Multi-Threaded Programming in Java®

EDAF85 Realtidssystem

Elin A. Topp, based on “Java-Based Real-Time Programming” by Klas Nilsson.
Revised for use with the standard Java API by Roger Henriksson.

2021-10-04

Preface

This document is merely a condensed version of the original book by Klas Nilsson ("Java-based real-time programming", first edition 2005, relevant revised version 2012) and aims to provide insight into the use of Java as a programming language in the realm of real-time systems, specifically for the course "EDAF85 Realtidssystem (Real-time systems)", taught at the Helsingborg Campus of Lund University during fall term. For the interested reader, the original document will still be available, however, this condensed version features only parts of the introduction and original chapter 3 (Multi-Threaded Programming in Java), just to make the student reader familiar with the specific tools applied throughout the course.

Lund, 2017-08-17, Elin A. Topp

From the fall 2020 only standard Java libraries are used to implement threads and thread synchronization in EDAF85 in contrast to earlier years when special API:s where required in order to facilitate cross compilation for proprietary embedded hardware. This decision has made it necessary to revise this document accordingly.

Lund, 2020-07-01, Roger Henriksson

Contents

1 Introduction

Goal: *Understanding of concurrently handled tasks and resources.*

Computers, such as microprocessors, are more and more often embedded in products that the user does not perceive as computers. This is the case for both consumer electronics (your TV set, mobile phone, etc.), home supplies (your micro-wave oven, washing machine, etc.), vehicles (the engine control in your car, airplane fly-by-wire control, etc.), building automation (elevator control, temperature control, etc.), as well as for industrial equipment. We want these things to be affordable, and we want to be able to trust their operation, in particular when our life depends on them. Additionally, in some cases, we want to be able to reconfigure, upgrade, or even program them, possibly by downloading software components from a trusted supplier via the Internet.

Essentially, we also want our systems to be able to react to certain events while already working on some other task, still producing correct (predictable) and timely results. Given the nice properties of the Java programming language, such as security and platform independence, we want to explore its possibilities for development of real-time control software, even for systems subject to severe demands on performance and predictability. Even though Java from the beginning was intended for programming embedded control devices, some industrially important control aspects were never dealt with fully within the standard Java platform. For demanding applications, specially adapted versions of the Java runtime system exist, but for illustrating all principles covered in the course, the standard Java API:s and runtime system will suffice.

To illustrate the notions of concurrency, real-time, and control (in the order mentioned) two simple application examples are given in the following.

Example 1: The LEGO®-brick machine

Computer control of a machine for manufacturing LEGO-bricks, as depicted in Figure 1.1, implies two requirements on the software:

1. The temperature of the plastic material should be measured and controlled periodically to prepare the material for casting with the proper quality. A too high temperature will cause the plastic to be damaged or even to start burning and a too low temperature will cause the plastic to set and block the ejection nozzle.

2. The piston of the cylinder should be controlled sequentially to actually perform the casting.

Note that the tasks according to items 1 and 2 need to be handled concurrently, and each task has to be carried out in real time. Otherwise the machine will not work.

Item 1 is a simple temperature regulator. Since reading the measured value (temp) and computing the control signal (doHeat) takes some time, such feedback control, when

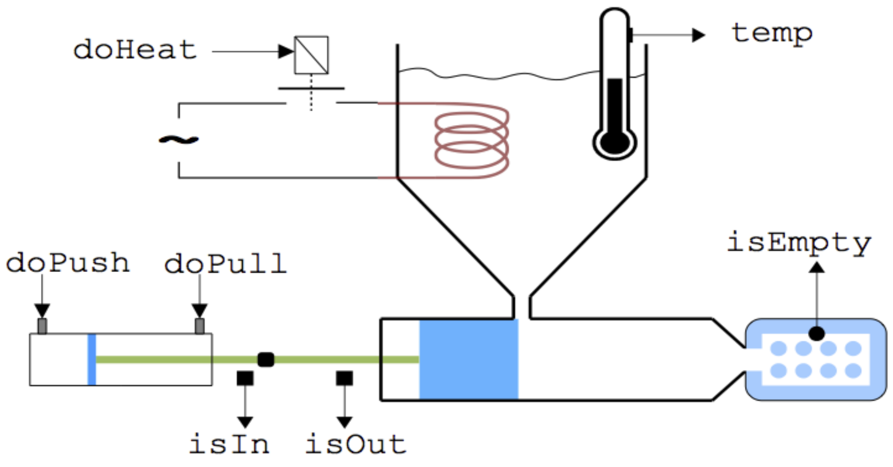


Figure 1.1: Schematic figure of a machine for manufacturing of LEGO bricks

done in software, by necessity has to be carried out periodically. In each period (also called a sample), the controlled system is measured (*sampled*), and the computed control signal is sent to the controlled system. For reasons of control performance, the time between each sample should not be too long. For computing (CPU-load) reasons, we want to have as long sampling period as possible without negatively affecting the controlled system. Hence, a tradeoff between control and computing efficiency has to be made. This type of control is *time driven* since it is run with respect to (real) time without waiting for external conditions/events.

Item 2 is an example of sequencing control, which is *event driven* (execution advances when certain events occur and arrive to the software); writing this as a periodic controller would imply unnecessary sampling and evaluation of the state of the application. On the other hand, if execution only advances on incoming events, how should then the temperature control be carried out (in the same program)?

The reader could try to write one sequential program performing both these tasks. Note that neither the temperature control nor the sequencing control may be delayed due to the dynamics of the other part. Furthermore, assume it should be simple (no reprogramming, only altering a data value) to change the sampling period of the temperature control. The resulting program will be both complex and difficult to maintain, so the goals are in practice incompatible.

A better way is to write two separate pieces of program, for instance two classes that handle the two control tasks. How should that be done, and what requirements does that put on the programming and run-time system?

Example 2: Bank-account transactions

A familiar example of concurrently occurring actions on the same data item, is bank account transactions. Assume that there are two transactions that are to be carried out at about the same time. One of them, which we call A, consists of taking out \$1000 from the account via an automated teller machine. The other transaction, which we call B, is to add a salary of \$10000 to the account. Depending on the actual timing, and depending on how the bank computer performs the transactions, we obtain different execution cases. If one transaction is performed before the other, we have one of the two cases shown in Figure 1.2. Each sequence of operations expresses the way the computer at the bank performs the requested transactions.

```
// Case 1: Withdraw first
A: Read 5000
A: Amount = 5000 - 1000
A: Write 4000
    B: Read 4000
    B: Amount = 4000 + 10000
    B: Write 14000
```

```
// Case 2: Salary first
    B: Read 5000
    B: Amount = 5000 + 10000
    B: Write 15000
A: Read 15000
A: Amount = 15000 - 1000
A: Write 14000
```

Figure 1.2: Two bank account transactions done in two different orders. In both cases, the original amount was 5000 and the resulting amount is 14000 as expected.

According to proper software engineering practice, the two sequences are preferably expressed separately as described in the previous example. Furthermore, assume that the run-time system of our computer is capable of interleaving the two sequences in such a way that they appear to be done simultaneously. Since the interleaving is not specified in the individual programs as shown in the figure, it could result in any order of operations depending on the underlying system, and it could possibly be different from time to time. For instance, we may for the above transactions obtain interleavings according to Figure 1.3. Even if automatic interleaving of code sequences often is desired to handle several tasks in parallel, we see that we must be able to specify a sequence of operations to be performed without being interrupted by other sequences manipulating the same (shared) data. We shall learn several ways to solve this type of problem.

```
// Timing 1:
A: Read 5000
    B: Read 5000
A: Amount = 5000 - 1000
    B: Amount = 5000 + 10000
A: Write 4000
    B: Write 15000
```

```
// Timing 2:
A: Read 5000
    B: Read 5000
    B: Amount = 5000 + 10000
    B: Write 15000
A: Amount = 5000 - 1000
A: Write 4000
```

Figure 1.3: Execution of the two bank account transactions interleaved in two different ways. With the Timing 1, the resulting amount is 15000, but the result happens to be 4000 in the second case. Hence, the final results are wrong and vary from time to time, which is clearly not acceptable.

The following section goes into more detail on how to use standard Java to solve issues like the ones mentioned in this introduction.

Already having some programming experience means that you, dear reader, have a lot of knowledge about how a computer interprets or executes a computer program. When you know it, it is all natural, and most programmers are not aware of the insights they have gained. We are now about to write programs that behave concurrently even if we only have one processor. Before doing so, it could be a good idea to take a step back and review the basic properties of program execution. This is the first thing that will be done in this chapter. Then, different notions of concurrency in hardware and software will be treated. Finally, we state the duties of the software systems we are about to develop.

The eager and experienced programmer may skip this chapter, but beware, you may be a very skilled programmer without having the profound understanding you think you have.

```
int x = 2;
while (x>1) {
    if (x==10) x = 0;
    x++;
};
/* What is x now? */
```

Figure 1.4: Code snippet illustrating sequential execution, trivial to any programmer but many beginners actually give the wrong answer 0.

1 Software execution is performed sequentially

To recall the sequential nature of program execution, assume you write an ordinary program to be run on an ordinary computer having one processor (CPU). A piece of the code may look as shown in Figure 1.4, so what value will x have after leaving the loop?

You know that the resulting value of x will be one. The novice answering zero believes that the loop-condition is continuously evaluated in parallel with evaluation of the expressions in the loop. Thus, the fact that the computer treats our instructions sequentially one after another cannot be considered natural to all of us; some of us have just learned that this is the way it works. Actually, it is a severe restriction that only one thing at a time can be done. Even if computers today are computing quickly enough to do the computations required by a parallel environment (like our real world).

So, when learning to program, we are first forced into the 'narrow alley' of sequential execution. Compare this with hardware design where each signal or information flow usually is taken care of by dedicated components ensuring continuous and timely operation. But for reasons of cost and flexibility, we cannot implement everything in hardware, so the problem that we are facing now is that the sequential software execution model does not fit the inherently parallel problems we want to solve. Even when solving parallel problems, however, some things are still sequential to their nature (like a sequence of control actions to be carried out in the correct order). Then, for such parts of the program, the sequential execution of software will be of great convenience.

2 Our physical world is parallel

The object oriented programming (OOP) paradigm, which we want to use for embedded programming, is based on creating software entities modelling our environment. Furthermore, since we aim at computer control of physical devices, some thoughts about the environment to be controlled may be appropriate[1].

As we all know, our physical world is inherently parallel. Mathematically, we need sets of coupled differential equations to describe it. Comparing such mathematical expressions with the expressions found in programming languages such as Java, they are not at all expressing the same thing. Basically, this is related to another beginners problem in programming; the notion of equality. When an mathematical expression states that x=y+2, that is an equality which should hold for whatever value x (or y) has. When we in a Java program write x=y+2, we instruct the computer to take the vale of y, add 2, and to store that in x. If we in Java write x==y+2, we test if the value of x equals y+2. The mathematical equality cannot be directly expressed in Java (or in any other widely used

[1]This section (1.2) aims at deeper understanding or wider perspectives, but can be skipped by the reader only caring about practical programming issues.

programming language like C, C++, etc.). Again, computers operate sequential from a programmers point of view.

For a single (scalar) differential equation, we then have an even harder type of equality. Apart from the previously mentioned problem, we have to compute derivatives of variables (representing physical states) and then we also get truncation and quantification effects due to the binary representation of data with finite wordlength. Therefore, we can only work with approximations of the real world object, even if its properties would be known exactly.

For a set of coupled differential equations, modelling our real-world objects, the sequencing effect is an even bigger problem since all equations are to hold in parallel. Additionally, state changes may occur at discrete times which cannot be exactly represented, and our models may include logical relations. To overcome these problems, the following techniques are used within systems engineering:

1. For less complex dynamics, equations may be solved and only the explicit solution need to be implemented. For instance, a real-time computer game for playing "pinball" includes simulation of the ball dynamics, which can be expressed as a simple type of numeric integration.

2. For systems with low demands on accuracy or performance, we may approximate or even omit the model equations. For instance, for soft-stop control of elevator motions we may use manually-tuned open-loop control without any internal model. This is opposed to optimal performance robot motions which require extensive evaluation of motion dynamics.

3. We may use special purpose languages and execution systems that transforms declarative descriptions to computable expressions, i.e., a combination of software technology and numerical analysis. For instance, it is well known within the software oriented part of the control community that object oriented models of dynamic properties requires declarative languages such as Modelica. Another example is that certain types of logical relations may be conveniently expressed in Prolog.

The computational difficulties are taken care of at 'system design time', which means that we use high performance workstations to analyze, simulate, and to determine a control algorithm that is simple to compute in the embedded system. For instance, based on the equations and a dynamic model of a system (like dynamic properties of a vehicle) we may use simulation tools to compute the behaviour (like rotation, position and velocity) when certain actions are applied (like speed and steering wheel commands), and some control design tool help us determine a control law expression which can easily be implemented and computed in the embedded software.

The conclusion from this section is that our real physical world can be quite troublesome from a programming point of view. Therefore, we should not apply object orientation and imperative languages (such as Java) outside their conceptual limits, and we should design our systems so that algorithmic parts (possibly needing special treatment as mentioned) are encapsulated in classes in such a way that they can be developed separately. And fortunately, that separate treatment can usually be carried out at design time.

We are then left with the much simpler task of standard concurrent and real-time programming which is the topic of this book. In other words, if we exclude advanced systems like certain types of autonomous vehicles and self-learning robots, the parallelity of the physical world is not a problem when programming embedded systems since we only communicate with the environment via a finite number of input-output (IO) connections with discrete (limited A/D and D/A resolution) values and in discrete time (sampled when the IO device driver is run).

3 Parallel computing

To meet the needs of high-end computing, computer hardware development continuously tries to improve computing performance. One popular research approach for the last twenty years has then been to build parallel computers. That is, computers that contain many processors working in parallel. Today, late nineties, parallel computers with up to 64 processors (CPUs) of the same type sharing a common memory has made it to the market. Then, what are the implications for concurrent programming of embedded systems?First, we should know that the mainstream industrial development of embedded computers is based on either single CPUs, or multiple CPUs that are separately programmed because they contain different (loosely connected) software modules. Modern CPUs are powerful enough for at least specific and encapsulated parts of the software functions. Hence, programming is still carried out on a single CPU basis, possibly using explicit interfaces between multiple processors.

- The parallellity of real-world models, as described in previous section, shows up in technical computing. Some types of problems can be decomposed into computations that can be done in parallel with each other. However, this is only worthwhile for special problems, and then mainly during system design. There are no indications that parallel computers requiring special programming techniques will be widely used for embedded computing. Exceptions may be applications like radar echo processing and computer vision. Such inherently parallel applications are important and demanding, but it is outside the scope of this book since it employs few programmers.

- The ordinary sequential programming model is so established that computer vendors have to stay with it. Therefore, the parallelity of the hardware is taken care of by special compilers and run-time systems, still maintaining an unchanged view to the programmer. There is even a trend that the parallelity is fully taken care of by the hardware. Hardware developers talk about the "sequence control barrier" which means that the hardware parallelity may not violate the sequencing property of the software execution as described in Section1. Thus, even if we should use a PC running Windows on four CPUs, we do not have to care about that when writing our software.

In conclusion, the sequential execution model described earlier is almost always valid even when parallel computers are used. Comments about software interfaces between different processors will be given in a few places in this book, but generally we should think in terms of using a single CPU.

4 Concurrency

Our software applications interact with users and external equipment via input/output interfaces, generally called the IO. The computer receives input data concurrently from sensors and from user input devices like keyboard, mouse, and microphone, as depicted in Figure 1.5. From the user we talk about input events, and from the sensors we talk about samples. Both input events and samples reach our program in form of some kind of data entity or record. For a mouse click, for example, the event typically contains the horizontal

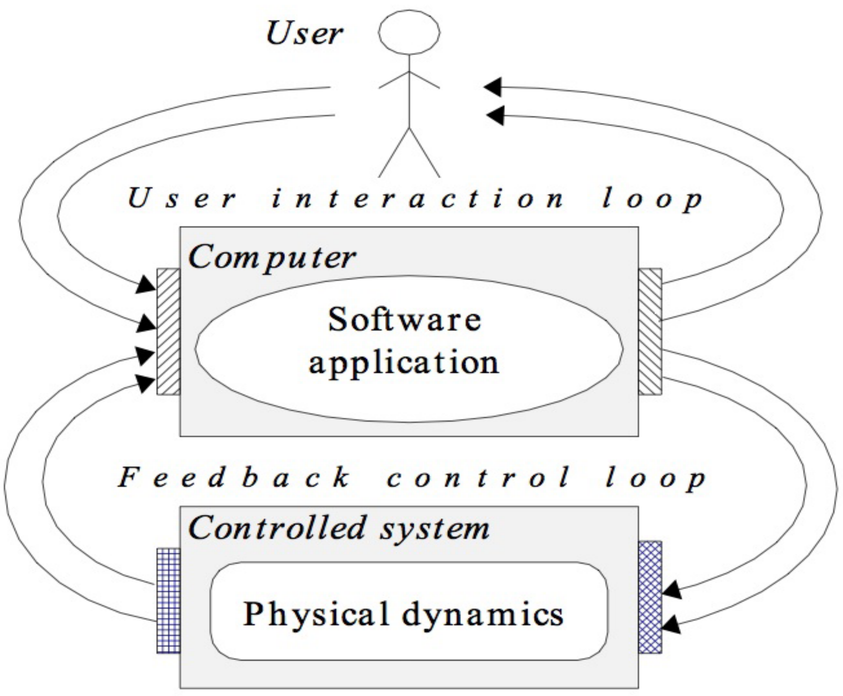


Figure 1.5: Input in terms of user commands/actions and sampled measurements must be concurrently and timely handled by our software. (Re)computed outputs for user perception and control actuation form two feedback loops.

and vertical coordinates on the screen and the time when the click occurred. Sampled physical values for feedback control are usually obtained via dedicated IO interfaces with well defined timing and buffering; extra delays in terms of unforeseen buffering could cause the feedback control to be unstable. There is no clear distinction between the two loops in Figure 1.5 (for instance real-time games only have one type of loop and physics is directly available to the user/driver in vehicles), but it is normally good to be aware of it for engineering purposes. Assume we only know ordinary sequential programming. Even if we use object oriented design and programming, with classes and objects defined according to real-world objects, methods are sequential and called sequentially so we still have the same situation as commented in the beginning of this chapter. Physically, all input may occur in parallel, and we can generally not make any assumptions about the resulting order of the data that is input to our program. The problem with pure sequential programming is that the programmer must define in what order computing should be done, but it is not known in what order input data will arrive, or the program has to waste CPU time on excessive polling and evaluation of conditions that may influence the output.

For example, reconsider the LEGO-brick machine. Assume we have implemented the control program as one loop in which all IO and control is carried out sequentially. Then consider what the changes will be if, say, the sampling frequency of the temperature control is substantially changed. Since the programmer has to manually do the interleaving of the code for the different activities, changes in concurrency or timing have severe impact on the source code. As mentioned, a more attractive approach would be to have the temperature control expressed well separated from the sequencing control of the piston, which also would be natural from an object-oriented programming point of view. The problem is, however, that object orientation by itself does not support concurrency.

The same type of problems show up in interactive software. For example, an applet containing a web animation and a button to select if the animation should be played forward or backward is for the purposes of this treatment equivalent to the LEGO-brick machine. Animation with a certain frame rate corresponds to the periodic temperature control, and handling of the push button corresponds to handling the position sensor. For the purposes of this book, however, we will mainly consider control applications which are subject to more severe demands in later chapters. When we know how to cope with both concurrency and real time for control applications, we will also be able to handle (the simpler) interactive systems.

Behind the term concurrent programming is the fact that our software has to handle concurrently occurring external events. But before making a definition of the term, let us review some alternative approaches to handle the situation.

4.1 Event processing

In the simplest case, each input event can be immediately and fully handled before any following events must be taken care of. Then our program may consist of only data structures, so called *event handlers* which perform the processing of each event, and a so called *dispatcher* that calls the appropriate event handler for each incoming event. There would, for example, be one event handler taking care of mouse clicks which then would result in changes of the data structures, usually reflected on the screen. The dispatcher is called in a simple loop which thereby is 'driving' the application.

The loop retrieving the events (or messages) and calling the handlers is (in Microsoft terms) called the message pump. When programming user interfaces using popular class libraries like MFC (Microsoft Foundation Classes), AWT (the Java-based Abstract Window Toolkit providing an interface to native graphics and window systems), or JFC (Java Foundation Classes, now known as Swing), the message pump is hidden inside the class library. Application programming then means writing the event handlers/listeners and registering them to the system.

Event processing is easy to program but in most applications, and particularly in control applications like our LEGO-machine, this execution model alone is not sufficient. The reasons are:

- Our single control loop will consume all available CPU time for the successive testing (called polling, or busy wait if nothing else is being done). This is waste of processing power which prevents the computer from simultaneously being used for other tasks.

- Most control functions, but not this one, need a notion of time to work correctly. For example, if there would have been disturbances on our measurement signal, or if we would need to predict the temperature some time ahead, sampling and computing would require proper and known timing

The straightforward solution now is to convert our control function into a form suitable for event processing, and to have control events periodically generated with a certain time period. This is actually a useful method for very small software systems, like software for so called micro-controllers. For our example application this means that we should:

1. Remove the loop statement around the control algorithm and the waiting for next sample to simply get a function computing one new control output based on one single input event.

2. Connect some external timer hardware generating periodic time events to the program. This is built into so called micro-controller chips, and available on (almost) any computer board. The timer should be set up to expire often enough to obtain good control, but seldom enough not to waste CPU time. (To find an appropriate value is a control engineering task which is not covered.)

However, such an interrupt/event-driven technique does not scale up very well. Imagine a control computer handling many control loops with different sampling periods, that would require non-standard timer hardware, and the dispatching of timer events would be troublesome. Further disadvantages will be mentioned below. Thus, event processing as a general mechanism is mainly useful when developing user interfaces or computing functions that can run to completion and return whenever triggered (by time, mouse clicks, etc.). Here we also need other more powerful concepts.

4.2 Time and encapsulation

The main limitation with purely event-driven software is that the processing must be completed immediately not to delay processing of the subsequent events. When the required computing takes to long time to complete, we need some means to schedule to work to be done. For instance, the user pushing a button in a web-based application may results in a request for information to be downloaded via Internet. Clearly, we need some means to let the event handler start (or give data to) an ongoing activity taking care of the loading. That activity then needs to be interruptible by other events, and possibly by more important ongoing activities issued by those events.

Another reason that we need ongoing, and interruptible, activities is software quality. It is fundamental both within OOP and programming in general to keep information locally whenever possible, and to let the program clearly express the task to be performed. Consider a control loop such as the temperature control of the LEGO-brick machine. Putting the control statements (including timing requirements) in a loop within a method of the control object is clearly better than having the timing requirements registered in one part of the program and the control actions coded in another part. So again, we want to be able to express our computations and control algorithms in an object-oriented manner as we are used to, and have a run-time system that takes care of scheduling the different activities.

In Java applets the event handling is performed in functions like action and handleEvent, whereas ongoing activities are (or at least should be) done in functions like run. But before we learn how to design and program such functions, there are some things we should observe.

4.3 Programming of parallel sequences

The functionality we are aiming at resembles that of an operating system; several programs that run at the same time should be scheduled to share the CPU in order to let each of the programs behave timely. However, the concurrency of complete programs is another topic handled later in this book. For now, we assume that the concurrency stems from the parallelism of a single application or program. It is then a big advantage that separately programmed concurrent activities can share and operate on the same data structures or objects. Otherwise, we would need additional files, pipes, sockets, etc. to exchange data, and that would be slower and much more work to program. Furthermore, assume that our program containing the concurrently executing activities is the only program we are running. (We cannot make assumptions about other programs anyway, and when we load our software into an embedded computer this assumption is normally fulfilled.)

When an ordinary sequential program executes, from entering the main program until the final end, there is at each time one specific point of execution. We refer to that point in terms of a program counter (PC). We have so far realized that we require the program to be concurrently ongoing, typically executing in several objects simultaneously. That is, the program has to switch between its concurrent activities in such a way that, with the time scale of the application at hand, it appears (to the user and to the controlled system) to run in parallel (on different CPUs). To achieve this, the run-time system has to provide:

1. A function for creating a new concurrently executing activity, i.e., an additional PC and the states comprising the context associated with the point of execution.
2. A scheduler that provides sharing of the CPU in common by interleaving the executions. This should be without requiring the source code to be cluttered with statements not modeling the task to be performed.
3. When concurrent activities share the same data or is dependent on each others computational results, there must be some means of synchronization and communication.

The need for a scheduler is related to the obligations of the operating system for running complete programs, but as mentioned, here we are concerned with the parallel behavior of one program. Assuming that our system is equipped with such a scheduler, interlacing the different points of execution, we may define the following requirement for a concurrent program to be correct:

Definition: The correctness of a concurrent program does not only require each programmed sequence of computation to be correct, the correct result must also be obtained for all possible interleavings.

In terms of the bank example this means that money should not be (virtually) lost or created due to the timing of transactions and computer operations. Even if that is an obvious requirement, it is actually a quite severe demand when it comes to verification of the correctness and finding any faults; the task we call test and debugging. Debugging an ordinary sequential program can be done by running the program with all possible sets of input data, and one set of input data gives the same result if the program is run again. For a concurrent program, however, the interleaving depends on external events triggering different sequences of scheduling and execution. With a concurrent program, we can usually not expect to obtain the same execution order twice; an input event occurring just a microsecond later may result in another interleaving which may course an incorrect program to crash. That in turn must not happen for a safety critical application. Hence, we may have severe requirements on correctness but complete testing is almost never possible.

It is interesting to note that the definition of correctness does not say anything about time or timing, but timing is both the main reason for introducing concurrency and the key aspect for the behavior of an incorrect program. Considering the difficulties writing and testing concurrent software, our hope now is that we can find and follow certain rules for development of such software.

5 Interrupts, pre-emption, and reentrance

Just like we did not allow the user program to poll the inputs for possible events (which would be waste of computing power when no new event has arrived), the same applies to the scheduler or scheduling. The invocation of the scheduler and its implications for the software application deserve some comments.

5.1 Obtaining the time base

As mentioned, we resort to using hardware interrupts to trigger our software functions, even if the interrupt service routines as such are hidden inside the operating system and device drivers. Of course, we may have input signals which do not generate any interrupt, but then we read that value at certain times, usually periodically when the data is used for feedback control. Therefore, the scheduler should also be able to handle time requests, such as "sleep one second" which then should suspend execution for that activity during that period of time.

Hence, our computer must be equipped with some kind of timer interrupt occurring periodically, thereby providing a time base for the software. Of course, the time in software will then be discrete (incremented one step for each timer interrupt) opposed to the true time which is continuous. Discrete time is, however, inherent to computer control anyway so it suits our purposes well.

Note that if we were only simulating our system, the time variable could be stepwise increased by the scheduler; when all processing is done for one point of time, the scheduler simply lets time jump to the next time referred to by the application. But here we aim at software that interacts with the physical environment which (by the laws of nature) has time as an independent and continuously increasing variable. The time variable in our software is the sampled value of the real time.

The timer interrupt actually turns out to be a general and minimum requirement for both desktop and control computers. Additionally, there are some low-level and system-specific events that require particularly timely service. For instance, the Windows-95/98 user may review his/her IRQ settings, typically finding them allocated by the sound card, the game port, the hard disk, etc. (The settings are found by selecting IRQ in the panel found by clicking on Control panel -> System -> Device Manager -> Computer -> Properties.) Then note that the first interrupt (IRQ 00, having the highest priority) is used by the "System Timer" which then provides the time base for the entire operating system. Giving the timer interrupt the highest priority, allowing it to interrupt other interrupts, is to ensure maintenance of the time variable. Otherwise, a too excessive number of interrupts could not be detected.

In embedded computers there are usually also some kind of time-out or stall-alarm interrupt connected to the very highest (unmaskable) interrupt level. It is then the obligation of the so called idle loop (the loop being executed when there are nothing else to do) to reset that timer so it never expires. But if that should happen, the CPU is overloaded and the controlled machine must be shut down because without enough time for the feedback control the system may behave very badly (crazy robot, crashing airplane, etc.). To prevent this, we will use special programming techniques in later chapters. In desktop applications, on the other hand, the user simply has to wait.

5.2 Pre-emption

When the scheduler, as a consequence of a timer interrupt, suspends execution of one activity and lets the program resume execution at another point (where execution was suspended at some earlier stage), we call it pre-emption. The term is used also in connection with operating systems; pre-emptive multitasking. Pre-emption means that execution may be suspended even if there is nothing in the executing code that explicitly admits the change of execution point. We may say that the system forces the application to give up the use of the CPU.

The opposite to pre-emptive scheduling is non-pre-emptive scheduling, which means that execution is only suspended when there is some call of scheduling functions or resource allocation. For instance, an attempt to open a network connection may include waiting for name-servers and remote computers to reply. Even if it is not desirable to wait, we know (and the compiler and run-time system knows) that execution may very well be suspended. These so called blocking operations are handled the same when pre-emption is used, but with pre-emption the suspension may also be enforced.

Different types of pre-emption exist. In general terms, there are three alternatives for the granularity of execution suspension:

1. The most primitive case is when pre-emption is not supported; execution is only suspended when certain system calls are made. The advantage is efficiency since the state of the execution is well known by the compiler and run-time system. For instance, all temporary registers that by the compiler are assumed to be destroyed by a function call, need not be saved. The disadvantage is that an application (or function) written without considering the need for other activities to execute concurrently will not give up the right to use the CPU, and may therefore lock the system or make response times too long.

2. Execution may be interrupted/suspended between each line of source code. From a programming point of view, it is very convenient to know that each statement gets fully executed without interrupt. Early-days Basic interpreters often worked this way. One disadvantage is that complex statements may delay change of activity too long. Not to interfere with other interrupt services, interrupts such as the timer interrupt is served immediately and ongoing execution may be marked for suspension. Another disadvantage is then that the checking between each statement costs processing power. An interesting version of this technique is implemented in the Lund Simula system which by utilization of the hardware (Sun Sparc) performs the check at almost no cost. The key drawback is, however, that concurrency requires support from compiler and run-time system, which means that most available code (written in C/C++) cannot be used when timing requirements are severe.

3. Execution may be interrupted/suspended between each machine instruction. The advantages are that any sequential code (without compiler support) can be interrupted, and that the interrupt and possible rescheduling will be started immediately. There will, however, be some delay since we must save the entire status of the execution (all registers etc.), and that may for some types of hardware be slower than case 2. Another disadvantage is that programming gets more complicated since it may be hard to know from the source code if a statement is atomic or not.

Summing up advantages and disadvantages, and considering the importance of timely operation also when non-concurrent code is used, alternative 3 is the best choice. That solution is assumed in the sequel. In the Java case, it is not clear what we mean by machine level. There are three cases:

- The Java program has been compiled into byte code which is then executed on a Java Virtual Machine (JVM) hardware. Then, the machine instructions actually used are the byte codes, and the program may be interrupted between each byte code which implies that a statement in the source code may be partially evaluated.

- The Java program, again represented by its byte codes, may be run on a JVM implemented in software. Internally in the JVM, there may be a so called JIT (Just In Time) compiler which compiles the byte codes into native machine code. The JVM may then permit pre-emption between byte codes, or between machine instructions when performing a byte code, depending on implementation. In both cases, the Java statements may be partially evaluated.

- If we know what hardware (and OS) we will run our program on, there is also the possibility to have a conventional compiler that compiles the Java source directly into machine code. Also in this case, pre-emption on machine level implies that expressions may be partially evaluated. For example, assignment of a double variable may be performed partially, resulting in one half of the bytes from the old value and one half from the new value for another thread.

Thus, from a programming point of view, it does not matter which of these cases that describe the actual situation. The fact to consider is that our sequential execution may be interrupted also in the middle of one Java statement. We will, however, in some situations know that an operation is atomic (one byte code or one machine instruction depending on the case) and then utilize that to simplify the program.

5.3 Reentrance

Now when we have multiple points of execution in our program, and we realize that access of shared data requires special care (which will be covered in the next chapter), a good question is: What about shared functions? In other words: what happens if the same function is being called concurrently from several points of execution? We say that we re-enter the function, and if that is permitted and working, we say that the code/class/function is reentrant.

Lack of reentrance is common problem in badly written C programs. The reason is the use of static variables, i.e., variables that are allocated on a fixed place in memory instead of on the stack or on the heap.

In most run-time systems, local variables in functions and blocks are allocated (pushed) on the stack, which means that they are deallocated (popped from the stack) when the PC leaves the scope. Data created dynamically, using operator new in Java or C++, are allocated on the heap which is sometimes called the free store. But also for heap allocation we usually have the reference to the object declared locally in some scope which means that it is put on the stack. This is a proper programming style which means that all data is pushed on the stack of the caller. If, on the other hand, we use static variables, two callers of one function will access the same local data (referred to by absolute addresses in the compiled code of the function).

The real problem is when the one who wrote the function is not aware of concurrent programming. Example: A numerical equation solver was implemented in C, and then it was packaged together with other routines in a library. To optimize the code for the hardware at that time, the temporary variables of the algorithm were declared static. The function was then successfully used for years on an old-style UNIX machine, where each program only contains one point of execution (except for the signal handlers). Then, the library was compiled and linked with applications running on the newer so called multi-threaded UNIX dialects, such as Sun Solaris, HP-UX 10 and later, and later versions of IRIX from SGI. It was also used on the Microsoft Win32 platform. In all these cases, when using the equation solver concurrently from different parts of the application, the results were sometimes wrong. In this case, the function was correct as a program, but not correct when used in a concurrent program where some interleavings resulted in undesired mixing of temporary data from different equations.

Also in Java it is easy to declare static variables, but we should only do that for so called class variables needed for the structure of our program, and in that case we should treat them as other types of shared data (which we will return to). Note that when you write an ordinary function, you cannot know if it later will be used concurrently. In conclusion, one should learn concurrent programming before doing any type of professional programming.

6 Models of concurrent execution

The concurrency issues made our simple sequential execution model (from Section 2.1) more complex because we now have several sequences that interact with each other. With each sequence, there are:

- A PC (the Program Counter) which according to the previous section refers to the next machine instruction to be executed.
- An SP (the Stack Pointer) which refers to the stack of that particular execution sequence. On the stack, there are the activation records of all blocks/functions that are currently entered, as required for reentrant code.
- The machine state (of the CPU or some virtual machine depending on the implementation of the run-time system) holds part of the status of our execution sequence.
- If common resources (like shared data) are reserved in one sequence, that affects the others and should therefore be dealt with somehow.

Continuing our attempt to bring order in the complexity by exploring fundamental properties, some basic terms will now be described. First we have the following two definitions:

Definition: A context is the state of an active program execution, which includes PC, SP, and machine state. From an operating system point of view, execution may also be associated with permission and user privileges which then may be included in the context. In such a case we call it a process context or a heavy-weight context. From this point on, we will use the former minimal definition which is also called a light-weight context.

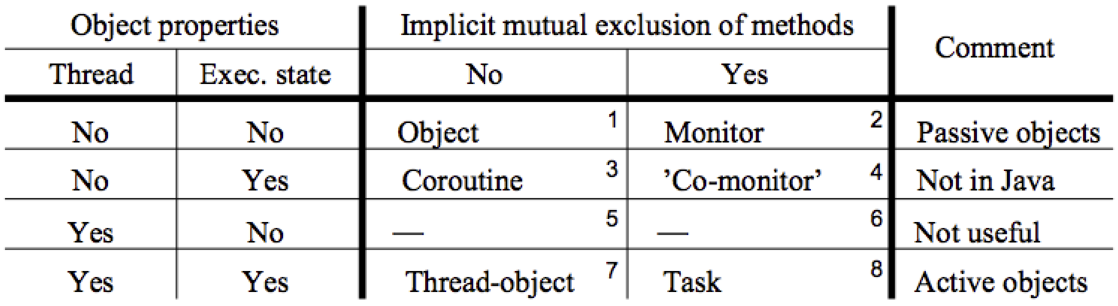
Definition: A context switch refers to the change of context typically performed by some scheduler or operating system.

Since we aim at appropriate programming methods, there is the question about what support we have from the language and from the programming interfaces (libraries of available classes). This has a great impact on the way we have to write our code. For instance, when programming in C, the language itself does not give any support for handling concurrency which makes the use of special libraries important (and tricky). In Ada there is language support in terms of tasks and *rendezvous*, but using that support has a significant impact on both design and implementation. A language with extensive support, like Occam, is in danger of being too special and therefore not being widely used.

6.1 Fundamental abstractions

The question now is, to what extend should a general purpose programming language support concurrency? As suggested by Buhr in a proposed extensions of C++, it is appropriate to separate the context from the execution sequence as such, and the following execution properties should be supported by the language:

Thread - is execution of code that occurs independently and possibly concurrent with other execution. The execution resulting from one thread is sequential as expressed in ordinary programming languages such as Java. Multiple threads provide concurrent execution. A programming language should provide features that support creation of new threads and specification of how these threads accomplish execution. Furthermore, there



Object properties		Implicit mutual exclusion of methods		Comment
Thread	Exec. state	No	Yes	
No	No	Object 1	Monitor 2	Passive objects
No	Yes	Coroutine 3	'Co-monitor' 4	Not in Java
Yes	No	— 5	— 6	Not useful
Yes	Yes	Thread-object 7	Task 8	Active objects

Figure 1.6: Fundamental properties of software execution. Language support for an execution property is marked with "yes". For mutual exclusion to be supported, it must be implicitly obtained from the types or declarations.

must be programming language constructs whose execution causes threads to block and subsequently be made ready for execution. A thread is either blocked or running or ready. A thread is blocked when it is waiting for some event to occur. A thread is running when it is actually executing on a processor. A thread is ready when it is eligible for execution but is not being executed.

Execution state - is the state information needed to permit concurrent execution. An execution state can be active or inactive, depending on whether or not it is currently being used by a thread. An execution state consists of the context and the activation state. These are typically stored (internally in the run-time system) in a data structure, which traditionally has been called process control block/record (PCR). An inactive execution state is then completely stored within the PCR, whereas the hardware related parts (registers, interrupt masks, etc.) of an active execution state are stored in the hardware/CPU. Referring to the context as the main item, we call the change between the active and inactive states a context switch. The switch includes storing or restoring the hardware state, and it occurs when the thread transfers from one execution state to another.

Mutual exclusion - is the mechanism that permits an action to be performed on a resource without interruption by other operations on that resource. In a concurrent system, mutual exclusion is required to guarantee consistent generation of results, and it should therefore be supported in a convenient way by the programming language. Furthermore, for efficiency at run time, it needs to be provided as an elementary execution property.

In an object-oriented framework, these three execution properties are properties of objects. Therefore, an object may or may not have/be a thread, it may or may not have an execution state, and it may or may not have/provide mutual exclusion. The latter means that it may or may not exclude concurrent access via call of methods of that object. The next issue is to know: What are the possible and useful combinations of the execution properties? All eight combinations (which would make up a 2 by 2 by 2 cube) is depicted in Figure 1.6. The numbers in that table refers to the following items:

1. This is an ordinary object as known from object oriented programming, or an ordinary piece of code in the case that object orientation is not supported. This is the only case that programmers not acquainted with concurrent programming use. A class programmed in this way can still be useful in an concurrent environment if its member functions are reentrant, which they are if only local data is used and the if language/execution supports recursion (direct or indirect). Since there is neither a thread nor an execution state, we say that the object is a passive object.

2. It is still a passive object if we add mutual exclusion, but only one thread at a time may run the code of the object. This abstraction is called a *monitor*.

3. Having an execution state without having a thread permanently associated with it forms an abstraction called a coroutine. A coroutine must be used by a thread to advance its execution state. If you have multiple coroutines but only one thread, only one coroutine at a time may execute. This is useful to handle parallellity issues in algorithms or in simulations because there the (virtual) time of the model can be held or advanced arbitrarily. That is, the real time used by a specific thread for its execution is unimportant, and a single thread using multiple coroutines is appropriate. An implementation based on coroutines can always be rewritten to work without them, but that may require more complex data structures and code. In combination with timer interrupts triggering a scheduler, coroutines can be used to implement thread objects.

4. Adding mutual exclusion to coroutines is not really necessary in the case of just one thread, and in the case that a group of coroutines run by one thread interacts with other threads, this alternative leads to difficult and error-prone programming. The name co-monitor is not established and this case is neither used nor commented in the sequel.

5. A thread without an execution state cannot execute on its own, and borrowing another thread for its execution does not provide concurrency. This case it therefore not useful.

6. Also with mutual exclusion, this case is not useful for the reasons mentioned in item 5.

7. An object with both thread and execution state is capable of execution of its own. We call this an active object or a thread object. The problem is that access to the attributes requires explicit mutual exclusion. That has to be simple to express in the program in order to make this alternative useful. Note that even if an active object calls member functions of other (active or passive) objects, the execution of those functions is driven by the calling thread, and the stacked activation records are part of the callers execution state.

8. A thread object with implicit mutual exclusion is called a task. This abstraction can be found in the Ada language.

The abstractions supported by different programming languages differs. Concurrency in the Ada language is based on the task, which in that language is augmented with a specification of how such objects interact. Modula-2 provides coroutines with pre-emption which allows a pre-emptive scheduler (supporting threads) to be built. Simula provides coroutines that, together with interrupts checked between each statement, can be used to build a scheduler and thread support. Neither Simula nor Modula support implicit mutual exclusion, which means that shared data must be explicitly reserved by calling certain functions for locking and unlocking. In C and C++, there is no concurrency support at all.

6.2 Concurrency in Java

So, what are then the design choices for Java? As with other parts of Java, we find fair engineering choices rather than new unique features.

- There is no implicit mutual exclusion on object level, but individual methods (and even blocks) implicitly provide mutual exclusion by the explicit use of the synchronized keyword. Unfortunately, only code and not the accessed data is synchronized. Some extra programming discipline is therefore still required. Compared to having implicit mutual exclusion fully built into the language, the advantage is that the programmer is more free to tailor his/her code for specific application needs (such as efficiency).
- The notion of a thread object is supported via the object orientation by inheritance from the Thread base-class. When inheritance is rather used for other properties (recall that Java only provides single inheritance), the interface Runnable can be used to add the thread property to an object.
- Coroutines are not supported. Normally, threads are used to achieve the parallel behavior of coroutines. Therefore, large-scale event-driven simulations may be very inefficient/slow.[2] In normal applications, including embedded systems, thread objects provide almost the same execution efficiency. The major reasons for omitting the coroutines in the Java language are: 1) The language can be kept simpler. 2) Most programmers do not know how to utilize coroutines anyway, particularly not in combination with threads. 3) Having the programmer to use threads instead means that the run-time system is free to let activities run in parallel if there are multiple CPUs available.
- The threads are based on so called native methods, i.e., functions that are implemented external to the JVM. In the case of thread support, native methods may be implemented by the operating system. Thread creation, interaction, and scheduling are then handled by the system functions which are optimized for the actual hardware used (and run in parallel as mentioned in the previous item). Of course there is a lack of elegance compared to a system based on coroutines, having the scheduler implemented as in the language itself. However, for the majority of applications running on an operating system such as the Solaris operating system (the Sun UNIX which supports up to 64 processors), using the native threads boosts performance (without changing the source code).

In conclusion, Java provides passive objects and active objects. A passive object is either an ordinary object, or it can be an object with no data exposed and all methods synchronized which then functions like a monitor. There are, however, no real monitors in the language; the programmer may expose data to concurrent access and some methods may be left unsynchronized. Active objects are those with the thread property, but we do not have (and we will not talk about) tasks since we do not have threads with implicit mutual exclusion. We will therefore only speak about classes, objects, synchronized methods, and threads. These are our building blocks for concurrent and real-time programming. There is, however, very little support for handling timing requirements, which we will return to in the final chapters.

6.3 Processes and interrupts

In the above treatment of programming language support for concurrency, we neglected two aspects, processes and interrupt service routines. In short, a process is the type of

[2] To overcome this (unusual) problem, a superset of the Java language and a dedicated compiler and run-time system (replacing the JVM) could be developed. By implementing the language extensions as a translator implemented in Java, generating ordinary Java-code, the application would still be 100% Java when not optimized.

concurrent execution we have when we run some program under an operating system such as UNIX or Windows NT, and an interrupt service is the concurrent execution we have when a hardware interrupt starts a new thread of execution. These have to do with the interaction between the Java program and its environment in terms of hardware or other programs. Since this is quite dependent of the specific environment, it is probably a good idea not to provide any language support. Instead, we use classes that provide certain programming interfaces. Here, we will use the term interrupt to denote the execution carried out to service the actual hardware interrupt.

Both a process and an interrupt contain threading and some kind of execution state, but a process is more powerful than a thread whereas an interrupt is less powerful. In more detail:

- A process may contain one or several threads which we then say are internal to the process. The internal threads share a common address space in which they can share data and communicate without having to copy or transfer data via pipes or files. An object reference (or pointer in C/C++) can be used by several threads, but sharing objects between processes requires special techniques. The same applies to calls of member functions. All threads of one process have the same access rights concerning system calls, file access, etc. Different processes, on the other hand, may have different access rights.

  The concept of a process makes it possible to isolate execution and data references of one program. By controlling the MMU (Memory Management Unit) accordingly, an illegal data access (due to dereference of a dangling pointer in a C program for instance) can be trapped and handled by the operating system. To the UNIX user, there will be something like a "segmentation fault" message. In Windows 95/98, which lacks this support, one application programming error may crash other applications or even hang the operating system.

- An interrupt starts a new execution thread, but it is subject to special restrictions. First, it must complete without blocking because it may not change execution state (switch context). Second, the interrupt may only start in a static function/method because this type of hardware call does not provide function arguments such as the implicit this pointer. Third, an interrupt may run within the context of another thread because the final restore of used registers and the requirement of "execution until completion" ensures that the context of the interrupted thread will be restored. The priorities of interrupts are managed by the hardware.

Thus, whereas threads will be the most common way to accomplish concurrently executing software entities, an ISR is sometimes more appropriate or even necessary (device drivers), or there may be reasons for using (OS) processes instead of threads. Device drivers cannot be (purely) implemented in Java, and such machine-level programming is outside the scope of this book; it is well known how to do that and there are several hardware/OS-specific manuals/books on the topic. The issue of multithreading versus multiprocessing is more relevant, in particular for large complex systems, deserving a more detailed treatment.

7 Multi-process programming

Programming languages include constructs for expressing the execution of one program, while interaction between programs are handled via library calls. Exercises in basic programming courses are solved by writing one program, possibly in several files that are compiled separately, but linked as one executable unit which is executed within one OS process. Possibly, event handlers (in terms of event listeners or callback routines) or multiple threads are used, but it is one program and one process sharing a common memory space. Note that even if we let different OS processes share memory (via system calls such as mmap in UNIX), we have no language support for signaling and mutual exclusion; that too has to be handled via system calls.

Using multiple threads (multi-threaded programming) can, in principle, as mentioned, be used to create any concurrency. Also for utilization of multiple CPUs (with shared memory), threads are enough. So, in real-world systems, what are the reasons for using multiple processes instead of just multiple threads? In short, reasons for dividing a software application into multiple OS processes include the following:

1. Different parts of the applications need to run on different computers which are distributed (not sharing memory).

2. Different parts of the application need different permissions and access rights, or different sets of system resources.

3. In the Java case, different virtual machine properties (such as GC properties) can be desirable for different parts of the system, depending on timing demands etc.

4. If the application consists of (new) Java parts and other (typically old, so called legacy) parts written in other languages like C, these parts can be interfaces and linked together using the Java Native Interface (JNI). However, if the virtual machine and the legacy code requires specific but different native threading libraries, this may not be possible (resulting in linkage errors). This is solved by using multiple processes.

5. Licensing terms for different parts of the application can forbid execution as one process. For instance, GNU-type of free software (GPL) may not be linked with proprietary software.

6. A large application including code written in an unsafe language (such as C/C++) will be more reliable if it is divided into several executable units, each with its own memory space and with memory accesses checked by the MMU (memory management unit) hardware. Instead of a 'blue-screen' due to an illegal access, just one part of the application may stop. The use of protected memory is a main reason why large systems have been possible to develop despite unsafe languages. Still, any program may contain fatal bugs, but a safe language such as Java ensures that at least the errors are kept local and exception handling works.

7. Libraries, subsystems, or legacy software in general may not be thread safe. That is, variables (or attributes or fields) for storage of local data within a function (or method or procedure) may be declared static. That used to be a way to speed up programs and to save stack space, but the result is that the function can only be called by one thread at a time. By obtaining concurrency only via OS processes, we do not have to require the software to be thread safe since each process has its own memory allocated by the OS.

Sometimes you may want to develop classes and active objects which can be run either as threads or as a processes, without changing the code. Such issues and the principles of how to design and implement multi-process applications are topics of Chapter 7.

Note that from a *concurrent design* point of view, neglecting implementation issues such as if concurrency should be accomplished by threads or processes, we usually use the term *process* to refer to a (concurrently executing) activity. This is in line with non-technical vocabulary where a process refers to some ongoing activity that cannot be completed immediately. (Pay attention to how the word is used in the news, in economy, and in management principles.)

8 Object interaction and encapsulation

The interaction (information exchange) between objects may be synchronous or asynchronous. It can be built into the programming language, and/or it can be programmed. Synchronous communication can be implemented on top of asynchronous communication, or vice versa.

Here, we consider synchronous communication as more fundamental. With the execution properties of Java this simply means a method invocation, which is based on ordinary function calls (works exactly as in C/C++ and most other languages). Thus, it is built into the language. This is the most computing-efficient way to achieve object interaction in compiled languages.

The asynchronous communication means passing some kind of message without requiring the sender/caller to wait until the receiver/callee replies; the caller performs its communication asynchronously with the callee. The messages are often referred to as events. But note, however, most of the event processing done for instance in Java's AWT or Microsoft's MFC actually constitutes synchronous communication because there are no buffers providing the asynchronous behaviour. It depends on the application demands if we want buffering or not. We will take a closer look at this issue in later chapters.

In this chapter we have reviewed the properties of software execution, as expressed in the source code. The issues we are about to study are about software development, assuming that the underlying (virtual) machine machine (and system software) is correct. Assuming also that the compiler is correct (producing binary code, preferably efficient, that behaves as expressed in the source code), we might believe that we could do the programming in any (convenient) language. However, if a program (for instance during testing) works correctly, and is correct from the concurrency point of view, we cannot know that the program works correctly in another case, even if it was tested with all combinations of input data. The reason is that most languages do not prevent programs with undefined behavior, which may result in damaged data within the same object or at any other location within the program, or even outside the program if the OS does not provide memory protection. Undefined behavior can stem from use of uninitialized data, dangling pointers, unchecked typecasts, etc. Therefore, in order to be able to build systems that scale up for complex applications, we should use a language that only permits well-defined programs. We call such a language safe.

Definition: A programming language is safe if and only if all possible executions of any program written in that language is expressed by the program itself.

In particular, Java is safe (but native/unsafe code can be called), C# is not quite safe (safe except where the keyword unsafe is used, or when native code is called), Ada is unsafe, while C and C++ are clearly unsafe. Every language must permit unsafe code to be called, otherwise device drivers could not be used. You may consider it as a deficiency that neither viruses nor device drivers can be written in Java; some implementation has to be done in some other language. One could also consider it as an advantage; hardware access (which will not be portable anyway) has to be written well separated from the rest of the application. Those (usually small) pieces of software also have to be more thoroughly debugged since a programming error could crash the entire system without exception handling of system shutdown working, opposed to safe programs which terminates in a controlled way with supervisory functions and recovery systems still running. We use a Java-based approach to embedded systems programming since we need to trust their behavior.

9 Software issues

Based on the properties of software execution in general, and on the, for our purposes, quite good design choices made in the Java language, we are now ready to tackle the issue of concurrent and real-time programming. To sum up, we should be able to develop software such that the following is obtained:

- Correctness from a concurrent point of view; all allowed interleavings as expressed in the source code should result in the same correct result independent of the underlying scheduler or operating system.
- Correctness from a real-time point of view; the timing requirements are considered so that control output is computed in time.
- Proper object oriented design and implementation; we use the established principles of object orientation to improve structure, readability, maintainability, etc. of embedded software.
- Appropriate trade-off between active and passive objects; the thread class should be used where appropriate but not too excessive for efficiency and readability reasons.
- Proper protection of shared resources; the concepts of semaphores, synchronized methods, and messages (events) should be used appropriately.

Since software development tends to take a bigger and bigger part of system development resources, it is very important that we are able to handle the above items. In a wider perspective, there are different approaches to development of embedded systems. The focus could be on hardware, control theory, and/or software issues. All of these focuses are important for industrial development of control systems. Here we approach the development of embedded software from a programming point of view, leaving the rest (like selection of computing and IO hardware, discretizing control algorithms, etc.) for other books.

2 Multi-Threaded Programming in Java

Goal: *To know concurrent object-oriented programming using standard Java*

With understanding of general application demands and concepts presented previously in the course material, we will in this chapter look into the issue of handling concurrency within one program, utilizing the features of the Java-platform. Acquaintance with both (sequential programming in) Java and object oriented programming (OOP) is assumed. The following properties and restrictions should be clear to the reader:

- One program means one executable unit in which different objects (possibly concurrently) can address data in a common memory space. The program is run either as one process using an Operating System (OS) or as an embedded application linked together with a so called real-time kernel (or RTOS).

- As should be clear from the application examples, the software needs to respond to concurrently occurring external events. Sequential computing and handling of sequences of such events, expressed as ordinary sequential programs, then has to be executed concurrently.

- By simply using the concept of thread objects, we assume that the underlying OS/kernel (which includes a scheduler) schedules the concurrent threads (normally many) for execution on the available CPUs (normally one). Hence, the system interleaves the code at run-time based on system calls which have been programmed according to this chapter.

- For the software to be correct, it should produce the correct result regardless of the actual interleaving. That is, the correct result should be obtained for any interleaving (for given input sequences) that comply with the programmed system calls. The correct result may depend on time, and different actions could be taken depending on time, but there is no guarantee concerning the time of execution.

- Specifically, priorities are suggestions for the scheduling (and thereby the timing) but the correctness must not depend on priorities. Furthermore, waiting a specified time (sleep) or until a specified time (sleepUntil) only means that execution is suspended for at least that long. That permits the scheduler to have other threads running during the specified amount of time, but the software alone does not specify any upper limit of the real-time delay.

These are basically the preconditions for the Java platform. It agrees well with concurrency for Internet applications; the network exhibits unpredictable timing anyway. It also agrees with the properties of the most well-known operating systems. Without any further assumptions, our concurrent system is not a real-time system. By simulating the environment we can, however, simulate a real-time system and carry out almost all of the software test and debugging.

Hence, multi-threaded programming (i.e., concurrent programming based on threads) should therefore not be confused with real-time programming![1]

In the following sections, the basics of concurrent programming in Java are explained.

[1]Some further comments on that: In order to obtain correctness from a real-time point of view, a concurrent program has to be run on a real-time platform. Such a platform must fulfil additional requirements concerning CPU-speed, available memory, IO and interrupt handling, scheduling algorithms, and the input to the scheduler from the application software. Even on a non-real-time platform we may obtain a system behaviour that appears to be real-time. A common approach, for instance when using PCs in industrial automation, is to only let one or a few well-known programs/daemons/ services run, to have some activities run in some kind of kernel or driver mode, to tailor the application software for improved handling of sporadic delays (for instance by temporarily adjusting parameters and decrease performance), and by using a powerful computer. Still, since the system is not guaranteed to produce a quite correct result at the right time, it is not real time. But from an application point of view, the system behaviour can be good enough. Therefore, in a practical industrial context the borderline between real-time and concurrent programming is not that clear.

1 Threads

For now we assume that our concurrent activities can run independently of each other, with no need for synchronization and mutual exclusive methods.

1.1 Thread creation

At start of a Java Virtual Machine (JVM), the so called main thread starts executing the Java program by calling the main method of the class given to the JVM. To have another thread running, a system call has to be done (within the JVM or to the OS, at this stage we do not care which).

In most programming languages there is no built-in support for concurrency, and creating new threads has to be done via a platform and language specific system/library call. In the Java platform, we have to create an object of type java.lang.Thread and call its method start (which is native, i.e., not written in Java). The caller of start returns as for any method call, but there is now also a new thread competing for CPU time. The newly created thread must have an entry point where it can start its execution. For this purpose there must be a runnable object available to the thread object. A runnable object provides a method run according to the interface java.lang.Runnable; that interface simply declares a `public void run()` and nothing more. It is that run method you should implement to accomplish the concurrent activity.

What run method to be called by start is determined at thread-object creation time, i.e., when the thread object constructor is run. There are two alternatives depending on what constructor that is used, extending a thread object or implementing a runnable object:

- By *extending* class **`Thread`**, `Runnable` will be implemented since public class Thread extends Object implements Runnable. In other words, this will be an instance of Runnable. The default constructor will select this.run as the beginning for the new thread.

  Advantages with this alternative is that you can override also the start method (to do some things before/after thread creation/termination, calling super.start in between), utility methods of class Thread can be called without qualification and asking the run-time system for the currently executing thread (described in the sequel), and the inheritance is a good basis for developing further threading subclasses .

- By *implementing* the **`Runnable`** interface, the object can be passed to the Thread constructor which then will use the run of the supplied object as the beginning for the new thread. The advantage with this approach is that the (in Java single) inheritance can be used for other purposes, such as inheriting graphical properties.

In short, as indicated in the second item, which alternative that is most suitable depends on the primary issue of the software. For instance, a graphical user interface using the javax.swing classes and only some threading for handling the network will probably inherit graphical properties and implement Runnable when needed. For an embedded control application, on the other hand, threading (and timing) is a primary issue and inheritance from class Thread is a natural choice.

Some programmers prefer to always use the Runnable alternative as their standard way of doing things, whereas others always create Thread subclasses (sometimes having an internal thread object when there is a need to inherit from other classes). In the sequel, threading is the primary issue so we will mostly use Thread as a base class, but utilizing interface Runnable is covered as well.

1.2 Time

Even though concurrent programs do not ensure timing, it is very useful to let actions depend on real time, i.e., reading the real-time clock which is available on any useful computing platform. In Java the default time unit is milliseconds, and the current clock time is obtained by calling System.currentTimeMillis which returns the time in milliseconds counted from beginning of 1970. Since the return value is of type long, the y2k type of problem that comes with the finite time range is quite far in the future (year 292272993, which should mean no problem for most applications).

The advantage with currentTimeMillis stored in a built in numeric datatype (long) is that it can be conveniently used for computations. For human-readable time Java also provides a Date class which also copes with time zones, locales, and the like. That is, however, a matter of formatting and for our needs it is the system time in milliseconds that is relevant.

Assume we want to compute the trend (or numerical derivative) of an analog input signal. Further assume we have a method inValue that handles the AD conversion and returns a scaled value as a float, and let us say that we have an infinite resolution in value. Concerning resolution in time, we have the millisecond time described above. That time is updated via a clock interrupt routine in the JVM or OS. Some embedded computers are equipped with a high resolution real-time clock, but using it would require a native method and loss of portability. Using only the standard 1 ms resolution, an attempt to compute the trend could be:

```
1 float trend() {
2   float x, x0 = inValue();    // [V]
3   long t, t0 = System.currentTimeMillis();
4   while ((t=System.currentTimeMillis())==t0) {/*Busy waiting.*/};
5   x = inValue();
6   return (x-x0)/(t-t0)*1000;  // [V/s]
7 }
```

With an input signal that increases constantly with 1 V/s we would like to obtain the value 1.0, but already with such a small piece of software it is hard to foresee the value actually returned. Let us look into a few basic aspects:

- Time quantization: Waiting for the next ms does not mean waiting for 1 ms. In fact, running on a 100 MHz CPU it may take only 10 nano-seconds (1 CPU cycle) until a new clock interrupt is served and the time (in software) is incremented. The real time between the readings will then be only the time to run lines 2 to 5, plus the service time for the clock interrupt, say 0.1 us giving a relative error of 1000000%! Ignoring the (quite small) computation time we see that even with the same numerical time values, the difference in real-time between two time readings (depending on when during the so called tick period each reading was done) can be up to (but not quite) two clock ticks.

- Scheduling: It can on the other hand also be the case that the new clock tick causes another thread to be scheduled and the next reading takes place one second later. So even with a maximum of two ticks (2 ms) quantization error, the maximum trend error will be 0.2%. Note again, the actual schedule (and thereby the returned value) is not expressed in the program; it depends on the system.

- Efficiency: The longer execution is suspended between lines 3 and 4, the better the estimation of a constant trend gets[2]. The waiting for the second sampling time above was done by a so called busy wait. In rare situations this way of waiting for a condition to be fulfilled can be useful, but especially for longer delays it causes efficiency problems (and also response problems if real time is considered). A busy wait actually asks for 100% of the CPU time. A better approach would be to inform the thread scheduling that we permit other threads to run for a certain period of time. That is the purpose of the sleep method treated below.
- Time-stamp deficiency: Suspended execution (and incrementing time) between lines 2 and 3, or between lines 4 and 5, also causes a bad trend error to be computed. This holds regardless of how small the time increments are, that is, even with the high resolution clock mentioned above. Also, if we swap lines 2 and 3, the program should be the same from a sequential computing point of view, but due to the System call it is not; the t can deviate in different directions which creates trend errors in different directions.

The fact that we care about when an input value was sampled can be looked upon as if we would like to have a time-stamped value. That is, the value and the time of sampling it should be correctly packaged in a single object, which the inValue method (in this case) should return. Creating such an object in this example typically requires native or hardware support depending on the system and its timing requirements. In a pure Java and concurrent setting, some improvements can, however, be made. For example the clock can be read before and after sampling:

```
long timestamp, t0, t1; float x;
do {
    t0 = System.currentTimeMillis();
    x = inValue();
    t1 = System.currentTimeMillis();
} while (t1-t0 > eps);  // eps is small, possibly zero.
timestamp = (t0+t1)/2;  // Statistical improvement.
```

The disadvantage is that on a heavily loaded system, with certain scheduling, it can take a quite long time until the do-loop finishes. Limiting the looping by using a for-statement instead would be another solution which can easily be accomplished by the reader.

Despite the ideal setting of this example, we found all these problems. With measurement noise and input value quantization the situation gets even more severe.

1.3 Sleeping

By replacing the comment **`/*Busy waiting.*/`** in the above example with a call to `sleep(1)` (defined in class `Thread`), the efficiency issue is solved. The purpose of sleep is to provide a way of informing the scheduler that other threads may run during that long time from now, and that the calling thread should not be rescheduled for execution within that time. By calling for instance `sleep(500)`, a minimum discrete time delay of 500 milliseconds is obtained, which means (due to the quantization in time as described above) a continues/real-time delay that is greater than (but not equal to) 499 ms.

[2]For a changing input value, however, we need a short delay to get a fresh estimation. In real-time software we need to manage such trade-offs and preferably express the required execution timing in the source code, or we have to impose requirements on the run-time system. For now we only deal with time-dependent software which should not be confused with real time.

Assume you want to do something approximately every new second, such as updating a progress bar while downloading a file from Internet. We may have a piece of code like

```
long t, t0 = System.currentTimeMillis();
while (!transferFinished()) {
    t = System.currentTimeMillis();
    displayProgress(t0, t);
    Thread.sleep(1000);
}
```

If this code is in a subclass of Thread, the Thread qualification before sleep can of course be omitted. Since sleep is a static method, it can be called without having an instance of a thread object. In fact, even if you have a thread object threadOne, and call threadOne.sleep(1000) from another thread object (or any object), the calling thread (i.e., the thread of execution, not the thread object) will be put to sleep while threadOne is not effected. This is because sleep first obtains the currently executing thread, which you can also do by calling Thread.currentThread(), and then that thread (the caller of course) will sleep. In other words, Thread.sleep(t) is the same as Thread.currentThread().sleep(t), and a thread can put itself to sleep but *cannot put another thread to sleep.* If that should have been possible, certain restrictions on run-time systems and code optimizations would need to be imposed, resulting in decreased (possible) performance.

Note that sleep(1000) above does (in most cases) not give an update every second; the 1000 is a minimum sleeping time but there is no strict upper limit. The actual time before execution continues after sleep typically depends on the processor load. Most often, especially in control applications, it is a periodic behaviour that is desired, though.

We have a similar situation if we in our example wants to call displayProgress every second. Still we do not impose any upper limit on the waiting for each new second (which would be a real-time requirement), we just require no long-term drift as long as the computer is not permanently overloaded. This is accomplished in the following:

```
long t, t0, diff;
t = t0 = System.currentTimeMillis();
while (!transferFinished()) {
    displayProgress(t0, t);
    t += 1000;
    diff = t - System.currentTimeMillis();
    if (diff > 0) Thread.sleep(diff);
}
```

Note that t now is ideal, not reflecting temporary excessive delays. To facilitate implementation of periodic activities, there should of course have been a method sleepUntil in class Thread, but there is not. Even if it can be emulated as shown, an increment of the real time after assignment of diff but before calling sleep results in a requested sleeping time that is too long. That does not really matter since we can be delayed after the sleeping anyway. (But from a real-time systems point of view, it would be preferable to correctly inform the real-time scheduling.)

1.4 Scheduling – Priorities

The number of concurrently executing activities (threads or processes) is in practice always greater than the number CPUs available, which implies that all work cannot be worked on immediately. The support for concurrent execution (multi threading etc.) solves this problem, if we have no demands on timely response. In other words, for purely concurrent software, we can let the run-time system schedule our activities in any feasible way that lets all activities run. However, if the application requires certain response times or performance, in particular when the CPU load is high, we may need to tell the run-time system (that is, the scheduler) what work should be done first, and what activities that can wait.

Consider the case with a single processor, and compare with every-day life at home, in school, or at work. Being just one person but with many things to be done, is a situation that all readers should be familiar with. People differ in their principles of 'what to do first'; they are using different sorts of scheduling. The same applies to operating systems. In both worlds, the way of selecting 'what to do next' is closely related to the externally experienced properties of the person/system. Doing the right thing first is often more important than doing it in a superior way.

For now, we restrict the discussion to a single multi-threaded application running on one JVM. For embedded computers, that corresponds to the software running on one processor, not necessarily using a JVM since cross compilation is often more efficient but from a programming point of view that does not matter. Thus, we need some method to inform the scheduling (in the OS, RTOS, kernel, or JVM) about the preferences of our application.

Expressing importance

In a company you may hear things like: "This is a high priority project...", "We have to put priority on...", or "That bug has to be fixed, with the highest priority.". Likewise, the by far most common way to influence the scheduling in operating systems is via priorities. Since Java is designed to be portable, providing the ability to set priorities on threads is a natural and understandable choice.

In the `java.lang.Thread` class there are set- and get-methods for the priority of the thread. Additionally, there are three integer constants named `MIN_PRIORITY`, `NORM_PRIORITY`, and `MAX_PRIORITY`, which can be use to set typical levels of importance. The actual values of these constants are not defined and the number of priority levels are not defined either in the Java specification; you can only assume that

`MIN_PRIORITY < NORM_PRIORITY < MAX_PRIORITY`.

The rest is up to each platform and Java implementation. On, for instance, Linux the constants have values 1, 5, and 10 respectively. Thus, a higher number means a higher priority. A new thread by default gets the priority `NORM_PRIORITY` but that is not required in any standard.

The lack of a well defined meaning of priorities in Java may give the impression that they are almost useless. We should, however, be aware of:

1. The correctness of our software, except for timing which is what priorities are for, should never depend on priorities. For instance, it should be considered as an error to accomplish mutual exclusion by utilizing the knowledge that one thread will have a higher priority than another. Even if that always is the case, there is no guarantee that priorities are strictly obeyed on all platforms. That is, using an RTOS a higher priority thread always given precedence in favor of a low priority thread, but on Solaris, Linux, Windows, and other desk-top and server platforms, you cannot know. Those types of systems are optimized for overall performance and can in principle not be trusted for hard real-time systems. So, if embedded software requires a certain RTOS anyway, additional requirements on the number of priorities etc. go along the the system requirements. Still, the logical and concurrent behavior of the system should remain on any Java-enabled platform.

2. An application that depends on scheduling principles and the number of priority levels can obtain the type of OS and other properties at run time (e.g., by calling

System.getProperty("os.name");), and behave accordingly. A OS-dependent but portable application simply has to cope with reality (rather than Java principles).

3. Priorities are managed globally for the software application and they influence the global properties of the system. However, a certain thread created in a class library or in a third-party component too often gets a (usually hard-coded) priority that does not suit the rest of the system. Furthermore, the desired system properties are not necessarily easy to express using priorities. Response time, control performance, quality of service, etc. are more related to end-user values. Obtaining priorities from such properties is outside the scope of this chapter. Another approach would be to have a real-time platform with built-in timing and/or performance optimization, perhaps not using priorities at all. The classical approach, found in almost any OS is to use priorities, and that is also what Java provides.

1.5 Thread interruption and termination

There is much to say about thread termination, but in short there are some simple rules to follow:

1. Return from the run method terminates the thread. After the run method (which was called by the thread) returns or runs to its end, control is back into the system call that started the thread (due to the call of start). That call then completes by terminating the thread. Thereafter, calling the isAlive of the terminated thread object returns false.

2. If one thread wants to terminate another thread, that other thread has to be prepared to commit suicide. Since there is no predefined methods for doing this, it has to be accomplished by the programmer, either by changing some state variable (such as a field/attribute named terminate), or by interrupting the thread which then (by convention) knows that it should die whenever interrupted.

3. The interrupt method is called in the manner "otherThread.interrupt();" to interrupt another thread. However, unlike hardware interrupts, the other thread is not really interrupted; it continues its execution as usual, except that a predefined internal state variable has registered the interrupt request.

4. Each thread that should be possible to interrupt has to explicitly test if it has been interrupted. For this purpose, there is a static method Thread.interrupted() that returns true if interrupt has been called for the currently running thread. Note that this method also clears the interrupt flag. To check if another thread (i.e., not the currently running one) has been interrupted, isInterrupted() has to be called for that thread *object*. That does not effect the interrupted status.

5. If a thread was interrupted when blocked on a blocking system call, such as sleep, it is scheduled immediately for execution but throwing an InterruptedException. In this case the interrupted flag is not set, but by catching the InterruptedException we can determine that the thread was interrupted. Thus, it is the execution of the catch clause that tells that the thread was interrupted, and the exception handling decides if the thread should resume its work or terminate (by returning from run).

6. In accordance with the previous item, if a thread already has been interrupted when a system call (that throws InterruptedException) is issued, the exception is thrown immediately without blocking.

Due to these rules, the standard way of implementing a thread is to have a run method looking like:

```
public void run() {
    while (!isInterrupted()) { // !interrupted() OK in most cases.
        // here is where the actual code goes....
    }
} // End of life for this thread.
```

After the loop, isInterrupted can be used to check how the loop was exited. Instead of terminating after an interrupt, you can have an outer loop to restore the state and continue execution. Additionally, you may need to `catch InterruptedException`, or other `Throwable` objects such as those of type `Error`, either outside the while loop above or at several places inside that loop depending on the code. Note, however, you should normally *not try to catch and recover from an* `InterruptedException` thrown from a blocking call that have to do with access or allocation of shared resources/data, if resources are not left in a consistent state (bank transaction not half made, etc.). When resuming normal operation after such a call has been interrupted, there is a big risk that execution continues as if resources where locked even if that is not the case. Having each method to pass the exception (instead of catching it locally) to the caller, by appending `throws InterruptedException` to its signature, messes up the code too much. In practice, the result is often empty catch-clauses with an intention to add the code later, which then never is done and the application silently compiles and runs, but interrupting it may result in disaster. The recommended approach therefore is to always convert an `InterruptedException` to a suitable subclass of `Error`. In the example below we throw a general error:

```
try {
    argout = methodThrowingInterruptedException(argin);
} catch (InterruptedException exc) {
    throw new Error(exc.toString());
}
```

which means that the method containing the above code is not defined to throw any `InterruptedException`, and the caller(s) of that method do not need to be cluttered with try-catch clauses. Recall that the intended use of the Error class is for throwing severe exceptions that the user is not expected to catch, but he/she can catch them if that is needed. Hence, throwing an error when a resource allocation request is interrupted agrees well with the Java API conventions.

A different case is when an independent sequence of operations is to be interrupted. Sequencing on an application level (that is, advancing the state of the application, not caring about advancing the program counter for each instruction) is typically done by waiting for time to pass, or on certain conditions to be true. Interrupting such a thread, and letting the interrupted thread handle the Exception/Error, can be a perfectly acceptable thing to do:

- If the interrupted sequence is a web-server transaction, closing files and returning any resources in their initial state should be possible and straight forward.
- If the operation of an industrial robot is interrupted because it detects a human worker within its working range, ongoing motions have to stop but the work-piece should not be dropped, which requires engineered exception handling and not simply termination of the thread.

Still, it is better to throw an Error which is much more likely to reach the outer level of error handling despite 'lazy' (or 'too busy') programmers. Do like in the code example above.

Aliveness

Since interrupt is more of an order to commit suicide than actually interrupting, the interrupted thread may continue for quite a while before it is terminated/dead. It is sometimes useful to know when the thread really has stopped its execution. That is done by calling the isAlive method of the thread object. A live thread object is what we call an *active object*.

Notice that even if a thread has terminated, the object as such remains (as the remains :-) but after termination, even when isAlive returns false, it cannot be brought back to life by calling start again. Thus, a thread object can only be started once. However, trying to restart a terminated thread is silently accepted with no errors or exceptions; you have to keep track of the thread status yourself.

If calling `isAlive` returns `true`, the thread was alive at the time of the call but it could be dead right after if it was just about to terminate; we would need to lock the object and check both aliveness (via `isAlive`) and termination status (as determined by isInterrupted and user code) if we need to be sure that the thread still is running (accepting new orders etc.). If calling `isAlive` returns `false`, the thread either has not been started, or it has terminated. If a call of start results in `isAlive` returning `true`, it was not started. Otherwise, it was terminated.

Joining

You normally do not call isAlive because normally a thread is started right after the thread object was created, and if you need to await the thread termination you preferably suspend execution until that happens (not wasting CPU-time on polling the aliveness status). For this purpose, there is a method join which blocks the caller until isAlive of that thread object returns false. Details:

- Calling `join` of a thread object which already has terminated simply returns without blocking.
- A thread object calling its own `join` method (`this.join()` or `currentThread().join()`) will put itself into a state that we could call 'coma'; alive sleeping until the death, unless interrupted which will permit recovering.
- Joining a thread object that has not been started returns directly (without errors) since it is not alive (yet).

In practice, in your own code if you are aware of the limitations, thread termination is not a problem. To specify a time-out for joining a thread, there are versions of join with time-out arguments.

Inappropriate thread methods

Among the useful and well designed methods of class Thread, there are also some 'features' that resembles 'bugs' and should be avoided.

First there is a method destroy which is supposed to actually interrupt and kill the thread. However, allocated resources could then not be returned since that (for run-time reasons) has to be done by the thread that is going to be killed. Since destroy most likely would leave the application in a badly working state, it should not be called. It is even written in the Java2 documentation: “This method is not implemented”.

There is also a method stop which in the Java2 API documentation is described as “*inherently unsafe ..., potentially resulting in arbitrary behavior*”. Its purpose is/was to force the thread to stop executing. It is unsafe for the same reasons as destroy is. Actually calling stop will permit some cleanup and return of resources to be done, but that exhibits undefined timing and resource management on the user level cannot (in practice) be handled in a system routine such as stop.

In the beginning of the real-time programming subject, students often want to use something like stop (or even destroy) instead of dealing with actual execution and OS properties. It is hard to understand why the Java engineers @sun.com introduced stop in the first place.

Finally, there are two methods that could have been useful if designed properly: suspend and resume. The problem is that, as their names suggest, suspending and resuming execution was accomplished as a mechanism well separated from the ordinary thread management. With two unrelated ways of suspending execution, there was a big risk for a set of threads to be (depending on timing) permanently blocked. These methods and their deficiencies are outside the scope of this text.

1.6 The Thread class

We are now ready to sum up the content of class lava.lang.Thread which in a abbreviated form can be expressed as shown below(all shown members are public which is omitted, and deprecated methods are not shown). The task to be carried out by the thread is implemented in the run method, either by subclassing Thread or by implementing the Runnable interface which simply states the presence of the run method:

```
public interface Runnable {
    void run();
}
```

When implementing Runnable you need to call Thread.currentThread() to get to the public non-static methods of class Thread, which in short includes the following:

```
public class Thread implements Runnable {

  static int MAX_PRIORITY;      // Highest possible priority.
  static int MIN_PRIORITY;      // Lowest possible priority.
  static int NORM_PRIORITY;     // Default priority.

  Thread();                     // Use run in subclass.
  Thread(Runnable target);      // Use the run of 'target'.

  void start();                 // Create thread which calls run.
  void run() {};                // Work to be defined by subclass.
  static Thread currentThread();// Get currently executing thread.

  void setPriority(int pri);    // Change the priority to 'pri'.
  int getPriority();            // Returns this thread's priority.

  static void sleep(long t);    // Suspend execution at least 't' ms
  static void yield();          // Reschedule to let others run.

  void interrupt();             // Set interrupt request flag.
  boolean isInterrupted();      // Check interrupt flag of thread obj.
  static boolean interrupted(); // Check and clear for current thread.

  boolean isAlive();            // True if started but not dead.
  void join();                  // Waits for this thread to die.
  void join(long t);            // Try to join, but only for 't' ms.
}
```

Note that the Thread class does not relate to real time. Instead, the real-time clock is, as mentioned, obtained by calling `System.currentTimeMillis();` which returns the value of the real-time clock as a long expressed in milliseconds. See the JDK class documentation for further information.

2 Resources and mutual exclusion – Semaphores

Semaphores are of interest as a basic mechanism that is available in one form or another in all (?) operating systems and real-time kernels. In some industrial systems, the only available mechanism is semaphores, on which other concepts could be built. Also historically, the semaphore was the first (introduced by Dijkstra in 1968) principle for efficiently handling mutual exclusion and synchronization, solving problems such as the bank account transactions described earlier.

A *Semaphore* is a mechanism that can be used for implementation of the fundamental abstractions shown in the previous chapter. Such a mechanism requires certain features from the operating or run-time system, and cannot be purely implemented in, for instance, Java without help from native systems calls. Therefore, the semaphore *methods* we are about to introduce *should be considered as system calls* (part of the operating or runtime system) with certain properties, not to be confused with ordinary methods that can be implemented on the Java level.

We may, on the other hand, be able to implement semaphores in Java if we can use some built-in or native Java feature (synchronized methods in this case) to accomplish suspension of execution until certain conditions are fulfilled. Alternatively, some way of implementing atomic methods could be available (such as native methods for disabling and enabling interrupts). An atomic method is a method that cannot be interrupted by execution of other parts of the software. Anyhow, no matter how we implement semaphore mechanism, natively or built on some Java feature, we think of the methods as atomic and with properties according to the sequel.

2.1 Semaphore basics

Recall that semaphores are the primary mechanism in many small embedded systems, so we need semaphore classes that model the semaphore concept and supports cross compilation to native code. Linking with native implementations of semaphores as provided in most real-time kernels then permits full efficiency also when special hardware is used. To this end, let us first study Java semaphores on the concept leveland then look into the details.

Core operations

In Java, the class `Semaphore` implements a traditional semaphore. Such a semaphore is basically a non-negative integer-valued counter with two atomic methods and a queue for blocked/waiting threads. The core methods (with return type void) of `Semaphore` are:

`acquire();` decrements the counter if it is positive, but if the counter is zero, execution of the caller is suspended and that thread is put in some kind of queue waiting for the counter to become positive. Hence, an operation that increments the counter must check the queue and wake up the first thread.

`release();` increments the counter and checks if there are any threads waiting in the queue. If so, one of those threads is made ready for execution, which can resume when the caller has completed its (atomic) call of release.

Apart from some initialization of the counter when the semaphore is created/initialized, these two methods are the only two methods available according to the original definition of a semaphore. When a blocked thread is placed in the waiting queue, the order depends on the scheduling of that particular run-time system. Normally, threads are ordered according to priority, and equal priority threads are placed in the order they arrived (FIFO).

Pseudo code implementation of a basic semaphore

For a better understanding of the behaviour of the basic semaphore primitives we here give a sketch of how a semaphore could be implemented using pseudo code.

```
class Semaphore {
  private int count;
  public void acquire() {
    while (count<1) {
      "suspend execution of caller";
    }
    --count; // Got the semaphore, continuing
  }
  public void release() {
    if ("anyone suspended") {
      "resume the first thread in the queue";
    }
    count++;
  }
}
```

Supporting timeout

In addition to the core methods, for convenience, the following method (return type boolean) is also defined:

tryAquire(long timeout, TimeUnit unit) The same as acquire, but the caller gives up aquiring the semaphore after the number of time units specified in timeout. The time unit is specified by unit. If the semaphore was taken, true is returned. Otherwise false is returned.

Using this method requires some care. First, there is no guarantee that the caller is unblocked exactly after the timeout time, it can be later (but not earlier). The release time may depend on what other threads that have issued blocking calls with timeouts, the CPU load, etc. Thus, the same type of problems as with the Thread.sleep method.

Secondly, the returned boolean value must be taken care of (which is not checked by the compiler in Java, just like in C/C++/C#). If false is returned, perhaps after a long time of operation, and not considered due to a programming error, shared resources can by accident be accessed without mutual exclusion. The name `tryAcquire`, instead of overloading on acquire for instance, is emphasises the characteristics of the operation.

Finally, consider if your design is right if you use the `tryAcquire` operation. It should be used instead of implementing additional timer threads, but not for cyclic polling of the semaphore value. Some systems provide a way of polling the value without actually trying to acquire the semaphore. Since such a method easily could be misused by unskilled programmers, such a method is not provided here.

2.2 Semaphore reference

The `Semaphore` class provides a number of variants of the `acquire` and `release` methods but here we only present the ones we need in the course. See the online Java documentation for a complete presentation of the available methods.

The `Semaphore` class is available in the `java.util.concurrent` package so to use it make sure you include it using:

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

In this way you also get access to the `TimeUnit` enum used to specify time units.

Constructors and methods

`public Semaphore(int permits)` Creates a semaphore with a starting value of permit.

`public void release()` Increments by 1 the counter represented by this semaphore, and resumes one, if any, of the waiting threads.

`public void acquire() throws InterruptedException` Causes the calling thread to block until the counter that represents this semaphore obtains a positive value. On return the counter, named count below, is decremented by one. Throws an InterruptedException if another thread calls `interrupt()` while the calling thread is blocked in aquire or the calling thread is already has its interrupt flag set.

`public boolean tryAcquire(long timeout, TimeUnit unit) throws InterruptedException` Causes the calling thread to block until the counter that represents this semaphore obtains a positive value (just like the ordinary acquire method) or until the timeout time has passed. Throws an InterruptedException if another thread calls `interrupt()` while the calling thread is blocked in aquire or the calling thread is already has its interrupt flag set. The time unit for the timeout value is given by unit. `TimeUnit` is an enum (enumerated value) that can have the following values: TimeUnit.DAYS, TimeUnit.HOURS, TimeUnit.MINUTES, TimeUnit.SECONDS, TimeUnit.MILLISECONDS, TimeUnit.MICROSECONDS, or TimeUnit.NANOSECONDS.

Note that the semaphore operations are not ordinary methods; they have to utilize some type of system call to suspend and resume execution. That is, when acquire results in suspended execution as mentioned in the comment above, the thread is no longer ready for execution and the scheduler of the operating system (or JVM) puts the thread (in terms of data including the context) in the waiting queue of the semaphore. Hence, the thread consumes no CPU time while being blocked.

Multiple step semaphore use

In some situations it can be useful to atomically increase/decrease the value of the semaphore by more than one, for example to reserve a number of resources in one step. For this the `Semaphore` class has variants of acquire, tryAcquire, and release that takes an extra parameter indicating the number of steps the semaphore should be increased/decrease. See the online Java documentation for more information on these methods.

2.3 Locks

The `Semaphore` class is a generally applicable semaphore, but often all you need is a mechanism to achieve mutual exclusion. Mutual exclusion is desired when several threads access the same resource and simultaneous use of the resource would cause incorrect results. Using a semaphore to achieve mutual exclusion would require you to create a semaphore with a starting value of 1 and surround your mutual exclusion code with calls to acquire and release while taking into account that acquire can throw an InterruptedException you need to handle, i.e. something like:

```java
Semaphore mutex = new Semaphore(1);
...
try {
  mutex.acquire();
  /* Mutual exclusion code here */
  mutex.release();
} catch(InterruptedException e) { /* Handle InterruptedException here */ }
```

Instead a `Lock` can be used to simplify the code. `Lock` is a Java interface which is implemented by the class `ReentrantLock`. The above code would now look like the following:

```
Lock mutex = new ReentrantLock;
...
mutex.lock();
/* Mutual exclusion code here */
mutex.unlock();
```

`Lock` and `ReentrantLock` are available in the `java.util.concurrent.locks` package.

2.4 General disadvantages of semaphores

When using semaphores, one should also be aware of the drawbacks (both in general and compared to other techniques introduced later). If semaphores are inlined in the code wherever mutual exclusion is to be accomplished, it gets hard to read and maintain the program. If it is not clear where the critical sections actually are, the risk for shared data not being locked increases. Another case is statements like if, throw, and return unintentionally results in a resource being left in a locked state, and the entire application gets locked up or does not work anymore. A first good rule therefore is to place critical sections as methods in separate classes.

Another disadvantage is when a thread temporarily has to leave a critical section, waiting for some condition to be fulfilled, it can in some cases be rather complicated to implement. The semaphore concept is as such sufficient for implementation of any multi-threaded program, but the application may be unnecessarily complex compared to using monitors or messages as explained below. Therefore, semaphores are best suited for low-level and small software modules with special demands on efficiency.

3 Objects providing mutual exclusion – Monitors

The key idea with the concept of monitors is to combine object oriented programming with mutual exclusion. An object with methods that are mutually exclusive is called a monitor. Actually, the monitor concept as initially defined by Hoare in 1974 was only based on abstract data types, but classes as in Java accomplish that and more. Monitors do in principle, however, require the mutual exclusion to be implicit. That would mean, in Java, that the class defining the monitor should be tagged to comprise a monitor (for instance by using a keyword monitor preceding the class), which should imply mutual exclusion between all methods and all fields being protected. For performance reasons, that is not the way monitors are supported in Java. Instead, the following applies.

3.1 Synchronized

When threads concurrently call methods of an object, the calls are asynchronous since they can occur independently without any synchronization. When, on the other hand, calls are to be mutually exclusive (as for the bank account) we must restrict concurrent calls so that they are not asynchronous with respect to each other. In other words, methods need to be called synchronous, which in Java is accomplished by declaring them as synchronized.

Synchronized methods

All methods that should be mutually exclusive has to be declared synchronized. Methods that are not synchronized, as well as the data fields (or attributes), only obeys the usual protection rules and are from a concurrency point of view depending on programming discipline. It could be considered as a design flaw of the Java language that *a class as such cannot be declared synchronized*. Such a solution could have ment that all methods are implicitly synchronized and only synchronized methods can access the attributes. But, as mentioned, this is not the case in Java; you have to be aware about how concurrent access can be made and implement classes so that they are *thread safe* (reentrant methods and mutual exclusion wherever multiple threads can call) if that is required.

A design decision not to make an object/package/library/application thread safe should be well documented. One such example is the so called Swing graphic package (`javax.swing.*`), which is basically not thread safe; having all those methods synchronized would be a waste of GUI performance since almost all calls are made sequentially by a single thread anyway, which is accomplished by converting the concurrently occurring user actions into EventObject:s that are buffered and processed one at a time. While event processing is the topic of the next section, we here only consider the case when concurrently executing threads actually need to call methods of the same object.

Similar to when we use a `Semaphore` (or a `Lock`) as a lock to provide exclusive access to an object, we need some kind of lock also when methods are synchronized. In Java, there is such a lock available implicitly in each object, even for the ordinary ones! The synchronized keyword tells the compiler to generate code that uses the lock. Thereby, the program gets more easy to read and write. For instance, consider the bank account in Figure 2.1.

Synchronized blocks

While the use of synchronized methods improves readability (compared to the use of semaphores), Java also provides a less (compared to proper use of monitors) structured alternative for locking an object; the *synchronized block*. Within the braces the object obj is locked for exclusive use:

```
class Account  {
    int balance;
    Semaphore mutex = new Semaphore(1);
    void deposit(int amount)
        mutex.acquire();
        balance += amount;
        mutex.release();
     }
    void withdraw(int amount) {
        mutex.acquire();
        balance -= amount;
        mutex.release();
     }
 }
```

```
class Account  {
    int balance;
    synchronized void deposit(int amount) {
        balance += amount;
     }
    synchronized void withdraw(int amount) {
        balance -= amount;
     }
 }
```

Figure 2.1: The bank account class to the left uses a semaphore for locking the object to accomplish mutual exclusion (ignoring that we in reality also would have to handle an InterruptedException thrown by the calls to acquire). By declaring the methods synchronized, as to the right, the same thing is achieved but in a simpler and more readable manner. In Java, there is an invisible lock in every object.

```
synchronized(obj) { /* Exclusive access to obj here.. */ }
```

Of course, a synchronized block uses the same object lock as the synchronized methods of that object, thereby making the block and the methods mutually exclusive. A synchronized method is equivalent to a method with the outermost block

```
{ synchronized(this) { /* Method body here. */ } }
```

The benefit of language support

Both synchronized methods and blocks are very convenient when returning from a method. Compare this with the use of semaphores which resembles the situation when the language/compiler does not support synchronization or concurrency, like in C/C++. Returning a value from a monitor method often requires the return value to be declared as an additional local variable, called ans in the following implementation of the `class Account`, not permitting the account to be overdrawn:

```
class Account {
  int balance;
  Semaphore mutex = new MutexSem();

  void deposit(int amount) { // As in implemented above.... }

  int withdraw(int amount) {
    int ans = 0;
    mutex.acquire();
    if (amount > balance) {
      ans = amount - balance;
      balance = 0;
    } else {
      balance -= amount;
    }
    mutex.release();
    return ans;
  }
}
```

Based on this code, consider the following:

- Having the local variable ans initialized and returned no longer makes the semaphore operations that clear as a scope for the critical section.

- Line 8 and 9 could be swapped, technically, but what would happen if we swap lines 16 and 17?
- If the introduced if-statement would be inserted in a larger method, there is the risk that lines 10 to 15 would have been implemented as

  ```
  int withdraw(int amount) {
    mutex.acquire();
    if (amount > balance) {
      balance = 0;
      return amount - balance;  // Bug one.
    } else {
      balance -= amount;
      return 0;                 // Bug two.
    }
    mutex.release();
  }
  ```

  and the resource would have been locked forever after the first call. That would be an easy to find bug, but what if the return from inside the method (between acquire and release) only occurs in a very special case as evaluated in a more complex if statement? Deadlock in the shipped product after a few weeks at the customer site?
- If the ans variable would be declared as an object attribute, like the balance attribute, we have a concurrency error. Mutual exclusion between deposit and withdraw still works (since ans is used only by the withdraw method) but concurrent calls of withdraw can give the wrong result. This is because the withdraw method no longer is reentrant. For instance, one caller (with low priority) locks the resource and carries out the transaction, while a second thread (with high priority) preempts the first thread between lines 16 and 17. When resuming, the first thread will return the result of the second call and the result of the first transaction is lost, that is, if amount>balance.

Making a method or function reentrant by only using local variables (allocated on the stack of each thread) is standard; we need to understand that no matter if we use a language supporting synchronized or not. However, with the availability of synchronized as in Java, we do not need to introduce the ans variable at all. Furthermore, returning from a method never results in the resource being left in its locked state; on return the Java compiler produces the executable code that unlocks the object.

Basic use of keyword synchronized

There are cases when synchronized blocks are appropriate, but in general we better follow a few simple rules:

1. For each class, decide if an instance should be an ordinary object (not caring about concurrency), a thread object, or a monitor. *Do not mix threads and monitors. I.e., a thread should not have any public methods except* `run()`.
2. *Monitors should have all public methods synchronized* (except the constructors) and no public attributes. Subclasses of monitors should also have to declare all methods synchronized since synchronized is not inherited.
3. If you need *to make an ordinary class thread safe*, you could create a monitor by defining a subclass, but then you have to override all methods in order to declare them as synchronized. Methods that are final is a problem. Instead of subclassing, it is usually better to *write a wrapper class being a monitor containing the ordinary object.*

4. *Do not use synchronized blocks*, which are contradictory to proper object-oriented design of concurrent software and to the monitor concept as such.

The reason for rule A is that the actions of a thread is defined via a method (run) which from a purely object-oriented point of view is a method like any other, with access to all attributes of the object. Mutual exclusion between the internal thread (executing the run method) and the external threads (calling the monitor methods) would require the run method to be synchronized as well. Another alternative could be to only access the attributes of the object via synchronized methods. However, the concurrency correctness would then heavily depend on programming discipline; neither the compiler nor the run-time system can complain about the run method accessing the attributes (run is nothing special from a language point of view). The public, protected, and private, keywords have to do with visibility between ordinary objects; they have nothing to do with concurrency. In case of mixing threads and monitors we get no help from the compiler as when we use synchronized methods to access protected attributes. The lack of support for the Task abstraction could be considered as a deficiency of Java, but as will be argued later, the task is not a suitable abstraction for the design of object oriented concurrent (or real-time) systems anyway.

Use and misuse of keyword volatile

To further describe the special need for concurrent software to be well-structured the following code snippet contains a synchronized method in a thread object. By breaking rule A above we may say that we have one error, and there is a risk that mutual exclusion will not work, but what other problems can result?

```
/*
 * Find four faults, concerning concurrency, in this class.
 */
import I.robot.dest; // Import static robot destination class Dest.

public class PathCoord extends Thread {
    private double x,y;
    public synchronized int dist() {return Math.sqrt(x*x+y*y);}
    public void int run() {
        double path = 0.0; // Path coordinate, locally on thread stack.
        while(!isInterrupted()) {
            path = Dest.step(path); // Local data and reentrant method.
            x = Dest.fx(path);      // Calling one thread-safe method,
            y = Dest.fy(path);      // and calling another one.
            Dest.nextStep(x,y);     // Use x,y and wait for next sample.
        }
    }
}
```

Even if this program is run on 64-bit machine (with the doubles being atomically assigned, more on that below), it can happen that a valid value x is not consistent with a valid value y (together defining an invalid coordinate outside the path for the destination of the robot motion). This is a concurrency error due to preemption (between or in the calls of fx and fy, since run as usually is not synchronized), independently of the types of x and y, and despite that fx and fy each are thread safe. This happens if another thread calls dist() after a new x value has been computed, but before the corresponding new y value has been calculated. Hence, this is a concurrency fault and the outcome of running the software is dependent on the execution timing and scheduling. In combination with breaking the rule not to mix threads and monitors, we may say this constitutes two errors in this class. However, there are also two other concurrency problems that are common in embedded software since most programmers also in industry are not aware of this:

1. There is the risk that x or y gets corrupted when the program is not run on a 64-bit machine (where a double is stored in a single machine word), which is the normal case for embedded software. For example, storing a 64-bit value on a 32-bit machine would require two separate memory store operations, and a context switch could occur inbetween the two rendering the 64-bit value only half updated. One remedy is to use the atomic classes that are part of Java from J2SE 5.0, but that is not easily applicable in other programming languages (and therefore not further treated here).

2. Even if we would use single precision (float) for x and y, thereby avoiding the problem of item 1 without using atomic classes (that decrease performance), and even if we never get a context switch at the critical point (e.g., due to the priorities and scheduler/OS used) between the calls of fx and fy, the result of another thread calling dist could still be wrong or outdated! The reason is that data can remain in registers of the CPU, as resulting from an optimizing compiler since we have not provided any information about the data being shared by multiple threads. The remedy here is to declare the attribute `volatile` like:

   ```
   private float volatile x, y;
   ```

   Problems arise when data shared by multiple threads is not volatile or is not protected with proper mutual exclusion (such as synchronized that results in the runtime system being informed and the contant of the cash is written to memory). Without volatile, a variable updated by one thread may remain in a register while another thread reads from the primary memory according to the address of the variable, and consequently the old value is obtained instead of the one updated by the other thread.

Even if there is really only one error in the `PathCoord` class above (the mix-in of the monitor in the thread), we have now listed four different faults that can be the result. Note that these problems all disappear if we access object attributes by methods that are synchronized, and also note that the first four problems illustrated in the class PathCoord are *the same if we use monitors or semaphores* for mutual exclusion, and the problems are *the same in Java as in C/C++*. Thus, independently of which of these languages you use, implement monitors and mutual exclusion as described[3], and **do not mix active objects and monitors**.

Synchronization details

As with any set of basic rules, there are some cases when the experienced programmer for good reasons, or the beginner for bad reasons, may want to break the rules. Before doing that, be aware of the previous section and the following:

1. As a way of doing manual performance optimization, small get and set methods (belonging to a monitor with other methods synchronized) are often *not* declared synchronized. However, the way it is done is (as far as the author has seen) mostly wrong, resulting in very-hard-to-find bugs when the program is run on some platforms. Therefore, keep the following in mind:

[3]When accomplishing mutual exclusion in Java via a proper monitor with synchronized methods, or when a program written in C uses semaphore functions from some library of the system, professional programmers are usually aware of the preemption aspect represented by the described concurrency fault (first error) of the PathCoord class. As a (positive) side effect the other possible runtime errors also disappear.

- Only access data that is consistent all the time. Even if getting an integer can be done atomically, that attribute may have an inconsistent value due to computations in the synchronized methods. In a non-final class with non-private attributes, you cannot know what subclasses will do.
- Only access a single single-word value (built in type or object reference) by a non-synchronized monitor method, and if not declared volatile, make sure that the updated value is obtained (by other system calls resulting in memory being updated, or by only permitting a single-threaded write during startup or similar). Even if you figure out that several attributes can be changed due do the logic of other methods, the risk is too high that errors are introduced, for instance when the class is extended or enhanced. An acceptable single-word access could be an object reference that is final (set in constructor) but needs to be frequently obtained by other threads.
- Attributes (except those of types double or long that need synchronization to be portable in any case) accessed without synchronization should be declared volatile. According to the Java language specification, all types except long and double are atomically accessed. For object references it is of course crucial that the address value does not get corrupted, which would break the built-in safety (the so called sandbox model) of Java. The only data-type you can assume being atomically accessed in native code on all platforms is the byte, so implementation of native methods (and virtual machines and Java native compilers) requires some care, in particular for CPUs with word length less than 32 bits.
- Comment your assumptions and decisions when you are doing this type of (what you think is) clever programming.

Hence, be careful when skipping synchronization; such optimization is better done by compilers and class loaders.

2. When developing software libraries or components that should stand also improper usage, such as a synchronized block locking a monitor outside the monitor class, you better use a private lock object which is locked by a synchronized block in each monitor operation. A monitor then looks like:

```
public class RobustMonitorExample {
    private Object lock = new Object();
    // More attributes are declared here ....

    public void methodOne(int x) {
        synchronized(lock) {
            // Attributes are accessed here....
        }
    }
}
```

   This way, other objects cannot interfere with the synchronization. Again, the unfortunate Java design decision to support synchronized blocks but not synchronized classes with appropriate semantics causes extra trouble when developing reliable software. Still, the situation is better than in other major languages. Recall this item when reading about wait and notify below. Another example will be the non-native Java implementation of semaphores.

3. Do not synchronize on public attributes. That is, if you (against the general guideline) synchronize on an attribute like the lock in the previous item, that attribute

should be private or final. Otherwise, that object reference may change from one synchronization point to another, thereby removing the synchronization. Locking cannot be ensured if the look can replaced by anyone.

4. Do not remove synchronization by subclassing (unless it is a final class and you really know what you are doing). If a base class declares a method as synchronized, the subclass should do so too. There is nothing in Java enforcing such a practice, and you might want to optimize methods in a subclass by omitting synchronized, but the risk for hard-to-find concurrency errors normally is too high considering the interplay between monitor methods.

5. If synchronization is needed between multiple monitors, such as interconnected buffers, they of course need an object in common to synchronize on. An alternative would be to merge the different monitors to form one monitor with all the functionality included, but suitable monitor classes may be available already; just the signaling and mutual exclusion between them need to be added. Using synchronized blocks to synchronize on some shared lock object (such as one of the monitors if that code has been prepared for it) is clearly preferable compared to rewriting the application.

As mentioned in the last item, the need for synchronization between objects can be due to the need for signaling (which we earlier accomplished by semaphores). When an object has been locked by synchronized (for a method or a blocks), there can be conditions that need to be fulfilled before the operations on the shared data can be fulfilled. Java provides the following support for that situation.

3.2 Conditions – wait and notify

Consider a buffer with operations put and get. To accomplish mutual exclusion during manipulation of the buffer, those methods are typically declared synchronized. However, both in this case as well as in many other types of monitors, certain conditions may need to be fulfilled before an operation can be completed. For instance, there must be something in the buffer to fetch before get can be performed, and the buffer may not be full in order to let put be carried out. If such conditions can be evaluated before any shared data is manipulated we could simply cancel the operation and make a new call when the status has changed, but we also require:

- When waiting for a certain condition to be fulfilled, that thread should be blocked to prevent waste of CPU time (polling or busy wait, possibly leading to starvation of other threads, is not allowed).
- When the condition is fulfilled we need a way of waking up the blocked thread(s).
- When a thread continues execution after being blocked on some condition, it has to compete for the resource again since we have to ensure that only one thread at a time is executing inside the monitor (i.e., inside the critical region).

To accomplish these features, each Java object is equipped with a wait/notify mechanism in terms of the methods wait, notify, and notifyAll according to (see Figure 2.2):

`wait()` The thread executing within the monitor (inside a synchronized method or block) gives up the exclusive access, stepping out to the 'back yard' containing the condition queue. The runtime system *puts the blocked thread in the condition queue*, where it typically is inserted in priority order with the thread with the highest priority first.

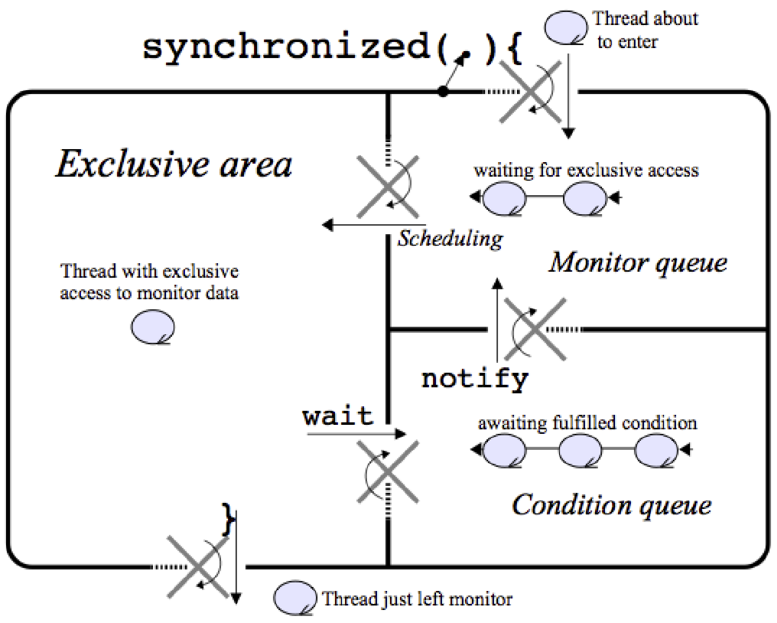


Figure 2.2: The monitor concept, including the wait-notify mechanism as a back yard. The crosses are revolving doors, that is, the door turns in the direction of the bent arrow and threads can only pass in the direction shown by the straight arrow.

The saved context of the blocked thread, referring to the stack of the thread and thereby to the state of execution, of course contains the state of the half made monitor operation.

`notify()` Threads waiting in the condition queue will remain there until notified by some other thread, or until a timeout occurs as explained later. Threads that inside the monitor change any data that affects conditions that the threads in the condition queue might waiting for, should *call notify to wake up one of the blocked threads*. If the condition queue is empty, notify does nothing. It depends on the runtime system which of the blocked threads that are selected for notification, but here we assume the normal case that it is the first in queue thread, which in turn is the one with the highest priority.

After the thread is notified and ready for execution, it has to compete for CPU time with all other threads to advance its execution point at all. When execution advances, the thread also has to compete (again!) with other threads for exclusive access to the monitor, and if not successful it is blocked and placed in the monitor queue (in priority order we assume, even if not specified by Java).

`notifyAll()` *All the threads in the condition queue are notified*, and thereby made ready for execution. They are then all subject to scheduling as explained for notify. Since scheduling is not part of the Java specification, and since knowing what threads that might be using the monitor requires global information, the reliable and portable way of notification is to always call notifyAll instead of notify; the use of notify is purely for optimization and performance tuning (to prevent excessive context switches that can be the case when many threads first are unblocked and then all but one calls wait again when the condition again is not fulfilled).

Note that even if the compiler lets you call these methods anywhere in the program (since they are public methods of class Object), they only work if the calling thread first has locked the object via synchronized. Otherwise, an `IllegalMonitorStateException` is thrown.

Be aware of the big difference between calling wait (i.e., Object.wait) and calling sleep (i.e., Thread.sleep) from inside a monitor, that is, from inside a synchronized method or block. Calling wait results in other threads being permitted to enter the monitor, like temporary unlocking to let some other thread change the state of the monitor as desired. Calling sleep means that the monitor is locked during the sleep time, and no other threads are allowed to enter during that time.

Basic use of wait and notify

A notified thread is moved from the condition queue to the monitor queue where it has to compete with other threads, as decided by the underlaying runtime system (here we assume priority ordered queues). When the thread holding the lock/monitor leaves, the first thread in the monitor queue is unblocked (put in the ready queue) and is subject to CPU time scheduling. Implications:

- No matter how low priority or what OS, a thread that got into the monitor will complete its monitor operation (critical section) if given CPU time and if not calling wait. We say there is *no resource preemption.*[4]
- Due to preemption (that is, execution preemption, not to be confused with the resource preemption), the thread holding the monitor will remain there if higher priority threads are scheduled for execution instead.
- There is nothing saying that a thread that has been notified, which means it has been in the exclusive are already, is given access again before any of the newly arrived threads in the monitor queue. This depends on the scheduler, however, and is not specified by Java.

The major point now is that *when a thread is continuing after wait, the condition that was waited for still cannot be assumed to actually be fulfilled.* As an example, assume one thread calling obj = buffer.fetch() to get the next object from a buffer, and that the buffer turned out to be empty. A consumer thread calling fetch should of course be blocked until some other (producer) thread puts another object in the buffer by calling buffer.post(obj). Correspondingly, a producer thread calling post is to be blocked until the buffer is not full. Assuming the simplest case with only one producer and one consumer, a straightforward but fragile implementation could be according to Figure 2.3. In a more general setting, with more threads involved or the buffer functionality being extended, the following problems arise:

1. Even if the notifying thread calls notify only when the condition to proceed is OK for the waiting thread, *the condition may again not be true when the waiting thread is continuing*! Assume we have one producer P and two consumers C1 and C2, using the Buffer like:
    - C1 tries to fetch an object, but gets blocked on since the buffer is empty.
    - P posts an object to the buffer, and since the buffer was empty that results in a notification of C1 that is moved from the condition queue to the monitor queue.

[4]In the original version of the monitor concept, proposed by Hoare in 1974, the notified thread did actually preempt the thread owning the monitor. Therefore, notify was only to be called as the last thing within the monitor operation. No such system is known to exist today.

```
class Producer extends Thread {
  public void run() {
    prod = source.get();
    buffer.post(prod);
  }
}
class Consumer extends Thread {
  public void run() {
    cons = buffer.fetch();
    sink.put(cons);
  }
}
```

```
class Buffer {
  synchronized void post(Object obj) {
    if (buff.size()==maxSize) wait();
    if (buff.isEmpty()) notify();
    buff.add(obj);
  }
  synchronized Object fetch() {
    if (buff.isEmpty()) wait();
    if (buff.size()==maxSize) notify();
    return buff.remove();
  }
}
// This Buffer is badly implemented!
```

Figure 2.3: Classes, with attributes and initialization left out for brevity, implementing producer-consumer communication via a bounded buffer based on java.util.ArrayList. The if (...) wait(); makes the buffer fragile: additional calls of notify (e.g., in other methods) or additional threads could course the buffering to fail.

- Before C1 is scheduled for execution, that is, before C1 actually enters the exclusive area, C2 (having a higher priority) continues (for example after a sleep that was completed after a clock interrupt) and tries to enter the monitor. If this happens before P has left the exclusive area, both C1 and C2 are blocked in the monitor queue. (C1 about to continue after wait, but C2 first in the queue due to higher priority.)
- When P leaves the monitor, there are one object in the buffer. C2 enters the exclusive area, gets the object, and when leaving C1 is moved to the ready queue of the scheduler and is allowed to enter the monitor (owning the lock).
- Since C1 is continuing after wait, according to the shown implementation assuming that the buffer is not empty, C1 tries to get the next object. There are no such object, and the buffer (and the application) fails[5].

2. It can be *difficult to keep track of what condition other threads are blocked on*, or it can be hard to verify correctness or revise the code. In the referred buffer example it may appear to be simple, but does the buffer work correctly even for maxSize equal to one?
   If we would add another method awaitEmpty that should block until the buffer is empty (useful during application shutdown), or an additional method awaitFull that should block until the buffer is full (useful for special handling of overflows), when and how many times should notify then be called?
   It is so easy to overlook some cases, and actually taking care of all cases by the appropriate logic (such as counters for threads blocked on this or that condition) leads to unnecessary complex code that is hard to understand and maintain.
   Remedy: *Use notifyAll instead of notify*, and let other threads reevaluate their conditions (assuming the remedy for item 1 is used). The exception from this rule is when optimizing for well known and performance critical parts of the application.

3. How do we ensure *concurrency correctness and predictability* on any (Java-compatible) platform? As an example, assume we have two consumer threads (C1 and C2) and five producer threads (P1 to P5), and assume these use our buffer according to the following:

[5]Programming in Java, application *failure* typically means an ArrayOutOfBoundException or a NullPointerException that can be caught and handled, or at least we know what and where it went wrong. Using an unsafe language such as C++, the system can crash (at a later stage) without messages.

- The buffer (Buffer in Figure 2.4) is given a capacity of four (`maxSize==4`).
- Both C1 and C2 call fetch, and hence they get blocked since the buffer is empty.
- P1 puts an object in the buffer by calling post, which calls notify. One of C1 and C2, say C1, is moved to the monitor queue while the other consumer remains in the condition queue.
- Before C1 gets scheduled for execution, the other producers (P2 to P5) try to post an object. Since the buffer was not empty, C2 was not notified. This is either a bug or C1 has to notify C2 somehow. However, the last producer, say P5, then also gets scheduled before C2 and post blocks since the buffer now is full.
- We now have two threads in the condition queue, one consumer blocked on buffer empty and one producer blocked on buffer full, and one of them will most likely be stuck there for a long time (possibly forever if it was the producer and if that thread was to serve the next application event). This holds even if we would have implemented the Buffer using while instead instead of if according to item 1, but the remedy of item 2 works.

Items 2 and 3 are similar in that they both motivate the use of notifyAll instead of notify. The difference is that while we according to item 2 actually may be able to optimize and use notify, fully tested to be logically correct on one JVM/OS, item 3 points at the concurrency problem that implies the risk of application failure sometimes on some platforms.

Another problem is that at any place within your program, where a reference to your monitor is available, synchronized(monitor){monitor.notifyAll();} can be called and interfere with your methods. The same of course applies to notify and wait. The remedies above help also in this case. For optimized libraries, however, private notification objects can be used, like the private lock object described in the above section about Synchronization details.

A radical remedy would be to always notify all threads whenever anything has changed. For instance, the Buffer class in Figure 2.3 could be implemented like in the left version in Figure 2.4, where the suggestions from the above items have been considered. Decreased performance comes from notifyAll (which is a system call that is more expensive than checking ordinary data) being called even if there for sure is no thread in the condition queue, and there are no logic separating between the full and empty cases. A reasonable trade-off (using notifyAll to avoid the described problems) is shown to the right in Figure 2.4.

In short, as rules of thumb for programming:

1. Always use while in front of wait: while (!okToProceed) wait();
2. Instead of notify(): use notifyAll();
3. Tune performance when needed, using notify() or condition objects explained below.

Timeout on waiting

There are situations when waiting for a condition to be fulfilled may not take more than a certain amount of time. After that time we want to get a timeout. For this purpose, there are two versions of wait accepting timeout arguments:

```java
class Buffer {    // Inefficient!!
  synchronized void post(Object obj) {
    while (buff.size()>=maxSize) {
      wait();
    }
    buff.add(obj);
    notifyAll();
  }

  synchronized Object fetch() {
    while (buff.isEmpty()) {
      wait();
    }
    return buff.remove();
    notifyAll();
  }
}
```

```java
class Buffer { // Well done.
  synchronized void post(Object obj) {
    while (buff.size()>=maxSize) {
      wait();
    }
    if (buff.isEmpty()) notifyAll();
    buff.add(obj);
  }

  synchronized Object fetch() {
    while (buff.isEmpty()) {
      wait();
    }
    if (buff.size()>=maxSize) notifyAll();
    return buff.remove();
  }
}
```

Figure 2.4: Two alternative implementations of the Buffer class. Declarations, initializations, and exception handling is omitted as in Figure 2.3. In the left version, all methods always call notifyAll (does not need to be last), possibly resulting in performance problems. In the version to the right, robustness and performance should be acceptable.

`wait(long timeout)` The thread executing within the monitor temporarily leaves the monitor, waiting for a notification as normal. If notify is not called within timeout milliseconds, we give up waiting and return to the caller anyway. The timeout is minimum value; it can be longer as an implication of the actual scheduling of involved threads and the internal timer. A timeout equal to zero is equivalent to calling wait without any timeout argument. A negative timeout results in an IllegalArgumentException.

`wait(long timeout, int nanos)` The same as previous wait but with the timeout time set to 1000000*timeout+nanos nanoseconds. Again, this is a minimum time that depends on the actual scheduling, if the resolution of time in our runtime system is high enough. However, with ms resolution or lower, the nanos are rounded to the nearest ms (!) which implies that the minimum timeout time can be up to 0.5 ms shorter than requested, but not shorter than the timeout since that value is rounded upwards if the clock resolution is lower than 1 ms.

Note the difference between calling java.lang.Thread.sleep(delay) from within a monitor, and calling java.lang.Object.wait(delay). The former keeps the monitor locked during waiting/sleeping, whereas the latter lets other threads access the monitor during the delay time. It is, however, difficult to ensure that wait(delay) actually delays for at least delay ms since notify in other methods (for example in unknown subclasses) can interfere. Therefore, a delay within a monitor should be implemented similar to:

```java
  synchronized monitorSleep(long timeout) throws InterruptedException {
    long tf = System.currentTimeMillis()+timeout;
    while ((timeout=tf-System.currentTimeMillis())>0) wait(timeout);
  }
```

Also note that if timeout milliseconds elapsed from the call of wait until the calling thread executes the instruction following wait, we cannot distinguish between if:

1. The wait was called and timeout ms later we continued after a timeout.
2. The wait was called and before the timeout time some other thread called notify, but due to scheduling more than timeout ms elapsed before the statement after wait was executed.

3. Other threads were scheduled for execution right before wait was actually called, then during the wait some other thread called notify, but due to the initial delay more than timeout ms elapsed.

Thus, even if it is only in case 1 that the runtime system issued a timeout, we can obtain waiting for that amount of time due to the (platform dependent) scheduling. In code this means, dt can be greater than timeout even if wait was completed due to a notification:

```
long t0 = System.currentTimeMillis();
wait(timeout);
long dt = System.currentTimeMillis()-t0;
```

Is this a problem? Normally not; it does usually not matter if we were notified or not, it is the inferred delay that matters for the calling thread. Hence, if dt>timeout it is considered as a timeout. On the other hand, if the state of the monitor stated that our thread was waiting, and some other thread calling notifyAll determined that it issued the notification on time (before a timeout occurred) and subsequently acted according to that, the possible cases 2 and 3 above may imply a need for some more monitor logic. That is, if it logically is not a timeout, only checking if dt>timeout after the previous code fragment is not sufficient; concurrency correctness may require some more monitor logic to cope with all possible interleavings, as always.

Interrupted waiting

When a thread that is blocked on wait (with or without timeout), or on sleep as described on page 36, the blocking is interrupted and an InterruptedException is thrown to the calling thread. The appropriate handling of such an interrupted wait is typically known/defined in the calling thread outside the monitor methods. Hence, it must propagate to the caller, so the obvious choice would be to let every potentially blocking monitor methods be declared to throw an InterruptedException. However, in particular when there are several levels of calls, this soon gets very messy. Furthermore, the monitor method cannot complete its operation if the condition is not fulfilled, and it cannot simply return before the operation is completed either (both these alternatives would violate the correctness of the monitor).

So, the only reasonable alternative is to throw a Throwable object that we do not need to catch or declare. Java provides two alternatives for doing that: throwing an RuntimeException which is an Exception, or throwing an Error. To emphasis that the normal user code is not supposed to catch an interrupted wait (since it in most cases needs to propagate to the outermost level of the thread), throwing an Error is the appropriate solution. That is, we throw a suitable instance of of a subclass to java.lang.Error.

The implication is that we normally call wait according to:

```
{ //.....
  while (!ok) {
    try {
      wait();
    } catch (InterruptedException exc) {
      throw new Error(exc);
    }
  }
}
```

Note that when an InterruptedException is thrown, the interrupted-flag of that thread is cleared. Hence, it is only the fact that we get to the catch clause that tells us that the thread is interrupted, and after the catch it is only the thrown Error that carries the interrupt information when the call stack is popped.

Sometimes interrupting a thread is used to actually interrupt a command or action in an almost expected way. That is, instead of cluttering the code with a lot of tests if the operation is to be aborted, we can let execution continue until it reaches a blocking call and then utilize the thrown Error. For instance, a robot performing its task according to a sequence of instructions can be interrupted by the operator who issues another task or simply stops the robot due to some fault. The thread that handles operator input can then interrupt the ongoing robot work, and since computers are faster than mechanics the work thread will be blocked on getting a new order or on the completion of last order. By catching InterruptedException and converting it to an Error in whatever monitor it occurs, we can then catch the Error and terminate the thread like:

```
class RobotDuty extends Thread {
  public void run() {
    try {
      while (doWork(Master.nextInstruction())) {};
    } catch (Error) { Master.workAborted();} // Acknowledge.
  } // No more work; end of duty!
}
```

Terminating the thread (and creating a new one for the next work) like in this example is convenient for handling sessions that have a finite life time, such as processing a batch recipe in a chemical industry, or carrying out an e-business transaction; the state of the session is kept by the thread, and both the activity and the state gets garbage-collected when the session is finished. In feedback control, on the other hand, we usually create the threads that execute during the entire lifetime/uptime of the system; the state of the controlled system needs to be maintained also over mode changes and the like, and the control can be sensitive to delays during creation of new threads.

3.3 More on monitor programming

The following applies to the use of standard Java packages.

Static monitors – locking class variables

Executing synchronized code results in the synchronized **object** being locked. That is, mutual exclusion is provided for the object (not the class) and concurrent access to different objects of the same class is, of course, still permitted. However, there may be attributes that are shared between all objects of a certain type, so called static attributes or class variables. Mutual exclusion then requires special care. Even if the object provides mutual exclusion on its attributes, for instance by having all attributes private and all non-static methods synchronized, the class variables are still not locked. The reason is that the class as such is another object which needs to be locked separately, either by a static synchronized method or by a synchronized(static_attribute){} block. In the same way, since a static method has no reference to 'this' object, a static method does not provide mutual exclusion for any non-static methods. Hence, locking for classes (static attributes) and objects are different things, implying that a inside a synchronized method locking either the class or the object, you have to put a synchronized block for explicitly locking the other, if both the class and object need to be locked at the same time.

Monitors versus semaphores

The monitor concept can be accomplished in different ways. In Java, monitors are not exactly supported, but synchronized methods (and a few simple rules) let us conveniently accomplish monitors. In practice, we may say that Java provides monitors, as well as possibilities to avoid (by omitting synchronized) the overhead of object locking when we (for sure) can deduce that the program will be correct even without locking. Anyhow, having synchronized built into the programming language is of great value. In languages without such support, we have to use library functions in combination with programming conventions.

Monitors and semaphores have equal expressive power; a program using on of the mechanisms can always be rewritten using only the other mechanism, but it can be hard in practice.

- Since a semaphore class is easily implemented by a monitor (providing the methods acquire and release) it is usually straight forward to rewrite the program using only monitors. However, if an interrupt routine (a static method without arguments that is connected to a hardware interrupt) is to call release (e.g. to signal that data is available in a hardware buffer), we cannot use an implementation that is based on synchronized (since an interrupt cannot be blocked even if the synchronized object is locked).
- Implementing monitors using semaphores is troublesome when wait/notify is used. The problem is that the waiting thread has to acquire one semaphore (to be used for signaling when the condition is fulfilled) and release one semaphore (the mutex to let other threads access shared data) *atomically as one operation.* The solution is to let each thread have its own signaling semaphore.

Clearly, both semaphores and monitors are needed, but in different situations.

Polling locking state

Since JDK1.4, class java.lang.Thread contains a method

```
  static boolean holdsLock(Object obj)
```

that returns true if and only if the current thread holds the monitor lock on the specified object. Do not use this method, except for special cases such as test-cases where the locking of objects need to be verified. In particular, keep in mind that even if holdsLock returns one value, the opposite could be the case one machine instruction (of that thread, and a few context switches) later. Just like there (on purpose) are no methods for obtaining the state of a semaphore (without trying to acquire it), to avoid bad solutions on concurrency problems, you should not design your application such that holdsLock is needed.

4 Message-based communication – Mailboxes

Instead of the low-level characteristics of semaphores, the desire to support concurrency in an object oriented way led to monitors; passive objects used to implement collaboration between active objects. In many cases, however, we can benefit from having such passive objects with additional functionality for:

Buffering: Often threads work in producer-consumer like situations when transmitted objects need to be buffered in some type of queue. That lets the threads work more asynchronously with respect to each other. That is, even if there are some temporary blocking when posting/fetching objects into/from the queue (to accomplish mutual exclusion during the assumably short queuing operations), a buffer overall lets the threads operate more independently of each other.

Activity interaction: Traditionally, before the email era, messages were sent via mailboxes (passive objects providing buffering) and distributed to the mailbox of the addressee. If the sender and receiver are close to each other (like threads in the same JVM), and the sender knows (has a reference to) where the receiver is, the overhead of distribution (cost of the stamp and mail delay) can be avoided by putting the message directly in the receivers mailbox. Anyhow, even if the messages (transmitted objects) are buffered in a passive object (the mailbox), the aim is to send it to some other active object (person), which deserves special support.

Distribution: If the sender and the receiver of a message are not close to or not aware of each others location, it is desirable to have some mail distribution system. Even if the topic of distributed systems is outside the scope of this chapter, communicating via messages makes us prepared for the situation when each message has to be sent via ordinary mail; the message can look the same, only the mailing differs. If we would call the receiving active object directly, the addressee would need to be available at that particular time. Therefore, communicating via messages can be a good generic solution.

Encapsulation: Data protection (using private or protected in Java) ensures encapsulation of sequential programs, which means that objects will be ensured to function independently of each other. For real-time programs that is not enough; when methods of another object is called from a time-aware thread (looking at the clock frequently), the caller “gives up it’s own will” and commits itself to performing the instructions given in the (hidden) implementation of the called method, which can result in blocking for a long time. Hence, threads give up their timely behavior unless communication is performed differently.

To improve on these aspects, communication via messages (instead of via methods and shared data) will be introduced in this section. Originally and traditionally, this was referred to as *mailboxes*.

From one point of view, a so called mailbox is nothing but a special type of monitor, so we do not need any additional support from the language (no new keywords like synchronized which supports monitors). From another point of view, when the above aspects are important, message based object interaction can be considered to be fundamental, so let us start with some considerations about the abstraction we deal with.

4.1 More on object interaction

Method calls in a language/system providing only passive objects are by definition a synchronous type of communication. That is, passive objects work synchronously since they share the same thread of execution. In other words, this type of communication between objects is synchronous since the called object remains in its well defined state during the data transfer.

During a method call, the caller enters the scope of the method within the scope of the called object, performing exactly what the method states. In a single-threaded system this is fine; data access via methods provide encapsulation and data abstraction. When all objects are executed by the same thread, we actually do not think of any thread at all. Data is transferred via arguments and return values, in both directions between the objects, which is straightforward and efficient in single-threaded applications.

To support multi-threaded applications, the programming language could provide active (concurrently executing) objects, and models other than method calls for data transfer between objects. In fact, within object-oriented programming in general, communication between objects are (as in the language Smalltalk) often referred to as message passing. In Java, object interaction stems from the traditional (computationally efficient) C/C++ approach with method calls which are basically the same as function calls in C. Hence, in Java as well as in most other languages, asynchronous communication should be accomplished by a class implementing some type of mailbox or buffer supporting the principle of message passing.

Thinking about real-life actors, sending a message is a natural way of transferring information, in particular when the sender and the receiver are acting concurrently but not synchronized (that is, performing actions independently of each other). The message need to be temporarily stored (buffered) somehow after being sent until being received. Today, information is most often stored and transmitted electronically. On the Internet, messages between humans are mainly accomplished by e-mail, but in technical systems we also use many other message principles depending on application and development needs.

4.2 Events and buffers

Assuming the sending and receiving thread belong to the same program, which is the case in this entire chapter, the sender and receiver can refer to shared objects. Thus, the buffer we need can be a monitor. We may compare with messages within user interaction packages such as the AWT. Such messages are called events since they represent an application event, but they are not buffered except before being fed into the application. Additionally, the AWT InputEvents are time-stamped so the graphics classes and the application can react accordingly to when the event (such as a mouse click) occurred physically, which can be quite different from when the event is processed (in particular on a heavily loaded system).

For instance, assume two mouse clicks should be considered a a double-click if there is less than 1 s between them, and that the user clicks twice with the second click 0.9 s after the first. Then if there is a 0.2 s delay right after processing/registering the first click, the second click will be processed 1.1 s after the first one. Using the time-stamps it can still be deduced that the user actually double-clicked.

Similar situations appear in feedback control where the delay (difference between current time and event time) results is so called phase lag, which in turn results in bad damping or even instability. Since our aim is both concurrent and *real-time* programming, time-stamping our messages (at the time of creation) is often preferred.

4.3 Mailboxes in Java

It is now time to move on to describe how mailboxes are supported by the standard Java library and to give some advise on how to use them.

The BlockingQueue interface

The key mechanism for mailboxes is the `BlockingQueue` interface. It represents a buffer in which a thread can put messages of an arbitrary type which can later be retrieved, possibly by another thread, in a first-in/first-out order. In this way, it can be used for asynchronous communication between two threads in the form of message passing. The central functionality of the interface is (see the online Java documentation for a full description):

```
public interface BlockingQueue<E> extends Queue<E> {
  // --- methods for sending a message ---
  void put(E e) throws InterruptedException;
  boolean add(E e);
  boolean offer(E e);
  boolean offer(E e, long timeout, TimeUnit unit)) throws InterruptedException;

  // --- methods for receiving a message ---
  E take() throws InterruptedException;
  E poll();
  E poll(long timeout, TimeUnit unit) throws InterruptedException;
}
```

As you can see, the interface is generic which means that you can use it to send messages of any type, The type parameter `E` represents the type of the messages sent.

The many alternate ways of inserting and retrieving a message might seem a bit confusing at first. The difference between them is how they handle the case where the operation cannot be satisfied immediately. Such situation includes attempting to retrieve a message from a buffer that is empty or trying to insert a message into a buffer with no spare room for new messages. Let us go through their behaviour one by one.

put(e) Inserts the message into the queue. If the queue is full, it blocks the calling thread until space becomes available and the method can be completed.

add(e) Inserts the message into the queue if there is space for the message in the queue. Returns `true` if the operation succeeds[6]. Otherwise an `IllegalStateException` is thrown.

offer(e) Inserts the message into the queue if there is space for the message in the queue. Returns `true` if the operation succeeds and false if it does not (full queue).

offer(e,timeout,unit) Like `offer()`, but waits up to the specified amount of time for space to become available in the queue if it is full[7].

take() Fetches the oldest message in the queue if any is available. If the queue is empty, the calling thread is blocked until a new message arrives.

poll() Fetches the oldest message in the queue if any is available. Otherwise returns `null`.

poll(timeout,unit) Like `poll()`, but waits up to the specified amount of time for a message to arrive if the queue is empty.

[6]Interestingly, the method never returns false. This has to do with how add() is specified in the interfaces it is inherited from.

[7]See the semaphore reference on page 41 for a description of how to specify the timeout.

BlockingQueue implementations

In order to actually create a mailbox we need a class that implements the `BlockingQueue` interface. There are several such classes available in the standard Java library. Let us look at a few of them and their properties.

LinkedBlockingQueue This type of queue ()implemented using a linked list) has no limit on the number of messages it can hold (other than the limit set by available memory). This means that you do not have to handle the case of the queue becoming full in any special way. We can then safely use the nonblocking `add()` or `offer()` methods to insert a new message without worrying about having to catch a possible `interruptedException`. This could seem advantageous and in many cases it is, but it also has its drawbacks. Often threads communicate with each other according to a producer-consumer pattern. I.e., on thread produces data and sends it on to another thread as messages in a mailbox. If for some reason the consumer does not keep up with the producer, the queue will quickly fill up with large amounts of unprocessed data. Depending on the timing and memory requirements of the application this can be a minor or a major problem. In a realtime control setting this is usually unacceptable. In order to produce a robust system we therefore need a way to throttle the production of new messages in order to avoid overload. This can be achieved by instead using a bounded queue which limits the amount of messages in the queue.

ArrayBlockingQueue This class represents a bounded queue implemented using an array. Being a bounded queue makes it more suitable for systems with hard realtime requirements on timing and memory use. You provide the maximum number of messages the queue should be able to hold as a parameter to the constructor. Here, we typically use `put()` to send a message and `take()` to receive it.

PriorityBlockingQueue An unbounded blocking queue similar to the `PriorityQueue` you might be used to from earlier courses.

Messages and timestamps

As discussed in Section 4.2 it can sometimes be advantageous to work with messages that have a timestamp. One way of achieving this is to define a superclass for your message classes as follows.

```
public class TimestampedMessage {
    private long timestamp = System.currentTimeMillis();
    public long getTimestamp() { return timestamp; }
}
```

Then let your message classes inherit from this class and they will automatically be timestamped with their creation time.

A thread with its own mailbox

A mailbox is a very general abstraction that does put very few restrictions on which thread actually puts messages into it and which thread later retrieves the message from it. You could let several threads fetch messages from the same mailbox, although you would have little control over which thread actually gets which message. In some situations this makes perfect sense, for example if you have a number of equivalent worker threads that receives requests via a common mailbox, thus sharing and parallelizing the workload. In most situations we find, however, that each thread needs its own mailbox since we need to have control over which thread receives each message. Compare with sending letters using traditional post or for that matter, sending an e-mail. You typically address it to a particular person. In the same way, we typically want to send a message to a particular thread. In order to do this we for each thread need to also create a separate mailbox and associate it with the corresponding thread. In order to send a message to a thread the sender also need to hava a way of knowing which mailbox belongs to the addressee. This situation is so common that it might be a good idea to write a special variant of the `Thread` class that does this for us. It could look like this:

```
public class MessagingThread<M> extends Thread {
  private final BlockingQueue<M> queue;

  /** Creates a new thread with a mailbox capable of holding 'size' messages.
       To invoke this constructor call 'super(size);' in the constructor
       of your subclass. */
  public MessagingThread(int size) {
    queue = new ArrayBlockingQueue<>(size);
  }

  /** Called by another thread, to send a message to this thread. */
  public void send(M message) throws InterruptedException {
    queue.put(message);
  }

  /** Returns the first message in the queue, or blocks if none available. */
  protected M receive() throws InterruptedException {
    return queue.take();
  }

  /** Returns the first message in the queue, or blocks up to 'timeout'
        milliseconds if none available. Returns null if no message is obtained
        within 'timeout' milliseconds. */
  protected M receiveWithTimeout(long timeout) throws InterruptedException {
    return queue.poll(timeout, TimeUnit.MILLISECONDS);
  }
}
```

Note that the methods for receiving messages are declared protected. This is to prevent other classes from stealing messages from the mailbox. Another thing that the observant reader might note is that the code breaks the rule that says that a thread should not have any public methods except `run()` (see page 46). That is true, but since the concept of sending a message to a particular thread is so strongly connected with the thread as such, it seems reasonable to allow ourselves to deviate from the rule in this case.

Putting it all together

Finally we present a short example that illustrates how to use the classes `MessagingThread` and `TimestampedMessage` as well as how to pass a message from one thread to another. In this example we have two threads that work as a producer/consumer pair. The producer thread waits for the user to input a line of text on the keyboard and sends it on to the consumer as a timestamped message. The consumer awaits messages and when they arrive the consumer prints the contents of the message together with the timestamp showing when the message was created.

```
public class MyMessage extends TimestampedMessage {
  private String mess;
  public MyMessage(String m) { mess = m; }
  public String getMessage() { return mess; }
}

public class Consumer extends MessagingThread<MyMessage> {
  public Consumer() {
    super(10);
  }
```

```
    public void run() {
      try {
        while (true) {
          MyMessage m = receive();
          System.out.println(m.getMessage()+" "+m.getTimestamp());
        }
      } catch(InterruptedException e ) { System.err.println("Consumer failure: "+e); }
    }
  }

  public class Producer extends Thread {
    private Consumer cons;
    public Producer(Consumer c) { cons = c; }
    public void run() {
      Scanner scan = new Scanner(System.in);
      try {
        while (true) {
          cons.send(new MyMessage(scan.nextLine()));
        }
      } catch(InterruptedException e ) { System.err.println("Producer failure: "+e); }
    }
  }

  public class MainProgram {
    public static void main(String [] args) {
      Consumer c = new Consumer();
      Producer p = new Producer(c);
      c.start();
      p.start();
    }
  }
```

With the help of the explained concepts and tools it is now assumed that the student reader is equipped appropriately to design and implement multi-threaded Java-programs, that follow the principles taught in the course “EDAF85 - Realtidssystem”.