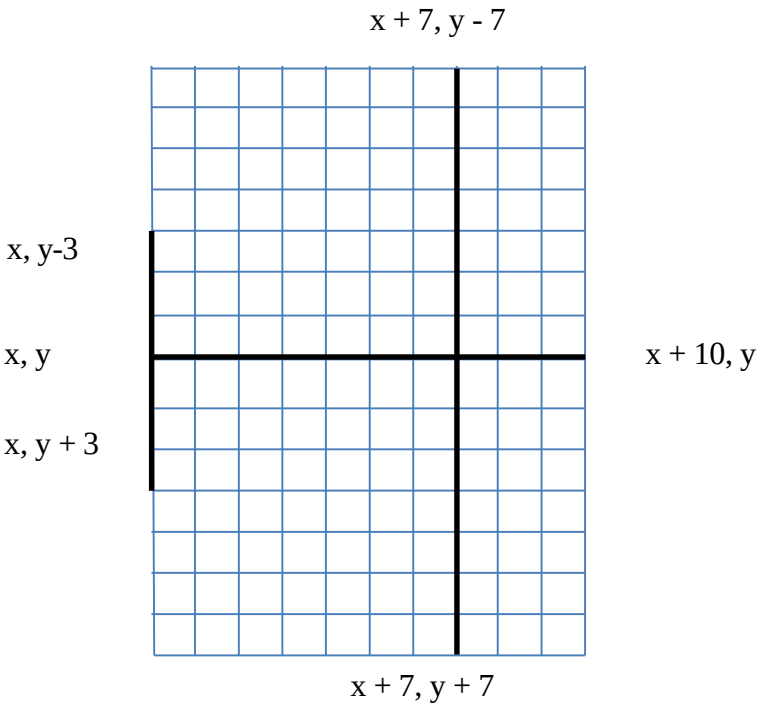
```
thread = new Thread(this);
```

- When a class inherits from Thread, it inherits all Thread methods
- The Runnable interface only has ‘run’
  - Other thread methods are invoked on the Thread object

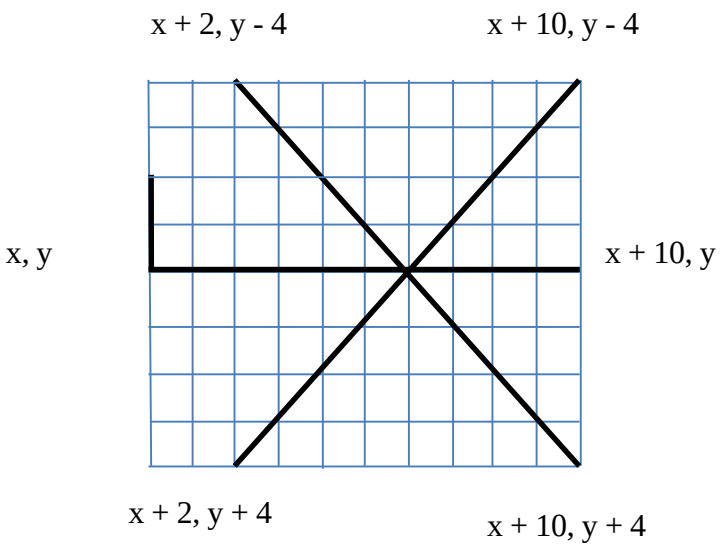
```
thread.start();
```

FlyingMachine Coordinates

- The threaded objects draw themselves from an x,y point using relative coordinates



Airplane



Helicopter

Rotating Lines

- An AffineTransform is used to rotate the shapes by quadrants

```
AffineTransform transform = AffineTransform.getQuadrantRotateInstance
    (getDirection()-1, getX(), getY());
```

- The transform converts from an array of coordinates to a transformed array

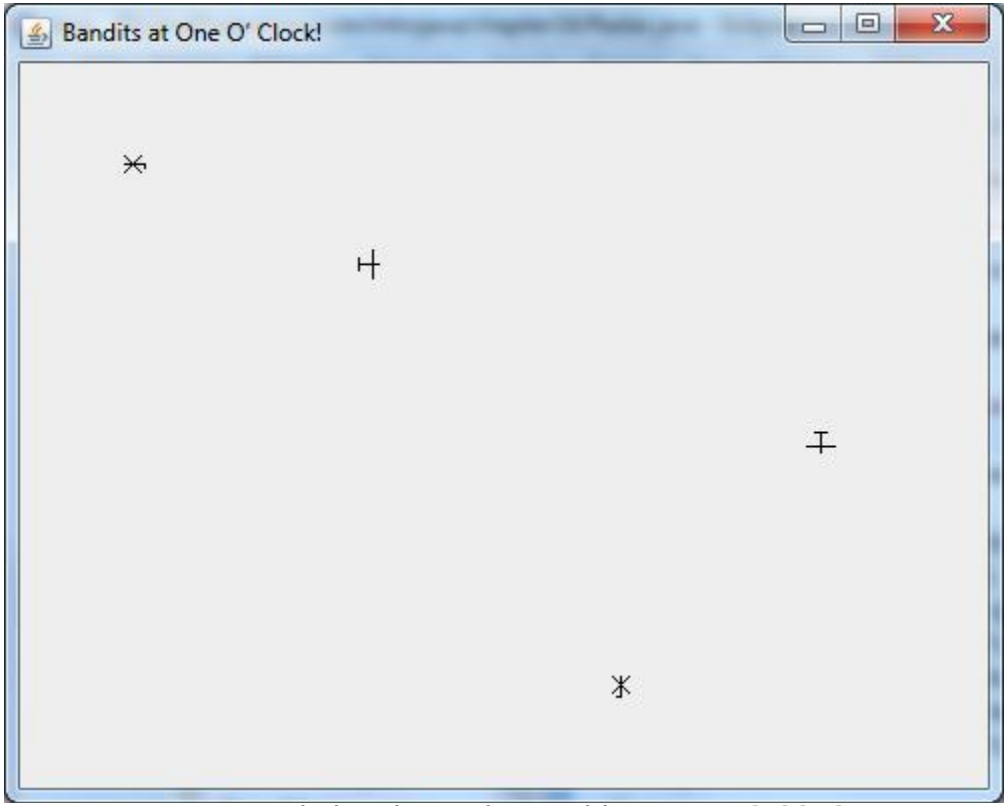
```
transform.transform(values, 0, result, 0, 6);
```

- The transformed array can be used to draw on the screen

```
getGraphics().drawLine((int) result[0], (int) result[1],
    (int) result[2], (int) result[3]);
```

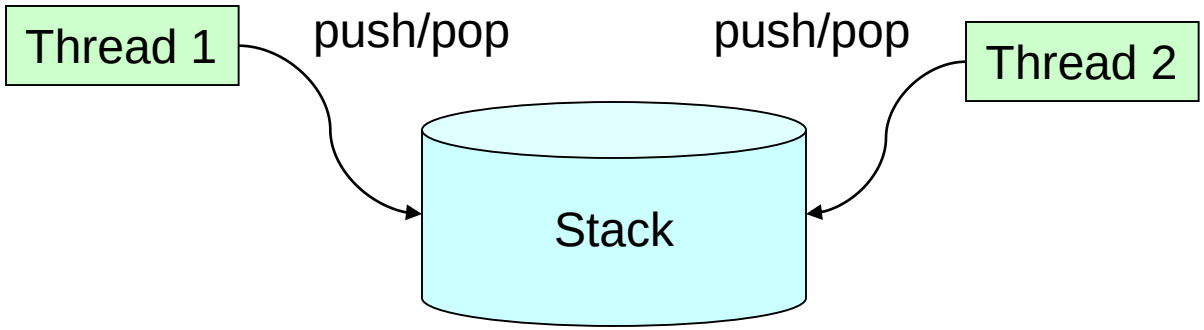
The Radar Example

- In the ‘run’ method, each flying machine updates its position and redraws itself



Synchronizing Threads

- Threads become more useful when we can synchronize their activities
- But we need to control multiple access to parts of the system



- There are some keywords and Object methods that support multiple thread management

Monitors and Synchronization

- Where multiple threads are used, some code may need to be isolated from concurrent thread access
- An object that can block threads and notify them when it becomes available is known as a *monitor*
- In Java, any object with one or more *synchronized* methods is a monitor
- A monitor represents some kind of shared resource that needs to be thread safe

Synchronized Code

- The ‘synchronized’ keyword can be used to indicate code that is single threaded
- A method can be synchronized

```
public synchronized double[] getResults()
```

- Or just a block of code

```
synchronized (object)
{
        // code here
}
```

‘wait’, ‘notify’ and ‘notifyAll’

- All Java objects are thread aware
- In synchronized code, ‘wait’ and ‘notify’ can be used to manage multiple thread access
  - ‘wait’ methods block threads
  - A waiting object can be released by ‘notify’ methods
- These methods can only be used inside synchronized code
  - Otherwise you will get an IllegalMonitorStateException

‘wait’ Methods

Wait Method	Effect
void wait()	Causes the current thread to wait until another thread notifies it to be able to run again
void wait(long timeout)	like wait, but has a millisecond timeout
void wait(long timeout, int nanos)	like wait, but has a nanosoecond timeout

- ‘wait’ should always be called in a loop, to ensure that when the thread is ready to run, the correct conditions are in place
  - If not, we wait again

‘notify’ and ‘notifyAll’ Methods

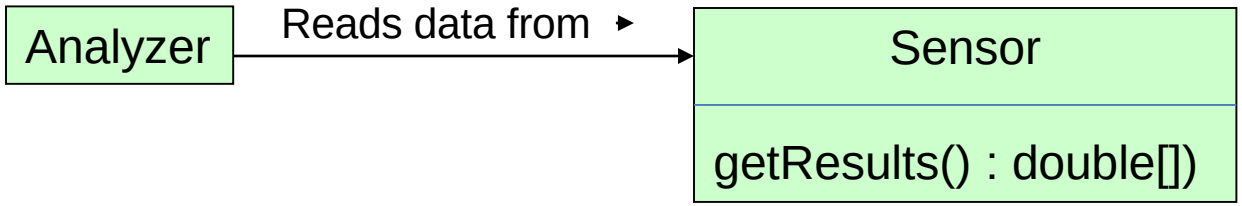
Notify Method	Effect
void notify()	Wakes up a single thread that is waiting on this object's monitor
void notifyAll()	Wakes up all threads that are waiting on this object's monitor

- If ‘notifyAll’ is called, it will be up to the thread scheduler in the Java runtime to decide which thread actually gets to run

Synchronized Code Example

Analyzer wants to read data in batches of 10 values from the Sensor

Sensor takes time to assemble 10 results
Makes the Analyzer wait, and notifies it when it is ready



- We can make the Sensor program wait until 10 results are available, then notify it

Volatile Fields

- Threads may cache object fields for optimisation
- If an object field is cached and accessed by multiple threads, inconsistent state may result
- To avoid this, mark fields as ‘volatile’
  - This prevents the threads from caching

```
private volatile double[] results= new double[10];
```

Waiting Thread

- In this synchronized method, we wait if there are not enough results

```
public synchronized double[] getResults()
{
  if (!ready) {
    try {
      wait();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  return results;
}
```

Notifying

- In this synchronized method, we notify that the results are ready for a waiting thread

```
public synchronized void addResult(double result)
{
  results[index] = result;
  if (index == MAX_READINGS - 1)
  {
    ready = true;
    notify();
    ready = false;
    index = 0;
  }
  else
  {
    index++;
  }
}
```

The Analyzer Class

- The waiting Thread
- Creates an instance of the Sensor class and starts it running

```
public Analyzer(Sensor sensor)
{
  dataSource = sensor;
  dataSource.start();
}
```

The ‘run’ Method

- The ‘run’ method calls the ‘getResults’ method of the Sensor, which will make it wait until it is notified

```
public void run()
{
  while (true)
  {
    double[] data = dataSource.getResults();
    double result = 0.0;
    for (int i = 0; i < data.length; i++)
    {
      result += data[i];
    }
    System.out.println("Result: " + result);
  }
}
```

Exercise 16.3

- Create three classes; Buffer, DataSource and Display
  - A Buffer object can only store one integer at a time
  - A DataSource object should start a thread that counts up from zero and stores each value in a Buffer object
  - A Display object should create a thread that keeps reading values from the Buffer object and printing them on the console
- Manage the thread access so that each number is displayed on the console only once
- The Buffer class needs to have some synchronized code that makes the DataSource and Display objects wait until their threads can have access, and then notifies them when it is ready

Concurrent Collections

- A concurrent collection is thread-safe, but not governed by a single exclusion lock
  - Synchronized collections prevent all access via a single lock, at the expense of poorer scalability
- Where multiple threads are expected to access a common collection, concurrent versions are normally preferable
  - Unsynchronized collections are preferable when either collections are unshared, or are accessible only when holding other locks

Sample Classes

- java.util.concurrent.ConcurrentLinkedQueue
  - An efficient scalable thread-safe non-blocking FIFO queue
- java.util.concurrent.ArrayBlockingQueue
  - Implements the BlockingQueue interface
  - Representing a fixed size queue where threads may be blocked if, for example, they try to add elements when the queue is full

BlockingQueue

Four forms of methods:

1. throws an exception
2. returns a special value (either null or false, depending on the operation)
3. blocks the current thread indefinitely
4. blocks for only a given maximum time limit

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	*not applicable*	

Summary

- Inheritance from the Thread class
- Implementing the Runnable interface
- Thread priorities
- Making threads less selfish with ‘yield’
- Object methods ‘wait’, ‘notify’ and ‘notifyAll’ for synchronized code
- Concurrent collections in the java.util.concurrent package