http://www.odi.ch/prog/design/newbies.php

**Java Anti-Patterns**

This page collects some bad code that may not look so obviously bad to beginners. Beginners often struggle with the language syntax. They also have little knowledge about the standard JDK class library and how to make the best use of it. In fact I have collected all examples from everyday junior code. I have modified the original code to give it example character and such that it highlights the problems. Many of these problems can easily be detected by [FindBugs](http://findbugs.sourceforge.net), which is available as a simple Eclipse Plug-in. I strongly recommend this tool to any beginner programmer. Also pros should run it from time to time on their codebase, and review its output carefully. It an easy to use tool and I always find some bugs when I use it. An even more complete suite is [SonarQube](http://www.sonarsource.com/).

Some of these may seem like micro-optimization, premature optimization without profiling or constant factor optimizations. But performance and memory wasted in thousands of these small places adds up quickly and will grind an application to a crawl. And when I say application, I mean a server-side application running on an application server. That's what I do for a living. On desktop GUI applications the situation may not be as bad. But then, what's the only relevant platform that runs client-side Java applications? Android. An embedded platform with very limited resources (memory!). Here even constant factor optimizations pay off quickly. Like iterating over arrays instead of lists.

If you are interested in how to pogram compiler friendly, look at the [JDK Performance Wiki](https://wiki.openjdk.java.net/display/HotSpot/PerformanceTechniques).

In the end a lot of your application's performance depends on the overall quality of your code. By the way you should never underestimate the importance of memory footprint. I can't stress that enough. I have seen too many applications with crazy garbage collection overhead and out of memory errors. Even though garbage collection is quite fast, most server-side code's scalability is dominated and limited primarily by its *memory use per request/transaction* and the *request/transaction duration*. Improving either of these by a constant factor will directly give you a higher throughput by that factor. If the factor is 10, it can mean supporting 100 or 1000 users, which can make all the difference to your customer.

Compare these scenarios (assume 100MB young generation):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Scenario** | **thread pool** | **tx duration** | **=> max. tx / s** | **mem / tx** | **=> garbage / min** | **GC / min** |
| base | 30 | 100 ms | **300** | 50 KB | **900 MB** | **9** |
| slower | 30 | 1000 ms | **30** | 50 KB | **90 MB** | **0.9** |
| more mem | 30 | 100 ms | **300** | 500 KB | **9 GB** | **90** |
| excess mem | 30 | 100 ms | **300** | 5 MB | **90 GB** | **900** |

In the *slower* scenario the transaction duration is 10 times longer. This immediately cuts the maximum number of transactions per second by the factor of 10 as well (limited thread-pool, limited CPU resources). In the *more mem* scenario each transaction uses 10 times as much memory. This directly bumps up the number of garbage collections to over one per second, which causes non-negligible overhead. Using much more memory like in scenario *excess mem* this would lead to 15 collections per second, leaving 66ms per collection which is clearly not enough. The system will thrash. Also 66ms is below the transaction duration of 100ms, so many running transactions will still hold onto memory, preventing it from collection, and causing a propagation of that memory to older generations. This means the older generations will start growing and will need a large (slow) collection sooner. The application in that scenario no longer performs. I think this clearly shows how bad excess memory consumption is, compared to just slow code. All your superfast code can't help you when you allocate too much memory.

* [String concatenation](http://www.odi.ch/prog/design/newbies.php#0)
* [Lost StringBuffer performance](http://www.odi.ch/prog/design/newbies.php#1)
* [Testing for string equality](http://www.odi.ch/prog/design/newbies.php#2)
* [Converting numbers to Strings](http://www.odi.ch/prog/design/newbies.php#3)
* [Not taking advantage of immutable objects](http://www.odi.ch/prog/design/newbies.php#4)
* [XML parsers are for sissies](http://www.odi.ch/prog/design/newbies.php#5)
* [Assembling XML with String operations](http://www.odi.ch/prog/design/newbies.php#6)
* [The XML encoding trap](http://www.odi.ch/prog/design/newbies.php#7)
* [char is not int](http://www.odi.ch/prog/design/newbies.php#8)
* [Platform dependent filenames](http://www.odi.ch/prog/design/newbies.php#9)
* [Undefined encoding](http://www.odi.ch/prog/design/newbies.php#10)
* [Unbuffered streams](http://www.odi.ch/prog/design/newbies.php#11)
* [Unbuffered writes to an OutputStreamWriter](http://www.odi.ch/prog/design/newbies.php#12)
* [Infinite heap](http://www.odi.ch/prog/design/newbies.php#13)
* [Infinite time](http://www.odi.ch/prog/design/newbies.php#14)
* [Assuming a cheap timer call](http://www.odi.ch/prog/design/newbies.php#15)
* [Catch all: I don't know the right runtime exception](http://www.odi.ch/prog/design/newbies.php#16)
* [Exceptions are annoying](http://www.odi.ch/prog/design/newbies.php#17)
* [Re-wrapping RuntimeException](http://www.odi.ch/prog/design/newbies.php#18)
* [Not properly propagating the exception](http://www.odi.ch/prog/design/newbies.php#19)
* [Catching to log](http://www.odi.ch/prog/design/newbies.php#20)
* [Incomplete exception handling](http://www.odi.ch/prog/design/newbies.php#21)
* [The exception that never happens](http://www.odi.ch/prog/design/newbies.php#22)
* [The transient trap](http://www.odi.ch/prog/design/newbies.php#23)
* [Overkill initialization](http://www.odi.ch/prog/design/newbies.php#24)
* [Log instances: static or not?](http://www.odi.ch/prog/design/newbies.php#25)
* [Chosing the wrong class loader](http://www.odi.ch/prog/design/newbies.php#26)
* [Poor use of reflection](http://www.odi.ch/prog/design/newbies.php#27)
* [Synchronization overkill](http://www.odi.ch/prog/design/newbies.php#28)
* [Wrong list type](http://www.odi.ch/prog/design/newbies.php#29)
* [The HashMap size trap](http://www.odi.ch/prog/design/newbies.php#30)
* [Hashtable, HashMap and HashSet are overrated](http://www.odi.ch/prog/design/newbies.php#31)
* [Lists are overrated](http://www.odi.ch/prog/design/newbies.php#32)
* [Object arrays are soooo flexible](http://www.odi.ch/prog/design/newbies.php#33)
* [Premature object decomposition](http://www.odi.ch/prog/design/newbies.php#34)
* [Modifying setters](http://www.odi.ch/prog/design/newbies.php#35)
* [Unnecessary Calendar](http://www.odi.ch/prog/design/newbies.php#36)
* [Relying on the default TimeZone](http://www.odi.ch/prog/design/newbies.php#37)
* [Time zone "conversion"](http://www.odi.ch/prog/design/newbies.php#38)
* [Using Calendar.getInstance()](http://www.odi.ch/prog/design/newbies.php#39)
* [Dangerous Calendar manipulation](http://www.odi.ch/prog/design/newbies.php#40)
* [Calling Date.setTime()](http://www.odi.ch/prog/design/newbies.php#41)
* [Assuming SimpleDateFormat was thread-safe](http://www.odi.ch/prog/design/newbies.php#42)
* [Having a global Configuration/Parameters/Constants class](http://www.odi.ch/prog/design/newbies.php#43)
* [Not noticing overflows](http://www.odi.ch/prog/design/newbies.php#44)
* [Using == with float or double](http://www.odi.ch/prog/design/newbies.php#45)
* [Storing money in floating point variables](http://www.odi.ch/prog/design/newbies.php#46)
* [Not freeing resources in a finally block](http://www.odi.ch/prog/design/newbies.php#47)
* [Abusing finalize()](http://www.odi.ch/prog/design/newbies.php#48)
* [Involuntarily resetting Thread.interrupted](http://www.odi.ch/prog/design/newbies.php#49)
* [Spawning thread from static initializers](http://www.odi.ch/prog/design/newbies.php#50)
* [Canceled timer tasks that keep state](http://www.odi.ch/prog/design/newbies.php#51)
* [Holding strong references to ClassLoaders and unflushable caches](http://www.odi.ch/prog/design/newbies.php#52)

**String concatenation**

String s = "";

for (Person p : persons) {

s += ", " + p.getName();

}

s = s.substring(2); //remove first comma

This is a real performance killer: O(persons.length�). The repeated concatenation of strings in a loop causes excess garbage and array copying. Moreover it is ugly that the resulting string has to be fixed for an extra comma.

StringBuilder sb = new StringBuilder(persons.size() \* 16); // well estimated buffer

for (Person p : persons) {

if (sb.length() > 0) sb.append(", ");

sb.append(p.getName);

}

**Lost StringBuffer performance**

StringBuffer sb = new StringBuffer();

sb.append("Name: ");

sb.append(name + '\n');

sb.append("!");

...

String s = sb.toString();

Despite good intentions the above code is not perfect. The most obvious mistake is the string concatenation in line 3. In line 4 appending a char would be faster than appending a String. An also major omission is the missing length initialization of the buffer which may incur unnecessary resizing (array copying). In JDK 1.5 and above a StringBuilder instead of StringBuffer should have been used: because it is only a local variable the implicit synchronization is overkill. Actually, using simple String concatenation compiles to almost perfect byte code: it's only missing the length initialization.

StringBuilder sb = new StringBuilder(100);

sb.append("Name: ");

sb.append(name);

sb.append("\n!");

String s = sb.toString();

String s = "Name: " + name + "\n!";

**Testing for string equality**

if (name.compareTo("John") == 0) ...

if (name == "John") ...

if (name.equals("John")) ...

if ("".equals(name)) ...

None of the above comparisons is wrong - but neither are they really good. The compareTo method is overkill and too verbose. The == operator tests for object identity which is probably not what you want. The equals method is the way to go, but reversing the constant and variable would give you extra safety if name is null.

if ("John".equals(name)) ...

if (name.length() == 0) ...

if (name.isEmpty()) ...

**Converting numbers to Strings**

"" + set.size()

new Integer(set.size()).toString()

The return type of the Set.size() method is int. A conversion to String is wanted. These two examples in fact do the conversion. But the first incurs the penalty of a concatenation operation (translates to (new StringBuilder()).append(i).toString())). And the second creates an intermediate Integer wrapper. The correct way of doing it is one of these

Integer.toString(set.size())

**Not taking advantage of immutable objects**

zero = new Integer(0);

return Boolean.valueOf("true");

Integer as well as Boolean are immutable. Thus it doesn't make sense to create several objects that represent the same value. Those classes have built-in caches for frequently used instances. In the case of Boolean there are even only two possible instances. The programmer can take advantage of this:

zero = Integer.valueOf(0);

return Boolean.TRUE;

**XML parsers are for sissies**

int start = xml.indexOf("<name>") + "<name>".length();

int end = xml.indexOf("</name>");

String name = xml.substring(start, end);

This naive XML parsing only works with the most simple XML documents. It will however fail if a) the name element is not unique in the document, b) the content of name is not only character data c) the text data of name contains escaped characters d) the text data is specified as a CDATA section e) the document uses XML namespaces. XML is way too complex for string operations. There is a reason why XML parsers like Xerces are a over one megabyte jar files! The equivalent with JDOM is:

SAXBuilder builder = new SAXBuilder(false);

Document doc = doc = builder.build(new StringReader(xml));

String name = doc.getRootElement().getChild("name").getText();

**Assembling XML with String operations**

String name = ...

String attribute = ...

String xml = "<root>"

+"<name att=\""+ attribute +"\">"+ name +"</name>"

+"</root>";

Many beginners are tempted to produce XML output like shown above, by using String operations (which they know so well and which are so easy). Indeed it is very simple and almost beautiful code. However it has one severe shortcoming: It fails to escape reserved characters. So if the variables name or attribute contain any of the reserved characters <, >, &, " or ' this code would produce invalid XML. Also as soon as the XML uses namespaces, String operations may quickly become nasty and hard to maintain. Now XML should be assembled in a DOM. The JDom library is quite nice for that.

Element root = new Element("root");

root.setAttribute("att", attribute);

root.setText(name);

Document doc = new Documet();

doc.setRootElement(root);

XmlOutputter out = new XmlOutputter(Format.getPrettyFormat());

String xml = out.outputString(root);

**The XML encoding trap**

String xml = FileUtils.readTextFile("my.xml");

It is a very bad idea to read an XML file and store it in a String. An XML specifies its encoding in the XML header. But when reading a file you have to know the encoding beforehand! Also storing an XML file in a String wastes memory. All XML parsers accept an InputStream as a parsing source and they figure out the encoding themselves correctly. So you can feed them an InputStream instead of storing the whole file in memory temporarily. The byte order (big-endian, little-endian) is another trap when a multi-byte encoding (such as UTF-8) is used. XML files may carry a byte order mark at the beginning that specifies the byte order. XML parsers handle them correctly.

**char is not int**

int i = in.read();

char c = (char) i;

The above code assumes that you can create a character from a number. It's true technically: a character's number is the 16 bit Unicode codepoint number. But it is semantic nonsense. In Java the character is a semantic entity of its own. The character's byte representation is completely decoupled from that. If we encounter a char we don't need to worry whether the character is stored in UTF-8, UTF-16, USC-4 or ISO-8859-1 internally. It simply doesn't matter. We can compare it to other characters and it will always behave as expected. This concept is not known in C for example. In C the char type is just a numeric type. It can contain anything, even invalid data that does not represent characters. In C you have to know exactly which character encoding a char array uses or you may do wrong things when sorting, printing, searching etc. Also C programs may wrongly assume that a char is one byte long and contains values 0-127 or 0-256, which is true for ASCII, but not for many other character encodings (known as "multi-byte" character encodings). Anyway, in Java use Reader/Writer or CharsetEncoder/CharsetDecoder instead to convert between characters and their byte representation (see following paragraph).

**Platform dependent filenames**

File tmp = new File("C:\\Temp\\1.tmp");

File exp = new File("export-2013-02-01T12:30.txt");

File f = new File(path +'/'+ filename);

Never hard code paths in a filesystem. Different platforms have different conventions, and you can never be sure that a hard coded path is actually available on a random system. Use API calls to create temporary files. Mind that different file systems have different restrictions on what makes a valid file name. Here the exp file contains a colon character, which is illegal on Windows file systems. When you construct absolute or relative paths in the filesystem, be careful of the platform dependent separator character.

File tmp = File.createTempFile("myapp","tmp");

File exp = new File("export-2013-02-01\_1230.txt");

File f = new File(path + File.separatorChar + filename);

// or even better

File dir = new File(path);

File f = new File(dir, filename);

**Undefined encoding**

Reader r = new FileReader(file);

Writer w = new FileWriter(file);

Reader r = new InputStreamReader(inputStream);

Writer w = new OutputStreamWriter(outputStream);

String s = new String(byteArray); // byteArray is a byte[]

byte[] a = string.getBytes();

Each line of the above converts between byte and char using the default platform encoding. The code behaves differently depending on the platform it runs on. This is harmful if the data flows from one platform to another. It is considered bad practice to rely on the default platform encoding at all. Conversions should always be performed with a defined encoding.

Reader r = new InputStreamReader(new FileInputStream(file), "ISO-8859-1");

Writer w = new OutputStreamWriter(new FileOutputStream(file), "ISO-8859-1");

Reader r = new InputStreamReader(inputStream, "UTF-8");

Writer w = new OutputStreamWriter(outputStream, "UTF-8");

String s = new String(byteArray, "ASCII");

byte[] a = string.getBytes("ASCII");

**Unbuffered streams**

InputStream in = new FileInputStream(file);

int b;

while ((b = in.read()) != -1) {

...

}

The above code reads a file byte by byte. Every read() call on the stream will cause a native (JNI) call to the native implementation of the filesystem. Depending on the implementation this may cause a syscall to the operating system. JNI calls are expensive and so are syscalls. The number of native calls can be reduced dramatically by wrapping the stream into a BufferedInputStream. Reading 1 MB of data from /dev/zero with the above code took about 1 second on my laptop. With the fixed code below it was down to 60 milliseconds! That's a 94% saving. This also applies for output streams of course. And it is true not only for the file system but also for sockets.

InputStream in = new BufferedInputStream(new FileInputStream(file));

**Unbuffered writes to an OutputStreamWriter**

Writer w = new OutputStreamWriter(os, "UTF-8");

while (...) {

w.write("something");

}

As [demonstrated](http://www.odi.ch/weblog/posting.php?posting=671) OutputStreamWriter uses memory for each call to its write() methods. This is very unfortunate and not the behaviour that one would expect! If you do many writes, you should wrap it in a BufferedWriter, which (also unexpectedly) seems to use no memory at all:

Writer w = new BufferedWriter(new OutputStreamWriter(os, "UTF-8"));

**Infinite heap**

byte[] pdf = toPdf(file);

Here a method creates a PDF file from some input and returns the binary PDF data as a byte array. This code assumes that the generated file is small enough to fit into the available heap memory. If this code can not make this 100% sure then it is vulnerable to an out of memory condition. Especially if this code is run server-side which usually means many parallel threads. Bulk data must never be handled with byte arrays. Streams should be used and the data should be spooled to disk or a database.

File pdf = toPdf(file);

A similar anti-pattern is to buffer streaming input from an "untrusted" (security term) source. Such as buffering data that arrives on a network socket. If the application doesn't know how much data will be arriving it must make sure that it keeps an eye on the size of the data. If the amount of buffered data exceeds sane limits an error condition (exception) should be signalled to the caller, rather than driving the application against the wall by letting it run into an out of memory condition.

**Infinite time**

Socket socket = ...

socket.connect(remote);

InputStream in = socket.getInputStream();

int i = in.read();

The above code has two blocking calls that use unspecified timeouts. Imagine if the timeout is infinite. That may cause the application to hang forever. Generally it is an extremely stupid idea to have infinite timeouts in the first place. Infinity is extremely long. Even by the time the Sun turns into a red giant (it explodes), it's still a looong way to Infinity. The average programmer dies at 72. There is simply **no** real-world situation, where we want to wait that long. Infinite timeout is just an absurd thing. Use an hour, day, week, month, 1 year, 10 years. But not Infinity. To connect to a remote machine I personally find 20 seconds plenty of timeout. A human is not even as patient and would cancel the operation before. While there is a nice override for the connect() method that takes a timeout parameter, there is no such thing for the read(). But you can modify a Socket's socket timeout before every blocking call. (Not just once! You can set different timeouts for different situations.) The socket will throw an exception on blocking calls after that timeout. Also frameworks that communicate over the network should provide an API to control these timeouts and use sensible default values. Infinity is not sensible - it's insane and drives you mad. Who came up with this absolutely useless infinity timeout anyway?

Socket socket = ...

socket.connect(remote, 20000); // fail after 20s

InputStream in = socket.getInputStream();

socket.setSoTimeout(15000);

int i = in.read();

Unfortunately the file system API (FileInputStream, FileChannel, FileDescriptor, File) provides no way to set timeouts on file operations. That's very unfortunate. Because these are the most common blocking calls in a Java application: writing to stdout/stderr and reading from stdin are file operations, and writing to log files is common. Operations on the standard input/output streams depend directly on other processes outside of our Java VM. If they decide to block forever, so will reads/writes to these streams in our application. Disk I/O is a limited resource for which all processes on a system compete. There is no guarantee that a simple read/write on a file is quick. It may incur unspecified wait time. Also today remote file systems are ubiquitous. Disks may be on a SAN/NAS, or file systems may be mounted over the network (NFS, AFS, CIFS/Samba). So a filesystem call may actually be a network call: too bad that we don't have the power of the network API here! So if the OS decides that the timeout for the write is 60 seconds you're stuck with it. It is a failure to assume that any disk/file operation is fast, or even remotely instantaneous. An application can do the user a favour by assuming that a file operation can takes seconds. So it's best avoided or done asynchronously (in background). Solutions to this problem are: adequate buffering and queueing/asynchronous processing.

**Assuming a cheap timer call**

for (...) {

long t = System.currentTimeMillis();

long t = System.nanoTime();

Date d = new Date();

Calendar c = new GregorianCalendar();

}

Creating a new Date or Calendar performs a syscall to obtain the current time. On Unix/Linux this is the syscall gettimeofday which is considered "extremely cheap". Well, extremely cheap only compared to other syscalls! In that it usually doesn't require a switch from userspace to kernelspace but is rather implemented as a read from a memory mapped page. Still calls to gettimeofday are expensive compared to normal code execution. The exact penalty of the call strongly depends on the architecture and even configuration (modern x86 systems have numerous timers that can be used by the OS: HPET, TSC, RTC, ACPI, clock chips etc.). On my Linux-2.6.37-rc7 system the timer calls also seem to be synchronised over the system. That means the total available bandwidth of ~800 calls per ms is shared by all threads/processes. Consequently my dual core running with 2 threads was able to make ~400 calls per ms per thread. (Thanks to J. Davies for that hint) And last but not least the resolution of this timer is not infinite. At best it is milliseconds, but it may well be rather something like 25 to 50 milliseconds with a large jitter. Modern Linux system can easily achieve the full ms resolution in System.currentTimeMillis. But that has not always been the case. System.nanoTime will certainly not have its full theoretical resolution: 1ns = 10-9s which corresponds to 1GHz. So on a CPU with 3GHz this would allow ~3 instructions to execute the call, which is obviously not enough. I measured a large jitter between 800ns and 1000000ns(1ms). Clearly calling gettimeofday every 100 nano seconds is wasteful.

Most of the time you don't need the current time as precicely. Caching it outside of the loop is trivial. This way you only access the timer once. You can still decide to clone the Date instance, if you really need different objects. Cloning is extremely cheap compared to a timer access (factor 50 on my system).

Date d = new Date();

for (E entity : entities) {

entity.doSomething();

entity.setUpdated((Date) d.clone());

}

Caching the time may not be an option if the loop runs for more than a couple of milliseconds. In that case you may setup a timer that periodically updates a timestamp variable with the current time (using interrupts). Set it to the exact granularity that you need. The coarser that granularity is, the better. On my system this loop is 200 times faster than creating a new Date each time.

private volatile long time;

Timer timer = new Timer(true);

try {

time = System.currentTimeMillis();

timer.scheduleAtFixedRate(new TimerTask() {

public void run() {

time = System.currentTimeMillis();

}

}, 0L, 10L); // granularity 10ms

for (E entity : entities) {

entity.doSomething();

entity.setUpdated(new Date(time));

}

} finally {

timer.cancel();

}

**Catch all: I don't know the right runtime exception**

Query q = ...

Person p;

try {

p = (Person) q.getSingleResult();

} catch(Exception e) {

p = null;

}

This is an example of a J2EE EJB3 query. The getSingleResult throws runtime exceptions when a) the result is not unique, b) there is no result c) when the query could not be executed due to database failure or so. The code above just catches any exception. A typical catch-all block. Using null as a result may be the right thing for case b) but not for case a) or c). In general one should not catch more exceptions than necessary. The correct exception handling is

Query q = ...

Person p;

try {

p = (Person) q.getSingleResult();

} catch(NoResultException e) {

p = null;

}

**Exceptions are annoying**

try {

doStuff();

} catch(Exception e) {

log.fatal("Could not do stuff");

}

doMoreStuff();

There are two problems with this tiny piece of code. First, if this is really a fatal condition then the method should abort and notify the caller of the fatal condition with an appropriate exception (so why is it caught in the first place?) Hardly ever can you just continue after a fatal condition. Second, this code is very hard to debug because the reason of the failure is lost. Exception objects carry detailed information about where the error occurred and what caused it. Individual subclasses may actually carry a lot of extra information that the caller can use to deal with the situation properly. It's a lot more than a simple error code (which is so popular in the C world. Just look at the Linux kernel. return -EINVAL everywhere...). If you catch highlevel exceptions then at least log the message and stack trace. You should not see exceptions as a necessary evil. They are a great tool for error handling.

try {

doStuff();

} catch(Exception e) {

throw new MyRuntimeException("Could not do stuff because: "+ e.getMessage(), e);

}

**Re-wrapping RuntimeException**

try {

doStuff();

} catch(Exception e) {

throw new RuntimeException(e);

}

Sometimes you really want to re-throw any checked exception as RuntimeException. The above piece of code doesn't take into account however, that RuntimeException extends Exception. The RuntimeException doesn't need to be catched here. Also the exception's message is not propagated properly. A bit better is to catch the RuntimeException separately and not wrap it. Even better is to catch all the checked exceptions individually (even if they are a lot).

try {

doStuff();

} catch(RuntimeException e) {

throw e;

} catch(Exception e) {

throw new RuntimeException(e.getMessage(), e);

}

try {

doStuff();

} catch(IOException e) {

throw new RuntimeException(e.getMessage(), e);

} catch(NamingException e) {

throw new RuntimeException(e.getMessage(), e);

}

**Not properly propagating the exception**

try {

} catch(ParseException e) {

throw new RuntimeException();

throw new RuntimeException(e.toString());

throw new RuntimeException(e.getMessage());

throw new RuntimeException(e);

}

This codes just wraps a parsing error into a runtime exception in different ways. None of them provides really good information to the caller. The first just loses all information. The second may do anything depending on what information toString() produces. The default toString() implementation lists the fully qualified exception name followed by the message. Nesting many exceptions will produce an unwieldy long and ugly string, unsuitable for a user. The third just preserves the message, which is better than nothing. The last preserves the cause, but sets the message of the runtime exception to toString() of its cause (see above). The most useful and readable version is to propagate only the cause message in the runtime exception and pass the original exception as the cause:

try {

} catch(ParseException e) {

throw new RuntimeException(e.getMessage(), e);

}

**Catching to log**

try {

...

} catch(ExceptionA e) {

log.error(e.getMessage(), e);

throw e;

} catch(ExceptionB e) {

log.error(e.getMessage(), e);

throw e;

}

This code only catches exception to write out a log statement and then rethrows the same exception. This is stupid. Let the caller decide if the message is important to log and remove the whole try/catch clause. Its only useful when you know that the caller doesn't log it. That's the case if the method is called by a framework which is not under your control.

**Incomplete exception handling**

try {

is = new FileInputStream(inFile);

os = new FileOutputStream(outFile);

} finally {

try {

is.close();

os.close();

} catch(IOException e) {

/\* we can't do anything \*/

}

}

If streams are not closed, the underlying operating system can't free native resources. This programmer wanted to be careful about closing both streams. So he put the close in a finally clause. But if is.close() throws an IOException then os.close is not even executed. Both close statements must be wrapped in their own try/catch clause. Moreover, if creating the input stream throws an exception (because the file was not found) then os is null and os.close() will throw a NullPointerException. To make this less verbose I have stripped some newlines.

try {

is = new FileInputStream(inFile);

os = new FileOutputStream(outFile);

} finally {

try { if (is != null) is.close(); } catch(IOException e) {/\* we can't do anything \*/}

try { if (os != null) os.close(); } catch(IOException e) {/\* we can't do anything \*/}

}

**The exception that never happens**

try {

... do risky stuff ...

} catch(SomeException e) {

// never happens

}

... do some more ...

Here the developer executes some code in a try/catch block. He doesn't want to rethrow the exception that one of the called methods declares to his annoyance. As the developer is clever he knows that in his particular situation the exception will never be thrown, so he just inserts an empty catch block. He even puts a nice comment in the empty catch block - but they are famous last words... The problem with this is: how can he be sure? What if the implementation of the called method changes? What if the exception is still thrown in some special case but he just didn't think of it? The code after the try/catch may do the wrong thing in that situation. The exception will go completely unnoticed. The code can be made much more reliable by throwing a runtime exception in the case. This works like an assertion and adheres to the "crash early" principle. The developer will notice if his assumption was wrong. The code after the try/catch will not be executed if the exception occurred against all honest hope and expectation. If the exception really never occurs - fine, nothing changed.

try {

... do risky stuff ...

} catch(SomeException e) {

// never happens hopefully

throw new IllegalStateException(e.getMessage(), e); // crash early, passing all information

}

... do some more ...

**The transient trap**

public class A implements Serializable {

private String someState;

private transient Log log = LogFactory.getLog(getClass());

public void f() {

log.debug("enter f");

...

}

}

Log objects are not serializable. The programmer knew this and correctly declared the log field as transient so it is not serialised. However the initialisation of this variables happens in the class' initialiser. Upon deserialization initializers and constructors are not executed! This leaves the deserialized object with a null log variable which subsequently causes a NullPointerException in f(). Rule of thumb: never use class initialization with transient variables. You can either solve this case here by using a static variable or by using a local variable:

public class A implements Serializable {

private String someState;

private static final Log log = LogFactory.getLog(A.class);

public void f() {

log.debug("enter f");

...

}

}

public class A implements Serializable {

private String someState;

public void f() {

Log log = LogFactory.getLog(getClass());

log.debug("enter f");

...

}

}

**Overkill initialization**

public class B {

private int count = 0;

private String name = null;

private boolean important = false;

}

This programmer used to code in C. So naturally he wants to make sure every variable is properly initialized. Here however it is not necessary. The Java language specification guarantees that member variables are initialized with certain values automatically: 0, null, false. By declaring them explicitly the programmer causes a class initializer to be executed before the constructor. This is unnecessary overkill and should be avoided.

public class B {

private int count;

private String name;

private boolean important;

}

**Log instances: static or not?**

This section was edited and before actually suggested not to store log instances in static variables. Turns out I was wrong. Mea culpa. I apologize.  
Store the darn log instance in a static final variable and be happy.

private static final Log log = LogFactory.getLog(MyClass.class);

Here is why:

* Automatically thread-safe. But only with the final keyword included!
* Usable from static and non-static code.
* No problems with serializable classes.
* Initialization cost only once: getLog() may not be as cheap as you might suppose.
* Nobody is going to unload the Log class loader anyway.

**Chosing the wrong class loader**

Class clazz = Class.forName(name);

Class clazz = getClass().getClassLoader().loadClass(name);

This code uses the class loader that loaded the current class. getClass() might return something unexpected, like a subclass, or a dynamic proxy. Something out of your control. This is hardly ever what you want when you dynamically load an additional class. Especially in managed environments like Application servers, Servlet engines or Java Webstart this is most certainly wrong. This code will behave very differently depending on the environment it is run in. Environments use the context class loader to provide applications with a class loader they should use to retrieve "their own" classes.

ClassLoader cl = Thread.currentThread().getContextClassLoader();

if (cl == null) cl = MyClass.class.getClassLoader(); // fallback

Class clazz = cl.loadClass(name);

**Poor use of reflection**

Class beanClass = ...

if (beanClass.newInstance() instanceof TestBean) ...

This programmer is struggling with the reflection API. He needs a way to check for inheritance but didn't find a way to do it. So he just creates a new instance and uses the instanceof operator he is used to. Creating an instance of a class you don't know is dangerous. You never know what this class does. It may be very expensive. Or the default constructor may not even exist. Then this if statement would throw an exception. The correct way of doing this check is to use the Class.isAssignableFrom(Class) method. Its semantics is upsidedown of instanceof.

Class beanClass = ...

if (TestBean.class.isAssignableFrom(beanClass)) ...

**Synchronization overkill**

Collection l = new Vector();

for (...) {

l.add(object);

}

Vector is a synchronized ArrayList. And Hashtable is a synchronized HashMap. Both classes should only be used if synchronization is explicitly required. If however those collections are used as local temporary variables the synchronization is complete overkill and degrades performance considerably. I measured a 25% penalty.

Collection l = new ArrayList();

for (...) {

l.add(object);

}

**Wrong list type**

Without sample code. Junior developers often have difficulties to chose the right list type. They usually choose quite randomly from Vector, ArrayList and LinkedList. But there are performance considerations to make! The implementations behave quite differently when adding, iterating or accessing object by index. I'll ignore Vector in this list because it behaves like an ArrayList, just slower. NB: n is the size of the list, not the number of operations! I refrain from using the O() notation here because it doesn't give a useful image of what's happening. The table lists the cost of list operations.

|  |  |  |
| --- | --- | --- |
|  | **ArrayList** | **LinkedList** |
| **add (append)** | const or ~log(n) if growing | const |
| **insert (middle)** | linear or ~n\*log(n) if growing | linear |
| **remove (middle)** | linear (always performs complete copy) | linear |
| **iterate** | linear | linear |
| **get by index** | const | linear |

The insert performance of the ArrayList depends on whether it has to grow during the insert or if the initial size is reasonably set. The growing occurs exponentially (by factor 2) so growing cost is logarithmic. The exponential growing however may use much more memory than you actually need. The sudden need to resize the list also makes the response time sluggish and will probably cause a major garbage collection if the list is large. Iterating over the lists is equally inexpensive. Indexed list element access however is very slow in linked lists of course.  
Memory considerations: LinkedList wraps every element into a wrapper object. ArrayList allocates a completely new array each time it needs to grow and performs an array copy on every remove(). All standard Collections can not reuse their Iterator objects, which may cause Iterator churn especially when recursively iterating large tree structures.  
Personally I almost never use LinkedList. It would really only make sense when you wanted to insert objects in the middle of a list. But without access to the wrapper object this doesn't scale and has linear cost because you must first traverse the list until you find the insert position. So what exactly is the point of the LinkedList class? I recommend using ArrayLists only.

**The HashMap size trap**

Map map = new HashMap(collection.size());

for (Object o : collection) {

map.put(o.key, o.value);

}

This developer had good intentions and wanted to make sure that the HashMap doesn't need to be resized. He thus set its initial size to the number of elements he was going to put into it. Unfortunately the HashMap implementation doesn't quite behave like this. It sets its internal threshold to threshold = (int)(capacity \* loadFactor). So it will resize after 75% of the collection have been inserted into the map. The above code will thus always cause extra garbage.

Map map = new HashMap(1 + (int) (collection.size() / 0.75));

**Hashtable, HashMap and HashSet are overrated**

These classes are extremely popular. Because they have great usability for the developer. Unfortunately they are also horribly inefficient. Hashtable and HashMap wrap every key/value pair into an Entry wrapper object. An Entry object is surprisingly large. Not only does it hold a reference to key and value, but also stores the hash code and a forward reference to the next Entry of the hash bucket. When you look at heap dumps with a memory analyzer you will be shocked by how much space is wasted by them in large applications like an application server. When you look at the source code of HashSet you will see that the developers were extremely lazy and just used a HashMap in the backend!  
Before using any of these classes, think again. IdentityHashMap can be a viable alternative. But be careful, it intentionally breaks the Map interface. It is much more memory efficient by implementing an open hashtable (no buckets), doesn't need an Entry wrapper and uses a simple Object[] as its backend. Instead of a HashSet a simple ArrayList may do similarly well (you can use contains(Object)) as long as it's small and lookups are rare.  
For Sets that contain only a handful of entries the whole hashing is overkill and the memory wasted for the HashMap backend plus the wrapper objects is just nuts. Just use an ArrayList or even an array.  
Actually it's a shame that there is no efficient Map and Set implementations in the standard JDK!

**Lists are overrated**

Also List implementations are very popular. But even lists are often not necessary. Simple arrays may do as well. I am not saying that you should not use Lists at all. They are great to work with. But know when to use arrays. The following are indicators that you should be using an array instead of a list:

* The list has a fixed size. Example: days of the week. A set of constants.
* The list is often (10'000 times) traversed.
* The list contains wrapper objects for numbers (there are no lists of primitive types).

Let me illustrate that in code:

List<Integer> codes = new ArrayList<Integer>();

codes.add(Integer.valueOf(10));

codes.add(Integer.valueOf(20));

codes.add(Integer.valueOf(30));

codes.add(Integer.valueOf(40));

versus

int[] codes = { 10, 20, 30, 40 };

// horribly slow and a memory waster if l has a few thousand elements (try it yourself!)

List<Mergeable> l = ...;

for (int i=0; i < l.size()-1; i++) {

Mergeable one = l.get(i);

Iterator<Mergeable> j = l.iterator(i+1); // memory allocation!

while (j.hasNext()) {

Mergeable other = l.next();

if (one.canMergeWith(other)) {

one.merge(other);

other.remove();

}

}

}

versus

// quite fast and no memory allocation

Mergeable[] l = ...;

for (int i=0; i < l.length-1; i++) {

Mergeable one = l[i];

for (int j=i+1; j < l.length; j++) {

Mergeable other = l[j];

if (one.canMergeWith(other)) {

one.merge(other);

l[j] = null;

}

}

}

You save an extra list object (wrapping an array), wrapper objects and possibly lots of iterator instances. Even Sun realized this. That's why [Collections.sort()](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collections.html#sort%28java.util.List%29) actually copies the list into an array and performs the sort on the array.

**Object arrays are soooo flexible**

/\*\*

\* @returns [1]: Location, [2]: Customer, [3]: Incident

\*/

Object[] getDetails(int id) {...

Even though documented, this kind of passing back values from a method is ugly and error prone. You should really declare a small class that holds the objects together. This is analoguos to a struct in C.

Details getDetails(int id) {...}

private class Details {

public Location location;

public Customer customer;

public Incident incident;

}

**Premature object decomposition**

public void notify(Person p) {

...

sendMail(p.getName(), p.getFirstName(), p.getEmail());

...

}

class PhoneBook {

String lookup(String employeeId) {

Employee emp = ...

return emp.getPhone();

}

}

In the first example it's painful to decompose an object just to pass its state on to a method. In the second example the use of this method is very limited. If overall design allows it pass the object itself.

public void notify(Person p) {

...

sendMail(p);

...

}

class EmployeeDirectory {

Employee lookup(String employeeId) {

Employee emp = ...

return emp;

}

}

**Modifying setters**

private String name;

public void setName(String name) {

this.name = name.trim();

}

public void String getName() {

return this.name;

}

This poor developer suffered from spaces at the beginning or end of a name entered by the user. He thought to be clever and just removed the spaces inside the setter method of a bean. But how odd is a bean that modifies its data instead of just holding it? Now the getter returns different data than was set by the setter! If this was done inside an EJB3 entity bean a simple read from the DB would actually modify the data: For every INSERT there would be an UPDATE statement. Let alone how hard it is to debug these side-effects! In general, a bean should not modify its data. It is a data container, not business logic. Do the trimming where it makes sense: in the controller where the input occurs or in the logic where the spaces are not wanted.

person.setName(textInput.getText().trim());

**Unnecessary Calendar**

Calendar cal = new GregorianCalender(TimeZone.getTimeZone("Europe/Zurich"));

cal.setTime(date);

cal.add(Calendar.HOUR\_OF\_DAY, 8);

date = cal.getTime();

A typical mistake by a developer who is confused about date, time, calendars and time zones. To add 8 hours to a Date there is no need for a Calendar. Neither is the time zone of any relevance. (Think about is if you don't understand this!) However if we wanted to add days (not hours) we would need a Calendar, because we don't know the length of a day for sure (on DST change days may have 23 or 25 hours).

date = new Date(date.getTime() + 8L \* 3600L \* 1000L); // add 8 hrs

Calendar cal = new GregorianCalender(TimeZone.getTimeZone("Europe/Zurich"));

SimpleDateFormat df = new SimpleDateFormat("dd.MM.yyyy HH:mm");

df.setCalendar(cal);

Here the Calendar object is completely unnecessary. The DateFormat object already contains a Calendar instance. Reuse that.

SimpleDateFormat df = new SimpleDateFormat("dd.MM.yyyy HH:mm");

df.setTimeZone(TimeZone.getTimeZone("Europe/Zurich"));

**Relying on the default TimeZone**

Calendar cal = new GregorianCalendar();

cal.setTime(date);

cal.set(Calendar.HOUR\_OF\_DAY, 0);

cal.set(Calendar.MINUTE, 0);

cal.set(Calendar.SECOND, 0);

Date startOfDay = cal.getTime();

The developer wanted to calculate the start of the day (0h00). First he obviously missed out the millisecond field of the Calendar. But the real big mistake is not setting the TimeZone of the Calendar object. The Calendar will thus use the default time zone. This may be fine in a Desktop application, but in server-side code this is hardly ever what you want: 0h00 in Shanghai is in a very different moment than in London. The developer needs to check which is the time zone that is relevant for this computation.

Calendar cal = new GregorianCalendar(user.getTimeZone());

cal.setTime(date);

cal.set(Calendar.HOUR\_OF\_DAY, 0);

cal.set(Calendar.MINUTE, 0);

cal.set(Calendar.SECOND, 0);

cal.set(Calendar.MILLISECOND, 0);

Date startOfDay = cal.getTime();

**Time zone "conversion"**

public static Date convertTz(Date date, TimeZone tz) {

Calendar cal = Calendar.getInstance();

cal.setTimeZone(TimeZone.getTimeZone("UTC"));

cal.setTime(date);

cal.setTimeZone(tz);

return cal.getTime();

}

If you think this method does something useful, please go and read the [article about time](http://www.odi.ch/prog/design/datetime.php). This developer had not read the article and was desperately trying to "fix" the time zone of his date. Actually the method does nothing. The returned Date will not have any different value than the input. Because a Date does not carry time zone information. It is always UTC. And the getTime / setTime methods of Calendar always convert between UTC and the actual time zone of the Calendar.

**Using Calendar.getInstance()**

Calendar c = Calendar.getInstance();

c.set(2009, Calendar.JANUARY, 15);

This code assumes a Gregorian calendar. But what if the returned Calendar subclass is a Buddhistic, Julian, Hebrew, Islamic, Iranian or Discordian calendar? In these the year 2009 has a very different meaning. And a month called January doesn't exist. Calendar.getInstance() uses the current default locale to select an appropriate implementation. It depends on the Java implementaton which implementations are available. The utility of Calendar.getInstance() is thus very limited, and its use should be avoided as it's result is not well defined.

Calendar c = new GregorianCalendar(timeZone);

c.set(2009, Calendar.JANUARY, 15);

**Dangerous Calendar manipulation**

GregorianCalender cal = new GregorianCalender(TimeZone.getTimeZone("Europe/Zurich"));

cal.set(Calendar.SECOND, 0);

cal.set(Calendar.MILLISECOND, 0);

if (cal.before(other)) doSomething();

cal.setTimeZone(TimeZone.getTimeZone("GMT"));

cal.set(Calendar.HOUR\_OF\_DAY, 23);

Date d = cal.getTime();

This code manipulates a Calendar object in ways that are bound to yield undefined results. Calendar objects have complex inner state: individual fields for day, hour, year etc., a millisecond since epoch value (like Date) and a time zone. Depending on what you change, some of these fields are invalidated and are only recomputed from other values when you call certain methods:

* set() invalidates the millisecond since epoch value and dependent fields (changing DATE obviously invalidates DAY\_OF\_WEEK)
* setTimeZone() invalidates all fields execpt the millisecond since epoch value
* get(), getTime(), getTimeInMillis(), add(), roll() recomputes the millisecond since epoch value from the fields
* get(), add() also recompute invalid fields from millisecond since epoch

Whenever you change fields with set(), then dependend fields do not get updated until you call get(), getTime(), getTimeInMillis(), add(), or roll(). The first paragraph of above code calls set() followed by before(). There is no guarantee (according to the API Doc) that before() will see the modified time value.

The second paragraph invalidates all fields and the millisecond since epoch value by calling setTimeZone() and set(), losing the calendar's data completely. See also [bug 4827490](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4827490)

Calendar objects should always be manipulated according to these simple rules:

* Initialize TimeZone (and Locale if you need) already in the constructor
* After calls to set() add a call to getTimeInMillis()
* After a call to setTimeZone() add a call to get()

GregorianCalender cal = new GregorianCalender(TimeZone.getTimeZone("Europe/Zurich"));

cal.set(Calendar.SECOND, 0);

cal.set(Calendar.MILLISECOND, 0);

cal.getTimeInMillis();

if (cal.before(other)) doSomething();

cal.setTimeZone(TimeZone.getTimeZone("GMT"));

cal.get(Calendar.DATE);

cal.set(Calendar.HOUR\_OF\_DAY, 23);

Date d = cal.getTime();

**Calling Date.setTime()**

account.changePassword(oldPass, newPass);

Date lastmod = account.getLastModified();

lastmod.setTime(System.currentTimeMillis());

The above code updates the last modified date of the account entity. The programmer wants to be conservative and avoids creating a new Date object. Instead she uses the the setTime method to modify the existing Date instance.

There is actually nothing wrong with that. But I just do not recommend this practice. Date objects are usually passed around carelessly. The same Date instance could be passed to numerous objects, which don't make a copy in their setters. Dates are often used like primitives. Thus if you modify a Date instance, other objects that use this instance might behave unexpectedly. Of course it is unclean design if an object exposes its intrinsic Date instance to the outside world, if you write code that strictly adheres to classical OO-principles (which I think is too inconvenient). General everyday Java practice however is to just copy Date references and not clone the object in setters. Thus every programmer should treat Date as immutable and should not modify existing instances. This should only be done for performance reasons in special situations. Even then the use of a simple long is probably equally good.

account.changePassword(oldPass, newPass);

account.setLastModified(new Date());

**Assuming SimpleDateFormat was thread-safe**

public class Constants {

public static final SimpleDateFormat date = new SimpleDateFormat("dd.MM.yyyy");

}

The above code is flawed in several ways. It's broken, because it shares a static instance of a SimpleDateFormat with possibly any number of threads. SimpleDateFormat is not thread-safe. If multiple threads concurrently use this object the results are undefined. You may observe strange output from format and parse or even exceptions. Unfortunately this mistake is very common!

Yes, sharing a SimpleDateFormat requires proper synchronization. Yes that comes at a price (cache flushes, lock contention, etc.). And yes, creating a SimpleDateFormat is not free either (pattern parsing, object allocation). But simply ignoring thread-safety is not a solution, but a sure way to break your code.

Of course this code also doesn't take the time zone into account. And then defining a class called Constants screams of yet another anti-pattern (see next section).

**Having a global Configuration/Parameters/Constants class**

public interface Constants {

String version = "1.0";

String dateFormat = "dd.MM.yyyy";

String configFile = ".apprc";

int maxNameLength = 32;

String someQuery = "SELECT \* FROM ...";

}

Often seen in large projects: one class or interface that contains all sorts of constants that are used throughout the application. Why is this bad? Because these constants are unrelated to each other. This class is the only thing that they have in common. And the reference to this class will pollute many again unrelated components of the application. You want to later extract a component and use it in a different application? Or share some classes between a server and a remote client? You may need to ship the constants class as well! This class has introduced a dependency between otherwise unrelated components. This inhibits reuse and loose coupling and gives way to chaos.

Instead put constants where they belong. In no case should constants be used across component boundaries. This is only allowed if the component is a library, on which an explicit dependency is wanted.

**Not noticing overflows**

public int getFileSize(File f) {

long l = f.length();

return (int) l;

}

This developer, for whatever reason, wrapped a call to determine the size of a file into a method that returns an int instead of a long. This code does not support files larger than 2 GB and just returns a wrong length in that case. Code that casts a value to a smaller size type must first check for a possible overflow and throw an exception.

public int getFileSize(File f) {

long l = f.length();

if (l > Integer.MAX\_VALUE) throw new IllegalStateException("int overflow");

return (int) l;

}

Another version of an overflow bug is the following. Note the missing parantheses in the first println statement.

long a = System.currentTimeMillis();

long b = a + 100;

System.out.println((int) b-a);

System.out.println((int) (b-a));

And last, a true gem that I uprooted during code review. Note how the programmer tried to be careful, but then failed so badly by assuming an int could ever become larger than its maximum value.

int a = l.size();

a = a + 100;

if (a > Integer.MAX\_VALUE)

throw new ArithmeticException("int overflow");

**Using == with float or double**

for (float f = 10f; f!=0; f-=0.1) {

System.out.println(f);

}

The above code doesn't behave as expected. It causes an endless loop. Because 0.1 is an infinite binary decimal, f will never be exactly 0. Generally you should never compare float or double values with the equality operator ==. Always use less than or greater than. Java compilers should be changed to issue a warning in that case. Or even make == an illegal operation for floating point types in the Java Language Spec. It makes really no sense to have this feature.

for (float f = 10f; f>0; f-=0.1) {

System.out.println(f);

}

**Storing money in floating point variables**

float total = 0.0f;

for (OrderLine line : lines) {

total += line.price \* line.count;

}

double a = 1.14 \* 75; // 85.5 represented as 85.4999...

System.out.println(Math.round(a)); // surprising output: 85

System.out.println(10.0/3); // surprising output: 3.333333333333333**5** (precision lost twice during division and on conversion to decimal)

BigDecimal d = new BigDecimal(1.14); // precision has already been lost

I have seen many developers coding such a loop. Including myself in my early days. When this code sums 100 order lines with every line having one 0.30$ item, the resulting total is calculated to exactly 29.999971. The developer notices the strange behaviour and changes the float to the more precise double, only to get the result 30.000001192092896. The somewhat surprising result is of course due to the difference in representation of numbers by humans (in decimal format) and computers (in binary format). It always occurs in its most annyoing form when you add fractional amounts of money or calculate the VAT.

Binary representation of floating point numbers was invented for inherently *inexact* values like measurements. Perfect for engineering! But unusable when you want exact math. Like banks. Or when counting.

There are business cases where you can not afford to lose precision. You lose precision when converting between decimal and binary and when rounding happens in not a well-defined mannor or at indeterminate points. To avoid losing precision you must use fixed point or integer arithmetics. This does not only apply to monetary values, but it is a frequent source of annoyance in business applications and therefore makes a good example. In the second example an unsuspecting user of the program would simply say the computer's calculator is broken. That's of course very embarassing for the programmer.

Consequently an amount of money should *never ever* be stored in a floating point data type (float, double). Please note that it is not just any calculation that is inexact. Even a simple multiplication with an integer can already yield an inexact result. It is the mere fact of *storing* a value in a binary representation (float, double) that may already cause rounding! *You simply can not store 0.3 as an exact value in float or double*. Because float and double are [binary IEEE754](http://en.wikipedia.org/wiki/Binary64) types. If you see a float or double in your financial code base, the code will most likely yield inexact results. Instead either a string or fixed point representation should be chosen. A text representation must be in a well-defined format and is *not* to be confused with user input/output in a locale specific format. Both representations must define the precision (number of digits before and after the decimal point) that is stored.

For calculations the class [BigDecimal](http://java.sun.com/j2se/1.5.0/docs/api/java/math/BigDecimal.html) provides an excellent facility. The class can be used such that it throws runtime exceptions if precision is unexpectedly lost in an operation. This is very helpful to uproot subtle numerical bugs and enables the developer to correct the calculation.

BigDecimal total = BigDecimal.ZERO;

for (OrderLine line : lines) {

BigDecimal price = new BigDecimal(line.price);

BigDecimal count = new BigDecimal(line.count);

total = total.add(price.multiply(count)); // BigDecimal is immutable!

}

total = total.setScale(2, RoundingMode.HALF\_UP);

BigDecimal a = (new BigDecimal("1.14")).multiply(new BigDecimal(75)); // 85.5 exact

a = a.setScale(0, RoundingMode.HALF\_UP); // 86

System.out.println(a); // correct output: 86

BigDecimal a = new BigDecimal("1.14");

**Not freeing resources in a finally block**

public void save(File f) throws IOException {

OutputStream out = new BufferedOutputStream(new FileOutputStream(f));

out.write(...);

out.close();

}

public void load(File f) throws IOException {

InputStream in = new BufferedInputStream(new FileInputStream(f));

in.read(...);

in.close();

}

The above code opens an output stream to a file, allocating a file handle in the operating system. File handles are a rare resource and need to be properly freed, by calling close on the FileOutputStream (same for FileInputStream of course). To ensure that even in the case of an exception (the filesystem may become full during the write), closing must happen in a finally block. Here the stream is also wrapped into a buffering stream. That means not all data will have been written to disk by the time we arrive at the close() call. The close call itself will flush the pending data in the buffer to disk and may thus itself fail with an IOException. If that close fails the file on disk is incomplete (truncated) and thus probably corrupt. The method should therefore propagate the IOException in that case. In the case of a FileInputStream we can safely ignore the potential IOException from a close() call. We have read all data that we need, and there is nothing useful that we can do if the underlying close() failed anyway. It's not even worth logging it.

In a perfect world BufferedOutputStream.close() would be implemented correctly. But sadly it has a [bug](http://bugs.sun.com/view_bug.do?bug_id=6335274) that's not going to be fixed: it loses any IOException from the implicit flush and truncates your file silently. So here we give the proper workaround with an explicit flush before close.

To be exact the corrected code below can leak in one small corner case: when the file stream was allocated but then allocating the buffered stream fails mysteriously (with out of memory for instance). As a pragmatic person I think in such a pathological case we can safely rely on the garbage collector to clean up the mess. It's not worth the hassle to deal with it.

// code for your cookbook

public void save() throws IOException {

File f = ...

OutputStream out = new BufferedOutputStream(new FileOutputStream(f));

try {

out.write(...);

out.flush(); // don't lose exception by implicit flush on close

} finally {

out.close();

}

}

public void load(File f) throws IOException {

InputStream in = new BufferedInputStream(new FileInputStream(f));

try {

in.read(...);

} finally {

try { in.close(); } catch (IOException e) { }

}

}

Let me give you also the cook book recipe for another ubiquitous pattern: database access. Again this is the pragmatic approach. Yes, rs.close() could fail with mysterious Errors, except they only occur in your university lecture on Quantum Mechanics and not in The Real World (tm). And only perverts would write the try/finally cascade that no Error neutrino can escape. Forgive my sarcasm. Here once and for all this is how to deal with SQL objects:

Car getCar(DataSource ds, String plate) throws SQLException {

Car car = null;

Connection c = null;

PreparedStatement s = null;

ResultSet rs = null;

try {

c = ds.getConnection();

s = c.prepareStatement("select make, color from cars where plate=?");

s.setString(1, plate);

rs = s.executeQuery();

if (rs.next()) {

car = new Car();

car.make = rs.getString(1);

car.color = rs.getString(2);

}

} finally {

if (rs != null) try { rs.close(); } catch (SQLException e) { }

if (s != null) try { s.close(); } catch (SQLException e) { }

if (c != null) try { c.close(); } catch (SQLException e) { }

}

return car;

}

With that said, don't miss the next paragraph.

**Abusing finalize()**

public class FileBackedCache {

private File backingStore;

...

protected void finalize() throws IOException {

if (backingStore != null) {

backingStore.close();

backingStore = null;

}

}

}

This class uses the finalize method to release a file handle. The problem is that you can don't know when the method is called. The method is called by the garbage collector. If you are running out of file handles you want this method to be called rather sooner than later. But the GC will probably only invoke the method when you are about to run out of heap, which is a very different situation. It may take anything from milliseconds to days until GC and finalization runs. The garbage collector manages memory only. It does that very well. But it must not be abused to manage any other resources apart from that. **The GC is not a generic resource management mechanism!** I find Sun's API Doc of the finalize method very misleading in that respect. It actually suggest to use this method to close I/O resources - complete bullshit if you ask me. Again: I/O has *nothing* to do with memory!

Better code provides a public close method, which must be called by a well-defined lifecycle management, like JBoss MBeans or so.

public class FileBackedCache {

private File backingStore;

...

public void close() throws IOException {

if (backingStore != null) {

backingStore.close();

backingStore = null;

}

}

}

JDK 1.7 (Java 7) will introduce the [AutoClosable](http://download.java.net/jdk7/docs/api/java/lang/AutoCloseable.html) interface. It enables an automatic call to a close method, when the variable (not the object) goes out of scope of a try-with-resource block. It is very different from a finalizer. Its time of execution is well-defined at compile time.

try (Writer w = new FileWriter(f)) { // implements Closable

w.write("abc");

// w goes out of scope here: w.close() is called automatically in ANY case

} catch (IOException e) {

throw new RuntimeException(e.getMessage(), e);

}

**Involuntarily resetting Thread.interrupted**

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

// ok

}

------------------------or --------------------

while (true) {

if (Thread.interrupted()) break;

}

The above code resets the interrupted flag of the Thread. Subsequent readers will not know that the Thread has been interrupted. If you need to pass on the information about the interrupt, rewrite the code like so.

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

------------------------or --------------------

while (true) {

if (Thread.currentThread().isInterrupted()) break;

}

**Spawning thread from static initializers**

class Cache {

private static final Timer evictor = new Timer();

}

java.util.Timer spwans a new thread in its constructor. Therefore the above code spawns a new thread in its static initializer. The new Thread will inherit some properties from its parent: context classloader, inheritable ThreadLocals, and some security properties (access rights). It is therefore rarely desireable to have those property set in an uncontrolled way. This may for instance prevent GC of a class loader.

The static initializer is executed by the thread that first loads the class (in any given ClassLoader), which may be a totally random thread from a thread pool of a webserver for example. If you want to control these thread properties you will have to start threads in a static method, and take control of who is calling that method.

class Cache {

private static Timer evictor;

public static setupEvictor() {

evictor = new Timer();

}

}

**Canceled timer tasks that keep state**

final MyClass callback = this;

TimerTask task = new TimerTask() {

public void run() {

callback.timeout();

}

};

timer.schedule(task, 300000L);

try {

doSomething();

} finally {

task.cancel();

}

The above code uses a timer to enforce a timeout on doSomething(). The TimerTask contains an (implicit) instance reference to the outer class. Thus as long as the TimerTask exists the instance of MyClass may not be GC'ed. Unfortunately the Timer may keep cancelled TimerTasks around until their scheduled timeout has expired! That would leave the program 5 minutes with a dangling reference to the MyClass instance during which it can not get collected! It's a temorary memory leak. A better TimerTask would override the cancel() method and null the reference there. It requires slightly more code.

TimerTask task = new Job(this);

timer.schedule(task, 300000L);

try {

doSomething();

} finally {

task.cancel();

}

static class Job extends TimerTask {

private volatile MyClass callback;

public Job(MyClass callback) {

this.callback = callback;

}

public boolean cancel() {

callback = null;

return super.cancel();

}

public void run() {

MyClass cb = callback;

if (cb == null) return;

cb.timeout();

}

}

**Holding strong references to ClassLoaders and unflushable caches**

In a dynamic system like an application server or OSGI, you should take good care not to prevent ClassLoaders from garbage collection. As you undeploy and redeploy individual applications in an application server you create new class loaders for them. The old ones are unused and should be collected. Java isn't going to let that happen if there is a single dangling reference from container code into your application code.

As various libraries are used throughout an enterprise application, that directly means that libraries should do their very best not to hold involuntary strong references to objects (and thus their class loaders).

This is not easy. Classes like java.beans.Introspector from the JDK or org.apache.commons.beanutils.PropertyUtils from Apache BeanUtils or org.springframework.beans.CachedIntrospectionResults from Spring implement caches to speed up their inner workings. They keep strong references to classes you pass them for analysis. Fortunately they provide methods to flush their caches. But finding all classes that may have internal caches and flushing them at the right time is a near to impossible job for the developer.

If you happen to use org.apache.commons.el.BeanInfoManager from Apache Commons EL you probably have a leak. This ancient class keeps a cache of strong references that only ever grows until out of memory. And it has no flush method. Even Tomcat had to implement a [workaround](http://issues.apache.org/bugzilla/attachment.cgi?id=18538&action=diff) involving reflection to clean it.

It would be much better if these libraries just used soft or weak references in the first place. A quick reminder:

Soft and weak references basically differ in the point in time when they are nulled.

* WeakReference: nulled more or less at the same time when the last strong reference to the object goes away. Typical for classloader references (of what use is a classloader if none of its classes are loaded). But be careful if you use this *within* a ClassLoader implementation.
* SoftReference: the reference is kept even if the last strong reference to the object goes away as long as memory allows. Typical for caches.

Only if the library just caches objects from its own packages (with no external references), it may be fine not to use these special references and just use normal references.

Using soft or weak references also helps the runtime behaviour of your application: if memory gets tight, the last thing you want to spend memory on is caches. So the garbage collector will reclaim the memory used by caches if necessary. A bad example here is JBoss' SQL statement cache: it's compeletely static and can use a lot of memory, even when that is tight. Another bad example is JBoss' authentication cache.

Also every static cache must always provide a simple way to flush its contents. It's the nature of a (clean)cache (as opposed to e.g a write cache) that its contents are not valuable and can be safely discarded at any time. The limits of the cache are another trap. Caches should never grow large, and never cache objects for too long. A really bad example here is the default settings for the JDK DNS cache (it completely ignores DNS record lifetimes, and stores negative lookups forever in an unbounded list). Your API documentation should state if and when caching happens. This also helps the user to estimate runtime performance.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Date and time in Java

During the course of my career I have met (too) many developers who are not familiar enough with issues revolving around time and date. I am pointing all developers with knowledge deficiency in that area to this page. This absence of knowledge can manifest in different statements:

* You want a time zone in the date format? No problem, I'll just always append +01:00 to the string.
* We don't need time zone information because we don't have overseas customers.
* We don't need time zone information because Switzerland doesn't stretch across time zones.
* No, there can't be a time zone issue: Webserver and DB are in the same room.
* Why is there a 'Z' at the end of this XML datetime string?

As with so many things everything said here is only true to some precision. The precision here ends with leap seconds. I am not taking these into account as in normal IT systems they are not relevant and our clocks are not as accurate. If you work in astronomy, or rocket science this may not be precise enough for you.

### What is time?

Time is a physical property. For Physicists it is a dimension tightly coupled with space (spacetime). Time as a physical property (or dimension) passes uniformly. As far as we know. And to any precision that is relevant to us. Even if it did not we were probably unable to tell because we lack something to compare it to. We simply can't define how fast time passes - we live inside of it. Just like a piece of driftwood in a river is unable to change its speed relative to the current we can not change our speed within time. Because to measure speed you compare the difference of what you measure to a difference of time. That sort of measurement is absurd for time itself of course. So we can safely assume that physical time passes uniformly. For physical time there is no notion of "date" or "clock" or "second". These are notions defined by humans.

### Measuring time

We don't live inside a particle accelerator. We live on earth. We know day and night. We can see time passing with every day, month, year. These are marked by periodic events: sunrise, full moon, the longest day. That's the way we have been measuring time forever: we use natural periodic events. Even the most scientific definition of today's "second" bases on that: The duration of 9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the Caesium-133 atom (Wikipedia). Here the periodic events are the waves of (a very precise) electromagnetic radiation. Note that, lacking any other option, our definition of the second is completely arbitrary.

We live on Earth, which spins around its axis. This is an accelerated motion by the terms of General Relativity, which means that we measure (slightly) shorter seconds than someone with a different acceleration e.g. in a space ship. But as stated above our definition of the second is arbitrary anyway. So as long as you don't work for NASA and as long as SETI hasn't found anything out there, don't worry.

### Measuring duration

So we employ the definition of the duration of a second to measure the passing of time. We can notably use seconds (and its fractions) to measure durations: how long it takes us to go shopping, how long it takes to boil an egg. A duration is the answer to the question "how long". You measure durations with a stop watch. Specifically a duration does not answer the question "when".

### UTC

If we want an answer to the question "when" other than "in 27 seconds from now" we must have a means to reference any point in time. We can do this simply by laying out a time scale much like a ruler. A ruler has a zero mark and a positive (and maybe negative) part on its scale. We can do the same in time. We define some arbitrary point in time as 0. From that we can make a reference to any point in the future or in the past by telling how many seconds it is away from point 0. That is what the Coordinated Universal Time (UTC) does. It gives every second on that scale a name: "2007-12-31 23:59:59.000 UTC" for example. UTC always increases. It never goes backwards. Some minutes however may not have exactly 60 seconds. Some may have 59 (that has not happened so far), some may have 61 (happens a couple of times a year). They are called leap seconds and are used for corrections.

### Computer clocks

Computers contain clocks (also referred to as timers). Several of them usually. With different precisions and capabilities. The operating system gives us access to the "system clock", which in practice uses one of these timers. On a "good" operation system this clock always advances, and never appears to go backwards. That means for consecutive readings a and b we can rely on: a <= b, but not necessarily a < b. Because these timers have limited precision, consecutive readings may return the same value, even though the numerical precision would allow to represent a difference. On modern Linux systems System.currentTime() normally achieves millisecond precision, but System.nanoTime() does not achieve nanosecond precision. Different restrictions are valid for Object.wait(). This method uses interrupts to get a timeout notification. The precision of this timing may vary widely. Please note that these clocks and interrupts may exhibit a lot of jitter: the clock appears to jump wildly even when read in precise intervals.

### NTP

Computer clocks tend to be very bad clocks compared to your watch. Their speed varies constantly. They need to be adjusted all the time to avoid going either too fast or too slow. NTP does just that. With some millisecond precision. Don't be scared that NTP would set your clock backwards and you could observe a > b. The operating system will only advance the clock or delay it (slew), but never reset it. That is, only to really adjust a wrong clock for more than 2 seconds.

Time synchronise your servers. Servers whose time is not in sync with the global time cause problems. The UNIX [NTP](http://www.ntp.org/) daemon can keep time in sync with millisecond precision. This minimal /etc/ntp.conf is enough to keep your server synced (needs UDP port 123 in your firewall open):

server pool.ntp.org iburst

driftfile /var/lib/ntp/ntp.drift

restrict default nomodify nopeer

restrict 127.0.0.1

### Local time

When you look at your watch, most of you do not read UTC. You read local time. Also known as wall clock time. It is a representation of UTC in a certain calendar and time zone. More on this later.

### Ambiguous Day

This may seem obvious, but it is not. A day is defined as 86400 seconds (24 hours). This is the time the earth takes for a single rotation. Exactly it is the time until the sun appears in the same place in the sky again. This is different from the time it takes to see distant stars in the same place which is called a sideral day (the earth advances on its orbit around the sun). As you will see in the paragraph about Daylight Saving Time a day in your local time is not always 24 hours. In many countries there is one day a year with 23 and one with 25 hours. Applications should never assume that a local day has 24 hours. The notion of a "day" is not well-defined without a calendar!

This ambiguity easily manifests in Java Timers. The method java.util.Timer.schedule(TimerTask task, Date firstTime, long period) is unsuitable for executing daily tasks. If you pass 86400000 milliseconds (24 hours) as the period, the timer will fire every 24 hours, but after a DST change the timer appears to be firing one hour off schedule. While the timer sticks exactly to its 24 hour interval, people usually expect timers to fire at the same local time every day.

### Representing points in time by UNIX time

In IT systems we sometimes need to refer to precise points in time. Examples are:

* When was a file last modified
* Start of an appointment
* When an email was sent

These points in time have the characteristics of an event. If two events have the same time stamp they occurred in the very same moment. It doesn't matter if one occurs in Shanghai and the other in London: they still occur in the very same moment. Independent of any time zone. We usually use UNIX time to represent points in time.

UNIX time stamps are the number of seconds that have elapsed since midnight 1.1.1970 UTC. Uh... here appears a "date". Never mind, we'll get to that in a minute. Java's java.util.Date class effectively encapsulates a UNIX time stamp. It represents a point in time by a millisecond counter in a 64 bit long variable. While 32 bit representation on UNIX will overflow in 2038 the 64 bit signed long will last for roughly another 18 billion years. Please note that none of the deprecated methods and constructors of the Date class should ever be used. They are from a time when the Sun people were confused about time themselves. [javax.management.timer.Timer](http://java.sun.com/j2se/1.5.0/docs/api/javax/management/timer/Timer.html#field_summary) has some convenient constants for the most used millisecond values.

// these are equivalent

long now = System.currentTimeMillis();

Date now = new Date();

// long and Date are mutually exchangeable

// Date is actually just a long wrapper.

long ts = (new Date()).getTime();

Date d = new Date(1095379201L);

// simple arithmetics. Be careful of int overflows!

Date now = new Date();

Date in8Hours = new Date(now.getTime() + 8L \* Timer.ONE\_HOUR);

~~Date in7Days = new Date(now.getTime() + 7L \* Timer.ONE\_DAY);~~ // assumes 24h day!

// this is overkill

~~Date now = new Date(System.currentTimeMillis());~~

### Calendars

Referring to a point in time by its UNIX time stamp may be quite exact. But it is far from usable. If I tell you "let's meet at 1,095,379,201" it'll probably take you a while until you noticed that this point in time already passed several years ago. That's why we use calendars. Not only in the western world today we use the Gregorian Calendar. It defines twelve months a year of different length and inserts a leap day every four years (rule is a little more complex) at the end of February. It is a very exact calendar which means that in some hundred years time it will be snowing in July in Switzerland not because of an aberrant calendar but much more likely because of a change in climate.

### Calendar and time zone

Calendar define dates. Without a calendar "31 December 2001" has no meaning. A calendar is tightly coupled to the notion of a "day". Our notion of "day" is also strongly coupled to sunrise and sunset. Sunrise and sunset depend on where on earth you are, don't they. Hence, it is apparent that a Calendar is only well-defined together with a geographical position. This is especially visible on New Year's eve when around the globe people launch fireworks to celebrate the beginning of a new year. The toasting and launching of fireworks happens roughly 24 times (some time zones are not much inhabited, there are "odd" time zones) during a whole day!

Each one of these partying people of course claims the hour 24:00 for them - be they in Australia or Western Europe. It is also apparent that at any given moment there is not just one date around the globe but two. While people in Europe are already drunk and celebrating January 1st, people in New York are just starting to cook dinner on December 31st in the same moment. The two date "halves" of the Earth are separated by the time zone where midnight is and the international date boundary in the Pacific.

This leads to the shocking fact that "31 December 2001" is not well defined. Because there are two dates around the globe at any given moment this date can refer to anything within 48 hours. Not even "31 December 2001 24:00:00" is well defined at all without specifying a time zone. It can refer to any point in time within 24 hours.

All of the following expressions are useless without a time zone!

* 1.1.2007
* 14:30:00
* 1.1.2007 14:30:00
* 29.10.2006 02:30 Swiss local time (see Daylight Saving Time paragraph)
* Monday
* 2 days (see Ambiguous Day paragraph)

### Time zones

You have certainly heard of time zones. But do you really know exactly what they are? Think of them as a unit of measure for the thing called "time". Comparable with the unit of measure for "length". A stick of 1 meter length is 1000 millimetres, or 3.28 feet. The length of the stick doesn't change whether I say it's 1000 millimetres long or 3.28 feet. Same goes for time. The point in time doesn't change whether I express it in one time zone or another.

The mother of all time zones is [Greenwich](http://maps.google.ch/maps?f=q&hl=en&geocode=&q=greenwich&ie=UTF8&z=12&iwloc=addr) Mean Time (GMT). This is the time zone used by UTC. Offsets of all other time zones are specified in offsets from GMT. Zones to the east have positive offsets, zones to the west have negative ones. Again this nicely compares to length: the "meter" is the mother of all SI length units of measure (the original prototype is a [platinum bar](http://en.wikipedia.org/wiki/Metre#Prototype_metre_bar) kept in France). Other units are expressed as multiples of the meter: 1 millimeter = 1/1000 meter, 1 kilometer = 1000 meters.

Converting between time zones is easy: 18:00 GMT = 18:00+00:00 = 19:00+01:00 = 17:00-01:00

The suffix is the offset from GMT that has been added to the GMT time.

The Java java.util.TimeZone can represent two things depending on how it is used (and this may surprise and confuse most people):

* A time zone
* A time zone database of a location

Time zones are not assigned to locations by astronomers but by politicians. That's why that time is called legal time. These assignments [change](http://news.gmane.org/gmane.comp.time.tz/). That's why TimeZone keeps a database of historical time zone assignments of many cities. Also (and this may again surprise many people) many countries in the world change their time zone twice a year: daylight savings time is nothing else but a time zone. More on DST below. NB: Also England uses DST. So their time zone is not always GMT!

Most of the time applications use a location database like TimeZone.getTimeZone("Europe/Zurich"). This frees the application from most DST issues. See below. Of course an application can also use a simple exact time zone if required: TimeZone.getTimeZone("GMT+04:30")

Using the correct time zone for java.util.Calendar and java.text.DateFormat is crucial to make your application time zone safe. If you don't specify the time zone, Java will use the system default time zone. In a server environment that is hardly the one that's right.

Localizing an application doesn't just mean to support different Locales, but also different TimeZones. So classes in your codebase that are Locale dependent, are probably also TimeZone dependent.

It should be noted that TimeZone.getTimeZone() does not throw an exception when passed an unknown time zone name. Instead it returns GMT. If the name of the time zone comes from an untrusted source, such as a GUI, you need to validate it. Otherwise your application code will behave unexpectedly.

Read on for a lot more [detailed information](http://cs.ucla.edu/%7Eeggert/tz/tz-link.htm).

### Representing a point in time in a Calendar

As we have seen above, in IT systems, points in time do not have to be represented in a calendar. It is enough to use a UNIX time stamp. Representing a point in time in different calendars and time zones has no influence on the point in time that is being referred to (the length of the stick doesn't change). The only time a representation in a calendar is desirable, is for user interfaces and calendar arithmetics (finding the begin/end of a day, adding days, weeks, etc.). When a human needs to read a point in time she expects it in a human readable form in her local time zone.

In Java we use the java.text.DateFormat and it's most used implementation [java.text.SimpleDateFormat](http://java.sun.com/j2se/1.5.0/docs/api/java/text/DateFormat.html) to convert Date instances to and from user readable time representation. Important here is not to forget to specify the time zone. Otherwise the system default time zone will be used which is only meaningful for Desktop applications. Also note that SimpleDateFormat instances are not thread-safe. You should thus avoid static instances.

// not thread-safe

public ~~static~~ SimpleDateFormat dfm = new SimpleDateFormat("yyyy-MM-dd");

// This is how to initialise a well-known Date (often used in unit test fixtures)

DateFormat dfm = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

dfm.setTimeZone(TimeZone.getTimeZone("Europe/Zurich"));

Date a = dfm.parse("2007-02-26 20:15:00");

Date b = dfm.parse("2007-02-27 08:00:00");

// Don't use the deprecated constructors

~~Date badhabit = new Date(106, 1, 26, 20, 15, 00);~~

// output is likewise simple

System.out.println("Result: "+ dfm.format(a));

// if you don't initialise the DateFormat with a TimeZone, include it in the format!

DateFormat dfm = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss Z");

Date a = dfm.parse("2007-02-26 20:15:00 +0200");

Be careful with the symbols of SimpleDateFormat. "hh" corresponds to Calendar.HOUR which is the hour on the 12-hour clock. It is ambiguous without AM/PM. Furthermore the Calendar class doesn't throw an exception when you do calendar.set(Calendar.HOUR, 20), but it will do something completely wrong! Use "HH" and Calendar.HOUR\_OF\_DAY for the 24-hour clock.

Java also offers the java.util.Calendar and [java.util.GregorianCalendar](http://java.sun.com/j2se/1.5.0/docs/api/java/util/GregorianCalendar.html) pair. It allows to convert to and from Date with the setTime and getTime methods. It also allows modification of individual fields. This is the class to use when one must determine the start and end of a day. Also here, set the right time zone or what you are doing is undefined.

// simple way to get a specific point in time

GregorianCalendar cal = new GregorianCalendar(tz);

cal.set(2009, Calendar.DECEMBER, 31, 20, 15, 00);

cal.set(Calendar.MILLISECONDS, 0);

Date d = cal.getTime();

/\*\*

\* Calculates midnight of the day in which date lies with respect

\* to a time zone.

\*\*/

public Date midnight(Date date, TimeZone tz) {

Calendar cal = new GregorianCalendar(tz);

cal.setTime(date);

cal.set(Calendar.HOUR\_OF\_DAY, 0);

cal.set(Calendar.MINUTE, 0);

cal.set(Calendar.SECOND, 0);

cal.set(Calendar.MILLISECOND, 0);

return cal.getTime();

}

/\*\*

\* Adds a number of days to a date. DST change dates are handled

\* according to the time zone. That's necessary as these days don't

\* have 24 hours.

\*/

public Date addDays(Date date, int days, TimeZone tz) {

Calendar cal = new GregorianCalendar(tz);

cal.setTime(date);

cal.add(Calendar.DATE, days);

return cal.getTime();

}

### AM/PM

Some nations are very comfortable using the 12-hour clock and using AM or PM to clarify which half of the day they mean. But for most of the people the 24-hour clock is far more familiar. Be very careful when using the 12-hour clock. Some people (especially in central Europe) are extremely unfamiliar with it. Imagine you hand out plane tickets with a boarding time like "10. January 12:30 AM". You can be sure that half the passengers will show up half past midnight (which is correct), and the others half past noon (having missed their flight).

As a rule of thumb: There is no zero hour on the 12-hour clock, and a day starts with AM. So 12:00 AM is equal to 0:00 on the 24-hour clock, and 12:00 PM is equal to 12:00 on the 24-hour clock.

I strongly suggest you use the 24-hour clock in applications, and never the AM/PM notion. Or at least let the user choose his preference. On printed documents, such as tickets, you should never use the 12-hour clock. In Java this means to avoid Calendar.HOUR, Calendar.AM\_PM and the "hh" symbol of SimpleDateFormat. Instead use Calendar.HOUR\_OF\_DAY and the "HH" symbol.

### Conversion between time zones

As a rule you can not convert a java.util.Date to a different time zone. This is semantically impossible as a Date is always in UTC. Only a Calendar or a textual representation of a date can be converted into a different time zone. In practice time zone conversions are only necessary when dates are read from or written to text. Typically on a GUI. Or in data exchange interfaces. In these cases a java.text.DateFormat should be used with the right time zone set.

DateFormat indfm = new SimpleDateFormat("MM/dd/yyyy HH'h'mm");

indfm.setTimeZone(TimeZone.getTimeZone("Australia/Sydney"));

Date purchaseDate = dfm.parse("12/31/2007 20h15");

DateFormat outdfm = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

outdfm.setTimeZone(TimeZone.getTimeZone("GMT"));

csvfile.println(outdfm.format(purchaseDate) +" GMT");

Please avoid coding the following method. It has no effect and demonstrates a lack of understanding of Date and Calendar.

// BULLSHIT CODE! This method does nothing.

public static Date convertTz(Date date, TimeZone tz) {

Calendar cal = Calendar.getInstance();

cal.setTimeZone(TimeZone.getTimeZone("UTC"));

cal.setTime(date);

cal.setTimeZone(tz);

return cal.getTime();

}

When changing the time zone of a Calendar object there is a [small catch](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4827490): you need to call Calendar.get(int) immediately after, or your Calendar object may not be updated correctly!

GregorianCalendar c = new GregorianCalendar(TimeZone.getTimeZone("GMT"));

c.set(2012, Calendar.APRIL, 19, 22, 0, 0); // 20.04 00:00

c.set(Calendar.MILLISECOND, 0);

c.getTime(); // calculates other fields and ms value after calls to set()

c.setTimeZone(TimeZone.getTimeZone("Europe/Zurich")); // invalidates all fields

c.get(Calendar.DATE); // calculates fields from ms value

### Periods of time (intervals)

Imagine the following examples that refer to a period of time:

* Begin and end of a Java method call (bench mark)
* The meeting is from 8h30 to 10h00
* The date a piece of hardware was repaired
* My mother's birthday
* During my vacation
* When I was young

Each of these are not as precise as a UNIX time stamp. They refer to more than a certain second in time. Some of these designate the duration of a single day. Others several days.

### Representing a period

Periods of time can have very different semantics. They may require different representations:

* Begin and end of a Java method call: Two UNIX time stamps
* The meeting is from 8h30 to 10h00: UNIX time, duration in minutes
* During my vacation: Two Calendar objects with time zone
* The date a piece of hardware was repaired: Calendar object with time zone
* My mother's birthday: Calendar object with time zone
* When I was young: fuzzy correlation between "young" and an age

When two UNIX time stamps are used for a period, their general contract should be that the beginning time stamp is considered inside the interval, while the end time stamp is considered outside the interval. This way it is possible to represent consecutive intervals without gaps. For representation to a user it may be necessary to specially handle boundary cases: "Monday 00:00 until 24:00" reads easier than "Monday 00:00 until Tuesday 00:00".

### Representing days of the week

In very special occasions it is even necessary to think carefully of how to represent a day of the week in a database. Consider the following situation: A calendaring web application stores recurring calendar entries in the DB. Example: "Monday March 12 2007 01h00 GMT+02:00, recurring every Monday". So the next occurrence is "Monday March 19 2007 01h00 GMT+02:00".

The application chooses to store the start date in GMT without explicitly storing the time zone. Because the user may travel around the globe and wants to see date and time of his calendar entries in the local time zone of his current location. So it makes no sense to store the date in that random time zone that he was in at the time when the calendar entry was created. Thus the start date becomes "2007-03-11 23h00 GMT" which in GMT belongs to a Sunday. If the webapp now stores "recurring every Monday" it will calculate the next occurrence as "2007-03-19 23h00" in GMT (19. is the Monday) which in the user's local time zone is "Tuesday March 20 2007 01h00 GMT+02:00" — one day too late! This happens of course due to the fact that the notion "Monday" has a different meaning in each time zone.

It is apparent that the application also needs to convert the day of the week to GMT. It does that by choosing 0h00 of a nearby Monday in the user's time zone, converting that point in time to GMT and evaluating the day of the week again — resulting in Sunday of course.

### Exchanging date and time information

Care must be taken when transferring date and time information between IT systems. As outlined above the time zone information may be an important part of a date which must not be lost or changed. In other situations the data is independent of a time zone.

In text formats like CSV, XML, HTTP or SMTP headers dates are usually formatted in a human readable way (not as UNIX times). Generally such date formats should always incorporate a time zone to avoid any ambiguities. XML schema for instance uses the ISO 8601 standard where the time zone is optional, where SMTP uses RFC 2822 with a mandatory time zone. The time zone for a given point in time must be given as an absolute offset (not a geographical location). Otherwise interpretation of a such a time may be ambiguous with respect to DST (see below).

ISO 8601 date and time: "2007-02-26T21:23:14.250+01:00"

RFC 2822 date and time: "Mon, 26 Feb 2007 21:23:14 +0100"

Rules of thumb:

* UNIX times may be transferred in any time zone. You can safely choose GMT.
* The time zone information of periods must not be modified.

For persistence (databases) the same rules apply. A store / load cycle of a UNIX time stamp must not modify the data even when in between any of the following has happened: DST has changed, client has changed time zone, database has changed time zone.

### Daylight savings time

During daylight savings time one hour is added to normal time. It usually starts on a Sunday night in spring at 02h00 when the clock jumps to 03h00, skipping one hour. And in autumn the clock jumps back from 03h00 to 02h00, repeating one hour.

As already noted above, DST is nothing else than a time zone. It is important to understand that countries that use DST thus change time zone twice a year. Also Great Britain. Thus just because a guy can see the Greenwich observatory from his window doesn't mean his time zone is GMT (but could be GMT+1 in summer).

For instance in Switzerland normal time is GMT+01:00 and DST is GMT+02:00. DST start and end are a matter of the most absurd political decisionmaking (Australia postponed its DST start in 2006 because of a sports event...). Also some countries like India once used to have DST and now haven't anymore. All this history is recorded in the Java TimeZone database (and also the UNIX time zone databases). Because of the ever changing nature of this data it is important that you keep this DB uptodate. As of 2007 for example the USA will greatly expand the duration of DST (due to some Energy Conservation Act or something).

Please note that while the DST change no such "clock jumping" is observed in GMT: In spring: 02:00+01:00 = 03:00+02:00 = 01:00 GMT and in autumn: 03:00+02:00 = 02:00+01:00 = 01:00 GMT. It is really just a change of the time zone. Our clocks just measure the time in a different zone. But UTC keeps ticking on.

DST causes a bit of a pain for calendaring applications: two days in the year do not have 24 hours in local time. The day in spring has 23 and the one in autumn 25 hours. I don't know of any calendaring application that supports that in its week view...

Sometimes DST is called "summer time" because it is employed during the summer months. But careful, when it's summer on the northern hemisphere, it's winter on the southern hemisphere. Obviously the difference in time zones can vary greatly because of that. Let's compare Zurich, Switzerland and Auckland, New Zealand. During July Zurich is in the GMT+2 and Auckland in the GMT+12 zone, resulting in a clock difference of 10 hours. During December however, Zurich is in the GMT+1 and Auckland in the GMT+13 zone, creating a difference of 12 hours. So better check before skyping with your Kiwi friends!

DST also causes a tiny ambiguity on the end day when users have to enter a time. Not a GUI that I know asks the user for the time zone that he is entering a date and time in. Most GUIs just take the location of the terminal the user is currently on as a reference. Consider the user entering "29.10.2006 02:30" in Zurich. This is the night when Europe switches from DST to normal time. So the hour between 02 and 03 in the morning shows up twice on local clocks. The time entered by the user is therefore not well-defined. It's not clear whether she means 02:30+02:00 or 02:30+01:00 one hour later. Note that the ambiguity is not present in spring. In that night 01:30 is always GMT+01:00, 02:30 does not exist and 03:30 is always GMT+02:00. If this case is important for your application you should check for it. Unfortunately in Java there is no API call that directly tells you the DST end date. But you can test like so (checks if a day has 23 hours):

public boolean isDSTend(Calendar cal) {

int year = cal.get(Calendar.YEAR);

int month = cal.get(Calendar.MONTH);

int date = cal.get(Calendar.DATE);

TimeZone tz = cal.getTimeZone();

return isDSTend(year, month, date, tz);

}

public boolean isDSTend(int year, int month, int date, TimeZone tz) {

java.util.Calendar cal = new java.util.GregorianCalendar(tz);

cal.set(java.util.Calendar.MILLISECOND, 0);

cal.set(year, month, date, 00, 30, 00);

cal.add(java.util.Calendar.HOUR\_OF\_DAY, 24);

return (23 == cal.get(java.util.Calendar.HOUR\_OF\_DAY));

}

### Week numbers

Week numbers are popular among managers: "can we arrange a meeting in week 34?". By week numbers they usually mean the number displayed in MS Outlook. What they don't know: [ISO-8601](http://en.wikipedia.org/wiki/ISO_8601) defines the week numbers, Outlook's implementation is not (necessarily) compatible with that. And week numbers have a small ambiguity. Java's Calendar class can be made ISO compatible with setMinimalDaysInFirstWeek(4).

The thing is: 30 December 2002 is in week 1 of the year 2003. But Java's Calendar class only has one YEAR field. This year field has different meanings depending on how the Calendar object is used:

* When setting the WEEK\_OF\_YEAR and YEAR field, YEAR is interpreted as the year of the week.
* When setting the DATE and YEAR field, YEAR is interpreted as the year of the date.
* When getting the YEAR field, always the year of the date is returned.
* There is no direct way to determine the year for the week field.

Sun [refuses](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6218127) to fix the bug. Use the following method to determine the correct year for the week field:

public int getYearForWeek(GregorianCalendar cal) {

int year = cal.get(Calendar.YEAR);

int week = cal.get(Calendar.WEEK\_OF\_YEAR);

int dayOfMonth = cal.get(Calendar.DAY\_OF\_MONTH);

if (week == 1 && dayOfMonth > 20)

return year + 1;

if (week >= 52 && dayOfMonth < 10)

return year - 1;

return year;

}