

Large Scale Programming - Lecture Notes

Mehmet Gençer
Istanbul Bilgi University
Department of Computer Science

Contents

Preface	iv
1 Introduction: Motivations for concurrent computing	1
1.1 An example problem	2
1.2 Mechanisms for concurrency	3
1.3 Exercises	4
2 Using Multiple threads of control	5
2.1 Thread Mechanics	8
2.2 Exercises	10
3 Controlling Access to Data	12
3.1 Sharing data between processes: producer consumer pattern and critical sections	12
3.1.1 Critical section problem, atomicity, and the need for mutual exclusion	16
3.2 Using system services for mutual exclusion: Locks and semaphores	16
3.2.1 Example: A synchronized queue implementation using binary semaphores	18
3.3 Using non-binary semaphores	19
3.4 Atomicity. the Volatile modifier and java.util.concurrent.atomic library	20
3.5 Motivations for mutual exclusion practice	21
3.6 Exercises	21
4 Java object oriented mechanisms for mutual exclusion: Ob- ject monitors and guarded methods	23
4.1 Exercises	24
4.2 Example: Buffer implementation revisited	25
4.3 Exercises	26

5	Thread coordination and deadlocks	27
5.1	Efficient guarded blocks	29
5.1.1	Example: the buffer implementation	31
5.2	Deadlocks(starvation) and livelocks	32
5.3	Profiling Java processes	32
5.4	Exercises	33
6	Thread pooling	34
6.0.1	Choosing the thread pool size	36
6.0.2	Short running tasks and Executors	36
6.0.3	Callables	37
6.1	Exercises	37
7	Distributed programming	39
7.1	Different standards	39
7.2	Requirements	40
7.3	Topologies, and concurrency issues	41
7.4	Potential problems	41
8	Remote Method Invocation (RMI)	42
8.1	RMI Concurrency	46
8.2	Remote objects versus object transportation	47
8.3	Exercises	47
9	XML-RPC	48
9.0.1	XMLRPC Introspection	50
9.0.2	Asynchronous XMLRPC calls	50
9.1	Code Examples	51
9.1.1	XMLRPC Example - The shared object	51
9.1.2	XMLRPC Example - Server	52
9.1.3	XMLRPC Example - Client	53
9.1.4	XMLRPC Example - A Python client	55
9.1.5	XMLRPC Example - An Asynchronous Client	56
10	Advanced topics	58
10.1	Architectures for distributed programming	58
10.1.1	Exercises	58
10.2	Java network-attached-memory	59
10.3	Network Security	59
10.4	Center-less architectures	60

<i>CONTENTS</i>	iii
A A guide for coding style	61
A.1 Demarcation	61
A.2 Organization	63
A.3 Explanation	64
A.4 Explication	65
A.5 Naming conventions	66
Notes	67
References	67

Preface

These lecture notes are aimed to support Large Scale Programming course offered by the Computer Science Department of İstanbul Bilgi University. The course is offered to students who have taken courses on essential programming techniques based on functional programming using Scheme, and later object oriented and imperative programming in Java.

Several other resources are referenced in the text but nevertheless these lecture notes are prepared to be used as a standalone material.

Chapter 1

Introduction: Motivations for concurrent computing

All computer programs you have written so far during your computer science education relies on the assumption of sequentiality that there is a single thread of execution in the process. An important consequence of this assumption that your program can rely on the stability of variables that no other process is changing them while your sequence of statements do they work.

In this course we will work through the consequences of removing this assumption, techniques for creating concurrent rather than sequential processes, and example computational problems that motivates doing so.

Sequential processes follow the program and does one thing at any given instant. However in many computing problems it is necessary and possible to speed up task completion time by utilizing multiple processors or computers to divide the task into smaller pieces.

In recent decades processor features labeled as ‘duo’ or ‘quad’ core, 32 or 64 bit, etc., have become commonplace. The transition from 32-bit to 64-bit memory bus enabled processors to use more memory if available¹. Duo or quad-core CPU’s have actually two and four CPU’s inside a single chip, which work in parallel. While starting with Intel 386 family of CPUs, the chip features enabled operating systems to have multitasking features and improved system responsiveness, in GUI programs for example, the availability of multiple cores creates actual computing speed gains. Furthermore wide availability of high speed networks and Internet makes possible to combine power of separate (and even remote) computer to implement parallel

¹32-bit addresses can use up to 2^{32} bytes of RAM (4 GB), whereas in 64 bit processors the number goes up to 2^{64} .

algorithms.

In summary the motivations for concurrency are as follows:

- Speed up completion of parallel algorithms that require a long time to complete otherwise.
- Improve program responsiveness, e.g. in GUI applications.
- Implement real time applications, e.g. multimedia players or audio or visual processing.
- Combine remote computing resources (computing cycles, databases, human contribution).

1.1 An example problem

Consider the problem of finding whether each in a list of natural numbers are prime or not. When doing such a task, the assessment of each number's primeness is a process independent of assessment of the others. Such tasks are commonly called embarrassingly parallel since there is no coordination between such independent parts of the whole process. Therefore these are relatively easier to implement using hardware parallelism (e.g. multiple CPUs). For example one can run a code like follows to do this:

```
boolean isPrime(int n) {
    if (n%2==0)
        return false;
    for (int i=3;i<n/2;i=i+2)
        if (n%i==0)
            return false;
    return true;
}

int[] list= ... some list of numbers ...
for (int i=0;i<list.length;i++)
    is list[i] prime = RUN IN PARALLEL(isPrime(list[i]));
```

For the moment we leave the details of how to run functions in parallel or store their return value to further chapters.

Despite relative ease of the problem one must be very careful in implementing the solution in a program. Consider for example the list of numbers (5,12, 2356789). In this specific case even if one runs three computers, one

for each number, to assess primality of numbers the total task completion time will not change, because the time needed for assessment of the first two numbers will be negligibly small compared to the time needed for assessment of the third number's primality. As a consequence it will be a good idea to further parallelize primality checking, for example a rough and incomplete algorithm for this would look like follows:

```
boolean isPrimeRange(int n, int from, int to) {
    if (n%2==0)
        return false;
    for (int i=from;i<to;i++)
        if (n%i==0)
            return false;
    return true;
}
```

```
boolean isPrimeParallel(int n, int numparts){
    if (n%2==0)
        return false;
    boolean[] results=new boolean[numparts];
    for(int i=0;i<numparts;i++)
        results[i]=RUN IN PARALLEL(isPrime(n, i*n/2/numparts, (i+1)*n/2/numparts))
    for(int i=0;i<numparts;i++)
        if (!results[i])
            return false;
    return true;
}
```

```
int[] list= ... some list of numbers ...
for (int i=0;i<list.length;i++)
    is list[i] prime = RUN IN PARALLEL(isPrimeParallel(list[i], list[i]/SOMETHRES
```

1.2 Mechanisms for concurrency

Concurrent processes to complete a task (provided that it can be broken down into independent sequential sub-tasks) can be achieved by using multiple threads on the same computer, or by coordinated execution of processes on different computers which communicate with one another using the Internet.

In the following sections we will first explore the first technique, of multi-threaded programming. However, as the above example demonstrates parallelism brings up a series of problems of its own. One of these is trashing

which is the situation where a computer systems stops functioning reliably and efficiently because too many threads are running. The very existence of multiple-threads causes an overhead on the operating system which manages them. Our exploration will also address such issues. Another problem important in multi-threaded programming is related to concurrent modification of data by multiple threads of execution. There are mechanisms for attaining synchronization between threads when modifying data, however further problems arise related to synchronization because of the risks in programming that may leave two or more processes awaiting forever for each other (so called a dead-lock situation) because of inappropriate use of synchronization mechanisms.

The second technique of running concurrent processes on different computers have one advantage over multi-threaded programming: the number of computers is not limited. On the other hand when we try to exploit the computation power of a single computer (e.g. one having multi-core CPU, or several CPUs), the computing power that can be reached on a single computer is usually very limited, although these days it is possible to buy 16 CPU, 64 GB systems that are commercially available. Multi-computer parallelism is widely used in scientific research which require heavy computations (e.g. physics simulations), or industrial systems such as heavyweight web applications or computer animation rendering.

1.3 Exercises

- E-1 Write a Java console application which reads natural numbers from the console repetitively until an empty entry is encountered, and after reading each number reports whether it is prime or not.
- E-2 To refresh your Processing system skills write a Processing program which animates 10 filled circles of different color, each of which are centered on a random position, starts from a diameter of 10, grows into diameter of 100 as they are drawn repetitively, then returns to diameter 10 again.

Chapter 2

Using Multiple threads of control

A group of threads work concurrently and independently, but they share the same memory space (variable scope) and have means of effecting one another's work.

Let us consider problem E-2 in the previous chapter. Each shape in that animation can be considered as independent of each other. Therefore we can start learning Java mechanisms to create multiple threads using this example, by creating a thread to control each circle.

Let us first start by looking at a single threaded version of animation program:

```
int sizeX=500;
int sizeY=500;
GrowingCircle[] circles=new GrowingCircle[10];

class GrowingCircle {
    float x,y;
    int radius,limit;
    color c;
    GrowingCircle(){
        x=random(sizeX);
        y=random(sizeY);
        c=color(random(255));
        radius=10;
        limit=(int)random(50)+50;
    }
    void draw() {
```

```
        fill(c);
        ellipse(x,y,radius,radius);
    }
    void move() {
        radius+=1;
        if(radius>limit) radius=10;
    }
}

void setup() {
    size(sizeX,sizeY);
    for(int i=0;i<circles.length;i=i+1) {
        circles[i]=new GrowingCircle();
    }
}

void draw() {
    background(255);
    for(int i=0;i<circles.length;i=i+1) {
        circles[i].draw();
        circles[i].move();
    }
}
```

Following is a version which runs a separate thread to control each circle.

```
int sizeX=500;
int sizeY=500;
int numcircles=5;
GrowingCircle[] circles=new GrowingCircle[numcircles];
Thread[] threads=new Thread[numcircles];

class GrowingCircle implements Runnable{
    float x,y;
    int radius,limit;
    color c;
    GrowingCircle(){
        x=random(sizeX);
        y=random(sizeY);
        c=color(random(255));
        radius=10;
        limit=(int)random(50)+50;
    }
}
```

```
void draw() {
    fill(c);
    ellipse(x,y,radius,radius);
}
void move() {
    radius+=1;
    if(radius>limit) radius=10;
}
void run() {
    while (true)
        move();
}
}

void setup() {
    size(sizeX,sizeY);
    frameRate(100);
    for(int i=0;i<circles.length;i=i+1) {
        circles[i]=new GrowingCircle();
        threads[i]=new Thread(circles[i]);
        threads[i].start();
    }
}
void draw() {
    background(255);
    for(int i=0;i<circles.length;i=i+1) {
        circles[i].draw();
    }
}
```

(Note the growth of circles with no control of growth speed!)

If you compare the two programs you will see that the class `GrowingCircle` in the second program implements an interface called `Runnable`. The cost of implementing this interface is to have a method “`void run()`”. This is the typical way of creating code to run in a thread of execution. The instance of such a class can later be given as the parameter to the constructor of “`Thread`” class. When the “`start()`” method of a thread instance is called, the `run()` method of our class instance (the `GrowingCircle` which implements `Runnable`) starts running in an additional thread of control. In other words the so called “main thread” (the one that invokes `Thread.start()`) continues to run from where it was as usual, while the new thread starts running the

run() method. It is very common that such run() methods have a long running or infinite loop as in our example.

There are two mechanisms to create threads in Java: to extend the Thread class or to implement the Runnable interface. The program uses the second method as it avoids limitations of polymorphism in most cases. Once a thread is created it must be started. The so called ‘main’ thread becomes the owner of the newly created thread, and among other things has the right to interrupt the child thread or declare it as a daemon (ie. non-interacting with user).

With threads it is not well defined how they receive system signals (e.g. to which thread the interrupt created by pressing CTRL-C will go?). These matters were attempted to be clarified in POSIX standard.

2.1 Thread Mechanics

Note that a Thread instance is like any other object in Java. The Thread class encapsulates several methods to control threads. For example a thread instance can be interrupted, joined, its ID can be retrieved, etc. Please refer to java reference documentation of the Thread class for a full reference of operations. One of these operations is sleep() which can be used to control the speed of growth of circles in the example above.

Of particular interest in multi-threaded programming is how threads are terminated and how the main thread coordinates their execution, for example when the program is interrupted by CTRL-C.

Let us consider the example problem of assessing primality of a set of numbers. Following is a program which creates and runs a thread for assessment of each number read from console:

```
/**
 * A Program to check primality of numbers given from console
 * Author: Mehmet Gençer, mgencer@cs.bilgi.edu.tr
 */
import java.util.ArrayList;

class NumberChecker implements Runnable {
    long n;
    boolean isPrime;

    public NumberChecker(long n) {this.n=n;}

    public void run() {
```

```
        isPrime=true;
        if (n%2==0)
            isPrime=false;
        else
            for(int i=3;i<n/2;i=i+2)
                if (n%i==0){
                    isPrime=false;
                    //break;
                }
        System.console().format("Finished checking primality of %d (%b)\n",
            n,isPrime);
    }

    boolean primeness() {return isPrime;}
}

public class PrimeCheck {
    public static void main(String[] args) {
        ArrayList<NumberChecker> checkers=new ArrayList<NumberChecker>();
        ArrayList<Thread> threads=new ArrayList<Thread>();
        while(true) {
            try{
                String input=System.console().readLine();
                long n=Long.parseLong(input);
                NumberChecker checker=new NumberChecker(n);
                checkers.add(checker);
                Thread thread=new Thread(checker);
                threads.add(thread);
                //thread.setDaemon(true);//set as daemon!
                thread.start();
            } catch (NumberFormatException e) {
                System.out.println(e);
                break;
            }
        }
        for(int i=0;i<threads.size();i=i+1)
            try {
                threads.get(i).join(); //wait for the thread
                System.console().format("Number %d is prime?: %b\n",
                    checkers.get(i).n,checkers.get(i).primeness());
            }catch(InterruptedException e) { //if CTRL-C is pressed
```

```
        System.console().format("Interrupted");
    }
}
}
```

The program sets the threads created as daemons. This is a sane practice since JVM will not exit if there are non-daemon threads running, even when the main thread is interrupted with CTRL-C. When the program does not have any more numbers to process it waits for the threads to finish by calling the `join()` method.

2.2 Exercises

- E-1 Run the single threaded and multi-threaded versions program for primality check by feeding prepared input files of varying length. Report the performance comparison with respect to single threaded version and number of threads.
- E-2 Write a Java console application which reads natural numbers from the console repetitively until an empty entry is encountered, finds their prime factors by running multiple threads (one for each number), and at the end reports the common prime factors of all numbers.
- E-3 A timed nuclear bomb needs to be stopped. The bomb can be stopped if the n byte pass code is known. Luckily we have the knowledge of the CRC-32 value of the pass code obtained from the bomb circuitry.
- Write a program to find all possible pass codes by brute force search using two threads. Your program must take the number n , and the CRC-32 checksum value as its first and second command line arguments, respectively.
 - By benchmarking the performance of your program. Compute the number of processors, C , that needs to be exploited by an appropriate multi-threaded program, to find the pass code of length n in S seconds. Defend your solution with the benchmarks for n of your choice.

HINTS:

- Use the `byte` data type in Java. The byte data type is an 8-bit signed integer. It has a minimum value of -128 and a maximum value of 127 (inclusive).

- Use the CRC32 class from java.util.zip package to compute CRC-32 checksums.
- Use the `time` command in Linux to benchmark your program.

Chapter 3

Controlling Access to Data

Consider that two threads accessing same variables by running two processes executing the code below:

```
i=i+1
```

Despite its simplicity the statement is in fact takes three steps to complete: (1)take value of i, (2) evaluate the addition, (3) put the result into i. When one thread is in the middle of this task what happens if another one comes and starts? It is very likely that the intended result that i is increased twice will not be achieved and instead i would have been increased once. Although the odds of such coincidence is very low in this example (since the operation is very fast), it may not be low in other situations and the consequences are usually very critical (think about a withdrawal from a bank account!). Therefore when one writes multi-threaded programs one needs appropriate mechanisms to coordinate/synchronize concurrent accesses to data.

3.1 Sharing data between processes: producer consumer pattern and critical sections

The advantage of threads is the fact that they share memory (variables) and hence have a good change of implementing necessary level of coordination. Consider the following Java class which implements an integer FIFO queue:

```
interface IIntegerQueue {
    int getCapacity();
    void put(int n) throws IllegalAccessException;
    int get() throws NoSuchElementException;
    int numberOfElements();
}
```

```
}

class FIFO implements IIntegerQueue {
    int capacity;
    int[] values;
    int numValues;

    public FIFO() {
        capacity=1024;
        values=new int[capacity];
        numValues=0;
    }
    public FIFO(int c) {
        capacity=c;
        values=new int[capacity];
        numValues=0;
    }
    public int getCapacity() {return capacity;}
    public int numberOfElements() {return numValues;}
    public void put (int n) throws IllegalAccessException {
        if (numValues>=capacity)
            throw new IllegalAccessException("Queue is full");
        values[numValues++]=n;
    }
    public int get () throws NoSuchElementException{
        if (numValues==0)
            throw new NoSuchElementException("No elements in queue");
        int val=values[0];
        numValues--;
        for (int i=0;i<numValues;i++)
            values[i]=values[i+1];
        return val;
    }
}
```

In this implementation, the `get..()` method of the FIFO queue is a time consuming one, since it shifts all elements in the queue after retrieval. Therefore if we use multiple threads which put elements into the queue (or take elements, for that matter), there is a high probability that things will coincide and operation will go wrong.

As an example run the following multi-threaded program on a queue:

```
class Producer implements Runnable{
    IIntegerQueue q;
    int e,c;
    public Producer(IIntegerQueue q, int c) {
        this.q=q;
        e=0;
        this.c=c;
    }
    public void run() {
        for (int i=0;i<c;i++)
            if (q.numberofElements()<q.getCapacity()) //THERE IS SPACE!
                try {
                    q.put(1);
                } catch(Exception ex) {e+=1;}
    }
}

class Consumer implements Runnable{
    IIntegerQueue q;
    int e,c;
    public Consumer(IIntegerQueue q, int c) {
        this.q=q;
        e=0;
        this.c=c;
    }
    public void run() {
        for (int i=0;i<c;i++)
            if (q.numberofElements()>0) //THERE IS SPACE!
                try {
                    q.get();
                } catch(Exception ex) {e+=1;}
    }
}

public class IntegerQueues {
    public static void main(String[] args) {
        int numProducers=10;
        int numConsumers=10;
        int c=10000000;
        IIntegerQueue q=new FIFO(c);
        Producer[] producers=new Producer[numProducers];
```

```

Consumer[] consumers=new Consumer[numConsumers];
Thread[] pthreads=new Thread[numProducers];
Thread[] cthreads=new Thread[numConsumers];
for(int i=0;i<numProducers;i++) {
    producers[i]=new Producer(q,c/numProducers);
    pthreads[i]=new Thread(producers[i]);
    pthreads[i].start();
}
for(int i=0;i<numConsumers;i++) {
    consumers[i]=new Consumer(q,c/numConsumers);
    cthreads[i]=new Thread(consumers[i]);
    cthreads[i].start();
}
int prod=0,perr=0, cons=0,cerr=0;
try {
    for(int i=0;i<numProducers;i++) {
        pthreads[i].join();
        prod+=producers[i].c;
        perr+=producers[i].e;
    }
    for(int i=0;i<numConsumers;i++) {
        cthreads[i].join();
        cons+=consumers[i].c;
        cerr+=consumers[i].e;
    }
}catch(InterruptedException e) {}
System.console().format("Produced: %d, Error:%d\n",prod,perr);
System.console().format("Consumed: %d, Error:%d\n",cons,cerr);
System.console().format("In queue: %d\n",q.numberofElements());
}
}

```

The scenario in which multiple threads put and get elements from a shared data structure is called the Producer/Consumer pattern. Working on common data concurrently and independently have certain dangers. Our example scenario exposes these problems. You will quickly notice that several things can go wrong when such a program is run:

- Each producer and consumer keeps track of the number of things removed/added to the queue. However when the program checks these counts against the length of remaining list of jobs at the end, the counts may not match!

- Although producer and consumer threads are careful to check whether the job queue is empty or full, the process encounters some exceptions. We have -yet- no mechanism to remedy such situations which require some communication between threads.

Aside from these problems, one generally observes that processes use fair shares of CPU time, as their jobcounts are similar.

3.1.1 Critical section problem, atomicity, and the need for mutual exclusion

The example above demonstrates that it is critical to ensure other concurrent processes does not mingle with the shared variable (or resource such as File I/O) while one is making an update on the resource. In other words, in the producer/consumer example the successive operations of getting and removing an item from job queue is an *atomic* operation which should not be interrupted. When one process is working through an atomic operation on a resource, others should be excluded from doing so.

Several such situations occur in real life other than the computer processes. Use of road junctions, or operations on a credit card or bank account are such examples. In the case of road junctions, mutual exclusion is obtained by allocation of successive time slots to drivers coming from different directions (consumers of the road junction). As a different solution, hunting swordfish change their color to indicate to others that it is attacking the hunt, to prevent concurrent attacks and possible injuries resulting from that.

Indeed the methods we will employ to solve mutual exclusion problem are similar to that of swordfish. Dijkstra was first to propose a solution (Dijkstra, 1965)

3.2 Using system services for mutual exclusion: Locks and semaphores

Dijkstra's solution for mutual exclusion requires enumeration of processes and sharing of common data to for synchronization. While cumbersome, but nevertheless valid, it does not work for some situations; specifically for file handle sharing. For these reasons several mechanisms were implemented by operating systems to provide mutual exclusion mechanics to user applications.

The most general mechanism for mutual exclusion is semaphore (also the name for traffic lights). The so called 'general semaphore' has a fixed number

of resources available for which processes race for. When all resources are taken, the semaphore will not give permit to other processes to proceed. A semaphore, s , provides two operations:

- $P(s)$: Acquire resource
- $V(s)$: Release resource

The algorithm for general semaphore is quite straightforward:

```
initialize (Semaphore s, int numberOfResources)
{
    s = numberOfResources;
}
P(s) //ATOMIC OPERATION!!
{
    wait until s > 0
    s = s-1;
}

V(s) //ATOMIC OPERATION!!
{
    s = s+1;
}
```

Despite its simplicity, the semaphores are different from application programs or libraries because operations on it must be atomic. For this reason semaphores are implemented at the operating system level.

Most common form of semaphores in use is binary semaphore, which is simply called as *lock*. In binary semaphores the number of resources is one. Therefore at any instance only one process can acquire the binary semaphore.

Basic definition of semaphores does not guarantee fairness of resource distribution. However, many implementations provide fairness by keeping a queue (first in first out, FIFO) of processes waiting to acquire the resource and delivering permits in the queue order.

A critical section in a multi-threaded program is the part(s) of the program whic access shared data/variables. Such sections of the program must be set up for mutual exclusion: i.e. when one thread is running that particular section of the program, another thread should not. This is usually achieved by using locks. For example:

```
lock.acquire()
criticalData.update()
lock.release()
```

Since the lock, when already acquired by one thread, cannot be acquired by another before the first one releases it, the critical section above is safe: only one thread will be executing it at any given time instance.

3.2.1 Example: A synchronized queue implementation using binary semaphores

The size limited queue used in the producer/consumer example above is essentially a buffer of entities. Since operations on unlimited size lists in Java utilities library (e.g. `ArrayList`) can be very slow, we instead implemented a fixed size buffer. However despite its speed, our queue is not reliable when accessed by multiple threads.

Following is a version of integer queue which uses locks for mutual exclusion of the critical sections:

```
class Sorted implements IIntegerQueue {
    int capacity;
    int[] values;
    int numValues;
    ReentrantLock lock;
    public Sorted() {
        capacity=1024;
        values=new int[capacity];
        numValues=0;
        lock=new ReentrantLock();
    }
    public Sorted(int c) {
        capacity=c;
        values=new int[capacity];
        numValues=0;
        lock=new ReentrantLock();
    }
    public int getCapacity() {return capacity;}
    public int numberOfElements() {return numValues;}
    public void put (int n) throws IllegalArgumentException {
        if (numValues>=capacity)
            throw new IllegalArgumentException("Queue is full");
        lock.lock(); // ACQUIRES THE BINARY SEMAPHORE!
        values[numValues++]=n;
        lock.unlock(); //RELEASES THE BINARY SEMAPHORE
    }
}
```

```

public int get () throws NoSuchElementException{
    if (numValues==0)
        throw new NoSuchElementException("No elements in queue");
    lock.lock();
    int max=values[0];
    int pos=0;
    for(int i=0;i<numValues;i++)
        if (values[i]>max) {
            pos=i;
            max=values[i];
        }
    numValues--;
    for(int i=pos;i<numValues;i++)
        values[i]=values[i+1];
    lock.unlock();
    return max;
}
}

```

The implementation uses the lock implementation in Java, named `ReentrantLock`. The name is due to the fact that if a thread already holds a lock, it can re-obtain it.

3.3 Using non-binary semaphores

Despite the sanity of operations in our queue implementation it is somewhat inconvenient to use since all users need to check for exceptions. Such user code is likely to be spinning until buffer operations succeed or capacity is available. Therefore it would be much more useful if we instead provide blocking calls for operating on the buffer. This can be done using a general semaphore (adopted from (Lea, 2000)):

```

import java.util.concurrent.*;

class SemaphoreBuffer {
    int size,putPointer,takePointer;
    Object[] objects;
    Semaphore putPermits,getPermits;
    public SemaphoreBuffer(int maxSize) {
        size=maxSize;
        objects=new Object[size];
    }
}

```

```

        putPermits=new Semaphore(size);
        getPermits=new Semaphore(0);
        putPointer=0;
        takePointer=0;
    }
    void put(Object x) {
        putPermits.acquireUninterruptibly();
        objects[putPointer]=x;
        putPointer=(putPointer+1)%size;
        getPermits.release();
    }
    Object get() {
        getPermits.acquireUninterruptibly();
        Object x=objects[takePointer];
        takePointer=(takePointer+1)%size;
        putPermits.release();
        return x;
    }
    int size() {
        return getPermits.availablePermits();
    }
}

```

Note that this buffer implementation is much more efficient than the FIFO example we have seen before because it does not shift elements, and instead spans the array cyclically.

3.4 Atomicity. the Volatile modifier and `java.util.concurrent` library

Even the simple statements like `x++` are not atomic in Java, as we have seen at the beginning of this chapter. One can always overcome such situations using the lock mechanisms we have seen. However, Java offers a syntactic solution to the problem concerning primitive data types: the `volatile` modifier. Although the volatile modifier does not eliminate common types of concurrent read and write problems, it does ensure that non-atomic operations such as writes on long and double data types become atomic when the variable is declared using the volatile modifier, as follows:

```
volatile long l;
```

In addition to volatile mechanism, Java SDK offers a library, `java.util.concurrent.atomic`, which can be used to make read and write operations on classes that are defined in the package and correspond to primitive data types, and also objects of any type. While reliable, use of these classes does not seem to produce readable code. Therefore the students are recommended to use the library on their discretion.

3.5 Motivations for mutual exclusion practice

Speed of updates is not the only reason for using lock to ensure mutual exclusion. Indeed it is rare that one encounters situations as the artificially fast updates in our examples above. However there are often situations in which steps of completion of a task involve time consuming operations such as Internet communication across multiple systems. Consider, for example, the money withdrawal from a bank account using an ATM machine. The process can be summarized as the steps below:

1. Read the amount which the customer wants to withdraw.
2. Communicate with the bank data center whether the amount is available in the account.
3. If the amount is available tell the banknote machinery to deliver the amount to customer.
4. If the delivery is successful, communicate with the data center to deduce the amount from the account.

Now in this operation the money delivery machinery may not be able to deliver the required amount, or some awkward error may happen during delivery. That is why the deduction must wait until delivery succeeds. On the other hand the bank must make sure that a concurrent withdrawal does not happen which can cause the account balance to go below zero. In this process a locking of the account is necessary across step 2 through step 4 of the process.

3.6 Exercises

- E-1 Consider a stock management system in a large retail store in which several cashiers are operating concurrently. When each cashier checks

a good through the barcode reader, the central system must reduce the amount of stock corresponding to the good by one. Write a program which implements the whole system in a single process and allows a cashier interface from the console. Although your program is single threaded, make sure the stock accounting classes are safe for concurrent use.

- E-2 Write a Java application to manage queues in a bank office. Assume that there is a single type of operation (ie. single queue), two queue ticket machines (i.e. producers), and n bank officers (i.e. consumers). Implement your producer and consumer threads to use random arrival interval and service time. Averages of arrival interval and service time, in addition to number of officers must be given as command line arguments, so that you can change these to see how system performs under different scenarios. Your application must be able to report average waiting time to newcomers taking a queue no ticket. The main thread should report every second the status of queue and average waiting time. (Hint: An advisable way to maintain running average of waiting time could be using exponential averages.)

To simplify the problem assume that the queue size is not limited and ticket numbers are not cyclic. i.e. the machines can issue tickets starting from 1 and goes up, without turning back to 1 as common in the actual bank applications.

Chapter 4

Java object oriented mechanisms for mutual exclusion: Object monitors and guarded methods

Every Java object has an intrinsic lock associated with it, which is called monitor lock, or shortly monitor (remember that primitive types are not objects, thus have no monitor). Based on the availability of monitor lock, Java provides a peculiar object oriented mutual exclusion mechanism. By using the `synchronized` modifier to enclose a code block one can obtain the result of what otherwise requires a `Lock`. Following is an example that uses `synchronized` modifier, and its translation into code using the familiar locks we have learned in the previous chapter:

SYNCHRONIZED CODE	----->	TRANSLATION
<code>Object o;</code>		<code>Object o;</code>
<code>...</code>		<code>ReentrantLock lock;//used only for 'o'</code>
<code>synchronized (o) {</code>		<code>lock.lock();</code>
<code>...someAction on object...</code>		<code>...someAction on object...</code>
<code>}</code>		<code>lock.unlock();</code>

The `synchronized` statement is more often used to declare ‘guarded methods’, which are intended to be mutually exclusive when multiple threads use a class’ instance. For example:

SYNCHRONIZED CODE	----->	TRANSLATION
<code>class C {</code>		

```

//already has an intrinsic 'lock'
...
synchronized void action() {
    ...
}
synchronized void anotherAction() {
    ...
}
}

void action() {
    intrinsicLock.lock();
    ...
    intrinsicLock.unlock();
}
void anotherAction() {
    intrinsicLock.lock();
    ...
    intrinsicLock.unlock();
}
}

```

The code above is equivalent to enclosing the whole body of a method within `synchronized (this)`, and what makes the synchronization possible is the use of intrinsic lock of every Java object, as indicated in the translation above.

Only one of any guarded methods of an object is ensured to be executed by only one thread at any time. In other words, if a thread is executing `action()` method of the object above, and another concurrent thread attempts to execute `anotherAction()` method, the latter thread must wait until the first one exists the synchronized method and releases the intrinsic lock. On the other hand the monitor lock are reentrant, i.e. if for example the `action()` method calls `anotherAction()` method, it will succeed, because the calling thread already holds the lock and therefore can re-obtain it (since it is re-entrant). In fact instances `ReentrantLock` is used to implement monitor locks, and this property is why the class is named as such.

Note that constructor methods cannot be synchronized using this mechanism!

4.1 Exercises

- E-1 `synchronized` modifier can also be used for `static` methods of classes. What does it mean in this case? What intrinsic monitor is being used?

4.2 Example: Buffer implementation revisited

Availability of monitor can make implementation of our Buffer object much easier. Following is a rewrite of controlled buffer using Java synchronization mechanics:

```
class SynchronizedControlledBuffer {
    int size, putPointer, takePointer, currentSize;
    Object[] objects;
    public SynchronizedControlledBuffer(int maxSize) {
        size=maxSize;
        objects=new Object[size];
        putPointer=0;
        takePointer=0;
        currentSize=0;
    }
    synchronized int size() {return currentSize;}
    synchronized void put(Object x) throws Exception{
        if (currentSize>=size)
            throw new Exception("Buffer is full");
        objects[putPointer]=x;
        putPointer=(putPointer+1)%size;
        currentSize+=1;
    }
    synchronized Object get() throws Exception{
        if (currentSize==0)
            throw new Exception("Buffer is empty");
        Object x=objects[takePointer];
        takePointer=(takePointer+1)%size;
        currentSize-=1;
        return x;
    }
}
```

The code is much shorter and readable. Since both methods are synchronized to the buffer instance, we can make sure that only one thread can operate on the buffer at any time.

4.3 Exercises

- E-2 Redo the exercises from the previous chapter using object monitors instead of explicitly defined locks.
- E-3 Write a Java class to implement a sorted queue to store strings. Test your class using a main method and example operations.

Chapter 5

Thread coordination and deadlocks

The locks and monitors provide a sanity mechanism to ensure that concurrent threads that operate on common data does not cause inconsistent modifications (or access inconsistent states) of data. However we face a different problem when operation of a thread needs to wait for something else (in another thread) to happen.

Consider for example the integer queue implementation example from Chapter 3. We have defined the `IIntegerQueue` interface so that the implementing classes throw exceptions in cases where there's nothing that can be taken from the queue, or there is no empty space to put new elements. However such an interface is inconvenient for using in programs. What is one supposed to do when such an exception is encountered? It is likely that the code using the queue needs to wait until some elements are available for taking, or space is available for putting new elements to the queue. Such a code will look like follows:

```
class SomeQueue implements IIntegerQueue {
    ...
}

SomeQueue q = new SomeQueue();
...
while(q.numberOfElements()==0) //no elements available for taking
{ //NOTHING TO DO!
}
q.get()
```

The above seems like a solution to our problem. However, it is not usable for

two reasons: (1)the while loop itself wastes CPU cycles unnecessarily, and (2)if two threads are executing the while loop above at the same time, they will both proceed to get an element from the queue when one is available, but only one will be successful! Such looping is called spinning or polling, and it is a serious impediment to software performance. Although the second problem can be overcome by wrapping the whole while loop in a synchronized block (synchronized to the queue), the performance problem is persistent.

In such situations it is more suitable to encapsulate the condition within the class implementation. For one thing this prevents one from writing such loops everywhere in the program that uses our queue. If we proceed with this design idea, the interface and the code can be changed as follows:

```
interface IIntegerQueue {
    ...
    int get();
    ...
}

class SomeQueue implements IIntegerQueue{
    ...
    synchronized void put() {
        ...
    }
    synchronized int get() {
        while (numberOfElements()==0) {}
        return ...;
    }
    ...
}
```

Calls to such method implementations which can take an indefinitely long time to return are called blocking calls. Blocking calls are used in many situations to simplify code, by awaiting for a condition rather than throwing an exception.

The above implementation seems to encapsulates the coordination issue inside the class implementation and can simplify our programs. However, it has even more serious problems. For one thing it does not remedy the spinning problem: it is simply placed elsewhere but still consumes unnecessary CPU cycles. More seriously the above program will never return an element from the queue! Since the `put()` and `get()` methods are synchronized, when the `get()` method waits for an element to arrive, the calling thread owns the object monitor lock. As a consequence no other thread will be able to enter

the `put()` method and satisfy the condition for which the `get()` is waiting for. Such a situation is called starvation, or deadlock in concurrent programming parlance (see the sections below).

5.1 Efficient guarded blocks

An efficient guarded method or block must not have the deficiency of consuming CPU cycles as in the above examples. Fortunately we have some mechanisms to avoid this and make our guarded code block more efficient. If you ever look at the Java reference documentation you will observe that each class inherits several methods from the ‘Object’ base class. Each class we define in Java or comes in its SDK libraries inherits some methods called `wait()` and `notify()/notifyAll()` from the Object parent. You are recommended to read the reference documentation for these methods in detail.

The `wait()` call puts the calling thread to sleep. If `notify()` method is invoked, one of such threads sleeping in this way is randomly chosen and awakened. `notifyAll()` awakens all threads waiting on the object.

A thread can call `wait` or `notify` methods on an object only if it is the owner of object’s monitor, and Java documentation tells that this happens in the following cases:

- By executing a synchronized instance method of that object.
- By executing the body of a synchronized statement that synchronizes on the object.
- For objects of type `Class`, by executing a synchronized static method of that class.

Only one thread at a time can own an object’s monitor. When a thread calls the `wait()` method, the monitor lock is released after the thread is put into sleep. Similarly when multiple threads are awakened by `notifyAll()`, each must wait for their turn to obtain the monitor lock. In other words they will all be awakened, but one by one rather than at once.

In summary here’s what happens when `wait` and `notify` calls are used:

Wait : Upon a `wait` action the monitor lock for the synchronization object is released. The thread calling `wait()` is placed in an internal wait set associated with the synchronization object.

Interrupt : If a thread is interrupted during `wait`, the `wait()` action exits immediately throwing an `InterruptedException`.

Notify When `notify` is called, an arbitrary thread among those which are waiting on the object is chosen. The chosen thread will return from the `wait` call and obtain the synchronization lock, as soon as the notifying thread releases the lock (i.e. it exits the synchronized method in which the `notify()` is called).

NotifyAll : Similar to `notify`, but all threads will be notified. However, only one can actually get the monitor lock and proceed. So in effect threads will continue one at a time.

The coordination mechanics can be useful in cases like our queue implementation above. We can now implement our queue as follows:

```
interface IIntegerQueue {
    ...
    void put(int n);
    int get();
    ...
}

class SomeQueue implements IIntegerQueue{
    ...
    synchronized void put() {
        while (numberOfElements()==getCapacity()) //no place
            wait();
        values[numValues++]=n;
        notifyAll()
    }
    synchronized int get() {
        while (numberOfElements()==0)
            wait();
        notifyAll();
        return ...;
    }
    ...
}
```

Our code uses `notifyAll()` instead of `notify()`, because if a getter thread receives a getter's notification, it is useless. As a consequence all threads are awakened, and they all re-check the condition (thus the while loop) before proceeding. However, this time the while loops are not CPU consuming as they were before. AS a general precaution, one must always re-check the

condition in such situations and should not assume that the notification was for the particular condition that one was waiting for.

The pseudo-code above is somewhat incomplete since one must take care of `InterruptedException` wherever the `wait()` method is called. The most sensible thing in such situations is to declare the methods as throwing this exception and let it pass through the call sequence. The example in the following section demonstrates this.

You will notice that there is a version of the `wait()` method which accepts a timeout argument. While this is not instrumental solving the types of problems we have mentioned, it can be necessary to improve responsiveness of programs. For example one may use timed waits and in between the calls can check, for example whether the user has pressed the cancel button.

5.1.1 Example: the buffer implementation

As another example of using thread coordination, an implementation of the buffer is shown below.

```
class CoordinatedBuffer {
    int size,putPointer,takePointer,currentSize;
    Object[] objects;
    public CoordinatedBuffer(int maxSize) {
        size=maxSize;
        objects=new Object[size];
        putPointer=0;
        takePointer=0;
        currentSize=0;
    }
    synchronized int size() {return currentSize;}
    synchronized void put(Object x) throws InterruptedException{
        while (currentSize>=size)
            wait();
        objects[putPointer]=x;
        putPointer=(putPointer+1)%size;
        currentSize+=1;
        System.out.print("+");
        notifyAll();
    }
    synchronized Object get() throws InterruptedException{
        while (currentSize==0)
            wait();
```

```
        Object x=objects[takePointer];
        takePointer=(takePointer+1)%size;
        currentSize--;
        System.out.print("-");
        notifyAll();
        return x;
    }
}
```

The implementation still throws exceptions, but this time only related to multi-thread operation, not to operation on the buffer.

When an interrupt and a notify occurs about the same time, it is not deterministic which one has precedence.

5.2 Deadlocks(starvation) and livelocks

The example at the beginning of the chapter well demonstrated the situation called deadlock, in which two threads cannot proceed because each is awaiting for the other's actions due to improper use of coordination mechanism. Another situation which is called starvation occurs when some greedy thread calls a synchronized method -unnecessarily- too frequently so that other threads are practically blocked out from using the object's synchronized methods.

A different situation called livelock happens when a chain of threads trigger a closed circuit of actions. In such situations the threads are not blocked, but rather in a situation like the two (or more) threads are continuously greeting each other.

5.3 Profiling Java processes

Even if one is very careful, certain programming mistakes can lead to programs which suffer from deadlocks, starvation, livelocks, or unexpectedly bad performance. In such situations use of a profiling tool can prove very useful. Recent versions of the Sun Java SDK comes with one such tool called **jvisualvm**. Using this tool, one can examine properties of live Java processes and JVMs, such as their memory usage, or where (which methods) the program spends its time. You are recommended to use this or similar tools to solve performance problems which you cannot figure out the reason by looking at your -potentially large- program code.

5.4 Exercises

- E-1 Implement a counter (similar to quiz question) which is safe for multi-threaded usage, and whose value stays between 0 and a maximum given at construction time, inclusive.

Chapter 6

Thread pooling

Thread pooling refers to the general strategy of having a fixed number of threads working on a common work queue. The producer/consumer examples implemented in the above sections are essentially thread pool implementations.

Java SDK has several facilities to simplify thread pool implementations. The Java thread pool implementation is based on Executors, which are slightly more general than threads. An Executor has a single thread and accepts Runnable objects which are in turn executed one by one. A second interface, ExecutorService, corresponds to a pool of Executor objects which are managed as a pool. The advantage lies in the fact that when these mechanisms are used one does not need to use notifications, etc., explicitly hence simplifying implementations.

The code example below demonstrates use of Java thread pooling libraries to solve prime factorization problem. For further information consult <http://java.sun.com/docs/books/tutorial/essential/concurrency/pools.html>. Please note that the example program ignores the return value of submit() call, which is a Future class instance that can be used to track the status of submitted task. You are advised to read the ExecutorService interface documentation.

```
/**
 * Using thread pool libraries in Sun Java SDK to
 * find prime factors of given numbers in parallel
 *
 * Author: Mehmet Gencer, mgencer@cs.bilgi.edu.tr
 */

import java.io.*;
```

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

class Factorizer implements Runnable {
    int n;
    boolean isFinished;
    ArrayList factors;
    Factorizer(int n){
        this.n=n;
        isFinished=false;
        factors=new ArrayList();
    }
    static boolean isPrime(int i){
        for(int j=2;j<i;j=j+1)
            if (i%j==0)
                return false;
        return true;
    }
    public void run() {
        int m=n;
        for(int j=2;j<n;j=j+1)
            if (isPrime(j))
                while (m%j==0) {
                    factors.add(j);
                    m=m/j;
                }
        isFinished=true;
        System.out.println(String.format("Prime factors of %d are: %s",n,factors
    )
}

}

public class PrimeFactorsUsingThreadPools {
    public static void main(String[] args) {
        int poolSize=2; //default value
        ExecutorService pool;
        try {
            poolSize=Integer.parseInt(args[0]);
        }catch(Exception e) {
            System.out.println("You can change the default thread pool size by g
        }
    }
}
```

```

    pool=Executors.newFixedThreadPool(poolSize); //Create a thread pool of g
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in))
    for(;;) {
        try{
            String input=in.readLine(); //read a number
            int i=Integer.parseInt(input);
            pool.submit(new Factorizer(i));
        }
        catch (IOException e){break;} //break statement exits out of the for
        catch (NumberFormatException e) {break;}
    }
    pool.shutdown();
    try {
        System.out.println("Awaiting for the thread pool to finish...");
        if (!pool.awaitTermination(60, TimeUnit.SECONDS))
            System.err.println("Thread pool did not terminate in a fair time
    }catch(InterruptedException ignored) {}
    }
}

```

Java concurrency API also provides a `ScheduledExecutorService` which allows scheduled and/or periodic execution of tasks.

6.0.1 Choosing the thread pool size

For computation intensive tasks, ideal thread pool size is the number of processors on the system. For tasks which use file IO or other non-computational resources, there's no single answer to the question. Usually a benchmarking is necessary to decide. However one must always be cautious to prevent resource trashing. Resource trashing occurs when a CPU or other resource (disk, network IO, etc.) has too many requesters such that system's overhead to multiplex them puts an additional restraint that causes the system to cease functioning.

6.0.2 Short running tasks and Executors

In most of our examples in the above sections, worker threads were essentially long running. In other words they were designed to do a task repeatedly until terminated. Java executor services are designed to work with threads that are either long or short running. Executors themselves can be considered as long running threads. When long running threads are submitted as tasks

to executors, we arrive a situation where one executor runs only a single thread, hence no different than out home made thread pools. However, when submitted tasks are short running threads the executor will reuse its thread to run them one after the other.

`ScheduledExecutorInterface` is commonly meaningful for short running tasks. For example:

```
ScheduledExecutorService se=new ScheduledThreadPool(2);
se.scheduleAtFixedRate(new MyShortRunningTask(), 10, 10, SECONDS);
```

Periodic scheduling would not make sense above if the submitted task was long running.

6.0.3 Callables

All executor services accept `Callable` objects in addition to `Runnable` objects. `Callables` are methods that take no arguments. As an alternative to implementing `Runnable` interface and providing a method named `run()`, one can also pass a `Callable` instance which provides a method `call()`, to `Executors`. The difference is that the `call()` method is not void type and can return a result.

As a consequence, one can use the `Future` object returned by `Executor` submissions to query status of task and eventually retrieve the result of `call()` method.

```
class MyCallable implements Callable {
    int i;
    MyCallable(int i) {this.i=i;}
    int call() {return i*i;}
}
Future future = executor.submit(new MyCallable(2));
while (!future.isDone()) {}
System.out.println((int)future.get());
```

6.1 Exercises

- E-1 The method of merge sort is based on recursively splitting a list into two equal parts, sorting each separately, then merging them into a single list. Since each halves of the list is sorted independently, merge sort can be run in parallel.

- (a) Implement a merge sort class to sort an array of real numbers in parallel using a thread pool.
- (b) Test your merge sort class using various array length and pool sizes.

E-2 Consider the problem of finding the greatest common denominator of n natural numbers. Design and implement an algorithm to solve the problem. How can the algorithm be parallelized?

E-3 The number π can be computed using the following series form:

$$\pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)}$$

In practice the series can be computed to a certain threshold, instead of infinity. Write an implementation of the series. How can the algorithm be parallelized to use n threads?

Chapter 7

Distributed programming

With the increased availability of low cost computers, there is a strong motivation for distributing computationally intensive tasks over multiple computers. Such tasks include industrial problems such as ray-trace computer animation rendering, to scientific problems such as protein folding, or crypt-analysis.

7.1 Different standards

While memory sharing is possible in multiple threads working on the same hardware, distributed computing requires different techniques and involves different risks. Some methods for distributed computing are as follows:

Distributed RAM : A recent family of technologies which provides access to RAM on other machines on a local area network using high-speed network connections and hardware supported mechanisms. In principle D-RAM allows threads on different computers to share raw memory. Technologies under different names such as RDMA (Remote Direct Memory Access), DSM (Distributed Shared Memory), has appeared in recent years which provide similar services.

Network Attached Memory : NAM is somewhat similar to D-RAM except that it is implemented at the user software level, rather than hardware. Because of this NAM can be implemented across great distances, but has lower performance. Despite the performance drawback, however, NAM can be embedded into the programming environment, hence allowing remote threads to transparently share objects, rather than raw memory. A well known example is Terracotta system for Java (See <http://www.terracotta.org/>).

Remote Procedure Calls (RPC) : A traditional and robust method for delegating tasks to a remote computer is calling/invoking a method on the remote computer. Various names is used for this general technique. The Remote Procedure Call (RPC) method has been used for years. A recent standard, XML-RPC, allows language independent communication hence allowing different programming languages to be mixed in a distributed application.

Remote Object Invocation Started with the CORBA standard, this family of techniques are based on accessing objects in another computer's memory, to invoke their methods. The Java version of this family is named as Remote Method Invocation (RMI). This family of methods rely on a communication language to indicate name of remote object and its methods.

Message Passing : Method invocation involves calling a method and waiting for it to return a value. Message passing, on the other hand, is a much simpler task since it is only one way. The industry standard, Message Passing Interface, allows not only one-to-one (unicast), but also one-to-many (broadcast) messaging.

7.2 Requirements

Since processes on different computers cannot share a memory space, a basic requirement for distributed programming is the ability to refer to a procedure or an object at a remote computer. The former is required by the RPC technologies, and the latter by CORBA/RMI.

A second requirement concerns the passing of method parameters and return values between the two computers. In the case of RPC what needs to be passed is the name of the method to be called, its parameters, and in the other way, its return value. Standards like XML-RPC provides a language independent way for realizing this, contrary to earlier days of C specific RPC standard.

In the case of object oriented languages the problem becomes more interesting. It is possible to return objects from methods, or instead references to objects that reside in the remote computer. This preference has important consequences for the programs written, as we will see in the next chapter.

7.3 Topologies, and concurrency issues

One commonly hears the term ‘server/client architecture’ in distributed or networked processes. It indicates a topology in which a single server awaits requests from multiple clients, and responds accordingly. In that case the server usually needs to be a concurrent program (see also asynchronous socket programming).

However, there are several other topologies possible. One is in which several servers are controlled by a single entity. Another is when there is no single central of control, called peer-to-peer networks. This last topology is of common interest recently as it is able to tolerate problems in the nodes of a distributed system.

7.4 Potential problems

Despite its promise, distributed programming involves certain risks as different from multi-threaded programming:

- Performance of a distributed process is directly effected by the performance of underlying network. A well designed monitoring and adaptive network traffic management is necessary.
- Malfunctioning of remote processes are hard to detect and recover from. Task sharing and performance monitoring mechanisms are required for the distributed process to be fault tolerant.
- Data security risks must be accounted for and their solution may require encryption/decryption techniques which are computationally intensive themselves. Task sharing must be designed to minimize data transfer needs to remedy the problem. Furthermore, unauthorized or malicious procedure/method invocation is also a problem which needs to be addressed in distributed application security.

Chapter 8

Remote Method Invocation (RMI)

Sun Java SDK provides a library (`java.rmi`) to enable distributed programming, for which extensive tutorial and reference material available at <http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html>.

Invoking a method of an object which exists in a remote process requires some conditions to be satisfied:

1. Caller and callee must agree on the name of method, list and data type of parameters, and return value type.
2. If the remote method creates an exception, there must be a way to pass the exception relation information back.
3. Both parameters and return value should be suitable for object serialization so that they can be send through a network connection.
4. A certain network socket must be arranged, if necessary with security precautions such as SSL (Secure Sockets Layer).

To satisfy the first condition above, a common method in distributed Java applications is to use an interface definition. Java RMI dictates that such interface definitions extend the `Remote` interface in `java.rmi` library. An example interface is given in code example below:

```
/**
 * An RMI Interface definition
 * Author: Mehmet Gencer, mgencer@cs.bilgi.edu.tr
 */
import java.rmi.*;
```

```
import java.util.*;

public interface RMIExampleInterface extends Remote {
    /** Check and return whether the given integer is prime*/
    boolean isPrime(int n) throws RemoteException;

    /** Return a list of prime factors of given integer */
    ArrayList factorize(int n) throws RemoteException;

    /** Return the number of factorizations or primality checks
     * carried out by the remote object*/
    int jobCount() throws RemoteException;
}

```

The interface is similar other interfaces, except that it extends the ‘Remote’ interface. This signals the Java system that it is intended for RMI usage.

The method definitions in the interface uses a special exception class, `RemoteException`, from `java.rmi` library, as a solution to the second problem in our list. Since all parameter and return value types are already serializable, we are not concerned with the object serialization problem for this example. But if they were instances of classes we have written, we must make sure the class implements `Serializable` interface.

The interface definition must be shared by the RMI server and client program in order for them to communicate using a common language. The server and client implementations in code examples below both use the defined interface:

```
/**
 * An example RMI Server
 * Author: Mehmet Gencer, mgencer@cs.bilgi.edu.tr
 */
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.util.*;

class RMIExampleServerImplementation implements RMIExampleInterface{
    private int jobcount;
    public RMIExampleServerImplementation() throws RemoteException {
        jobcount=0;
    }
    public boolean isPrime(int n) throws RemoteException {

```

```

        if (n<=0)
            throw new RemoteException("Cannot check non-positive numbers");

        for(int j=2;j<n;j=j+1)
            if (n%j==0)
                return false;
        return true;
    }
    public ArrayList factorize(int n) throws RemoteException {
        if (n<=0)
            throw new RemoteException("Cannot factorize non-positive numbers");
        synchronized (this) {
            jobcount+=1;
        }
        System.console().format("Factorizing %d\n",n);
        ArrayList factors=new ArrayList();
        int m=n;
        for(int j=2;j<n;j=j+1)
            if (isPrime(j))
                while (m%j==0) {
                    factors.add(j);
                    m=m/j;
                }
        return factors;
    }
    public int jobCount() throws RemoteException{
        return jobcount;
    }
}

public class RMIExampleServer {
    public static void main(String[] args) {
        try {
            //Create the shared object
            RMIExampleServerImplementation object =
                new RMIExampleServerImplementation();
            //Export the object to RMI subsystem to be served on
            //a random suitable network socket
            RMIExampleInterface shared = (RMIExampleInterface)
                UnicastRemoteObject.exportObject(object, 0);

```

```
// Bind the remote object's stub in the registry so that
// socket can be found by callers and named object can be located
Registry registry = LocateRegistry.getRegistry();
registry.bind("Factorizer", shared);

        System.out.println("Server is ready");
    } catch (Exception e) {
        System.out.println(e);
    }
}
}

/**
 * An example RMI client
 */
import java.io.*;
import java.util.*;
import java.rmi.*;
import java.rmi.registry.*;

public class RMIExampleClient {
    public static void main(String[] args) {
        String host="";
        String objectname="";
        try {
            host=args[0];
            objectname=args[1];
        }catch(Exception e) {
            System.out.println("Usage:\n RMIExampleClient <server> <object>\n fo
            System.exit(0);
        }
        try{
            //Locate the serve and the object on it
            Registry registry = LocateRegistry.getRegistry(host);
            RMIExampleInterface server = (RMIExampleInterface)
                registry.lookup(objectname);
            BufferedReader in = new BufferedReader
                (new InputStreamReader(System.in));

            for(;;) {
                try{
```

```

        String input=in.readLine();
        int n=Integer.parseInt(input);
        ArrayList factors=server.factorize(n); //call remote method
        boolean isPrime=server.isPrime(n);    //call remote method
        System.console().format("%d is prime? :%s \nPrime factors of
    } catch (Exception e){
        System.out.println(e);
        break;
    }
}
System.console().format("Server jobcount: %d\n",server.jobCount());
    } catch (Exception e) {
        System.out.println(e);
    }
}
}
}

```

The class in the server program implements the `RMIExampleInterface`. An instance of the class is registered and published using the `UnicastRemoteObject` class from `java.rmi.server` library. This class provides the infrastructure with which the created class instance can be attached to a suitable network socket and called using RMI protocol. This class also provides constructors suitable for using a non-standard network port or security enabled (SSL) sockets for communication. We skip these features for the moment.

Before we can run any RMI server or client program, an instance of ‘`rmiregistry`’ program must be running on the system. This program extends capabilities of Java VMs to find and use RMI servers. Simply running the ‘`rmiregistry`’ program satisfies this condition. Once the program is running the server program can be started. The server program finds the running RMI registry process on the system using `LocateRegistry` class.

The client program, similar to the server, first finds the RMI registry process on the host computer. After that the named object on the RMI service is located using `registry.lookup()`. Once the remote object is obtained, as can be seen in the client program, the object can be used much like a local object.

8.1 RMI Concurrency

Java RMI subsystem has a concurrent implementation of its network services. It creates threads automatically as client requests arrive. Therefore it is necessary to implement mutual exclusion in objects shared via RMI.

8.2 Remote objects versus object transportation

The interesting thing about the RMI program above is that the server object resides at the server, whereas the client has a reference to it and can invoke its methods using this reference.

There are certain situations, however, which one wants to transport an object as the return value of a remote method invoked. This is possible if the returned object is an instance of `Serializable` (i.e. its class implements `Serializable` interface). However one must be very careful with this since although the code written for such objects look very much like the use of server object in the above client program, the outcome is very different. The returned object is a copy of the one in the server, hence its reference in the client program does not refer to the object at the server anymore! On the other hand objects which are instances of `Remote`, behave the other way around.

It is also a general strategy for application security to return `Remote` objects which are not published at the RMI registry to a client which provides sufficient credentials.

8.3 Exercises

- E-1 Consult the reference documentation of `ArrayList` class. How was that possible to return an instance of this class in the RMI program above?
- E-2 Modify the RMI program above so that only the clients which provide the correct username/password can get access to the factorizer.
- E-3 Implement a stock control system using RMI. Each control point in the system must login to the server before any stock operation can be made, and then make stock entries/exits by entering barcode of a good and number of items (plus for entries, minus for exits). The server method must fail if the stock is not sufficient. Propose a method to inform the client of such failures.

Chapter 9

XML-RPC

XML-RPC (Remote Procedure Call) is a language-independent standard protocol for executing remote procedures. It lacks object orientation of Java-specific RMI technology, but nevertheless provides similar capabilities.

The design priority in XMLRPC has always been language-independence. The standard is described in <http://www.xmlrpc.com/spec>. XML-RPC calls and responses are converted into XML coded language independent form as in the following example:

```
##### REQUEST #####
POST /RPC2 HTTP/1.0
User-Agent: Mozilla/1.0 (Ubuntu)
Host: cs.bilgi.edu.tr
Content-Type: text/xml
Content-length: 181

<?xml version="1.0"?>
<methodCall>
  <methodName>example.factorize</methodName>
  <params>
    <param>
      <value><i4>12</i4></value>
    </param>
  </params>
</methodCall>
##### RESPONSE (HTTP Headers omitted)#####
<?xml version="1.0"?>
<methodResponse>
  <params>
```

```

    <param>
      <array>
        <data>
          <value><i4>3</i4></value>
          <value><i4>2</i4></value>
          <value><i4>2</i4></value>
        </data>
      </array>
    </param>
  </params>
</methodResponse>
##### ERROR RESPONSE (HTTP Headers omitted)#####
<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value><int>4</int></value>
        </member>
        <member>
          <name>faultString</name>
          <value><string>Too many parameters.</string>
        </value>
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>

```

XMLRPC standard defines a limited, but nevertheless sufficient set of data types. Any `value` can be one of:

- `i4` or `int`: four-byte signed integer.
- `boolean`: `true/false` value.
- `double`: double precision, signed floating point number.
- `String`: a text string.
- `dateTime.iso8601`: date/time, e.g. “19980717T14:08:55”

- base64: base64 encoded binary data. This is useful if the data is a binary content such as a picture, etc.

These values can appear alone, or as part of `jarray` or `jstruct` constructs, as shown in the examples above. If different languages needs to be supported in strings, the character set encoding can be identified in XML tag. However since default for the Apache Java XMLRPC library is UTF8, this is rarely necessary.

Since XML-RPC data is in textual form, it is transported easily over HTTP protocol used for web pages. The term 'web services' is commonly used for web servers which provide XML-RPC based programmatic interface in addition to regular web pages service.

The Java support for XMLRPC comes from Apache project rather than standard Sun JDK (see <http://ws.apache.org/xmlrpc/>). The examples 9.1.1 through 9.1.4 exemplify server and client side XMLRPC with Java, with an additional client in Python. The class of object shared via XMLRPC must be declared public and for this reason put on a separate file. Unlike RMI, each client request causes an instance of this class to be created. For this reason the job counter in the example is declared as a static class variable rather than as an instance variable.

XMLRPC standard was later 'relaxed' to allow transfer of language-specific data types, by enabling the so called vendor extensions. These facilities are omitted in our examples since it seriously undermines the portability of the mechanism. Furthermore if one is to use Java specific facilities, RMI is a much better choice.

9.0.1 XMLRPC Introspection

XMLRPC allows a client to ask method signature, and even get method help from server. These extensions are called introspection extensions, and their use is exemplified in the code referred to above.

9.0.2 Asynchronous XMLRPC calls

The Apache library has a mechanism to make asynchronous calls, which can be utilized for algorithm parallelization purposes. See example 9.1.5. By using asynchronous callback objects, one can replace or augment thread pool-like techniques for parallelization.

9.1 Code Examples

9.1.1 XMLRPC Example - The shared object

XMLRPCShared.java

```
/**
 * A class to be shared over XMLRPC
 * Author: Mehmet Gencer, mgencer@cs.bilgi.edu.tr
 *
 * Since the class has to be public, it is put on a separate file
 */
import java.util.*;

public class XMLRPCShared{
    private static int jobCount;

    /** Return whether given number is prime*/
    public boolean isPrime(int i){
        for(int j=2;j<i;j=j+1)
            if (i%j==0)
                return false;
        return true;
    }

    /** Return prime factors of given number */
    //XMLRPC limits return value to some basic types in the standard
    // or objects or arrays of objects
    public Object[] primeFactors(int n) {
        System.console().format("Factorizing %d\n",n);
        ArrayList<Integer> factors=new ArrayList<Integer>();
        int m=n;
        for(int j=2;j<n;j=j+1)
            if (isPrime(j))
                while (m%j==0) {
                    factors.add(new Integer(j));
                    m=m/j;
                }
        synchronized (XMLRPCShared.class) {
            jobCount+=1;
        }
        System.console().format("Factors of %d: ",n);
    }
}
```

```

        System.out.println(factors);
        return factors.toArray();
    }

    public int getJobCount() {
        return jobCount;
    }
}

```

9.1.2 XMLRPC Example - Server

XMLRPCServerExample.java

```

/*
 * Simple XMLRPC server using Apache's XML-RPC library
 *
 * Author: Mehmet Gencer, mgencer@cs.bilgi.edu.tr
 *
 * This example uses builtin web server of Apache library.
 * Download and extract the library from http://ws.apache.org/xmlrpc/
 * This program is tested with version 3.1.1 of the Apache xmlrpc library.
 * Once you have the library extracted somewhere you must compile and run this code
 *
 * CLASSPATH=../xmlrpc-3.1.1/lib/commons-logging-1.1.jar:../xmlrpc-3.1.1/lib/ws-c
 * export CLASSPATH
 * javac XMLRPCServerExample.java
 * java XMLRPCServerExample 8000
 *
 */
import org.apache.xmlrpc.webserver.*;
import org.apache.xmlrpc.server.*;
import org.apache.xmlrpc.common.*;
import org.apache.xmlrpc.metadata.*;
import java.util.*;

public class XMLRPCServerExample{
    public static void main(String args[]){
        //Read the port number to use for XML-RPC enabled Web server
        int port=0;
        try {
            port=Integer.parseInt(args[0]);

```



```

* export CLASSPATH
* javac XMLRPCClientExample.java
* java XMLRPCClientExample http://localhost:8000/xmlrpc
*/

import java.util.*;
import java.net.*;
import org.apache.xmlrpc.*;
import org.apache.xmlrpc.client.*;

public class XMLRPCClientExample {
    public static void main(String[] args) throws Exception {
        String serverUrl="";
        try{
            serverUrl=args[0];
        }catch(Exception e) {
            System.out.println("Usage:\n XMLRPCClientExample <server-url>\n Wh
            System.exit(0);
        }
        XmlRpcClientConfigImpl config=new XmlRpcClientConfigImpl();
        config.setServerURL(new URL(serverUrl));
        //Following could be set if Java specific objects (eg. primitive types) :
        // the so called Vendor specific extensions
        //config.setEnabledForExtensions(true);
        //config.setContentLengthOptional(true);
        XmlRpcClient client= new XmlRpcClient();
        client.setConfig(config);
        try {
            // The parameters needed for call
            Object[] params = new Object[1];
            int n;
            for(;;) {
                try{
                    n=Integer.parseInt(System.console().readLine("Enter a number
                } catch(NumberFormatException e) {break;}
                //parameters are always an array and always contain Object types
                //so the following 'packaging' is necessary
                params[0]=new Integer(n);
                Object[] factors=(Object[]) client.execute("Factorizer.primeFact
                System.console().format("Factors of %d are: ",n);
                for (int i=0;i<factors.length;i++)

```

```

        System.console().format("%d ",((Integer)factors[i]).intValue()
        System.out.println();
    }
    //The following call has no parameters, hence an array of length 0
    System.console().format("Server job count: %d \n",client.execute("Fa
}catch(Exception e) {
    System.out.println(e);
}
}
}
}

```

9.1.4 XMLRPC Example - A Python client

XMLRPCClient.py

```

import sys,xmrlpplib

try:
    serverurl=sys.argv[1]
except:
    print "Usage:\n %s <server-url>"%sys.argv[0]
    sys.exit()
server = xmrlpplib.ServerProxy(serverurl) #create server connection
#Display method help
for method in server.system.listMethods():
    print "%s: %s\n %s"%(method, server.system.methodSignature(method), ser
while 1:#Repeatetly input from user
    try:n=input("Enter an integer:")
    except:
        print "Server job count:",server.Factorizer.getJobCount()
        break
    print "N is prime? ", server.Factorizer.isPrime(n)
    print "N factors: ", server.Factorizer.primeFactors(n)

```

9.1.5 XMLRPC Example - An Asynchronous Client

XMLRPCAsyncClient.java

```
/*
 * Asynchronous XMLRPC Client using Apache's XML-RPC library
 *
 * Author: Mehmet Gencer, mgencer@cs.bilgi.edu.tr
 *
 */
import java.util.*;
import java.net.*;
import org.apache.xmlrpc.*;
import org.apache.xmlrpc.client.*;

class MyCallback implements AsyncCallback {
    int n;
    boolean done;
    public MyCallback(int n) {
        this.n=n;
        done=false;
    }
    public void handleResult (XmlRpcRequest request, Object result) {
        System.console().format("Factors of %d are: ", n);
        Object[] results=(Object[]) result;
        for (int i=0;i<results.length;i++)
            System.console().format("%d ",((Integer)results[i]).intValue());
        System.out.println();
        done=true;
    }
    public void handleError (XmlRpcRequest request, Throwable error) {
        System.console().format("An exception is thrown when factorizing %d:\n %", n, error);
        done=true;
    }
}

public class XMLRPCAsyncClient{
    // this method returns a string
    public static void main(String args[]){
        String serverUrl="";
        try{
            serverUrl=args[0];
        }
    }
}
```

```
    }catch(Exception e) {
        System.out.println("Usage:\n XMLRPCAsyncClient <server-url>\n When
        System.exit(0);
    }
    try{
        XmlRpcClientConfigImpl config=new XmlRpcClientConfigImpl();
        config.setServerURL(new URL(serverUrl));
        XmlRpcClient client= new XmlRpcClient();
        client.setConfig(config);
        Object[] params = new Object[]{new Integer(1234)};
        MyCallback c1=new MyCallback(1234);
        client.executeAsync("Factorizer.primeFactors", params, c1);
        params=new Object[]{new Integer(12345)};
        MyCallback c2=new MyCallback(12345);
        client.executeAsync("Factorizer.primeFactors", params, c2);
        while (!(c1.done&& c2.done));
    }catch(Exception e){
        System.out.println(e);
    }
}
}
```

Chapter 10

Advanced topics

10.1 Architectures for distributed programming

The fact that one side of remote execution is passive leads to asymmetric server/client architecture in distributed programs. The term suggests a single server and multiple clients. However, realization of distributed execution commonly requires multiple servers and a single client. For this reason it would be more sensible to call this architecture as coordinator/servant. In this architecture it is imperative that the coordinator process be a multi-threaded one, since it must control multiple servants concurrently.

However, other architectures are possible. For example if servants are pro-actively requesting tasks instead of passively waiting for requests, the coordinator/servant architecture can be reversed. But one must keep in mind that delivery of results poses a problem in this situation.

Both types of architectures are sensitive to failures of the central host. More recently, cloud computing systems using un-centralized (or peer-to-peer) communication is being investigated as they are resistant to failures.

Regardless of the architecture, content secrecy and authorization issues must be addressed in data transport and access control mechanisms, respectively.

10.1.1 Exercises

E-4 Consider that each servant computer has more than one CPU in a distributed system. Propose an RMI based method to exploit such computing power.

10.2 Java network-attached-memory

An NAM solution is based on using remote objects seamlessly in a program. A Java based implementation called TerraCotta (see terracotta.org) provides such a platform to reduce development costs of implementing such an architecture. The solution is based on (1) wrapping both server and client JVMs in a modified JVM, and (2) declaring which objects should be shared/accessed on/from the server. The rest of the programs work without any modification.

10.3 Network Security

Any data that travels over the Internet poses a security threat if the content can be used to obtain otherwise illegitimate privileges. This is especially true if sensitive information, such as passwords, are sent in the clear. In recent years, security outbreaks, or even suspicions of such, has been damaging to credibility of IT solutions.

A widely supported method for data transport security is SSL (Secure Sockets Layer) which uses asymmetric public-key encryption. In public key encryption data encrypted with one key can be decrypted with a complementary key, and one of the keys is made public to avoid key exchange problems.

To avoid fraud in SSL security, security certificates are used which rely on trusted certificate authorities. Normally one should create a public/private key pair and send it to an authority (like VeriSign, or Tubitak) to get a valid certificate. Any certification chain ends in a widely accepted certificate authority is considered valid by programs like web browsers. Following examples instead create a 'self-signed' certificate, but nevertheless demonstrate steps in certificate management:

```
openssl genrsa -out privkey.pem
openssl req -new -key privkey.pem -out cert.csr
openssl req -new -x509 -key privkey.pem -out cacert.pem -days 1095
```

It may not always be possible or feasible to convert insecure application programs to secure ones. For example our RMI programs are insecure, and making them secure requires replacing underlying network sockets, a task which requires advanced network programming knowledge. In such cases secure tunneling programs, such as stunnel, can solve the problem:

```
### Make the private key and certificate available to stunnel in a single file
cat privkey.pem > stunnel.pem
echo >> stunnel.pem
```

```
cat cacert.pem >> stunnel.pem
## as a security precaution, make the file readable by owner only
chmod og-r stunnel.pem
## run stunnel
sudo stunnel -f -p stunnel.pem -d https -r cs.bilgi.edu.tr:http
```

Now try to connect to URL `https://localhost`.

If you want to use the above solution to protect RMI applications, you must fix the port number of your RMI service to a specific port and wrap stunnel around it.

10.4 Center-less architectures

All system architectures we have exemplified before has the common theme that the parts are designed to work together to form a grand design and as a result they rely on central coordination to function. On the other hand most real world phenomena demonstrates a different behavior in which complex systems emerge from behavior and interaction of uncoordinated parts. Agent based design is a recent approach to simulate, replace or support such processes. Appearing under a variety of names such as multi-agent programming, peer-to-peer networks, and sensor networks, this approach has proved useful in creating fault tolerant systems that resemble biological and social systems. Multi-agent approach has proved useful especially in simulating and experimenting with systems that are otherwise not easily accessible (e.g. biological systems, economic systems) or support decision making (e.g. emergency decision making simulations). Some well known examples of multi-agent systems are peer-to-peer file sharing networks, HeatBugs (biological system), distributed auction frameworks, etc.

The key element in agent based design and multi agent simulations is that agents have only local information and there is no central coordination which they rely upon. However, some simulation systems utilize global variables and registries to facilitate communication between agents.

There is a Java project, Jxta, which focuses on delivering a Java based platform for peer-to-peer application development. See [<http://www.jxta.org>]

Appendix A

A guide for coding style

Computer programs are not merely for compilers and interpreters. On the contrary, a more frequent type of access happens when we open the program for improving, or finding bugs, or when a team-mate does the same. Therefore, a program needs to be understandable when we need to recollect what we have done previously, or when it is the object of collective work. In this regard, programming is an art form, not only because it is an applied craft of mathematical abstraction, but also as the presentation of this abstraction in the form of a program is a piece of writing which should demonstrate a pleasant and readable style.

A first condition for coding style is consistency in using symbols and delimiting parts of program. When one writes literature, one follows certain established rules in using punctuation marks (i.e. periods at the end of sentence), capital letters (i.e. first letter of sentence, or private names start with capital letters), or demarcating text (i.e. starting new paragraphs or pages). Similar rules emerged through the programming practice in half a century of its existence. These rules are not universal, just as different national languages vary in their rules of writing them. Different programming languages, or some programmer communities can differ in the style rules they follow. In this guide we try to present most common rules used in the world of programming.

A.1 Demarcation

One of these established rules regards demarcation of different scopes in a program. This is established by using consistent indentation for program statements that are at the same level, and increasing indentation level as scopes are nested within one another. Following piece of Java code demon-

strates how this is applied:

```
int factorial (int n) {
    if (n<=1)
        return 1;
    else
        return n * (n-1);
}
```

In the above program the function scope is indented with four spaces. The nested scope of 'if' statement is indented further:

```
int factorial (int n) {
    --->if (n<=1)
    ----->return 1;
    --->else
    ----->return n * (n-1);
}
```

The program also demarcates parts of statements with spaces. For example if is written as below:

```
int factorial(int n){
if (n<=1) return 1; else return n*(n-1);}
```

it becomes considerably less appealing to its reader and very hard to understand.

There are minor variations in how this rule is applied. For example some prefer to place brackets encapsulating scopes as follows:

```
int factorial (int n)
{
    if (n<=1)
        return 1;
    else
        return n * (n-1);
}
```

Although this variant of the rule seems more logical, we will follow the first version of the rule, merely due to the fact that it is more common in the programming community.

A.2 Organization

Generally the data has precedence over the process. This is reflected in our second rule of placing variable declaration statements before other statements in the programs. For example in defining a Java class as follows:

```
class Complex {
    double real;
    double imaginary;

    Complex (double real, double imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    double getMagnitude () {
        return Math.sqrt(real*real + imaginary*imaginary);
    }
    ...
}
```

Different programming languages may follow further refinements in code organization. For example in Java programming, it is a common practice to place static members before non-static members, and constructor methods before any other method. Therefore on top of variables preceding methods, static variables must precede non-static ones, and static methods precede non-static ones but come after constructors. For example:

```
class Circle {
    static double PI = 3.14;
    double radius;

    Circle (double radius) {
        this.radius =radius;
    }

    static double area (Circle c) {
        return PI * c.radius * c.radius;
    }

    double area () {
        return area(this);
    }
    ...
}
```

A.3 Explanation

Each programmer have a different and unique approach, and even ones approach will be different in two year's time. Although how you do things may in many cases seem trivial to you, it is a good practice to explain key elements of your program in plain natural language. For this reason all programming languages provide a means of placing 'comments', lines or blocks of text which is marked to be excluded from program compilation or interpretation, but rather put for the human readers. Today it became a common practice to place short or long comments at top of program files, near classes, functions, or variables, and even within intermediate points inside functions to explain what is going on. The following example uses such comments extensively (some lines are split to fit on the page):

```
/** A program to draw Julia set fractals.
 * Author: Mehmet Gencer, mgencer@cs.bilgi.edu.tr
 * Created: 24 Nov 2008
 *
 * This program is free software and provided without any guarantees.
 * You can redistribute this program in original or modified form
 * provided that this copyright notice is preserved.
 */

/**
 * Complex class represents a complex number, and provides additional
 * methods for convergence checking in quadratic Julia sets.
 */
class Complex {
    double real;          // real part of the complex number
    double imaginary;    // imaginary part of the complex number

    ...

    /**
     * Return square of the complex number
     */
    Complex square () {
        double newr = real*real - imaginary*imaginary;
        double newi = real*imaginary*2;
        return new Complex(newr,newi);
    }
}
```

```

/**
 * Check whether a quadratic function in the form  $z_{i+1}=z_i^2+c$ 
 * diverges for given  $z$  when iterated. If the magnitude of  $z$ 
 * exceeds 'threshold' after at most 'maxIterations' number of iterations
 * it returns true, otherwise it returns false.
 */
static boolean checkDivergence
(Complex z, Complex c, double treshold, int maxIterations) {
    Complex znext; //variable to store next value in iteration
    //iterate up to maxIterations,
    // but if threshold is exceeded in the meantime, return true immediatel
    znext=z;
    for (int i=0; i < maxIterations; i = i+1) {
        znext=znext.square().add(c);
        if (znext.getMagnitude() > treshold)
            return true;
    }
    return false;
}
...

```

In the above example the comments at the top of the program briefs what program does, who wrote it and when, and what one can do with it. Each class and method is preceded with a long comment. All variables are succeeded with short comments. And the relatively longer method, `checkDivergence()`, has comments within the method body.

The above example demonstrates a very common way of code documentation, used in java and C/C++. Indeed there are various programs written to produce software reference documentation automatically from such code comments. However this rule also has variations. For example Python programmers place code comments *after* method or class definitions, an Python environment has documentation tools based on this.

A.4 Explication

Excessive use of comments may become a handicap to readability of the program. One remedy for this problem is to choose explicit, self-explanatory names for your methods or variables which eliminates further explanation in code comments. For example if the factorial function was declared as follows, it would definitely require code comments:

```
int f (int i) {  
    ...  
}
```

as the function name is nowhere near being self-explanatory. Long names may feel cumbersome at the beginning, but you'll come to appreciate their virtue as you re-visit your programs.

A.5 Naming conventions

There are various names in a program: for classes, for variables, for methods, etc. Also in related to explication, you may have to have variable or method names that are longer than a single word. There are competing trends in how one should choose names. Here we describe the most common methods in the Java programming world:

- Class or Interface names start with a capital letter, and interfaces are commonly prefixed with an 'I'. For example: Complex, IteratorInterface, LogicException, etc.
- Variable or method names start with a small case, but if it consists of multiple words first letter of each word is capitalized for ease of reading. For example: i, x, real, nextValue, getSize(), etc.
- Constants are in capital letters, and if they consist of multiple words the words are separated by underscores. For example: PI, MAX_LENGTH, etc.

Bibliography

Dijkstra, E. W. (1965, Sept). Solution of a problem in concurrent programming control. *Communications of the ACM* 8(9), 569.

Lea, D. (2000). *Concurrent programming in Java*. Addison Wesley.