Concurrent Programming with Java
Tony Mullins

Most programmers have a love-hate relationship with concurrency.

Programming concurrency is hard, yet the benefits it provides make all the troubles worthwhile. The processing power we have at our disposal, at such an affordable cost, is something that our parents could only dream of. We can exploit the ability to run multiple concurrent tasks to create stellar applications. We have the ability to write applications that can provide a great user experience by staying a few steps ahead of the user. Features that would've made apps sluggish a decade ago are quite practical today. To realize this, however, we have to program concurrency.

(Venkat Subramaniam, Programming Concurrency on the JVM, 2011)

The swing away from assembly language, which gained genuine momentum during the seventies, was slow to affect the area of concurrent systems – operating systems, embedded control systems and the like. What happened was that three people – Edsger Dijkstra, Tony hoare and Per Brinch Hanson – independently developed key abstractions which were taken up by researchers worldwide, realized in experimental languages, reported on, adapted and refined. In this way, the problems of concurrency could be expressed in well understood notation and principles gradually evolved.

(Judy Bishop, Data Abstraction in Programming Languages, 1986)

For a parallel programming language the most important security measure is to check that processes access disjoint sets of variables only and do not interfere with each other in-time dependent ways.

(Tony Hoare, 1974)

The only secret about secure concurrent programming languages was that they could be designed at all.

(Per Brinch Hanson, The Origin of Concurrent Programming, 2002)

In the past it was the purpose of our programs to instruct our machines: now it is the purpose of machines to execute our programs

(Edsger Dijkstra)

Table of Contents

Introduction

We begin, contrary to Dijkstra's position, not with a discussion of concurrent programming languages but rather with a discussion of modern processors. The reason simply is that programming modern processors makes it imperative that we, as programmers, understand the main concepts behind concurrent programs and how they impact in our understanding and ability to write correct, efficient programs that are deployed on modern hardware machines.

Processors were traditionally developed with single central processing units, called a core in to-days terminology. For many decades the performance of single CPU's improved every couple of years. Clock rates increased in measurement from megahertz in the 1980's to gigahertz in the 1990's. However, increased power led to greater problems with cooling and hardware manufacturers changed direction and developed multi-core systems. These systems consisted initially of single processor units with dual cores. That is, two CPU's in a single processor (e.g. AMD Phenom II X2, Intel Core Duo). Consider the new Intel 4th Generation Core Processor Core i7-4770K that will be used in desktop machines. This processor is a quad core machine capable of running 8 independent processing threads in parallel using 8 MB of what is termed *smart cache memory*. Intel's Gulftown processor runs up to 12 threads in parallel and has a 12MB cache. Clearly, these processors are designed to do more than a single task at the same time. This means simply that they can execute multiple tasks in parallel. If so, how are tasks defined. The key word is: **thread**. What are threads and how do they encapsulate actions or tasks to be executed by a processor? How do we write these threads? Is it possible to harness multiple threads to accomplish a given task? Can these threads communicate with each other? These are just some of the questions we plan to answer in this text.

But first there is another reason why we, as programmers, must understand concurrency and its implications for programming. Even if you never write a thread in your program there are most likely threads in your code. All the modern frameworks that we use tend to create threads behind the scenes. When the JVM starts it creates threads that manage certain tasks, e.g. garbage collection. The library used to write graphical user interfaces uses threads behind the scenes to manage user interface events. Many web-based programs use servlets to handle user requests on a server. These servlets use threads to execute the code of the servlet that implements the request. To quote Goetz:

> *Frameworks introduce concurrency into applications by calling application components from framework threads. Components invariably access application state, thus requiring that all code paths accessing that state be thread-safe.*

(Java Concurrency in Practice, p9)

This statement suggests that we have to write classes so that they are *thread-safe*. What does this mean for programmers and how does it impact in the way we write classes? Again, we will address this fundamental question in the text.

Concurrent Processes

A process is an executing program. On modern operating systems (Windows, Unix, Linux, Mac OS) we can execute multiple processes or programs at the same time. When this happens the execution of the running processes may take one of three forms:

1. All processes share a single processor (CPU);
2. Each process may have its own processor and the processors share common memory;
3. Each process may have its own processor and the processes are distributed.

Two or more processes are said to be executing in **parallel** if they are both executing at the same time. To do so one of 2 or 3 forms described above must hold. That is, they must have a processor each or they must be running on multiple separate machines.

We say that two or more processes are **concurrent** if they have the potential for executing in **parallel**. This means that they will run in parallel if the conditions exist to allow them to do so. But what happens if there are more processes than processors? In such situations the operating system uses a scheduling policy, e.g. round robin priority based scheduling, that controls process access to the existing pool of cores on the underlying machine. The purpose of this scheduling policy is to optimize the performance of the executing processes. It may appear that this seriously impairs parallel processing but in reality this is not so because many processes spend long periods waiting for external events to occur. Many programs are I/O bound and spend much of the time waiting for events to occur. Consider a word-processor in action. Such a program spends about 99.9% of its executing time doing nothing. This means that other processes can use the processor while it is sleeping. Similarly, a web browser spends vast periods of time waiting for data to arrive over the network. When we say vast we are measuring time in terms of computing speed. A modern computer executing instructions at the rate of 10 million per second can do as much work in one second as a human can do in 300 years! Only tasks that engage in large-scale number crunching are actually, what we term, CPU bound. An example of such a process would be one to calculate Pi to 10 million places or a program that sorts a trillion numbers.

In modern programming languages we tend to think in terms of **threads** rather than processes. A thread defines a task and is considered a *lightweight process*. A single process when executed may consist of one or more threads. Every process must have at least one thread called `main(…)`. Additional threads are launched from `main` and threads, once running, will compete for the processors. In this way all threads in a process may run concurrently. If there are a sufficient number of cores then threads will run in parallel. All threads share the same memory space and, hence, execute under form 2 defined above for process execution.

Therefore, we can conclude that in modern programming environments there are two types of multi-tasking: process-based and thread-based. Process based multi-programming involves executing multiple processes concurrently and thread-based multi-tasking involves executing multiple threads of control in a single process. We will deal with both types of programs in detail in this text.

Concurrent Programming Languages

There are a number of modern programming languages that provide direct support for threads. The most popular of these are Java and C#. Ada-95 provides threading through its tasks. C++ in general does not provide direct support and requires the use of an external library called `PThreads`. Microsoft provides thread support though the .Net Framework. In the case of C++ you can use the Microsoft Foundation Class (MFC) library or the C run-time library and the Win32 API. Ruby and Python also support threads but with certain limitations. Scala provides a multi-threaded model based on actors that communicate through message passing. Threads are not explicit but are used in the background to implement actors. We will discuss actors when we deal with message passing systems later in the text.

None of the languages provide direct support for multiple processes. To deal with communication between multiple processes you need to access the services of the underlying operating system application programmer's interface. We will discuss this later in the text.

Concurrent Paradigm

A programming paradigm provides a methodology that allows the construction of correct, reliable programs constructed in a particular style. We have the imperative and object-oriented paradigms for program construction. These provided a methodology that allowed programs to be constructed in a particular style. Programming languages were written that reflected particular styles. For example, Pascal was created to reason in the imperative style and Eiffel for the object-oriented one. These languages provided a medium of expression. To quote Perlis: *a language that doesn't affect the way you think about programming is not worth knowing*. A programming language is not just a collection of statements it also encompasses in its design a notion of composition, a paradigm that allows *good* programs to be constructed. Interestingly, modern languages, such as Java and C#, not only reflect a particular style, the object-oriented one, but also provide a window to the world of concurrent programming. This should mean that not only do we get to write programs in an object-oriented style but also that we can do so in a multi-threaded universe.

The fundamental issue is, from a problem solving perspective, how does it affect the way we reason about solving problems. In a single threaded universe we just have one path of control. This is provided by the main method in our programs. Now we can think in terms of multiple paths of control that may co-operate to solve problems. This will change forever the way we think about solving lots of problems. *Many hands make light work* says the proverb. Consider the task of adding the elements in a very large integer array. We could write a function that simply adds all the values and returns their sum. However, in a parallel world we could get multiple threads to each take a segment of the array and do the addition. Finally, when all the threads finish we sum the result returned by each one. This divides the task and reduces the work each thread has to do. Two threads reduces it in half, four in a quarter, etc. There are many tasks that we could solve in this way. For example, the program listed below divides the task of searching an array for a given value between two threads. Each thread takes half of the elements and performs the search in its allocated segment. When the both threads finish the search main simply checks the values of fnd1 and fnd2 to see if x is contained in the array. The details of how we write the threads, etc, are not important here and will be explained in detail later.

```
class ConcurrencyExs{
    public static void main(String args[]){
        int f = new int[1000];
```

```
            // create data for f
            // input value for x
            Boolean fnd1 = new Boolean(false);
            Boolean fnd2 = new Boolean(false);
            Thread SearchT = new Thread(Search(f,x,0,500,fnd1);
            Thread SearchT1 = new Thread(Search(f,x,500,1000,fnd2);
            SearchT.start();
            SearchT1.start();
            // wait for search to end
            if(fnd1 || fnd2)
                System.out.println("found");
            else
                System.out.println("Not found");
    }
    // Code for Search(..)

}
```

It is important to note that the performance of this multi-threaded program will depend on the underlying architecture of the machine. If the machine has a processor with two cores the program will run about twice as fast as the standard sequential solution. If it has only one processor with one core it will be slightly slower because of the overhead of context switching between threads. It is important to note that the reasoning about a concurrent program is independent of the number of processors. The number of processors just affects the run time performance of the program.

Another interesting advantage of the concurrent paradigm is that it can simplify modeling systems that involve different types of task. It allows the programmer to focus on an individual task and deal with it in isolation from other tasks. Each separate task can be encoded as a thread of control and allowed to execute under its own constraints. This facility to encode tasks as separate threads of control is exploited by frameworks such as servlets and RMI (remote method invocation). It is also used in the design of servers to deal with multiple concurrent clients. Each client is allocated a separate thread that deals with it directly. This optimizes the service provided to clients and greatly simplifies the coding of services to clients. Again, we will discuss the design of servers later in the text.

To illustrate this approach we consider writing a program to model a simple embedded system. Embedded systems typically control environments. You find them in cars, aeroplanes and modern buildings. We consider a control system that takes readings from sensors and carries out some action based on a reading. Typically, such a system would be periodic in nature, i.e. it must take a reading every fixed number of units of time. Suppose we have two controls modeled by controlA and controlB and that controlA must do a reading every 20 seconds and controlB every 35 seconds. To write a single threaded solution to such a problem is awkward in the extreme. However, using threads each task can be implemented as a separate thread that can monitor its sensor in its own period of time. The following code fragment offers a sketch of a possible solution. Two threads called ControllerA and ControllerB are started by main. Each thread is written so that it periodically reads and processes its own sensor in its own time frame.

```
// declarations and prototypes here
void main(){
     // Create threads for both controllers
    new Thread(ControllerA()).start();
    new Thread(ControllerB()).start();
}

//code for controllerA
ControllerA (){
  while ( true){
     //take reading and process
    sleep(20);
  }
}
//code for controllerB
ControllerB (){
  while ( true ){
    //take reading and process
    sleep(35)
  }
}
```

Similarly, aperiodic tasks, i.e. tasks that respond asynchronously to events, can be implemented with threads. An alarm thread can be implemented so that it sleeps waiting on an event. When the alarm it is sleeping on sounds it immediately wakes up and deals with it.

Another important use of concurrent systems is that they make systems more **robust**. In a situation where multiple threads are monitoring an environment if one fails then the other threads don't necessarily fail with it. This is particularly relevant in situations where the software system is maintaining the environment. In many real-time systems the computer system is managing the environment and it becomes critical that the whole system does not fail when a single unit fails.

Problems with Concurrency

Writing concurrent programs is wonderful if each thread you write can always do its own thing and does not have to share things with other threads. However, in the real world this is not possible and in the case of concurrent threads lots of things that we take for granted in a single threaded universe no longer hold. In what follows we consider some of the major difficulties that arise in writing correct concurrent programs. If you do not grasp the full significance of the issues raised here do not worry because we will come back to them again and again as we proceed to develop an understanding of concurrent programming.

Software Testing

In a single-threaded world software testing usually involves designing test input data and then checking that the resulting output from the program matches the required output. If a mismatch occurs then you try to find where the error occurs by doing a trace of the code. This is possible because single threaded programs are *deterministic*. Given a particular input it is possible to clearly follow its sequential path through the code. This simplifies *bug* fixing somewhat. You can find the point where the error occurs, fix it and then test your program again with the same set of data. However, multi-threaded systems are *non-deterministic*. When multiple threads combine to solve a given problem there is no longer a single sequential traceable path through the code. Each time you execute a concurrent program the threads may combine or execute in completely different sequences. This fact makes it very difficult to use testing to check their correctness and also to fix *bugs* when they occur. Another consequence of this *non-determinism* is that a concurrent system may behave perfectly correctly for a long period of time, even for years, and then suddenly fail.

The reason is that the particular sequence combination just had never occurred before. This poses a big problem for programmers and makes concurrent programming a very difficult paradigm to master.

Sharing Resources and Race Conditions

When concurrent processes share resources such as screens, global variables, files, etc, problems can arise. Suppose two processes write text to the screen at the same time. In this case the text will be unreadable.

Problems like this one, where one process interferes with another at a critical moment, are known as **race conditions**. An example from programming: two processes adding nodes to the same linked. Even less obvious, two threads updating the same memory variable. Consider the case of sharing a single integer value x, that has an initial value of 1, where one thread executes `x = x + 1` and the other `x = x – 1`. One would expect the final value of x to be 1. However, we will find that sometimes it is 0 or 2. If the processor performs a context switch in the middle of the increment or decrement strange things happen. If the assignment `x = x + 1` is treated as an atomic action then the result will always be 1. But since this cannot always be relied on we need to devise ways to protect this type of shared action so that it becomes "atomic".

Sharing access to resources is difficult and requires certain protocols to be put in place so that one process does not interfere with another. Synchronizing access to a shared resource means that when two processes try to access it one waits for the other to finish. The area of code where the synchronization is necessary is called a *critical section*. The idea is that no two processes can be in their *critical sections* at the same time. Ensuring this is called **mutual exclusion**. For example, we might code access to a common screen as follows:

Thread A:

```
 while (true){
    wait until the screen is free and then signal that I have it
    // entry protocol
      System.out.print("  .....");
      signal that I have finished with the screen
      // exit protocol
    }
```

```
}

Thread B:
  while (true){
      wait until the screen is free and then signal that I have it
       // entry protocol
        System.out.print("  .....");
        signal that I have finished with the screen
        // exit protocol
      }
}
```

For parallel processes to co-operate correctly and efficiently using shared data we need four conditions to hold:

- no two processes may be simultaneously inside their critical sections;
- no assumptions may be made about the speeds or number of CPUs;
- no process running outside its critical section may block other processes;
- no process should have to wait forever to enter its critical section.

Deadlock and Starvation

Suppose you have two processes A and B. A tries to print a large tape file. It requests the printer and gets it. But before it can access the tape process B requests it. Now B requests the printer. In this scenario A is waiting for the tape to proceed and B is waiting for the printer to proceed. They are said to be **deadlocked**.

Def: Deadlock

> *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.* (Tanenbaum)

It has been shown (Coffman et al) that four conditions must hold for deadlock to take place:

1. Mutual exclusion condition: each resource is assigned to exactly one process or is available
2. Hold and wait condition: processes currently holding resources granted earlier can request new resources;
3. No pre-emption condition: resources previously granted cannot be taken away from a process - only the process holding them can release them;
4. Circular wait condition: there must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

All four conditions must be present for deadlock to occur. If one of them is absent, no deadlock is possible.

In general, four strategies are used for dealing with deadlocks:

1. Just ignore them altogether;
2. Detection and recovery;
3. Dynamic avoidance by careful resource allocation;
4. Prevention, by structurally negating one of the four necessary conditions.

Starvation

Starvation occurs when a process cannot gain access to required resources. It can happen for a number of reasons. Consider the case where two processes A and B have different priority levels. Suppose A has a higher priority than B. If A gains access to a resource required by B it could hold it and starve B. Another example might be taken from the Dining Philosophers problem:

Five philosophers are seated around a table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each plate is a fork. The life of a philosopher consists of alternate periods of eating and thinking. To eat a philosopher requires the fork to the right and left. Now suppose each of the five philosophers pick up the fork to their right they will all starve unless some are prepared to lay down their fork and wait!

Livelock

In sequential programs we are all familiar with infinite loops - the program just goes off and never comes back! This can also happen in parallel systems where the system is communication internally and fails to respond to external stimuli. From a user's perspective livelock and deadlock appear similar but are in fact different things.

History of Concurrent Programming

It may appear that the concurrent paradigm is new in the world of programming because it now gets so much attention. However, this is not the case. In fact, like all the other paradigms we studied, it has been around since the very beginning, certainly by 1960. There are a number of reasons for this. Firstly, it was realized very early on that peripheral devices were magnitudes slower than the central processing unit and, hence, had major consequences for optimizing the work rate of the CPU. This lead to the separation of tasks and peripheral devices were de-coupled from the processor. When they needed attention they simply signaled the processor or the processor polled the device every so often. The main point is that the processor could execute other processes while the peripheral devices were working. A key factor in all of this of course is cost. Machines cost at least a million dollars at the time and it was imperative to keep them busy. Hansen points out that Kilburn and Howarth *used interrupts to simulate concurrent execution of several programs on the Atlas computer* in 1961. The second development leading to the emergence of concurrent programming was the design of multi-user operating systems. All of these systems were at this time written in Assembly language and it was not until 1972 that the C language was developed by Ritchie and used by both Thompson and Ritchie to re-implement the Unix operating system originally written in assembler on a PDP-7 machine.

Hansen points out that:

> *In the mid 1960s computer scientists took the first steps towards a deeper understanding of concurrent programming. In less than fifteen years, they discovered fundamental concepts, expressed them by programming notation, included them in programming languages, and used these languages to write model operating systems. In the 1970s the new programming concepts were used to write the first concise textbooks on the principles of operating systems and concurrent programming.*
>
> *The development of concurrent programming was originally motivated by the desire to develop reliable operating systems. From the beginning, however, it was recognized that principles of concurrent programming "have a general utility that goes beyond operating systems"—they apply to* any *form of parallel computing.*
>
> *(The Origin of Concurrent Programming, Hansen 2002)*

A key figure in the design of concurrent systems was Edsger Dijkstra and his paper, *Cooperating Sequential Processes(1965),* is a seminal work in the field. In this paper he laid the foundations for abstract concurrent programming, introduced the notion of a semaphore that can be used to implement mutual exclusion, solved the bounded buffer problem using semaphores that allow processes to communicate with each other in a controlled manner. He also provided a solution for deadlock prevention known as the *banker's algorithm* and proposed extending the *Algol* language with the parallel statement `parbegin s1; s2; … sn parend`.

In 1971, Tony Hoare published *Towards a Theory of Parallel Programming* in which he proposed extending programming languages with abstract features for parallel programming. He introduced statements for parallel execution of processes, resource sharing, critical regions and conditional critical regions. Also in 1971 Hansen wrote the first comprehensive textbook on operating system principles in which he *defined operating system concepts by abstract algorithms written in Pascal extended with a notation for structured multiprogramming. My (unimplemented) programming notation included concurrent statements, semaphores, conditional critical regions, message buffers, and monitors. These programming concepts are now discussed in all operating system texts (The Origin of Concurrent Programming, Hansen 2002).* In 1973 Hansen introduced the idea of shared classes and used them to write a solution to the bounded buffer problem. The idea of a shared class was developed further by Hoare, in a paper on Monitors, in 1973. We state his solution here because you will see similarities to the solution provided later using condition variables in Java.

```
bounded buffer: monitor
    begin buffer: array 0..N–1 of portion;
    lastpointer: 0..N–1;
    count: 0..N;
    nonempty, nonfull: condition;
procedure append(x: portion); begin
    if count = N then nonfull.wait;
    note 0 ≤ count < N;
    buffer[lastpointer] := x;
     lastpointer := lastpointer ⊕ 1;
    count := count + 1;
    nonempty.signal
end append;
procedure remove(result x: portion);
begin
    if count = 0 then nonempty.wait;
```

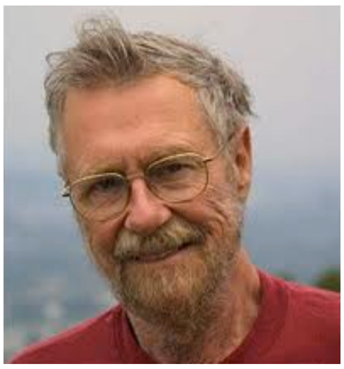
```
        note 0 < count ≤ N;
        x := buffer[lastpoint ⊖ count];
         count := count – 1;
        nonfull.signal
    end remove;
        count := 0; lastpointer := 0
    end bounded buffer;
```

In 1975 Hansen together with Hartman implemented a compiler for the new language *Concurrent Pascal* and Hansen published a paper on it. This compiler generated platform independent code, which was executed by a small kernel written in assembly language. This language incorporated all the abstract ideas about concurrent programming developed in the 1960's and early 1970's . To quote Hansen:

Concurrent Pascal extends Pascal with abstract data types known as processes, monitors, and classes.

Note: Twenty years later, the designers of the Java language resurrected the idea of platform independent parallel programming. Hansen comments *that unfortunately, they replaced the secure monitor concept of Concurrent Pascal with insecure shortcuts.* **End note**.

Biographical Note



Edsger Dijkstra(1930 – 2002) received a Turing Award in 1972 for his contributions to the development of programming languages. During his lifetime Dijkstra contributed to all fields of computing. He worked on the design of operating systems, invented the semaphore used to synchronize access to shared resources, designed programming languages and wrote many very important algorithms. To-day when you use mapping software to plot a journey between two cities behind the scenes the application uses Dijkstra's shortest path algorithm to calculate the optimal journey in terms of distance. From the 1970's on he developed the theory of axiomatic semantics and applied it to the construction of programs. Dijkstra's methodology was to construct programs and their proofs in parallel. One starts with a mathematical *specification* of what a program is supposed to do and applies mathematical transformations to the specification until it is turned into a program that can be executed. The resulting program is then known to be *correct by construction.* Dijkstra had an unusual method of writing. All his papers (numbered *EWD0, EWD1, ..*) were written free hand with a fountain pen in black ink. They were then copied and distributed to a number of people who in turn copied them and distributed them to more people. More than 1300 papers are now available on the Web.

Tony Hoare (1934 -) is probably best known to students' world wide for his algorithm Quicksort that he developed in 1960 to sort a sequence of values in optimal time. He also developed Hoare logic for verifying the correctness of programs and created the notation CSP (communicating sequential processes) that specifies interactions between concurrent and parallel processes. He received a Turing Award in 1980 for his fundamental contributions to the definition and design of programming languages and in 1982 he was elected Fellow of the Royal Society. In 1977 he became Professor of Computer Science at Oxford University and on retiring from this position in 1999 he became a researcher with Microsoft. Tony Hoare has devoted his life to thinking about software systems. Here are some of his thoughts:

> *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*
>
> *Almost anything in software can be implemented, sold, and even used given enough determination. There is nothing a mere scientist can say that will stand against the flood of a hundred million dollars. But there is one quality that cannot be purchased in this way — and that is reliability. The price of reliability is the pursuit of the utmost simplicity. It is a price which the very rich find most hard to pay.*
>
> *My original postulate, which I have been pursuing as a scientist all my life, is that one uses the criteria of correctness as a means of converging on a decent programming language design—one which doesn't set traps for its users, and ones in which the different components of the program correspond clearly to different components of its specification, so you can reason compositionally about it.*

Per Brinch Hansen(1938-2007) graduated as an Electrical Engineer from the Technical University of Denmark without any knowledge of programming. The discipline had not been invented at the time! (In the photo he is 21.) He spent the next forty years working to develop computing in both industry and academia. In his early years he worked on the RC 4000 min-computer and developed a multi-programming operating system for it. In the early 1970's he wrote the first textbook on operating systems and extended the programming language Pascal by adding concurrent features. This new language was called Concurrent Pascal. The main new feature was the introduction of monitors that combine synchronization procedures with the shared variables upon which they operate. He also wrote the first textbook on concurrent programming called *The Architecture of Concurrent Programs*. In the early 1970's he emigrated to the United states to take up a position in Carnegie Mellon University. He subsequently held a number of different posts in California and finally became Professor of Computing in Syracuse University, New York state. He also developed a number of different concurrent programming languages for writing parallel programs and used them to write operating systems. The most notable of these were called Edison and Joyce. Unusually for a computer scientist he published an autobiography in 2004 entitled A Programmer's Story. He famously said:

> *Programming is the art of writing essays in crystal clear prose and making them executable.*

Reading List

The following titles are relevant to the material covered in the lecture course. The most relevant of these is that by Brian Goetz.

Doug Lea. *Concurrent Programming in Java*, Second Edition. Addison-Wesley, 2000

Andrew Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley & Sons, 2004.

Brian Goetz. *Java Concurrency in Practice*. Addison-Wesley, 2006.

There are also a number of articles by Brian Goetz under the title *Java Theory and Practice* on the web.

Burns Alan, Davies Geoff Concurrent Programming, Addison-Wesley 1993

Haller & Summers. Actors in Scala. Artima Press, 2012

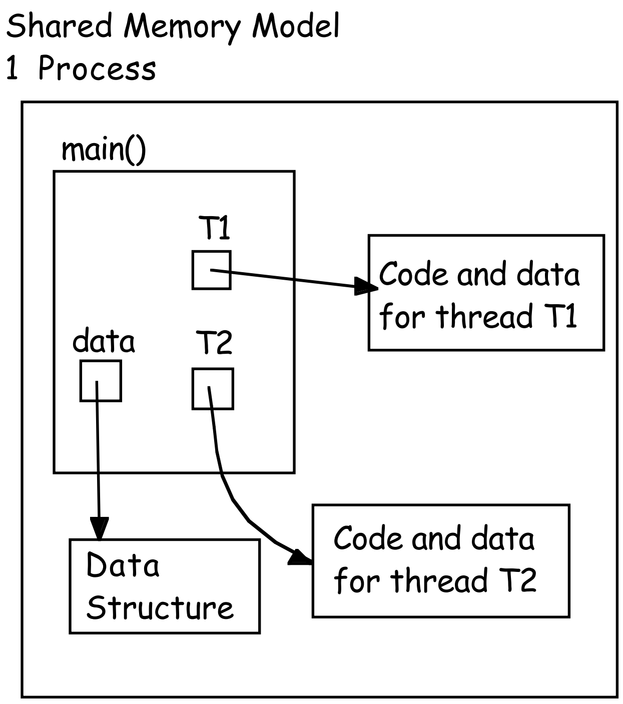
Programming with Threads

Under the Windows Operating system a process is a kernel object that characterizes a 4 GB address space within which threads execute. A process itself is inert. To work it must have a primary thread. This thread is created by the operating system when the process is created and initialized. The primary thread enters the program code through one of two functions - `main()` in the case of console applications and `winmain()` in the case of Win32 GUI based applications. After the process is initialized, the primary thread starts to execute. The run time startup code is set up so that returning from the application causes `ExitProcess` to be called and the program exits. During the execution of a process it typically passes through a number of different states. Process state transformations occur because of pre-defined rules and are controlled by the scheduler. The most common states are listed in the table below.

State	**Description**
Running	Process is currently executing on the cpu
Ready	Process is ready to run waiting on a ready to run queue
Blocked or Sleeping	Process is blocked waiting on an event to occur
Standby	Process is selected to run from the ready queue - only one process can be in this state at the one time
idle/new	Initial state of process
suspend-ready	Process temporarily removed from the ready queue
Zombie	Process has finished executing and the resources de-allocated but has not returned its exit code to its parent

A *thread* is defined as a path of code execution. It is a *lightweight process*. Each thread includes a set of instructions, related CPU register values, and a stack. A thread runs in the address space of its process and uses the resources allocated to its process. The context of a thread consists of only a stack, a register set and a priority. It does not have an address space. The text of a thread is contained in the text segment of its process. Everything that a thread needs to function is supplied by the process. All threads have a thread ID, a set of registers that define the state of the thread, its priority and its stack. This information gives each thread a separate identity. Just think of a thread as a sequence of byte code instructions executed by the Java virtual machine.

The following diagram provides a pictorial representation of a single process with multiple threads. It captures a model of what is called **Shared Memory Multi-Threaded Concurrency**. In this model we have a single process with a main thread. The code of main contains references, T1 and T2, to two threads and also a data variable that references some data structure. In this model the threads are independent and do not share any data. The process contains three threads, main, T1 and T2. Each of these threads compete for access to the CPU. The order of execution is controlled by the underlying scheduler and is independent of the threads themselves, unless programmed specifically to execute in some given order. Because three threads are executing concurrently the sequence of interleaving is non-deterministic.



In Java multiple threads have the following properties:

- Each thread begins at a pre-defined location. For one thread of the threads that location is main() method;
- Each thread executes its code independently of any other thread;
- Co-operation between threads is possible using pre-defined mechanisms;
- Threads run concurrently - the order of execution is determined by the scheduler - the two most common policies are pre-emptive Round Robin and Winner Takes All;
- Each thread has its own local variables that are private;

- Objects and their instance variables can be shared between threads - static variables are automatically shared between all threads in a process.

Note

In pre-emptive Round Robin, each process gets the processor for a fixed time-slice or until it executes a blocking operation (such as sleep() or wait() or join()). In Winner Takes All, the process that currently has the CPU retains it until it executes a blocking operation.

Creating Threads

In Java there are two slightly different ways to write threads. One approach is to use inheritance and extend the Thread class with your own class. A second approach is to implement what is termed the Runnable interface. This class will define the code for the new class but must be wrapped in a Thread class before it can be executed. In what follows we will give examples of both approaches.

We begin by giving an example of a thread by extending class Thread and overriding run(). Any class that extends the Thread class must override the run() method. This method is public, has no parameters and cannot return a value. It's signature is public void run(). A template for writing threads is:

```
class <TName> extends Thread{
    // local variables declared here
    public <TName>(param list){
        //initialization code here
    }
    public void run(){
        //code for thread here
    }
}
```

An inherited Thread class will have a constructor that will be used to construct an instance of the class in the normal way. This constructor can take arguments that can be used to initialize variables that are private to the thread instance. A thread instance encapsulates data in the normal way and should be written so that the rules governing encapsulation are preserved. Any reference variables passed through the constructor provide the thread with access to the external world. This provides a way to allow threads to share data and we will discuss this issue below. However, it breaks the rule on encapsulation for objects because it allows potential access to other threads that may modify the shared data. This has important consequences when writing concurrent threaded applications because threads will now have to obey rules governing sharing of data. We will discuss this topic in detail later in the text.

For now we concentrate on writing a simple thread that will print a message on the screen a given number of times. The message to print and the number of times to print it are passed as arguments to the MessagePrinter constructor. The run method uses a for loop to print the given message n times.

```
class MessagePrinter extends Thread{
   private String message;
   private int n;
   MessagePrinter(String m, int p){
      message = m; n = p;
   }
   public void run() {
       for(int k = 0; k < n; k++)
          System.out.println(message);
    }
 }
```

Now that we have a thread class we need to create an instance of the class and execute it. This we do as part of main(). Threads can be created from other existing threads but for now we restrict ourselves to creating them from the main thread. The variable p references the instance of the MessagePrinter that has two arguments – a message and the number of times to print it. Then the method start(), inherited from the Thread class, is invoked to schedule the thread instance to run. This method simply requests the scheduler to make the thread *runnable*. Remember it has to compete with main() for access to the CPU. When the thread actually executes it begins printing its message. It may or may not get to complete the task on its first visit. It depends on the scheduler.

```
   public static void main(String[] args) {
      Thread p =  new MessagePrinter("Happy days are here again", 10);
      p.start();
  }
```

The second way to write threads is to implement the interface **Runnable**. Again, this involves implementing a public **run()** method that has no parameters or return value. The only change required is to amend the first line.

```
   class MessagePrinter implements Runnable{
      private String message;
      private int n;
      MessagePrinter(String m, int p){
          message = m; n = p;
      }
      public void run() {
         for(int k = 0; k < n; k++)
            System.out.println(message);
      }
   }
```

However, when we want to create a thread in main() we have to do it in two stages. Firstly, we create an instance of the Runnable class and secondly we create a new thread that takes an instance of Runnable as argument. The variable p references the MesagePrinter that is an instance of Runnable. This variable is then passed to the Thread class as an argument. Then the thread t is executed using start().

```
   public static void main(String[] args) {
      Runnable p =  new MessagePrinter("Happy days are here again", 10);
      Thread t = new Thread(p).
      t.start();
   }
```

Writing it this way is a little convoluted and it is possible to avoid keeping a reference to either the Runnable instance or indeed the thread by using the following line of code.

```
   new Thread(new MessagePrinter("Happy days are here again", 10)).start();
```

Of course, if you take this approach you no longer have a handle on the thread or the runnable instance. If you don't need any reference variables then this is fine. But much of the time we will need a thread reference.

Note that in the two cases `main()` continues after initiating its child thread. Although it immediately returns, the child thread continues to run. If the parent or any child thread executes `System.exit()` then all threads die.

Both approaches to coding threads are relevant and there are situations where it is necessary to write threads by implementing the `Runnable` interface. We have to use this approach when we use what are called `ThreadPools`. See below for details. For now, however, we will use inheritance to implement threads.

Thread Methods

You can think of a thread as an *active object* to distinguish it from a normal object that communicates with the outside world through message passing. A normal object only executes when one of its methods is invoked through a reference variable one by some thread, e.g. main. Threads are *active objects* in the sense that every time a thread `start`s its `run` method is automatically executed by the runtime scheduler on the system. In effect it is an executing program. However, the Thread class itself provides a public interface whose methods can be invoked through a thread reference variable or directly in the code of the thread itself through the self-reference `this`. In this section we look at some of the most important of these methods.

Putting a Thread to Sleep

It is possible to force a thread to sleep for a period of time, usually measured in milliseconds, and then continue. To do this use:

```
try{
  this.sleep(<period>);
}catch(InterruptedException e){..}
```

To demonstrate this we-write the message printer thread above so that it prints a message every 100 milliseconds.

```
class MessagePrinter extends Thread {
   private String message;
   private int n;
   MessagePrinter(String m, int p){
      message = m; n = p;
   }
   public void run() {
      int k = 0;
      while (k < n) {
          System.out.println(message);
          try{
             Thread.sleep(100);
          }catch(InterruptedException e){..}
          k++
      }
    }
 }
```

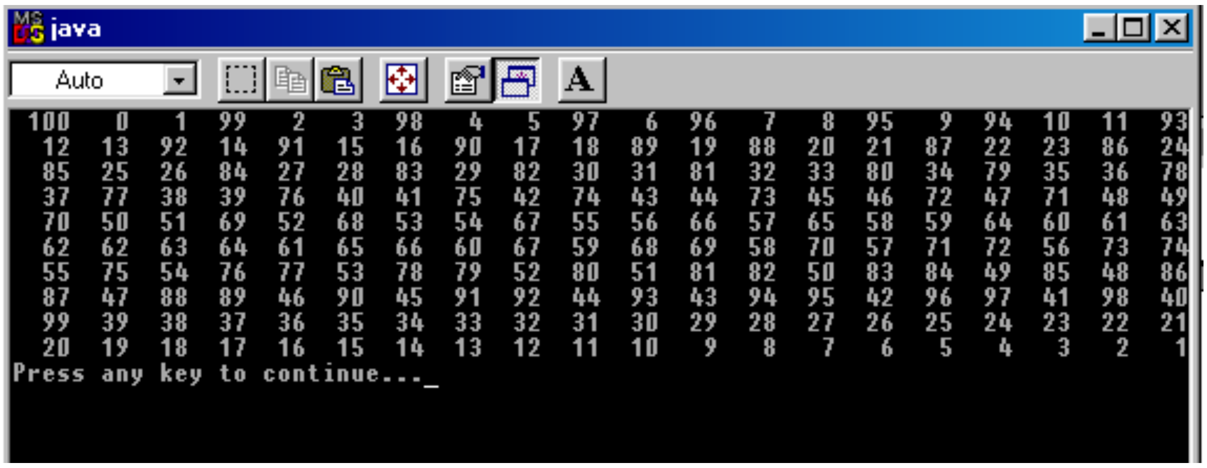
As a second example we use two counting threads – one incrementing and the other decrementing - where both threads print their output on a share screen. Each thread prints the next value in its sequence and then sleeps for a prescribed period of time.

```
class countDown extends Thread{
    private int n;
    private int pause; // in milliseconds
    public countDown(int p, int t){
      n = p; pause = t;
    }
    public void run() {
      for (int j = n; j > 0; j = j - 1){
        System.out.print(j+" ");
         try {
            Thread.sleep(pause);
         } catch(InterruptedException e) { }
      }
  }
}

class countUp extends Thread{
  private int n;
  private int pause; // in milliseconds
  public countUp(int p, int t){
      n = p; pause = t;
  }
  public void run() {
      for (int j = 0;  j < n; j = j + 1) {
        System.out.print(j+" ");
         try {
            Thread.sleep(pause);
         } catch(InterruptedException e) {return;}
      }
  }
}
```

The test program, `counterTest`, given below executes two instances of the threads concurrently.

```
class counterTest{
    public static void main(String args[]){
        new countUp(100, 300).start();
        new countDown(100, 500).start();
    }
}
```

The output from a test run was:



```
   100     0     1    99     2     3    98     4     5    97     6    96     7     8    95     9    94    10    11    93
    12    13    92    14    91    15    16    90    17    18    89    19    88    20    21    87    22    23    86    24
    85    25    26    84    27    28    83    29    82    30    31    81    32    33    80    34    79    35    36    78
    37    77    38    39    76    40    41    75    42    74    43    44    73    45    46    72    47    71    48    49
    70    50    51    69    52    68    53    54    67    55    56    66    57    65    58    59    64    60    61    63
    62    62    63    64    61    65    66    60    67    59    68    69    58    70    57    71    72    56    73    74
    55    75    54    76    77    53    78    79    52    80    51    81    82    50    83    84    49    85    48    86
    87    47    88    89    46    90    45    91    92    44    93    43    94    95    42    96    97    41    98    40
    99    39    38    37    36    35    34    33    32    31    30    29    28    27    26    25    24    23    22    21
    20    19    18    17    16    15    14    13    12    11    10     9     8     7     6     5     4     3     2     1
Press any key to continue..._
```

Can you explain why the output is not as one might expect? For instance, sometimes countUp prints two values and at other times only one. Why is this the case? You also note that countDown executed first even though countUp started before it. Why is this the case?

Waiting for a Thread to Finish

A thread can delay itself until another thread t has terminated, by executing

```
try {
  t.join();
}catch(InterruptedException e) {}
```

An example, we re-write the MessagePrinter so that main() waits for the printer thread to complete and then prints its own message on the screen.

```
class MessagePrinterTest{
   public static void main(String[] args) {
      Thread t = new MessagePrinter("Happy days are here again", 10);
      t.start();
      try{
         t.join();
      }
      catch(InterruptedException e){}
     System.out.println("All messages are printed");
```

```
    }
}
```

Naming Threads

To give a thread a particular name use `setName(String name)`. To retrieve the name of a thread use `getName(String name)`. In both cases a thread reference variable is required.

Thread identity

Every thread has an identity number and it can be accessed by the following method in class Thread:

```
    long getId()
```

For `t` a reference to a thread object, t.getId() returns a unique identifying positive integer associated with thread t. Thread ID's are unique, but may be recycled after the thread has terminated.

Current Thread

It is possible for a thread to acquire a self-reference. The static method `Thread.currentThread()` yields a handle on the current thread. In this way an executing thread can get a reference to itself. The following example illustrates its use by printing the name of the thread that is currently running at any given instant in time followed by the thread's identity number. The thread SelfRefT gets a reference to itself and then uses this reference to get its name and id number. The main thread creates 10 instances of SelfRefT and executes them.

```
public class SelfRefTest{
    public static void main(String args[]){
        for(int j = 0; j < 10; j = j + 1){
            Thread tst = new SelfRefT();
            tst.setName("No:" + j);
            tst.start();
        }
    }
}
class  SelfRefT extends Thread {
    private Thread t;
    public void run() {
        t = Thread.currentThread();
        System.out.println(t.getName() +" Id:"+t.getId());
    }
}
```

A sample output from the program is:

```
    No:1 Id:11
    No:0 Id:10
    No:3 Id:13
    No:4 Id:14
    No:2 Id:12
    No:5 Id:15
    No:6 Id:16
    No:7 Id:17
    No:8 Id:18
    No:9 Id:19
```

Voluntarily giving up the Processor

A thread invoking yield indicates that it is willing to give up the processor, i.e. it is willing to yield the processor to the thread at the head of the ready to run queue. The output in the following program should alternate forever.

```
class AlternatingTest{
    public static void main(String args[]){
        new MPrinter("It's Joe here")).start();
        new MPrinter("It's Donal here")).start();
    }
}
class MPrinter extends Thread{
    String message;
    public MPrinter(String m){message = m;}
    public void run(){
        while(true){
            System.out.println(message);
            Thread.yield();
        }
    }
}
```

Thread States
A thread executes in the context of the virtual machine. During its lifetime it may pass through a number of different states because it shares the processor with other threads including main. A thread can only be in one state at any given time and it changes state under a given set of pre-defined rules. The list of possible states is given in the following table. The names of the thread states form an enumerated class and, hence, are written with capital letters.

Name	**State Description**
NEW	A thread that has not yet started
RUNNABLE	A thread executing in the Java virtual machine. A thread in this state may be waiting for resources from the underlying operating system. For example, it might be on a ready-to-run queue waiting to be allocated a processor.
BLOCKED	A thread that is blocked waiting for a monitor lock. (We will discuss this state when we come to dealing with sharing resources.)
WAITING	A thread that is waiting for another thread to perform a particular action. (We will discuss this state when we come to dealing with sharing resources.)
TIMED_WAITING	A thread that is waiting for another thread to perform an action, for up to a specified waiting time. For example, a thread is in this state when it is sleeping for a given period of time.
TERMINATED	A thread that has exited

The following code fragment will print the list of thread states on the screen.

```
    for (Thread.State c : Thread.State.values())  System.out.println(c);
```

The static method Thread.State.values() returns an array containing the constants of the enum type, in the order they are declared.

The following program lists the different states of the thread T1 that it creates and ultimately waits to terminate. It shows that the thread goes through 4 states: NEW followed by RUNNABLE, followed by TIMED_WAITING, followed by TERMINATED.

```
public class StateTest{
  public static void main(String args[]){
    Thread t = new T1();
    System.out.println(t.getState());
    t.start();
    System.out.println(t.getState());
    try{
     Thread.sleep(1000);
    }catch(InterruptedException e){}
    System.out.println(t.getState());
    try{
     t.join();
    }catch(InterruptedException e){}
    System.out.println(t.getState());
  }
}
class T1 extends Thread{
  public void run(){
    try{
     this.sleep(4000);
    }catch(InterruptedException e){}
  }
}
```

Thread Priority and Scheduling

Each thread that runs in a Java virtual machine is given a priority value. Threads with higher priority values are usually given precedence over threads with lower priority values. When threads are created they are given the same priority as the thread that constructed them. However, the underlying operating system can influence the order in which threads are executed. In Java there are 10 different priority values. The range is 1..10. There are three priority constants. These are Thread.MAX_PRIORITY, Thread.NORM_PRIORITY and Thread.MIN_PRIORITY. These have values 10, 5 and 1, respectively.

In the Java runtime system, the **preemptive scheduling** algorithm is applied. If at execution time a thread with a higher priority than all other RUNNABLE threads then the runtime system usually chooses the new **higher priority** thread for execution. On the other hand, if two threads of the same priority are waiting to be executed by the CPU then the **round-robin** algorithm is applied in which case the scheduler chooses one of them to run according to their round of **time-slice**. It should be pointed out that there are no guarantees that changing the priority of a thread will guarantee preferential treatment.

Problems can arise when there is a conflict between the range of priority values in Java and those found in the underlying operating system. For example, Solaris has 2^{31} different priority values and Windows has only 7. It could happen that a Java priority value of 8 is given the same priority as a thread of priority 10 in Windows. However, we can be pretty sure that the three constants will map to different values so we shall use these in the following discussion.

It is possible to set the priority of a thread using `setPriority(int p), p>=1 && p <=10`. This can be done before the thread starts or while it is executing. It is also possible to inspect the priority of a thread using `getPriority()`.

Daemon Threads

In Java there are user threads and daemon threads. The only difference between them is that the JVM automatically kills daemon threads when there are no more user threads running. Daemon threads are useful for background processing. To set a daemon thread use: `t.setDaemon(true);`

Stopping a Thread

Threads normally terminate when the `run` method completes or if at any point in the code of a `run` method a `return` is executed. However, sometimes we want to get a control program to terminate a thread that is currently running. Earlier versions of Java had a method `stop()` to do this but it has been deprecated because it caused deadlock problems. Instead of using this we can use a local Boolean value to control termination. Initially this value is set to `true` and a public method is added to the thread subclass that when invoked sets this value to `false`. To illustrate this we write a thread that models an egg timer by printing the time in seconds elapsed since it starts. An instance of the egg timer can be stopped at any time after it starts. It defaults to a time of 30 seconds. The public method terminate can be invoked by a control program to stop the egg timer at any moment after it starts. (Note that this method cannot be called stop() because it is declared final in the Thread class).

```
class EggTimer extends Thread{
  private int et = 0;
  private volatile  boolean go = true;
  public void run(){
    while(et < 30 && go){
      try{
        this.sleep(1000); //1000 milliseconds == 1 sec
      }catch(InterruptedException e){}
      et++;
      System.out.print(et+" ");
    }
  }
  public void terminate(){ go = false; }
}
```

A program that tests the EggTimer is given. It simply starts an instance of EggTimer and prompts the user to stop it by pressing the enter key.

```
import java.util.Scanner;
public class EggTimerTest{
  public static void main(String args[]){
    Scanner in = new Scanner(System.in);
    EggTimer et = new EggTimer();
    System.out.println("Press enter to stop egg timer");
    et.start();
    in.nextLine();
    et.terminate();
  }
}
```

Note that to invoke the method terminate() the reference variable et must be of type EggTimer and not simply of type Thread. This is necessary because of the meaning of inheritance. An EggTimer is an instance of a Thread but a Thread is not an instance of an EggTimer.

Exercise 1

Question 1

Write a thread that tosses a coin 1000 times and computes the frequency of heads and tails.

Question 2

The method void join (long) throws InterruptedException can be used to force the invoking thread to wait on one or more threads for a given period of time. Write a program that waits 2 seconds for a thread to complete and then stops it if it hasn't.

Question 3

Write a thread that continuously prints a message every 100 milliseconds while it is still alive. It can be terminated at any time by a control program.

Question 4

Write a program that prints out the prime numbers is ascending order, and stops when a user presses the return key.

Question 5

Write thread that takes an integer array as argument and computes and prints the sum of the elements in the array.

Question 6

Design an experiment to show that pre-emption occurs.

key.

Question 7

Design a pseudo round robin scheduler that manages a number of threads under its control.

Hint

A suggestion is to set the priority of the scheduler to max and use the two other levels to manage threads under its control. Only one thread is at the normal priority level at any one time. The scheduler sleeps for a timeslice and then forces a context switch because it pre-empts the currently executing thread. It then changes its priority to low and selects the next thread to run by setting its priority to normal. It then puts itself to sleep to allow it execute.

Returning the Result of a Thread Computation

There are many situations where we will want to employ threads to carry out particular tasks and return results. The **run** method cannot be modified and, hence, must always be **void**. Therefore, it cannot return a result to the calling thread. Threads that compute results and return them are often referred to as Asynchronous Procedures. Strictly speaking an asynchronous call is one that returns as soon as the procedure has been invoked, with the result that the caller and the invocation of the procedure proceed in parallel. (This is how concurrency is supported in the C programming language.) An example of an asynchronous thread might be one to sum the elements in an array and return the sum to the calling thread. One way to implement such a procedure in Java is to extend the Thread class and add a public method, say **getResult()**, that returns the result of the computation. A template for an asynchronous thread is given. Note that the calling thread will have to wait until the thread completes to retrieve the result. (When we discuss Futures later in the text we will examine an alternative model to this one. With the use of Futures the calling thread can examine the result a computation anytime in the future.) In the template a result variable is initialized at the end of the run method and its value is passed back to the caller through the **getResult()** method.

```
class AsyncFunc extends Thread {
   private … result;
   public AsyncFunc(…) {
      …
   }
   public void run() {
      //do calculation;
    result = ?;
   }
   public … getResult() {
     return result;
   }
}
```

An example that uses an asynchronous function to calculate factorial n is given. The thread FacN takes a non-ngative integer, n, as argument, computes the factorial of n and assigns this value to the variable result. The method getResult() can then be invoked by the calling thread. This is demonstrated in the main program given.

```
import java.util.Scanner;
class FactorialTest{
    public static void main(String args[]){
        Scanner in = new Scanner(System.in);
        System.out.print("Enter value for n: ");
        int n = in.nextInt();
        FacN t = new FacN(n);
        t.start();
        try{
            t.join();
        }
        catch(InterruptedException e){}
        System.out.println("Fac("+n+") = "+ t.getResult());
    }
}
class FacN extends Thread{
    private int n;
    private long result;
    public FacN(int nn){
        n = nn;
    }
    public void run(){
      long fac = 1; // fac(0) = 1
      int k = 0;
      while(k < n){
        fac = fac*(k+1);
        k++;
      }
      result = fac;
    }
    public long getResult(){
      return result;
    }
}
```

An alternative approach to writing asynchronous functions that return a result to the caller is to create a separate Result class that is passed as an argument to the thread constructor. The calling thread simply creates an instance of a Result class and passes its reference variable as an argument to the thread. When the thread completes its computation it invokes a set method provided by the Result class. The caller then retrieves the result by invoking a get method. A template for this class is given.

```
class Result{
    private … result = ..; //default value
    void set(… x){result = x;}
    … get(){return result;}
}
```

The following solution uses this approach to solving the factorial n problem discussed above. In main we create an instance of the Result class and pass a reference to it to the FacN thread. Then wait for it to finish and extract the result using the get method.

```
import java.util.Scanner;
class FactorialTest1{
    public static void main(String args[]){
        Scanner in = new Scanner(System.in);
        System.out.print("Enter value for n: ");
        int n = in.nextInt();
        Result res = new Result();
        FacN t = new FacN(n, res);
        t.start();
        try{
            t.join();
        }
        catch(InterruptedException e){}
        System.out.println("Fac("+n+") = "+ res.get());

    }
}
```

```
class FacN extends Thread{
    private int n;
    private Result res;
    public FacN(int nn, Result r){
        n = nn; res = r;
    }
    public void run(){
      long fac = 1; // fac(0) = 1
      int k = 0;
      while(k < n){
        fac = fac*(k+1);
        k++;
      }
      res.set(fac);
    }
}
class Result{
    private long result = -1; //dummy value
    void set(long x){result = x;}
    long get(){return result;}
}
```

Parallelising Sequential Algorithms

In this chapter we look at exploiting the power of multi-core processing by implementing parallel algorithms that solve some standard problems. The idea is to take standard sequential algorithms and develop parallel solutions that use multiple concurrent threads to solve the problem. The motivation for this is not just simply to write concurrent solutions for their own sake but rather to see if our concurrent solutions outperform the standard single threaded ones. By dividing the work to be done between multiple threads it is certainly an expectation that our programs will deliver results in a shorter time frames. In reality this may not always be the case because there are overheads in working with threads and also the use of shared caches on multi core machines can cause delays. In particular, if variables are shared across multiple local caches and if any of them are modified then the whole collection has to be updated. This can cause unforeseen delays and should be taken into account by programmers when parallelising algorithms. There are also issues around balancing workloads between threads and distributing these loads over all threads executing in parallel. You also need to take into account whether the threads are CPU bound or I/O bound. For example, if there are only 4 cores on your machine and you divide a CPU bound task into six separate tasks employing 6 threads in the process then the threads will compete with each other for access to the CPU causing context switching to take place. This will simply slow down the overall performance of your program. Alternatively, if tasks involve some I/O then it is a good idea to employ more threads than processor cores because all threads may sleep during their lifetime waiting on an I/O event.

The plan is to explore some of the above and discuss possible ways to optimise our parallel solutions. We begin with a simple example that employs two threads to search a shared array of integer values for some given value x. To keep things simple we just use linear searching to perform the search. We assume that the reader is familiar with this algorithm. A searcher thread simply searches its own segment for the given value. A segment is delineated by a lower bound and an upper bound passed as arguments to the thread. Each thread performs a linear search on its own segment terminating when x is found or when there are no more elements level to inspect. When the search is complete the outcome of the search can be retrieved by the caller by invoking the public method `getResult()`. That is, we implement the searcher thread as an asynchronous function.

```
class Searcher extends Thread {
        private int f[];
        private int lb, ub;
        private int x;
        private boolean found;
        Searcher(int f1[], int a, int b, int x) {
           f = f1; lb = a; ub = b; this.x = x;
        }
        public void run() {
          int k = lb; found = false;
          while (k < ub && !found){
              if(f[k] == x) found = true;
              k++;
           }
        }
        boolean getResult() {
           return found;
        }
    }
```

The main program initializes the data array by assigning it random values in the range 0..99, displays the data on the screen, asks a user to enter a search value and then divides the task of doing the search between two searcher threads. It then waits for them to complete the search, retrieves the result from each thread and prints the outcome of the total search.

```
    import java.util.*;
    class LinearSearch{
        public static void main(String args[]){
            int list[] = new int[100];
            for(int j = 0; j < list.length; j++) list[j] = (int)(Math.random()*100);
            for(int y : list) System.out.print(y+" ");
            System.out.println();
            System.out.print("Enter number to search for: ");
            Scanner in = new Scanner(System.in);
            int x = in.nextInt();
            Searcher t = new Searcher(list,0,50,x);
            Searcher t1 = new Searcher(list,50,100,x);
```

```
         t.start(); t1.start();
         try{
            t.join(); t1.join();
         }
         catch(InterruptedException e){}
         boolean found = t.getResult() || t1.getResult();
         System.out.println("Found = " + found);
      }
   }
```

This solution simply demonstrates the division of labour between two tasks but it does not yield any great results in terms of performance. The size of the data set is so small that a single threaded solution would be a better option. Also the cost of setting up the threads in the first instance is probably greater than the actual cost of searching. However, if the data set had billions of elements then the average performance should be significantly faster than a single threaded solution. It should be pointed out here that there is a major design flaw in this search algorithm because if one of the threads finds x they have no way of telling the other thread to stop searching. Each thread searches its own segment separately and there is no communication between them. If x is contained in the left segment only, then every element in the right segment is checked to no avail. To communicate the threads need to share an object. To make this possible we write the class Found that has a single Boolean attribute that defaults to the value false when an instance of the class is created. There are three public methods: set() that changes the value of found to true; found() that returns the current value of found and toString that returns a string representation of the current value of found.

```
class Found{
   private boolean found = false;
   public void set(){found = true;}
   public boolean found(){return found;}
   public String toString(){return found+"";}
}
```

The main thread creates a single instance of this class and passes it’s reference to each of the threads. The code for the Searcher is amended so that it takes an argument that refers to an instance of Found and it checks it’s attribute, found(), at each iteration of the loop. If x is found then the attribute is set by the finding searcher and the change is perceived by the other searcher next time it completes an iteration of its loop. This allows it to stop searching. The code is:

```
class Searcher extends Thread {
   private int f[];
   private int lb, ub;
   private int x;
   private Found fnd;
   Searcher(int f1[], int a, int b, int x, Found fd) {
     f = f1; lb = a; ub = b; this.x = x; fnd = fd;
   }
   public void run() {
     int k = lb;
     while (k < ub && !fnd.found()){
        if(f[k] == x) fnd.set();
        k++;
     }
   }
}
```

The amended section of main() to take these changes into account is given by the code fragment:

```
    Found fnd = new Found();
    Searcher t = new Searcher(list,0,50,x, fnd);
    Searcher t1 = new Searcher(list,50,100,x,fnd);
    t.start(); t1.start();
    try{
        t.join(); t1.join();
    }
    catch(InterruptedException e){}
    System.out.println("Found = " + fnd);
```

This solution will certainly improve the overall performance of the program. You should note that sharing objects between threads can only be done under certain conditions and the study of these will be covered in the next chapter of this work. In the example given the only possibility available to threads is to check the state and modify it by assigning it a single Boolean value. This action is an atomic one and, hence, is what we call thread safe.

As a second example we consider parallelising selection sort. We know from our study of data structures that sorting is computationally expensive and that linear sorting is at best *O(N*logN)* for Quicksort. However, selection sort is computationally more expensive again and is *O(N*N)*. Writing a parallel solution in this instance should significantly reduce the time taken. The program given below creates an array of 1 million integers and uses two threads to sort the data. Each thread is given its own segment to sort and, on completion of both threads, the program merges the two sorted sub-sequences giving a fully sorted array of values. (We assume the reader knows the selection sort algorithm).

This solution performed approximately 4 times faster than a single threaded solution on a number of different architectures. For example, on an Apple Mac with an 8-core processor it took 1252 seconds for a single threaded solution to sort the data and only 316 seconds for the two threaded concurrent version below. This results concurs with the mathematical analysis because *O(N/2*N/2)* approximates to *1/4 O(N*N)*.

```
class Sorting{
  public static void main(String args[]){
    int f[] = new int[1000000]; //1 million
    int n1 = 500000;
    int n2 = 1000000;
    for(int j = 0; j < f.length; j++) f[j] = (int)(Math.random()*10000);
    Thread t = new SelectionSort(f,0,n1);
    Thread t1 = new SelectionSort(f,n1, n2);
    long startTime = System.currentTimeMillis();
    t.start();t1.start();
    try{
      t.join();t1.join();
    }
    catch(InterruptedException e){}
    //merge component parts
    int c[] = new int[f.length];
    int k = 0;int j = 0;int h = n1;
    while(j < n1 && h < n2){
      if(f[j] <= f[h]){
        c[k] = f[j];
        j++;
```

```
            }
            else{
                c[k] = f[h];
                h++;
            }
            k++;
        }
        while(j < n1){ c[k] = f[j];  k++; j++; }
        while(h < n2){c[k] = f[h]; k++; h++;}
        f = c;
        long endTime = System.currentTimeMillis();
        long runningTime = endTime-startTime;
        System.out.println(runningTime + " millisecs (" +(runningTime/1000.0) +
")");
        System.out.println();
    }
  }

class SelectionSort extends Thread{
    int dt[];
    int a, b;
    public SelectionSort(int f[], int a, int b){
              // f[a..b) to be sorted
        dt = f; this.a = a; this.b = b;
    }
    public void run(){
        for(int i = a; i < b; i++){
            int j = i; int k = i + 1;
            while(k < b){
                if(dt[j] > dt[k])
                   j = k;
                k++;
            }
            //swap dt[j] with dt[i]
            int temp = dt[i]; dt[i] = dt[j]; dt[j] = temp;
        }
    }
}
```

Distributing the Workload Fairly over Threads

The underlying hardware provides direct support for writing programs that employ multiple threads running in parallel to compute solutions to problems. Typically, a modern processor will support at least as many threads, running in parallel, as the number of cores on the processor. Therefore, a quad core processor will support 4 threads running in parallel. In general, to optimise processing times we will try to allocate the maximum number of processors available to compute results. This means distributing the workload as fairly as possible over the maximum number of threads supported by the underlying processor. Generally, introducing more threads than available processors is counter productive due to the significant system overhead in creating and managing the threads. There are occasions, however, when we might allocate more threads than available processors. For example, in situations where threads may be I/O bound for periods of time it makes sense to use more threads than available processors. The reason simply is that a thread waiting on an I/O event is sleeping on the event and is not using the processor.

Rank Sorting Example

The task is to try to optimise the rank sorting algorithm. This algorithm works by finding the rank of each element in an array and then assigning it to its rank position in a new blank copy array. The algorithm does not sort the elements in situ but rather copies elements to their ranked position in a copy array. This copy contains only the elements in the original array in a sorted permutation of the original. This, of course, means that it requires double the memory to sort the data. An outline solution is:

```
int data[] = ...;
int copy[] = new int[data.length];
for(int i = 0; i < data.length; i++){
  int rank;
  //set rank = rank of data[i]
  copy[rank] = data[i];
}
```

Calculating the rank of a given number requires iterating over the whole sequence counting the number of elements less than data[i]. The code is:

```
rank = 0;
for(int j = 0; j < data.length; j++)
```

```
        if(data[j] < data[i]) rank++;
```

The listing given below is for a program that implements this algorithm for a small sequence of unique data elements. The algorithm is *O(N*N)* and its performance is not affected by the state of the data. Therefore, its best and worst performance is always *O(N*N)*.

```
class RankSort{
  public static void main(String[] args) {
    // No duplicates allowed
    int data[] = {4,7,1,56,78,23,5,69,85,90,45,81,10};
    int copy[] = new int[data.length];
    for(int i = 0; i < data.length; i++){
      //set rank = rank of data[i] and assign it to copy[k]
      int rank = 0;
      for(int j = 0; j < data.length; j++)
          if(data[j] < data[i]) rank++;
     copy[rank] = data[i];
    }
    for (int i = 0; i < copy.length; i++)
        System.out.print(copy[i]+" ");
    System.out.println();
  }
}
```

Note that the solution provided only solves the problem for lists with no duplicates. Allowing duplicates slightly complicates the algorithm because equal values have equal rank. To allow duplicates we have to calculate the frequency of occurrence of a given element and assign all of them to the copy array starting at the rank position. The solution is given below.

```
public class RankSortDup {
    public static void main(String[] args) {
    // duplicates allowed
    int data[] = {4,7,1,56,78,23,5,69,81,90,5,5,45,81,5,10,10,10};
    int copy[] = new int[data.length];
    for(int i = 0; i < data.length; i++){
```

```
    //set rank = rank of data[i] and assign it to copy[k]
    int rank = 0; int freq = 0;
    for(int j = 0; j < data.length; j++){
      if(data[j] < data[i]) rank++;
      else if (data[j] == data[i]) freq++;
    }
    for(int j = 0; j < freq; j++)
       copy[rank+j] = data[i];
  }
  for (int i = 0; i < copy.length; i++)
      System.out.print(copy[i]+" ");
  System.out.println();
 }
}
```

End Note

An initial approach to transforming this algorithm into a parallel solution might be to allocate a thread to each element in the array. In effect this means allocating the work of the inner loop in the single threaded solution to a thread. A consequence of this is that you will have as many threads as there are elements in the array. This is an expensive overhead but one that we will pursue at this stage to see how it performs. It is also interesting to see how to manage a sequence of threads. Then we will proceed to explore alternative approaches to distributing the workload over threads to try to optimise a parallel solution.

Allocating a thread to each unique element requires n, where n equals the number of elements in the sequence, clone threads, each of which does exactly the same thing. The code for the thread simply takes a reference to the data source, a reference to the copy array and the index of the element whose rank has to be calculated. The run method computes the rank of data[i] and copies it to copy[rank]. The code is:

```
class RankerNoDup extends Thread{
 int data[]; int copy[];int i;
 public RankerNoDup(int d[], int c[], int k){
     data = d; copy = c; i = k;
 }
```

```
  public void run(){
      int rank = 0;
    for (int j = 0; j < data.length; j++)
       if (data[j] < data[i])  rank++;
      copy[rank] = data[i];
  }
}
```

The main program initializes the data array with N integer values in the range 0..N-1, shuffles the elements and then creates a Thread array. The main loop iterates over the data sequence allocating a new thread to each unique value. Once the threads are started it waits for all the threads to complete. The start time and end time of the actual sorting is recorded. It then checks that the data is indeed sorted (not included in temporal cost of the sort) and prints the time taken to do the sort.

```
class ParallelRankSort {
    static final int N = 100000;
    public static void main(String[] args) {
       // No duplicates allowed
       int data[] = new int[N];
       //Create list of unique values and shuffle them
       for(int j = 0; j < data.length;j++) data[j] = j;
       shuffleData(data);
       int copy[] = new int[data.length];
       Thread [] workers = new Thread[data.length];
       //Record start time
       long startTime = System.currentTimeMillis();
       for(int i = 0; i < data.length; i++){
          workers[i] = new RankerNoDup(data,copy,i);
          workers[i].start();
       }
       //now wait for threads to finish
       for (int k = 0; k < workers.length; k++)
      try {
         workers[k].join();
      }catch (InterruptedException e) {}
```

```
         //Record end time
         long endTime = System.currentTimeMillis();
        long runningTime = endTime-startTime;
        //check that data sorted
        data = copy;
        copy = null;
         boolean sorted = true;
         for (int i = 0; i < data.length - 1; i++)
             if(data[i] >= data[i+1]) sorted = false;
         System.out.println("Sorted list: "+sorted);
         System.out.println(runningTime + " millisecs (" +(runningTime/1000.0) + ")");
         System.out.println();
    }
    static void shuffleData(int f[]){
      for(int j = 0; j < f.length/2; j++){
          int k = (int)(Math.random()*f.length);
          int m = (int)(Math.random()*f.length);
          while(k == m) m = (int)(Math.random()*f.length);
          int temp = f[k]; f[k] = f[m]; f[m] = temp;
      }
    }
}
```

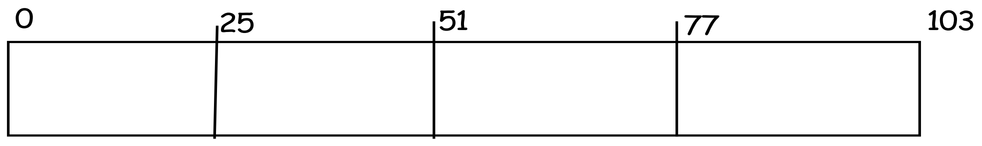
This is a complicated solution and one would hope that it would return some benefits in terms of performance. However, this may not be the case because the overhead of creating 100000 threads and managing them may use up lots of processing time. This is in fact borne out by testing. On a dual core Intel processor it took 75.4secs to sort 100000 integer values. However, a single threaded solution running on the same machine only took 72.5secs. This is not too promising. We might consider increasing the processing power of the machine. Suppose we execute it on a machine with more cores. Then will it outperform the single threaded solution? Testing it on a dual processor quad core machine (Apple Mac Pro) gave some disappointing results. The single threaded solution finished in 9.88secs but the multi-threaded solution took 13.45secs.

Next we consider developing a solution that only employs as many threads as there are processors on the machine. This means that we have to distribute the workload fairly over k threads, where k equals the number of processors available on the machine executing the parallel rank sort. To do this we need an algorithm to distribute the work evenly over the worker threads.

Dividing work on an array evenly among k processors is not always possible. Suppose there are N elements in the array and k processors available to work on it. We could take a segment size of N/k as a base. If there were 100 elements and k equals 4 then each thread would be allocated a block of 25 elements to process. However, if there were 103 elements you would end up with 3 blocks of 25 elements and 1 block of 28 elements. As the number of cores increases this final block could become quite big. The goal would be to have an algorithm that divides the data such that block sizes differ by no more than one. Such an algorithm will give an even distribution of tasks. To obtain this result we use a loop where the starting index, j, of each block is calculated iteratively by the formula (j*N)/k. The initial value of j is 0. The iteration is given by the code fragment:

```
 for(int j = 0; j <= k; j++){
   int ind = (j*N)/k;
  System.out.print(ind+" ");
}
```

Given N = 103 and k = 4, this will generate the sequence 0, 25, 51, 77, 103. This gives blocks of size 25, 26, 26, and 26.



All blocks differ in size by at most 1.

Now we can return to our task of optimising our rank sorting algorithm. To distribute the workload evenly over the array of threads we calculate the number of processors on the machine executing the program and use our algorithm above to distribute the workload evenly over them. To store the indexes we use an index array of size nThreads + 1 because we have to store the indices of both the first and last element. Each block or segment has index range index[j], index[j+1] for j in the range 0..index.length-1. We then create a Thread array that holds references to each worker thread created. Each thread is given its own block of data to process. The code is:

```
int nThreads = Runtime.getRuntime().availableProcessors();
```

```
    //assume nThreads <= N
    int index[] = new int[nThreads+1];
    for(int j = 0; j <= nThreads; j++)
      index[j] = (j*N)/nThreads;
    //segment index ranges are index[j], index[j+1]
   //Create array to reference and create worker threads
    Thread workers[] = new RankerNoDup[nThreads];
    for(int j = 0; j < nThreads; j++){
     workers[j] = new RankerNoDup(data,copy,index[j], index[j+1]);
     workers[j].start();
    }
```

Distributing the workload in this balanced way over the number of cores on the machine did yield improved results. On the dual core machine this new version sorted the list in 47.5secs, approximately half the time taken by the single threaded solution. Similarly, on the dual processor quad core machine the time was approximately 8 times faster (1.2secs as opposed to 9.9secs).

The moral of the story is that to employ multiple threads to solve a problem we cannot create large numbers of threads because the overhead involved in managing them may reduce the overall system performance. This is particularly true in the case of cpu bound tasks. Tasks that involve lots of i/o spend time sleeping and, hence, other threads can work.

The algorithm to distribute workload fairly over nThreads is, in general, as follows:

```
   int nThreads = Runtime.getRuntime().availableProcessors();
    //assume nThreads <= N
    int index[] = new int[nThreads+1];
    for(int j = 0; j <= nThreads; j++)
      index[j] = (j*N)/nThreads;
    //segment index ranges are index[j], index[j+1]
   //Create workers
    Thread workers[] = new …[nThreads];
    for(int j = 0; j < nThreads; j++){
     workers[j] = new …(…, index[j], index[j+1]);
     workers[j].start();
    }
```

The listing for this optimised rank sorting algorithm is given below:

```
class OptimisedRankSort {
   static final int N = 100000;
   public static void main(String[] args) {
     // No duplicates allowed
     int data[] = new int[N];
     //Create list of unique values and shuffle them
     for(int j = 0; j < data.length;j++) data[j] = j;
     shuffleData(data);
     int copy[] = new int[data.length];
     int nThreads = Runtime.getRuntime().availableProcessors();
     System.out.println("Number of processors = "+nThreads);
     int index[] = new int[nThreads+1];
     for(int j = 0; j <= nThreads; j++)
       index[j] = (j*N)/nThreads;
     long startTime = System.currentTimeMillis();
     Thread workers[] = new RankerNoDup[nThreads];
     for(int j = 0; j < nThreads; j++){
       workers[j] = new RankerNoDup(data,copy,index[j], index[j+1]);
       workers[j].start();
     }
     try{
       for(int j = 0; j < nThreads; j++)
        workers[j].join();
     }
     catch(InterruptedException e){}
     long endTime = System.currentTimeMillis();
     long runningTime = endTime-startTime;
     //check that data sorted
     data = copy; copy = null;
     boolean sorted = true;
     for (int i = 0; i < data.length - 1; i++)
          if(data[i] >= data[i+1]) sorted = false;
     System.out.println("Sorted list: "+sorted);
     System.out.println(runningTime + " millisecs (" +(runningTime/1000.0) + ")");
```

```
        System.out.println();
   }
   static void shuffleData(int f[]){
     for(int j = 0; j < f.length/2; j++){
         int k = (int)(Math.random()*f.length);
         int m = (int)(Math.random()*f.length);
         while(k == m) m = (int)(Math.random()*f.length);
         int temp = f[k]; f[k] = f[m]; f[m] = temp;
     }
   }
}
class RankerNoDup extends Thread{
  int data[]; int copy[];int lb; int ub;
  public RankerNoDup(int d[], int c[], int k, int m){
       data = d; copy = c; lb = k; ub = m;
  }
  public void run(){
    for(int i = lb; i < ub; i++){
          int rank = 0;
         for(int j = 0; j < data.length; j++)
           if(data[j] < data[i]) rank++;
         copy[rank] = data[i];
    }
  }
}
```

Happy Numbers Problem

A given positive integer number n is defined to be *happy* if the sequence of numbers generated by taking the sum of the squares of successive terms, starting at n, eventually terminates with a term whose value is 1. Numbers that are not happy, called *sad*, generate an infinite sequence of cyclical terms. For example, the number 19 is happy because the sequence of terms generated by the algorithm is 19, 82, 64, 100, 1. We calculate this sequence as follows:

$19 => 1^2 + 9^2 = 1 + 81 = 82$

$82 => 8^2 + 2^2 = 64 + 4 = 68$

$$68 => 6^2 + 8^2 = 36 + 64 = 100$$

$$100 => 1^2 + 0^2 + 0^0 = 1 + 0 + 0 = 1$$

All numbers that occur in the sequence are themselves *happy* numbers. The number of terms generated is actually quite small because the sequence converges very quickly. For example, the number 986543210 is the greatest happy number with no repeated digits but the sequence only consists of 7 terms: 986543210, 236, 49, 97, 130, 10, 1. Of course numbers that are not happy are sad! But how do we distinguish between them, i.e. how do we know when to stop generating new terms if a number is not happy. This is important because *sad* number sequences will never converge to 1. It turns out that *sad* numbers eventually generate cyclical terms and these terms can be listed. They are the numbers: 4, 16, 20, 37, 58 and 145. The following sequential single threaded program takes a number as input and prints the terms generated in determining if a number is *happy*. It terminates if a number generated in the sequence occurs in the *sad* list given above is encountered. It uses two functions: sumSquares that returns the sum of the squares of each digit in n and isSad that returns true if n occurs in the list{4,16,20,37,58,145}.

```
import java.util.*;
public class HappyNumSeqTest {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter number: ");
      int n = in.nextInt();
      while(n != 1 && !isSad(n)){
        System.out.print(n+" ");
        n = sumSquares(n);
      }
      if(n == 1) System.out.println(1);
      else
        System.out.println("Sad number");

   }
   static int sumSquares(int n){
   int s = 0;
   while(n > 0){
       int r = n%10;
```

```
        s = s + r*r;
        n = n/10;
    }
    return s;
    }
    static boolean isSad(int n){
      int cycleNums[] = {4,16,20,37,58,145};
      boolean found = false;
      int j = 0;
      while(j<cycleNums.length && !found)
       if(cycleNums[j] == n)
           found = true;
       else
           j++;
      return found;
    }
}
```

There are an infinite number of *happy* numbers and we want to find how many occur in a given range. For the example we will try to find the frequency of happy numbers in the range 1..Integer.MAX_VALUE. To do this we simply iterate over the range of values checking for a *happy* numbers as we go. The function isHappy(n) returns true if n is happy, false otherwise. The program is listed below:

```
import java.util.*;
public class HappyNumbersTest {
    public static void main(String[] args) {
    System.out.println(Integer.MAX_VALUE);
    long startTime = System.currentTimeMillis();
    int n = 1;
    int freq = 0;
    while(n < Integer.MAX_VALUE){
       if(isHappy(n)){
           freq++;
       }
       n++;
    }
    long endTime = System.currentTimeMillis();
    long runningTime = endTime - startTime;
    System.out.println("Running Time = " + runningTime/1000.0+ " secs");
    System.out.println("Frequency =  "+freq );
   }
   static boolean isHappy(int n){
     //assume n >=1
     int h = n;
     while(h != 1 && !isSad(h)) h = sumSquares(h);
     if(h == 1)
       return true;
     else
       return false;
   }
   static int sumSquares(int n){// same as above }
   static boolean isSad(int n){ // same as above }
}
```

This program takes approximately 420secs on our dual processor quad core Mac and about … on our dual core laptop.

We now investigate parallelizing this algorithm. Allocating one thread per number is pointless so we explore dividing the work fairly between threads, where the number of threads equates to the number of processors on the machine executing the code. The program given below uses our original algorithm to distribute the workload between the threads (HappyNumFinder()) and each thread calculates the frequency of happy numbers in its given range of values. The thread returns this value through the public method freq(). The functions used in the original program are encapsulated in the class HappySad as static methods. This simplifies making the functions available to different threads. This solution completes the task in 159secs on the Mac and .. on the laptop.

```
import java.util.*;
public class HappyNumsParallel {
   public static void main(String[] args) {
      int numProc = Runtime.getRuntime().availableProcessors();
      System.out.println("Number of processors =" + numProc);
```

```
        int base = Integer.MAX_VALUE/numProc;
        HappyNumFinder[] nf = new HappyNumFinder[numProc];
        int n = 1;
        for(int j = 0; j < numProc;j++){
          nf[j] = new HappyNumFinder(n,base*(j+1));
          n = (j+1)*base;
        }
        long startTime = System.currentTimeMillis();
        for(int j = 0; j < numProc; j++) nf[j].start();
        try{
          for(int j = 0; j < numProc; j++) nf[j].join();
        }
        catch(InterruptedException e){}
        long endTime = System.currentTimeMillis();
        long runningTime = endTime - startTime;
        int total = 0;
        for(int j = 0; j < numProc; j++) total = total + nf[j].num();
        System.out.println(total);
        System.out.println("Running Time = " + runningTime/1000.0+ " secs");
    }
}
class HappyNumFinder extends Thread{
    int total = 0;
    int lb, ub;
    public HappyNumFinder(int a, int b){
        lb = a; ub = b;
    }
    public void run(){
        int n = lb;
        int freq = 0;
        while(n < ub){
         if(HappySad.isHappy(n)){
           freq++;
         }
         n++;
        }
        total = freq;
    }
    public int freq(){
        return total;
    }
}

class HappySad{
    static boolean isHappy(int n){
    //assume n >=1
    int h = n;
    while(h != 1 && !isSad(h)) h = sumSquares(h);
    if(h == 1)
        return true;
    else
        return false;
    }
    private static int sumSquares(int n){
    int s = 0;
    while(n > 0){
        int r = n%10;
        s = s + r*r;
        n = n/10;
    }
    return s;
    }
    static boolean isSad(int n){
    int cycleNums[] = {4,16,37,58,145,20};
    boolean found = false;
    int j = 0;
    while(j<cycleNums.length && !found)
        if(cycleNums[j] == n)
            found = true;
        else
            j++;
    return found;
```

```
   }
}
```

Threadpools

A Java library, called the **Executor Framework**, provides a framework that standardizes invocation of threads, thread scheduling, and execution. This frame work provides support for controlling asynchronous tasks according to a set of execution policies and manages the execution of what are termed worker threads. These worker threads are created by the framework independently of the programmer. The task of the programmer then becomes one of defining tasks for the worker threads to do. Essentially writing concurrent solutions to problems becomes one of breaking the given problem into a set of discreet components or sub-tasks that can be executed by worker threads to solve the overall task. A task is defined or encapsulated by a class that implements the Runnable interface. The Executor then creates what is called a threadpool that executes submitted instances of Runnable tasks. This interface provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc. Tasks are queued in a shared queue. When a thread in the pool completes a task, it gets another from queue, waiting if none available. Therefore, the life cycle of a worker thread in the pool is: get next job from work queue, execute it, and go back to waiting for the next job. Threadpools are useful because they can be created once and re-used. Often used in servers because it avoids the overhead of creating threads continuously to deal with client requests.

To create a thread pool use:

```
   ExecutorService pool = Executors.newFixedThreadPool(numThreads);
```

The method newFixedThreadPool(numThreads) returns a thread pool object running numThreads threads. The ExecutorService class provides a number of methods that can be used to interact with the thread pool.

```
pool.submit(r)
```

- submit Runnable r to pool; returns immediately

```
pool.shutdown()
```

- shut pool when all submitted tasks complete

awaitTermination(long timeout,
TimeUnit unit)

throws InterruptedException

- Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

public interface Future<V>

A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready. Cancellation is performed by the cancel method. Additional methods are provided to determine if the task completed normally or was cancelled. Once a computation has completed, the computation cannot be cancelled

get()

throws InterruptedException,

ExecutionException

Waits if necessary for the computation to complete, and then retrieves its result.

We will use this interface to wait for threads to complete and also to retrieve results from the Callable interface. See examples below.

The first example uses a threadpool of two threads to execute 4 instances of the Runnable interface T1. Task T1 is implemented as an implementation of the Runnable interface and the code is contained in the run method. In this simple example the task is to print a number 10 times on the screen followed by a newline. The array of class Future is used to hold references to each task submitted to the pool of threads. In this case 4 jobs are going to be submitted and, hence, the array has size 4. Once the jobs are submitted we want main() to wait for all the tasks in the threadpool to complete. This is done by invoking get on each job submitted. Then the pool is shutdown.

```
import java.util.concurrent.*;
class ThreadPool1{
    public static void main(String args[]){
      ExecutorService pool;
      pool = Executors.newFixedThreadPool(2);
      Runnable r;
      Future f[] = new Future[4];
      for(int j = 0; j < 4; j++){
        r = new T1(j);
        f[j] = pool.submit(r);

      }
      // wait for tasks to complete
      try{
       for (Future x : f) x.get();
      }
      catch(InterruptedException e){}
      catch(ExecutionException e){}
      pool.shutdown();
      System.out.println("Main finished");
    }
}

class T1 implements Runnable{
    int k;
    public T1(int kk){
        k = kk;
    }
    public void run(){
        for(int j = 0; j < 10; j++)
          System.out.print(k+" ");
        System.out.println();
    }
}
```

As a second example, we use a threadpool to initialize the elements in a two dimensional array on a row by row basis. We choose to initialize the data on a row by row basis to try to optimize the use of caching on the machine. Opting to process it on a column by column basis can give rise to lots of copying to and from cache that is costly. The number of threads created for the threadpool in this program equates to the number of processors on the machine being used to test the application. This optimizes the possibility of parallel execution.

```
import java.util.concurrent.*;
public class ThreadPool2 {
    public static void main(String args[]){
        int f[][] = new int[100][500];
        int numProcessors = Runtime.getRuntime().availableProcessors();
        ExecutorService pool;
        pool = Executors.newFixedThreadPool(numProcessors);
        Runnable r;
        for(int j = 0; j < f.length ; j++){
          r = new Init(f,j);
          pool.submit(r);
        }
        try{
          pool.shutdown();
          pool.awaitTermination(1000L,TimeUnit.SECONDS);
        }
        catch(InterruptedException e){}
           // output result
        for(int i = 0; i < f.length; i++){
            for(int j = 0;j < f[0].length; j++)
                System.out.printf("%3d",f[i][j]);
            System.out.println();
        }
    }
}
class Init implements Runnable{
  int ff[][];
  int row;
  public Init(int f1[][],int rr){
     ff = f1; row = rr;
  }
  public void run(){
    for(int i = 0; i < ff[0].length; i++)
      ff[row][i] = (int)(Math.random()*100);
  }
}
```

In the next example we illustrate how to use the Callable interface to execute asynchronous functions. A callable task is one that returns a result and may throw an exception. A class that implements this interface must implement a single method with no arguments called call. The method call is similar to the run method in Runnable, except that it returns a value on termination. This value is retrieved through a Future get method invocation. As each job is submitted to the pool the client keeps a Future reference and uses its get method to retrieve the result of the computation when the job completes. The interface is declared as public interface Callable<V>. This means that it is generic and, therefore, any class that implements it must define its type. The call method must also return an instance of this generic type.

The sample program given below creates a class Square that implements the Callable interface. The class has a single method call that calculates the result and returns it. A pool of threads equal to the number of processors on the machine executing the code is created and a data array is used to hold the elements whose square must be calculated. The get method from the class Future is used to retrieve the result of the invocation of each call by the worker threads.

```
import java.util.concurrent.*;
import java.util.*;
public class Callable1{
public static void main(String[] args){
    int nProc = Runtime.getRuntime().availableProcessors();
    ExecutorService pool = Executors.newFixedThreadPool(nProc);
    int data[] = {2,4,6,8,10,12,14,16,18,20,22,24,26,28,30};
     ArrayList <Future<Integer>> future =  new ArrayList <Future<Integer>>();
    for(int j = 0; j < data.length;j++){
        Future<Integer> f = pool.submit(new Square(data[j]));
        future.add(f);
    }
    //create array to store results
    int result[] = new int[data.length];
    for(int j = 0; j < result.length; j++){
        try {
            Future<Integer> f = future.get(j);
            result[j] = f.get();
        }
        catch (InterruptedException e) {}
        catch (ExecutionException e) {};
    }
    pool.shutdown();
    for(int x : result)
        System.out.printf("%4d",x);
    System.out.println();
  }
}
class Square implements Callable<Integer>{
    private int n;
    Square(int k) {
        n = k;
    }
    public Integer call() { return n*n;}
}
```

For a second, more useful, example of implementing the Callable interface with threadpools we develop a program that calculates the sum of the elements in an array by distributing the workload fairly between the processors on the machine being used to execute the code. The array is broken into discreet segments where each segment differs in size by at most 1. We do this using the algorithm developed earlier.

```
import java.util.concurrent.*;
import java.util.*;
public class Callable2 {
    public static void main(String[] args) {
```

```
    int nProc = Runtime.getRuntime().availableProcessors();
    ExecutorService pool = Executors.newFixedThreadPool(nProc);
    int data[] = new int[1000];
    init(data);
     ArrayList <Future<Integer>> future = new ArrayList <Future<Integer>>();
    //divide the work evenly between processors
    // need to calculate the distribution using a fixed block scheme
    // assume size of data significantly larger than no of processors
      int[] index = new int[nProc+1];
    for (int i=0; i<=nProc; i++) {
          index[i] = (i*data.length)/nProc;
    }
      for(int j = 0; j < index.length-1 ;j++){
       Future<Integer> f = pool.submit(new Sum(data,index[j],index[j+1]));
       future.add(f);
    }
    //create array to store results
    int result[] = new int[index.length - 1];
    for(int j = 0; j < result.length; j++){
       try {
         Future<Integer> f = future.get(j);
         result[j] = f.get();
       }
       catch (InterruptedException e) {}
       catch (ExecutionException e) {};
    }
    pool.shutdown();
    //calculate total by adding partial results
    int total = 0;
    for(int x : result)
       total = total + x;
    System.out.println();
    System.out.printf("Total = %d",total);
    System.out.println();
}
static void init(int dd[]){
      for(int i = 0; i < dd.length;i++)
         dd[i] = (int)(Math.random()*100);
   }
}
class Sum implements Callable<Integer>{
   private int from, to;
   private int data[];
   Sum(int dd[], int st, int en ) {
      data = dd; from = st; to = en;
   }
   public Integer call() {
     int sum = 0;
     for(int j = from; j < to; j++)
      sum = sum + data[j];
     return sum;
   }
}
```

Prime Number Problem

Consider the problem of computing the frequency of prime numbers in the range 1 to 10000000 in as fast a time as possible based on the underlying hardware available to execute our algorithm. We might begin by writing a single threaded solution and then record its performance on our hardware. This would give a base performance against which we can benchmark all subsequent algorithms. The idea is to try to find an optimal solution based on the original single threaded solution. We will not consider alternative approaches to solving the given problem. The proposed single threaded sequential solution is given below. The program uses the services of a static method `isPrime(n)` encapsulated by the class `PTest`. This method returns `true` if `n` a prime number; `false` otherwise. The loop checks values up to and including the integer square root of `n` and terminates on finding a value that divides `n`. The `main` method records the starting time and counts all odd numbers in range that are prime. It begins with `freq` equal to `1` because `2` is the smallest prime and the only even number that is one. Hence, the loop only checks odd numbers in its search for those that are prime. (Note that this is not the only possible solution to this problem. However, for the purposes of our discussion here it is the one we consider).

```
public class PrimeNumberSequential {
    static final int N = 10000000;
    public static void main(String[] args) {
      long startTime = System.currentTimeMillis();
      int freq = 1; // 2 is prime
      for(int k = 3; k < N; k = k + 2)
         if(PTest.isPrime(k))
           freq++;
      long endTime = System.currentTimeMillis();
      long runningTime = endTime-startTime;
      System.out.printf("Number of primes in range %d to %d = %d\n", 1,N,freq);
      System.out.println(runningTime + " millisecs (" +(runningTime/1000.0) +
                                                                    ")secs");
    }
}
class PTest{
    static boolean isPrime(int n){
      boolean prime = true;
      if(n < 2) prime = false;
      for(int k = 2; k <= (int)Math.sqrt(n) && prime; k++)
        if(n % k == 0) prime = false;
       return prime;
    }
}
```

This solution completed the task, running on a dual core processor, in approximately 36secs. This we consider to be a base time and we will measure performance in terms of it.

Now we want to explore ways to improve this program, in terms of performance, by developing a parallel solution that tries to utilize the power of the underlying multi-core hardware. A good starting point is to split the work involved evenly over the cores of the underlying machine. Each core is given approximately the same amount of work to do and, hopefully, this will deliver a faster solution. To divide up the workload we use our work distribution algorithm from above and introduce a thread class called `PrimeCounter`. This thread takes a range of values to test for *primeness* and returns the number found through its public method freq. The main method retrieves the number of available processors, divides the workload fairly between them, records the start time, creates an array of threads and sets them to work. On completion of the work it retrieves the results and calculates their sum. It then prints its findings and the time taken.

```
public class PrimeNumberParallel {
  static final int N = 10000000;
  public static void main(String[] args) {
    int nThreads = Runtime.getRuntime().availableProcessors();
    PrimeCounter workers[] = new PrimeCounter[nThreads];
    int index[] = new int[nThreads+1];
    for(int j = 0; j <= nThreads; j++)
       index[j] = (j*N)/nThreads;
    long startTime = System.currentTimeMillis();
    for(int j = 0; j < workers.length; j++){
       workers[j] = new PrimeCounter(index[j], index[j+1]);
       workers[j].start();
    }
    //wait for workers to finish
   try{
     for(int j = 0; j < workers.length; j++)
       workers[j].join();
   }
   catch(InterruptedException e){}
   int freq = 0;
   for(int j = 0; j < workers.length; j++)
      freq = freq + workers[j].freq();
   long endTime = System.currentTimeMillis();
   long runningTime = endTime-startTime;
   System.out.printf("Number of primes in range %d to %d = %d\n", 1,N,freq);
   System.out.println(runningTime + " millisecs (" +(runningTime/1000.0) +
                                                                       ")secs");
   }
}
class PrimeCounter extends Thread{
   private int lb, ub; int freq = 0;
   PrimeCounter(int a, int b){
     lb = a; ub = b;
   }
   public void run(){
      freq = 0;
      for(int k = lb ; k < ub; k = k + 1)
        if(PTest.isPrime(k))
         freq++;
   }
   public int freq(){return freq;}
}
```

One would expect that this solution would execute approximately k times, where k equals the number of cores on the machine, faster than the single threaded sequential solution. However, this did not turn out to be the case. There was an improvement but not as much as we would like. It took approximately 24secs on a dual core machine, an improvement of 33%. The question is why? It turns out, on inspection, that some threads finished working before others and lay idle for considerable periods of time. The reason for this is that threads testing large values for *primeness* do more work than those checking smaller values. This would suggest that distributing the workload in such a way as to utilize all worker threads at all times might deliver a better performance. To do this we need to carve up the range of values to test into smaller blocks and use a threadpool to manage the threads. This would mean that when a thread checking small numbers finished it would go back to the job queue and select another block of numbers to test. This solution is listed below. The range size is fixed at 1000 giving 10000 tasks to complete. We experimented with different range sizes and these made little difference to the overall performance. But the good news was that the program delivered, on the same dual core machine, approximately 50% improvement in performance.

```
import java.util.concurrent.*;
import java.util.*;
public class PrimeNumberTPool {
  static final int N = 10000000;
  static final int RangeSize = 1000;
  public static void main(String[] args) {
    int nThreads = Runtime.getRuntime().availableProcessors();
    ExecutorService pool = Executors.newFixedThreadPool(nThreads);
    ArrayList <Future<Integer>> future = new ArrayList <Future<Integer>>();
   //divide the work of finding primes into small range segments
   int[] index = new int[N/RangeSize+1];
   for (int i=0; i < index.length; i++)
       index[i] = i*RangeSize;
   long startTime = System.currentTimeMillis();
   for(int j = 0; j < index.length-1 ;j++){
      Future<Integer> f = pool.submit(new PrimeCounter(index[j],index[j+1]));
      future.add(f);
   }
   //create array to store results
   int result[] = new int[index.length - 1];
   for(int j = 0; j < result.length; j++){
     try {
       Future<Integer> f = future.get(j);
       result[j] = f.get();
     }
    catch (InterruptedException e) {}
    catch (ExecutionException e) {};
   }
   pool.shutdown();
   //calculate total by adding partial results
   int freq = 0;
   for(int x : result)
     freq = freq + x;
   System.out.println();
   long endTime = System.currentTimeMillis();
   long runningTime = endTime-startTime;
   System.out.printf("Number of primes in range %d to %d = %d\n", 1,N,freq);
    System.out.println(runningTime + " millisecs (" +(runningTime/1000.0) +
                                                                     ")secs");
   System.out.println();
  }
}
class PrimeCounter implements Callable<Integer>{
   private int lb, ub;
   PrimeCounter(int a, int b){
     lb = a; ub = b;
   }
   public Integer call(){
     int freq = 0;
     for(int k = lb ; k < ub; k = k + 1)
        if(PTest.isPrime(k))
        freq++;
     return freq;
   }
}
```

Fork Join and Work Stealing

Java 7 introduced what it calls a ForkJoinPool that provides an implementation of a divide and conquer algorithm that uses what is called work stealing for enhanced performance. The divide and conquer algorithm takes the form (D. Lea, A Java Fork/Join Framework, 2000)

```
    Result solve(Problem problem) {
         if (problem is small)
            directly solve problem
         else {
            split problem into independent parts
            fork new subtasks to solve each part
            join all subtasks
            compose result from subresults
         }
    }
```

This is a recursive algorithm that divides a task until it is manageable and then solves the sub task. To do this it recursively forks new tasks and then waits (join) for all sub-tasks to complete. The underlying tasks are managed by a threadpool that uses *work stealing*. The ThreadPoolExecutor had a central inbound queue for new tasks (Runnables or Callables), which is shared by all worker threads. One disadvantage of this model is that there is no support for multiple workers to collaborate when solving tasks. The idea behind *work stealing* is that workers steal work from other workers, if they themselves are idle. Just like the ThreadPoolExecutor, the ForkJoinPool uses a central inbound queue and an internal thread pool. In the case of ForkJoinPool, however, each thread in the pool manages its own job queue and can add new jobs to it. In the event that it has no work left to do in its own queue it inspects the queue of another worker in the pool and tries to help by *stealing* a job from it. If it fails to find one it then searches the central pool and, if none found, it rests for a small period before trying again. This is necessary to avoid workers wasting CPU time searching for jobs that may not exist.

The class ForkJoinPool creates an environment that will execute and manage fork-join tasks. By default it allocates as many threads as there are processors on the machine to handle the pool of tasks. There are two types of tasks that correspond to Runnable and Callable for threadpools: RecursiveAction<V>, that defines a task and does not return a value and RecursiveTask<V> that returns a value on completion. The format for writing tasks is the same in both cases. Classes that inherit from these super classes must override the compute method. This method takes no arguments and may or may not return a value. The template for a RecursiveTask based on the divide and conquer algorithm is given below. The constant BaseBlockSize is used to terminate the recursion. When the job size reaches some minimum limit or size then execute the code to solve it. The else is used to divide the task recursively in half, forking new tasks to complete. To retrieve the result from the task, (compute method), use a join to wait for the task to complete. The type of the result must match the generic type V.

```
class <Name> extends RecursiveTask<V> {
   static final int BaseBlockSize = …;
   <Name>(…){
      //constructor
   }
   protected <V> compute() {
      if(… <= BaseBlockSize) {
         // execute sequential code
       } else {
         //divide job size
         <Name> left  = new <Name>(…);
         <Name> right = new <Name>(…);
         left.fork();
```

```
        right.fork();
          <V> <varName> = right.join();
          <V> <varName>  = left.join();
          return <result>;
        }
    }
}
```

Note: It is important to fork both tasks before invoking join. Interleaving them actually creates a sequential list of tasks where each task waits for the previous one to complete before executing its task. Therefore, you never write:

```
left.fork();
<V> <varName>  = left.join();
 right.fork();
  <V> <varName>  = right.join();
```

This sequence forces right to wait for left to complete and, hence, inhibits parallelism.

End note

Any problem that can be sub-divided until some data size threshold is reached can be implemented using a ForkJoinPool. A simple example of such a recursive task is to compute the sum of the elements in a huge array. Addition distributes over the sum of sub segments. This means that we can break the sequence into a collection of sub-tasks, calculate the sum of each segment and then add all of the partial sums to get the total sum. To optimize our solution we don't want to reduce the segment size below some given threshold because then we would create too many tasks and the overhead of executing them would possibly give a worse performance than a simple sequential solution. In this case we choose a minimum segment size of, say, 5000. Any segment whose block size is less than or equal to 5000 will be summed in sequence. This size will, therefore, be used to terminate the recursion. The class Sum inherits RecursiveTask and returns a result of type Long. Its constructor takes a reference to the global data array and a lower (lb) and upper (ub) bound denoting the sequence dt[lb..ub-1]. The method compute implements the recursive construction of sub-tasks. The base case uses a simple for loop to iterate over the given segment computing the sum of the elements as it goes. It then returns the sub-total, sum. The else block sub divides the problem into its constituent tasks. Each invocation results in the creation of two sub-tasks. Each of these forks a new instance of Sum and waits to retrieve the result of the task pair. It creates at most log_2N sub-tasks and all sub-tasks of size 5000 or less are executed sequentially.

The task is encapsulated as follows:

```
class Sum extends RecursiveTask<Long> {
   static final int BaseBlockSize = 5000;
   int lb, ub;
   int[] data;
   Sum(int[] dt, int a, int b){
      data = dt;
      lb = a; ub = b;
   }
   protected Long compute() {
      if(ub - lb <= BaseBlockSize) {
         long sum = 0;
         for(int i=lb; i < ub; ++i)
            sum = sum + data[i];
         return sum;
      } else {
         int mid = lb + (ub - lb) / 2; //calculates middle index
         Sum left  = new Sum(data, lb, mid);
         Sum right = new Sum(data, mid, ub);
         left.fork();
        right.fork();
         long rSum = right.join();
         long lSum  = left.join();
         return lSum + rSum;
      }
    }
}
```

Now we need to write a main program that creates the data and sets up a ForkJoinPool of threads to manage and execute the tasks created by the

RecursiveTask class, Sum. The pool is created and then it's invoke method is used to begin the recursive tasks.

```
ForkJoinPool fjPool = new ForkJoinPool();
long sum = fjPool.invoke(new Sum(f,0,f.length));
```

The code for the test program is:

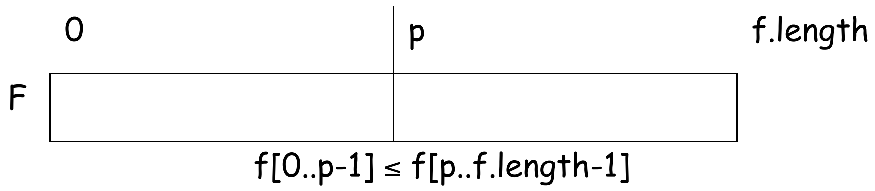
```
import java.util.concurrent.*;
public class ForkJoinTest1 {
   public static void main(String[] args) {
     int f[] = new int[10000000];
     for(int j = 0; j < f.length; j++)
       f[j] = (int)(Math.random()*1000);
      ForkJoinPool fjPool = new ForkJoinPool();
      long sum = fjPool.invoke(new Sum(f,0,f.length));
      System.out.println("Sum = "+sum);
   }
}
```

ForkJoinPool Methods

The class ForkJoinPool has a number of methods that can be used to interact with an instance of the pool. The method invoke performs the given task and returns its result, on completion. To check if all tasks are complete and, as a consequence worker threads are idle the method isQuiescent() can be used. This method returns true if all workers are idle. The method shutdown() causes an orderly shutdown of the pool and after an invocation of this method no new tasks will be accepted by the pool. The method awaitTermination(…) can be used to wait for all jobs to complete after a shutdown request. This method uses a timeout value. (Please see Oracle Docs for more methods relating to ForkJoinPools)

Quicksort Algorithm

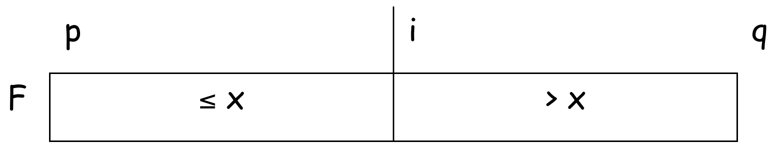
The strategy adopted in *quicksort* is to shuffle the elements in the array into two non-empty partitions such that all the elements in the left partition are less than or equal to those in the right partition. The elements in each partition are not necessarily sorted. The following diagram describes this situation.



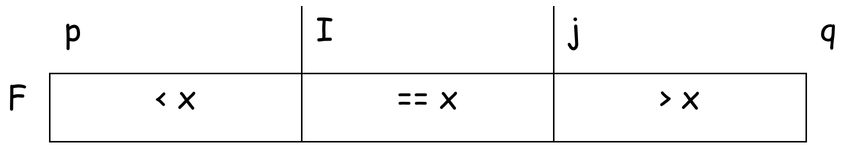
It now only remains to quicksort f[0..p-1] and then quicksort f[p..f.length-1], and the whole array is sorted. An outline solution is:

```
static void quickSort(int f[], int p, int q){
  if(q-p <= 1)
     ; //skip
  else{
    // re-arrange f[p..q-1] into two non-empty partitions
    // f[p..i-1] and f[i..q-1] (p ≤ i ≤ q) and each element in
    // f[p..i-1] ≤ each element in f[i..q-1]
    quickSort(f,p,i);
    quickSort(f,i,q);
  }
}
```

The task re-arrange f[p..q-1] … must now be solved. The strategy is to choose some x in the segment and partition the elements about x, i.e. establish



But if x happens to be the largest element in the array, then the right hand partition would be empty and the condition that both segments be non-empty is violated. To avoid this situation we choose to arrange the array into three segments as follows:



The final two calls now become:

```
    quickSort(f,p,i);
  quickSort(f,j,q);
```

The problem of partitioning the array into three segments is similar to the problem of the Dutch national flag and can be done in a single iteration of the data segment. The solution is presented here without discussion. (See Data Structures book on Fast Sorting for a discussion of this)

```
int x;
int i, j, k;
// let x = middle element in f[p..q-1]
x = f[(p+q)/2];
i = p; j = p; k = q;
while(j != k){
  if(f[j] == x)
     j = j + 1;
  else if(f[j] < x){
    //swap f[j] with f[i]
  }
  else{ // f[j] > x
     // swap f[j] with f[k-1]
   }
 }
```

The derivation of *Quicksort* is now complete. Analysis of the algorithm suggests that it is at best $O(n * log_2 n)$ and at worst $O(n^2)$. The reason for this variation in performance is due in the main on the choice of value given x during the partition phase. In the solution given above, we chose to give x the middle value. To my knowledge, this choice was suggested by Nicklaus Wirth. In the original solution given by Tony Hoare the value chosen was the first value in the partition. Depending on the given ordering of the data in the original array this could greatly impact on performance. If the original data sequence is sorted then the performance is $O(n^2)$ because the size of the leftmost partition is typically only 1.

This is a divide and conquer algorithm and is a prime candidate for our ForkJoinPool model. It is naturally recursive in nature and we can implement it by extending the RecursiveAction class.

```
class QuickSort extends RecursiveAction{
  int lb;
```

```
 int ub;
 int[] data;
 QuickSort(int[] dt, int a, int b){
    data = dt;
    lb   = a; ub  = b;
 }
 protected void compute() {
  if(ub - lb > 1){
     int i, j, k;
       // let x = middle element in f[p..q-1]
       int x = data[(lb+ub)/2];
       i = lb; j = lb; k = ub;
       while(j != k){
         if(data[j] == x)
              j = j + 1;
         else if(data[j] < x){ //swap f[j] with f[i]
              int temp = data[j];
              data[j] = data[i];data[i] = temp;
              j = j + 1; i = i + 1;
         }
         else{ // f[j] > x
              // swap f[j] with f[k-1]
              int temp = data[j];
              data[j] = data[k-1];data[k-1] = temp;
              k = k - 1;
         }
     }
     QuickSort left  = new QuickSort(data, lb, i);
     QuickSort right = new QuickSort(data, j, ub);
     invokeAll(left,right);
     left.join(); right.join();
  }
 }
}

import java.util.*;
import java.util.concurrent.*;
public class QuickSortForkJoin {
  public static void main(String[] args) {
     int data[] = new int[10000000];
     ForkJoinPool fjPool = new ForkJoinPool();
     for(int j = 0; j < data.length; j++) data[j] = (int)(Math.random()*10000);
     long startTime = System.currentTimeMillis();
     fjPool.invoke(new QuickSort(data,0,data.length));
     long endTime = System.currentTimeMillis();
     long runningTime = endTime-startTime;
     //check that data sorted
     boolean sorted = true;
     for (int i = 0; i < data.length - 1; i++)
        if(data[i] > data[i+1]) sorted = false;
     System.out.println("Sorted list: "+sorted);
     System.out.println(runningTime + " millisecs (" +(runningTime/1000.0) + ")");
  }
}
```

This solution running on a dual core processor delivered performance on average of approximately 2.9secs for 10 million randomly generated integer values in the range 0 .. 9999999.

Exercise

Question 1

Write a parallel bubblesort with two threads. Conduct tests to measure the performance of the parallel solution.

Question 2

Given is a program that initializes the elements in a 100x10 matrix. Write a parallel solution that uses individual threads to initialise each column.

```
class MatrixInit{
    public static void main(String args[]){
        int f[][] = new int[100][10];
        for(int i = 0; i < 100; i++){
            for(int j = 0; j < 10; j++)
                f[i][j] = (int)(Math.random()*100);
        }
        //print f
        System.out.println();
        for(int i = 0; i < 100; i++){
            for(int j = 0; j < 10; j++)
                System.out.print(f[i][j]+ " ");
            System.out.println();
        }
    }
}
```

Question 3

Re-write parallel rank sort to allow for duplicates.

Question 4

Matrix addition – write a program to add two matrices. You may assume both matrices have the same dimensions. The single threaded solution is given below. Your task is to write a parallel version.

```
class MatrixAddition{
    public static void main(String args[]){
        int f[][] = new int[10][5];
        int g[][] = new int[10][5];
        int add[][] = new int[10][5];
        // init both matrices
        for(int i = 0; i < f.length;i++){
            for(int j = 0; j < f[0].length; j++){
                f[i][j] = (int)(Math.random()*10);
                g[i][j] = (int)(Math.random()*10);
            }
        }
        // add corresponding elements on a row by row basis
        for(int i = 0; i < f.length; i++){
            for(int j = 0;j < f[0].length; j++){
                add[i][j] = f[i][j] + g[i][j];
            }
        }
                    // output result
        for(int i = 0; i < f.length; i++){
            for(int j = 0;j < f[0].length; j++){
                System.out.print(add[i][j]+" ");
            }
            System.out.println();
        }
    }
}
```

Question 5

Given is a program that computes the sum of each column in a matrix. Your task is to write a parallel version for a dual core machine.

```
public class SumColumns {
    public static void main(String args[]){
    int f[][] = new int[10][5];
    for(int i = 0; i < f.length;i++)
        for(int j = 0; j < f[0].length; j++){
        f[i][j] = (int)(Math.random()*10);
         }
      // Compute sum of columns
       int sum[] = new int[f[0].length];
       for(int j = 0; j < sum.length;j++)sum[j] = 0;
       for(int i = 0; i < f.length; i++){
         for(int j = 0;j < f[0].length; j++){
        sum[j] = sum[j]+ f[i][j];
         }
       }
       // output result
       for(int i = 0; i < f.length; i++){
          for(int j = 0;j < f[0].length; j++){
            System.out.printf("%3d",f[i][j]);
         }
        System.out.println();
       }
       for(int j = 0; j < sum.length;j++)
          System.out.printf("%3d",sum[j]);
   }
}
```

Question 6

Given below is a sequential single threaded solution to the problem of multiplying matrices. Your task is to write a parallel version that minimizes the overhead of caching.

```
class MatrixMultiplication{
 static final int row1 = 2;
 static final int col1 = 3;
 static final int row2 = 3;
 static final int col2 = 2;
 // Note: to multiply two matrices the number of columns
 // in the first matrix must equal the number of rows
 // in the second one. The resultant matrix has dimensions
 // row1 x col2
 public static void main(String[] args){
  int matrix1[][] = new int[row1][col1];
  int matrix2[][] = new int[row2][col2];
  int res[][]     = new int[row1][col2];

  initMatrix(matrix1);
  initMatrix(matrix2);
  multiply(matrix1,matrix2,res);

  printMatrix(matrix1);
  System.out.println();
  printMatrix(matrix2);
  System.out.println();
  printMatrix(res);
  System.out.println();
 }

 static void initMatrix(int m1[][]){
   for(int j = 0; j < m1.length; j = j + 1){
     for(int k = 0 ; k < m1[0].length; k = k + 1){
        m1[j][k] = (int)(Math.random()*6);
     }
   }
 }

 static void multiply(int m1[][],int m2[][], int p[][]){
   for(int c = 0; c < p[0].length; c = c + 1){
    for(int r = 0; r < p.length; r = r + 1){
      p[r][c] = product(m1,m2,r,c);
    }
   }
 }

 static int product(int m1[][],int m2[][], int r1, int c1){
   int sum = 0;
   // multiply the row into the column
   for(int j = 0; j < m2.length; j = j + 1){
         sum = sum + m1[r1][j] * m2[j][c1];
   }
   return sum;
 }
```

```
 static void printMatrix(int a1[][]){
   for(int j = 0; j < a1.length; j = j + 1){
     for(int k = 0; k < a1[0].length; k = k + 1) System.out.print(a1[j][k]+" ");
     System.out.println();
   }
 }
}
```

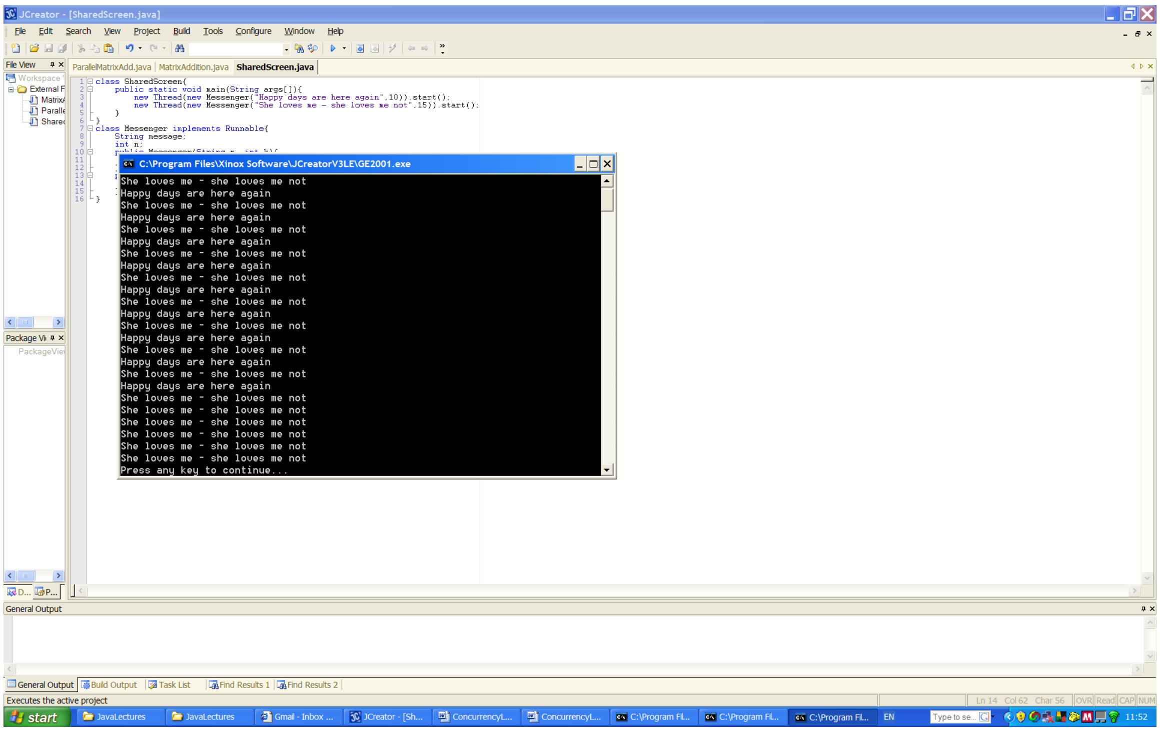
Sharing Resources

We have seen in the previous chapter that threads may share data structures by granting each thread a reference to the global instance of the data. In the cases we studied each thread was given its own individual segment of the data to process and there was no interleaving of segments. However, it is necessary to allow threads to share data where multiple threads may require access to the same data segments. There are a myriad of examples where this is necessary. In an on-line library multiple readers may try to borrow the same book at the same time. Only one borrower can actually do so. In a banking system two people may own a joint account and both owners may try to borrow money at exactly the same time at different ATM's. This action, if not synchronized, may lead to both people removing money that is not actually present. Both could check the account and see that there are sufficient funds for their transaction and then request the money. However, there may not be sufficient funds for both transactions. A server may manage a single printer and many users may try to print documents at the same time. The server has to ensure that only one user can print at a time and that a print job once started keeps the printer until it is finished. This ensures that documents are not printed piecemeal. In a program you may use multiple threads to solve a problem and some of the threads may need to write output to a shared terminal. It is important that each thread gains exclusive access to the display device if you want an end-user to be able to read the text correctly. Similarly, there are situations where we need crowd control. For example, a stadium will have a maximum capacity and to protect users of the stadium we must ensure that the number of occupants never exceeds this value. If you think of the stadium as a shared resource that has limited capacity you can see that the problem of managing access is similar to the other examples given. However, it differs in that there is an upper limit in terms of user access. A similar example, from operating systems, is that of a bounded buffer that has some upper limit. When the buffer fills up writers to the buffer must wait for space in the buffer to become available. There are many similar examples where we will need to use limited shared resources and we need protocols to control user access. The moral of the story is that sharing is necessary at all levels and that it needs to be managed so that users don't cause chaos. From a technical perspective it means that protocols have to be put in place to manage shared resources so that threads do not interfere with each other when using these resources. A discussion of shared resources will be the focus of this chapter.

We begin with a simple example of two threads outputting data to the screen. If they happen to print at the same time then most likely the output will be unreadable. The following example illustrates this.

```
class SharedScreen{
    public static void main(String args[]){
      new Messenger("Happy days are here again",10)).start();
      new Messenger("She loves me - she loves me not",15)).start();
    }
}
class Messenger extends Thread{
   String message;
   int n;
   public Messenger(String m, int k){
      message = m; n = k;
   }
   public void run(){
      for(int j = 0; j < n; j++) System.out.println(message);
   }
}
```

The output listed on executing the program shows that the threads printed all the messages mixed together.



```
class SharedScreen{
    public static void main(String args[]){
        new Thread(new Messenger("Happy days are here again",10)).start();
        new Thread(new Messenger("She loves me - she loves me not",15)).start();
    }
}
class Messenger implements Runnable{
    String message;
    int n;
```

C:\Program Files\Xinox Software\JCreatorV3LE\GE2001.exe

```
She loves me - she loves me not
Happy days are here again
She loves me - she loves me not
Happy days are here again
She loves me - she loves me not
Happy days are here again
She loves me - she loves me not
Happy days are here again
She loves me - she loves me not
Happy days are here again
She loves me - she loves me not
Happy days are here again
She loves me - she loves me not
Happy days are here again
She loves me - she loves me not
Happy days are here again
She loves me - she loves me not
Happy days are here again
She loves me - she loves me not
She loves me - she loves me not
She loves me - she loves me not
She loves me - she loves me not
She loves me - she loves me not
She loves me - she loves me not
Press any key to continue...
```

This is not correct. It is necessary that we enforce a rule that only allows one thread to print all its output in one go. To achieve this we need a protocol that manages access to the shared resource in such a way that only one thread can obtain access at any one time. To do this we must add synchronizing code so that they do not interfere with one another destructively. In Java this is achieved by using a synchronized block:

```
synchronized (obj1) {
    stmts1
}
```

Here, `obj1` can be any object reference variable at all; we say that code block stmts1 is *protected by obj1*. You cannot synchronize on a primitive variable type. For example, it is not possible to synchronize on `int x`. There may be many such synchronized blocks in a program. The scheduler guarantees that at any given time, at most one process will be executing the code protected by `obj1`, no matter where how many pieces of such protected code exist. Processes delayed on entry to a synchronized block do not consume resources; they are put to sleep and awoken when their turn has come. The scope of the lock is the code enclosed in the curly brackets. When the code block completes the lock is automatically released and, subsequently, acquired by another, possibly waiting, thread. If the program also contains a different lock, say,

```
synchronized (obj2) {
    stmts2
}
```

there is no ban on `stmts1` and `stmts2` being executed in parallel. Each code block is protected by its own lock. Now we can re-write our messenger program above so that one of the threads, we don't care which one, gets the resource (screen) first and keeps it until it is finished. Then the other thread can print. In this way each thread gets to print all its data without interruption from any other thread. We implement this by creating an instance of the `Object` class and passing a reference to it to each of the threads. Each thread agrees to synchronize on this object before beginning to print on the screen.

```
class SharedScreen{
  public static void main(String args[]){
    Object lock = new Object();
    new Messenger("Happy days are here again",10,lock).start();
    new Messenger("She loves me - she loves me not",15,lock).start();
  }
}
class Messenger extends Thread{
    String message; int n;
    Object lock;
    public Messenger(String m, int k, Object l1){
        message = m; n = k; lock = l1;
    }
    public void run(){
        synchronized(lock){
            for(int j = 0; j < n; j++) System.out.println(message);
        }
    }
}
```

Because each thread synchronizes on the lock reference variable one of them will acquire it, print all of its output on the screen, and then automatically release the lock. The other thread sleeps until this happens and then prints its own output to the screen. The order that this happens in is non-deterministic.

The strategy of synchronized blocks requires cooperation among all the competing threads. Every thread agrees:

- To access critical code only within an appropriate synchronized block.
- Not to delay in a synchronized block, for example, by going to `sleep`.

Every execution of a synchronized block degrades performance and your program's response time, because threads get delayed. Therefore, it is important to keep code in synchronized blocks as short as possible. It is also important to point out that every thread accessing the shared block of code must obey the rules. For example, in the simple program above should `main` try to write to the screen without synchronizing on the lock created, then it would possibly interleave with output from one of the other threads.

It is possible to protect access to shared data structures in this way too. For example, if we have an array that is shared between multiple threads then each thread can agree to lock access to it when it is processing the sequence of values. The two threads `Summer`, that sums the elements in the array, and `Editor`, that modifies the values in the array, both enforce locking and, hence, meet the requirement for sharing given above. The main program creates an instance of the Object class and passes its reference to each of the threads.

```
public class SynchronizedSharedArray {
   public static void main(String[] args){
   int data[] = new int[10];
   for(int j=0; j<data.length;j++) data[j] = (int)(Math.random()*10);
    Object lock = new Object();
    new Summer(data,lock).start();
    new Editor(data,lock).start();
    new Summer(data,lock).start();
  }
}
class Summer extends Thread{
   int data[];
   Object lock;
   Summer(int f[], Object l1){ data = f; lock = l1;}
   public void run(){
      synchronized(lock){
        int sum = 0;
        for(int j=0; j<data.length;j++) sum = sum + data[j];
        System.out.println(sum);
      }
   }
}
class Editor extends Thread{
   int data[];
   Object lock;
   Editor(int f[],Object l1){ data = f;lock = l1;}
   public void run(){
      synchronized(lock){
        for(int j=0; j<data.length;j++) data[j] = j;
      }
   }
}
```

This example uses an explicit reference variable to implement the lock on the shared data structure. However, this is not necessary. The array reference variable, `data`, can itself be used to lock access. This simplifies the coding because the threads can simply lock the data reference variable when processing the shared data structure. In the case of the thread, `Summer`, the code for `run` would be:

```
synchronized(data){
    int sum = 0;
    for(int j=0; j<data.length;j++) sum = sum + data[j];
    System.out.println(sum);
}
```

Atomic Actions

An atomic action is one that can be executed in the knowledge that no interference can occur from another thread when it happens. Java only guarantees that reading from and writing to a memory *word* is atomic. A memory word is 32 bits on 32-bit machines and 64 bits on 64-bit machines. This means that if two integer variables attempt to write to the same address simultaneously, the writes will occur in some order. The memory *word* will have one of the values that were written to it, not a mixture of the two. Similarly, if a read and a write on the same memory *word* in memory is executed simultaneously, the write will be successful and the read will deliver either the value that was in the word before the write, or the value just written. At the language level, this means that assignments `x := e` are atomic, where `x` is a variable of a *word* type, and `e` is a variable or constant. In the case of Java, `int`, `boolean`, `char` and references, are `word` types, but not `long` or `double`. The reason why reads and writes of `long` and `double` are not atomic is because it requires two reads, the first *word* (32 low order bits), followed by the second *word* (32 high order bits) to complete a read. A similar situation occurs in the case of a write instruction.

In fact the situation is even more complicated because modern compilers generate optimized code which includes caching local variables in registers, and changing the order in which assignments take place. These optimizations do not affect the behaviour of single-threaded programs, but may not be valid in the presence of multi-threading. For example, caching variables can lead to a lost assignment, if one process is working on the cached version while another has updated the original variable. For this reason, the compiler should be informed of shared word variables by declaring them **`volatile`**. See our discussion of Visibility below.

This means that in fact almost no instructions can be taken as being atomic. To illustrate why we take a simple `Counter`, listed below, class that encapsulates a single integer attribute `x`. This class has three public methods: `inc`, that increments the current value of `x`, `dec`, that decrements its value, and `get`, that returns its current value. Suppose an instance of this class is shared by two threads; one of the threads increments its value and the other one decrements its value. Suppose the initial value of the variable is 1. We will show that the final value after both threads have modified it could be 1, 0, or 2. Both threads invoke their respective methods at the same time, i.e. they execute c.inc() and c.dec() simultaneously, where c is the name of the shared reference variable. Because assignment involving an increment or a decrement is not atomic, it will give rise to a sequence of machine instructions and a context switch may occur between any one of these. The increment is equivalent to the instructions move x to register 1, followed by incrementing its value, followed by moving the new value to x. A similar situation occurs for decrement.

```
x = x + 1 ≡ mov reg1, x; inc reg1; mov x, reg1
x = x - 1 ≡ mov reg1, x; dec reg1; mov x, reg1
```

As a consequence, the following sequence of events is possible.

```
mov reg1, 1
inc reg1  //reg1 == 2
//context switch
mov reg1, 1
dec reg1
mov x, 0
//now inc resumes
mov x, 2
```

The final value of x is now 2. Similarly it is possible to show that the final result could be 0. To protect against this type of error (called a *race condition*) we need to ensure that both the inc and dec operations are *atomic* because then the final value will always be 1. The order of execution is not important. What is crucial is that each operation once started must complete before the other reads the memory value of x.

```
public class CounterTest {
   public static void main(String[] args) {
   Counter c = new Counter();
     new IncT(c).start();
     new DecT(c).start();
     System.out.println(c.get());
   }
}
class Counter{
  // not thread safe
   private int x = 1;
```

```
    void inc(){x = x + 1;}
    void dec(){x = x - 1;}
    int get(){return x;}
}
class IncT{
    Counter c;
    IncT(Counter cc){ c = cc;}
    public void run(){ c.inc(); }
}
class DecT{
    Counter c;
    IncT(Counter cc){ c = cc;}
    public void run(){ c.dec(); }
}
```

We could solve this problem by ensuring that both threads synchronized on the shared reference variable. In this case each thread would create a synchronized block inside its run method. In the case of `IncT` this would give: `synchronized(c){c.inc();}`

Exercise: Show that if two separate threads invoke `c.inc()` in parallel that the final value of x may only increase by 1 and not 2.

However, we are now going to look at an alternative approach to solving this problem.

Intrinsic Locks

In the object-oriented world it is important that encapsulation is preserved and that class objects control access to their data attributes through their public methods. Users of class instances only see and modify the encapsulated data through the methods published by the class. In this way class invariants are preserved and public users may not modify the data directly nor may they modify the data through reference *leaks*. Writing classes so that data encapsulation is preserved is crucial in an object-oriented world. With the introduction of concurrent access to shared data this obligation does not disappear nor is it weakened. In fact the obligation becomes even more central because now classes must deal directly with the fact that internal operations may require synchronization. Therefore, it becomes an obligation of the class itself to ensure that data is not *corrupted* through sharing. This means that classes must protect their own data and may not rely on external threads to enforce locking. To make this possible Java provides synchronized blocks for each class instance. These synchronized blocks are atomic and all code inside a block may only be executed by a single thread at a time. The scope of the lock may be the scope of a complete method or it may be a code block local to a particular method. In the case of a method the lock must belong to the object on which the method is being invoked. In the case of a code block local to a method the reserved word `this` is used to reference the object itself. Therefore, the following are equivalent

```
synchronized type meth1(...) {
      stmts1
}
```

```
type meth1(...) {
      synchronized(this) {
         stmts1
      }
}
```

(Note, it is possible for a class to use a local object created internally to enforce locking and we will discuss this approach later when we discuss the `Lock` class.) By synchronizing the public methods in our `Counter` class we can now make it *thread safe*.

```
class Counter{
   private int x = 1;
   synchronized void inc(){x = x + 1;}
   synchronized void dec(){x = x - 1;}
   synchronized int get(){return x;}
}
```

Now the class itself is *thread safe* and it is no longer necessary for threads to synchronize on their respective shared variables.

These types of lock are called *intrinsic locks* or *monitor locks*. Each object instance has a single lock and it may only be acquired by one thread at a time. This means that all synchronized methods or local blocks synchronized on `this` are thread safe because only a single thread can execute one of them at a time. It is not possible for two different threads to access two different synchronized methods or blocks in an object at the same time. When a thread attempts to execute a synchronized method or block it automatically acquires the lock, if free, and holds it for the duration of the code block. The lock is automatically released when control exits the block or when an exception is thrown. If the lock is not free then an acquiring thread is put to sleep until the lock becomes free. In the case that a lock is not released then the thread will sleep forever. If two or more threads are sleeping on a lock then when it becomes free one of them will acquire it and the remainder continue to sleep. Ideally, sleeping threads should be managed fairly, first-come-first-served, but this is not guaranteed.

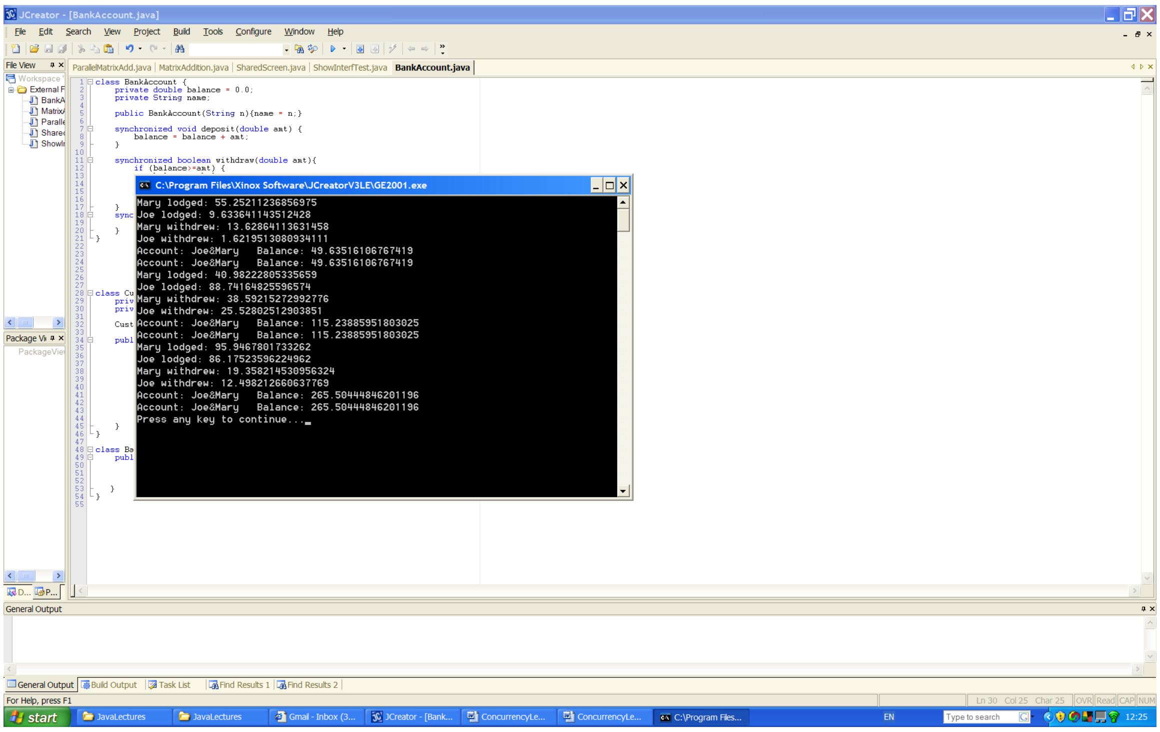
It is important to note that only synchronized methods and blocks synchronized on `this` are protected by the lock. All other methods are unprotected and may be executed in parallel by other threads. This means that there is scope for destructive behaviour. The fact that one thread holds the lock on the object is not a guarantee that another thread may not be executing an unsynchronized code block in another method of the class that modifies one of the variables in its atomic block. For example, if only `inc()` in class Counter was synchronized then another thread could be executing `dec()` in parallel with the thread executing `inc()`. To make a class thread safe you have to ensure that all methods that modify the state of the class are synchronized properly. Constructor methods cannot be declared synchronized. In the case of observer methods if you want to retrieve the most recent state of the data then they should be synchronized as well. See the discussion on Visibility below for a further discussion on this point.

As a second example, we now consider the problem of managing possibly joint bank accounts (i.e. one which may be accessed by more than one person). The class `BankAccount` must be *thread safe* because it may be accessed by two independent customers at the same time. It has three synchronized methods that allow a user to deposit a sum of money, withdraw money, or check the current balance. Because `checkBalance` is synchronized it means that a `deposit` or `withdraw` transaction must complete before returning the balance. This means that the most recent balance is visible to the user. In this example, synchronization enforces both mutual exclusion (atomicity) and reliable communication(visibility).

In the main method we create two customer threads that share a bank account. Each thread performs its own transactions on the account.

```
class BankAccount {
   private double balance = 0.0;
   private String name;
   public BankAccount(String n){name = n;}
   synchronized void deposit(double amt) {
      balance = balance + amt;
   }
   synchronized boolean withdraw(double amt){
      if (balance>=amt) {
            balance = balance - amt;
            return(true);
         }
         else return(false);
      }
   synchronized double checkBalance() {
         return balance;
      }
}
class Customer extends Thread {
   private BankAccount acct;
   private String name;
   Customer(BankAccount b, String n) { acct = b;name = n;}
   public void run() { // simulate some account activity
      for (int j = 0;j < 3;j = j + 1) {
         double x =  Math.random()*100;
         acct.deposit(x);
         System.out.println(name+" lodged: "+x);
         x = Math.random()*50;
         acct.withdraw(x);
         System.out.println(name+" withdrew: "+x);
         System.out.println (acct.checkBalance());
      }
   }
}
class Bank  {
   public static void main(String[] args) {
      BankAccount b = new BankAccount("Joe&Mary");
      new Customer(b,"Joe")).start(); // two customers ...
      new Customer(b, "Mary")).start(); // ... share an account
  }
}
```

Output from the program might be:



```
class BankAccount {
    private double balance = 0.0;
    private String name;

    public BankAccount(String n){name = n;}

    synchronized void deposit(double amt) {
        balance = balance + amt;
    }

    synchronized boolean withdraw(double amt){
        if (balance>=amt) {
```

```
Mary lodged: 55.25211236856975
Joe lodged: 9.633641143512428
Mary withdrew: 13.62864113631458
Joe withdrew: 1.6219513080934111
Account: Joe&Mary   Balance: 49.63516106767419
Account: Joe&Mary   Balance: 49.63516106767419
Mary lodged: 40.98222805335659
Joe lodged: 88.74164825596574
Mary withdrew: 38.59215272992776
Joe withdrew: 25.52802512903851
Account: Joe&Mary   Balance: 115.23885951803025
Account: Joe&Mary   Balance: 115.23885951803025
Mary lodged: 95.9467801733262
Joe lodged: 86.17523596224962
Mary withdrew: 19.358214530956324
Joe withdrew: 12.498212660637769
Account: Joe&Mary   Balance: 265.50444846201196
Account: Joe&Mary   Balance: 265.50444846201196
Press any key to continue...
```

Synchronization inhibits parallel activity because it forces other threads to sleep while a single thread is executing. It makes access to the methods of a class sequential. Therefore, it is important to ensure that the cost in terms of execution time of a synchronized block is minimized. This means that threads should **never sleep** inside the scope of a lock and, where possible, costly computations should be done outside the scope of the lock. In light of this second point it is reasonable to ask: what minimal requirements must be met when coding atomic blocks. The answer is that **all compound actions must be atomic and all class invariants must be preserved**. By class invariant we mean any formal relationship between attribute values that must be preserved. For example, suppose we have two integer attributes x, y such that the values of both variables must be the same, i.e. x == y. Therefore, all changes to x must be reflected in y. Consequently, the sequence x = x + a; y = y + a must be atomic because the class invariant must be preserved. Similarly, suppose we have an array f and a variable n such that n == current number of values in f indexed 0..n-1. A method that inserts a new value in f would have, at a minimum, the synchronized code block:

```
insert(int x){
  …
  synchronized(this){
    f[n] = x;
    n = n + 1;
  }
```

```
        ...
    }
```

This atomic action preserves the class invariant.

Therefore, to preserve state consistency we must ensure that all updates to related variables are performed as atomic actions managed by the same lock.

Visibility

In a single threaded world when you write to a variable and subsequently read it, without modifying it in the intervening code, the value is unchanged and is the value you expected it to be. However, in a multi-threaded world this guarantee is broken. *There is no guarantee that a reading thread will see a value written by another thread in a timely basis, or even at all*. (Goetz, et al, 2006. See also Bloch, 2008, Item 66). To ensure that variable values are visible we must use synchronization. Compilers, processors and runtime environments don't guarantee that order is preserved and using caches create multiple copies of variables. This means that without synchronization it is impossible to reason about the order in which memory events happen. As a consequence, the sequence: `x = x + 1; y = x + 2;` could be executed in any order. Synchronization can be used to ensure that the execution is in fact sequential. If two threads `T1` and `T2` share a synchronized block or method and `T1` gets the lock first, and hence, executes before `T2`, then all changes brought about by `T1` are visible to `T2` when it gets the lock.

The astute reader may now be asking: *but this means that maybe our attempts to terminate a thread may fail?* Remember in our earlier discussion on terminating threads we extended the thread class and added a `terminate()` method that changed the value of a Boolean variable used in a control loop inside the `run` method. This `terminate` method set the global variable `go` to `false` and, as a consequence, forced the loop to terminate, terminating the thread in the process. The thread in question was called `EggTimer` and is listed below.(The guard on the control loop has been modified slightly to emphasize the point being made) The problem with this solution is that the result of a `terminate()` invoked by a controlling thread may never become *visible* to the instance of the `EggTimer` thread. Hence, it may loop forever! One way to ensure this cannot happen is to `synchronize` the method.

Another solution to this problem is to declare the variable `go` as `volatile`. Doing so, guarantees that the thread will *see* the change. However, it is important to note that declaring a variable as `volatile` does not protect it from concurrent modification. In other words, it does not make it *atomic*. It only makes it *visible*. To declare it as volatile use: private volatile boolean go = true;.

```
class EggTimer extends Thread{
  private int et = 0;
  private boolean go = true;
  public void run(){
    while(go){
      try{
        this.sleep(1000); //1000 milliseconds == 1 sec
      }catch(InterruptedException e){}
      et++;
      System.out.print(et+" ");
    }
  }
  public void terminate(){ go = false; }
}
```

The important point to note here is that synchronizing on a common lock guarantees both *atomicity* (mutual exclusion) and memory *visibility*. Declaring attribute variables in threads as *volatile* only guarantees memory *visibility*.

Static Methods

Classes can have class constants, class variables and class methods that belong to the class directly and that can be invoked independently of any instance of the class. In this sense a class is used to group together a set of variables and methods under a common name. The Math class defined in java.lang is such a class. To implement these variables and methods we use the static modifier. The class Counter below has both a static attribute variable and a static method.

```
class Counter{
    private static int next = 0;
    public static int getNext(){
        next++;
        return next;
    }
}
```

This class not thread safe. To make it thread safe it is not enough to declare its attribute next as volatile because this would not guarantee that getNext was atomic. Hence, we synchronize getNext(). This gives the class:

```
class Counter{
   private static int next = 0;
   synchronized static int getNext(){
      next++;
      return next;
   }
}
```

It is, however, possible to mix static and non-static members in a class. This raises issues for sharing access to static and non-static members. We know that static methods cannot refer to non-static variables or methods simply because they do not exist, i.e. are not members of the class itself. However, instance methods may refer to static variables. What does this mean in the context of multi-threaded access and the requirement that classes be thread safe? In fact, it has significant consequences because there are two locking mechanisms involved: one at the Class level and the other at the instance level. What we know is that:

- If two processes attempt to invoke (the same or different) synchronized non-static methods at the same time *in the same object*, then they will execute their methods in some sequential order. Hence they will not destructively interfere with the non-static data in that object.
- If two processes attempt to invoke (the same or different) static synchronized methods at the same time, then they will execute their methods in some sequential order. Hence they will not destructively interfere with the static data.

But we have no guarantees if a non-static method references static class data. The situation is that if one process invokes a synchronized non-static method, and another process executes a static synchronized method in the same class, then both processes potentially have simultaneous access to the static data of the class.

To illustrate this we consider the following version of the Counter class. The class now has two synchronized methods, one static and the other non-static, both modifying the static variable next. It is possible for two threads to simultaneously access the class and modify, in parallel, the attribute next. This is not atomic and may give rise to incorrect results.

```
class Counter{
   private static int next = 0;
```

```
    synchronized static int getNext(){
        next++;
        return next;
    }
    synchronized void reset(){
      next = 0;
    }
}
```

To prevent this it is necessary for the non-static method to acquire the `Class` lock before modifying `next`. The `getClass()` method (`java.lang.Object.getClass`) returns the runtime class of an object. This is the object that is locked by the static synchronized methods of the represented class. By acquiring this lock non-static methods will make modifications to static values thread safe. The `reset` method, listed in the amended version below, uses the class lock to provide thread safety.

```
class Counter{
    private static int next = 0;
    synchronized static int getNext(){
        next++;
        return next;
    }
    public void reset(){
      synchronized(this.getClass()){
         next = 0;
      }
    }
}
```

Note that if a non-static synchronized method requires access to static data it is possible to do so by synchronizing on the class lock from within the body of its code. This will never deadlock because it is not possible for a static method to invoke a non-static one.

Reentrant Locking

Intrinsic locks are, what is termed, *reentrant*. This means that when a thread acquires a lock and, from within this lock, it tries to invoke another synchronized method or block in the same class then it is allowed to acquire it. This means that locks are acquired on a per thread basis rather than per invocation basis. Without this a thread holding a lock and trying to acquire another lock on the same object would deadlock. The JVM keeps a record of what thread holds the lock and manages reentrant acquisition by the thread. Reentrant locks are also necessary to support sharing locks in the context of inheritance. A synchronized method that overrides a synchronized method in a super class has access to the lock of its super method. To illustrate *reentrant* locks we implement a thread safe finite generic Stack class that manages a stack of a given type T. The implementation should be self explanatory. The interesting point for us is that all the methods are synchronized. But method push needs to check if there is space on the stack for a new value. To do so it uses its synchronized method full(). A thread invoking push will acquire the lock on the instance of the class and then *reenter* the lock when it invokes full. A similar situation exists for threads invoking top and pop.

```
class Stack<T> {
  private T[] stack;
  private int head;
  public Stack(int n){
    stack = (T[])(new Object[n]);
    head = 0;
  }
  synchronized Boolean push(T x){
     if(!full()){
       stack[head] = x;
       head = head + 1;
       return true;
     }
     else return false;
  }
  synchronized void pop(){
    if(!empty())
       head = head - 1;
  }
  synchronized T top(){
    if(!empty())
      return stack[head - 1];
    else
      return null;
  }
  synchronized Boolean full(){
    return head == stack.length;
  }
  synchronized Boolean empty(){
    return head == 0;
  }
}
```

Encapsulation and Sharing

It is important to ensure that classes are constructed in such a way that they do not *leak* references. By this we mean that classes should only publish or return variable types that are immutable to ensure that encapsulation is preserved. This means that classes may return primitive types, references to the immutable classes Integer, String, Double, etc or to any class defined as immutable. In all other cases classes should make defensive copies of the data owned by it and only allow modification under rules defined by the class itself. This means that all access to the data encapsulated by it is controlled through the methods that it makes public. In this way the class can preserve its invariants and provide a thread safe service to its own thread clients by synchronizing actions that must be atomic.

To illustrate this we implement a simple `Library` class that manages a collection of `Book`. An `ArrayList` is used as a data structure to manage the collection of books in the library. This class is not thread safe. However, the encapsulating `Library` class ensures that all operations on the state of the library are *thread safe* by the use of synchronization. The `add` method preserves the class invariant by ensuring that `size` equals the frequency of books in the library. The other methods return information about the current state of the class. Therefore, all its state is guarded by its own intrinsic lock. However, while the `Library` class is *thread safe*, the class `Book` may be problematic on two levels. On one level, if the `Book` class has methods that modify the state of a book then encapsulation is broken by both `add` and `get` methods and, hence, these methods may not be thread safe. On the other level, the public methods of `Book` may not be *thread safe*. To ensure that the `Library` class functions correctly we need to know the answers to these questions. We return to these issues in the next section.

```
class Library{
    private ArrayList<Book> lib;
    private int size = 0; //size == #boks in lib
    Library(){lib = new ArrayList<Book>();}
    synchronized void add(Book bk){
        lib.add(bk); size++;
    }
    synchronized Book get(String t){
        Book sbk = new Book(t);
        if(lib.contains(sbk))
            return lib.get(lib.indexOf(sbk));
        else
             return null;
    }
    synchronized boolean searchTitle(String t){
        return lib.contains(new Book(t));
    }
    synchronized int size(){return size;}
}
```

Immutable Classes

One way to resolve the issues in relation to the Book class discussed above is to implement it as an immutable class. An immutable object is an object whose state cannot be modified after it is constructed. Examples of immutable objects are the String and Integer classes. All the information or data for the class is fixed at the point of construction and cannot change during the lifetime of the object. It turns out that there are some very compelling reasons to write classes that are immutable, especially in a multi-threaded universe. They are easy to design and implement and they are less error prone and secure. Bloch asserts that:

> *Classes should be immutable unless there's a very good reason to make them mutable....If a class cannot be made immutable, limit its mutability as much as possible.(Bloch, 2008)*

To make classes immutable we use the following list of rules.

1) Do not provide any methods that allow a client to modify the class state.
2) Ensure that public methods cannot be overridden. This prevents subclasses from compromising the immutable behaviour of the class. This can be done by making the class final so that other classes cannot inherit or extend it.
3) Make all attributes private.
4) Make all fields final

5) Ensure exclusive access to any mutable components. This ensures that clients cannot access references to mutable objects and, hence, violate encapsulation. If access is required then make defensive copies of the mutable object components.

In a multi-threaded universe immutable objects are automatically thread safe. Their state cannot be modified. Therefore, references to these objects may be shared with impunity between threads. Taking this approach the class Book is implemented as a class that cannot be inherited and cannot be modified. The methods equals, compareTo and hashCode are included to make it Collection compliant. Using this version of the class in the Library class above makes the entire system *thread safe*.

```
final class Book implements Comparable<Book>{
   private final String title;
   Book(String t){
      title = t;
   }
   public String title(){return title;}
   public String toString(){return title;}
   public boolean equals(Object p){
      Book b = (Book)p;
      return(title.equals(b.title));
   }
   public int compareTo(Book b){
      return title.compareTo(b.title());
   }
   public int hashCode(){return 31*title.hashCode();}
}
```

Of course it is not always possible to make all classes immutable. Therefore, we explore what changes are necessary in the event that we make the class Book modifiable by adding a method to change the title of a book. To make it thread safe the methods title, set and toString, now become synchronized.

```
class Book implements Comparable<Book>{
   private String title;
   Book(String t){
       title = t;
   }
   synchronized String title(){return title;}
   synchronized void set(String nt){title = nt;}
   public String toString(){
       synchronized(this){return title;}
   }
   public boolean equals(Object p){
       Book b = (Book)p;
       return(title.equals(b.title));
   }
   public int compareTo(Book b){
       return title.compareTo(b.title);
   }
   public int hashCode(){return 31*title.hashCode();}
}
```

The class Library also has to be modified quite significantly because it now has to protect encapsulation and also allow the title of a book under its control to be modified. Also the methods `equals`, `compareTo` and `hashCode` are not thread safe and may only be used in relation to instances of `Book` owned by it. Method `add` creates its own copy of a book and method `get` returns a copy of an existing book. Hence, encapsulation is preserved and both methods are *thread safe*. To support editing the title of an existing book a new method is added to the class. This method is *thread safe* because its arguments are strings and all changes occur under the control of the intrinsic lock on the class.

```
class Library{
    private ArrayList<Book> lib;
    private int size = 0; //size == #boks in lib
    Library(){lib = new ArrayList<Book>();}
    synchronized void add(Book bk){
        Book b = new Book(bk.title());
        lib.add(b); size++;
    }
    synchronized Book get(String t){
        Book sbk = new Book(t);
        if(lib.contains(sbk)){
            Book b = lib.get(lib.indexOf(sbk));
            return(new Book(b.title()));
        }
        else
             return null;
    }
    synchronized boolean editBk(String oldT, String newT){
         Book bk = new Book(oldT);
         Book fbk = null;
         if(lib.contains(bk))
            fbk = lib.get(lib.indexOf(bk));
         if(fbk != null){
            fbk.set(newT);
            return true;
         }
         else return false;
     }
    synchronized boolean searchTitle(String t){
        return lib.contains(new Book(t));
    }
    synchronized int size(){return size;}
}
```

Note: In the case of the immutable Book class to modify (editBk()) an existing title in the Library it would be necessary to create a new book, delete the old one and replace it with the new one.

Coarse-Grained versus Fine Grained Locking

The more a program synchronizes in such a way as to inhibit concurrency, the more *coarse-grained* it is said to be. In contrast, code that synchronizes in such a way as to allow as much concurrency as possible is said to be *fine-grained*. Intrinsic locking in the case of classes is *coarse-grained* because it only permits a single thread at a time access to the resource. This, of course, is necessary to protect atomic actions on the data. However, there are certain circumstances when more than a single thread may access safely the data encapsulated by a class. There are degrees of access and the more access allowed the more *fine-grained* the locking becomes. For example, suppose a class contains an array data with methods accessing only one element of data at the time. It is a *coarse-grained* solution simply to make each method synchronized, because then when multiple threads wish to access different elements of data, one is given access and all the rest are needlessly held up. A *coarse-grained* solution, that assumes all parameter indexes are valid, is given below.

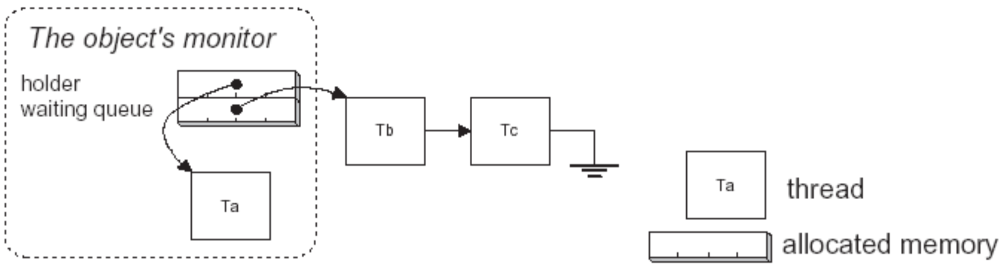
```
class SharedArray {
    int data[];
    public SharedArray(int f[]){
       data = new int[f.length];
        for(int j = 0; j < f.length;j++){ data[j] = f[j];}
    }
    synchronized void assign(int j, int x) { data[j] = x;}
    synchronized void inc(int j, int x) { data[j] = data[j] + x;}
    synchronized  int get(int j){
       return data[j];
    }
}
```

A *fine-grained* implementation optimizing thread access uses an array of Object to provide a lock for each element in the array. Multiple threads accessing the same element are forced to wait on a given element lock. This guarantees thread safety while optimizing thread access.

```
class SharedArray{
    int data[];
    Object locks[];
    public SharedArray(int f[]){
        data = new int[f.length];
        locks = new Object[f.length];
        for(int j = 0; j < f.length;j++){
          data[j] = f[j];
          locks[j] = new Object();
        }
    }
    public void assign(int j, int x){
       synchronized(locks[j]){
          data[j] = x;
       }
    }
    public void inc(int j, int x){
       synchronized(locks[j]){
        data[j] = data[j] + x;
       }
    }
    public int get(int j){
        synchronized(locks[j]){
       return data[j];
        }
    }
  }
```

Note on Monitors

The Java solution to synchronisation and mutual exclusion is based on the concept of a monitor developed independently by Hoare and Hansen in 1973. The Java implementation differs in a number of respects from the original solution. In Java every object that is created, inherits the monitor functionality from the Object class. The JVM doesn't solve problems like dead-locks and starvation. Any class that is declared synchronized, or that contains methods declared as synchronized, will function as a monitor. Every time a monitor is created, the JVM creates an additional C-struct, called `syncobject`, to handle the monitor functionality of that object. A monitor gives a thread exclusive access to the data in the monitor i.e. no other thread may enter the monitor and use the data inside it. Another thread may enter the monitor when the thread occupying the monitor leaves it. The monitor maintains a waiting queue where all the threads that want to enter the monitor are placed. The diagram shows the monitor and its waiting queue.



Thread Ta has access to the monitor. There are two waiting threads, Tb and Tc. They are suspended and will be asked to resume their operation as they enter the monitor.

Problems with Locking

Locking is necessary to ensure thread safety when sharing resources. However, it is also problematic because it can cause systems to stop functioning. The most common causes of this malfunctioning are: **deadlocks, starvation** and **livelock**. We look at each of these in turn.

Two threads are said to be **deadlocked** if each thread holds a resource that the other is waiting for. For example, thread *Ta* holds the lock on the printer *Pr* and tries to acquire the lock on file *Fl* and thread *Tb* holds the lock on the file *Fl* and tries to acquire the lock on printer *Pr*. Both threads need the others resource to proceed. Hence, they will wait forever.

It has been shown (Coffman et al) that four conditions must hold for deadlock to take place:

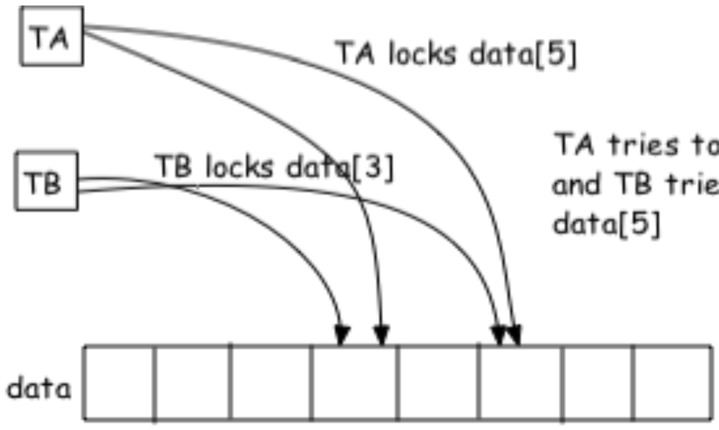
5. Mutual exclusion condition: each resource is assigned to exactly one process or is available
6. Hold and wait condition: processes currently holding resources granted earlier can request new resources;
7. No pre-emption condition: resources previously granted cannot be taken away from a process - only the process holding them can release them;
8. Circular wait condition: there must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

All four conditions must be present for deadlock to occur. If one of them is absent, no deadlock is possible.

Different strategies are used to avoid deadlock situations. One possible solution is to ensure that *resource ordering* is guaranteed, i.e. threads always request resources in a given order. This makes a circular chain impossible and, hence, prohibits condition 4 above.

To illustrate this type of scenario we consider solving the problem of writing a concurrent program that shuffles the elements in an array of size *N*, $N \geq 0$. Elements in the array are shuffled by swapping them with other elements in the data set.

The task of swapping two elements needs to be atomic and, in order to maximize thread access, we guard each element with a lock. Locking the entire array for each swap would make the solution sequential. To swap a pair of elements a thread needs to acquire two locks. As a consequence, the scenario described in the diagram may occur and the threads may deadlock. Thread TA locks data[5] and then tries to lock data[3]. But TB holds the lock on data[3] and tries to acquire the lock on data[5]. The threads now find themselves in deadlock.



One solution to this problem is to use *resource ordering*. Each thread always ensures that the left index is less than the right index. This prohibits cycles and, hence, makes deadlock impossible. The class Shuffler implements this strategy. It repeatedly chooses two random index values ensuring that left is less than right, locks the indexes in order left followed by right and then swaps the values.

```
class Shuffler extends Thread{
   int data[]; int shuffles;
   Object locks[];
   public Shuffler(int d[], Object lk[],int sh){
      data = d; shuffles = sh; locks = lk;
   }
   public void run(){
      int k = 0;
      while(k < shuffles){
       //select two random index values and exchange data
         int left = (int)(Math.random()*data.length);
         int right = (int)(Math.random()*data.length);
         while(left >= right){
            left = (int)(Math.random()*data.length);
            right = (int)(Math.random()*data.length);
         }
```

```
            //left < right
            synchronized(locks[left]){
                synchronized(locks[right]){
                    int temp = data[left]; data[left] = data[right]; data[right] = temp;
                }
            }
            k++;
        }
    }
}
```

A program that shuffles the elements in a shared array using two Shuffler threads is given below. Both threads share the data array and the Object array used to enforce locking.

```
public class ArrayShufflerTest {
    public static void main(String[] args) {
        int data[] = new int[1000];
        Object locks[] = new Object[data.length];
        for(int j = 0; j < locks.length; j++)
            locks[j] = new Object();
        for(int j = 0; j < data.length; j++){
          data[j] = (int)(Math.random()*100);
          System.out.printf("%2d",data[j]);
        }
        System.out.println();
        Thread t1 = new Shuffler(data,locks,100);
        Thread t2 = new Shuffler(data,locks,200);
        t1.start(); t2.start();
        try{t1.join(); t2.join();}
        catch(InterruptedException e){}
        for(int j = 0; j < data.length; j++)
              System.out.printf("%2d",data[j]);
        System.out.println();
    }
}
```

Starvation

Starvation occurs when a process cannot gain access to required resources. It can happen for a number of reasons. Consider the case where two processes A and B have different priority levels. Suppose A has a higher priority than B. If A gains access to a resource required by B it could hold it forever and, as a consequence, starve B. Another example might be taken from the Dining Philosophers problem:

Five philosophers are seated around a table. Each philosopher has a plate of spaghetti. To eat the spaghetti a philosopher needs two forks two chopsticks. Between each plate is a single chopstick. The life of a philosopher consists of alternate periods of eating and thinking. To eat a philosopher requires the chopstick to the right and left. Now suppose each of the five philosophers pick up the chopstick to their right they will all starve unless some are prepared to lay down their chopstick and wait!

Livelock

In sequential programs we are all familiar with infinite loops - the program just goes off and never comes back! This can also happen in parallel systems where the system is communication internally and fails to respond to external stimuli. From a user's perspective livelock and deadlock appear similar but are in fact different things.

The Lock Class

The Lock interface and its associated Lock classes provide an alternative to synchronized blocks. Locks offer a choice of unconditional, polled, timed and interruptible lock acquisition, and all lock and unlock operations are explicit. The Lock interface is defined as follows:

```
public interface Lock{
   void lock();
   void lockInterruptibly() throws InterruptedException;
   boolean tryLock();
   boolean tryLock(long timeout, TimeUnit unit)
        throws InterruptedException;
   void unlock();
   Condition newCondition();
}
```

The class `ReentrantLock` implements the `Lock` interface and provides the same basic behaviour as `synchronized`. The semantics for acquiring and releasing a

ReentrantLock is the same as entering and exiting a synchronized block (or method). At most one thread at a time can hold a ReentrantLock. It provides both atomicity (mutual exclusion) and memory visibility.

The pattern for using locks is:

```
Lock lock = new ReentrantLock();

…

lock.lock();

try{

   // synchronized block

}finally{  //ensures lock is released

   lock.unlock();

}
```

The reason for try{..}finally{..} is to ensure that the lock is always released, even if the code block throws an exception.

The main reasons for introducing a separate Lock class is that intrinsic locking has some limitations. Once a thread attempts to acquire a synchronized lock it cannot escape and, hence, may wait forever and it is not possible to interrupt a thread sleeping on a lock. Also they must be released in the same code block that acquires them.

To illustrate this form of locking we implement an automated booking system for a theatre. The system must support multiple users that are allowed to book a single seat at a time from a list of available seats. The system must ensure that double booking is not permitted whilst allowing clients free choice of available seats. This means that a seat may appear to be free although it may be booked or in the process of being booked by another client. Implement a class that allows users to book seats whilst not permitting double booking.

```
class Theatre{
    private boolean seats[];
    private int freeSeats;
    private ReentrantLock lock = new ReentrantLock();
    public Theatre(int n){
        seats = new boolean[n];
        for(int j = 0; j < n;j++) seats[j] = true; // true = available
        freeSeats = n;
    }
    public int[] getSeats(){
        // find list of free seats
        lock.lock();
        try{
            int free[] = new int[freeSeats];
            int k = 0;
            for(int j = 0; j < seats.length; j++){
                if(seats[j]){
                    free[k] = j;
                    k++;
                }
            }
            return free;
        }finally{
            lock.unlock();
        }
    }

    public boolean bookSeat(int k){
        lock.lock();
        try{
            if(seats[k]){
                seats[k] = false;
                freeSeats--;
                return true;   //"Booked";
            }
            else
              return false; //"Seat not available now";
        }finally{
            lock.unlock();
        }
    }
}
```

Fairness

ReentrantLock does not provide a fair lock in the sense that it does not guarantee first come first served. When a thread releases a lock on which several threads are blocked, there are no guarantees as to which thread will be next to acquire the lock.

If fairness is a requirement, then a fair lock can be created by:

```
new ReentrantLock(true);
```

This creates a fair lock which is allocated to processes on a FIFO (first-come-first-served). However, it is computationally more expensive than the standard locking.

Conditional Acquisition

This means locks can be acquired on a conditional basis using the tryLock() method. If all the required locks cannot be acquired the developer has control to decide what action to take, e.g. release all the locks that were acquired. This facilitates error recovery and could help avoid deadlock. **tryLock()** acquires the lock only if it is free at the time of invocation. It acquires the lock if it is available and returns immediately with the value true. If the lock is not available then this method will return immediately with the value false.

A typical pattern for this method would be:

```
 Lock lock = ...;
if (lock.tryLock()) {
   try {
      // manipulate protected state
   } finally {
      lock.unlock();
   }
} else {
   // perform alternative actions
}
```

This usage ensures that the lock is unlocked if it was acquired, and doesn't try to unlock if the lock was not acquired.

We can now re-visit our shuffling thread written using resource ordering to prohibit deadlocks. Using tryLock it is no longer necessary to use resource ordering because it is possible to check if the second lock is free and, if not, free previously acquired resources.

```
class Shuffler extends Thread{
    int dd[];int shuffles;
    Lock locks[];
    public Shuffler(int d[], Lock ll[],int sh){
        dd = d; shuffles = sh; locks = ll;
    }
    public void run(){
        int k = 0;
        while(k < shuffles){
            //select two random index values and exchange data
            int g = (int)(Math.random()*dd.length);
            int h = (int)(Math.random()*dd.length);
            while(g == h){
                g = (int)(Math.random()*dd.length);
                h = (int)(Math.random()*dd.length);
            }
            locks[g].lock();
            try{
                if(locks[h].tryLock()){
                    try{
                       int temp = dd[g]; dd[g] = dd[h];
                       dd[h] = temp;
                    }
                    finally{ locks[h].unlock();}
                }
            } finally{ locks[g].unlock();}
            k++;
        }
    }
}
```

The use of tryLock() in the given code block below ensures that deadlock is avoided.

Timed Locks

Locks can also be acquired on a timed basis, meaning, an acquisition of that lock is only tried for the specified time period. Timed locks are useful in avoiding potential deadlock situations. The acquiring thread waits only for a specified period of time to acquire the lock.

```
boolean tryLock(long time,TimeUnit unit)
            throws InterruptedException
```

Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted.

If the lock is available this method returns immediately with the value `true`. If the lock is not available then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of three things happens:

- The lock is acquired by the current thread; or
- Some other thread interrupts the current thread, and interruption of lock acquisition is supported; or
- The specified waiting time elapses

If the lock is acquired then the value `true` is returned.

Exercise

Question 1

Consider the following example where one thread repeatedly swaps two arbitrary elements of an integer array, and the other repeatedly displays the total of the elements in b. When the program is run we may get a list of different totals displayed. Explain why this can happen and re-write the solution so that the list of totals is correct.

```
class ShowInterfTest{
  public static void main(String[] args) {
      int data[] = new int[100];
      Runnable summer = new Summer(data);
      Swapper swapper = new Swapper(data);
      Thread tSwapper = new Thread(swapper);
      Thread tSummer = new Thread(summer);
      // initialise the data
      for(int j = 0; j < data.length; j = j + 1){
       int t = (int)(Math.random()*100);
       data[j] = t;
      }
      tSummer.start();
      tSwapper.start();
      //wait for tSummer to finish
      try { tSummer.join();
      }
      catch(InterruptedException e) {}
      // now stop tSwapper
      swapper.stopSwap();
  }
}
class Summer implements Runnable {
   // display the sum of the elements of b
   private int data[];
   public Summer(int b[]){
    data = b;
   }
   public void run() {
      for (int j = 0;j < 10;j = j +1) {
         int total = 0;
         for (int i=0; i < data.length; i++){
            try{
               Thread.sleep(2);
            }catch(Exception e){}
            total = total + data[i];
         }
         System.out.println("Sum = "+total);
      }
   }
}
class Swapper implements Runnable {
   // repeatedly swap two arbitrary elements of b
   private int data[];
   private volatile boolean go = true;
   public Swapper(int b[]){data = b;}
   public void stopSwap(){go = false;}
   public void run() {
      int j, k, t;
      while (go) {
            j = ((int)(Math.random()*1000)) % data.length;
            k = ((int)(Math.random()*1000)) % data.length;
            t = data[j]; data[j] = data[k]; data[k] = t;
      }
   }
}
```

Question 2

A large ornamental garden is open to the public, who must pay an admission fee to view the beautiful roses, shrubs and aquatic plants. Entry to the garden is by two turnstiles. The management want to know, at any time, the total number of members of the public currently on the grounds. They propose that a computer system be installed that has connections to each turnstile and a terminal from which they can ascertain the current total. Implement such a system. You may assume that the garden has an unlimited capacity!

Question 3

Implement a shared bank account using locks.

Question 4

Class Queue below describes a queue of integers, i.e. a first-in-first-out list, intended for use in a concurrent environment. However, the shared variables are not safe from concurrent access. Using locks modify it to make it thread safe.

```
class Queue <T> {
   private int[] b =  (T[])(new Object[1000]);
   private int tail = 0; int head = 0;
   public void put(T x) {
         // Insert x into b, waiting if b full
         while(head-tail==1000) {
            try{sleep(10);} catch(InterruptedException e) {};
         }
         b[head%1000] = x;
         head++;
   }
   public T take() {
         // remove oldest element from b, waiting if b empty
         while(head==tail) {
            try{sleep(10);} catch(InterruptedException e) {};
         }
         T x = b[tail%1000]; tail++;
           return x;
   }
}
```

Question 7

With respect to class BankAccount (Question 4 above) add a method

```
boolean transferFrom(BankAccount acct; int amt)
```

which transfers amt money from account acct to this account, returning a boolean to indicate success or failure. Ensure that your solution is deadlock free.

Sharing Limited Resources

In the previous chapter we examined issues of sharing so that both visibility and mutual exclusion were guaranteed and we also discussed issues around deadlocks and how to avoid them. This, however, is not the whole story. There are situations where we need to share limited resources. Doing so will involve all the issues around sharing discussed previously with the added complication that the object being shared may have limitations on accessing the resources it manages. For example, a stadium will have a maximum capacity and to protect users of the stadium we must ensure that the number of occupants never exceeds this value. If you think of the stadium as a shared resource that has limited capacity you can see that the problem of managing access is similar to the other examples given. However, it differs in that there is an upper limit in terms of user access. A similar example, from operating systems, is that of a bounded buffer that has some upper limit. When the buffer fills up writers to the buffer must wait for space in the buffer to become available. Similarly, readers may have to wait if the buffer is empty. There are many similar examples where we will need to use limited shared resources and we need protocols to control user access.

Monitors and Condition Variables

A large ornamental garden is open to the public, who must pay an admission fee to view the beautiful roses, shrubs and aquatic plants. Entry to the garden is by two turnstiles. The management want to know, at any time, the total number of members of the public currently on the grounds. There is an upper limit on the number of people who may enter the garden at any one time. Visitors wishing to access the garden when it is full will have to wait for people to leave before gaining access. In general programming terms threads wishing to access limited resources will, under certain pre-defined conditions, have to wait for the resource to become available. The classical name for a program construct that supports thread waiting on limited resources is a Monitor. In Java one way monitors are implemented is through condition variables.

By adding a condition variable to a lock it is possible to allow one thread to suspend execution, i.e. wait, until another thread notifies it that some condition is now true. The terms used in condition variables are `await()` and `signal()` or `signalAll()`.

To create a condition variable for a lock:

```
   Condition cond  = lock.newCondition()
```

To wait on cond use

```
   try {cond.await();} catch (InterruptedException e) {}
```

To wake up a thread(s) waiting in cond use

   cond.signal() - wake up one waiting thread
or
   cond.signalAll() – wake up all waiting threads.

A thread executing cond.await() within a lock has the effect of releasing the lock on the block, and of suspending execution of the thread. A thread executing cond.signal() within a lock has the effect of resuming execution of a single thread suspended in this lock If there are no suspended threads, then signal() has no effect. If there are many, then the thread that has been waiting longest on the condition variable is awoken. However, this does not guarantee that it will get access to the resource. It is important to note that **a resumed thread has to regain entry to the block before it actually continues.** It may fail to get it and as a result must go back to the waiting state. This means that waits must be enclosed in a loop as follows:

```
while(…){
try{
 cond.await();
}
catch(InterruptedException e){}
```

The examples that follow will illustrate how to use monitors.

Example 1

Consider the garden problem given above where there is a restriction on the number of visitors that can be in the garden at any one time. Suppose there is an upper limit of 10. When this figure is reached new visitors must wait for someone to leave before gaining access.

There are two turnstiles and each one must enforce the upper limit. The code is:

```
   public void T1(){
      lock.lock();
      try{
        while(count == max){
         try{
             notFull.await();
         }
         catch(InterruptedException e){}
       }
       count++;
     }finally{
         lock.unlock();
     }
 }
```

Note the use of the while loop to allow for the possibility that the thread may fail to gain entry when woken.

In the case of **leave()** we only need to invoke signal when there is the possibility that another thread may be sleeping on a wait event. Hence, the use of:

```
public void leave(){
     lock.lock();
     try{
         count--;
         if(count == max - 1)
           notFull.signal();
     }finally{
         lock.unlock();
     }
 }
```

The complete solution is given below.

```
import java.util.concurrent.locks.*;
class Garden{
   private int max = 10;
   private int count = 0;
   private Lock lock = new ReentrantLock();
private Condition notFull = lock.newCondition();
public void T1(){
    lock.lock();
    try{
      while(count == max){
        try{
            notFull.await();
        }
        catch(InterruptedException e){}
      }
      count++;
    }finally{
        lock.unlock();
    }
}
public void T2(){
    lock.lock();
    try{
      while(count == max){
        try{
            notFull.await();
        }
        catch(InterruptedException e){}
      }
      count++;
    }finally{
        lock.unlock();
    }
}
public void leave(){
    lock.lock();
    try{
        count--;
        if(count == max - 1)
          notFull.signal();
    }finally{
            lock.unlock();
        }
    }
}
class GardenTest{
    public static void main(String args[]){
        Garden g = new Garden();
        for(int j = 0; j < 17;j++) new Visitor(g,j).start();

    }
}
class Visitor extends Thread{
    Garden garden; int num;
    public Visitor(Garden g, int j){
        garden = g;num = j;
    }
    public void run(){
        // chose a turnstile
        int t = (int)(Math.random()*2);
        if(t == 0)
          garden.T1();
        else
          garden.T2();
        //stay for a random period of time
        System.out.println("Visitor "+ num+" entered");
        try{
            t = (int)(Math.random()*10000);
            Thread.sleep(t);
        }
        catch(InterruptedException e){
        }
        garden.leave();
        System.out.println("Visitor "+ num+" left");
    }
}
```

Example 2

A class is required to manage the allocation of 10 ports to clients. When a client requests a port it receives a port number, if one is available. In the event that no port is available the client waits indefinitely for one to become free. Once a port is allocated no other client can use it. When a client finishes using a port it frees it. Implement the port class and write client threads to that interact with it by requesting port numbers.

```
class Ports{
    boolean ports[] = new boolean[10];
    int available = 10;
    Lock lock = new ReentrantLock();
    Condition port = lock.newCondition();
    public Ports(){
        for(int j = 0; j < 10;j++) ports[j] = true; // all available
    }
    public int getPort(){
        lock.lock();
        try{
            while(available == 0){
              try{
                port.await();
              }catch(InterruptedException e){}
            }
            //search for free port
            int k = 0;
            while(!ports[k]) k++;
            ports[k] = false;
            available--;
            return k;
        } finally{ lock.unlock();}
    }
    public void freePort(int k){
        lock.lock();
        try{
            ports[k] = true;
                            available++;
            port.signal();
        } finally{lock.unlock();}
    }
}
```

As an exercise, write a program to test this class implementing clients as threads that request ports.

Bounded Buffer - Producer Consumer Problem

Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out. Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up. Write a class that implements a solution to this problem and write a routine to test it.

We implement a generic Buffer class that uses an ArrayList as a data structure to manage the actual queue or buffer. There are two public methods that force a producer to wait when the buffer is full and a consumer to wait when the buffer is empty. Note that two condition variables are used: notFull for the producer and notEmpty for the consumer. This means that two separate wait queues are maintained one each for the producer and consumer

```
class Buffer<E>{
  private int max;
  private int size = 0;
  private ArrayList<E> buffer;
   private Lock lock = new ReentrantLock();
   private Condition notFull = lock.newCondition();
   private Condition notEmpty = lock.newCondition();
  public Buffer(int s){
    buffer = new ArrayList<E>();
    max = s;
  }
  public void put(E x){
   lock.lock();
   try{
       while(size == max){
           try{
               notFull.await();
           }catch(InterruptedException e){}
       }
       buffer.add(x);
       size++;
       if(size - 1 == 0) notEmpty.signal();
   }finally{lock.unlock();}
  }
  public E get(){
   lock.lock();
   try{
       while(size == 0){
           try{
               notEmpty.await();
           }catch(InterruptedException e){}
       }
       E temp = buffer.get(0);
       buffer.remove(0);
       size--;
       if(size + 1 == max) notFull.signal();
       return temp;
   } finally{lock.unlock();}
  }
}
```

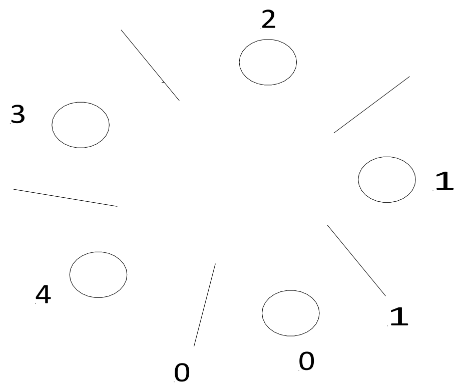
A test program is given below. The producer simply generates and writes 50 consecutive values to the buffer. The consumer removes exactly the same number. A good test is to change the frequency of both the producer and consumer. In the test below the producer puts a value every 100 milliseconds and the consumer tries to get a value every 500 milliseconds. The producer is working 5 times faster than the consumer so the buffer will never be empty. Change this around to test for an empty buffer. What would happen if the consumer tried to remove more elements than the producer generated?

```
import java.util.*;
class ProducerConsumer{
    public static void main(String args[]){
        Buffer <Integer> buffer = new Buffer<Integer>(10);
        new Thread(new Producer(buffer)).start();
        new Thread(new Consumer(buffer)).start();
    }
}
class Producer implements Runnable{
    Buffer <Integer> buffer;
    public Producer(Buffer <Integer> k){
        buffer = k;
    }
    public void run(){
        for(int j = 0; j < 50; j++){
            Integer x = new Integer(j);
            buffer.put(x);
            //set production speed
            try{
                Thread.sleep(100);
            }catch(InterruptedException e){}
        }
    }
}
class Consumer implements Runnable{
    Buffer <Integer> buffer;
    public Consumer(Buffer <Integer> k){
        buffer = k;
    }
    public void run(){
        System.out.print("Buffer data: ");
        for(int j = 0; j < 50; j++){
            Integer x = buffer.get();
            System.out.print(x+" ");
            //set consumption cycle
            try{
                Thread.sleep(500);
            }catch(InterruptedException e){}
        }
    }
}
```

The Dining Philosophers Problem

Five philosophers are seated around a table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two chopsticks to eat it. Between each plate is a chopstick. The life of a philosopher consists of alternate periods of eating and thinking. To eat a philosopher requires the chopstick to the right and left. Now suppose each of the five philosophers pick up the chopstick to their right they will all starve unless some are prepared to lay down their chopstick and wait!

The following diagram illustrates the philosophers sitting at a round table.



Each philosopher has a number and the chopstick to the left of each philosopher has the same number. In the diagram they are numbered 0..4. Each philosopher can be viewed as having two states: *thinking and eating*. The life of a philosopher consists of transitions from one state to the next. That is: *thinking* followed by *eating* followed by *thinking* .. Any solution to the problem will involve modelling these states and their transitions. The problem is interesting because if each philosopher picks up the chopstick to their left and waits for the one on their right they will all starve to death! To avoid this situation we must ensure that a philosopher always picks up both chopsticks simultaneously, i.e., he waits for both to become available and gets both instantly. Therefore, at any one time at most two philosophers are eating. When a philosopher is eating, i.e., has both chopsticks, he cannot be interrupted. In the following solution we model each state:

Thinking

Each philosopher will think for a fixed amount of time. This can be modeled by using a private method with a delay.

```
    private void think(int phil){
      System.out.println("Philospher "+phil+"  is thinking");
      try{
        Thread.sleep(5000);
       }catch(InterruptedException e){}
    }
```

Eating

Each philosopher will eat for a fixed amount of time.

```
   private void eat(int phil){
     System.out.println("Philospher "+phil+" is eating");
     try{
         Thread.sleep(3000);
     }catch(InterruptedException e){}
  }
```

Philosopher

A philosopher can be modeled with the use of a thread for each one. The code just describes the philosopher's life!

```
class Philosopher extends Thread{
   private int phil;  // philosopher id
   public Philosopher(int p){
       phil = p;
   }
   public void run(){
       while(true){
          think(phil);
          //pickup chopsticks
          eat(phil);
          // putdown chopsticks
       }
   }
```

Chopsticks

The philosophers must share the five chopsticks to avoid starving to death. A chopstick is a shareable resource that can only be used by a single philosopher at the time. From our perspective, we do not care what it looks like or how long it is – we only care about whether or not it is in use. Therefore, we implement it as a boolean with two public methods: pickUp() and putDown()

```
class ChopStick{
    private boolean inuse;
    Lock lock = new ReentrantLock();
    Condition notInUse = lock.newCondition();
    public ChopStick(){
        inuse = false;
    }
    public void pickUp(){
        lock.lock();
        try{
            while(inuse){
                try{
                    notInUse.await();
                }catch(InterruptedException e) {}
            }
            inuse = true;
        }finally{lock.unlock();}
    }
    public void putDown(){
        lock.lock();
        try{
            inuse = false;
            notInUse.signal();
        }finally{lock.unlock();}
    }
}
```

This allows us to re-write the life of the philosopher as follows:

```
class Philosopher implements Runnable{
    private ChopStick ch1, ch2; // chopsticks
    private int phil;  // philosopher id
    public Philosopher(int p, ChopStick left, ChopStick right){
        phil = p;
        ch1 = left;
        ch2 = right;
    }
    public void run(){
        while(true){
           think(phil);
           //pickup chopsticks
           ch1.pickUp();
           ch2.pickUp();
           eat(phil);
           // putdown chopsticks
           ch1.putDown();
           ch2.putDown();
        }
    }
```

Deadlock

Philosophers have the power to create a deadlock situation. They could each grab the chopstick on their left and refuse to give it up! They also have the power to eat forever, hence forcing others to think forever! Both problems can be solved by ensuring that at most n-1 philosophers can be eating, at any one time. By stopping one philosopher from eating deadlock is prevented and since only one philosopher is delayed (and thereby freed when a single philosopher stops eating) liveness is preserved.

```
class DeadLockPrevention{
    volatile int max;
    volatile int eating = 0;
    Lock lock = new ReentrantLock();
    Condition eat = lock.newCondition();
    public DeadLockPrevention(int n){
             // n = number of philosophers
           max = n-1;
    }
    public void enters(){
        lock.lock();
        try{
            while(eating == max){
                try{
                    eat.await();
                }catch(InterruptedException e){}
            }
            eating = eating + 1;
        }finally{lock.unlock();}
    }
    public void leaves(){
        lock.lock();
        try{
            eating = eating - 1;
            eat.signal();
        }finally{lock.unlock();}
    }
}
```

By integrating deadlock prevention into the life of a philosopher we can ensure a happy and contented infinite cycle of thinking and eating!

```
import java.util.*;
class Philosopher implements Runnable{
    private ChopStick ch1, ch2; // chopsticks
    private int phil;  // philosopher id
    private DeadLockPrevention deadLock;

    public Philosopher(int p, ChopStick left,ChopStick right,
                            DeadLockPrevention d1){

       phil = p;
       ch1 = left;
       ch2 = right;
       deadLock = d1;
    }

    public void run(){
       while(true){
         think(phil);
         // check deadlock
         deadLock.enters();

         //pickup chopsticks
         ch1.pickUp();
         ch2.pickUp();

         eat(phil);

         // putdown chopsticks
         ch1.putDown();
         ch2.putDown();

         //Deadlock prevention
         deadLock.leaves();
       }
    }
    private void think(int phil){
       System.out.println("Philospher "+phil+" is thinking");
       try{
          Thread.sleep(5000);
```

```
        }catch(InterruptedException e){}
    }
    private void eat(int phil){
        System.out.println("Philospher "+phil+" is eating");
        try{
            Thread.sleep(3000);
        }catch(InterruptedException e){}
    }
}
```

The philosophers and their chopsticks are launched by the following code:

```
import java.util.concurrent.locks.*;
class DiningPhilosphers{
    public static void main(String args[]){
        Runnable p0,p1,p2,p3,p4;
        DeadLockPrevention deadlocks = new DeadLockPrevention(5);
        ChopStick chStick0 = new ChopStick();
        ChopStick chStick1 = new ChopStick();
        ChopStick chStick2 = new ChopStick();
        ChopStick chStick3 = new ChopStick();
        ChopStick chStick4 = new ChopStick();

        p0 = new Philosopher(0,chStick0,chStick1, deadlocks);
        p1 = new Philosopher(1,chStick1,chStick2, deadlocks);
        p2 = new Philosopher(2,chStick2,chStick3, deadlocks);
        p3 = new Philosopher(3,chStick3,chStick4, deadlocks);
        p4 = new Philosopher(4,chStick4,chStick0, deadlocks);

        new Thread(p0).start();
        new Thread(p1).start();
        new Thread(p2).start();
        new Thread(p3).start();
        new Thread(p4).start();

    }
}
```

Terminating, Suspending and Resuming Threads

In the normal course of things threads terminate when the run() method completes. A thread can also terminate at any time by invoking `return` from the run() method. However, sometimes we need to give a control program the power to terminate a thread under its control. Java did provide a thread method – `stop()` – that did just this but it has side effects and can leave shared resources in an unstable state. This method has been deprecated. In this section we will explore how we can write threads in such a way that we can terminate them safely.

To avoid leaving the system in an unstable state we should avoid killing threads directly. The goal is to signal threads to terminate themselves.

Use the interrupt mechanism to let the controlling thread send an interrupt to the thread it wants to terminate. The thread is written so that it catches the interrupt and chooses to terminate in its own time. This means that it can complete any critical tasks that it is doing, free any potentially shared resources it has and leave the system in a stable state.

The command `t.interrupt()` sets the interrupt status in thread t. Thread t can test the interrupt using `me.interrupted()` or `me.isInterrupted()`. Both of these return the interrupt status but the former clears it, whereas the latter leaves its status unchanged. In the event that the thread is sleeping on a wait event an `InterruptedException` is thrown by `t.interrupt` and caught by the thread.

The following example illustrates how to do this.

```
class ThreadTerminate{
   public static void main(String args[]){
      Thread t = new Ex1();
      Thread t1 = new Ex2();
      t.start(); t1.start();
      try{
         Thread.sleep(2000);
      }
      catch(InterruptedException e){}
      t.interrupt();
      t1.interrupt();
```

```
    }
}
class Ex1 extends Thread{
    public void run(){
        // use infinite loop to demo working
        boolean b = true;
        while(b){
            //do some work
            if(this.isInterrupted()){
                 System.out.println("Ex1 finished");
                 return;
            }
        }
    }
}
class Ex2 extends Thread{
    public void run(){
        volatile boolean b = true;
        while(b){
            // do some work
            try{
                Thread.sleep(100000);
            }
            catch(InterruptedException e){
               System.out.println("Interrupt caught");
               b = false;
            }
        }
        System.out.println("Ex2 finished");
    }
}
```

Suspend/Resume

The idea is for a control program to be able to suspend a target thread for an indefinite period of time and then allow it to resume.

Java does have two deprecated methods called **suspend()** and **resume()** that can be used to this. However, these have been deprecated because they are deadlock prone. Suspended threads hold all locks when they are dormant. This means that no other thread can access the resource until the suspended thread is resumed and completes. If the thread that would invoke resume should try to access the resource held by the suspended thread prior to calling resume deadlock results.

The problem can be solved by allowing the target thread to suspend itself in its own time. This means that it will not sleep while holding a lock on a monitor. The thread simply polls a Boolean variable indicating the desired state (active or suspended) at certain intervals. When the desired state is suspended then the thread waits to resume using a condition variable. When the thread is resumed, the target thread is notified using resume.signal().

The following example illustrates how to do this. Note that suspended is declared volatile. Why?

```
class TSuspendResume extends Thread{
    private volatile boolean suspended = false;
    private Lock lock = new ReentrantLock();
    private Condition resume = lock.newCondition();
    private boolean go = true;
    public void tSuspend(){
        suspended = true;
    }
    public  void tResume(){
        suspended = false;
        lock.lock();
        try{
          resume.signal();
        }finally{lock.unlock();}
    }
    public void run(){
        while(go){
          // do work
          System.out.println("working");
          lock.lock();
```

```
            try{
                while(suspended)try{
                resume.await();
                }catch(InterruptedException e){}
            }finally{lock.unlock();}
          }
      }
      public void terminate(){go = false;}
  }
```

This test program simply suspends and resumes the thread 5 times and then exits.

```
class ThreadSuspendResume{
    public static void main(String args[]){
        Ex1 t = new Ex1();
        t.start();
        for(int j = 0; j < 5; j++){
            try{
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
            t.tSuspend();
            try{
                Thread.sleep(1000);
            }
            catch(InterruptedException e){}
            t.tResume();
        }
        //exit process & terminates all threads
        System.exit(0);
    }
}
```

Exercise

Question 1

A parent manages a bag of sweets that is shared by all children. The parent repeatedly puts a sweet in a bag. Each child forever takes one out, waiting if there are none. Write a program to simulate the behaviour of the parent and the children.

Question 2

A car park has one entrance, one exit and a simple display device which always prints the message “FULL” or “SPACES”. Design a program that simulates the behaviour of the car park. You can assume a maximum of 500 spaces and that cars enter at regular intervals of say 5.0 seconds and leave at regular intervals of 7.0 seconds. If the car park becomes full the process controlling entry must be suspended until spaces are available.

Question 3

A server manages the allocation of 10 ports to clients. When a client requests a port from the server it receives a port number, if one is available. In the event that no port is available the client waits indefinitely for one to become free. Once a port is allocated no other client can use it. Implement the server class and write client threads to that interact with it by requesting port numbers.

Question 4

A message board is a synchronisation control that may contain only a single message that is available to any thread that requests it. If no message is available then reading threads must wait indefinitely for one to arrive. The control should have three public methods: `read()` that returns the current message, if one is available; `write(…)` that changes the message on the board; `clear()` that clears the current message from the board. Implement a protocol for a message board and also write a test program for your protocol.

Question 5

Implement a pigeon box that holds a single message in each slot. Users extract messages, waiting if necessary, by checking a pigeon box. Multiple users may check for a message in the same box.

Question 6

Write a class called SingleBuffer that holds a single integer value. The class should have two public methods called read() that returns the current value of the buffer and write(k) that writes integer k to the buffer. A thread invoking read() must wait for a value to be written to the buffer. Once an item is read from the buffer it is cleared to allow a writer thread write. Similarly, a writer thread must wait for a reader thread to read the current value and, hence, clear the buffer.

Question 7

Readers/Writers Problem – Write a control that manages access to a given data set. There are two kinds of thread that require access. Readers that need to access information and writers that need to update the data set. The control must enforce the following list of rules:

- only allow one writer at a time;
- readers must be blocked when a writer is writing;
- writers must be blocked when readers are reading;
- allow multiple concurrent readers;
- give writers precedence over readers.

Question 8

The thread used to illustrate suspend/resume above does not take account of interrupts. If a controlling thread throws an interrupt then the target thread is woken up. In the example given this means it will repeat its task and then go back to sleep because its state is still suspended. Re-write it so that in the event of an interrupt it immediately goes back to waiting.

Question 9

Write a thread so that it handles terminate and also suspend/resume.

Question 10

Write a thread that provides a service on asynchronously. Once it has provided the service it immediately suspends itself until called upon to provide the service again. Note that the thread once started should wait to be asked to provide the service. An example of a service might be to print a list of numbers.

Question 11

A platform has space for at most 100 people at any one time. People are only admitted when the platform is open and the number of persons does not exceed the prescribed limit. Write a class that could be used to control access to the platform.

Question 12

Write a program to control access to a merry-go-round at a carnival. The merry-go-round has 20 horses. At most 20 people are allowed access at any one time and horses are chosen by the entrant on a first come first served basis. It should not be possible for two persons to lay claim to the same horse.

Question 13

A semaphore is data structure with two operations `waitSem()` and `releaseSem()`. The `waitSem()` operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value and just continues. If the value is 0, the process is put to sleep. Checking the value, changing it, and possibly going to sleep is all done as a single, indivisible, **atomic action.** It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. The `releaseSem()` operation increments the value of the semaphore. If one or more processes were sleeping on that semaphore, unable to complete an earlier `waitSem()` operation, one of them is chosen by the system (e.g., at random), and is allowed to complete its `waitSem()`. Using wait/notify write a semaphore class.

Question 14

An event object has one of two states signaled or non-signaled. When the event is signaled all or one waiting threads are released. When it is reset to the non-signaled state all user threads will be forced to wait. An event stays in the signaled state until it is reset to the non-signaled state. The basic idea is that one or more threads can wait for some event to happen. When the event waited for occurs the threads take whatever action is required. Using wait/notify write an Event class.

Semaphores, Latches and Barriers

Semaphores

A semaphore is a thread synchronizing data structure invented by Dijkstra to solve, amongst other problems, the mutual exclusion problem (*Cooperating Sequential Processes, 1965*). This control manages a single integer value that is initialized to a non-negative value and has two public methods `acquire()` and `release()`. When a thread invokes the `acquire()` operation on a semaphore it checks to see if the value is greater than 0. If so, it decrements the value and allows the thread to continue. If the value is 0, the thread is put to sleep. Checking the value, changing it, and possibly going to sleep is all done as a single, indivisible, **atomic action.** It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. The `release()` operation increments the value of the semaphore. If one or more processes were sleeping on that semaphore, unable to complete an earlier `acquire()` operation, one of them is chosen by the system (e.g., at random), and is allowed to complete its `acquire()`. It is important to note the meaning of the value, n, of a semaphore: n positive means that n threads can decrement its value without going to sleep; n negative means that there are n threads sleeping on the semaphore; n zero means that there are no threads waiting but the next thread to invoke `acquire` will sleep. Note that `acquire()` throws an `InterruptedException` and, hence, must be used in a `try…catch` block.

If a thread invokes `acquire` on a semaphore it may or may not sleep and it has no way of determining this fact beforehand. It is possible to use the method `availablePermits()` to check the number of available permits but this value may change before the `acquire` gets invoked. Remember this is a concurrent universe! In a situation where a thread does not want to sleep if no permit is available it can use `tryAcquire()`. This method returns false if no permits are available. In the case where one is available it returns true reducing the number of available permits in the process. There are also methods to `acquire` and `release` multiple permits at a time and also timing constraints. Consult Java Docs for a discussion of these.

Note: The methods `acquire()` and `release()` correspond to Dijkstra's `P` and `V` operations, respectively.

Semaphores can be used to control access to a number of resources. Consider the problem of modeling access to a merry-go-round with N horses at a carnival. A semaphore with an initial count of N would be used to control access to the horses. People waiting to ride on the merry-go-round would be forced to wait on the semaphore object. Each time a person entered the count on the semaphore is decremented, i.e. each person entering calls acquire(). That is, as long as there were seats available waits on the semaphore would immediately be satisfied and people could go find a horse to ride on. When N people enter the semaphore count would be zero and new comers would have to wait. Of course, the semaphore only controls access. Allocating horses when inside is a separate problem - two people could try to get on the same horse - and would require synchronization.

A semaphore with an initial value of 1 can be used to synchronize access to a shared resource. The value of the permit never exceeds 1 but may be negative. This type of semaphore is often called a binary semaphore. The problem of accessing a shared terminal can be solved by using a semaphore with an initial value of 1. Each thread invokes acquire on the shared semaphore before printing and release after printing is complete. Only a single thread holds the semaphore at any one time and all other subsequent threads invoking acquire sleep until one is woken up by a release. In a similar way we can also solve a race condition on a shared variable. The Counter class could also be written using a semaphore to enforce atomicity. The solution, given below, uses a semaphore that has an initial value of 1. Both methods, inc and dec, invoke sem.acquire() before changing the value of x and release it after the change takes place.

```
class Counter{
   private volatile int x = 1;
   private Semaphore sem = new Semaphore(1);
   void inc(){
      try{sem.acquire();}
      catch(InterruptedException e){}
      x = x + 1;
      sem.release();
   }
   void dec(){
      try{sem.acquire();}
      catch(InterruptedException e){}
      x = x - 1;
      sem.release();
   }
   int get(){return x;}
}
```

Semaphores can be used in lots of different contexts to enforce synchronization behavior and the following examples illustrate some of these.

Example 1

Suppose you are given two threads TA and TB that are ready to run, i.e. start has been invoked, and you need TA to always execute first. A possible solution is to use a single semaphore to synchronize them. The main thread creates a semaphore, sem, with an initial value of 0. This semaphore is shared by both threads TA and TB. When thread TB executes it immediately invokes acquire and, hence, waits on the semaphore until TA releases it. This ensures that TA always executes first and, when it is ready, it signals TB so that it can then run concurrently with it.

```
import java.util.concurrent.*;
class SemaphoreEx1{
    public static void main(String args[]){
        //Use Semaphore to ensure that TA executes before TB
        Semaphore sem = new Semaphore(0);
        new TA(sem).start();
        new TB(sem).start();
    }
}
class TA extends Thread{
    Semaphore sem;
    public TA(Semaphore s){sem = s;}
    public void run(){
        //put code for TA here
        System.out.println("TA going first!");
        sem.release();
    }
}
class TB extends Thread{
    Semaphore sem;
    public TB(Semaphore s){sem = s;}
    public void run(){
        try{
        sem.acquire();
        }
        catch(InterruptedException e){}
        //put code for TB here
        System.out.println("TB going second!");
    }
}
```

Example 2: Rendezvous Problem

Suppose you have two threads TA and TB and you want them to rendezvous at a given point, i.e. the threads must wait for each other to reach a given state in their respective computations. An example,

```
        TA{              TB{
          st1;           st3;
          //wait for TB       //wait for TA
          st2;           st4
        }            }
```

We don't care who goes first but st2 is not executed by TA until TB executes st3 and vice versa.

One solution is to use two semaphores to enforce the rendezvous. The semaphores are called aArrived and bArrived. Thread TA signals TB by invoking release on aArrived and then waits for TB by invoking acquire on bArrived. TB does the same in reverse order. It invokes release on bArrived and acquire on aArrived. In this way they each wait for each other to reach their respective rendezvous point.

```
import java.util.concurrent.*;
class Rendezvous{
    public static void main(String args[]){
        //Use Semaphore to ensure that TA executes before TB
        Semaphore aArrived = new Semaphore(0);
        Semaphore bArrived = new Semaphore(0);
        new TA(aArrived,bArrived).start();
        new TB(aArrived,bArrived).start();
    }
}
class TA extends Thread{
    Semaphore aArrived, bArrived;
    public TA(Semaphore s1, Semaphore s2){
        aArrived = s1; bArrived = s2;
    }
    public void run(){
        System.out.println("Stat1");
        aArrived.release();
        try{
            bArrived.acquire();
        }
        catch(InterruptedException e){}
        System.out.println("Stat2");
    }
}
class TB extends Thread{
    Semaphore aArrived, bArrived;
    public TB(Semaphore s1, Semaphore s2){
        aArrived = s1; bArrived = s2;
    }
    public void run(){
        System.out.println("Stat3");
        bArrived.release();
        try{
            aArrived.acquire();
        }
        catch(InterruptedException e){}
        System.out.println("Stat4");
    }
}
```

The task of forcing more than two threads to rendezvous at a given point would require a separate control called a Barrier and we discuss this class below. I should point out that it is possible to write a Barrier class using semaphores and we will defer such an option to the exercises listed below.

Example 3

A program is required that controls access to a tunnel by cars. At most n cars are allowed in the tunnel at any one time.

The tunnel can be implemented as a shared resource that is protected by a semaphore. The semaphore has an initial count of n. Cars enter and leave the tunnel by invoking public methods `enter()` and `leave()`. The attribute `count` records the number of cars currently in the tunnel and is updated by both `enter` and `leave`. The `lock` variable is used to ensure mutual exclusion on modifying its value. By declaring it `volatile` we ensure that `get()` returns the most recent value.

```
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.concurrent.locks.*;
class TunnelTest{
   public static void main(String args[]){
      Tunnel jackLynch = new Tunnel(10);
      for(int j = 0; j < 21; j++){
         new Car(jackLynch).start();
      }
      while(true){
         System.out.print(jackLynch.get()+" ");
         try{Thread.sleep(1000);}
         catch(InterruptedException e){}
      }
   }
}

class Tunnel{
   private Semaphore control;
   private volatile int count = 0;
```

```
    private Lock lock = new ReentrantLock();
    public Tunnel(int n){
        control = new Semaphore(n);
    }
    public void enter(){
        try{
            control.acquire();
        }
        catch(InterruptedException e){}
        lock.lock();
        try{count++;}
        finally{lock.unlock();}
    }
    public void leave(){
        lock.lock();
        try{count--;}
        finally{lock.unlock();}
        control.release();
    }
    public int get(){
        return count;
    }
}
class Car extends Thread{
    Tunnel tunnel;
    public Car(Tunnel t){tunnel = t;}
    public void run(){
        tunnel.enter();
        try{ //stay in tunnel for random period
            int t = (int)(Math.random()*5000);
            Thread.sleep(t+2000);
        }catch(InterruptedException e){}
        tunnel.leave();
```

```
   }
}
```

Notes

1. The semaphore only controls access to the tunnel. Updating the current number of cars in the tunnel could also have been solved by using an instance of `AtomicInteger`. This class is thread safe and has methods: `decrementAndGet()` and `incrementAndGet()` with the obvious semantics.
2. To avoid deadlock do not ever use `acquire()` within a synchronized method. The following class will deadlock by invoking method t(). Why?

```
class A{
  Semaphore sem;
  public A(){
     sem = new Semaphore(0);
  }
  synchronized void t(){
     ...
     try{
         sem.acquire();
     } catch(InterruptedException e){}
  }
  synchronized void t1(){
     ...
     sem.release();
  }
}
```

Example 4: Producer Consumer with Semaphores

In this example we re-code the bounded buffer producer-consumer problem discussed above. The trick is to use two semaphores: `empty` that controls the consumer and `full` that controls the producer. The producer calls `acquire()` on `full` and `release()` on `empty`. The consumer calls `acquire()` on `empty` and `release()` on `full`. At the start the buffer is empty therefore empty has an initial value of 0 and full has an initial value of max. Updates to the actual buffer are protected by a lock.

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import java.util.*;
class SemProducerConsumer{
    public static void main(String args[]){
        Buffer <Integer> buffer = new Buffer<Integer>(10);
        new Thread(new Producer(buffer)).start();
        new Thread(new Consumer(buffer)).start();
    }
}
class Buffer<E>{
   private int max;
   private int size = 0;
   private ArrayList<E> buffer;
   private Semaphore empty;  // control consumer
   private Semaphore full;   // control producer
   private Lock lock = new ReentrantLock();
   public Buffer(int s){
     buffer = new ArrayList<E>();
     max = s;
     empty = new Semaphore(0);
     full = new Semaphore(max);
   }

   public void put(E x){
     try{
        full.acquire();
     }
     catch(InterruptedException e){}
     // synchronize update of buffer
     lock.lock();
     try{
        buffer.add(x);
        size++;
        empty.release();
     }finally{
        lock.unlock();
     }
   }
   public E get(){
        try{
            empty.acquire();
        }
        catch(InterruptedException e){}
        // synchronize update of buffer
        lock.lock();
        try{
        E temp = buffer.get(0);
            buffer.remove(0);
            size--;
            full.release();
            return temp;
        }finally{
        lock.unlock();
        }
   }
}
class Producer implements Runnable{
    Buffer <Integer> buffer;
    public Producer(Buffer <Integer> k){
        buffer = k;
    }
    public void run(){
        for(int j = 0; j < 50; j++){
            Integer x = new Integer(j);
            buffer.put(x);
            //set production cycle
            try{
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
        }
        buffer.put(null); //sentinel value
    }
}

class Consumer implements Runnable{
    Buffer <Integer> buffer;
    public Consumer(Buffer <Integer> k){
        buffer = k;
    }
    public void run(){
        System.out.print("Buffer data: ");
        Integer x = buffer.get();
        while(x != null){
            System.out.print(x+" ");
            //set consumption cycle
            try{
                Thread.sleep(500);
            }
            catch(InterruptedException e){}
            x = buffer.get();
        }
    }
}
```

Example 5

Given three threads TA, TB, TC where TA repeatedly prints the letter *a*, TB the letter *b* and TC the letter *c*. The output from the 3 threads must be synchronized so that it satisfies the following three conditions:

1) A *b* must be output before any *c*'s are output;
2) *b*'s and *c*'s must alternate in the output string, that is, the first *b* output must be followed by a *c* and so on;
3) The total number of *b*'s and *c*'s that have been output at any given point in the output stream may not exceed the number of *a*'s that have been output up to that point.

We begin a solution to this problem by ignoring TA and using semaphores force threads TB and TC to alternate their output TB going first. This can be achieved by using two semaphores. Thread TB waits on TC by invoking `acquire` on `semC` and signals TC by invoking `release` on `semB`. Thread TC does the opposite. This guarantees that they alternate output. The initial values of the shared semaphores then determine the starting order. By ensuring that `semB` has an initial value of 0 and `semC` an initial value of 1 the output order will be *b* followed by *c*. This solves parts 1 and 2 above. The code for the separate threads is given below.

```
class TB extends Thread{
    Semaphore semB, semC;
    public TB(Semaphore sb, Semaphore sc){
        semB = sb; semC = sc;
    }
    public void run(){
        while(true){
            try{
                Thread.sleep((int)(Math.random()*2000));
            }
            catch(InterruptedException e){}
            // must wait for TC to print a C
            try{
             semC.acquire();
            }catch(InterruptedException e){}
            System.out.print('b');
            semB.release();
        }
    }
}

class TC extends Thread{
    Semaphore semB, semC;
    public TC(Semaphore sb, Semaphore sc){
        semB = sb; semC = sc;
    }
    public void run(){
        while(true){
```

```
        try{
           Thread.sleep((int)(Math.random()*1000));
        }
        catch(InterruptedException e){}
        // must wait for TB to print a B
        try{
          semB.acquire();
        }catch(InterruptedException e){}
        System.out.print('c');
        semC.release();
      }
   }
}
```

To solve part 3 we introduce a counting semaphore that is shared by all three threads. Every time TA prints an *a*, it invokes release on semA. Both threads TB and TC invoke acquire on semA before outputting their respective values. This ensures that the number of *b*'s and *c*'s printed so far in the output stream never exceeds the number of *a*'s.

The complete solution, including a main thread that initializes the semaphores and creates the threads, is listed below. The different sleeps in each thread are used to control the speed of output. Without them the speed of output would be such that it would be unreadable. Note that both constructors for threads TB and TC have been modified to take three Semaphores as parameters.

```
import java.util.concurrent.*;
public class SemaphoreABCTest {
  public static void main(String args[]){
      // TB must wait on semaphore C and TC must wait on semaphore B
      // TB must go first.Therfore set semB to 0 and SemC to 1
      Semaphore semB = new Semaphore(0);
      Semaphore semC = new Semaphore(1);
      // At start no A's printed therefore both B and C must wait for an A
      // therefore A's initial count is 0
      Semaphore semA = new Semaphore(0);
      new TA(semA).start();
```

```
        new TB(semA,semB,semC).start();
        new TC(semA, semB, semC).start();
    }
}
class TA extends Thread{
    Semaphore semA;
    public TA(Semaphore s){semA = s;}
    public void run(){
        while(true){
            try{
                Thread.sleep((int)(Math.random()*1000));
            }catch(InterruptedException e){}
            System.out.print('a');
            // TA just releases the semaphore to indicate another A is printed
            semA.release();
        }
    }
}
class TB extends Thread{

    Semaphore semA, semB, semC;
    public TB(Semaphore sa, Semaphore sb, Semaphore sc){
        semA = sa; semB = sb; semC = sc;
    }
    public void run(){
        while(true){
            try{
                Thread.sleep((int)(Math.random()*2000));
            }catch(InterruptedException e){}
            // must wait for TC to print a C
            try{
             semC.acquire();
             //number of B's and C's cannot exceed number of A's
             semA.acquire();
            }catch(InterruptedException e){}
            System.out.print('b');
            semB.release();
        }
    }
}
class TC extends Thread{
    Semaphore semA, semB, semC;
    public TC(Semaphore sa, Semaphore sb, Semaphore sc){
        semA = sa; semB = sb; semC = sc;
    }
    public void run(){
        while(true){
            try{
                Thread.sleep((int)(Math.random()*1000));
            }catch(InterruptedException e){}
            // must wait for TB to print a B
            try{
              semB.acquire();
              //number of B's and C's cannot exceed number of A's
              semA.acquire();
            }catch(InterruptedException e){}
            System.out.print('c');
            semC.release();
        }
    }
}
```

Example 6

In this problem we use semaphores to create a first-in-first-out queue of waiting threads. The idea is that each thread joins the queue by passing a semaphore as argument to the join method in the ThreadQueue. The initial value of this semaphore, created by the thread, must be 0. Each thread waits on its own semaphore. A LinkedList class of type Semaphore is used to manage the actual queue. A lock is used to enforce synchronization on the queue. Invoking acquire on sem forces the thread to sleep. The method next() removes the semaphore at the head of a non-empty queue and releases it, thereby releasing the waiting thread.

```
class ThreadQueue{
    Queue<Semaphore> queue = new LinkedList<Semaphore>();
    Lock lock = new ReentrantLock();
    public void join(Semaphore sem){
        lock.lock();
        try{
          queue.add(sem);
        }finally{lock.unlock();}
        try{
            sem.acquire();
        }catch(InterruptedException e){}
    }
    public void next(){
        lock.lock();
        try{
            if(queue.size() > 0){
               Semaphore ss = queue.poll();
               ss.release();
            }
        }finally{lock.unlock();}
    }
    public int size(){return queue.size();}
}
```

To join the queue a thread creates a binary Semaphore with an initial value of 0 and waits on it. A sample thread is given below.

```
 class QThread extends Thread{
    Semaphore sem = new Semaphore(0);
    ThreadQueue tqueue;
    int number;
    public QThread(ThreadQueue tq, int n){ tqueue = tq;number = n;}
    public void run(){
        //join queue and wait turn
        tqueue.join(sem);
        System.out.println("Turn = "+number);
```

```
   }
}
```

A control program, listed below, creates 10 threads that join a shared queue and then releases the threads in order of arrival in 1 second intervals.

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
public class ThreadQueueTest {
   public static void main(String[] args) {
     ThreadQueue tq = new ThreadQueue();
     for(int j = 0; j < 10; j++){
        new QThread(tq,j).start();
     }
     //release threads one per second
     for(int j = 0; j < 10; j++){
        try{
           Thread.sleep(1000);
        }catch(InterruptedException e){}
        tq.next();
     }
   }
}
```

Example 7

Write a resource allocator that manages a fixed number of a given resource. Clients may request multiple instances of the resource waiting for them to become available. All resources are granted at the same time. Client requests should be satisfied first come first served.

Clients make requests for a number of items and are prepared to wait for them to become available. Each client submits a request to the resource allocator and then waits for the total number of items to become available. To wait it sleeps on its own semaphore. The class `Request`, listed below, encapsulates both the number of items required and a reference to the semaphore created by the client making the request. It has two public methods: `getRequest` that returns the number of items requested by the thread; `allocate` that invokes `release` on the semaphore used by the thread to wait.

```
class Request{
    private int numItems;
    private Semaphore sem;
    public Request(Semaphore ss, int n){
        numItems = n; sem = ss;
    }
    public int getRequest(){ return numItems;}
    public void allocate(){sem.release();}
}
```

The ResourceAllocator class uses an ArrayList of type Request to manage a first-in-first-out queue of client requests. The lock attribute is used to enforce synchronization on queue updates. The attribute available keeps a record of the number of resources currently available and is always less than or equal to maxResources. The number of resources currently allocated is equal to maxResources less available. The public method getResources takes a Request instance as argument, checks that it is satisfiable and then proceeds to check if there are existing requests. If there are clients waiting the new request is appended to the queue; otherwise it tries to satisfy the request immediately updating the value of available in the process. If this is not possible the request is added to the tail of the queue. The method returnResources acquires the lock, updates the number of available resources, checks for client requests waiting to be satisfied and then proceeds to distribute resources on a first come first served basis. Because both methods share the same lock it is not possible for a new request to be satisfied while the current queue is being dealt with.

```
class ResourceAllocator{
    ArrayList<Request> queue = new ArrayList<Request>();
    Lock lock = new ReentrantLock();
    int maxResources;
    int available;
    public ResourceAllocator(int max){
        maxResources = max;
        available  = max;
    }
    public int getMax(){return maxResources;}
public void getResources(Request req) throws Exception{
    lock.lock();
    try{
        if(req.getRequest() > maxResources){
            throw new Exception("Not enough resources to satisfy request");
        }
        else{
            if(queue.size() > 0) //don't allow new requests if clients waiting
                queue.add(req);
            else{
              if(req.getRequest()<= available){
               available = available - req.getRequest();
               req.allocate();
             }
             else queue.add(req);
           }
     }
    }finally{lock.unlock();}
}
public void returnResources(int n){ // assume n positive
    lock.lock();
    try{
        available = available + n;
        if(queue.size() > 0){
          //try to satisfy all pending requests
          Request req = queue.get(0);
          while(queue.size() > 0 && req.getRequest() <= available){
              available = available - req.getRequest();
              req.allocate();
              queue.remove(0);
              if(queue.size()>0)
                  req = queue.get(0);
          }
        }
    }finally{lock.unlock();}
}
```

```
    public int size(){return queue.size();}
}
```

To test the resource allocator class we write a thread that requests a random number of resources in the range 1..maxResources. It creates an instance of the Request class, submits the request and waits for the resources to become available. It then sleeps and returns the resources. The code is:

```
class QThread extends Thread{
    Semaphore sem = new Semaphore(0);
    ResourceAllocator resources;
    int number;
    public QThread(ResourceAllocator rs, int n){ resources = rs; number = n;}
    public void run(){
        int items = (int)(Math.random()*resources.getMax()) + 1;
        Request req = new Request(sem,items);
        try{
          resources.getResources(req);
          try{
              sem.acquire();
          }catch(InterruptedException e){}
        }catch(Exception e1){System.out.println(e1.getMessage());}
        System.out.printf("Thread %3d got %3d resources\n",number, items);
        try{
            Thread.sleep(2000);
        }catch(InterruptedException e){}
        resources.returnResources(items);
        System.out.printf("Thread %3d returned %3d resources\n",number,
                              items);
    }
}
```

Finally a simple test program that creats 10 threads that all access an instance of the resource allocator.

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
```

```
public class ResourceAllocatorTest {
   public static void main(String[] args) {
     ResourceAllocator rAllocator = new ResourceAllocator(20);
     for(int j = 0; j < 10; j++){
        new QThread(rAllocator,j).start();
     }
   }
}
```

Exercise

Question 1

Use a binary semaphore to allow multiple writers to share the screen.

Question 2

Write two periodic threads TA and TB that alternate. TA goes first.

Question 3

Given N threads ensure that they execute in order 0..N-1. Each thread should print a message listing its number in the sequence when it gets to execute. The values must be in order.

Question 4

A server manages the allocation of 10 ports to clients. When a client requests a port from the server it receives a port number, if one is available. In the event that no port is available the client waits indefinitely for one to become free. Once a port is allocated no other client can use it. Using a semaphore and locks implement the server class and write client threads to that interact with it by requesting port numbers.

Question 5

Re-write the dining philosophers using Semaphores to manage access to the shared resources. The problem of deadlocks can be solved by making one of the philosophers left handed. Use this approach to avoid deadlocking.

Question 6

Write the `ThreadQueue` class so that threads are released based on their priority value. Higher priority threads released first.

Question 7

A Barrier is a control that forces *N* threads to rendezvous at a given point. When all threads reach the barrier then they are all released. The threads are automatically released by the barrier when the N^{th} thread invokes method `await` on the barrier. Using Semaphores write a class that implements a barrier.

Question 8

Bounded Overtaking Resource Manager

In the resource allocator example discussed above client requests are satisfied in first-come-first-served order. In some situations this is not the fairest way to allocate resources because clients with large requests can hold up unnecessarily clients with small requests. One solution is to allow clients with small requests to proceed within certain limits. This is called the bounded overtaking solution because it puts a limit on the number of times clients with large requests may be passed over. Your task is to write a resource manager that sets a limit on the number of times a client may be passed over before receiving their requested resources. Each client has an overtaken count and when this reaches its limit this client must be served next.

CountDownLatch

A latch is a synchronizer that delays the progress of threads until it reaches what is termed its terminal state. Once the terminal state has been reached the latch remains in that state and it cannot change or reset. It acts as a gate stopping all threads until its terminal state is reached. In the terminal state the gate is open and all waiting threads pass through. A latch could be used to ensure a service does not start until other services on which it depends have started. The CountDownLatch class is a flexible latch implementation that allows one or more threads to wait for a set of events to occur. The CountDownLatch is initialized with a positive number, which represents the number of events to wait for. Once an event has occurred the count is decremented. When all the events have occurred the count will be zero (terminal state) and the waiting threads can proceed to execute concurrently.

Two examples are given. In the first case 10 threads wait on a starting latch that is controlled by main(). The latch has an initial value of 1 and is set to the terminal state, i.e. 0 when main() invokes start.countDown().

```
import java.util.concurrent.*;
class LatchesTest1{
   public static void main(String args[]){
      // let n threads wait to get the signal to go
      CountDownLatch start = new CountDownLatch(1);
      for(int j = 0; j < 10; j++){
         new TA(start,j).start();
      }
      System.out.println("Threads under starters orders");
      try{
         Thread.sleep(3000);
      }
      catch(InterruptedException e){}
      System.out.println("Starting threads now!");
      start.countDown();
   }
}
class TA extends Thread{
   CountDownLatch start;
   int num;
```

```
    public TA (CountDownLatch s, int n){
        start = s; num = n;
    }
    public void run(){
        //wait for the signal to go
        try{
            start.await();
        }
        catch(InterruptedException e){}
        System.out.println("Thread "+num+ " going");
    }
}
```

In the second example main uses a latch to wait for the threads to terminate. Each thread invokes `end.countDown()` before terminating.

```
import java.util.concurrent.*;
class LatchesTest2{
    public static void main(String args[]){
        // let n threads wait to get the signal to go
        // main waits for all the threads to finish before proceeding
        CountDownLatch start = new CountDownLatch(1);
        CountDownLatch end = new CountDownLatch(10);
        for(int j = 0; j < 10; j++){
            new TA(start,end,j).start();
        }
        System.out.println("Threads under starters orders");
        try{
            Thread.sleep(3000);
        }
        catch(InterruptedException e){}
        System.out.println("Starting threads now!");
        start.countDown();
        try{
            end.await();
        }
        catch(InterruptedException e){}
        System.out.println("The race is over");
    }
}
class TA extends Thread{
    CountDownLatch start, end;
    int num;
    public TA (CountDownLatch s,CountDownLatch s1, int n){
        start = s; end = s1; num = n;
    }
    public void run(){
        //wait for the signal to go
        try{
            start.await();
        }
        catch(InterruptedException e){}
        System.out.println("Thread "+num+ " going");
        // do something!
        end.countDown();
    }
}
```

CyclicBarrier

Barriers are similar to latches in that they block a group of threads until some event has occurred. However all the threads must come together, at a barrier point, at the same time, in order to proceed. `CyclicBarriers` allow a fixed number of threads to rendezvous repeatedly at a barrier point. A barrier can be reset and used again if required. `CyclicBarriers` are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called *cyclic* because it can be re-used after the waiting threads are released. The constructor, `CyclicBarrier(n),` creates a barrier for n processes. The method `await` is invoked by each thread when it reaches its rendezvous point. The class throws both: `InterruptedException and BrokenBarrierException.` The example, listed below, uses three threads that co-operate in the task of initializing a shared integer array and then process the whole array in different ways. The threads all rendezvous after initializing their own individual segments denoted by lower bound, lb, and upper bound, ub. The rendezvous point is implemented by a shared barrier instance. Each thread then proceeds to process the whole array. None of the threads modify the state of the array after initialization.

```
import java.util.concurrent.*;
class CyclicBarrierTest{
    public static void main(String args[]){
        int data[] = new int[100000];
        CyclicBarrier barrier = new CyclicBarrier(3);
         // barrier used to force threads to rendezvous when
         // initialization complete
        CountDownLatch latch = new CountDownLatch(3);
         // latch used by main to wait for the threads to complete
         //distribute initialization work over three threads
        int ind[] = new int[4];
        for(int j = 0; j < 4;j++) ind[j] = (j*data.length)/3;
        new SumOdd(data,ind[0],ind[1],barrier,latch).start();
        new SumEven(data,ind[1],ind[2],barrier,latch).start();
        new FreqOdd(data,ind[2],ind[3],barrier,latch).start();
        try{latch.await();}
        catch(InterruptedException e){}
    }
}

class SumOdd extends Thread{
    CyclicBarrier barrier;
    CountDownLatch latch;
    int data[];
    int lb, ub;
    public SumOdd(int d[], int a,int b, CyclicBarrier cb,CountDownLatch lh){
        barrier = cb; latch = lh;
        data = d; lb = a; ub = b;
    }
    public void run(){
        //init segment block
        for(int j = lb; j < ub;j++) data[j] = (int)(Math.random()*1000);
        //wait for others to rendezvous
        try{barrier.await();}
        catch(InterruptedException e){}
        catch(BrokenBarrierException be){}
        long sum = 0;
        for(int j = 0; j < data.length;j++)
            if(data[j] % 2 == 1) sum = sum + data[j];
        System.out.println("Sum odd = "+sum);
        latch.countDown();
    }
}
class SumEven extends Thread{
    CyclicBarrier barrier;
    CountDownLatch latch;
    int data[];
    int lb, ub;
    public SumEven(int d[], int a,int b, CyclicBarrier cb,CountDownLatch lh){
        barrier = cb; latch = lh;
        data = d; lb = a; ub = b;
    }
    public void run(){
        //init segment block
        for(int j = lb; j < ub;j++) data[j] = (int)(Math.random()*1000);
        //wait for others to rendezvous
        try{barrier.await();}
        catch(InterruptedException e){}
        catch(BrokenBarrierException be){}
        long sum = 0;
        for(int j = 0; j < data.length;j++)
            if(data[j] % 2 == 0) sum = sum + data[j];
        System.out.println("Sum even = "+sum);
        latch.countDown();
    }
}
class FreqOdd extends Thread{
    CyclicBarrier barrier;
    CountDownLatch latch;
    int data[];
    int lb, ub;
    public FreqOdd(int d[], int a,int b, CyclicBarrier cb,CountDownLatch lh){
        barrier = cb; latch = lh;
        data = d; lb = a; ub = b;
    }
    public void run(){
      //init segment block
      for(int j = lb; j < ub;j++) data[j] = (int)(Math.random()*1000);
      //wait for others to rendezvous
      try{barrier.await();}
      catch(InterruptedException e){}
      catch(BrokenBarrierException be){}
      int freq = 0;
      for(int j = 0; j < data.length;j++)
        if(data[j] % 2 == 1) freq = freq + 1;
      System.out.println("Frequency odd = "+freq);
      latch.countDown();
    }
}
```

A CyclicBarrier supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This *barrier action* is useful for updating shared-state before any of the parties continue. The constructor is:

CyclicBarrier(int n, Runnable barrierAction).

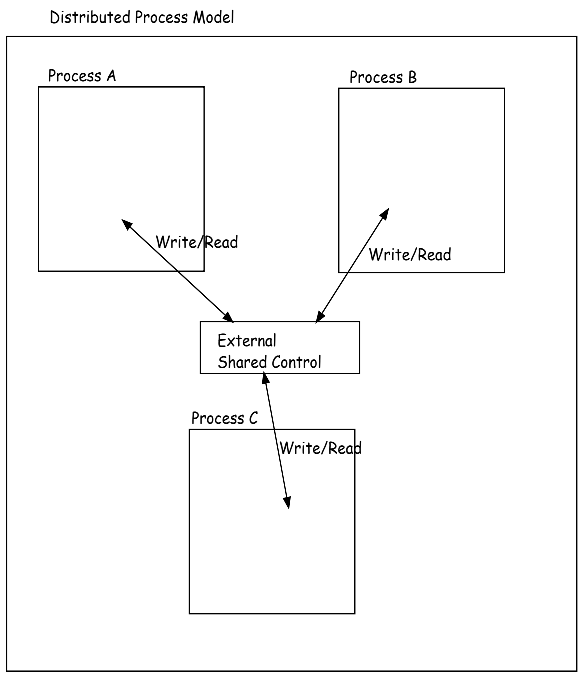
The program, listed below, uses 5 threads to initialize a shared integer array. All the threads wait at a barrier after completing the initialization phase. But before releasing them a thread under the control of the barrier verifies that all values in the array are non-zero. It modifies the value of valid in the single instance of the VerifiedRes class shared by all threads. Each thread checks this value when it proceeds after the barrier and prints a simple message.

```
import java.util.concurrent.*;
```

```
class CyclicBarrierTest1{
    public static void main(String args[]){
        int data[] = new int[100000];
        VerifiedRes ver = new VerifiedRes();
        CyclicBarrier barrier = new CyclicBarrier(5,new Verifier(data,ver));
        // barrier used to force threads to rendezvous when
        //initialization complete
        int seg = data.length/5;
        for(int j = 0; j < 5;j++)
          new Initializer(data,0,seg*(j+1),barrier,ver).start();
    }
}
class VerifiedRes{
    private boolean valid = true;
    synchronized void set(){
        valid = false;
    }
    synchronized boolean valid(){return valid;}
}
class Initializer extends Thread{
    CyclicBarrier barrier;
    int data[];int lb, ub;
    VerifiedRes ver;
    public Initializer(int d[], int a,int b, CyclicBarrier cb, VerifiedRes vr){
         data = d; lb = a; ub = b;
         barrier = cb; ver = vr;
    }
    public void run(){
        //init segment block
        for(int j = lb; j < ub;j++) data[j] = (int)(Math.random()*1000);
        //wait for others to rendezvous
        try{barrier.await();}
        catch(InterruptedException e){}
        catch(BrokenBarrierException be){}
        if(ver.valid()) System.out.println("Verified");
        else System.out.println("Failed Verification");
    }
}
class Verifier implements Runnable{
    int data[];
    VerifiedRes ver;
    Verifier(int d[], VerifiedRes vr){data = d; ver = vr;}
    public void run(){
        boolean found = false;
        for(int j = 0; j < data.length; j++)
            if(data[j] == 0) found = true;
        if(found) ver.set();
    }
}
```

Interprocess Communication

The model of concurrency we have been studying is called the shared memory model because all the threads share common memory in the context of a single process. This makes thread communication simple because threads can share access to data by sharing reference variables. However, it is also necessary to address the problem of inter-process communication where multiple independent programs communicate with each other. By this we mean communication between multiple concurrent programs working together to solve problems. To work together these programs need to share data and communicate with each other. An example of this inter-process sharing is the Clipboard application in Windows that is used to copy data from one application to another. The Clipboard is, in fact, a kernel object managed by the operating system to allow applications copy and paste data. A user application can write data to the clipboard and another application can read from it. Operating systems, in general, provide application programmer interfaces (API) that support inter-process communication. Along with the Clipboard Windows supports shared memory, named pipes and named semaphores. The Unix and Linux operating systems also have pipes, shared memory, message passing and semaphores. These controls can be used by programmers to write systems where multiple individual programs can execute concurrently, while co-operating, to provide services. However, these systems are operating system specific and, as a consequence, different versions are required for the different systems.



The diagram above illustrates an example of the **distributed memory model**. It shows three processes that use an external shared control to communicate. Each process is an independent program that has its own memory and its own threads. Of course, every process has its own main method. Processes may not write to each others address spaces. Hence, to communicate they need to access external controls such as named pipes, shared memory or named semaphores. These controls belong to the underlying architecture and are accessed in user programs through system calls. Using this approach to support communication and co-operation between processes makes your applications operating system specific. The different operating systems use different system calls and as a result it is not possible to make a program that is platform independent.

In platform independent environments there is no direct support for inter-process communication. This simply means that neither Java nor C# provide support at the language level for it. However, it is possible to program it using remote method invocation (RMI) or through the socket network layer. In this chapter we discuss in detail inter-process communication using sockets. Anyone interested in RMI might consult Mastering RMI by Rikard Oberg, Wiley 2001. We should point out that one of the advantages of using sockets is that it allows inter-process communication not just between local processes but also communication over multiple networked machines and, hence, provides an infrastructure for truly parallel distributed processing.

Networking

Java provides classes that support communication for networked resources. The basic classes are provided by the `java.net` package. There are two main categories of classes: the sockets API and tools for working with URLs. In this section we look at sockets and how to use them to allow processes communicate locally.

Socket

The sockets API provides access to the standard network protocols used for communications between any client and server on the Net. They are low level tools that send streams of data between applications that may or may not be on the same host. The programmer must provide their own application level protocol for handling and interpreting the data. The basic socket class provides a connection-oriented protocol that is similar to a telephone conversation. After establishing a connection two applications can send data back and forth. The protocol ensures that no data is lost and that data arrives in the order in which it is sent. The underlying protocol class is TCP IP.

The Server

The server opens a socket and listens for incoming conversations. It does this by creating a ServerSocket object and waits, blocked in a call to its accept until a connection arrives. When a client connects the accept() method creates a Socket object that the server then uses to communicate with the actual client. The code to create the ServerSocket is:

```
    ServerSocket servesock = new ServerSocket(portNumber);
```

(The port number should be greater than 1024. Numbers less than or equal to this value are used for system processes.)

A server may carry on conversations with multiple clients at the same time. Each client has its own Socket object – but there is only one ServerSocket. Typically, the server is in a loop listening for new clients. When a client communicates, a socket is created and a new thread is launched to deal with the communication. The server then goes back to listening. The following code fragment provides a template of how this is done. The port number is set to 1234, as an example, and a server socket is created. The server then loops forever listening for clients. When a client connects it creates a thread to handle the communication and goes back to listen for more clients.

```
import java.io.*;
import java.net.*;
class …Server {

    final static int port = 1234; // any number > 1024

    public static void main(String[] args) {
        System.out.println("Server running ...");

        try {
           ServerSocket servesock = new ServerSocket(port);
            // for service requests on the given port number
            while (true) {
                // wait for a service request on the port
                Socket socket = servesock.accept();
                // start thread to service request
                new ThreadName(socket).start();
            }
        } catch (IOException e) {}
    }
}
```

The Client

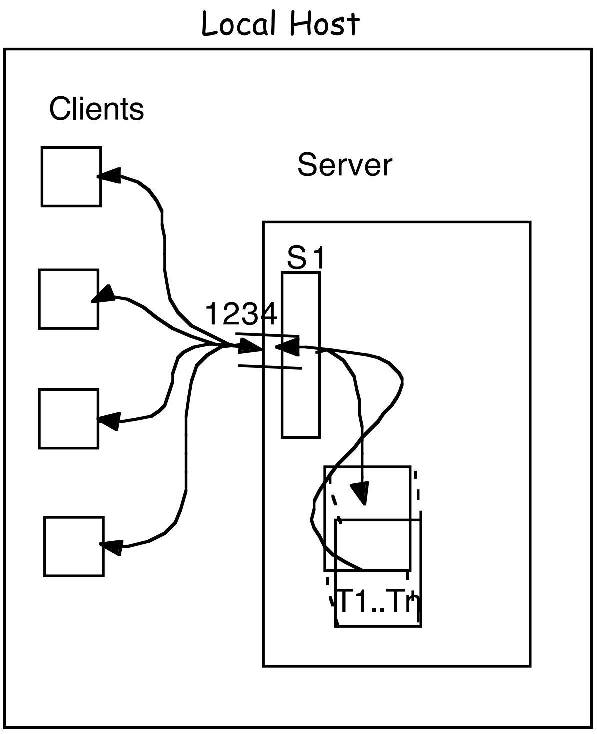
The client needs to know two pieces of information to connect to a server on the Net: the *hostname* and a *port* number. The hostname can be an IP address or an actual host name and the port number is any valid port number for the host machine. The following code sequence gives an example of how to do this where the host is the local machine.

```
    Socket socket;
    socket = new Socket(InetAddress.getLocalHost(), port);
```

Once a connection is established, input and output streams can be retrieved with the Socket getInputStream() and getOutputStream() methods. The following program fragment shows how to do this. The client creates a Socket instance and then retrieves both input and output streams. These are then used to send and receive data from the server providing the service.

```
import java.io.*;
import java.net.*;
class ..Client {

  private final static int port = 1234;

  public static void main(String[] args) {
       try {
         Socket socket;
         socket = new Socket(InetAddress.getLocalHost(),port);
         … in = new … (socket.getInputStream());
         …  out = new … (socket.getOutputStream());

        // use out to send data
        // use in to get reply

          socket.close();
        } catch (IOException e) {System.out.println(e);}
    }
}
```

The input and output streams are byte streams and to simplify working with them you need to wrap them in DataInputStream and DataOutputStream classes. These two classes allow an application to read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream. The most relevant public methods for DataInputStream with obvious semantics are: readBoolean, readChar, readDouble, readInt, readFloat, readLong. There is no readString method but you can read a Unicode character string encoded in UTF-8 format using readUTF. The methods for DataOutputStream are: writeBoolean, writeChar, writeDouble, writeInt, writeFloat, writeLong and writeUTF.



The diagram opposite shows a model of a server and a number of client applications. The server and clients all reside on the same machine. The server listens on port 1234 and when clients connect each client is allocated a thread that manages the communication until it is terminated by the thread or the client. Data can be sent in both directions.

To illustrate all this we write a simple server that provides an integer square root service. Clients send positive integer values and the server returns the integer square root of each of the values sent.

The server just uses the template listed above and creates an instance of the thread SquareRoot to deal with separate client requests. The thread SquareRoot takes the socket returned by the server socket as argument and creates its own data input and data output streams from the socket streams and uses them to communicate with the client. The agreed sentinel, -1, sent by the client is used to indicate that the input list of data is complete. The local method sqrt(x) is used by the thread to calculate the integer square root of the number. It assumes all values are non-negative. When the sentinel is read the loop terminates and the thread then closes the socket and terminates the communication.

```
import java.io.*;
import java.net.*;
class SqrtServer {
    final static int portSqrt = 1234; // any number > 1024
    public static void main(String[] args) {
        System.out.println("Server running...");
       try {
           ServerSocket servesock = new ServerSocket(portSqrt);
           // for service requests on port portSqrt
           while (true) {
                  // wait for a service request on port portSqrt
                  Socket socket = servesock.accept();
                  // start thread to service request
                  new Thread(new SquareRoot(socket)).start();
           }
       } catch (IOException e) {e.printStackTrace();}
    }
}

class SquareRoot implements Runnable{
  Socket socket;
  SquareRoot(Socket s){socket = s;}
  public void run() {
   try{
    DataInputStream in = new DataInputStream(socket.getInputStream());
    DataOutputStream out = new DataOutputStream(socket.getOutputStream());
    int x = in.readInt(); // get number from client
    while(x != -1){ //let -1 be sentinel
```

```
        // calculate square root
        int k = sqrt(x);
        out.writeInt(k); // send square root
        x = in.readInt(); //get next number
   }
   socket.close(); // close connection
  }
  catch (IOException e) {}
}
 private int sqrt(int x){
    int k = 0;
    while ((k+1)*(k+1)<= x) k++;
    return k;
  }
}
```

The Client creates a socket on the local host using the same port number as the server and then creates instances of data input and data output streams on the socket streams. It sends 10 random numbers to the server and waits for the square root of each one to be returned. When the loop terminates it sends the sentinel value to tell the server that the work is complete.

```
import java.io.*;
import java.net.*;
class SqrtClient {
  private final static int portSqrt = 1234;
  public static void main(String[] args){
    try{
      Socket socket;
      socket = new Socket(InetAddress.getLocalHost(),portSqrt);
      DataInputStream in = new DataInputStream(socket.getInputStream());
      DataOutputStream out = new
                                              DataOutputStream(socket.getOutputStream());
      int x = 0;
      while (x < 10){
        int k = (int)(Math.random()*1000);
        out.writeInt(k);
        int n = in.readInt(); // wait for result from server
        System.out.println("Integer square root of "+ k + " = " + n);
        x = x + 1;
      }
      out.writeInt(-1); //sentinel
      socket.close();
    } catch (IOException e) {System.out.println(e);}
  }
}
```

Note that the client application and the server application are separate programs. You must run the server first because if there is no server listening on the port when the client tries to connect it throws an exception and terminates. Once the server is running you can execute as many clients as often as you like. (If you are using JCreator you must execute the server in a separate application of the IDE).

Reading and Writing Objects over Sockets

The data input and data output streams read and write primitive types over sockets. To read and write actual class instances we need to put in place a protocol. There are two possible ways to do this: the class can *know* how to read and write its data over a socket by providing public methods to do so or it can implement what is called the Serializable interface. We will discuss both approaches and give examples of each. The first approach ultimately relies on data input and data output streams to transfer data. Each class *knows* its own data and agrees to provide public methods that write and read the data attributes in the correct order. To illustrate this approach we take a class Person that has attributes first name, surname and age. The class has two constructors: an all argument constructor and a default one that initializes fName and sName to null and age to 0. This second constructor is used primarily to support reading over sockets. The important methods here are writeOutputStream and readInputStream because they are the public methods that *know* how to read and write data over sockets. The method writeOutputStream takes a DataOutputStream as argument and uses its methods to send the data. The try..catch block is required to handle possible i/o exceptions.

```
public void writeOutputStream(DataOutputStream out){
   try{
     out.writeUTF(fName);
     out.writeUTF(sName);
     out.writeInt(age);
   }catch(IOException e){e.printStackTrace();}
 }
```

The method readInputStream also takes a DataInputStream as argument and uses its methods to read data, in the same order as written by write, from the socket stream.

```
public void readInputStream(DataInputStream in){
    try{
      fName = in.readUTF();
      sName = in.readUTF();
      age = in.readInt();
    }
    catch(IOException e){e.printStackTrace();}
 }
```

The complete class is listed below. The equals method is included simply because we will use it for searching an ArrayList in the sample server discussed below.

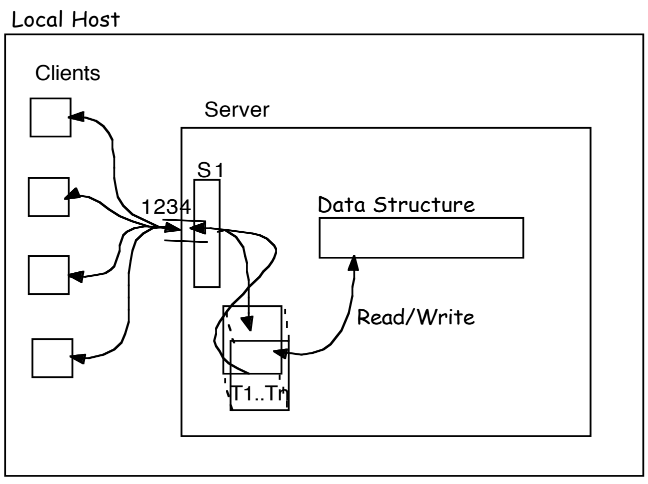
```
import java.io.*;
final public class Person {
    private String fName;
    private String sName;
    private int age;
  public Person(String fn,String sn, int a){
    fName = fn; sName = sn; age = a;
  }
  public Person(){
    fName = null; sName = null; age = 0;
  }
  public String fName(){return fName;}
  public String sName(){return sName;}
  public int age(){return age;}
  public String toString(){
    return sName+" "+fName+" "+age;
  }
```

```
 public boolean equals(Object ob){ //equality based on surname only
   if(!(ob instanceof Person)) return false;
   Person p = (Person)ob;
   return sName.equals(p.sName);
 }
 //=====================================================
 //Methods used to read and write to streams over sockets
 public void writeOutputStream(DataOutputStream out){
   try{
    out.writeUTF(fName);
    out.writeUTF(sName);
    out.writeInt(age);
   }catch(IOException e){e.printStackTrace();}
 }
 public void readInputStream(DataInputStream in){
   try{
    fName = in.readUTF();
    sName = in.readUTF();
    age = in.readInt();
   }
   catch(IOException e){e.printStackTrace();}
 }
}
```

Serialization is the process of converting a data structure or object state into a binary format that can be stored on file or sent over a network. A *serialized* object is an object represented as a sequence of bytes that includes all data as well as information about its type and the types of data stored in the object (its methods are not included). It is a binary stream of bytes that represent the state of a complete object. This object may be a single instance of a simple class or one that instantiates a collection of items. A complete data structure, such as an ArrayList, can be serialized. A serialized object may be written to a file using what is called an ObjectOutputStream. Once written to file it can be read and *deserialized*, i.e. it can be re-constructed in memory. Several object-oriented programming languages support object serialization. Python, PHP, Java, and the .NET family of languages all support it. Java provides automatic serialization as long as the class implements the java.io.Serializable interface. All classes that implement this interface may be copied to file using object streams or sent over sockets. A class simply declares that it implements the java.io.Serializable interface. There are no actual methods to implement. However, to write and read the actual objects you must use an ObjectOutputStream and an ObjectInputStream. The method writeObject(Object ob) writes primitive data types and graphs of Java objects to an output stream. The method readObject deserializes an object written by writeObject. We will illustrate this approach in an example later in this chapter.

UploadDownload Person Server

Write a server that manages a collection of person data. The server should support two types of client: one that uploads new personal data and one that retrieves personal data from the server. The server, in this instance, should be a local host that provides a storage service that allows two types of client to share access to common data.



The diagram provides a model for this service. Clients connect through port 1234 and the server supports access to shared data. Each client is allocated a thread by the server to handle its communication. Writer threads read input from the client and write it a shared data structure. This access must be synchronized because multiple clients may upload data concurrently.

The data structure class, called Data, is listed below. It uses an ArrayList to store the actual Person details. This class is thread safe and has three public methods: add that appends a new person to the list; search that takes a string representing a surname as argument and uses the contains method from the ArrayList class to return a Boolean value indicating present or not present; retrieve that takes a surname as argument and returns an ArrayList of Person for matching surnames.

```
class Data{
   private ArrayList<Person> data = new ArrayList<Person>();
   private Lock lock = new ReentrantLock();
```

```
    void add(Person p){
        lock.lock();
        try{
            data.add(p);
        }finally{lock.unlock();}
    }
    boolean search(Person p){
        lock.lock();
        try{
            return data.contains(p);
        }finally{lock.unlock();}
    }
    ArrayList<Person> retrieve(String sname){
        lock.lock();
        try{
            ArrayList<Person> dt = new ArrayList<Person>();
            Person p = new Person("",sname,0); //use for search
            for(int j = 0; j < data.size();j++){
                Person p1 = data.get(j);
                if(p1.equals(p)) dt.add(p1);
            }
            return dt;
        }finally{lock.unlock();}
    }
}
```

The server follows the template given earlier. It creates an instance of Data and passes it to each thread launched to handle client requests.

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.concurrent.locks.*;
public class UpDownPersonServer{
    final static int portPerson = 1234; // any number > 1024
    public static void main(String[] args){
      System.out.println("Server running...");
      Data data = new Data();
      try{
        ServerSocket servesock = new ServerSocket(portPerson);
        while (true) {
          // wait for a service request on port portSqrt
          Socket socket = servesock.accept();
          // start thread to service request
          new PersonUpDown(socket,data).start();
        }
      }catch (IOException e) {e.printStackTrace();}
  }
}
```

The thread PersonUpDown is designed to deal with both types of client. The client begins the conversation by indicating what action they wish to undertake. Those uploading send a 0, those retrieving send 1. In the case of upload the thread creates a new default instance of Person and uses its readInputStream method to upload the information. It then adds the new Person object to the data structure and sends a Boolean value to the client indicating success. In the case of a retrieve client it reads the surname sent by the client, invokes the retrieve method on the data structure and then sends the data to the client. Note that it begins this communication by sending the number of matches to the client first. Again the actual data is sent by invoking the writeOutputStream method of the person instance.

```
class PersonUpDown extends Thread{
  Socket socket;
  Data data;
  PersonUpDown(Socket s, Data d){socket = s; data = d;}
  public void run() {
    try{
      DataInputStream in = new DataInputStream(socket.getInputStream());
      DataOutputStream out = new
                                DataOutputStream(socket.getOutputStream());
      int opt = in.readInt();
      if(opt == 0){ //upload
```

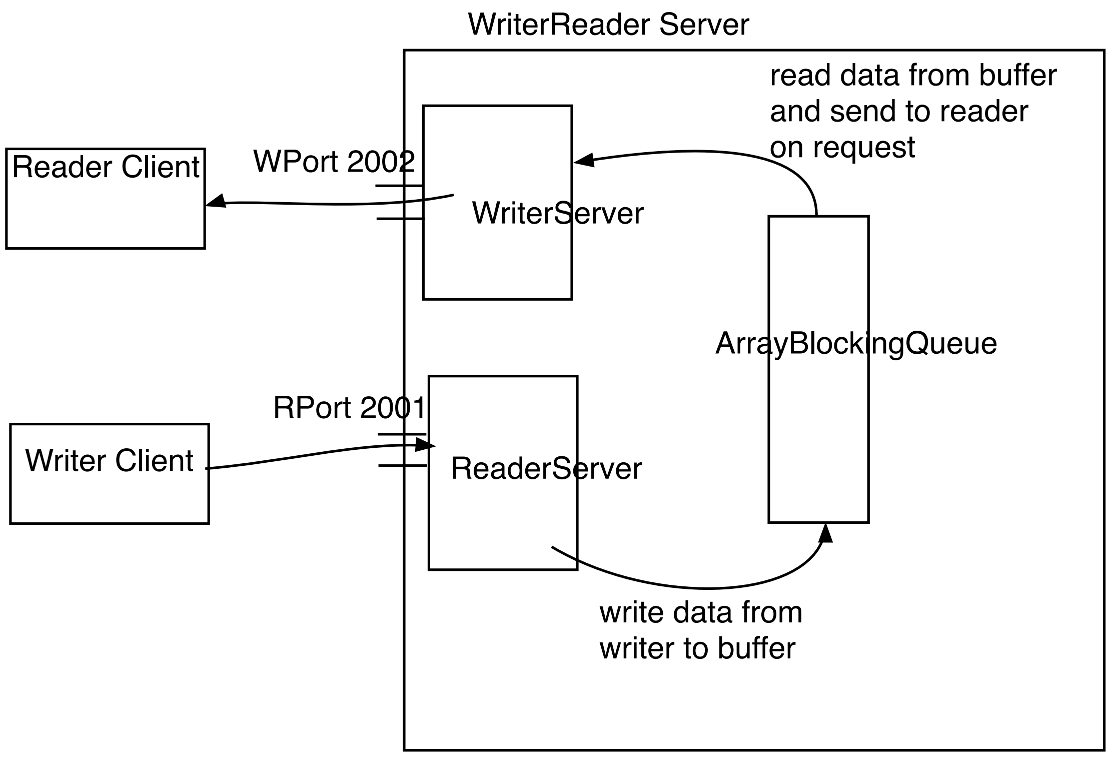
```
        Person p = new Person();
        p.readInputStream(in);
        data.add(p);
        out.writeBoolean(true);
        socket.close();
      }
      else{ //download
        String sname = in.readUTF();
        ArrayList<Person> lst = data.retrieve(sname);
        out.writeInt(lst.size());
        for(int j = 0; j < lst.size();j++){
           Person p = lst.get(j);
           p.writeOutputStream(out);
        }
        socket.close();
     }
    }
   catch (IOException e){}
 }
}
```

To test the system two very simple clients are provided. The program PersonUploadClient writes a number of instances of the Person class to the server and PersonRetrieveClient retrieves all persons whose surname is Joyce. The upload client uses the function write(Person p) that connects to the server sending 0 to indicate it is an upload client and then uses the writeOutputStream of the person class to transfer the actual data. The download client connects to the server sending 1 to indicate that it is a download client, sends the surname to search for and then retrieves the data. If the return value is 0 it prints the message *No matches found*; otherwise it reads the data from the input stream using the readInputStream method of the Person instance and displays it on the screen.

```
import java.io.*;
import java.net.*;
import java.util.*;
public class PersonUploadClient {
 final static int portPerson = 1234;
 public static void main(String[] args) {
      write(new Person("James","Joyce",50));
      write(new Person("Nora","Joyce",40));
      write(new Person("Stan","Joyce",60));
      write(new Person("Sam","Beckett",40));
      write(new Person("Lucia","Joyce",10));
 }
 static void write(Person p){
   try{
     Socket socket;
     socket = new Socket(InetAddress.getLocalHost(),portPerson);
     DataInputStream in = new DataInputStream(socket.getInputStream());
    DataOutputStream out = new DataOutputStream(socket.getOutputStream());
    out.writeInt(0); //send upload option
    p.writeOutputStream(out);
    boolean ok = in.readBoolean();
    if(!ok) System.out.println("Error");
    socket.close();
   } catch (IOException e) {System.out.println(e);}
  }
}

import java.io.*;
import java.net.*;
import java.util.*;
public class PersonRetrieveClient {
  final static int portPerson = 1234;
  public static void main(String[] args) {
   try{
     Socket socket;
     socket = new Socket(InetAddress.getLocalHost(),portPerson);
     DataInputStream in = new DataInputStream(socket.getInputStream());
     DataOutputStream out = new
                                    DataOutputStream(socket.getOutputStream());
     out.writeInt(1); // send download option
     out.writeUTF("Joyce");
     int k = in.readInt(); //retrieve number of matches
     if(k == 0)
        System.out.println("No matches found");
     else{
      Person p = new Person();
      for(int j = 0; j < k; j++){
        p.readInputStream(in);
        System.out.println(p);
      }
      socket.close();
     }
   }catch (IOException e) {System.out.println(e);}
  }
}
```

Bounded Buffer Server

This example illustrates the consumer-producer problem where a server is used to manage the bounded buffer that holds data passed from a producer to a consumer. The server uses an instance of ArrayBlockingQueue to implement the bounded buffer. This is a standard buffer that forces consumers to wait when the buffer is empty and producers to wait when it is full. We could use any of the bounded buffers implemented earlier with condition variables and semaphores but to illustrate Doug Lea's version is useful.



The diagram of the communication server shows two servers in the one application. One server, ReadServer, reads data from the producer (writer client) and stores it in the buffer. The other server, WriteServer, manages the consumer (reader client) by sending data on request from the buffer. To simplify the solution the servers only handle one client at the time. They literally copy data from one application to another. This model could be used to transfer any data type but here we just use integer values. A sentinel is used to mark the end of the update and is stored in the buffer by the reader. The writer reads from the buffer until the sentinel is retrieved. It then terminates its read loop and sends the sentinel to the consumer. A consumer attempting to read from an empty buffer will sleep until data is produced by the producer. Note that in this example we use two different ports: one for producers and the other for consumers.

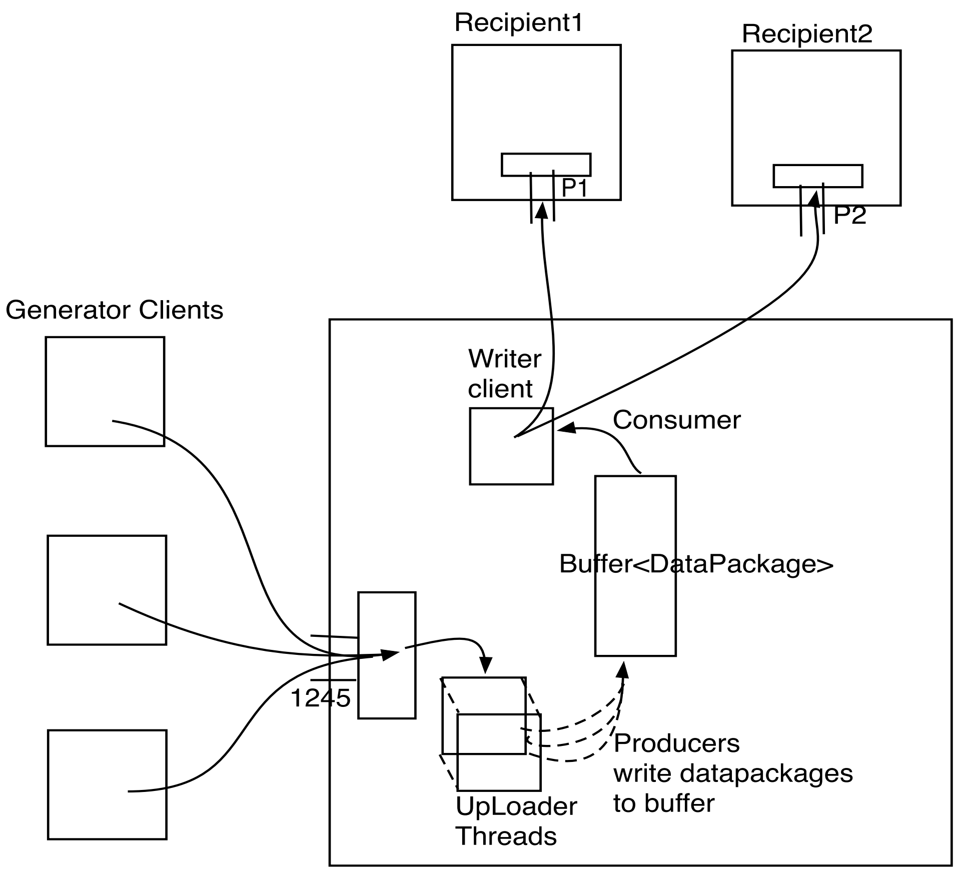
```
import java.io.*;
import java.net.*;
import java.util.concurrent.*;
import java.util.*;
public class WriterReaderServer {
    final static int RPort = 2001;
    final static int WPort = 2002;
    final static int sentinel = Integer.MIN_VALUE;
    public static void main(String[] args) {
      ArrayBlockingQueue<Integer> buffer = new
                                              ArrayBlockingQueue<Integer>(10);
      new ReaderServer(buffer, RPort).start();
      new WriterServer(buffer, WPort).start();
    }
}
class ReaderServer extends Thread{
    private ArrayBlockingQueue<Integer> buffer;
    private int port;
    ReaderServer(ArrayBlockingQueue<Integer> b, int pt){buffer = b; port = pt;}
    public void run(){
       try {
          ServerSocket readsock = new ServerSocket(port);
           while (true) {
              Socket socket = readsock.accept();
              readData(socket);
           }
       } catch (IOException e) {e.printStackTrace();}
    }
    private void readData(Socket sk){
      try {
        DataInputStream in = new DataInputStream(sk.getInputStream());
        int x = in.readInt();
       while(x != sentinel){
         try{
           buffer.put(new Integer(x));
          System.out.print(x+" ");
        }catch(InterruptedException ei){}
        x = in.readInt();
       }
       try{
          buffer.put(new Integer(Integer.MIN_VALUE));
        }catch(InterruptedException ei){}
        in.close();
     } catch (IOException e) {}
   }
}
class WriterServer extends Thread{
    private ArrayBlockingQueue<Integer> buffer;
    private int port;
    WriterServer(ArrayBlockingQueue<Integer> b, int pt){
          buffer  = b; port = pt;
     }
    public void run(){
       try {
          ServerSocket writesock = new ServerSocket(port);
          while (true) {
             Socket socket = writesock.accept();
             writeData(socket);
          }
       } catch (IOException e) {e.printStackTrace();}
    }
    private void writeData(Socket sk){
      try {
       DataOutputStream out = new DataOutputStream(sk.getOutputStream());
       Integer x = null;
       try{x = buffer.take();}catch(InterruptedException ei){}
       while(x.intValue() != sentinel){
         out.writeInt(x.intValue());
         try{x = buffer.take();}catch(InterruptedException ei){}
       }
       out.writeInt(Integer.MIN_VALUE);
```

```
        out.flush();
        out.close();
    } catch (IOException e) {}
  }
}



import java.io.*;
import java.net.*;
public class WriterClient {
   final static int Port = 2001; // writer sends data to reader port on server
   public static void main(String[] args) {
     try{
      Socket socket = new Socket(InetAddress.getLocalHost(),Port);
      DataOutputStream out = new
                                        DataOutputStream(socket.getOutputStream());
      for(int j = 0; j < 100; j++){
          int x = (int)(Math.random()*100);
          out.writeInt(x);
          if((j+1) % 10 == 0)
              try{ Thread.sleep(2000);}catch(InterruptedException ei){}
      }
      out.writeInt(Integer.MIN_VALUE);
      socket.close();
   } catch (IOException e) {System.out.println(e);}
 }
}

import java.io.*;
import java.net.*;
public class ReaderClient {
  final static int Port = 2002; // reads from output port of server
  public static void main(String[] args) {
    try{
      Socket socket = new Socket(InetAddress.getLocalHost(),Port);
       DataInputStream in = new DataInputStream(socket.getInputStream());
```

```
        int x = in.readInt(); System.out.println();
        while(x != Integer.MIN_VALUE){
          System.out.print(x+" ");
          x = in.readInt();
        }
        socket.close();
      } catch (IOException e) {System.out.println(e);}
  }
}
```

Routing Server Example

In the client server model the server only responds to client requests. It sits in a loop waiting to receive a communication from a client. When a client connects it can upload or download information as appropriate. In this next example we want to show that servers can act as both the recipient of data from a client and also distribute data to external clients that act like servers. To illustrate the idea we develop a routing server that receives information from external clietns – called generator clients here – and routes the data to external client recipients. These recipient clients are actual servers listening on specific ports. To facilitate communication we introduce a class called DataPackage that encapsulates both the data to be routed and the port number to send it to. If the recipient was not a local host we would also need the IP address of the machine. In the example listed below the data is just a list of integer values. The diagram below shows the flow of data and lists the components of the server. A buffer is used to store data packages received by the individual uploader threads (producers) used to communicate with generator clients. A writer thread (consumer) reads from the buffer and sends data to recipient clients.



The Writer thread reads from this bounded buffer, waiting if the buffer is empty. The server only uses a single writer thread. Clients send the port number to deliver the data to and a list of numbers to deliver. This information is encapsulated by the class DataPackage. This class also *knows* how to write to its intended output buffer. The method writeClient opens a socket to communicate with the intended port number on the local host and uses a data output stream to send the data to the port. The code for this method is:

```
void writeClient(){
  try{
    Socket socket;
    socket = new Socket(InetAddress.getLocalHost(),port);
    DataOutputStream out = new
                              DataOutputStream(socket.getOutputStream());
    out.writeInt(data.size());
    for(Integer k : data) out.writeInt(k);
    out.flush();
    socket.close();
  } catch (IOException e) {System.out.println("Package");System.out.println(e);}
}
```

Each time a client connects the server creates an UpLoader thread instance that shares access to the bounded buffer. In effect this thread becomes the producer: it reads the intended port number to send the data to, creates a new data package and then reads all the data from the client. It then writes the data package to the buffer, waiting if the buffer is full.

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.concurrent.locks.*;
import java.util.concurrent.*;
public class AuotmaticDownloadServer {
   final static int portUp = 1234;
   public static void main(String[] args){
      System.out.println("Server running...");
      Buffer<DataPackage> store = new Buffer<DataPackage>(50);
      new Writer(store).start();
      try{
            ServerSocket servesock = new ServerSocket(portUp);
            while (true) {
                // wait for a service request on port portSqrt
               Socket socket = servesock.accept();
               // start thread to service request
               new UpLoader(socket,store).start();
            }
        }catch (IOException e) {e.printStackTrace();}
   }
}
class UpLoader extends Thread{
  Socket socket;
  Buffer<DataPackage> buffer;
  UpLoader(Socket s, Buffer<DataPackage> d){socket = s; buffer = d;}
  public void run() {
    try{
        DataInputStream in = new DataInputStream(socket.getInputStream());
        int port = in.readInt(); //get port number to send data to
        DataPackage dp = new DataPackage(port);
        int k = in.readInt();
        while(k != -1){ //-1 is sentinel
           dp.add(k);
           k = in.readInt();
        }
        socket.close();
        buffer.put(dp);
     }
     catch (IOException e){}
  }
}
class Writer extends Thread{
  Buffer<DataPackage> buffer;
  Writer(Buffer<DataPackage> d){buffer = d;}
  public void run() {
    while(true){
        DataPackage dp = buffer.get();
        dp.writeClient();
    }
  }
}
final class DataPackage{
    //assumes that it is on a local host
    // otherwise we would need the ip address as well
    private final int port;
    private final ArrayList<Integer> data = new ArrayList<Integer>();
    DataPackage(int p){port = p;}
    void add(Integer k){data.add(k);}
    void writeClient(){
      try{
        Socket socket;
        socket = new Socket(InetAddress.getLocalHost(),port);
        DataOutputStream out = new
                                    DataOutputStream(socket.getOutputStream());
        out.writeInt(data.size());
        for(Integer k : data) out.writeInt(k);
        out.flush();
       socket.close();
      } catch (IOException e) {System.out.println(e);}
    }
}
class Buffer<E>{
   private int max;
   private int size = 0;
   private ArrayList<E> buffer;
   private Semaphore empty;  // control consumer
   private Semaphore full;   // control producer
   private Lock lock = new ReentrantLock();
   public Buffer(int s){
     buffer = new ArrayList<E>();
     max = s;
     empty = new Semaphore(0);
     full = new Semaphore(max);
  }
  public void put(E x){
    try{
       full.acquire();
    }
    catch(InterruptedException e){}
    // synchronize update of buffer
    lock.lock();
    try{
       buffer.add(x);
       size++;
       empty.release();
    }finally{
       lock.unlock();
    }
  }
  public E get(){
       try{
           empty.acquire();
       }
       catch(InterruptedException e){}
       // synchronize update of buffer
       lock.lock();
       try{
       E temp = buffer.get(0);
          buffer.remove(0);
          size--;
          full.release();
          return temp;
       }finally{
       lock.unlock();
       }
  }
}
```

To test this server you need to create generator clients and some recipients. This is left as an exercise for the reader.

Using Threadpools and Controlling Server Access

The server template we have been using to implement our servers has some serious limitations. Firstly, it creates a new thread every time a client connects. If there is a lot of client activity then this over head is costly and greatly inhibits the performance of the server because creating and destroying threads frequently is computationally expensive. This overhead can be removed by using a thread pool because the server can create a pool of threads when it starts and then re-use them to handle client requests. A second problem with our server template is that it can be attacked by clients. The performance of a server is inversely proportional to the number of concurrent clients it has to deal with. The greater the number of clients to be communicated with the slower the service because threads are running concurrently and must be scheduled. Again, this is computationally expensive. To some extend the use of a thread pool helps to solve this because the server only deals with n clients, where n equals the number of threads in the pool, concurrently even though it does not impose a limit on the number of client requests that it receives. These are stored in a job queue waiting for threads to become available to service them. An additional aid is to use a semaphore to control the number of clients that can concurrently submit jobs at any one time. The initial value of the semaphore is set and as jobs get submitted the semaphore is decreased. When it reaches zero no more jobs are accepted. Every time a worker thread completes a job it increments the count on the control semaphore.

To illustrate this new template for servers we write a server that validates user names and passwords. The server is required to manage a list of user names and passwords. A client simply submits details to determine if they are registered.

In this case we will use a class that implements the serializable interface to specify a protocol for communication between the client and the server. A client submits an instance of the Password class to the server for verification. The class is collection compliant for use by an ArrayList container.

```
class Password implements java.io.Serializable{
    String name;
    String password;
    public Password(){name = ""; password = "";}
    public Password(String n, String p){
        name = n; password = p;
    }
    public boolean equals(Object ob){
        if(!(ob instanceof Password)) return false;
        Password pw = (Password)ob;
        return name.equals(pw.name) && password.equals(pw.password);
    }
    public String toString(){
        String s = name +" "+password;
        return s;
    }
}
```

The Server uses a thread pool to handle client communication. Client requests are submitted to the pool and handled by the threads in the pool. The static method loadPasswords reads a text file from disk containing the user names and passwords.

The file is formatted one user per line:

```
Tony tony123
Joe joe123
Fran fran123
Kevin kevin123
```

The data is copied to an ArrayList of Password in memory and shared by all threads in the server. The function uses a BufferedReader to read each user from the file and then uses a Tokenizer to extract tokens used to construct an instance of Password. The function is listed below.

```
static void loadPasswords(ArrayList<Password> p){
    String s;
    StringTokenizer t;
    try{
        FileReader fr = new FileReader("Password.txt");
        BufferedReader in = new BufferedReader(fr);
        s = in.readLine();
        while(s != null){
            t = new StringTokenizer(s);
            Password pWord = new Password(t.nextToken(),t.nextToken());
            p.add(pWord);
            s = in.readLine();
        }
        in.close();
    }catch(IOException e){System.out.println(e.getMessage());}
  }
```

The server creates a thread pool of 10 threads and allows a maximum of 100 jobs to be submitted to the pool at any one time. This is controlled by the semaphore. Each thread releases the semaphore when it completes a job. (These numbers are arbitrary and in reality may be quite different. They simply serve to illustrate the idea.)

```
class PasswordServer{
  final static int portNum = 1234;
  final static int numThreads = 10;
  static ExecutorService pool;
  static Semaphore sem = new Semaphore(100);
  public static void main(String[] args){
    ArrayList<Password> passwords = new ArrayList<Password>();
    loadPasswords(passwords);
    pool = Executors.newFixedThreadPool(numThreads);
    System.out.print("Server running ...");
    try{
      ServerSocket servesock = new ServerSocket(portNum);
      while (true){
        try{
          sem.acquire();
        }catch(InterruptedException e){}
        Socket socket = servesock.accept();
        //submit request to pool
        pool.submit(new CheckPassword(socket,passwords,sem));
     }
    }catch(IOException e){}
  }
  //...
}
```

The class CheckPassword has three arguments: the array list of persons, the socket used to communicate with the client and the semaphore used to manage the number of jobs submitted at any one time to the server. It uses an object input stream to read the serialized object from the client and returns a Boolean value, indicating registered or not registered, over the data output stream.

```
class CheckPassword implements Runnable{
    Socket socket;
    ArrayList<Password> pWords;
    Semaphore sem;
    public CheckPassword(Socket soc, ArrayList<Password> p,
                                                        Semaphore s){
        socket = soc;
        pWords = p;
        sem = s;
    }
    public void run() {
      try{
        ObjectInputStream in = new
                              ObjectInputStream(socket.getInputStream());
        DataOutputStream out = new
                              DataOutputStream(socket.getOutputStream());
        Password pw = (Password)in.readObject();
        if(pWords.contains(pw)) // password found
           out.writeBoolean(true);
        else
          out.writeBoolean(false);
        socket.close(); // close connection
        sem.release();
      }
      catch (IOException e) {}
      catch(ClassNotFoundException e1){}
   }
}
```

The client application reads a users name and password, constructs an instance of the Password class and uses an object output stream to send it to the password server. It then waits for a reply and prints an appropriate message.

```
import java.io.*;
import java.net.*;
import java.util.*;
class Login{
    private final static int port = 1234;
    public static void main(String args[]){
      try{
         Socket socket;
        socket = new Socket(InetAddress.getLocalHost(),port);
        DataInputStream in = new
                            DataInputStream(socket.getInputStream());
        ObjectOutputStream out = new
                            ObjectOutputStream(socket.getOutputStream());
        // get user name and password
        Scanner keyIn = new Scanner(System.in);
        System.out.print("User name: ");
        String user = keyIn.next();
        System.out.print("Password:  ");
        String pw = keyIn.next();
        Password pWord = new Password(user,pw);

        //write object and flush any buffered data
        out.writeObject(pWord);
        out.flush();

        // wait for reply from server
        boolean valid = in.readBoolean();
        if(valid)
             System.out.println("Registered user");
        else
             System.out.println("Not registered");
        socket.close();
        in.close();
        out.close();
     }catch(IOException e){System.out.println(e);}
   }
}
```

Non-blocking Synchronization and Lock-Free Data Structures

Many of the classes in java.util.concurrent provide better performance and scalability than alternatives that use synchronization. The main reason for this is the use of what is called non-blocking synchronization. This approach provides a way to write data structures that are thread safe and do not use locking. Exclusive locking is a pessimistic technique – it assumes a worst-case scenario. Locking assumes conflict is possible and protects against it whether it is necessary or not. It provides a guarantee that locked blocks are atomic and only one thread can be active in a locked block at any one time. It makes no assessment of the need to lock in a given context. Taking this approach is computationally expensive because threads have to be managed in queues and put to sleep. An alternative approach, called the optimistic approach, proceeds with an update, hopeful that it can complete without interference. This approach relies on collision detection to determine if there has been interference from other threads during the update, in which case the operation fails and can be retried. In situations where contention rates between threads are high this approach can result in lots of polling. This means that the threads are repeatedly trying to update and failing. This wastes cpu time. However, this approach takes the view that contention rates among competing threads in general are low and that using locks to guard against them is a computational expense that is in the main unnecessary.

Goetz (*Java Concurrency in Practice*, 2006) describes a benchmark test based on a random number generator implemented using locking and the optimistic non-blocking lock-free approach. He concludes that at high contention rates locking tends to outperform atomic variables because of the overhead of polling. The reason is that a lock reacts to contention by suspending threads, reducing cpu usage and synchronization traffic on the shared memory bus. But at more realistic contention levels atomic variables outperform locks because with atomic variables contention management is pushed back to the calling class. Note that the term *non-blocking* means that failure or suspension of one thread does not cause failure or suspension of another thread. The term *lock-free* means that at each step some thread can make progress. To quote Goetz: *An uncontended CAS always succeeds, and if multiple threads contend for a CAS, one always wins therefore makes progress*(p 329).

There are distinct advantages to this optimistic approach to resolving race conditions, mutual exclusion and atomic actions. These can be summarized as follows:

- No possibility of deadlocks because there is no locking
- Offer high performance when different items in the data structure are accessed because a single lock does not block threads.

- Atomic variables offer the same memory semantics as volatile variables but with additional support for atomic updates. Volatile variables cannot be used to construct atomic compound actions and, therefore, cannot be used to implement counters or mutexes.
- Atomic variables are finer-grained and lighter weight than locks.
- They do not cause priority inversion whereas locks do. By priority inversion we mean that a thread with a low priority value can hold up a thread with a higher priority value. A thread with a low priority value can obtain a lock, hold it and, hence, force a high priority thread to wait on the lock. This can have serious consequences in Real-time applications where priority values are used to denote thread that cannot miss deadlines. Usually, these environments try to resolve this problem by dynamically raising the priority of a thread when it gains a lock.

There are also some disadvantages. These can be summarized as follows:

- If there is contention they busy wait when updating the data structure. This can cause liveness problems. It also uses up lots of cpu scheduling time because threads get the cpu and don't do any useful work.
- Difficult to code. This will be demonstrated below!
- Hardware Support is necessary. Processors designed for multiprocessor operations must provide special instructions for managing concurrent access to shared variables. To write lock free code it is necessary to access the instructions provided by the machine.

Machine Support

Most modern processors have what they call a compare-and-swap(CAS) instruction. CAS is a low-level fine-grained atomic hardware primitive that allows multiple threads to update a single memory location.

Suppose we want to change `memWord` to `newValue` atomically so that it is thread safe we can use

```
cas(memWord,expectValue,newValue)
```

The variable `expectValue` is assigned the `memWord` and then the `cas` instruction is invoked.

The semantics of this instruction can be described by an `if` statement as follows:

```
expectValue = memWord;
if(memWord == expectValue){
    memWord = newValue;
    return true;
 }
 else
       return false;
```

Because cas may fail, i.e. return false, it must be used in a loop. This happens when another thread changes the memWord before the change takes place. For example, a function to add k to variable x might be written as follows:

```
    // x global
    void atomicAdd(k) {
      int cx = x; //copy memory value
      int newValue = cx + k; // update it
      while (!cas(x,cx,newValue) {
         cx = x;
         newValue = cx+k;
      }
}
```

The loop continues to iterate while cas continues to fail. It only terminates when it succeeds. Note that we re-read the memory value of x each time in the body of the loop because cas fails due to the fact that x has been modified by another thread between reading it and trying to update it with cas. This is where the polling comes into the picture. When multiple threads attempt to update the same variable simultaneously using cas, one wins and updates the variable's value, and the rest lose. However, the losers are not put to sleep because they failed to acquire a lock; instead they continue trying to complete an update. Of course, if cas succeeds the first time, and in normal contention situations this is the case, the update occurs immediately and there is no locking computational cost.

In version 5.0, and all subsequent versions, Java provided low-level support to the JVM to expose the compare-and-swap (CAS) operation and, hence, provide a conditional update operation that ultimately calls the hardware CAS instruction. The name of this method is:

```
boolean compareAndSet(expectedValue, updateValue);
```

In one atomic action, this operation compares the contents of a given memory location with a supplied expected value and if they are equal, updates the value with the supplied new value, otherwise the operation fails. This method is used in the library developed by Doug Lea. The concurrency utilities can be divided into the following:

- Atomic package
- Concurrency package
- Concurrent Collection classes

The Atomic Package offers a toolkit of classes that support lock-free thread safe programming on **single** variables. There are twelve atomic variable classes, divided into four groups: scalars, field updaters, arrays, and compound variables. We will just study the scalar collection because they are the most widely used group and also because they serve to illustrate the main points we wish to make here. The four most relevant of these scalars are called AtomicInteger, AtomicLong, AtomicBoolean and AtomicReference. The methods for AtomicInteger are: get() returns current value; set(int x) that changes the current value to x; getAndSet(int x) that changes value to x and returns old value; getAndIncrement() that increments current value; getAndDecrement() that decrements the current value. There is also methods for incrementAndGet() and decrementAndGet() with the obvious semantics. Similar methods are supported by AtomicLong. AtomicBoolean has methods: get(), set(boolean b) and getAndSet(boolean b). The class AtomicReference has methods: get(), set(V x) and getAndSet(V x). All four classes have a compareAndSet(expectedValue, newValue) that atomically sets the value to the newValue if the expectedValue equals the currentValue. These classes provide a generalization of volatile variables to support atomic conditional read-modify-write operations. They provide the same visibility as volatile variables with the added bonus that they support atomic state change. However, they only do so for single variables and are not suitable to maintain state invariants.

A thread safe implementation of a Counter class could use an AtomicInteger to hold the current value of an instance of a counter. In the following listing we provide such a thread safe class that has two public methods inc() and dec() with the obvious semantics. The counter has an initial value of 0.

```
class Counter{
   private final AtomicInteger value = new AtomicInteger(0);
   public void inc(){
      value.incrementAndGet();
   }
   public void dec(){
      value.decrementAndGet();
   }
   public String toString(){
      return value.toString();
   }
   public int count(){
      return value.get();
   }
}
```

We demonstrate how to use the method compareAndSet by writing a class that provides the same functionality as the Counter class except that we use the compareAndSet method provided by the AtomicInteger class to make both inc and dec atomic. Because compareAndSet may fail, simply because another thread won, it must repeatedly try until it succeeds. The pattern is to read the current value, use compareAndSet to atomically update the memory value if it has not changed. If change has occurred try again. This can be expressed as follows:

```
int v = value.get(); //read the current value
while (!value.compareAndSet(v, v + 1)){ //try to increment
   v = value.get(); //read again if it fails
}
```

The non blocking solution is given below.

```
class NonBlockingCounter {
   private AtomicInteger value;
   public int get() {
        return value.get();
    }
    public int inc() {
        int v = value.get();
        while (!value.compareAndSet(v, v + 1)){
            v = value.get();
        }
        return v + 1;
    }
    public int dec() {
        int v = value.get();
        while (!value.compareAndSet(v, v - 1)){
            v = value.get();
        }
        return v - 1;
    }
}
```

Note: The methods for `AtomicInteger` are lock-free and the code for `getAndDecrement()` is as follows:

```
public final int getAndDecrement() {
   for (;;) {
      int current = get();
      int next = current - 1;
      if (compareAndSet(current, next))
         return current;
   }
}
```

The `compareAndSet` used here is a local method that calls the underlying method to do the decrement. See `AtomicInteger` class. **End Note.**

Atomic classes are fine for providing atomic modifications on single variables. However, they cannot be used to guarantee state invariants where more than a single modification is required. To illustrate this we consider the case of an integer stack class where a bounded array is used to manage elements in the stack. The variable `size` records the current number of elements in the stack. Pushing a new element onto the stack involves updating two variables that are in an invariant relationship. Therefore, modifying the stack by adding a new element and updating size must be done atomically. This can easily be solved by using `Reentrant` locks or by using `synchronized` blocks. However, for this exercise we propose to use an `AtomicBoolean` variable to enforce locking and then use its `compareAndSet` method to enforce *spin* locking it. Only a single thread succeeds in modifying the state of the lock and all other competing threads will *spin-lock*, i.e. they will continue to iteratively try the lock until they succeed. The `AtomicBoolean` lock variable is initialized to `false`. When threads compete by concurrently trying to push values on a shared stack the `compareAndSet` method tests the expected value, `false`, with the current value and if they match its state changes to `true`. This is done atomically and, hence, will occur for only one thread. All other threads will find that the current value does not match and, hence, spin-loop. When the winning thread acquires the lock it checks that there is room, updates both the stack and its associated size and then sets the lock to `false`. The situation in the case of `pop` is similar.

```
class IntStack{
    private Integer[] stack = new Integer[100];
    private int size = 0;
    //invariant: stack[0..size-1] is stack
    private AtomicBoolean lock = new AtomicBoolean(false);
    public boolean push(Integer x){
        while(true){
            if(lock.compareAndSet(false,true)){//try to lock
                if(size < stack.length){
                    stack[size] = x;
                    size++;
                    lock.set(false);
                    return true;
                }
                else{
                    lock.set(false);
                    return false;
                }
            }
        }
    }
    public Integer pop(){
      while(true){
            if(lock.compareAndSet(false,true)){//try to lock
                if(size > 0){
                    Integer k = stack[size-1];
                    size--;
                    lock.set(false);
                    return k;
                }
                else{
                    lock.set(false);
                    return null;
                }
            }
        }
    }
}
```

Nonblocking Algorithms

Nonblocking algorithms are considerably more complicated to write than their lock based equivalents. The trick is to limit the scope of atomic changes to a single variable while maintaining data consistency. In the case of linear structures such as linked lists and stacks this is possible because state transformations can be limited to modifying individual links. This can be implemented using AtomicReference. To demonstrate we develop a generic linked list class where new nodes are inserted at the head of list. The generic class Node is as follows and is implemented as a private static class in the LinkedList class.

```
    class Node<E> {
       volatile Node<E> next;
       volatile E item;
    }
```

An AtomicReference variable of type Node<E> is used for the head of the list. New elements are inserted at the head and, hence, head will be modified possibly by competing threads sharing an instance of the list. Updating this reference variable becomes thread safe by using the methods of the AtomicReference class. The method put takes an instance of E as argument and creates a new node for it. It then assigns the reference of the current head to oldHead and assigns newHead.next the value of oldHead. Then the compareAndSet method is used to change the existing head to newHead. This may fail so it is controlled by a loop that *spins* until it succeeds.

```
public void put(E item) {
      Node<E> newHead = new Node<E>();
      newHead.item = item;
      Node<E> oldHead = head.get();
      newHead.next = oldHead;
      while (!head.compareAndSet(oldHead, newHead)) {
          oldHead = head.get();
          newHead.next = oldHead;
      }
  }
```

The get() method works in a similar manner. The complete class is listed below.

```
class LinkedList<E>{
  private AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();
  public void put(E item) {
      Node<E> newHead = new Node<E>();
      newHead.item = item;
      Node<E> oldHead = head.get();
      newHead.next = oldHead;
      while (!head.compareAndSet(oldHead, newHead)) {
          oldHead = head.get();
          newHead.next = oldHead;
    }
  }
  public E get(){
      Node<E> oldHead = head.get();
      if (oldHead == null) return null;
      Node<E> newHead  = oldHead.next;
      while (!head.compareAndSet(oldHead,newHead)) {
          oldHead = head.get();
          if (oldHead == null) return null;
          newHead = oldHead.next;
      }
      E temp = oldHead.item;
      oldHead.item = null;
      return temp;
   }
  public E head(){
    return head.get().item;
   }
  public boolean isEmpty(){
    return head.get() == null;
  }
   private static class Node<E> {
        volatile Node<E> next;
        volatile E item;
   }
 }
```

We use a linked list to implement a generic Stack. New items are inserted at the head of the stack and pop will remove the item at the head, if any.

```
class Stack<E>{
    LinkedList<E> stack = new LinkedList<E>();
    void push(E x){
        stack.put(x);
    }
    E pop(){
        return stack.get();
    }
    E top(){return stack.head();}
    boolean isEmpty(){ return stack.isEmpty();}
}
```

Functions and Parallel Streams

Introduction

There has been a lot of research into the relationship between object-oriented programming and functional programming and many of the modern programming languages are deemed to be multi-paradigm because they support both styles of program construction. Programming languages such as Scala, F# and Groovy (Strachen, 2007) all support both object-oriented and functional programming styles. The developers of both Scala and Groovy built their functional language on top of the Java platform. In doing so they provided a language that supports both state based object-oriented programming and functional programming based on immutable state. Prior to the release of Java 8 in 2014 Java supported only imperative state based object-oriented programming. All class methods are written using a sequential imperative style of programming. The language did not support a functional programming style. With Java8, Oracle have followed the lead taken by other modern programming languages by making it possible to write java programs in a functional or declarative style using both the functional library and the streams library. This style of program construction forces one to think in terms of functions, streams and immutable state instead of loops or recursion and mutable state. The supporters (promoters) of this style of programming argue that it puts much more emphasis on *what* we want our programs to do and much less stress on *how* models are constructed. Both the imperative and object-oriented models, they argue, place too much emphasis on *how* and not enough on *what*. Martin Odersky, creator of Scala, argues that we should adopt a functional style by concentrating on the transformation of immutable values instead of stepwise modifications of mutable state. Brian Goetz, chief architect on design of Java 8, argues that both sequential and parallel programming are greatly *facilitated by shifting the focus towards describing what computation should be performed, rather than how it should be performed.* Rich Hickey, creator of Clojure, argues that functional programming provides higher levels of abstraction that allow us to see problem solutions more clearly.

We now provide a summary discussion of lambda expressions, functions and streams. For a complete discussion of these topics see *Programming Paradigms with Java, Mullins, Tony 2014*.

Functions and Lambda Expressions

A function is a definition that associates a set of input parameters with a single output parameter. It consists of two parts: a signature and an equation that defines its semantics. We write functions using what are called Lambda Expressions (The Lambda Calculus was developed by Alonzo Church in the 1930's to define computable functions). These types of expression are also called function literals or anonymous functions. Java uses typed lambda expressions that have their origin in functional languages such as Haskell and ML. The syntax for a lambda expression is a list of parameters, in parentheses, a right arrow, and then the body of the function that may or may not be enclosed in curly brackets. An example of a function literal is:

```
(Integer x) -> x + 1
```

The part preceding the arrow is the parameter list and the part following the arrow is the body of the function. The result type of the expression in the body of the function is its result type. It is inferred from the context and is never specified.

In Java to implement lambda expressions we use what are called functional interfaces. In pure functional languages function types are structural but because Java is not a functional programming language they are implemented using specific interfaces where the interface declares its intent (called nominal typing). There are a number of different functional interfaces defined in the package java.util.function and these are discussed in detail in the text named above. In this summary we use the two most general cases: Function<T,R>, that takes a single argument of type T and has return type R, and BiFunction<T,U,R> that takes two arguments of type T, U and has return type R. Each of these has a single abstract method called apply that is implemented by a given lambda expression.

Using these defined functional interfaces we can give aliases to our function literals by assigning them to variables. The following code block assigns each of our lambda expressions to appropriately named variables. Each one has a single argument and a specific return type.

```
Function<Integer,Integer> inc = x -> x + 1;
Function<Integer,Boolean> even = x -> x % 2 == 0;
Function<Integer,Boolean> pos = x -> x > 0;
Function<Integer, Integer> abs = x -> x >= 0 ? x : -x;
Function<Integer,Integer> sum = n -> {
  int s = 0; for(int j = 0; j < n; j++) s = s +(j+1);
  return s;
};
BiFunction<Integer,Integer,Integer> add = (x,y) -> x + y;
```

At compile time each of these function literals is compiled into a class that when instantiated at run-time is a function value. A function value is an object just like any other object. We invoke the function instance by referencing its apply method as you would do for any other object instance.

The code fragment given below illustrates how to apply these function aliases to actual argument values. The values output on execution of this code are: *8, false, false, 7, 55, 8*.

```
System.out.println(inc.apply(7));
System.out.println(even.apply(9));
System.out.println(pos.apply(-9));
System.out.println(abs.apply(-7));
System.out.println(sum.apply(10));
System.out.println(add.apply(3,5));
```

A function Predicate takes a single argument T and a BiPredicate takes two arguments, not necessarily of the same type. Both return a boolean value. The method name for application is: test. Two examples are given:

odd that takes an integer as argument and returns true if it is an odd value, false otherwise;

existsOdd that takes a list of integers returning true if it contains an odd value, false otherwise;

```
Predicate<Integer> odd = x -> x % 2 != 0;
Predicate<List<Integer>> existsOdd = lst ->{
 for(Integer x : lst) if(odd.test(x)) return true;
 return false;
};
```

Specialized Function Types

Function Name	**Argument Type**	**Return Type**	**Abstract Method Name**	**Purpose**
Supplier<T>	None	T	get	Takes no argument and return a value of type T
Consumer<T>	T	void	accept	Consumes a value of type T
Function<T,K>	T	K	apply	A function that takes a value of type T as argument and returns a value of type K
BiFunction<T,U,R>	T,U	R	apply	A function that takes two arguments of type T, U ,and returns a value of type K
BiConsumer<T,U>	T, U	void	accept	Consumes values of type T and U
UnaryOperator<T>	T	T	apply	A function that takes a value of type T as argument and returns a value of type T
BinaryOperator<T>	T, T	T	apply	A function that takes two values of type T as argument and returns a value of type T
Predicate<T>	T	boolean	test	A function that takes a value of type T and returns a boolean value.
BiPredicate<T, U>	T, U	boolean	test	A function that takes two arguments of type T and U and returns a boolean value.

In this paradigm it is possible to pass functions as arguments to other functions or return functions as result of a function. Functions that take functions as parameters or that return them as results are called *higher-order* functions. We illustrate the technique with an example.

Write a function that implements the forAll quantifier for an array of integer values. In first-order predicate calculus the forAll quantifier returns true if all the terms in a set of values satisfy a given predicate p, otherwise false.

```
BiFunction<List<Integer>,Predicate<Integer>,Boolean> forAll = (lst,f)->{
for(Integer x:lst) if(!f.test(x)) return false;
return true;
};
```

We often want to find the frequency of occurrence of values in a sequence. We might want to know how often the number 0 occurs, or how many negative values there are, or how many times x occurs. All of these types of query could be simplified by writing a higher-order function that calculates the frequency of terms satisfying a given condition. Then all that is required is a function that defines the given condition. The function freq, defined below, satisfies this requirement and provides a higher-order function that calculates the frequency of values in an integer array satisfying a given predicate defined as a Boolean function on type Integer. This function iterates over the list, lst, adding 1 when a value in the list satisfies the constraint. It returns 0 for an empty list.

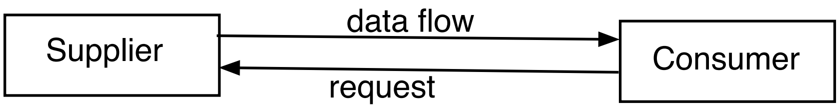
```
BiFunction<List<Integer>, Predicate<Integer>,Integer> freq = (lst,f)->{
   int count = 0;
   for(Integer x : lst) if(f.test(x)) count++;
   return count;
};
```

A code fragment that tests this function is given below.

```
List<Integer> dtlst = new ArrayList<Integer>(Arrays.asList(1,2,2,2,5,2));
assert freq.apply(dtlst,x -> x ==2) == 4;
assert freq.apply(dtlst,x -> x % 2 != 0) == 2;
```

Streams

A stream is a lazy pipeline linking data suppliers with data consumers. Streams are lazy because they only respond to requests on demand from a consumer. They are a special type of `Iterable/Traversable` whose elements are not evaluated until requested by a consumer. It can actively monitor the flow of data and in certain circumstances transform it. A simple model of a stream connecting a supplier with a consumer is:



The *request* originates with the consumer and the *data flows* from the supplier only in response to the request. Hence, the diagram uses two directed edges: the request originating with the consumer and the data originating with the supplier.

There are different types of suppliers: collection containers, functions that supply a stream of data, or even an infinite supplier. The data stored in a given data structure is the source of the stream and the stream in turn provides a channel through which the data flows to its consumer. The stream itself does not generally store data nor should it modify data contained in its source. It simply passes the data to a consumer at its end point.

To work with streams you need, as a minimum, a supplier and a terminal operation that consumes the data supplied through the stream by the data source. All the `Collection` classes have associated streams. To connect a stream to a list data source you just invoke the `stream()` method. The supplier methods are listed in the table below.

Supplier Name	**Argument Type**	**Return Type**	**Semantics**
`generate()`	`Supplier<T> f`	`static<T> Stream<T>`	Returns an infinite stream of values where each value is generated by the Supplier function f.

iterate()	T seed, UnaryOperator <T> f	static<T> Stream<T>	Returns an infinite stream of values obtained by recursive application of the the function generating the sequence seed, f(seed), f(f(seed)), ..
limit()	long maxSize	Stream<T>	Limits the number of elements supplied through a stream to at most maxSize.
of()	T … values	static<T> Stream<T>	Returns a stream whose elements are the specified values

The following code fragment creates a new list of integers and returns a stream that connects to it.

```
List<Integer> lst = new ArrayList<Integer>(Arrays.asList(1,2,2,7,15,20));
Stream<Integer> lstStr = lst.stream();
```

To activate the stream, lstStr, we need to supply a terminal operation or consumer. Four basic consumers are listed in the table below. They are: forEach, allMatch, anyMatch and count.

Consumer Name	**Argument Type**	**Return Type**	**Semantics**
forEach()	Consumer<? Super T>	Void	Executes consumer function for each element supplied by the stream
allMatch()	Predicate<? Super T>	boolean	Returns true if all elements supplied by the stream satisfy the given predicate; false otherwise

anyMatch()	Predicate<? Super T>	boolean	Returns true if any element supplied by the stream satisfies the given predicate; false otherwise
count()		long	Returns the number of elements supplied by the stream

The forEach consumer takes a Consumer function as argument and applies it to each element supplied by the stream. It continuously requests data until the data source terminates its supply of data. For example, to print the list above through a stream we could write:

```
  lst.stream().forEach(x -> System.out.print(x+" "));
```

If you want to write a more complex consumer you can always create a separate Consumer alias and pass it as an argument to forEach. For example the code fragment, given below, prints only the odd numbers supplied by the stream.

```
Consumer<Integer> printOdd = x -> {if(x % 2 != 0) System.out.print(x + " ");};
 lst.stream().forEach(printOdd)
```

The higher order methods allMatch and anyMatch each take a Predicate function as argument and return a Boolean value. In the case of allMatch it returns true if all elements supplied through the stream satisfy the predicate; false otherwise. Method anyMatch also takes a Predicate function as argument and returns true if at least a single value supplied satisfies the predicate. An example of each using list, lst, as a data source for the stream is given below. The first one checks that all values are positive and the second that the list contains at least one even value. The assert statement is used to test the result.

```
List<Integer> lst = new ArrayList<Integer>(Arrays.asList(1,2,2,7,15,20));
assert lst.stream().allMatch(x -> x > 0) == true;
assert lst.stream().anyMatch(x -> x % 2 == 0) == true;
```

Finally, the method count simply returns the number of elements supplied through the stream. We assert that lst.stream().count() == 6. Note that the return type is long.

Streams of data can also be sourced from supplier type functions. The two methods used to do this are called generate and iterate. Both of these data suppliers return potentially infinite streams. In effect this means that they continuously supply values on demand from a consumer function. For example, the following code fragment generates and prints an infinite stream of random numbers in the range 0..9.

```
Stream<Integer> infInt = Stream.generate(()->(int)(Math.random()*10));
infInt.forEach(x -> System.out.printf("%3d",x));
```

Stream.generate takes the Supplier function, () -> (int)(Math.random()*10)), and returns a Stream. This stream called infInt is then activated by the consumer function passed to the forEach method. The supply is infinite because there is no terminating condition: forEach requests forever and generate supplies forever. In effect what you get is an infinite non-terminating supply of random numbers that are forever consumed. To impose a limit on the number of values supplied by the stream we use the limit method. To restrict the stream to a max size of 10 values we could write:

```
 Stream.generate(()->(int)(Math.random()*10))
            .limit(10)
                .forEach(x -> System.out.printf("%3d",x));
```

The higher order method, iterate takes both a seed value and a function that recursively or iteratively returns a sequence of values beginning with the seed value. The sequence is always: seed, f(seed), f(f(seed)), ... Once again the sequence is potentially infinite. The stream Stream.iterate(0, n -> n+2) will generate, when activated, the sequence 0, 2, 4, 6, 8, .., because successive terms are generated by adding two to the previous term. The code fragment below prints the first 6 terms of the sequence 0, 2, 4, 6, 8, 10.

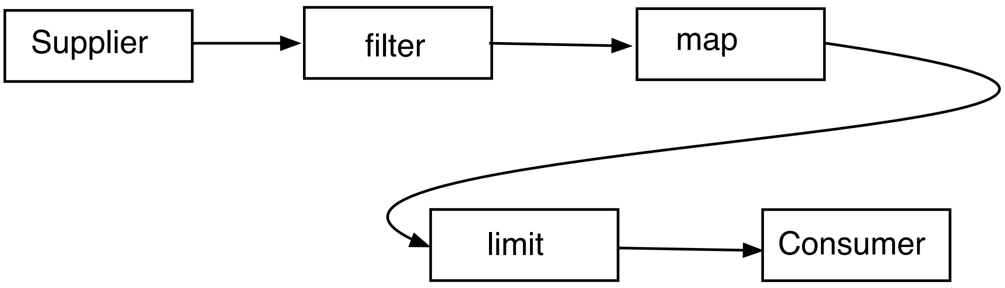
```
Stream.iterate(0,n -> n+2).limit(6).forEach(x -> System.out.print(x+" "));
```

Intermediate or Aggregate Operations

In previous examples we saw how the `limit` operation could control the supply of data between a consumer and a supplier. This is an aggregrate or intermediate operation on the stream linking a supplier with a terminal consumer. The higher order methods `filter` and `map` will play a similar role. Both receive a stream of data from an existing stream and return a stream. The `filter` operation only passes on elements that satisfy its predicate function. The `map` operation transforms all data values by applying its given function to each element before passing it on to the new stream. The code fragment, given below, creates a list of integer values and then uses the stream provided by the list class to apply intermediate operations between the source and the `forEach` consumer. The first one calculates the square of each value and passes it on resulting in output: `1  4  4  49  9  81`. The second one only passes on those elements that are odd numbers resulting in output: `1  7  3  9`. The third one combines both operations by filtering only odd values and then squaring each one in turn before passing it on to the terminal operation `forEach`. The output is: `1  49  9  81`.

```
List<Integer> lst = new ArrayList<Integer>(Arrays.asList(1,2,2,7,3,9));
lst.stream().map(x -> x * x)
                   .forEach(x -> System.out.printf("%3d",x));
System.out.println();
lst.stream().filter(x -> x % 2 != 0)
                   .forEach(x -> System.out.printf("%3d",x));
System.out.println();
lst.stream().filter(x -> x % 2 != 0)
                   .map(x -> x * x)
                      .forEach(x -> System.out.printf("%3d",x));
```

Operations on a stream are chained together where requests begin with a terminal consumer. In the diagram below, the consumer requests data from limit, limit in turn requests data from map, map requests from filter that, finally, contacts the generator that supplies the actual data. Data generated by the supplier flows through the pipe in order one element at the time undergoing filtering and mapping as it goes.



data flow indicated by arrow

An example of an implementation of the given model might be a stream that solves the problem: *generate 50 odd numbers in the range 0 to 99 and print the square of each one*.

```
Stream.generate(() -> (int)(Math.random()*100))
          .filter(x -> x % 2 != 0)
             .map(x -> x*x)
                  .limit(50)
                  .forEach(x->System.out.print(x+" "));
```

Note that the `limit` operation comes just before the consumer `forEach`. If you place it before the `filter` then it will count elements that may fail the test and, hence, are not passed to the consumer.

Table of Stream Intermediate Operation Methods

Name	**Argument Type**	**Return Type**	**Semantics**
filter()	Predicate<? Super T>	Stream<T>	Returns a stream of those elements it receives that satisfy the given predicate.
map()	Function<? Super T, ? extends R>	<R> Stream<R>	Returns a stream of the elements it receives mapped by the given function
flatMap()	Function<? Super T, ? extends Stream<? extends R>>	<R> Stream<R>	This operation applies a function that itself returns a stream to each element of this stream.

distinct()		Stream<T>	Returns a stream consisting of the distinct elements in this stream. The definition of equality is based on the equals method implemented for the given class.
sorted()		Stream<T>	Returns a stream of the elements of the consumed stream, sorted by natural order (based on implementation of compareTo)
sorted()	Comparator(? super T compare)	Stream<T>	Returns a stream of the elements of the consumed stream, sorted by order defined by the Comparator (based on implementation of compare).

Reduction Operations

A reduction operation is a terminal operation that takes a stream of values and accumulates a result by recursively applying a BinaryOperator function to the sequence of elements. We have seen an example previously when we used the count operation. More common examples where reductions are relevant for streams are: calculating the sum of the values, finding the max or min value or calculating the frequency of occurrence of a given value. In functional programming this type of operation is called *folding*. There are two general types: reduce and collect. A reduce operation takes two arguments: an identity value for the operator and a BinaryOperator function. The identity value for addition is 0 because a + 0 == a and the identity value for multiplication is 1 because a*1 == a. A BinaryOperator function takes two arguments of type T and returns a value of type T. Given the list of values, lst, we define reduction operations that: calculate the summation of the values in the list; the sum of the even values in the list; the product of the odd elements in the list; the frequency of even elements in the list and the largest value in the list. In each case the argument is an identity value for the operator and a BinaryOperator function. In the case of the frequency, freq, it is 0 because if no terms match the condition then the default answer is 0; in the case of the maximum value, largest, it is the smallest integer value because all values are greater than or equal to this value. The reduction operations that solve each of these in turn are:

```
lst.stream().reduce(0, (sum,x) -> sum+x);

lst.stream().reduce(0,(sum,x) -> x % 2 == 0 ? sum + x : sum);

lst.stream().reduce(1,(prod,x) -> x % 2 != 0 ? prod * x : prod);

lst.stream().reduce(0, (x,y) -> {if(y % 2 == 0) return x + 1; else return x ;});

lst.stream().reduce(Integer.MIN_VALUE,(x,y) -> {if (x > y) return x; else return
y;});
```

Note that in cases where the list is empty the value returned is the identity value in each case. The reduction operation must be associative. This means that the order of applying the binary function is not important. In the case of addition this is the case because (a+b)+c == a+(b+c). In functional terms what we get is a *foldLeft* operation. To explain the semantics of reduce we take the case: lst.stream().reduce(0, (sum,x) -> sum+x). The identity value, 0, is used to initialize. The reduction, in this case, equates to the expression

```
    lst.stream().reduce(0, (sum,x) -> sum+x)
 =  (((((0+1)+2)+2)+7)+3)+9.
 =  ((((1+2)+2)+7)+3)+9
 =  (((3+2)+7)+3)+9
 =  ((5+7)+3)+9
 =  (12+3)+9
 =  15+9
 =  24
```

A similar derivation applies for each of the other reductions listed above. The code fragment below uses an assert statement to check the answers in each case.

```
List<Integer> lst = new ArrayList<Integer>(Arrays.asList(1,2,2,7,3,9));

Integer s1 = lst.stream().reduce(0, (sum,x) -> sum+x);
assert s1 == 24;

Integer sEven = lst.stream().reduce(0,(sum,x) -> x % 2 == 0 ? sum + x : sum + 0);
assert sEven == 4;

Integer pOdd = lst.stream().reduce(1,(prod,x) -> x % 2 != 0 ? prod * x : prod);
assert pOdd == 189;

Integer freq = lst.stream().reduce(0, (x,y) -> {if(y % 2 == 0) return x + 1; else
return x + 0;});
assert freq == 2;

Integer largest = lst.stream().reduce(Integer.MIN_VALUE,(x,y) -> {if (x > y)
return x; else return y;});
assert largest == 9;
```

Collecting Data

Streams create pipelines connecting a data source with a consumer operation. The examples of consumers we have studied reduce the data to a single value or consume it by displaying it using `forEach`. But there are many situations where we want to extract multiple items from a stream. If a data source provides a stream of integer values we might want to extract all the even values from the stream or all multiples of 10. In the case of a stream of books we might want to extract a list of authors, or a list of titles.

The terminal operation used to create collection containers is called `collect`. It provides a mutable state based reduction operation because it collects data in a data structure. There are two implementations provided. The first form takes three arguments: a supplier function to construct new instances of the result container, an accumulator function to incorporate an input element into a result container, and a combining function to merge the contents of one result container into another. Its signature is:

```
<R> R collect(Supplier<R> supplier,
            BiConsumer<R, ? super T> accumulator,
        BiConsumer<R, R> combiner);
```

The second form is much simpler because it takes a single argument supplied by the Collector factory. The signature is:

```
<R,A> R collect(Collector<? super T,A,R> collector)
```

This second form provides all the functionality we require to consume stream output by storing it in collection containers of our choice. The Collector library provides methods that implement many different types of mutable reductions. The following examples will illustrate the main features of this library.

The static method toCollection(Supplier<C> collection) returns a collector that accumulates the input values from this stream into a new Collection, in encounter order. To illustrate its use we create a stream of 50 random numbers and collect them in an ArrayList. The collect method writes the data, in encounter order, provided through the stream to the data structure provided by the Collector.toCollection method. A constructor reference is used to define the type of data structure to create. The code is:

```
List<Integer> ls = Stream.generate(() -> (int)(Math.random()*50))
                                    .limit(50)

 .collect(Collectors.toCollection(ArrayList::new));
```

We now use the data structure ls to select sub-sets of this data. For example, to collect all the odd values in the list in a set we could use:

```
TreeSet<Integer> oddSet = ls.stream()
                                              .distinct().filter(x -> x % 2 != 0)

 .collect(Collectors.toCollection(TreeSet::new));
```

To create a LinkedList of all multiple of 10 squared from the list ls we write:

```
LinkedList<Integer> lst = ls.stream().filter(x -> x % 10 == 0)
```

```
                                        .map(x -> x * x)

  .collect(Collectors.toCollection(LinkedList::new));
```

Programming in a Declarative Style

Streams combined with functions make it possible to define *what* computation is to be performed rather than *how* it is performed. The implementation of the complete stream is managed internally and allows for different possible implementations. Each stream typically has a source, a consumer and possibly aggregrate operations. There is always a single source and a single consumer. Aggregrate operations may be implemented as a single stream or a parallel stream. The key point here is that the functionality of the stream is preserved and delivered through its interface higher-order methods that take functions as arguments. **The *how* is hidden from the user. This makes it possible for the programmer to state *what* they want as opposed to *how* the result is delivered.** Streams combined with functions is a paradigm shift that moves away from writing sequential state based programs to writing code in a declarative style. This means stating *what* the code should deliver, not *how* it is to be delivered. It does not mean moving away from the object-oriented paradigm. The goal is to keep this model because it is *so useful* (Odersky) and to compliment it by shifting to writing methods in a functional or declarative style.

To illustrate this declarative style we look at re-writing some functions written in an imperative style using both functions and streams. Re-writing functions in this style is often referred to as *re-factoring* code.

The higher-order function forAll takes both a list and a predicate as arguments and returns true only if all elements in the list satisfy the given predicate. It is written using a for loop. This can be written in a declarative style using a stream and the terminal operation allMatch that returns true if all elements match the given predicate. Both versions are listed below.

```
BiFunction<List<Integer>,Predicate<Integer>,Boolean> forAll = (lst,f)->{
  for(Integer x:lst) if(!f.test(x)) return false;
  return true;
};
```

```
 BiFunction<List<Integer>,Predicate<Integer>,Boolean> forAll = (lst,f)->{
   return lst.stream().allMatch(f);
 };
```

The higher-order function map takes both a list and a function as argument and returns a new list got by mapping all elements in the given list to the new list by applying the function to each element in sequence. This function can be re-written in a declarative style using a stream with an intermediate mapping function and the terminal operation collect. Again both functions are listed below.

```
 BiFunction<List<Integer>, Function<Integer,Integer>, List<Integer>> map = (ls,
f) ->{
     List<Integer> lst = new ArrayList<Integer>();
     for(Integer x : ls) lst.add(f.apply(x));
     return lst;
 };

BiFunction<List<Integer>, Function<Integer,Integer>, List<Integer>> map = (ls, f)
->{
   return  ls.stream().map(f).collect(Collectors.toCollection(ArrayList::new));
 };
```

Summary

a) Functions in general provide implementations of lambda expressions and provide a mapping from inputs to outputs. They are first class objects.
b) Higher-order functions and higher-order methods provide a mechanism to pass functions as arguments. This has implications for writing interfaces to classes.
c) Composition supports chaining of functions and, hence, allows for declarative programming.
d) Streams provide a structure that supports lazy evaluation and allow multiple functions to be chained together to compute results. In some sense streams provide *lazy composition* of functions because multiple stream methods can be chained together where each method takes a function as argument.
e) Streams combined with functions make it possible to define *what* computation is to be performed rather than *how* it is performed. The implementation of the complete stream is managed internally and allows for different possible implementations. The key point here is that the functionality of the stream is preserved and delivered through its interface higher-order methods that take

functions as arguments. The *how* is hidden from the user. This makes it possible for the programmer to state *what* they want as opposed to *how* the result is delivered.

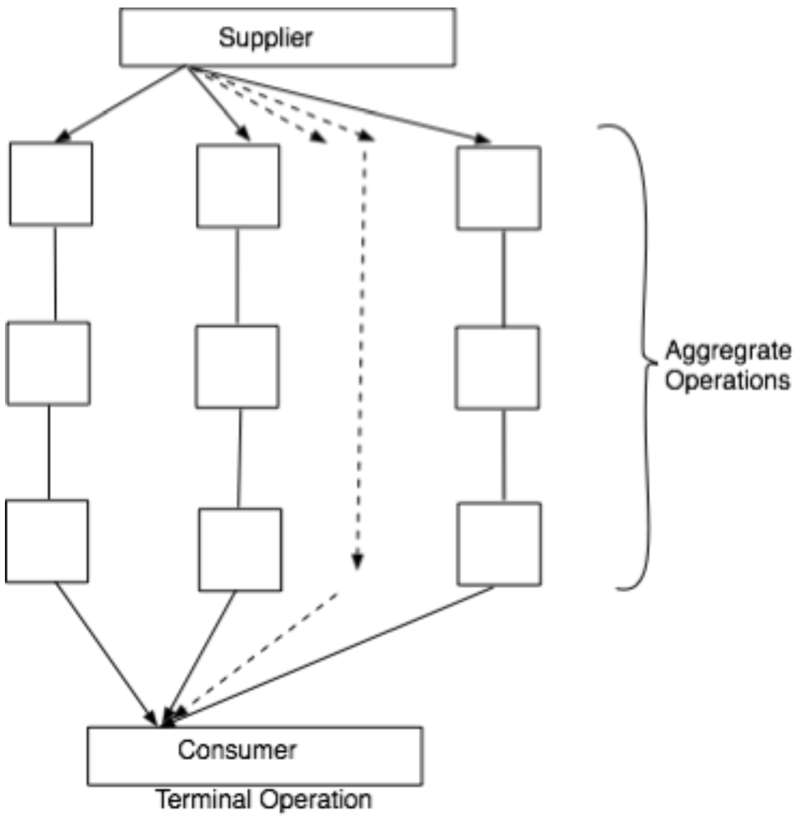
f) Streams combined with functions meet Odersky's requirement that what we need is the transformation of immutable values instead of stepwise modifications of mutable state. The emphasis on immutable state also meets Goetz's demand that we should *adopt the orientation that immutable state is better than mutable state*.

g) Streams combined with functions is a paradigm shift that moves away from writing sequential state based programs to writing code in a declarative style. This means stating *what* the code should deliver, not *how* it is to be delivered. It does not mean moving away from the object-oriented paradigm. The goal is to keep this model because it is *so useful* (Odersky) and to compliment it by shifting to writing methods in a functional or declarative style.

We now turn our attention to parallel streams.

Parallel Streams

A parallel stream is a stream with a single source and a single consumer where *aggregrate* operations are executed in parallel. The source data set is divided and distributed among the different streams and passed thorugh all intermediate operations to be consumed by a single terminal consumer. The diagram below provides a model for parallel streams.



The parallel stream example below has a single source and a single consumer and no aggregrate operations. The data is divided between the parallel streams and received by the `forEach` consumer. We don't know how many streams are actually used and we have no guarantees about the order in which the data elements will be consumed by `forEach`. In fact a sample execution of the code shows that the order of the data in original list is not preserved.

```
List<Integer> lst = new
        ArrayList<>(Arrays.asList(2,3,4,5,6,7,8,9,1,2,3,4));
 System.out.println(lst);
 lst.parallelStream().forEach(x->System.out.print(x+" "));
```

```
Output: 1 3 2 8 7 6 3 5 9 2 4 4
```

For a second example we calculate the frequency of even values in lst using a parallel stream and a filter method as follows:

```
long freq = lst2.parallelStream().filter(x -> x % 2 == 0).count();
```

It is also possible to perform reductions on parallel streams. To calculate the sum of the elements in lst using a parallel stream we write:

```
lst.parallelStream().reduce(0,(s,x)->s+x)
```

But how is this thread safe because the variable s appears to be shared by multiple threads supplying data to the reduction operation and therefore some form of synchronization should be required? The reason why synchronization is not necessary only becomes clear when we expand the reduction to the following form.

```
lst.parallelStream()
            .reduce(
                        0,
                         (s,x) -> s+x,
                         (sum1,sum2) -> sum1 + sum2
             )
```

Here the third line, (sum1,sum2) -> sum1 + sum2, actually combines partial sums in sequence. To fully explain this we need to go back to our study of the divide and conquer threadpool ForkJoinPool. Parallel streams are implemented by a ForkJoinPool and use its divide and conquer algorithm to split the data sequence into separate streams. In fact what is used is the static commonPool from the ForkJoinPool class. The previous reduction through the use of a parallel stream might be implemented as given below. The choice of BlockSize of 5000 is arbitrary. The case of partial sums is highlighted in bold. This partial summation corresponds to the line (sum1,sum2) -> sum1 + sum2 above.

```
class Sum extends RecursiveTask<Long> {
   static final int BaseBlockSize = 5000;
   int lb, ub;
   int[] data;
   Sum(int[] dt, int a, int b){
      data = dt;
      lb = a; ub = b;
   }
protected Long compute() {
      if(ub - lb <= BaseBlockSize) {
         long sum = 0;
         for(int i=lb; i < ub; ++i)
            sum = sum + data[i];
         return sum;
      } else else {
         int mid = lb + (ub - lb) / 2;
         Sum left  = new Sum(data, lb, mid);
         Sum right = new Sum(data, mid, ub);
         left.fork();
           right.fork();
         long rSum = right.join();
         long lSum  = left.join();
         return lSum + rSum;
      }
   }
```

Note: We can show the particular work stealing threads executing by printing the names of the threads actually executing when the parallel processing is taking place. The following code snippet creates a parallel stream in response to consumer forEach and the consumer simply prints the names of the executing threads.

```
List<Integer>lst=new ArrayList<>(Arrays.asList(2,3,4,5,6,7,8,9,1,2,3,4));
lst.parallelStream()
     .forEach(
          x->System.out.println(Thread.currentThread().getName());
```

```
    );
```

The output from a sample run was:

```
ForkJoinPool.commonPool-worker-4
ForkJoinPool.commonPool-worker-3
ForkJoinPool.commonPool-worker-3
ForkJoinPool.commonPool-worker-6
ForkJoinPool.commonPool-worker-1
ForkJoinPool.commonPool-worker-7
ForkJoinPool.commonPool-worker-2
ForkJoinPool.commonPool-worker-5
main
ForkJoinPool.commonPool-worker-6
ForkJoinPool.commonPool-worker-3
ForkJoinPool.commonPool-worker-4
```

End note

Note from Java Website

Consider again the following example (which is described in the section Reduction) that groups members by gender. This example invokes the `collect` operation, which reduces the collection `roster` into a `Map`:

```
Map<Person.Sex, List<Person>> byGender =
   roster
      .stream()
      .collect(
         Collectors.groupingBy(Person::getGender));
```

The following is the parallel equivalent:

```
ConcurrentMap<Person.Sex, List<Person>> byGender =
   roster
      .parallelStream()
      .collect(Collectors.groupingByConcurrent(Person::getGender));
```

This is called a *concurrent reduction*. The Java runtime performs a concurrent reduction if all of the the following are true for a particular pipeline that contains the `collect` operation:

a) The stream is parallel.
b) The parameter of the `collect` operation, the collector, has the characteristic `Collector.Characteristics.CONCURRENT`. To determine the characteristics of a collector, invoke the `Collector.characteristics` method.
c) Either the stream is unordered, or the collector has the characteristic `Collector.Characteristics.UNORDERED`. To ensure that the stream is unordered, invoke the `BaseStream.unordered` operation.

Note: This example returns an instance of `ConcurrentMap` instead of `Map` and invokes the `groupingByConcurrent` operation instead of `groupingBy`. (See the section Concurrent Collections for more information about `ConcurrentMap`.) Unlike the operation `groupingByConcurrent`, the operation `groupingBy` performs poorly with parallel streams. (This is because it operates by merging two maps by key, which is computationally expensive.) Similarly, the operation `Collectors.toConcurrentMap` performs better with parallel streams than the operation `Collectors.toMap`.

End Note

Parallelising Sequential Algorithms with Parallel Streams

Parallel streams can be used to write multi-threaded solutions to problems in a declarative style. We illustrate this approach by taking a number of multi-threaded algorithms and re-write them in this style.

The following program uses 4 threads to calculate the sum of the even elements in a large integer array. The coding involves writing a separate thread class called `Summer`, distributing the data fairly between the actual threads, waiting for the threads to complete and then extracting the partial sums in sequence by invoking the `result` method on each thread. This is quite a lot of work, or what they call *boiler-plate* in the industry, for what is in fact a simple task.

```
public static void main(String[] args) {
   Integer data[] = new Integer[1000000];
   for(int j = 0; j < 1000; j++) data[j] = j+1;
   Summer th[] = new Summer[4];
   for(int j = 0; j < 4; j++){
     th[j] = new Summer(data,j*250000,(j+1)*250000);
     th[j].start();
   }
   for(Summer t : th){
     try{t.join();}
     catch(InterruptedException e){}
   }
   Integer k = 0;
   for(int j = 0; j < 4; j++) k += th[j].result();
   System.out.println("Sum = "+k);
}

class Summer extends Thread{
  Integer[] dt;
  int lb, ub;
  Integer sum;
  Summer(Integer[] f, int l, int u){
    dt = f; lb = l; ub = u;
  }
  public void run(){
    sum = 0;
    for(int j = lb; j < ub; j++){
      if(dt[j] % 2 == 0) sum += dt[j];
    }
  }
  public Integer result(){return sum;}
}
```

This problem can be re-written using parallel streams. We first create a list of the integer values in the data array and then use a parallelStream combined with a filter and a reduction operation to calculate the result. The parallelStream uses a forkjoin pool to divide the data list into segment sizes that correspond with the number of processors on the machine executing the code. In effect it does the division automatically.

```
  List<Integer> dtLst = new ArrayList<Integer>();
  for(Integer x : data) dtLst.add(x);
```

```
k = dtLst.parallelStream().filter(x -> x%2 == 0 ).reduce(0,(sum,x)->sum+x);
System.out.println(k);
```

The use of a parallel stream allows the programmer to state *what they want rather than how to deliver it*.

For a second example we take a collector problem. The program listed below creates a data list of 100000 integer values and uses four `Collector` threads to extract all odd numbers from this data list and return a list containing the square of each odd number found. Each thread creates its own local list and the method `result()` returns a reference to this list that is used by main to construct the final list.

Note: the `assert` statement is used to check the result of the thread calculation. To turn assertion checking on use *java –ea ParallelStreamCollector* when executing the program. The actual file with appropriate imports is available on Moodle. **End Note**

```
public class ParallelStreamCollector{
    public static void main(String args[]){
        List<Integer> data = new ArrayList<>(100000);
        for(int j = 0; j < 100000; j++) data.add((int)(Math.random()*100000));
        Collector th[] = new Collector[4];
        for(int j = 0; j < 4; j++){
          th[j] = new Collector(data,j*25000,(j+1)*25000);
          th[j].start();
        }
        for(Collector t : th){
          try{t.join();}
          catch(InterruptedException e){}
        }
        //Now extract data
        List<Integer> lst = new ArrayList<Integer>();
        for(Collector t : th) lst.addAll(t.result());
        assert lst.size() == data.stream().filter(x -> x % 2 != 0).count();
    }
}
class Collector extends Thread{
    List<Integer> dt;
    int lb, ub;
    List<Integer> lst = new ArrayList<>();
    Collector(List<Integer> f, int l, int u){
        dt = f; lb = l; ub = u;
    }
    public void run(){
      for(int j = lb; j < ub; j++){
        Integer k = dt.get(j);
        if(k % 2 != 0) lst.add(k*k);
      }
    }
    public List<Integer> result(){return lst;}
}
```

Again this code can be written using parallel streams combined with a filter that excludes even values and a map that squares the odd values passedto it from the filter. The solution is:

```
    ArrayList<Integer> lst1 = data.parallelStream()
                             .filter(x -> x%2 != 0 )
                              .map(x -> x * x)
                               .collect(Collectors.toCollection(ArrayList::new));

    assert lst1.size() == data.stream().filter(x -> x % 2 != 0).count();
```

Functional Interfaces and ParallelStreams

Streams combined with functions is a paradigm shift that moves away from writing sequential state based programs to writing code in a declarative style. This means stating *what* the code should deliver, not *how* it is to be delivered. From an object-oriented perspective it means that we can write methods in a functional or declarative style. There are two issues here: one, we can write higher-order methods that take functions as arguments; two, we can write the code for the methods using streams or parallel streams. In what follows we discuss both of these approaches by taking a simple example to illustrate the technique.

A class IntManager uses an ArrayList to store a collection of Integer type. This class is Thread safe. The class is written in an imperative style with standard methods that do not use higher-order methods. For the purposes of the discussion we call this class IntManagerImpOld. The class has two constructors, an add method, three methods that modify all the data in the collection – squareAll(), incAll(), doubleAll(); two methods that test all the elements in the collection – allEven(), allOdd(); and two methods that retrieve elements from the collection – getOdd() and getNegSquared(). All methods are coded in an imperative style.

```
IntManagerImpOld{
 private ArrayList<Integer> lst;
 IntManagerImpOld(List<Integer> ls){lst = new ArrayList<>(ls);}
 IntManagerImpOld(){lst = new ArrayList<>();}
 synchronized void add(Integer x){lst.add(x);}
 synchronized void squareAll(){
  ArrayList<Integer> ls = new ArrayList<>();
  for(Integer x : lst) ls.add(x*x);
  lst = ls;
 }
 synchronized void incAll(){
  ArrayList<Integer> ls = new ArrayList<>();
  for(Integer x : lst) ls.add(x+1);
  lst = ls;
 }
 synchronized void doubleAll(){
  ArrayList<Integer> ls = new ArrayList<>();
  for(Integer x : lst) ls.add(2*x);
  lst = ls;
 }
 synchronized boolean allEven(){
   for(Integer x : lst) if(x % 2 != 0) return false;
   return true;
 }
 synchronized boolean allOdd(){
   for(Integer x : lst) if(x % 2 == 0) return false;
   return true;
 }
 synchronized List<Integer> getOdd(){
  List<Integer> ls = new ArrayList<>();
   for(Integer x : lst) if(x % 2 != 0) ls.add(x);
   return ls;
 }
 synchronized List<Integer> getNegSquared(){
  List<Integer> ls = new ArrayList<>();
   for(Integer x : lst) if(x < 0) ls.add(x*x);
   return ls;
 }
}
```

The public interface to this class can be simplified by replacing the existing methods with methods that take functions as arguments. In the version listed below, `IntManagerImp`, the method `map` takes a function as argument and uses it to modify all elements in `lst`; the method `forAll` takes a predicate as argument and returns true only if all the elements in `lst` satisfies it; the method `getMap` takes both a predicate and a function as arguments and returns a list of those values that satisfy the predicate `p` mapped by the function `f`. Each of these three methods is written in an imperative style. The original methods can now be defined by passing appropriate lambda expressions to these methods as arguments.

```
public class IntManagerImp{
 private ArrayList<Integer> lst;
 IntManagerImpOld(List<Integer> ls){lst = new ArrayList<>(ls);}
 IntManagerImpOld(){lst = new ArrayList<>();}
 synchronized void add(Integer x){lst.add(x);}
 synchronized void map(Function<Integer,Integer> f){
    ArrayList<Integer> ls = new ArrayList<>();
    for(Integer x : lst) ls.add(f.apply(x));
    lst = ls;
  }
  synchronized boolean forAll(Predicate<Integer> p){
    for(Integer x : lst) if(!p.test(x)) return false;
    return true;
  }
  synchronized List<Integer> getMap(Predicate<Integer> p,
                                          Function<Integer,Integer> f){
    List<Integer> ls = new ArrayList<>();
    for(Integer x : lst)
       if(p.test(x)) ls.add(f.apply(x));
     return ls;
  }
}
```

The final version listed below replaces the sequential imperative code for the methods with parallel streams. This optimizes the performance of the methods by using the full processing power of the underlying multi-core architecture. The threads are hidden from the user meeting the requirement that he *how* is hidden. This makes it possible for the programmer to state *what* they want as opposed to *how* the result is delivered and, at the same time, optimize the performance of their code.

```
class IntManagerPar{
  private ArrayList<Integer> lst;
  IntManagerPar(){lst = new ArrayList<>();}
  IntManagerPar(List<Integer> ls){lst = new ArrayList<>(ls);}
  synchronized void add(Integer x){lst.add(x);}
  synchronized void map(Function<Integer,Integer> f){
    lst = lst.parallelStream()
                   .map(f)
                    .collect(Collectors.toCollection(ArrayList::new));
  }
  synchronized boolean forAll(Predicate<Integer> p){
    return lst.parallelStream().allMatch(p);
  }
  synchronized List<Integer> getMap(Predicate<Integer> p,
                                          Function<Integer,Integer> f){
```

```
   return lst.parallelStream()
            .filter(p).map(f)
             .collect(Collectors.toCollection(ArrayList::new));
 }
}
```

Conclusion

The key points to remember about parallel streams are:

- Stream can be executed in parallel
- Provides parallel data processing
- Uses ForkJoinPool.commonPool to provide it
- Only intermediate or aggregrate operations on stream are executed in parallel
- Does not guarantee order preservation

Critique

- All parallel streams use the same forkjoin pool
- Can cause problems if some tasks give rise to delays. Consider case of parallel stream processing data from a file or a socket. This could greatly impede the performance of your streams.
- It must be possible to divide the source data easily to distribute it between the streams. For example, a linked list would not be suitable for use with parallel streams.

Message Passing with Actors

Programming constructs, such as semaphores and monitors, are based on shared-state synchronization. In effect, synchronization turns parts of a program into sequential code because only one thread at a time can access the global, or shared, state. Indeed, if control structures through a program rely on globally visible state, there is no way around serialized access to that state without risking incorrect behavior. In pure functional object-oriented programming you only have immutable objects. You cannot modify the state of an object once you create it. Although Java has immutable classes such as String, Class, and Integer, most classes are mutable objects. You create an instance and invoke methods to change the state of the object. To explain why mutable objects are not desirable we consider the following Counter class

```
public class Counter {
  private int count;
  synchronized public int getCount() { return count; }
  synchronized public void setCount(int value) { count = value; }
}
```

In order to protect against multiple threads accessing the count two synchronized methods are used. Unfortunately, this is not adequate. The following code is very problematic:

```
int currentValue = counter.getCount();
counter.setCount(currentValue + 100);
```

Suppose, an instance of the `Counter` is shared by multiple threads and each thread is performing an operation then value of count is totally unpredictable. Even though both the methods of the `Counter` are synchronized, between the call to `getCount()` and the call to `setCount()` another thread may gain the monitor or lock and modify the value. We have to place the two calls within a proper synchronized block for the given code sequence to ensure consistent views of the shared data.

This problem disappears with immutable objects because there is no state to change and no contention to worry about. If you want to make a change, you simply create another instance of the immutable object. Immutable objects offer quite a few advantages:

1. They are inherently thread safe. Since you can't modify their state, you can freely pass them between threads without fear of contention.
2. There is no need to synchronize them.
3. They can be shared and reused across the application. This can help ease the burden on resources in your application.
4. They are less error prone. Since you do not arbitrarily modify the state of objects, you will have fewer errors to deal with.
5. It is easier to verify the correctness of your code with immutable objects than with mutable objects.

But is it possible to write all programs so that all classes are immutable? The answer is no and all the modern functional languages, OCaml, Erlang, Scala, F# and Haskell support the introduction of state. Therefore, the question becomes: is it possible to allow state change and avoid synchronization and locking? The answer to this question would appear to be yes.

An alternative model to the one we have studied is based on the idea of passing immutable messages between threads. This provides an alternative model for supporting communication among cooperating threads. Threads receive immutable messages that are then owned by them exclusively. This means that the receiving threads can modify the state of these messages in a local context without recourse to the need for synchronization.

There are two important categories of systems based on message passing. In channel-based systems, messages are sent to channels (or ports) that processes can share. Several processes can then receive messages from the same, shared channels. Examples of channel-based systems are MPI(Message Passing Interface, 1994) and systems based on the CSP (Communicating Sequential Processes, Hoare 1985) paradigm. An example, of a modern programming language using named channels is the Google language Go.

Systems based on actors (or agents, or Erlang-style processes) are in the second category of message-passing concurrency. In these systems, messages are sent directly to actors; it is not necessary to create intermediary channels between processes.

An important advantage of message passing over shared-state concurrency is that it makes it easier to avoid data races. If processes communicate only by passing messages, and those messages are immutable, then race conditions are avoided by design. Moreover, anecdotal evidence suggests that this approach in practice also reduces the risk of deadlock. A potential disadvantage of message passing is that the communication overhead may be high. To communicate, processes have to create and send messages, and these messages are often buffered in queues before they can be received to support asynchronous communication. In contrast, shared-state concurrency enables direct access to shared memory, as long as it is properly synchronized.

A key contribution of the actor model is to define control structures in a way that minimizes reliance on global program state. Instead, all the state or knowledge needed to make control flow decisions are co-located with the objects that make those decisions, i.e the actors. Such objects, in turn, direct control flow only based on data locally visible to them. That principle of locality renders data flow and control flow in a program inseparable, reducing the requirement for synchronization. That, in turn, maximizes the potential for concurrency. The main mechanism for unifying control flow and data flow is a special abstraction, the actor, and the message-based communication that takes place between actors. An actor is any object with the capability to exchange messages with other actors. In this concurrent programming model, actors communicate solely by passing messages to each other.

Note on Scala. The programming language Scala was designed by Martin Odersky at the Programming Methods Laboratory of EPFL and is currently funded by the European Research Council. The design of Scala was influenced by the programming languages: Eiffel, Java, OCaml and Haskell. A stable version was released in April 2012. Scala runs on the Java platform, on the Android operating system and there is also a dot Net version. Scala is a functional object oriented language that provides concurrency support through what it terms an actor based model. The two main building blocks for programs in Scala are functions and the data structure, `List`. **End note**

Scala Actors

An actor in Scala provides an event-based lightweight thread. To create an actor extend the Actor class. If you want to send a message to an actor, use the !() method. To receive a message from an actor, use the receive() method. The receive() method typically uses pattern matching to process the received message. Scala's concurrency model depends on honoring immutability. Scala expects you to pass immutable objects as messages between actors. The following examples illustrate how to write actors. An actor template is:

```
class <name> extends Actor{
   def act(){
     //code for actor here
   }
}
```

Example 1

A simple actor that prints the sum of the first n integers is given.

```
import scala.actors._
import Actor._
class SumN (n : Int) extends Actor{
   def act(){
      var k = 0; var sum = 0
      while(k < n){
        sum = sum + (k+1)
        k = k + 1
       }
       println(sum)
   }
}
def main(args: Array[String]){
   val a1 = new SumN(100)
   a1.start()
}
```

Example 2

An actor that prints the numbers 0…n is given below.

```
 import scala.actors._
 import Actor._
 class Act1 extends Actor{
    def act(){
       while(true)
  receive{
   case (n : Int) =>{
       println()
       for(i <- 0 to n) print(i+" ")
  }
           case "quit" => exit
}
    }
}
object Actor1Test {
   def main(args: Array[String]){
 val a1 = new Act1()
 a1.start()
 a1 ! 10
 a1 ! 5
 a1 ! "quit"
  }
}
```

Example 3

An actor that checks if a number is prime. It listens for a number, checks if it is prime and sends result back to sender.

```
class Act2 extends Actor{
    def isPrime(k : Int) : Boolean = {
 var x : Int = 2
 while(x < k && k % x != 0){x = x + 1}
 if(x < k) false
else true
    }
  def act(){
  while(true)
   receive{
      case (sender : Actor, n : Int) =>
          sender ! (isPrime(n))
      case "quit" => exit
  }
   }
}
object Actor2Test {
   def main(args: Array[String]){
        val a2 = new Act2()
        a2.start()
        a2 ! (self,7)
        a2 ! (self,128)
        for(i <- 0 to 1)
receive{case msg => println(msg)}
         a2!"quit"
    }
}
```

Example 4

This example illustrates how to write a parallel version of a sequential program. The parallel version will utilize two cores on a machine.

```
object SumTest{
     def sum(l : List[Int] : Int =
  if(l.isEmpty) 0
   else l.head + sum(l.tail)

     def main(args : Array[String]{
  val list = List(1,2,3,4,5,6,8,9,12)
  val k = sum(list)
  println("Sum = "+k)
```

```
    }
}
```

The concurrent solution is given below.

```
import scala.actors._
import Actor._
class Summer extends Actor{
    def sum(l :List[Int]) : Int =
  if(l.isEmpty) 0
  else l.head + sum(l.tail)

    def act(){
        while(true)
           receive{
    case (sender : Actor, k : List[Int]) =>
  sender ! (sum(k))
    case "quit" => exit
  }
    }
}

object SumActorsTest {
    def main(args: Array[String]){
val list = List(1,2,3,4,5,6,7,8)
val s1 : Summer = new Summer
val s2 : Summer = new Summer
s1.start; s2.start
s1 ! (self,list.take(list.length/2))
s2 ! (self,list.drop(list.length/2))
var sum : Int = 0
for(i <- 0 to 1)
   receive{case x : Int => sum = sum + x}
 println("Sum = "+sum)
 s1 ! "quit"; s2 ! "quit"
   }
}
```

Example 5

Again this example takes a sequential linear search and converts it into a parallel solution.

```
object LinearSearctTest {
    def find(x:Int, l:List[Int]) : Boolean =
 if(l == Nil) false
 else if (l.head == x)true
else find(x,l.tail)

    def main(args: Array[String]){
 var list = List[Int]()
 for(j <- 0 to 1000)
   list = list.+:((Math.random*100).toInt)
  println(list.toString())
  val x : Int = (Math.random*100).toInt
  val found = find(x,list)
  println(x + " found = " + found)
    }
}
```

Actor Solution

```
import scala.actors._
import Actor._
class Searcher extends Actor{
     def find(x:Int, l:List[Int]) : Boolean =
          if(l == Nil) false
  else if (l.head == x)true
  else find(x,l.tail)
      def act(){
         while(true)
             receive{
         case (sender : Actor, k : List[Int], x : Int) =>
            sender ! (find(x, k))
     case "quit" => exit
    }
      }
  }
```

```
object LinearSearctTest {
    def main(args: Array[String]){
       var list = List[Int]()
       for(j <- 0 to 1000)
 list = list.+:((Math.random*100).toInt)
        println(list.toString())
        val x : Int = (Math.random*100).toInt
         val s1 : Searcher = new Searcher
        val s2 : Searcher = new Searcher
        s1.start; s2.start
        s1 ! (self,list.take(list.length/2),x)
        s2 ! (self,list.drop(list.length/2,x))
        var found : Boolean = false
        for(i <- 0 to 1)
      receive{case b : Boolean  => found = found || b}
         println(x + " found = " + found)
         s1 ! "quit"; s2 ! "quit"
}
```

Appendix

Scala Programming Language

Scala was designed by Martin Odersky at the Programming Methods Laboratory of EPFL and is currently funded by the European Research Council. The design of Scala was influenced by the programming languages: Eiffel, Java, OCaml and Haskell. A stable version was released in April 2012. Scala runs on the Java platform, on the Android operating system and there is also a dot Net version. Scala is a functional object oriented language that provides concurrency support through what it terms an actor based model. The two main building blocks for programs in Scala are functions and the data structure, List.

A simple program that implements three functions and then evaluates them is given below:

```
object Mathexp1 {
  def square(x : Int) : Int = x * x;
  def even(x : Int): Boolean = if(x % 2 == 0) true else false;
  def abs(x : Int): Int = if(x >= 0) x else -x;
  def main(args: Array[String]){
     val x = square(9)
     val y = even(2)
     println(x)
     println(y)
     val k = square( abs(-2))
     println(k)
     val t = even(square(8))
     println(t)
  }
}
```

Data Structure List

Lists are like arrays. A list containing the numbers `1,2,3,4,5` is written as `List(1,2,3,4,5)`. Like arrays, lists are homogeneous, i.e., the elements of a list all have the same type. However, lists differ from arrays in two important ways. Firstly, lists are immutable. This means that it is not possible to change or modify elements in a list by assignment. Secondly, lists have a recursive structure, whereas arrays are flat. In fact, lists are constructed as linked lists.

Constructing lists

The simplest way to create a list is to create a list literal. Examples are:

```
val ls0 = List(1,3,4,5,6)
val ls1 = List("Mary","Carmel", "Rory", "Donal")
```

We can also construct list literals by using the two fundamental building blocks, Nil and :: (pronounced *cons*). Nil represents the empty list and the infix operator, ::, expresses list extension at the front. That is, x :: ls represents a list whose first element is x, followed by the elements of list ls. This operator is right associative and we demonstrate its use by constructing the list literal 2,5,7,10.

```
    2 :: 5 :: 7 :: 10 :: Nil
≡   rightmost ::
    2 :: 5 :: 7 :: List(10)
≡   rightmost ::
    2 :: 5 :: List(7, 10)
≡   rightmost ::
    2 :: List(5, 7, 10)
≡   rightmost ::
    List(2, 5, 7, 10)
```

Operations on Lists

The three basic methods for working with lists are:

head – returns the first element in a non-empty list

tail - returns a list, possibly empty, of all elements in a non empty list except the first one

isEmpty – returns true if the list is empty; false, otherwise.

It is important to note that head and tail are only defined for non-empty lists. When selected from an empty list they throw an exception. The methods head and tail exploit the recursive structure of lists and can be used to express all operations on lists. The following example illustrates their meaning based on the given list literal ls.

```
val ls = List("Mary","Carmel", "Rory", "Donal")


ls.head == "Mary"
ls.tail == List("Carmel", "Rory", "Donal")
ls.tail.head == "Carmel"
```

We now consider some examples of processing integer lists using these three methods. The function getEven takes an integer list of values and returns the list of even values contained in it. It uses a similar approach to that taken in the function make defined above.

```
def getEven(ls : List[Int]): List[Int] ={
      if(!ls.isEmpty)
         if(ls.head % 2 == 0) ls.head :: getEven(ls.tail)
         else getEven(ls.tail)
      else Nil
 }
```

The result of getEven(List(2,66,96,90,55,5,3,17,23,42)) is the List(2, 66, 96, 90, 42).

As a second example we consider inserting a new element in a sorted list of values. The function insert takes a value x and a list ls, assumed sorted by <=, and inserts and returns a new list with x inserted so that the ordering is preserved. If ls.isEmpty it returns x :: Nil, or if x <= ls.head it returns x :: ls. Both of these cases are the same and can be handled by a single Boolean expression. The recursive case simply moves to the next value in the list.

```
def insert(x : Int, ls : List[Int]) : List[Int] ={
     if(ls.isEmpty || x <= ls.head) x :: ls
     else ls.head :: insert(x,ls.tail)
}


insert(3,List(1,2,5,6))
```

```
= def insert
   1 :: insert(3,List(2,5,6))
= def insert
   1 :: 2 :: insert(3,List(5,6))
= def insert
   1 :: 2 :: 3 :: List(5,6)
= def ::
  1 :: 2 :: List(3,5,6)
= def ::
  1 :: List(2,3,5,6)
= def ::
   List(1,2,3,5,6)
```

Prefixes and Suffixes

The methods that return the prefix and suffix of a list are called take and drop, respectively. The method ls.drop(n) drops the first n elements by returning a list all the remaining ones. If n is greater than the number of elements in the list then the empty list is returned. The method ls.take(n) returns the first n elements of the list. In this case if n is greater than the length of the list ls then it returns the whole list.

```
List(1,2,3,4,5,6).drop(3)  = List(4,5,6)
List(1,2,3,4,5,6).take(4) = List(1,2,3,4)
List(1,2,3,4,5,6).drop(2).take(2) = List(3,4,5,6).take(2) =  List(3,4)
```

In general, for a given list ls, ls.take(n) ::: ls.drop(n) = ls

You can also reference individual elements in a list using an index value. For example, ls(0) is the first element in list ls. However, the computational cost of using ls(n) is proportional to index n, i.e. it is O(n).

Pattern Matching

Pattern matching provides an alternative to the use of `if` expressions and `if` statements in function definitions. A pattern `match` includes a sequence of alternatives, each beginning with the reserved word `case`. Each `case` listed includes a pattern and one or more expressions that will be evaluated if the pattern matches. An arrow symbol, `=>`, separates the `case` pattern from the expressions. The `match` expression is evaluated top down by trying each of the patterns. The first match is chosen and the expression following the arrow is evaluated. There are a number of different kinds of pattern and we will discuss some of the different kinds in this section.

The simplest kind of pattern is the **constant** one. Given below is an example of a function that matches the values 1 and 2 only. It takes an integer as argument and matches it with the values 1 and 2. It returns the value as a string.

```
def matchConst(x:Int): String = x match{
      case 1 => 1.toString
        case 2 => 2.toString
}
```

This is not very useful but it illustrates how to write a pattern matching function. However, if you test this function with any value other than 1 or 2 it will throw an exception `MatchError` because it did not find a match. To overcome this problem we use a wildcard pattern, _ , that matches any object whatsoever. Using a wildcard we can write our match function so that it always returns a value.

```
def matchConst(x:Int): String = x match{
      case 1 => 1.toString
      case 2 => 2.toString
      case _ => "no match"
}
```

Now `matchConst(56)` returns `no match`. In this instance, a wildcard provides a default value. We will see later that they can also be used in other contexts.

A **variable** pattern matches any object, just like a wildcard. Unlike a wildcard it binds the variable to whatever the object is. This means that you can use the variable to interact with the object. The simple example below matches the parameter x with the value 1, returning the word one, and any other number, returning its value as a string. The function is:

```
def matchVar(x : Int):String = x match{
  case 1 => "One"
  case a => a.toString
}
```

A pattern **guard** can be used to define range patterns. A guard is defined as a Boolean expression and is placed after a pattern and starts with the word `if`. If a guard is used the match only succeeds if the guard evaluates to `true`. The function `matchRan` below tests a given integer value returning `true` only if it is in the range `1..99`.

```
def matchRan(x:Int):Boolean = x match{
     case a if a > 0 && a < 100 => true
     case _ => false
}
```

Pattern matching can also be used to write recursive functions. For example, the function `fac(n)` written as:

```
def fac(n : Int) : Int = {
  if(n == 0)  1
  else  n * fac(n-1)
}
```

could be re-written with pattern matching as follows:

```
def fac(n : Int) : Int = n match{
  case 0 => 1
  case a => a * fac(a-1)
}
```

A constant and a pattern variable are used to replace the if .. else.

A function, len, that returns the length of a given list can be written as:

```
def len(ls : List[Int]) : Int = ls match{
  case Nil => 0
  case x :: xs => 1 + len(xs)
}
```

The constant Nil denotes an empty list. The pattern x :: xs denotes a list with a head and a tail that may be Nil. This function is equivalent to the function len1 given below.

```
def len1(ls : List[Int]) : Int = {
  if(ls.isEmpty) 0
  else 1 + len1(ls.tail)
}
```

A scribe on completing a text exclaimed:

Explicit hoc totum;

Pro Christo da mihi potum