Multithreading in .Net and Java: A Reality Check

Andrej Zentner[1]*, Robert Kudelić[2]

[1] Siemens, Republic of Croatia.
[2] Faculty of Organization and Informatics, Republic of Croatia.

* Corresponding author. Tel.: 385911221116; email: andrej.zentner@mscommunity.net
Manuscript submitted April 5, 2017; accepted July 17, 2017.
doi: 10.17706/jcp.13.4.426-441

Abstract: Due to the availability of powerful computers and improved parallelization algorithms, multithreading has become a valid choice for software development. Therefore, it is worthwhile to compare the currently most widely used development environments to determine which of them yields better performance in terms of multithreading. In this paper we tested the multithreading performance of .Net C# and Java, which are probably the two most frequently used languages for software development today. We were surprised by these results as we expected that C#, which is optimized only for Windows OS, would be faster. However, Java, although it is intended to be a programming language well-suited for different operating systems, proved to be faster in all the tests. The obtained results provide insight into the current state of optimization of each of the two languages and are also valuable in selecting the language to be used for programming today's complex software systems.

Key words: Java, multithreading, net c#, performance comparison.

1. Introduction

As an algorithm empirical efficiency can greatly depend upon a programming language in which an algorithm was implemented it is interesting to compare performance of different programming languages to see how they fare against each other. In [1] the author has compared seven programming languages to see how they would compare according to "program length, program effort, runtime efficiency, memory consumption and reliability". While Java was included the C# was not – multithreading was also not assessed. In [2] the authors have assessed performance of different programming languages by using standard bioinformatics algorithms. Assessment was conducted in six programming languages, Java and C# were included, but multithreading was not tested. Also, in [3] the author has presented empirical evidence that "there is little or no difference between the Java Virtual Machine and the .NET Common Language Runtime, as regards the compilation and execution of object-oriented programs." The author has made its tests on Java Grande benchmark suite, nevertheless all tests were single-threaded. There are also some other studies that are dealing with the same issues but from different angles and according to our knowledge not from the angle we are going to present, therefore for the sake of succinctness we will not unnecessarily go into such details.

Ergo, the goal of this paper is to shed some light on multithreading with regard to its functioning, implementation, advantages and disadvantages, and also provide a performance comparison between two probably most widely used programming languages today (.Net C# and Java). After an introductory overview of the basics of multithreading, we will show how multithreading is realized and implemented in

these languages as well as present the affordances of these languages for programmers in terms of multithreading.

Section 3 contains a detailed description of the performed tests along with the statistics for each of the programming languages.

In the final section, based on all the previous sections including test results, conclusions will be drawn regarding the multithreading realization and efficiency in each of the two programming languages. Multithreading is a computer science concept which encompasses coding a program and organizing code lines in a way that threads can be executed concurrently in the same process [4], [5]. That is, multithreading is the ability of a program or a process to control how it is being used by multiple users simultaneously as well as to manipulate multiple queries from a single user without the need to run several copies (instances) of the same program [4], [5].

Multithreading is most commonly found in multitasking operating systems. In single-core processors multithreading is realized through multitasking in a way that the processor switches the execution between several threads during task performance so that each thread is executed in a particular time slice. The rate at which this occurs is sufficiently fast to produce an illusion for the user of all the threads being executed at the same time.

In multithreaded processors threads can actually be performed concurrently, however, the aforementioned operating mode can also be used on each core, increasing the task execution efficiency.

Multithreading improves the performance of a computer program – its throughput, speed or responsiveness, or a combination of these. Using threads can enhance the performance of single-core processors by allowing the overlap between slow operations with much faster operations executed by the processor. Furthermore, multithreading prevents parts of the code from being blocked. This is most evident on the example of graphical user interface. If a program is only executed on a single thread, the user interface will get blocked. As a result, user satisfaction is diminished. Moreover, based on the non-responsiveness of the program, the operating system itself can interrupt program execution or propose to the user to do so. This problem, as well as many other similar issues, can be avoided by multithreading, so that the user interface and all the operations related to it are executed on one thread, and all the remaining logic operations on the other, or several other threads.

It has to be noted that optimal exploitability of multithreading primarily depends on the programmer. In that respect, care needs to be taken to ensure that the code is written in a way to avoid a situation in which two or more threads wait for each other to terminate a certain operation, or any other behavior of program threads which results in a deadlock [6]. Mutually exclusive operations also need to be taken into account to prevent the resources that are jointly used by threads in program execution from being accessed by two or more threads simultaneously.

Multithreading is also useful in cases in which one thread does not utilize all the computing resources, enabling another thread to utilize them in order to reduce the processor wait time. Multithreaded programs are complex to write and maintain. For one thing, debugging is much more demanding in such programs. Maintenance of multithreading and the concurrent (or seemingly concurrent) thread execution is far more difficult and can give rise to new issues in the program.

Program testing also becomes more complex since eventual problems are mainly connected with execution order and are much harder to detect and reproduce. Also, an existing computer program that does not utilize multithreading is rather difficult to modify so as to enable it to exploit all the advantages of multithreaded operation, and even when it does get modified, it needs to undertake the testing process.

2. Programming Languages Considered

.Net is an application framework that provides services for the development, publishing and execution of desktop, web and mobile applications and services. The .Net framework was created by Microsoft, with its first version released in 2002 [7]. The platform supports a considerable number of programming languages, including C#, Visual Basic, Visual C++, F#, and others, all of which are interoperable within this platform. The platform's program libraries are placed on top of Common Language Runtime (CLR) that constitutes the core of the platform. CLR is a virtual machine responsible for the management and execution of programs in the .Net platform.

CLR enables the translation of the written program code into executable machine code. In addition, it is used for memory management, error management, memory allocation and deallocation as well as thread management [8]. Owing to these capabilities, the programmer's productivity is greatly improved in terms of speed. Since its first release, .Net has been constantly improved through several versions. The currently used version is 4.5.2. (while version 4.6 has also been released).

In this paper, C# programming language will be used for code examples. C# is a higher-level programming language which offers a higher level of abstraction in comparison with machine code or lower-level languages. It makes complex programming simpler since the programmer does not have to manage processor registers, memory addresses and stacks. C# is an object-oriented programming language.

Multithreading has been a feature of .Net since the early versions of the platform. It is supported in all the programming languages of the .Net platform (C#, Visual C++, VB.Net, etc.). Multithreading is managed internally via the thread scheduler, which is a function that is typically delegated to the operating system by CLR. The thread scheduler ensures that all the active threads have a predefined execution time in terms of time slices in which a particular thread is to be executed. It also prevents the threads that are on hold or those that are blocked from using processor time [9].

Besides the thread scheduler, the operating system is responsible for switching thread context as well as for the thread status manipulation queries. For each thread the following elements are allocated by the operating system: registers, program counter, stack and stack pointer, time share and priority. In single-core processors the thread scheduler performs thread time-sharing by switching thread execution between all active threads in short time slices. In the Windows operating system a thread execution time slice is equal to about a tenth of a millisecond.

In multi-core processors multithreading is implemented through a combination of thread execution time-sharing and actual parallel thread execution at a particular time, wherein several threads are executed concurrently in different processor cores.

Thread time-sharing is still necessary in multi-core processors since the operating system is in charge of its own threads as well as of those pertaining to other operations. The total number of these threads exceeds the number of processor cores in which threads may be executed in parallel.

In the previous section we mentioned that threads, unlike processes, can share resources among themselves. Sharing resources also encompasses access to variables (fields) and methods (functions). In that respect, static methods and functions need to be differentiated from instance methods and functions. .Net enables three ways of synchronizing access to these fields and functions, as shown in Table 1 and Table 2. Synchronization can be automatic (performed by the .Net platform) or manual:

- Synchronized contexts (classes) – The [Synchronization] attribute is used to enable simple automatic synchronization only for instance fields and methods. Various threads can access these objects, but since all the objects are placed within the same context they share a common lock, allowing access to only one thread at any given moment.
- Synchronized code regions – Using the Monitor class for synchronizing access to particular blocks of code, instance methods and static methods.

- Manual synchronization – Various synchronization objects can be used by the programmer to create their own synchronization mechanism.

Table 1. Synchronizing Access to Arrays and Functions [10]

Category	Global arrays	Static arrays	Static methods	Instance arrays
Without synchronization	No	No	No	No
Synchronized context	No	No	No	Yes
Synchronized code regions	No	No	Only if marked	No
Manual synchronization	Yes	Yes	Yes	Yes

In .Net, the multithreading functionality is contained within the System. Threading namespace which allows all the classes and interfaces to be used to enable multithreading. In addition to thread activity and data access synchronization classes, the namespace includes the ThreadPool class which makes it possible to use the system thread pool. It also encompasses the Timer class which performs methods callbacks on threads in the ThreadPool as well as the Thread class which enables creation and access to particular threads. A thread created by means of the Thread class is called a child thread of the main thread. It also needs to be noted that the main thread is the one that is automatically created when the .Net execution is initiated. For details, including functions and properties, see [8]-[15].

Table 2. Synchronizing Access to Arrays and Functions [10]

Category	Instance methods	Blocks of code
Without synchronization	No	No
Synchronized context	Yes	No
Synchronized code regions	Only if marked	Only if marked
Manual synchronization	Yes	Yes

Java is an object-oriented programming language developed by James Gosling at Sun Microsystems which was later acquired by the Oracle Corporation. Consequently, all the licensing rights to use Java are currently owned by Oracle. The first version of Java was released in 1995 as an essential component of the Java platform [16], which is a set of software products and specifications that constitute a software development system. Java is platform-independent, which means that the code executed on one platform does not have to be recompiled before it can be executed on another. Java applications are compiled into binary code that can be run on any Java Virtual Machine (JVM) regardless of computer architecture.

It is owing to its portability and the possibility of using the written code on various platforms that Java is the most widely adopted programming language at the moment.

Java uses an automatic garbage collector, that is, a system for the memory management of a particular object during its lifecycle. While the object creation timing is done by the programmer, the Java platform itself is in charge of memory allocation and deallocation once the object is no longer used. The advantage of such an approach is that the programmer is not responsible for memory management, which allows them to engage more productively in writing the code.

As mentioned earlier, JVM is a process virtual machine that can execute Java binary code. JVM interprets the compiled Java binary code for the computer processor in order to enable it to perform Java program instructions.

The key principles of the Java programming language are simplicity, object-orientation, robustness and safety, architecture independence and portability, as well as high performance and dynamism [17].

Java programs may be run on anything from desktop computers and data centers to gaming consoles and supercomputers used for scientific research. Over 930 million copies of Java Runtime Environment (JRE)

are downloaded annually. There are around 3 million mobile devices on which Java is currently used [16].

Java has supported multithreading from its very beginning. Each Java thread has a priority which signalizes the order in which the threads are to be organized for execution to the operating system. Thread priorities range from minimum priority MIN_PRIORITY (1) to maximum priority MAX_PRIORITY (10). The initial value of each thread is NORM_PRIORITY (5) between the minimum and the maximum priority.

Each program in Java creates at least one thread – the main thread. Additional threads are created via the Thread constructor or by instantiating classes that are inherited from the Thread class. Thread implementation in Java is realized in these two ways:

1. Inheriting from the Thread class – java. lang. Thread.

2. Implementing the Runnable interface – java. lang. Runnable.

In Java, threads are managed by JVM. Most contemporary JVMs will use the threading model employed by the operating system since it enables better performance of the program itself. JVM organizes data of the Java program that is executed in several different executable data areas: one or more Java stacks, heap and method area. Within each JVM each thread is assigned a Java stack which contains data that cannot be accessed by any other thread, including local variables, parameters and values returned by each method called by a particular thread.

In JVM there is only one heap shared by all the threads that only contains objects. It is not possible to add primitive types or object references to the heap – they must be included in the object [18].

In addition to Java stacks and heap, JVM encompasses a method area. It contains all classes or static variables that are used by the program. All the variables in the method area are available and are shared among all threads, which differentiates it from stacks [18]. In order to coordinate access to shared memory among threads, each lock is associated with each class or object by JVM. If a certain object or a class is to be locked by a thread, the thread sends a query to JVM.

After a while, the object or the class is locked by JVM. Once the lock is no longer needed by the thread, it is returned to JVM. If there is a request for the same lock by another thread, JVM forwards if to that thread. During its lifecycle a thread goes through several states: its creation, initiation, execution and destruction.

There are several points of difference between inheriting from the Thread class and implementing the Runnable interface, which are explained below.

Inheriting from the Thread class will prevent the class that was inherited from the Thread class to inherit from another class since Java does not support multiple inheritance, which means that each class can inherit one and only one class. The advantage of this approach is a simpler program code structure. On the other hand, implementing the Runnable interface makes it possible for the class that has implemented the interface to inherit from another class as well. In contrast to inheritance, interface implementation is thus not limited to a single interface.

This method enables better object-oriented design and consistency, while also resulting in higher program code complexity, which is its potential disadvantage. To achieve basic thread functionality it is sufficient to implement the Runnable interface and extend the run() method. For more complex thread management and the utilization of methods such as suspend() or resume(), which are not available in the Runnable interface, however, the Inheriting from the Thread class approach is preferable.

The Runnable interface needs to be implemented by any class whose instances have been planned to be executed within a thread. The interface is designed in such a way that it provides a standard protocol for objects that will be engaged in code execution during the active state. In object-oriented programming languages, the term interface refers to an abstract type which contains neither code nor data and instead defines behaviors as methods signatures. An interface represents a type definition. Simply put, an interface is an agreement by which classes that implement it bind themselves to implement methods whose

signatures are mentioned in the interface. A single class can implement various interfaces.

A class must define a method without arguments of the run() type which is provided by the Runnable interface. The run() method is the entry point for a thread and also contains programming logic to be executed after the thread has been run.

3. Results and Discussion

For the purpose of comparing multithreading performance between .Net and Java, we will write three programs. The first program will create and start an arbitrary number of threads that will not execute any tasks. Instead, we will merely compare the thread creating and starting performance.

The second program will acquire data from several webpages and for each webpage a thread will be created in which data acquisition from a webpage will be performed. The program will be tested in several iterations.

The third program will be used for a simple simulation of increasing and decreasing the bank account balance. We will create four threads, each of which will access and modify, increase or decrease, the bank account balance. The goal is to provide a simple example that would illustrate the competition between threads over a particular resource. The program will be tested in several iterations.

The testing environment that will be used consists of a Pentium G860 @ 3GHz Dual-Core processor, 8 GB of RAM and the Windows 7 64-bit operating system.

3.1. Experiment 1 – Creating and Starting x Number of Threads

In this test we will write equivalent programs in C# and Java. The program will start and create 7,000 threads. The test will be run in 100 instances. The time required for creating and starting a thread will be measured and consequently recorded in the results table.

Table 3. Results of Experiment 1–Creating and Starting Threads

Programming language	No. of threads	No. of instances	$\bar{x}$ / s
C#	7,000	1	0.722
Java	7,000	1	0.616

The results in Table 3 indicate that Java was 17% faster in creating 7,000 threads by 0.106 seconds. In spite of that, it is evident that both programming languages show very good performance when thread creation is concerned.

3.2. Experiment 2 – Multithreading Web Data Acquisition

In this test we will perform a simulation of acquiring data from a web server. If there was only one thread that works with the data and the graphical interface, the graphical interface would be blocked from the user response until the data acquisition is completed. In case of a large amount of data, the acquisition would take some time during which the user would not be able to use the graphical interface.

We shall assume that the graphical interface is available and will only test the acquisition of data. The data will be acquired from four webpages, and for each of them one thread will be used. By using new threads to acquire data, the user interface and other functionalities of the application will be available upon the user response.

The program will be tested using 300 iterations. The duration of each iteration will be measured and in the end the average time of all the iterations will be taken as the reference time of the test execution.

Table 4. Results of Experiment 2 – Multithreading Web Data Acquisition

Programming languages	No. of instances	No. of web pages
C#	300	4
Java	300	4
Elapsed time of a test / min	$\bar{x}$ elapsed time of one instance / s	
5.104	1.020	
4.462	0.798	

As shown in Table 4, Java proved to be faster in this test as well. When the average time of executing an instance is concerned, Java was faster by 27.8 %, while in terms of the total execution time of the entire test Java exceeded .Net by 14 %. It has to be noted that the advantage of Java manifested in individual instances was not fully reflected in the total execution time. As a matter of fact, this finding is rather interesting as it shows that .Net, although it is slower, is more consistent when it comes to execution speed. This can probably be attributed to data caching, which is a distinct feature of .Net. It is also possible that during the test another task interfered which had some impact on test results. However, owing to the large number of test instances, the former seems to be a more likely explanation.

3.3. Experiment 3 – Thread Synchronization

The third test consists of a simple simulation of competition between threads over a distributed resource, in our case, a bank account balance. Each of the four created threads will try to increase the bank account balance, after which they will try to decrease it. Since at any given moment a bank account can be accessed by one thread only – otherwise it would occur that, while the bank account is being manipulated by one thread, another thread may start to perform its task on that same account, which is not a desirable situation – we will also show how threads can be synchronized.

In this concrete case we will use the keyword lock for thread synchronization in .Net and the keyword synchronized in Java. The program will be tested using 300 iterations and the obtained results will be shown in the results table.

Table 5. Results of Experiment 3 – Thread Synchronization

Programming language	No. of instances	No. of threads
C#	300	4
Java	300	4
$\bar{x}$ elapsed time of one instance / s	Elapsed time of a test / s	
0.018	5.395	
0.008	3.014	

The third experiment showed similar results to those in the previous two tests. Again, Java outperformed .Net C#. When it comes to the average elapsed time of one instance, Java was quicker by 125 %, whereas in terms of the overall time of the test Java was also more efficient, this time by 78.998 %. Even though, in absolute terms, as expected, none of the running times is large, especially those obtained for one instance, relatively speaking the demonstrated differences are quite large.

3.4. Stress Tests

Before making any conclusions we have decided that we will repeat previously described experiments but on a much larger scale – so we can be sure in the results. The experiment of creating and starting threads was this time conducted on 100 000 threads. The experiment of multithreading web data acquisition was this time conducted on 10 000 instances with 31 web pages – for each web page data acquisition, as before, we have created a new thread. Last experiment, thread synchronization, was conducted on 100 000 instances and on 30 threads. Results of these experiments can be seen in tables 6 – 8.

Table 6. Results of Stress Test Experiment 1–Creating and Starting Threads

Programming language	No. of threads	No. of instances
C#	100,000	1
Java	100,000	1
$\bar{x}$ / s	σ / s	
1175.182	0.0060	
9.781	0.0052	

If we take a closer look at these new experiment results, we can again see that Java consistently outperforms C# in every experiment. Experiment with creating and starting threads shows that Java outperforms C# by extremely large margin – Java was 99.16771 % faster, when we look average running time, with standard deviations very small for both languages. Experiment with multithreading web data acquisition again confirms that Java outperforms C# in that department as well – Java was 16.6756 % faster when we look average elapsed time of one instance and 16.6930 % faster when we look at elapsed time of a test, with standard deviation which is significantly lower than that of C#. Lastly, when we look at experiment with thread synchronization we see that Java outperforms C# significantly in every situation – especially with regards to elapsed time of a test where Java performs 98.4567 % faster than C#.

Table 7. Results of Stress Test Experiment 2–Multithreading Web Data Acquisition

Programming languages	No. of instances	No. of web pages	$\bar{x}$ elapsed time of one instance / s	Elapsed time of a test / min	σ / s
C#	10,000	31	3.730	621.726	1.144
Java	10,000	31	3.108	517.941	0.471

Table 8. Results of Stress Test Experiment 3–Thread Synchronization

Programming language	No. of instances	No. of threads	$\bar{x}$ elapsed time of one instance / s	Elapsed time of a test / min	σ / s
C#	100,000	30	0.253	421.962	0.0849
Java	100,000	30	0.0038	6.474	0.0004

4. Conclusion

The main goal of this research was to determine the most common multithreading capabilities of the two probably most widely used programming languages today – namely, .Net C# and Java. For this purpose we devised three experiments: thread creation, online data acquisition and thread synchronization. In all the three tests Java was more efficient. Results of the experiments are shown in section 3 – in tables 3 - 8 in particular.

The results were actually quite surprising. For one thing, it was reasonable to expect that the .Net framework would be faster than Java since .Net is specifically optimized for one platform, which, however, did not prove to be the case. Even though the Java programming language is designed to run on multiple platforms, which makes such an environment more difficult to design and implement, it proved to be faster in each test – and in some of them differed by very large amount of time (see tables 6-8). The findings of this research are quite interesting in themselves since they provide additional insight into both programming languages. More importantly, they are also relevant to both scientists and practitioners since the choice of a programming language can significantly impact developed algorithms and complex systems of various kinds.

Appendix

Appendix A

Creating and starting 7000 threads in C#.

```
using System;
using System.Diagnostics;
using System.Threading;
class Program {
static void Main(string[] args) {
//creating Stopwatch object for measuring time
Stopwatch vrijeme = new Stopwatch();
//starting time measuring
vrijeme.Start();
//loop with 7000 iterations
for (int i = 0; i < 7000; i++) {
//creating and starting a thread
Thread nit = new Thread(Prazno);
nit.Start();
}
//stopping time measuring
vrijeme.Stop();
//printing of elapsed time
Console.WriteLine(vrijeme.Elapsed);
Console.ReadKey();
}
//a method without programming logic – this method is called by threads
static void Prazno()
{ //nothing
}
}
```

Appendix B

Creating and starting 7000 threads in Java.

```
package printing;
public class NitTest {
public static void main(String[] args) { //creating Runnable object with extended run() method
Runnable runnable = new Runnable() { //method that is called on thread start
public void run() {
//nothing
}
}; //beginning of time measurement
long pocetak = System.nanoTime(); //loop with 7000 iterations
for (int i = 0; i < 7000; i++) { //creating and starting a thread
Thread nit = new Thread(runnable);
nit.start();
} //stopping time measurement
long kraj = System.nanoTime();
long ukupno = kraj - pocetak; // printing of elapsed time
System.out.print(ukupno);
} }
```

Appendix C

Multithreading web data acquisition in C#.

```
class Program
{
static void Main(string[] args)
{
//web site list for data acquisition
```

```
List<String> sites = new List<string>();
sites.Add("http://www.reddit.com");
sites.Add("http://www.yahoo.com");
sites.Add("http://www.index.hr/");
sites.Add("http://www.jutarnji.hr");
Thread nit;
//list of threads
List<Thread> niti = new List<Thread>();
//Stopwatch object for time measurement
Stopwatch vrijeme;
//TimeSpan object for measuring overall time
TimeSpan ukupno = new TimeSpan();
//number of times the test will be repeated
int j = 300;
Thread.Sleep(4000);
for (int i = 0; i < j; i++) {
Console.WriteLine("Početak testa {0}", i + 1);
Console.WriteLine("");
vrijeme = new Stopwatch();
//beginning of time measurement – one test
vrijeme.Start();
foreach (String site in sites)
{
Console.WriteLine("Kreirnje i pokretanje niti za {0}", site);
//for every web page we create new thread
nit = new Thread(new ParameterizedThreadStart(Preuzmi));
//created thread is added in list of threads
niti.Add(nit);
//a thread is started – passed argument is a web site
nit.Start(site);
}
foreach (Thread d in niti)
{
//we are waiting for all threads to finish – before resuming main thread
d.Join();
}
//stopping time measurement – one test
vrijeme.Stop();
Console.WriteLine("Preuzimanje podataka završeno.");
Console.WriteLine("Ukupno vrijeme preuzimanja: {0} sekundi", vrijeme.Elapsed);
Console.WriteLine();
//increasing overall elapsed time
ukupno += vrijeme.Elapsed;
}
//printing of results for all tests
Console.WriteLine();
Console.WriteLine("-----------------------------");
Console.WriteLine("Prosječno vrijeme preuzimanja podataka na {0} testova je {1} sekundi", j, ukupno.TotalSeconds);
Console.ReadLine();
}
//method that is called by newly created thread
static void Preuzmi(object oUrl)
{
string url = (string)oUrl;
WebClient wClient = new WebClient();
byte[] html = wClient.DownloadData(url);
Console.WriteLine("Preuzeto {0} byte-a sa {1}", html.Length, url);} }
```

Appendix D

Multithreading web data acquisition in Java.

```
public class NitiTest {
public static void main(String args[]) {
//list of web pages from which we will acquire data
ArrayList<String> sites = new ArrayList<String>();
sites.add("http://www.reddit.com");
sites.add("http://www.yahoo.com");
sites.add("http://www.index.hr/");
sites.add("http://www.jutarnji.hr");
Thread nit;
long pocetak = 0;
long kraj = 0;
//variable for measuring time
double ukupno = 0;
//variable for overall time measuring
double ukupnoTestovi = 0;
//number of time the test will be repeated
int j = 300;
//list of threads
ArrayList<Thread> niti = new ArrayList<Thread>();
try {
Thread.sleep(4000);
}
catch (InterruptedException e) {
e.printStackTrace();
}
for(int i = 0; i < j; i++) {
System.out.format("\nPocetak testa %d.", i + 1);
System.out.format("\n");
// beginning of time measurement – one test
pocetak = System.nanoTime();
for (String site : sites) {
//create new thread – for every web page
nit = new Thread(new Preuzmi(site));
//add created thread to a list of threads
niti.add(nit);
// a thread is started – passed argument is a web site
nit.start();
}
for (Thread d : niti) {
// we are waiting for all threads to finish – before resuming main thread
try {
d.join();
} catch (InterruptedException e) {
e.printStackTrace();
} }
// stopping time measurement – one test
kraj = System.nanoTime() - pocetak;
ukupno = ((double)kraj / 1000000000);
//increasing overall time during current test
ukupnoTestovi += ukupno;
System.out.println("\nPreuzimanje podataka završeno.");
System.out.format("\nUkupno vrijeme preuzimanja: %f sekundi.", ukupno);
System.out.println("\nTest završen.");
}
// printing of results for all tests
System.out.println("\n----------");
System.out.format("\nProsječno vrijeme preuzimanja podataka na %d testova je %f sekundi.", j, ukupnoTestovi);
} }
//a class that will be called by a newly created thread, this class inherits from class Thread
```

```
class Preuzmi extends Thread {
String url;
URL siteName;
InputStream inputStream;
DataInputStream dataInputStream;
//web page name is being passed to a constructor
Preuzmi(String pUrl) { url = pUrl; }
//extended method run() that will be executed during starting of thread
public void run() {
try {
siteName = new URL(url);
}
catch (MalformedURLException e) {
e.printStackTrace();
}
try {
inputStream = siteName.openStream();
} catch (IOException e) {
e.printStackTrace();
}
dataInputStream = new DataInputStream(new BufferedInputStream(inputStream));
int preuzeto = 0;
try {
preuzeto = dataInputStream.available();
} catch (IOException e) {
e.printStackTrace();
}
System.out.printf("\nPreuzeto " + preuzeto + " byte-a sa " + url);
try {
inputStream.close();
inputStream = null;
} catch (IOException e) {
e.printStackTrace();
}
try {
dataInputStream.close();
dataInputStream = null;
} catch (IOException e) {
e.printStackTrace();
} } }
```

Appendix E

Thread synchronization in C#.

```
class Program
{
static void Main(string[] args)
{
//list of threads
List<Thread> niti = new List<Thread>();
//creating of object Racun – this object will be used by threads
Racun racun = new Racun();
int j = 300;
Stopwatch vrijeme;
TimeSpan ukupno = new TimeSpan();
for (int k = 0; k < j; k++)
{
vrijeme = new Stopwatch();
vrijeme.Start();
//starting of threads
```

```
for (int i = 0; i < 4; i++)
{
Thread nit = new Thread(new ParameterizedThreadStart(ObaviPosao));
nit.Name = i.ToString();
niti.Add(nit);
nit.Start(racun);
}
//waiting for threads to finish
foreach (Thread d in niti)
{
d.Join();
}
niti.Clear();
vrijeme.Stop();
Console.WriteLine("Ukupno vrijeme instance: {0} sekundi", vrijeme.Elapsed);
Console.WriteLine();
//measuring of time
ukupno += vrijeme.Elapsed;
}
Console.WriteLine();
Console.WriteLine("-----------------------------");
Console.WriteLine("Ukupno vrijeme izvršavanja na {0} testova je {1} sekundi", j, ukupno.TotalSeconds);
Console.WriteLine("Ukupno vrijeme instance je {0} sekundi", ukupno.TotalSeconds / j);
Console.ReadLine();
}
//a method that is called by each thread
static void ObaviPosao(object r)
{
Racun racun = (Racun)r;
Random broj = new Random();
int i = 0;
//increment of account balance
i = broj.Next(0, 30000);
racun.PovecajStanjeRacuna(i);
//decrement of account balance
i = broj.Next(-30000, 0);
racun.SmanjiStanjeRacuna(i);
}
}
public class Racun
{
//resource for which threads are competing
public int StanjeRacuna { get; set; }
//method for account balance increment
public void PovecajStanjeRacuna(int iznos)
{
//locking resource – only one thread at a time can access resource
lock (this)
{
StanjeRacuna += iznos;
IspisiStanjeRacuna();
}
}
public void SmanjiStanjeRacuna(int iznos)
{
lock (this)
{
StanjeRacuna += iznos;
IspisiStanjeRacuna();
}
}
```

```
public void IspisiStanjeRacuna()
{
Console.WriteLine("Stanje računa je: {0}, nit: {1}", StanjeRacuna, Thread.CurrentThread.Name);
}
}
```

Appendix F

Thread synchronization in Java.

```
public class Klasa {
public static void main(String[] args) {
ArrayList<Thread> niti = new ArrayList<Thread>();
Racun racun = new Racun();
long pocetak = 0;
long kraj = 0;
//variable for measuring of time
double ukupno = 0;
//variable for measuring of overall time
double ukupnoTestovi = 0;
//number of time a test will be repeated
int j = 300; for(int k = 0; k < j; k++) {
pocetak = System.nanoTime();
for(int i = 0; i < 4; i++) {
//starting of threads
Thread nit = new Thread(new ObaviPosao(racun));
niti.add(nit);
nit.setName(Integer.toString(i));
nit.start();
}
for(Thread d : niti) {
try {
d.join();
} catch (InterruptedException e) {
e.printStackTrace();
} }
niti.clear();
kraj = System.nanoTime() - pocetak;
ukupno = ((double)kraj / 1000000000);
ukupnoTestovi += ukupno;
System.out.format("\nUkupno vrijeme instance: %f sekundi.", ukupno);
}
System.out.format("\nUkupno vrijeme izvršavanja %d testova je %f sekundi.", j, ukupnoTestovi);
System.out.format("\nProsječno vrijeme instance je %f sekundi.", ukupnoTestovi / j);
}
public class ObaviPosao extends Thread {
Racun racun;
ObaviPosao(Racun r) { racun = r; }
public void run() {
int i = 0;
i = (int) (Math.random() * (30000 - 0)) + 0;
racun.PovecajStanjeRacuna(i);
i = (int) (Math.random() * (0 + 30000)) - 30000;
racun.SmanjiStanjeRacuna(i);
} }
public class Racun
{
// locking resource – only one thread at a time can access resource
private int StanjeRacuna;
private int getStanjeRacuna() {
return StanjeRacuna;
```

```
}
private void setStanjeRacuna(int stanjeRacuna) {
StanjeRacuna += stanjeRacuna;
}
//method for increment of account balance
public synchronized void PovecajStanjeRacuna(int iznos) {
setStanjeRacuna(iznos);
IspisiStanjeRacuna();
}
// method for decrement of account balance
public synchronized void SmanjiStanjeRacuna(int iznos) {
setStanjeRacuna(iznos);
IspisiStanjeRacuna();
}
public void IspisiStanjeRacuna() {
System.out.println("Stanje računa: " + getStanjeRacuna() + " nit: " + Thread.currentThread().getName());
} }
```

References

[1] Lutz, P. (2000). An empirical comparison of seven programming languages, computing practices. *IEEE,* 23–29.

[2] Mathieu, F., & Michael, R. (2008). A comparison of common programming languages used in bioinformatics BMC bioinformatics.

[3] Jeremy, S. (2003). JVM versus CLR: A comparative study. *Proceedings of the 2nd International Symposium on Principles and Practice of Programming in Java* (pp. 167-169).

[4] Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers, 28(9)*, 690–691.

[5] Hewlet, P. (2013). Multi-threading: Concepts and application tuning. Retrieved from http://h20195.www2.hp.com/V2/GetPDF.aspx%2F4AA4-4916ENW

[6] Silberschatz, A., Peter, B., & Greg, G. (2005). *Operating System Principles* (7th ed.).

[7] Net Framework. (2015). Microsoft, Overview of the .Net Framework. Retrieved from http://msdn.microsoft.com/enus/library/zw4w595w%28v=vs.110%29.aspx

[8] Language Runtime. (2016). Microsoft, Common. Retrieved from http://msdn.microsoft.com/enus/library/8bs2ecf4%28v=vs.110%29.aspx

[9] Microsoft. (2016). Parallel processing and concurrency in the net framework. Retrieved from http://msdn.microsoft.com/en-us/library/hh156548%28v=vs.110%29.aspx

[10] Yazan, D. (2016). Multithreading Basics. Retrieved from http://www.diranieh.com/NETThreading/MultithreadingBasics.htm#Multithreading%20in%20.Net

[11] Microsoft. (2016). Thread Properties. Retrieved from http://msdn.microsoft.com/enus/library/system.threading.thread_properties%28v=vs.110%29.aspx

[12] Microsoft. (2016). ThreadPool Methods. Retrieved from http://msdn.microsoft.com/enus/library/system.threading.threadpool_methods%28v=vs.110%29.aspx

[13] Microsoft. (2016). Timer Methods. Retrieved from http://msdn.microsoft.com/enus/library/system.threading.timer_methods%28v=vs.110%29.aspx

[14] Microsoft. (2016). BackgroundWorker Properties. Retrieved from http://msdn.microsoft.com/enus/library/system.componentmodel.backgroundworker_properties%28v=vs.110%29.aspx

[15] Microsoft. (2016). Dispatcher Properties. Retrieved from http://msdn.microsoft.com/enus/library/system.windows.threading.dispatcher_properties%28v=vs.1

10%29.aspx
[16] Oracle Corporation. (2014). Java Timeline. Retrieved from http://oracle.com.edgesuite.Net/timeline/java/
[17] Principles of Java. (2014). Federal office for information security. Retrieved from https://www.bsi.bund.de/EN/Topics/OtherTopics/Java/javaprinciples.html
[18] Oracle Corporation. (2016). Java virtual machine specifications. Retrieved from http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html#jvms-1.2

Andrej Zentner was born in Osijek, Croatia, on 22 November, 1986. Shortly after getting his first persnal computer, he became interested in software, and especially in computer programming. He earned his bachelor's degree in information systems in 2014, from the Faculty of Organisation and Informatics, in Varaždin, at the University of Zagreb, in Republic of Croatia.

After his formal education he started working as a professional software developer in 2013, as a NET developer role, then he moved on to Java programming language and is currently working as an android developer.

Robert Kudelić was born in Zagreb, capital of Republic of Croatia, on 10 July, 1985. During his education he was always interested in computer science. In 2015 he earned his Ph.D., in information and communication sciences, from the Faculty of Organization and Informatics, in Varaždin, at the University of Zagreb, in Republic of Croatia.

In his research he is mainly preoccupied with programming, algorithms, combinatorial optimization, artificial intelligence and information systems. Therefore his publication track record is mainly in publications that are dealing with issues in these areas.

Kudelić is one of the founding members, and currently a member, of artificial intelligence laboratory at the Faculty of Organization and Informatics, he was also a member of IEEE computational intelligence society. At the moment, he is working at the Faculty of Organization and Informatics, in a Department for theoretical and applied foundations of information sciences.