Error Handling With Exceptions

Jakob Spies

JSpies@users.sourceforge.net

28. 1. 2009

Contents

Preface

TEACHER: This was the most offensive science report I've ever heard. How could you do such a thing?
BUTT-HEAD: Beavis helped me.

Based on my experience in object-oriented software development projects I got the impression that error handling is one of the most neglected aspects of programming. Even highly talented and experienced developers tend to get careless when it comes to exception handling. Even expensive and critical software products show weird and annoying behavior in error situations ("An unknown error has occurred. OK?"). Even million-dollar software projects with dozens of developers do not have a rigorous concept for logging of errors.

This paper presents guidelines for error handling, reaching from general rules applicable everywhere to more detailed proposals that might be adopted or not, or only with variations. Special emphasis is put on examples for good and bad practices with respect to the treatment of exceptions. The focus is on Java since Java has the richest exception framework (checked and unchecked exceptions, stack traces, built-in nesting of exceptions since JDK 1.4), most statements, however, apply to other object-oriented languages as well.

Please do not think that any of the bad practices demonstrated in the examples has been invented for the purpose of this book. I have seen them all.

Audience

This paper has a practical goal, namely improving the world. More precisely, it aims at increasing software quality by improving programming practices, therefore it is addressed to developers and designers of object-oriented software. The reader of this paper should be familiar with the Java programming language.

Acknowledgements

Thanks are due to Frank Neugebauer for a good example where it is appropriate to throw a checked exception.

Chapter 1

Introduction

1.1 Errors

A rigorous definition of the term "error" in the context of computer software seems to be impossible. In my opinion it is best to just rely on common sense. The problem with errors is that they may be well anticipated, and in how far is an expected outcome of some operation an error? Usually this is part of the definition of the operation itself.

Example 1.1.1. Let ω be the operation to open a file specified by a path in the file system for writing. Then is it an error if this file does not exist? It depends on the specification details of ω. Possibilities:

1. the designer of ω decides to return a nil file handle in this situation and reserves completing the operation abruptly for really bad cases, e.g. if the file exists but cannot be opened
2. yes, a `FileNotFoundException` is thrown
3. ω silently creates the file if it does not exist and opens this new file for writing

Example 1.1.2. An HTTP server has a socket connection with an HTTP client. When the server tries to write the HTTP response to the socket, an exception occurs if in the meantime the client has closed the connection, after having sent its request. From the point of view of the write-to-socket operation it is an error to write to a closed socket and it is justified to throw the exception. But the HTTP server must not make assumptions on how long a client will keep the connection open, so from this perspective the incident is not an error, neither of the client nor of the server.

1.2 Significance of Error Handling

In real life operations of software systems *will* fail occasionally (on a large variety of reasons). Whether it is possible to make the operation work again and what amount of efforts this costs depends heavily on the quality of error handling in the software.

1.2.1 Usability

Adequate treatment of errors is a key factor for the usability of a software product.

An experience shared by almost every computer user is that programs fail with a cryptic, nonsensical, idiotic, useless, plainly wrong, or even missing error

message. Getting no useful hint as to the cause of the failure, the user spends one week with cursing, trying a lot of things like applying changes to the configuration, surfing Usenet newsgroups, hurting their friends' feelings, and in the worst case ends up with spending another week for a big backup, reinstallation of the operating system and all the software running on it. (And the real reason was a corrupted preferences file. Deleting this file would have resolved the trouble. The program noticed that its preferences file was corrupt, but it did not tell the user.)

Correct (which means first of all: complete) reporting of error information can make the difference between spending 5 minutes or a week on finding the cause of an error (cf. § 7.1).

Example 1.2.1. In a project with Enterprise JavaBeans, Java Data Objects, and a relational database, one of my unit tests suddenly failed in a strange way. When I tried to create a persistent object of a certain type, the transaction was rolled back, which I could see from the exception my test client received and also at the application server console, but I could not figure out why the transaction was rolled back, even though the error was perfectly reproducible. Neither the console output nor the application server log files contained a trace of a potential cause of my error. I analyzed the statements the JDO layer was sending to the database, they seemed to be correct. I tried this and that for several days, and eventually obtained the source code of the application server, debugged into the application server's EJB transaction management code, and found the following: The application server had dropped a lovely exception, containing the message that the `INSERT` operation in the database was impossible due to uniqueness constraints, into the bit bucket. If this exception had been logged, the bug in my software would have been immediately clear and would have been fixed in no time. But without the stolen error information I did not even know that the error occurred in the database.

And this is a matter of usability. It is a natural requirement to an application server that it be convenient to develop software on it. The application server has to be aware that the software it is hosting can be arbitrarily buggy and the server must not hamper debugging the foreign software. If an application server induces enormous development costs caused by excessive debugging efforts, it becomes unusable: customers will then prefer another application server.

1.2.2 Maintainability

This is about bugs in the software itself. The error handling has significant impact on how difficult it is to track down bugs. A good error handling strategy can even help to prevent the introduction of bugs, as will be shown in this paper (cf. e.g. Ex. 5.2.3).

Example 1.2.2. A complex method `foo` somewhere dereferences a nil pointer and therefore throws a `NullPointerException`. If it is called

```
try{ foo(); }
catch(Exception e){}
```

so the `NullPointerException` is completely swallowed, without any consequences, then the development team is going to have a hard time detecting and locating the bug, since the error will manifest itself only in unspecific misbehavior of the software rather than in an explicit error message. It is unfortunately an illusion to think that nobody would write such a `try` statement. Even a little `e.printStackTrace()` in the catch handler would improve things substantially.

1.3 Erroneous Error Handling

I can only speculate on the reasons why error handling code is nearly always the most deficient code in software systems. Users/customers need software because they want the software to help them in certain ways, so what they are interested in is the "tangible", "productive", "good-case" functionality. When you buy a car, you probably won't ask how it reacts if the fuel gets empty during driving (which you should really be aware of if it has a Diesel engine). And too often also business consultants are not aware of potential errors. That's why error handling tends to be given less weight in software requirements, software specifications, and consequently in the implementations. In addition to that, software developers like to concentrate on the challenge of designing elegant software models for the desired functionality. They can do so without paying the necessary attention to the more stupid, routine, tedious, and therefore boring error handling because

- the compiler does not punish them for poor error handling
- they do not violate error handling guidelines because these do not exist
- error handling is usually not tested

As to be expected in information processing technology, the majority of bad practices with respect to error handling is related to letting information escape:

- information available only at runtime is discarded (cf. 5.7, 7.1, 5.8, 7.4, 6.6, 7.6)
- information available at coding time is not reflected in the (compiled) code (cf. 4.1, 5.2, 5.3, 5.6.2, 5.5.4, Ex. 5.8.1, 7.6, 9.1)

Then there are of course the "trivial" cases of not using exceptions where exceptions should be used, using exceptions where they should not be used, using checked exceptions where unchecked exceptions should be used and vice versa. The guidelines presented in this paper will cover most of the situations in which doubts about such questions might exist.

Chapter 2

Terminology and Notation

Definition 2.1. A **static class** is a class having no instance.

Remark 2.0.1. A static class should have no accessible constructor (prevent instantiation) and no instance members (because they would be useless).

Definition 2.2. Classes are always considered to be derived from themselves. Every class is a subclass of itself. Let C be a non-final class. In practice, there is now still an ambiguity in the term "the set of all subclasses of C", because in a concrete software system (application, library) only finitely many subclasses of C really occur. Only a certain finite set of subclasses of C will be compiled together, and only finitely many subclasses of C will live together in one class loader or in one process at runtime. If it is necessary to distinguish between these notions, the set of all subclasses of C, regardless of whether they have already been written, belong to our software or not, will be referred to as the **abstract set of subclasses** of C. If we restrict this to a concrete software system, we speak of the **concrete set of subclasses** of C.

Definition 2.3. We say that a class X is **trivially derived** from a class S if X does not really extend S, i.e. it has no additional attributes or methods, does not override any methods or hide fields, has only constructors matching the constructors of S with respect to their signature and only calling the corresponding constructors of S.

Example 2.0.1. A trivial derivation:

```
class S
{
   // constructors

   S(){
   ...
   }

   S(int n){
   ...
   }

   // attributes, methods
   ...
}

class X extends S
{
   X(){
```

```
      super();
   }

   X(int n){
      super(n);
   }
}
```

Definition 2.4. The fact that a method f has a `throws` clause containing the exception class E will be referred to by saying "f **declares to throw** E" or, even shorter, "f **declares** E".

Definition 2.5. A **catch handler** is nothing but a `catch` block. This is where **handling** of exceptions takes place.

Definition 2.6. `throws` clauses as well as catch handlers specify an exception class c (a class derived from `Throwable`). When speaking of the set of exception classes throwable by the method f having the `throws` clause or being caught by a catch handler, respectively, we normally mean the concrete set of subclasses of c in the sense of Def. 2.2. If we think of the abstract set of subclasses of c, this is indicated by saying "the **abstract set of exceptions throwable** by f" or "the **abstract set of exceptions caught** by the catch handler", respectively.

Definition 2.7. To **rethrow** an exception means that a catch handler throws the exception it has caught.

Definition 2.8. An exception is **raised** if it is thrown but not rethrown; in other words: an exception is raised if it is thrown but has not been caught on its way up from deeper levels of the call stack.

In most cases raising an exception object implies creation (via a constructor) of this object; the exception to that being the case where the exception object had already been created before the catch handler was entered (5.9).

Unless otherwise stated, packages are understood hierarchically, i.e. the statement "class X belongs to package p" means "class X lies in p, or a subpackage of p, or a subpackage thereof etc." in the diction of [1].

For the examples it is assumed that we are an organization the Java code of which lives in the package name space `org.acme`, and the code under consideration is contained in a package `org.acme.project`.

Chapter 3

Exceptions

3.1 Benefit of Exceptions

Exceptions are one of the biggest advantages of modern programming languages over their predecessors without exception concept. The idea of exceptions existed before object-oriented languages, but the concept of exception *objects* of course not. The purpose of exceptions is simply to separate error handling from the normal control flow and to get some freedom on where to handle the errors.

Example 3.1.1. C code often tended to look cluttered like this (simplified; even worse in reality):

```
int foo(){
   int myErr;
   int* isBar1;

   myErr = bar1(isBar1);
   if (myErr != kNoErr){
      fprintf(
         stderr,
         "bar1 has failed with return code %i in %d of %s",
         myErr, __LINE__, __FILE__
      )
      return myErr;
   }

   if (isBar1){
      myErr = bar2();
      if (myErr != kNoErr){
         fprintf(
            stderr,
            "bar2 has failed with return code %i in %d of %s",
            myErr, __LINE__, __FILE__
         )
         return myErr;
      }
   }

  // and so on
}
```

As there was no simple or standard means for passing error information up the call stack, the best solution was to handle (e.g. log) each error immediately where it occurred, which had the consequence that logging and error handling code was scattered all over the business logic.

In Java (analogously in C++) this could be replaced with

```
void foo(){
   try{
      if (bar1()) bar2();
     // and so on; business control flow not obscured anymore
   }catch(SomeException e){
      // e knows where it has occurred
      e.printStackTrace();
   }
}
```

or even, if we let the client handle the error:

```
void foo() throws SomeException
{
   if (bar1()) bar2();
   // and so on
}
```

3.2 Exceptions and Control Flow

Exceptions, however, can be misused for controling the flow of normal business operations, which should be avoided since it leads to less easily comprehensible code (which is bad for bug prevention and maintainability), last but not least because a person reading the code expects exceptions to be used in error situations, as their very purpose is to separate error handling from business logic. For example, the custom of recording exception contents (cf. 7.1) would become wrong in a situation where the exception does not really indicate an error.

Another argument against using exceptions in non-error situations is performance: Raising an exception means creating (and later garbage-collecting) an object, which is *per se* a more time-consuming operation than, say, checking `if (x == null)` and returning `false` instead of `true`, but exceptions are especially expensive to create because their creation involves filling in the current stack trace. This is not a problem if the rule that exceptions are designed for exceptional situations is obeyed. It can cause severe performance problems, however, if exceptions are raised in normal situations. The cost of creating exceptions is quantified in the next section 3.3.

There are basically two ways in which said misuse can occur: implied by methods throwing exceptions in non-error situations, or by client code provoking exceptions deliberately. The first of these bad practices is discussed in 5.4. The second one shall be illustrated by some examples.

Example 3.2.1. A `NullPointerException` is provoked:

```
try{
   a = x.getA();
}catch(NullPointerException e){
   // x is nil
   a = 0;
}
```

The programmer of this code obviously thinks that the only reason for a `NullPointerException` can be that `x == null`, but this is wrong if there is any possibility that `getA` (maybe in a future version) throws a `NullPointerException` (which then should not be swallowed, cf. Ch. 7). The following code

```
a = x == null? 0 : x.getA();
```

does not misuse exceptions, expresses what the programmer really means, is shorter and more easily legible.

Example 3.2.2. The method `Class.getMethod` throws a `NoSuchMethodException` if it is called for a non-existing method. So is it an error if a given class does not contain a method with a certain signature? Of course not. What this `throws` clause really clearly expresses is that the designers of class `Class` mean that it is a mistake of the caller to call `Class.getMethod` for a non-existing method. If it is not sure which methods a given class has, the method `Class.getMethods`, listing all (public) methods of the class, is to be used. If one wants to check whether a class has a method with a given signature, the implementation

```
boolean isDefined(String name, Class[] parm, Class c){
   try{
      c.getMethod(name, parm);
      return true;
   }catch(NoSuchMethodException e){
      return false;
   }catch(SecurityException e){
      return false;
   }
}
```

unnecessarily provokes exceptions and does not use the `Class` API in the prescribed way. The implementation

```
boolean isDefined(String name, Class[] parm, Class c){
   Method[] m = c.getMethods();

   methodsLoop:
   for (int i = 0; i < m.length; i++){
      Method f = m[i];

      if (f.getName().equals(name)){
         Class[] p = f.getParameterTypes();
         int n = p.length;

         if (n != parm.length) continue methodsLoop;
         for (int j = 0; j < n; j++)
            if (p[j] != parm[j]) continue methodsLoop;

         return true;
      }
   }

   return false;
}
```

using `getMethods` is in accordance with the `Class` designers' intentions, does not misuse exceptions, and is probably not slower at runtime (a little bit slower at coding time, though, admittedly).

Example 3.2.3. The XML parser Xerces from Apache is built into Sun's JRE 5. It has the bad behavior of creating 7 `XMLConfigurationException`s every time it parses an XML document (successfully!). The reason is that it internally wants to know if certain "features" are available and to this aim calls a method `getFeature` which throws an `XMLConfigurationException` if the feature is not available. Literal quotation from the class `com.sun.org.apache.xerces.internal.impl.XMLEntityManager`:

```
try {
    fValidation = componentManager.getFeature(VALIDATION);
}
catch (XMLConfigurationException e) {
    fValidation = false;
}
```

A correct implementation would use a method that simply and naturally returns `true` or `false` for telling whether the feature is available or not. In a real-life application (for administration of an application server) doing a lot of XML parsing this bug caused a time overhead of 70 %.

Abusing exceptions for flow control is even dangerous when the exceptions are unchecked, because the compiler does not verify whether the called code really declares the exception.

Example 3.2.4. A method `getAttributeValue(String name)` of a certain class `org.acme.framework.Node` throws the unchecked exception `Precondition` if the `Node` on which it is invoked does not have an attribute of the given name. A client in a certain enterprise software platform, let's call it `org.acme.portal.Mapping`, uses this method as follows, for a `Node x`:

```
String v;

try{ v = x.getAttributeValue(name); }
catch (Precondition e) { v = null; }
```

As one can see, it is not an error from the point of view of the client if the attribute does not exist, the client just does not care. In addition to the unnecessary exception provoked here, which costs runtime performance, the client code exposes itself to the danger that the method `getAttributeValue` can change its behavior of throwing `Precondition` without breaking a contract visible to the compiler. The owner of `Node` might even decide to throw another unchecked exception in the case of the non-existent attribute, which would invalidate the client code semantically — but the client would not get notified by the compiler.

Remark 3.2.5. How should such a problem be solved? Regardless of whether the exception in question is checked or unchecked: if the class `Node` provides no other way to retrieve attribute values or to find out which attributes a `Node` has, there is no better way for the client to program the desired functionality. But this very fact shows that the API of `org.acme.framework` is not well-designed: there is an obvious need for a method, let's call it `Node.getAttributeValueMildly`, providing the value of an attribute identified by its name, and *not throwing* if such an attribute does not exist. It could just return nil in this case. Using such a method the code snippet in Ex. 3.2.4 would shrink to:

```
String v = x.getAttributeValueMildly(name);
```

, much simpler, clearer, and faster at runtime. If the owner of `Node` can be convinced to provide this method (which is not a problem since it is only a slight modification of `getAttributeValue`) and clients can be convinced to use it where appropriate, it can turn out that the old `getAttributeValue` is not used anymore — all the better: it can be deleted and we have achieved a nice refactoring, improving performance and maintainability.

3.3 The Price of Exceptions

As promised above, here are figures for the cost of the creation of exceptions. The Java application the source code of which is printed below compares two "algorithms" for finding out whether an object is nil: `goodCheckNil` and

`badCheckNil`. The first one is natural, the second one misuses an exception for flow control. The example is very simple, but prototypical.

On my old Macintosh PowerBook with a single 1.67 GHz PowerPC G4 processor and JRE 5 the `goodCheckNil` takes an average of 9.8 ns, whereas `badCheckNil` consumes the following time, depending on the length of the call stack (all times in nanoseconds):

Stack Length	Time
32	9815
64	16418
128	29059
256	55464
512	112518

On a Linux 2.6.5 machine with 4 2.6 GHz dual-core 64-bit processors and JRE 5 the `goodCheckNil` takes an average of 1.155 ns, whereas `badCheckNil` takes:

Stack Length	Time
32	3093.5
64	4921
128	8488.5
256	16416
512	32512

As you can see, the cost of the exception depends almost linearly on the length of the call stack. This includes the time for creation as well as garbage collection of the exception object. (I made sure that the exceptions were also garbage-collected during my test runs.) In a complex enterprise application running on an application server call stack lengths of 300 are normal, so we have to assume a time penalty of about 20 μs for the creation of a simple exception without error message (on this Linux server).

Here is the test code:

```
package jakob.exception.overhead.measurement;

public class TimeMeter
{
   private final static boolean USE_EXCEPTION =
      Boolean.getBoolean("use-exception");

   private static int maxStackCount;
   private static int currentStackCount;

   /**
    @param a
      <ol>
         <li>stack length
         <li>number of measurement rounds
         <li>number of executions per round
      </ol>
    */
   public static void main(String[] a){
      if (++currentStackCount == 1){
         maxStackCount = Integer.parseInt(a[0]);
         System.out.println("stack length " + maxStackCount);
         maxStackCount -= 4;
      }
```

```
      // maxStackCount is set
      if (currentStackCount < maxStackCount) main(a);
      else{
         // desired stack length reached
         test(a);
      }
   }

   private static void badCheckNilInner(Object x){
      if (x == null) throw new NullPointerException();
   }

   private static boolean goodCheckNilInner(Object x){
      return x == null;
   }

   private static boolean badCheckNil(Object x){
      try{
         badCheckNilInner(x);
         return false;
      }catch(NullPointerException e){
         return true;
      }
   }

   private static boolean goodCheckNil(Object x){
      return goodCheckNilInner(x);
   }

   private static void execute(Object x){
      if (USE_EXCEPTION) badCheckNil(x);
      else goodCheckNil(x);
   }

   private static String printTime(double ms, int n){
      return String.valueOf(ms * 1000000 / n) + " ns";
   }

   private static void test(final String[] a){
      final int nRounds = Integer.parseInt(a[1]);
      final int nExecutions = Integer.parseInt(a[2]);
      long s = Long.MAX_VALUE;
      long time;
      int n;
      final int warmUp = 50000;
      Object x = null;
      double sum = 0;

      // HotSpot warm-up:
      // ensure that execute() gets compiled to native machine code
      for (n = 0; n < warmUp; ++n) execute(x);

      for (int i = 0; i <= nRounds; ++i){
         time = System.currentTimeMillis();
         for(n = 0; n < nExecutions; ++n) execute(x);
         time = System.currentTimeMillis() - time;
```

```
            if (i == 0) continue;
            if (s > time) s = time;
            sum += time;
        }

        System.out.println("minimum " + printTime(s, nExecutions));
        System.out.println("average " + printTime(sum / nRounds, nExecutions));
    }
}
```

In my tests it was run with `a[1]` = 20 and `a[2]` = 50000 if `use-exception=true` and `a[2]` = 10000000 if `use-exception=false`.

3.4 Compile-Time Checking of Exceptions

Let me explain checking of exceptions briefly. This section does not contain anything that is not described in [1]; it is included here merely for convenience of the reader. For convenience of the writer and better understanding we introduce a little *terminus technicus*:

Definition 3.1. Let b be a block (cf. [1], 14.2) of code. The **visible exception set** of this block is the set of the classes of those checked exceptions raised in b and checked exceptions declared by methods called in b which can really get out of b, i.e. are not caught away by catch handlers.

Example 3.4.1. Consider the following block:

```
File f = new File("data");
try{
   if (f == null) throw new ClassNotFoundException();
   OutputStream o = new FileOutputStream(f);
   o.write(1);
   o.close();
}catch(FileNotFoundException e){
   e.printStackTrace();
}
```

Its visible exception set is the union of the set of all subclasses of `ClassNotFoundException` with the set of all subclasses of `IOException`, excluding `FileNotFoundException` and its subclasses: `ClassNotFoundException` is raised in the block (the question whether the code raising it is ever executed is irrelevant here) and not caught away. `IOException` is declared by methods called in the block, but its subclass `FileNotFoundException` is caught away. The `File` constructor in the first statement documents that it can throw `NullPointerException`, but this does not matter, as only checked exceptions are taken into account.

The crucial point is that the visible exception set of a block (and this is the justification for the attribute "visible") can be determined by looking only at the code of the block and the signatures of the methods called in this block, without knowledge about what exceptions the methods called in the block would really throw at runtime and without reasoning which code branches would be executed under which runtime conditions. In other words, the visible exception set is completely determined at compile time and its actual determination does not require sophisticated reasoning.

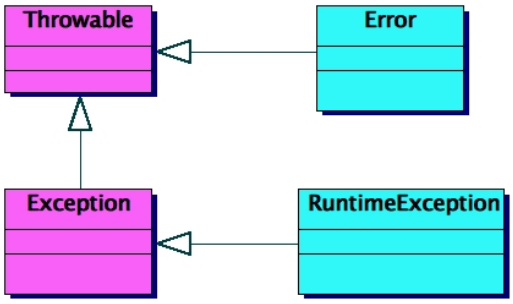
The semantical difference between checked and unchecked exceptions is that the compiler enforces that

- the visible exception set of the body of a method is contained in the set of checked exceptions declared by this method

- the set of checked exceptions declared by a method f' overriding (or implementing) another method f is contained in the set of checked exceptions declared by f
- for each catch handler following a `try` statement b, if its exception argument type is a checked exception class, it has to be the superclass of an element of the visible exception set of b (examples are given below),

whereas the compiler does not keep track of unchecked exceptions in any way — with one exception dictated by the prohibition of unreachable code ([1], 14.20): the compiler recognizes which types of unchecked exceptions are caught in catch handlers; it is not allowed for a catch handler that its argument type is contained in the set of exception classes covered by the preceding catch blocks of the same `try` block, and this is required also for unchecked exception classes.

The syntactical difference is that an exception class is unchecked if and only if it is derived from `java.lang.RuntimeException` or `java.lang.Error`:



Base exception classes in `java.lang`

Examples Consider the following block b:

```
if (isNight()) throw new ClassNotFoundException();
if (isWar()) throw new IllegalStateException();
if (isCold()) throw new IOException();
```

where we assume that the 3 functions called in b do not declare exceptions. The visible exception set S of b consists of all classes derived from `ClassNotFoundException` or `IOException` (`IllegalStateException` is not taken into account since it is derived from `RuntimeException`). The compiler does not care which unchecked exceptions might be thrown by b.

Example 3.4.2. Not allowed:

```
try{ b }
catch(CloneNotSupportedException e)
```

because no subclass of `CloneNotSupportedException` is contained in S.

Example 3.4.3. Allowed:

```
try{ b }
catch(FileNotFoundException e)
```

because `FileNotFoundException` is contained in S (and is a superclass of itself).

Example 3.4.4. Allowed:

```
try{ b }
catch(SecurityException e)
```

because `SecurityException` is unchecked and not caught by a preceding catch block.

Let b' be the following `try` statement:

```
try{ b }
catch(FileNotFoundException e){}
catch(SecurityException e){}
```

The visible exception set S' of b' is the set of all subclasses of `IOException` or `ClassNotFoundException`, excluding the subclasses of `FileNotFoundException`.

Example 3.4.5. The construction

b'
```
catch(ArrayIndexOutOfBoundsException e)
```

is allowed since the unchecked `ArrayIndexOutOfBoundsException` is not caught by the two preceding catch handlers.

Example 3.4.6. The construction

b'
```
catch(SecurityException e)
```

is not allowed since, although the compiler does not care about unchecked exceptions thrown by b, it keeps track of unchecked exceptions caught in the subsequent `catch` blocks, so it knows that all subclasses of the unchecked `SecurityException` have already been caught in b' and the additional catch handler would be unreachable.

Example 3.4.7. The construction

b'
```
catch(Exception e)
```

which will be subject to further investigation below, is allowed since `Exception` is the superclass of an element of S', e.g. `IOException`.

Chapter 4

Exception Types

4.1 Raise Custom Exceptions

It is generally a good idea to use custom exceptions. More precisely, code should create only exceptions the definition of which is under the control of the code owner.

Example 4.1.1. Whenever our code (in `org.acme.project`) creates an exception object, its class should belong to `org.acme.project` (our project) or `org.acme` (exception classes used in multiple projects). We should not create, e.g., `java.lang.IllegalArgumentException`s.

In cases where the set of exceptions is determined by foreign software, viz. if a foreign method is overridden or an abstract foreign method implemented, simply derive a custom exception class from the prescribed one.

Example 4.1.2. We write a custom `BufferedReader`:

```
import java.io.BufferedReader;
import java.io.IOException;

public class MyBufferedReader extends BufferedReader
{
   public int read() throws IOException
   {
      if (!ready()) throw new NotReady();
      return super.read();
   }
}

public class NotReady extends IOException
{}
```

If one had chosen to throw `new IOException("not ready")` (as of this writing, `IOException` has only 2 constructors, one taking a string and one taking no argument), how would the client handle the invokation of `MyBufferedReader.read`? Along these lines:

```
try{
   myBufferedReader.read();
}catch(IOException e){
   if ("not ready".equals(e.getMessage())){
      // probably (!) not ready
      // wait
      ...
   }else{
```

```
      // normal IOException stemming from java.io
      ...
   }
 }
```

whereas the introduction of the custom exception class `NotReady` allows for the less cumbersome and error-prone

```
try{
   myBufferedReader.read();
}catch(NotReady e){
   // certainly (!) not ready
   // wait
   ...
}catch(IOException e){
    // normal IOException stemming from java.io
    ...
 }
```

The advantages of not raising foreign exceptions are twofold: The client sees at a glance (viz. by the exception type) if the exception was raised by our code or by foreign code. This information can be useful if the contract with the client contains conventions about exception contents, or merely for knowing whom to ask about the meaning or reasons of the exception. On the other hand, and more importantly, due to the fact that the exceptions are ours, we are able to change their definition, e.g. for accommodating them to our logging framework (cf. 6) or for including additional data.

The requirement to raise only first-party exceptions can be lifted if the respective method has only local accessibility such that callers of the method are always visible together with this method, e.g. if it is a private method.

The only, and sometimes valid, argument against using first-party exceptions is lack of time to write these exception classes.

4.2 Nesting Exceptions

As it is often the case (cf. sec. 7) that on catching one exception one has to raise another exception and wants to put all available error information into it, it is a good idea to provide for attaching the original exception to the new one. This simply means that exceptions raised by our code should have at least an instance field of type `Throwable` where primary exceptions can be stored. With JDKs of version ≥ 1.4 this requirement is automatically satisfied, but if more than one exception is to be aggregated (which e.g. `javax.jdo.JDOException` supports), extensions are still necessary.

4.3 Mother of All Exceptions

The next good idea is to derive all checked exception classes (directly or indirectly) from one base exception class (the mother of all checked exceptions) and all unchecked exceptions from another exception class derived from `java.lang.RuntimeException`. In languages supporting multiple inheritance one would be tempted to write a single base class for these exception mother classes. This strategy makes it possible to change behavior of all exceptions (e.g. with respect to logging (cf. 6.1)) by just changing 1 or 2 classes.

4.4 Naming

Some people say that names of exception classes should always end with `Exception`, but I have never been presented any rational reason for this. IMHO, this is a case of confusing names and types (as known from the file systems used by the infamous operating system "Windows"). It would be one of the last rudiments of "hungarian notation". Java knows if a class is an exception or not. A class is `Throwable` if and only if it can be thrown. If one catches an object in a catch handler, one knows that it must be an exception, no matter what the class name is. Even Javadoc output separates exception classes from other classes, and this distinction is not based on the class name.

4.5 Module-Dependent Exceptions

This section is not referenced by any other section in this paper, therefore it is admissible to make some forward references to later sections without breaking logical consistency.

Let us consider a software system consisting of a set Γ of interdependent subsystems called modules. We want to assess the following convention, derived from 4.3:

(*) Each module $X \in \Gamma$ defines a checked exception class E_X such that all checked exceptions thrown out of this module are derived from E_X.

Let B be the most specific common superclass of the E_X, $X \in \Gamma$ (e.g. the mother class from 4.3, or just `java.lang.Exception`).

The convention (*) certainly makes sense for an $X \in \Gamma$ if E_X is non-trivially derived (Def. 2.3) from B, e.g. if one knows that every action in X is related to, say, a transaction ID, and this transaction ID should be an attribute of every exception raised in X, whereas there is no such attribute in B.

The problem with (*) is that if we have $X, Y \in \Gamma$, where X uses Y, then it is not possible for X to let exceptions from Y just fly through; it has to catch and convert them (cf. 7.2.2) into a subclass of E_X. (Unless E_Y was derived from E_X, which we will not assume here because this is rarely feasible or desirable in a sufficiently complex software system. (Just think of attributes of E_X that do not make sense in Y. Or what if Y is also used by another module having nothing to do with X (we have no multiple inheritance in Java)?) And this necessity of conversion requires a lot of otherwise superfluous catch handlers and the danger of loss of error information (cf. 7.4), especially of backtraces and attributes of the Y exceptions not matched in E_X.

The recommendation to be given here is consequently to view (*) with caution. Real objective reasons for requiring (*) are seldom found. At least (*) should be restricted to base modules (using no other modules) where E_X is a non-trivial derivation of B.

Moreover, a phenomenon commonly envisaged in modules X obeying (*) is that methods declare exactly E_X. The infomation carried by the exception type is then which module the method belongs to (which is redundant) and nothing else. This contradicts the principle that classes listed in `throws` clauses should correspond to *error types* requiring different handling (5.6.2).

Example 4.5.1. This time we write a J2EE application server. It consists of several modules, called services in this case, among them a persistence service responsible for accessing persistent configuration data used internally by the application server. (*) would suggest that there exists a class, say, `PersistenceException`, such that every checked exception declared in the public API of this persistence service is derived from it.

In fact, our application server has its own sophisticated exception hierarchy; all checked exceptions defined in our code have a common super class

`CheckedException`, this is the "most specific common superclass" B in our case.

(*) requires that all checked exceptions thrown by the deployment service also inherit from a common superclass; let us call it `DeploymentException`. Now it is likely that `DeploymentException` is not derived from `PersistenceException`, because, say, `PersistenceException` contains details of a database connection, which is not inherent to (all) deployment errors, which, e.g., could be caused by a incorrect deployment descriptor. Another motivation to keep `DeploymentException` independent of `PersistenceException` might be to hide the fact that the deployment service uses a database from clients of the deployment service.

Thus, assuming that `DeploymentException` is not derived from `PersistenceException`, we are in the bad situation described above: The deployment service cannot do anything if there is a problem with the database, nonetheless, it has to catch all `PersistenceException`s for no other purpose than converting (cf. 7.2.2) them into `DeploymentException`s, which implies a lot of catch handlers that could simply be omitted if (*) was not required of the deployment service.

Chapter 5

Throwing

5.1 Errors

Even though not explicitly stated in [1] or the JDK documentation, `java.lang.Error`s (i.e. objects being `instanceof Error`) are reserved for the Java runtime system (the JVM and inner (especially native) parts of the JDK). Not raising `Error`s has the simple advantage that client code can be sure that all `Error`s come from the Java runtime and not from our code. Everything that can be achieved with `Error`s can as well be achieved with `RuntimeException`s.

5.2 Imprecise Declarations

Object-oriented programming should aim at describing interfaces as exactly as possible with the means the language provides so the compiler can guarantee compliance to the contract.

Example 5.2.1. From this point of view it is better for a method `foo` that is to return a set of strings to be declared as

```
String[] foo()
```

than as

```
/**
@return collection of {@link String} objects
*/
Collection foo()
```

because in the first case the compiler would not allow `foo` to return objects that are not strings, whereas in the second case the contract is only defined by documentation and `foo` can be misimplemented by putting non-string objects into the collection returned, with the error showing up only at runtime.

There can, however, exist reasons, e.g. performance considerations, for not defining the interface as closely as possible in terms of the programming language.

With respect to the declaration of exceptions this means that the `throws` clause of a method should reflect the checked exception classes that can be thrown by this method and its overridings as closely as possible. Deviations from this principle are discussed below. As it is not admissible in the `throws` clause to omit checked exceptions that are thrown ([1], 8.4.4, 11.2), this boils down to saying that the set of classes defined in the `throws` clause should be as small as possible. Declaring exceptions that are not thrown makes exception handling harder for the client.

Example 5.2.2. Assume we have to write a method `foo` for creating a file "data" if it does not exist yet. We use the method `createNewFile` of class `java.io.File` declaring `java.io.IOException` and nothing else.

```
void foo() throws IOException, ClassNotFoundException
{
   new File("data").createNewFile();
}
```

Assuming that we know that no method overriding this `foo` will be required to throw `ClassNotFoundException`, this is nonsense and forces the client to implement some unnecessary handling of `ClassNotFoundException`s.

This example deserves to be called silly, but code like this is being written in real life.

Example 5.2.3. Low-level methods `foo` and `bar` declare to throw `Foo` and `Bar`, respectively. These exception classes are semantically different and require different handling, although they are both derived from a common superclass `Mother`, as every existing or future checked exception class defined in our software. The programmer Knut of the high-level method

```
/** @throws Mother on error */
void fnord() throws Mother
{
   foo();
   bar();
}
```

was too lazy to declare (and document) `Foo` and `Bar` individually. The caller Heinz of `fnord` should handle `Foo` and `Bar` separately:

```
try{ fnord(); }
catch(Foo e){ handleFoo(e); }
catch(Bar e){ handleBar(e); }
...
```

but how can he be expected to know this? Only by verbal tradition or by inspection of the implementation of `fnord` and the documentation of `foo` and `bar`. The declared interface of `fnord` and its documentation suggest to catch `Mother`; the actually relevant exceptions `Foo` and `Bar` are mentioned neither in the Java interface nor in its documentation.

Now assume that after falling on his nose with

```
try{ fnord(); }
catch(Mother e){
   ???
}
```

Heinz gets word from Knut about the hidden real contract of `fnord` and writes the correct catch handlers for `Foo` and `Bar`. But only catching `Foo` and `Bar` does not work; the compiler demands that `Mother` also be caught, for it is declared by `fnord`. How is `Mother` to be handled? Since Heinz was told that only `Foo` and `Bar` are really thrown, catching another exception indicates a bug in the guts of `fnord`, therefore (according to guideline (2) in 7.6) the `catch` block for `Mother` should throw some unchecked exception, let's call it `Bug`:

```
try{ fnord(); }
catch(Foo e){ handleFoo(e); }
catch(Bar e){ handleBar(e); }
catch(Mother e){ throw new Bug(e); }
```

This is O.K. — until `fnord` undergoes the following change: Due to his laziness, programmer Knut gets replaced with a programmer Knut' who is so careful to observe that it is necessary in `fnord` to call a certain method `baz` after `bar`. `baz` throws a `Baz` requiring special handling by the caller. Good luck for Knut', there is no conflict with the client code, as `Baz` is derived from `Mother`, hence the interface of `fnord` does not have to be changed:

```
/** @throws Mother on error */
void fnord() throws Mother
{
   foo();
   bar();
   baz();
}
```

What Knut' did not realize is that even though the new `fnord` is syntactically compatible with the calling code, a semantical incompatibility has been created: If a `Baz` occurs, rather than getting the aforementioned special handling, a `Bug` is raised, which is the totally wrong way to handle this exception. Knut and Heinz were familiar with each other, but Knut' did not know that the hidden change of the contract for `fnord` had to be communicated to Heinz by, say, e-mail.

To make my point "very clear", let us fabulate a little further: In testing of software, the code pieces getting the least test coverage are usually the catch handlers, so it is not unlikely to assume that the incorrect exception handling is not detected in the test phase and the buggy software is released. And then in the production system, `Baz` really occurs, business transactions are rolled back for reasons inexplicable to the customers; and before the buggy exception handler is detected, fixed, and the fixed software is tested and rolled out, customer dissatisfaction has reached a level where the company running this software goes bankrupt.

If it is not for this cataclysm, at least personal communication and additional bug finding and fixing could have been avoided if `fnord` had declared its exceptions precisely:

```
/**
@throws Foo in case of fooness
@throws Bar if something barish happens
 */
void fnord() throws Foo, Bar
{
   foo();
   bar();
}
```

in the first version, and

```
/**
@throws Foo in case of fooness
@throws Bar if something barish happens
@throws Baz on bazy failure
 */
void fnord() throws Foo, Bar, Baz
{
   foo();
   bar();
   baz();
}
```

in the second version.

Example 5.2.4. `fnord` could also change in that its new version does not throw `Bar` anymore. Again, the calling code would remain syntactically correct, but would now contain a superfluous catch handler.

Example 5.2.5. Another wide-spread bad practice to be mentioned in this connection is to even declare `java.lang.Exception`. As an example, let us consider a new `throws` clause for `foo` of Ex. 5.2.2 (the programmer did not want to restrict the throws clause to `IOException` since he was irritated by my silly Ex. 5.2.2 and not sure if perhaps a `ClassNotFoundException` could occur, too :-)

```
final void foo() throws Exception
{
   new File("data").createNewFile();
}
```

This is another example for loss of information. We *know* that only `IOException`s can occur, but this knowledge is hidden from the compiler. The calling method inevitably has to declare `Exception` or provide a catch handler `catch(Exception)` (or `catch(Throwable)`) where exceptions are handled which the client most probably did not want to catch, e.g. `RuntimeException`s, or at least did not want to be forced to do so by the careless author of `foo`.

But the easiest thing to do for the caller is to declare `Exception`, too, and in fact I have seen entire modules (for the curious: in a commercial application server) in which almost every method declared `Exception` just because some poorly designed basic methods in these modules unnecessarily declared `Exception`. The `throws` clauses thus contained no real information. An approach like this essentially means to resign on error handling.

A reason for declaring exception sets larger than necessary (but certainly not as large as the set of all exceptions, as in the bad example above) can be the primacy of interface stability over precision of its description. One leaves room for future implementations throwing more exceptions than the current implementation without having to change the interface and the client code. Sometimes it is desired that client code takes new exceptions into account, sometimes this is not necessary. A general judgement is impossible, but it seems that the cases are rare where it really makes sense to hide the exact set of exceptions.

Example 5.2.6. Have you ever seen `javax.naming.InitialContext.lookup()` seen throwing any other checked exception than `javax.naming.NameNotFoundException`? But it declares the superclass `NamingException` of `NameNotFoundException`. Assume for a moment that the implementation really cannot throw anything else (which is probably wrong for at runtime `InitialContext` internally uses the `Context` implementation of some JNDI provider). Nevertheless the thought that if some future implementation is required to react on JNDI error conditions in a way where throwing a `NameNotFoundException` would not be appropriate might, under this assumption, be the reason for a seemingly to wide `throws` clause, provided that not breaking client code has priority over fine-grained exception handling (where the client would be urged to handle `NameNotFoundException` and other exceptions individually).

5.3 Inheritance

Although the Java Language Specification [1], 8.4.4, 11.2, leaves no doubt that the `throws` clause of an overriding method may be narrower than that of the overridden method, it is a common mistake in Java code that overriding methods

declare more exceptions than they really throw because the `throws` clause is copied from the overridden method. A too imprecise `throws` clause determined in this way does not yield the slightest benefit, but can have bad consequences, as explained in § 5.2.

Example 5.3.1. We want to administer permissions in the form of persistent access control lists in an object-oriented database. For this purpose we write, among others, a class modeling an ACL. As this class is to be used by an application server security realm, it has to implement the interface `java.security.acl.ACL`:

```
import java.security.Principal;
import java.security.acl.*;

public final class CoolAcl implements ACL
{
   private Collection aclEntries;

   public void addEntry(Principal p, AclEntry e)
      throws NotOwnerException
   {
      aclEntries.add(e);
      // persistence handled automatically :-)
   }

   // other methods
   ...
}
```

`ACL.addEntry` declares `NotOwnerException`, but despite the fact that our implementation `CoolACL.addEntry` declares it, it does not throw this exception actually. Our naïve administration tool does not care about the `java.security` API, it just works with `CoolAcl` objects. And the programmers of this tool are fully justified to get mad on the programmer of `CoolAcl` for carelessly copying the throws clause from `ACL.addEntry`, because they have to write pointless exception handling code for the `NotOwnerException` that is actually never thrown in `CoolAcl.addEntry`.

5.4 Unnecessary Throwing

Do not use exceptions to indicate outcomes that are special but not erroneous. This guideline results from the rule to avoid using exceptions for flow control.

Example 5.4.1. A method for finding all customers with a given name:

```
/**
@throws NoSuchObject if no customer
        with the specified name exists
*/
Collection findCustomers(String name) throws NoSuchObject
```

It is not an error if there is no customer with a certain name, say “Jakop”. This bad method interface would force the caller to write a catch handler for handling the very normal and predictable situation that there is no such customer:

```
int countCustomers(String name){
   int result;
   try{
      result = findCustomers(name).size();
   }catch(NoSuchObject e){
```

```
      result = 0;
   }
   return result;
}
```

whereas the natural client code, possible if `findCustomers` returned an empty set if no customer was found, would look like this:

```
int countCustomers(String name){
   return findCustomers(name).size();
}
```

5.5 Unchecked Exceptions

In 3.4 the compiler's view on unchecked exceptions was explained. A good characterization of their purpose at runtime is the following: *Unchecked exceptions correspond to errors which require a change in the software or its informational or physical runtime environment for correct functioning.*

Throw unchecked exceptions, which due to 5.1 and 3.4 means objects being `instanceof java.lang.RuntimeException`, for errors the caller cannot be expected to handle in a way that the requested function works afterwards, e.g. for logical programming errors (of the callee as well as of the caller) and system-level errors.

Throwing unchecked exceptions is potentially dangerous since the client is not forced to catch these situations, so information can get lost. Another danger is that while the declared interface is unchanged the runtime behavior of the server can change, affecting the client in bad ways. Because of these dangers the usage of unchecked exceptions should be restricted to the really bad ("unrecoverable") cases defined above.

Example 5.5.1. Server method `foo` in version 1.0 of the server used to throw a `NullPointerException` if a certain service S was not started. Although this `NullPointerException` was not declared (cf. § 5.5.1), client programmers got to know this behavior and wrote a catch block for this `NullPointerException` so they could start S in this catch block and then call `foo` again.

Version 1.1 of the server is devoted to bug fixing. It is required that the old clients have to continue to function with server version 1.1. But the programmer of `foo` in the meantime has read § 5.5.3 of this paper and has been convinced to throw an unchecked `PreconditionViolation` (not derived from `NullPointerException`) if S is not started. The interface declaration of `foo` is not changed since it does not have to. The clients are syntactically compatible with the new server version and nobody suspects anything because unfortunately all tests were carried out with a running S. Now when the new server version goes into production, it turns out that all the client applications (hundreds of them, in world-wide locations, if you like) do not work anymore since they fail to start S automatically; the clients even crash due to a `RuntimeException` they do not handle. (And yes, believe me, stuff like that happens in reality.)

On the other hand, a lot of essentially redundant exception handling code can be saved by throwing unchecked exceptions in certain common situations.

Example 5.5.2. Imagine there were no unchecked exceptions. In server-side software, most any method can fall victim to system-level errors (broken connection, configuration errors, transaction timeout, and the like), so it would have to declare exceptions covering these cases. For the sake of simplicity let us assume that a (necessarily checked) exception class `Internal` is reserved for these errors (if there are multiple exception classes for this, the situation gets even worse). Now if we know that every method of our system throws `Internal`, it looks somewhat redundant to declare this fact for each individual method. Throwing

`Internal` should rather be viewed as an aspect of our software system, but this insight does not help us much further, for aspects of this type are not yet supported by the Java language. Clients either let `Internal` pass by, forcing them to declare `Internal` too, or have to catch it, forcing them to handle it, which they cannot in a sensible way (What is the client of the server system supposed to do if the server configuration contains a wrong database URL?). In real-life examples (some of them written by me) this turns out to result in ubiquitous pointless `catch(Internal)`-handlers.

Using unchecked exceptions for this type of errors has the following advantages:

- business interfaces are not polluted with technical details (again a matter of maintainability)
- essentially redundant exception declarations are avoided
- a lot of essentially useless error handling code can be avoided, which is a benefit in view of clarity, comprehensibility of the code, and sometimes also performance

The approach of this section has been adopted in the EJB API: `javax.ejb.EJBException`, in JDO: `javax.jdo.JDOException`, and in the JDK CORBA implementation: `org.omg.CORBA.SystemException`. In the author's humble opinion one should have made `java.rmi.RemoteException` unchecked too, because then arbitrary Java interfaces could be implemented by RMI objects, and the information whether the implementation is executed locally or remotely could be hidden from the client (which is, of course, exactly what the RMI designers wanted to prevent, in view of the inherent unsafety of remote connections).

5.5.1 Declaring Unchecked Exceptions

Unchecked exceptions should not be declared in `throws` clauses. First, this would be redundant code which is always bad for legibility and clarity. Second, it might lead the reader to think the exception was a checked one. However, it is a good practice to document unchecked exceptions being *raised* in a method in the method's Javadoc documentation.

5.5.2 Assertions

Program code is occasionally able to detect bugs by itself. The most common example is that an exception is caught that could not arise if the code was correct (cf. 7.6). Or self-checks, so-called assertions, could be built purposely into the code, in most cases for ensuring invariants, pre-, and postconditions.

Example 5.5.3. Class `Person` has an attribute `name`. The invariance condition, ensured by the implementation (if it is correct), is that `name` is never nil. Then a method `foo` of `Person` could check this:

```
void foo(){
   ...
   if (name == null){
      // bug detected
      ...
   }
   ...
}
```

The question whether or where and how code should try to detect bugs is not to be discussed here. The subject of this section is what should be done *if* a bug is detected, in whichever way, at runtime. (In the example above: Given that there is such an `if` statement, what should stand for the ellipsis after `// bug detected`?)

The answer is, you guessed it, to throw an unchecked exception in these cases. A checked exception would have to be declared, but bugs should be fixed rather than being perpetuated in interfaces, forcing the caller to write code for conditions that must not occur. Throwing is better than only logging the incident because logging without throwing can leave the caller in the illusion that the called operation had succeeded, thus creating wrong assumptions and possibly unnecessarily delaying the correction of the bug. Cf. also 7.6.

5.5.3 Preconditions

A special case of assertions is that of verifying preconditions. It is of major significance due to its frequent occurrence, and deserves special interest since precondition violations usually indicate bugs in the caller rather than the callee.

The "Design by Contract" principle [5] enforces formal declaration of preconditions, and the possibility of violations of these preconditions should not be reflected in business interfaces. The idea is that violation of a precondition is always a bug, hence these errors can occur only until the software is debugged, and bugs, logical programming errors, certainly are not the subject business interfaces should be concerned with. Therefore and according to 5.5.2 an unchecked exception should be thrown in such situations; my proposal is to use a custom exception class `PreconditionViolation extends RuntimeException` for this and only this purpose.

It is a good practice to explicitly verify that calling code obeys preconditions if this calling code is not being developed together with the called code, because giving explicit error messages pointing exactly to the error eases debugging. If the calling code is ours, we should rather make sure that it is correct than polluting our code with code pieces dealing with our own bugs.

The principle behind this reasoning is that we can and must develop correct code, but cannot ensure correctness of foreign code. Even if the foreign code is not buggy, it can be in a future version.

5.5.4 Unimplemented Functionality

During development of a software system, it frequently occurs that implementations of methods or parts of it are missing, and this is hidden from the compiler by providing dummy (possibly NOP) implementations. What one often sees in this situations is code like this:

Example 5.5.4. Rudimentary implementation:

```
package org.acme.project.banking;

import java.math.BigDecimal;

public class BankImp implements Bank
{
   ...

   public void withdraw(Account account, BigDecimal amount){
      // to be done
      // compiler is happy with this NOP implementation
   }
}
```

The method `withdraw` has to be there since otherwise `BankImp` would be syntactically incorrect as the interface `Bank` declares this operation. The problem with this preliminary solution is that the fact that the real implementation is missing is not only invisible to the compiler, it is also hidden from the caller, and this again is an unnecessary information leakage. The problem might manifest itself as follows: Tests involving `BankImp.withdraw` will probably fail. Then perhaps someone will try to find the cause of this failure and spend a lot of time checking the `Account` object, the persistence layer or so, until they discover the unimplemented `withdraw` method.

This waste of time (which I have seen to occur in real projects) is absolutely not necessary. The correct solution for marking unfinished implementations is to define a special unchecked exception

```
package org.acme.error;

public class NotImplemented extends RuntimeException
{}
```

and throw it if it is attempted to execute unimplemented code branches.

Example 5.5.5. Throwing in unimplemented code branches:

```
public void withdraw(Account account, BigDecimal amount){
   throw new org.acme.error.NotImplemented();
}
```

Now the caller gets an exception containing the necessary and sufficient information, viz. that `BankImp.withdraw` cannot be expected to work for it is not implemented yet.

The same approach can be taken of course for code pieces inside a method:

```
public void withdraw(Account account, BigDecimal amount){
   boolean isEasy = ...;

   if (isEasy){
      // implementation
      ...
   }else{
      //TODO
      throw new org.acme.error.NotImplemented();
   }
}
```

(Can you guess why the `throw` statement refers to the exception class using its fully qualified class name? Obviously I did not want to import the exception class. The reason is that one knows that `BankImp` will not use `NotImplemented` when it is finished, but then the import statement would be likely to be overseen and remain as a superfluous import statement, maybe even indicating a fictitious package dependency `org.acme.project.banking` → `org.acme.error`.)

Another neat effect of this approach is that the compiler will show us the implementation gaps if we try to compile the software with `NotImplemented` removed from the codebase.

5.6 Checked Exceptions

5.6.1 Business Errors

Throw checked exceptions in error situations that are predictable in the context of the business logic.

Example 5.6.1. The class `java.sql.PreparedStatement` has a method `executeUpdate` returning the number of affected rows in the relational database and throwing `SQLException` if something goes wrong. It is adequate to throw an exception here because all kinds of errors can occur if the database is accessed (broken connection, wrong SQL, violation of table constraints in the database, etc.). Aside from the principle that errors should be treated as errors and not be mixed with the "normal" logic, returning the error information is not possible since the return value is already defined to be something else. (An old-fashioned programmer would perhaps use negative return values for error codes, but this would mean to give the return value a very complex meaning, reduce the error information (one would not encode a long error message from the database and a stack trace in an `int`), and moreover the contract for this method would be defined in a lot of documentation instead of something the compiler can evaluate.) It would also not be appropriate to throw an *unchecked* exception because the caller has to react to the `SQLException`, e.g. by closing the database connection. Another reason for not throwing an unchecked exception is that the caller cannot ensure that the `executeUpdate` succeeds by something like checking certain conditions prior to the call, so the error is not necessarily a bug of the caller.

An example of a method throwing in a case where it should not throw at all has been given in 5.4.1.

5.6.2 Granularity

Throw exceptions of special types if and only if special exception handling is required.

Error conditions that require similar error handling should be indicated by the same exception class. The simple reason is that otherwise the client would be forced to write multiple catch handlers doing the same.

Example 5.6.2. In Ex. 5.6.1 two possible reasons for an `SQLException` were mentioned: syntactically wrong SQL code and violation of database constraints (e.g. non-nullity of primary key). Would it be correct to declare a `WrongSQL` and a `ConstraintViolation` exception for these cases, respectively? No, because the way the client reacts to the error is most probably absolutely the same: close the connection, log the exception or raise a new one, and hope that somebody fixes the software or the runtime configuration. Neither is the SQL code going to be altered at runtime for a new trial, nor would one expect a piece of code that tried to insert a nil primary key to learn from the exception to create a proper primary key — at runtime. The error information is completely encoded in the exception object's state (as opposed to its type).

Example 5.6.3. A method for creating a new customer record in the database.

```
/**
@throws Duplicate if this customer already exists
@throws BadInput if <code>x</code> is corrupt
*/
void newCustomer(CustomerData x) throws Duplicate, BadInput;
```

It is adequate here to declare different exception classes for the case that a customer with the ID contained in `x` already exists in the database and the case that the customer data is incorrect (illegal characters in the e-mail address or stuff like that) since these situations are likely to require a different handling (e.g. tell the user that he is already registered in the first case, and prompt for better input in the second). The client really needs 2 catch handlers.

If the designer of `newCustomer` had chosen to throw only one exception type, encoding the error type in the exception's state rather than its type:

```
/**
@throws CustomerException on error.
   Let <code>e</code> be the exception,
   <code>c = e.errorCode()</code>. Then
   <ul>
      <li><code>c == CustomerException.DUPLICATE</code>
         iff this customer already exists
      <li><code>c == CustomerException.BAD_INPUT</code>
         iff <code>x</code> is corrupt
   </ul>
*/
void newCustomer(CustomerData x) throws CustomerException;
```

then part of the interface description would have been moved from Java code to documentation. The client code would have only one catch handler with branches according to the exception contents. A different error cause would be reflected by a new error code instead of a new exception type, so the client code would not be forced by the compiler to handle the new error type.

5.7 Meaningful Exceptions

When an exception is thrown, some object (maybe the JVM) is going to catch it. And it can be expected that some (software or human) object will try to make sense of this exception, will probably try to fix or circumvent the error having caused the exception. Therefore an exception should contain sufficient information for supporting error investigation.

Example 5.7.1. An instance method that should only be invoked on an object if this object is in a certain internal state:

```
void method foo(){
   if (state != READY4FOO) throw new IllegalState();
   ...
}
```

The poor programmer examining an `IllegalState` exception originating here is likely to say: "So the object is not in state `READY4FOO`. This is an outrage, I mean, a logical error. How is this possible? If only I knew which actual state the object had when `foo()` was called...". If the author of `foo` had not been so lazy, he would have thrown `new IllegalState(state)` (it is assumed here that `IllegalState` has such a constructor), including the offending state in the exception, and this question would not have to be researched, possibly involving tedious debugging and reproduction of the error, maybe even costly setup of a test system.

Example 5.7.2. In a commercial application server I once saw a class loader implementation with the method

```
public Class loadClass(String name) throws ClassNotFoundException
{
   try{
      // find the class file in the file system using java.io
      ...
   }catch(IOException e){
      throw new ClassNotFoundException(name);
   }
}
```

This is in conformance with the class loader specification, but the user of the application server would be much happier if the information contained in the

original `IOException` was not sent to the bit bucket. When deployment of an enterprise application on an application server fails due to a `ClassNotFoundException`, it can be very helpful for the deployer to know whether

- the file does not exist
- the file exists, but access is denied
- the file is corrupt
- or what else is wrong with it

A better implementation would include the detail message or, even more helpful, the stack trace of `e` in the new `ClassNotFoundException`. In JDK version $\geq$ 1.4 one could simply

```
throw new ClassNotFoundException(name, e);
```

Example 5.7.3. A common special case is that of complaints about illegal arguments:

```
void foo(Customer x) throws BadInput
{
   if (!isExistent(x))
      throw new BadInput("customer does not exist");
   ...
}
```

Does anyone have any doubt that it would be much more useful for debugging if the exception contained at least the ID (customer name, customer number, or something like that) of the bad argument? Why not

```
if (!isExistent(x))
   throw new BadInput("customer " + x.id() + " does not exist");
```

?

5.8 `NullPointerExceptions` Are Bugs

Our software should not throw `NullPointerExceptions`. Every `NullPointerException` arising in our software deserves a bug report.

The simple reason is that `NullPointerExceptions` thrown by the JVM if a nil pointer is to be dereferenced contain very little information, much less than is actually available when an offending `null` is detected, hence this rule follows from 5.7. The only information is the stack trace (if available), containing the number of the line of code where the `NullPointerException` was raised (if available). If this information is not available, the client getting the `NullPointerException` has a very hard time figuring out what went wrong. But even if it is available, debugging might also be more difficult than it would have to be.

Example 5.8.1. Convincing example:

```
public void createUser(UserData x){
   Collection s = dataStore.findUsers(x.id());
   ...
}
```

Assume we get a `NullPointerException`, that the line number is contained, and it is the first line of code of `createUser()` above. We only know that one of `dataStore` or `x` must have been nil. We do not know which of them. And even if we knew, e.g., that `dataStore` was nil, we could be far from knowing the cause of this error. The programmer of `createUser()` probably would know, but this implementation keeps his knowledge secret.

An acceptable implementation of this method would be

```
/**
@pre x != null
*/
public void createUser(UserData x){
   if (x == null)
      throw new PreconditionViolation("x is nil in createUser");
   if (dataStore == null)
      // I know the reason and tell the caller:
      throw
         new Internal(
            "createUser: file datastore.properties missing"
         );
   Collection s = dataStore.findUsers(x.id());
   ...
}
```

Checking `dataStore` for being nil can of course be omitted if the implementation guarantees that `dataStore` cannot be nil.

The case of method arguments that must not be nil deserves some special attention. The rule is to check nullity of these arguments if the caller of the method is not under control of the server, because it is written by unknown organizations in the future, belongs to another module, or the like. And the reason for performing this check if there is any doubt on whether the client respects the preconditions is of course the prevention of `NullPointerException`s not optimal for debugging, as explained above.

5.9 Reusing Exception Objects

Performance-conscious programmers avoid unnecessary object creation. Let us investigate this with respect to throwing.

Example 5.9.1. How do you like this class?

```
class Foo
{
   private final static FooException EXC = new FooException();

   void bar(){
      ...
      if (...) throw EXC;
   }
}
```

(Even this technique is not my deliberate invention; it has been seen in an enterprise middleware system worth millions of dollars.) No exception object has to be created when the exception is raised. Well, the problem is of course that the stack trace of `EXC` is filled in at creation time and not when `EXC` is thrown, so the caller gets a wrong stack trace. But this can be fixed:

```
class Foo
{
```

```
   private final static FooException EXC = new FooException();

   void bar(){
      ...
      if (...){
         EXC.fillInStackTrace();
         throw EXC;
      }
   }
}
```

And it is also not a problem to fill in other error information into `EXC` if the respective attribute accessor methods exist in `FooException`.

Now have we discovered an ingenious object-creation-saving throw mechanism? I don't think so.

Assume that multiple threads operate with `Foo` objects. Then the construction above is impossible since one thread could modify `EXC` while in another thread it is on its way from the `throw` statement in `bar` to its handling. If one tries to fix this by making `EXC` an instance field rather than a class field, `Foo` objects still cannot be made thread-safe.

Assume now that we know that `Foo` objects are not used concurrently. I still have 3 arguments against the reuse of the exception object. First: memory is wasted for the permanent exception objects which are possibly never used. Second: a private member is exposed to other classes; `Foo` does not have any control over how the caller of `bar` modifies the state of `Foo`'s exception object. Third, and in my view most important, if you compare the implementation

```
class Foo
{
   private final FooException myExc = new FooException();

   void bar(){
      ...
      if (...){
         myExc.fillInStackTrace();
         throw myExc;
      }
   }
}
```

to the "normal" one

```
class Foo
{
   void bar(){
      ...
      if (...){
         throw new FooException();
      }
   }
}
```

which do you consider more maintainable, better readable, clearer, less error-prone?

To conclude this discussion, it does not make sense to avoid the creation of a simple object in an *exceptional* situation. If it is really a requirement that certain error situations be handled some milliseconds faster, then most probably exceptions are misused for business flow control, i.e. the corresponding error is not really an error, cf. 3.3.

5.10 `finally` Handlers

As I have witnessed multiple discussions of the question which exception is thrown if a `finally` clause, which is entered due to an exception, throws another exception, this question shall be answered here once and for all.

Example 5.10.1. Let `FooException` and `BarException` be checked exception classes, neither one a subclass of the other. Which exception does the caller of `foo` get if `bar` throws a `BarException`: `BarException` or `FooException` (or even both)?

```
void foo() throws FooException, BarException
{
   try{ bar(); }
   finally{ throw FooException(); }
}
```

The answer is in [1], 14.19.2: The exception thrown in the `finally` clause overrides exceptions from `try` or `catch` blocks executed before the `finally` clause.

Example 5.10.2. In 5.10.1, the caller gets a `FooException`, even if `foo` would be

```
void foo() throws FooException, BarException
{
   try{ bar(); }
   catch(BarException e){ throw e; }
   finally{ throw FooException(); }
}
```

The `BarException` is discarded.

And this regulation is only natural. Otherwise throwing in `finally` clauses would be pretty pointless.

5.11 Roll-Back

Although I have never seen this stated as an explicit convention, developers often assume without further asking that an object that has thrown a checked exception is in the same state as prior to the abruptly completed method invocation. It seems therefore advisable for implementors to ensure this aspect of transactionality where possible, and to document cases where throwing of checked exceptions alters state.

When an object throws an unchecked exception, however, one had better not rely on any unproven assumptions with respect to the object's state, for the implementor of the object's class has not necessarily been aware of the possibility of this unchecked exception at implementation time.

An exception to this last recommendation is for example the Java Data Objects API where only `RuntimeException`s are thrown, even in very predictable error situations, and the `PersistenceManager` may be perfectly healthy after it has thrown, e.g., a `JDOUserException`.

Chapter 6

Error Reporting

A fundamental question arising frequently in object-oriented software is: Should the exceptions know the log service or vice versa? Let me explain this and formulate it in a more "scientific" way:

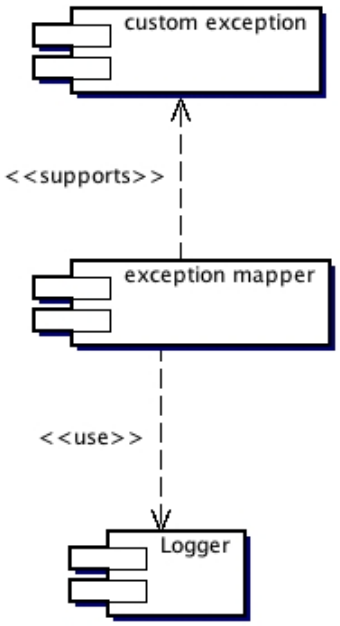
Example 6.0.1. `org.acme.project.error` contains generic exception classes for our project, `org.acme.project.log` is the logging component, a wrapper around a third party logging library like `java.util.logging`. Should `org.acme.project.error` depend on `org.acme.project.log` or vice versa?

This problem is solvable. In this chapter it is shown that in general exceptions should know the logger and *not* vice versa, essentially because there is only one logger but many exception classes. A possible exception from this principle is addressed in 6.3.

6.1 Generic Design

The term "logging" is used here as a shorthand for "error reporting"; it means exposing error information to humans (in rare cases also to error-examining software), for which there are several ways: Displaying a dialog window with an error message, writing something on the console, writing into a file ("log file"), sending it to another process via, say, JMS and hoping that the other process knows what to do, recording data in a database for future examination etc. Regardless of what the logging sink is, we will assume that our software uses some logging facility that is not specific for our project, be it `System.out.println`, Log4J [2], the JDK 1.4 Logging API (`java.util.logging`), or something else, that is to say third-party logging software. This logging facility knows what do to with character strings, maybe also has a notion of severity or log record categories, maybe even knows how to log exceptions or other objects. For the sake of simplicity and brevity, we assume that there exists an interface `Logger` with a method `log(String)`. We also assume that every object has easy access to a `Logger` instance (which holds obviously, e.g., for `System.out`, a Log4J `Category`, and any other reasonable logging service).

Moreover, we assume that our software uses proprietary custom exception classes (cf. 4.1). This already implies that we cannot simply use the `Logger` interface in a straightforward way (à la `Logger.log(myException)`) to log our exceptions because we said that `Logger` is independent from our project, so `Logger` does not know what our exceptions contain and what the contents mean, at least not for future versions where exceptions might be richer. Therefore there has to be a software layer, let's call it temporarily "exception mapper", knowing and using `Logger` and being under the control of our project.



Now we have to solve the following design problem, at least in a sufficiently complex software system: We certainly want to have as few variants of the exception mapper as possible. But potentially every exception class has to be logged in a specific way, which implies that our exceptions to some extent have to know how they are to be logged. Moreover, this way of logging should not be determined by the exception alone, since different software components may have different logging requirements for the same exception class (e.g. a different severity or priority might be assigned to the corresponding log record). In the worst case the number of exception mappers we would have to write is the number of exception classes times the number of logging policies.

A decoupling is achieved in the following (standard) way: we define a rather fine-grained interface `LogAdapter`. This and only this interface is then used by our exception classes for logging. Our exception classes use specific code for filling their data into the `LogAdapter`. This code may be contained within the exception class or, sometimes wiser, in helper classes used by the exceptions — as you like.

A `LogAdapter` definition could go along the following lines:

```
interface LogAdapter
{
   log(
      short errorCode,
      String[] messages,
      Throwable[] nestedExceptions,
      Object[] data
   );
   // maybe additional operations allowing exceptions to set data
}
```

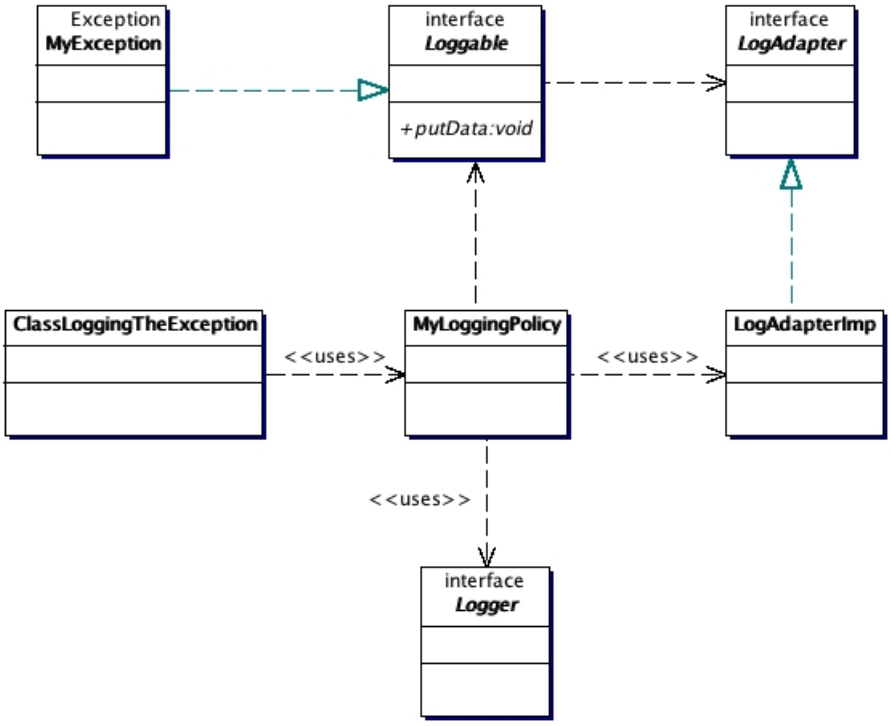
Our exceptions are asked to deliver their data to the `LogAdapter` coming as a visitor; for this purpose they have to implement an interface like

```
interface Loggable
{
   void putData(LogAdapter a);
}
```

The `LogAdapter` implementation, let's call it `LogAdapterImp`, provided by the object x that wants the exception to be logged then takes care of extracting the information from the nested exceptions (according to their type: calling `putData(this)` if they are `Loggable`, reading the array of nested exceptions in a `JDOException` etc.). This is still independent of the logging policy. The logging policy adopted by x now knows `LogAdapterImp` (or at least an additional interface of `LogAdapterImp` which the exception class does not "know") (`errorData()` in the example below), which is independent of the exception class, and can get the information from it and put it into the `Logger`:

```
class MyLoggingPolicy
{
   LogAdapterImp m;

   void log(Loggable e, String threadID, String location,..){
      e.putData(m);
      Logger.log(
         processToString(m.errorData(), threadId, location,..)
      );
   }
}
```



I admit that this construction looks complex, but this is due to the flexibility requirements that were assumed. We will see below how this simplifies if the requirements are weakened.

6.2 Exceptions Responsible

If there are no concurrent logging policies, for example, the exception mapper depends solely on the exception class.

Example 6.2.1. The mapper's functionality could, e.g., be confined to producing a string from the exception. This is not an unrealistic assumption since logging into a text file or to the console means nothing but presenting the log data as a string (and then writing it). In this case the mapper is even logically independent of the logger, so it would be most unnatural put this functionality into the logging component. An implementation in this case could look like this:

```
class MyException extends BaseException
{
   Object myAdditionalData;

   protected String convertToString(){
      return
            super.convertToString()
         +  '\n'
         +  myAdditionalData.toString();
   }
}
```

The code for logging an exception can be contained in the class that logs the exception:

```
myLogger.log(exc.convertToString());
```

or can be contained in the exception class itself:

```
   void log(Logger l){
      l.log(convertToString());
   }
```

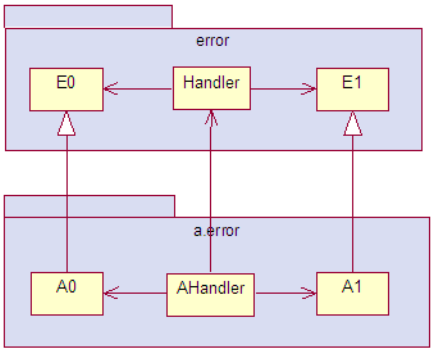
(`Logger` could of course also be a wrapper around the actual third-party logger.) (If requirements for `toString()` and `convertToString()` are compatible, one could use `toString()` instead of `convertToString()`). The exception mapper is part of the exception class here.

Since the case of exceptions determining their logging completely is the most common one, it shall be explored in more detail. As the basic checked and unchecked exception classes (4.3) tend to be very similar, it often turns out that the mapping (logging) code in them is identical, so should be factored out. This code cannot be contained in the logging component since, as stated above, exceptions depend on the log service, but the log service must not depend on the exceptions. The solution is to invent a static class (cf. Def. 2.1) H containing utility methods used by multiple exception classes. (If H is to implement some common interface, one uses a singleton instead).

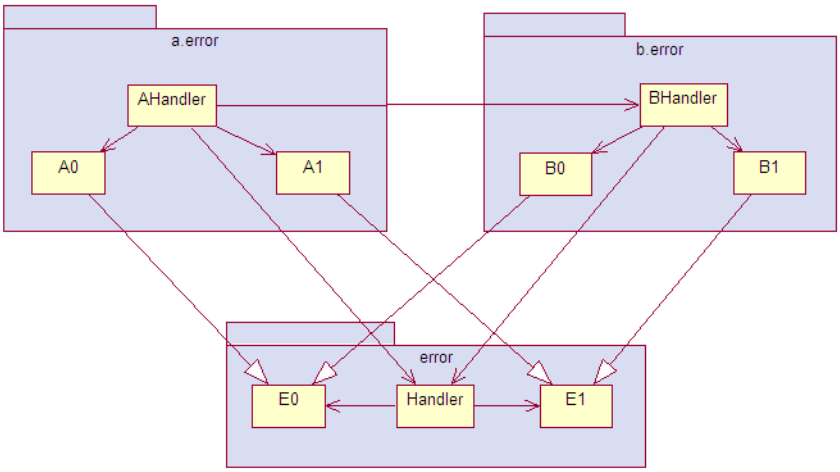
The next step in complexity is that our software consists of more or less autonomous components or modules coming with own sets of exception classes. Nothing is better suited to show that our construction is applicable in this case than a little

Example 6.2.2. Package `org.acme.project.error` contains the mother of all unchecked exceptions `E0`, the mother of all checked exceptions `E1` and a static class `Handler` containing the log (mapping) functionality used for both `E0` and `E1`. `org.acme.project.error` depends on the logging component `org.acme.project.log`.

A module A corresponding to package `org.acme.project.a` has a subpackage `org.acme.project.a.error` with custom exception classes `A0` and `A1` derived from `E0` and `E1`, respectively. Thus `org.acme.project.a.error` depends on `org.acme.project.error`, which is not a problem. The logging code for `A0` and `A1` (maybe a static class `AHandler`) can use `org.acme.project.error.Handler` without introducing unwanted package dependencies.



Next step: If there is another module B that A depends upon, and B defines exceptions similar to A, then it is likely that A has to handle the exceptions defined by B. Since A depends on B anyway, it poses no problem if the exception logging code in A uses `org.acme.project.b.error.BHandler`.



An interesting and common complication in software layering is that B uses a third module C that should be hidden from A; exceptions defined in and thrown by C fly from B to A; A is responsible for logging them but should not know them. Sounds strange? Not at all:

Example 6.2.3. A is a façade (perhaps a session EJB) having to log every unchecked exception occurring in its business methods. B is a persistence layer used by A, C is `javax.jdo` (with a JDO implementation). A must not depend on `javax.jdo`. C throws my favorite example `javax.jdo.JDOException` and subclasses thereof. `JDOException` is unchecked and requires a special logic for logging because it contains an array of nested exceptions. This logging code belongs to B since otherwise A would become dependent on C. The common solution complying to the guidelines formulated in this paper (6.4, 7.2.2) is that B catches every `JDOException`, logs its information, and then throws another exception that A may depend upon. Although this is clean design, it has flaws, namely that a single event causes multiple logging activities and raising multiple exceptions. Furthermore, it could be required that B should not take decisions on when and what to log.

The solution in this case is of course that A uses the mapper class `BHandler` of B for `JDOExceptions`. And how does A know when to use `BHandler` instead of its own `AHandler` if it does not know `JDOExceptions`? It does not need to. Letting B tell A whether a given exception is a `JDOException` would be a

breach of encapsulation because A would then become semantically dependent on JDO, after all. But it is not a breach of encapsulation if B tells A whether a given exception is not A's business since it is specific to B, and this is sufficient. Let's see how this works:

```
public class BHandler
{
   /**
   @return whether I consumed the exception <code>e</code>
   */
   public static boolean log(Exception e, Logger l){
      boolean isJDO = e instanceof JDOException;
      if (isJDO){
         // log it
         ...
      }
      return isJDO;
   }
}
```

and a catch handler in A could go like this:

```
catch(RuntimeException e){
   if (!BHandler.log(e, myLogger)){
      AHandler.log(e, myLogger);
   }
   throw new AO(...);
}
```

If the catch handler may rethrow the `RuntimeException`, it gets even simpler:

```
public class BHandler
{
   public static void error(Exception e, Logger l){
      if (e instanceof JDOException){
         // log it
         ...
         throw e;
      }
   }
}
```

and the catch handler in A:

```
catch(RuntimeException e){
   BHandler.error(e, myLogger);
   AHandler.error(e, myLogger);
}
```

(The code above has been written like this only for demonstration purposes. A better design would encapsulate the usage of `BHandler` in `AHandler`, thus avoiding replication of the 2 lines in the catch handler above, and, more importantly, keeping classes other than `AHandler` independent of `BHandler`, thereby providing higher flexibility and maintainability.)

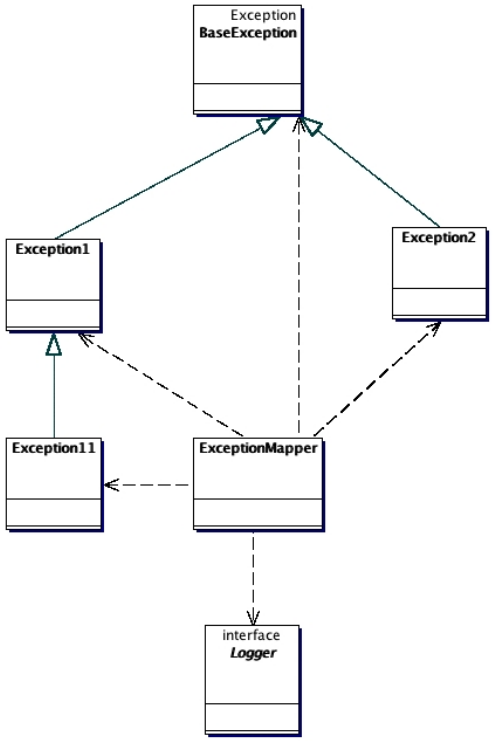
Remark 6.2.4. How do you like the idea of letting the exception class override `toString` such that the stack trace is included in the result string? If one decides to do so, one would probably call `printStackTrace` in `toString`. Be aware though that the standard implementation of `printStackTrace` (in `java.lang.Throwable`) calls `toString`, if you are not fond of `StackOverflowError`s.

6.3 Logging-Unaware Exceptions

Wouldn't it be nice if there was only one exception mapper? Exceptions would not have to care about logging. The problem with this approach is that the exception mapper would have to know the exception classes (unless it accesses the exceptions in an esoteric way via the Reflection API, which is neither recommendable (think of maintenance) nor usual), which has two serious consequences

- The exception classes must not belong to the components using the mapper, i.e. the components performing logging of exceptions, because otherwise a cyclic component dependency would result. This restriction is in fact seldom satisfied in component software.
- The mapper (as the logger) is then a fundamental component (in the sense that is far at the top of the dependency tree: many components (classes, packages) depend on it) and is therefore required to be pretty stable. But the approach discussed here brings instability into this of all project-specific components in that it has to be modified each time an exception class is changed in a substantial way (e.g. by a new attribute) or an essentially new exception class is introduced. (And of course an internal software component we are responsible for must not impose restrictions upon us with respect to adapting our exceptions to requirements to our software.)

If one, however, has the feeling that the forrest of custom exception classes is small and stable, and the exceptions do not belong to the components that perform logging, one might take these risks into account, achieving a simple design.



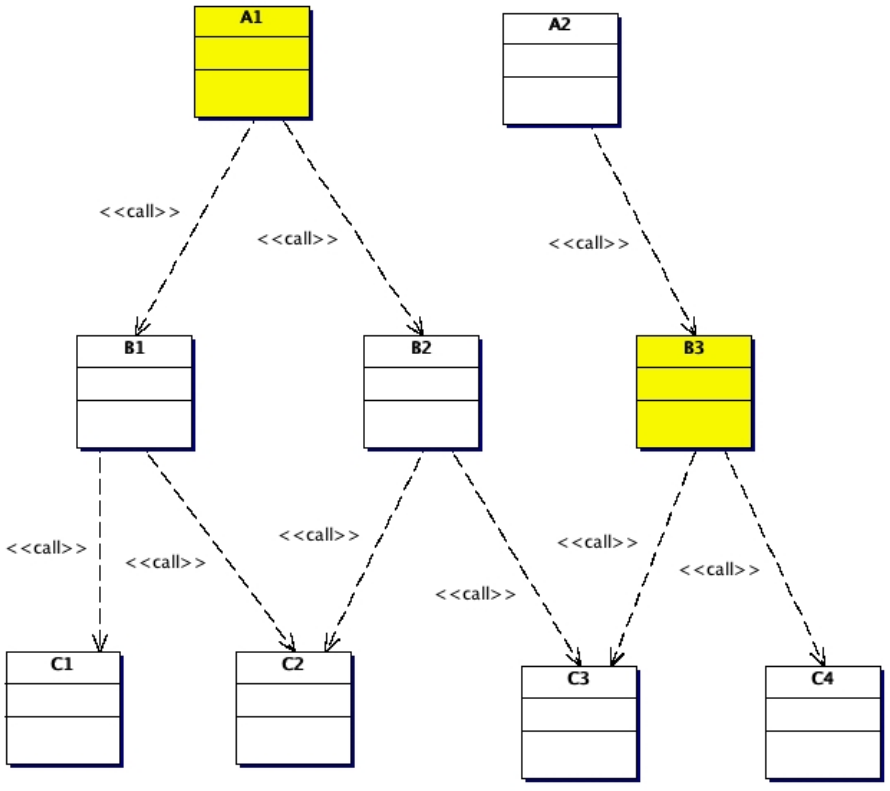
The exception mapper could then be viewed as a part of the project-specific wrapper around the third-party log service (for the discussion of dependencies above it does not matter if the mapper is used by the logger or directly by the components performing the logging.)

6.4 Where To Log

Logging of any information should be done at the highest possible level in the call stack. Reasons:

- If information can be logged at a low level and also be passed up the call stack, the question whether the information has already been logged poses itself to the caller. This problem does not arise if information that must be logged is logged if and only if it is not passed upwards.
- Logging is performed in a few central places. This helps to keep classes free from logging code. Why is this desirable?
  - A change in the logging strategy is more easily applied if the code to be changed is not scattered over a lot of classes.
  - And keeping classes independent of the logging component makes them more easily reusable.

Example 6.4.1. The following diagram shows a call graph. During execution, information (method invocation results as well as exceptions) travels upwards, from level C to level A. The yellow classes are the places where this information is not passed further.



The worst logging strategy would be to log the information where it originates, i.e. in the C classes: The logging functionality of level A and B becomes dependent on "apocryphic" knowledge about logging behavior of level C. The contract between class C2 and its clients in the B level with respect to logging could be expressed in the documentation of C2 ("when exception E is thrown, it has already been logged"), but in real life stuff like that tends to remain undocumented. If B2 has wrong assumptions on the logging policy of C2, which can always occur if C2 changes its policy, data gets logged twice or not at all.

All this cannot happen if one sticks to the rule to log information (that should be logged) if and only if it is not sent to the caller. In this case A1 and B3 are the only classes performing logging. All methods know whether they have to log or not because this depends on nothing but their own implementation. No need to document logging policies anymore; double-logging and omission of logging cannot happen anymore. If it is decided to include, say, a time stamp in the log records, only the classes A1 and B3 have to be changed.

Example 6.4.2. The deployment module M in a commercial application server had the policy to log exceptions where they were raised, à la

```
if (x == null){
   MyException e = new MyException("x is nil");
   e.log(myLogger);
   throw e;
}
```

Due to the rigorous rule about where to log, there were no doubts whether a caught exception (that could not be rethrown) should be logged or not (log it if and only if it has not been raised in M), at least if we suppose that it was easy to distinguish exceptions raised in M from others. But nevertheless, a lot of logging code (not to mention the code deciding whether an exception had its origin in M) could have been saved if the guideline of this section had been obeyed.

6.5 Robustness of Error Reporting

A common error is that the code responsible for logging errors encounters a new error while processing an exception, maybe reports the new error, but due to poor error handling fails to log the original exception. Error reporting is one of the most fundamental and indispensible functionalities of a serious software system, thus even under severe error conditions (e.g. `RuntimeException`s) some basic error reporting should be intact (and be it only `System.out.println`, if, say, there is no free space in the file system). The point of view that errors that cannot be handled need not be handled does not apply here. Even if we encounter an error that is considered fatal, such that it does not make sense to (pretend to) keep the system running anymore, this does not at all mean that this error should not be logged.

Example 6.5.1. In performance tests of a very complex software system (the largest Java project in Europe, I was told), one of the test cases failed with a `NoClassDefFoundError` that could be nicely seen in a log file. My investigation uncovered that this `NoClassDefFoundError` occurred when the application wanted to log some other exception. I was not able to find the cause for the `NoClassDefFoundError`, which has remained an eternal mystery. But actually I was not even particularly interested in the `NoClassDefFoundError`, for I assumed that that would be gone with the next deployment, which assumption later turned out to be correct. The really annoying thing was that the original exception did not get logged at all, though that would have been technically possible if the error reporting code had been more thoughtfully implemented, hence I had no clue for developing a workaround that would have circumvented the original exception and thereby also the `NoClassDefFoundError`. You may ask why I did not simply redeploy the software for getting rid of the `NoClassDefFoundError`, in view of my assumption mentioned above. Well, my experience at that time was that getting a release of the software installed and running used to take a team of several people more than a week.

6.6 Backtraces

If an error occurs and we want to find its cause, it is obviously most interesting to know the circumstances under which it occurred. If our code logs an error, we want to know where it occurred, i.e. the precise location in the code, which statement in which method of which class failed. Backtraces of Java exceptions contain almost exactly this information. Methods can be used in different scenarios (use cases). The backtrace can also tell us in which use case the error occurred.

Sometimes one does not want to find the cause for an exception. Maybe one knows it already, maybe it is not really an error. But certainly one wants to find it if the error is actually a bug (that has to be fixed).

The conclusion to be drawn here is that if an exception is logged and the exception points to an error that has to be fixed, praise your fate of being a Java programmer getting backtraces for free, and do not forget to include the backtrace in the log record. The backtrace (formerly known as stack trace) is usually the most valuable information an exception contains.

Chapter 7

Handling Exceptions

Exceptions are handled in `catch` blocks, which are also called catch handlers.

7.1 Save Error Information

Errors are to be avoided, *per definitionem*. This implies that if an error occurs, it is desired to take measures to prevent it from occurring again, as a rule. In most cases the best prevention measures can be chosen if one knows the cause of the error; sometimes this is even a necessary condition for preventing the error. In order to find the cause of an error, all information related to the error can be significant. Consequently it is important to record as much as possible information that might be relevant in an error situation. This information can consist of:

- exception contents, including nested exceptions, including stack traces
- values of input and configuration parameters
- thread name
- session ID
- caller ID
- transaction ID
- server node
- time of the event
- state of involved objects

Performance considerations should not play a role here since we are talking about *exceptional* situations. If exceptions are handled on a regular basis at runtime, this is an indication for misuse of exceptions for logical flow control.

One reason for excluding parts of the error information from being recorded can be that otherwise security requirements could be violated. This remark being made, this restriction will be ignored in the sequel of this paper.

As we are concerned with exception handling, we are also going to concentrate on the information contained in exceptions and exclude the environmental information listed above from the discussion.

Now the requirement deduced above reads: **Do not let exception contents get lost.** As simple as this sounds, it is the most common mistake in exception handling to send parts of the error information to the bit bucket.

7.2 Types of Exception Handling

It is time to explain what is understood by "exception handling". There are 3 basic ways to react to an exception in a catch handler.

7.2.1 Rethrowing

After some action is performed, the exception argument of the catch handler is thrown further:

```
catch(SomeException e){
   someAction(e);
   throw e;
}
```

This makes sense if and only if the caller of this method is not able to do what `someAction(e)` does because it has not the necessary data or something like that. Imagine e.g. that a private I/O stream has to be closed.

From the perspective of error information preservation rethrowing is innocuous — provided that the catch handler does not damage the exception by overwriting or deleting exception content, e.g. invoking `fillInStackTrace` or `setStacktrace` on it.

7.2.2 Conversion

There can be situations where rethrowing is impossible, viz. if the respective exception is not declared (if it is a checked one) or if the caller cannot be expected to cope with it.

Example 7.2.1. Imagine we have a proxy class for a session EJB with remote interface
`CustomerSession`:

```
class SessionProxy
{
   CustomerSession stub;
   ...
   public void foo() throws CustomerException
   {
      try{
         stub.foo();
      }catch(EJBException e){
         ???
      }
   }
}
```

We do not want to let the `EJBException` fly through to the client since the client needs to catch `RuntimeException`s and should not be tempted to handle `EJBException`s in a specific way because we do not want to make the client dependent on `javax.ejb`. Maybe we want to let room for changing the `SessionProxy` implementation to use a Web Service instead of a session EJB in a future version without invalidating the client's exception handling code. Therefore `SessionProxy.foo()` has to catch the `EJBException` and cannot rethrow it.

Exception conversion is one way of preserving error information without rethrowing. It means that a new exception containing the original error information is raised. An easy way to achieve this is provided if the new exception

can contain another exception (cf. 4.2). Then we can include the original exception in the new exception. Note, however, that this can be problematic if the caller lives in another JVM (cf. 10.2). Otherwise the exception data has to be extracted from the original exception and filled into the new one.

Example 7.2.2. Wrap the original exception with a new one:

```
void foo() throws FooException
{
   try{ bar(); }
   catch(BarException e){
      throw new FooException(
         Thread.currentThread().getName(),
         e
      );
   }
}
```

7.2.3 Logging

The third standard operation on encountering an exception is to report information contained in the exception (usually together with additional information describing the runtime situation (cf. 7.1)), which has already been explained in 6. It is of course possible to log part of the data and throw the remainder upwards as in 7.2.2.

7.3 Where To Handle Exceptions

Similar to the remarks concerning logging (6.4), exceptions should be handled at a point in the call chain where they can be handled and where the caller cannot handle them reasonably. If the information carried by the exception is to be logged, the remarks with respect to logging (6) apply for the question at which place in the call chain this logging has to be performed.

An important rule derived from this consideration is: "Don't catch what you don't have to!" If the caller can handle an exception, it is better to let him do it. Obeying this advice helps to minimize the amount of exception handling code, which, as stated earlier, tends to be particularly bug-prone in practice.

Here are some common sufficient conditions for handling an exception at a certain point. Let us consider the following scenario: A method c calls a method f which calls a method throwing an exception E. f should handle E

- by logging its contents if c and f belong to different processes and it is required that each process should log exceptions originating in it (e.g. because backtraces would otherwise get lost, cf. 10.1)
- if letting get E up to c would violate encapsulation restrictions, e.g. if E is specific for a software layer the layer of c should not know (depend upon) (example is given below (Ex. 7.3.2))
- if f can perform error recovery which c cannot (cf. Ex. 7.3.2)
- and of course in the trivial case that the prescribed interface of f does not permit to throw E to c

Example 7.3.1. In this example, we're going to write an online banking system. The software runs on multiple machines connected through a network. One reasonable requirement is that errors occurring in one process should be logged in a log file used exclusively for this process. Besides other operations that shall be ignored here, our system has a use case for money transfers. A servlet gets an HTTP request, then calls an Enterprise JavaBean operation, say

```
void transferEJB(Account src, Account dest, BigDecimal amount)
```

(exceptions omitted), which calls another method

```
void transferJCA(Account src, Account dest, BigDecimal amount)
```

(exceptions omitted) of a JCA resource adapter interface, which calls another method

```
/**
@throws BadDestAccount if the destination bank associated to
   <code>dest</code> does not exist
*/
void transferEIS(Account src, Account dest, BigDecimal amount)
   throws BadDestAccount
```

of an interface representing a legacy backend (Enterprise Information System).

The question is of course where to handle the `BadDestAccount`. The non-existing bank has probably been entered by a human user, so it is this user who is responsible for correcting the error. Hence the exception should be handled by the servlet by sending an appropriate error response to the user. The EJB and JCA layers should not be concerned with this exception since they *cannot* handle it; they should let it pass up to the servlet. This means in particular that the first 2 transfer methods also have to declare `BadDestAccount`.

Example 7.3.2. Continuation from Ex. 7.3.1. Assume now that `transferJCA` also declares a checked `javax.resource.ResourceException`:

```
/**
@throws BadDestAccount if the destination bank associated to
   <code>destAccount</code> does not exist
@throws ResourceException if connection to the EIS is lost
*/
void transferJCA(Account src, Account dest, BigDecimal amount)
   throws BadDestAccount, ResourceException
```

This one cannot be handled by the servlet, which should not depend on the fact that the EJB uses JCA internally, hence the EJB has to handle it by logging it and maybe re-establishing the connection or choosing an alternative (fallback) backend. If the transfer cannot be executed, the user has to be informed, of course, so the EJB should raise an unchecked exception containing the error information the servlet needs (for logging or displaying to the user), e.g. the string `"internal server error"`. The servlet is not accountable for logging the whole error information contained in the `ResourceException` because due to the requirements stated above the application server process is responsible for logging errors originating inside it.

7.4 Losing Error Information

As this mistake is so ubiquitous in program code, it seems worthwhile to present some examples of bad exception handling practices where error information is getting lost.

For demonstration purposes, we assume that logging is done via `System.err`, such that the remarks do not become dependent on some non-trivial logging interface. And as it is equivalent from the point of view of saving error information to log data or to include it in an exception that is thrown to the caller (who is then responsible for taking care of it), we assume that error information should not be passed to the caller (cf. 6.4), i.e. if something has to be logged, the current method has to log it.

Example 7.4.1. Complete neglection:

```
catch(FooException e){}
```

In most cases, this is wrong code because the fact that an error occurred as well as its circumstances should not be discarded. It is not wrong, however, in the rare case that the exception does not indicate a real error and neither the fact that it occurred nor the information contained in it are required to be logged.

Example 7.4.2. Handling without reporting. The deployment service of some J2EE application server reads deployment descriptors, which are XML files containing, among other data, size specifications for EJB pools.

```
try{
   poolSize = Integer.parseInt(poolSizeSpec);
}catch(NumberFormatException e){
   // default value
   poolSize = 250;
}
```

`poolSizeSpec` is the string from the deployment descriptor. If this cannot be interpreted as an integer, the pool size is set to the default value of 250.

Now imagine the EJB deployer preparing the deployment descriptor wants to restrict the pool size to 100 bean instances, but mistypes the string by adding an imprintable character that is invisible in the editor he uses for editing the XML file. Try to estimate how long it will take the poor deployer to notice that the pool size for the EJB is wrong at runtime (250 rather than 100) and find the invisible cause of this error!

If the deployment service produced an error message like "pool size †100 on line number 853 is not a number" (by throwing an appropriate exception), the deployer would be enabled to fix the deployment descriptor immediately.

Example 7.4.3. If, for example, one uses the method `getObjectById` of `javax.jdo.PersistenceManager` to get objects from the database, and it is specified to return nil if the desired object does not exist, then, since `getObjectById` throws `JDODataStoreException` if the object is not available, a correct implementation might be:

```
/**
@return null if such an object does not exist
*/
Object objectById(Object id){
   Object result = null;

   try{ result = myPersistenceManager.getObjectById(id); }
   catch(JDODataStoreException e){
      // no such object
   }

   return result;
}
```

Example 7.4.4. Stringifying:

```
catch(SQLException e){
   System.err.println(e);
}
```

Avoid this! `println` outputs `e.toString()` and who knows how this is implemented? Does it contain the stack trace? I have seen `SQLException`s implemented by a certain JDBC driver where `toString` not even contained the error message from the database (actually, it produced an empty string).

Example 7.4.5. Detail message:

```
catch(SQLException e){
   System.err.println(e.getMessage());
}
```

Now one can be sure to get the error message. But how useful is it to see the line: "wrong parameter type" on the console? At least one would like to know in which context this error happened, which of the SQL statements failed and in which use case. Consequently (cf. 6.6), what is missing here is the exception's stack trace.

Example 7.4.6. Stack trace:

```
catch(SQLException e){
   e.printStackTrace();
}
```

Assuming that the `SQLException`s `e` we catch have not overridden `printStackTrace` in a weird manner, this will log the stack trace, including stack traces of nested exceptions (those retrieved with `getCause`), and also including detail messages if `toString()` is not overridden strangely. But `SQLException`s can have additional nested `SQLException`s retrieved with `getNextException`. These can also contain precious error information, but get lost here. Moreover, `e.getErrorCode()` and `e.getSQLState()` are ignored.

Remark 7.4.7. Exceptions often have other ("inner") exceptions attached. Usually the inner exception is the cause of the outer exception, so if one wants to get to the cause of an error, the inner exception is more important than the outer one. Nevertheless it is a common mistake to omit careful logging of nested exceptions (recursively if the nested exceptions also have nested exceptions).

Not all the information relevant for tracking down an error mentioned in 7.1 is contained in the exception. Besides the information on the code location where the error occurred, the most important information are the actual data which were processed. A last prototypical example:

Example 7.4.8. We have an interface modeling persons with certain secret enciphered data.

```
interface Person
{
   byte[] secret();
   ...
}
```

We use this class in a software system that gets nightly updates for a database of persons. The update arrives as a big XML file containing data records for persons that have to be added to the database. The respective batch process parses the XML file, for each person record creates a corresponding `Person` object and invokes a method

```
void create(Person p){
   try{
      store(p, myCipher.doFinal(p.secret));
   }catch(BadPaddingException e){
      e.printStackTrace();
   }
}
```

using a `javax.crypto.Cipher myCipher` for deciphering the secret data and a method `store` for performing the actual database input. The

`BadPaddingException` can be thrown (among others) by `Cipher.doFinal` if the cipherdata is corrupt.

What's wrong here? Just imagine what the console displays: We learn that some person could not be added to the database because that person's secret could not be deciphered (it was probably corrupted on the way from the remote database to our XML parser). Now the 2 databases are inconsistent: the respective person is in the remote database but not in ours. How will we be able to fix this inadmissible state or get to the cause of the corruption if we do not have any idea which concrete person this is all about? This information is not in our console log, since the `BadPaddingException` will certainly not know which person the cipherdata was aggregated to. A much more useful catch handler is the following (assuming that the `Person` implementation has a sensible `toString`):

```
catch(BadPaddingException e){
   System.err.println("\n" + p + ':');
   e.printStackTrace();
}
```

7.5 Error Series

A common phenomenon in software reality is that one error occurring at runtime induces other errors. An error situation can cause multiple exceptions. The point to be made here is that usually the exceptions occuring earlier in time are more important than the later ones for the error analysis. The simple reason is that the chronological dependency in most cases corresponds to a causal dependency. If one erroneous state E_2 is implied by another one, E_1, then E_2 will not occur before E_1 has occurred. E_1 is the error to be tackled in order to avoid it, and this might already eliminate E_2. Consequently, programmers should make sure that the original error information does not get hidden by secondary errors. The preferred locations for overriding of errors are `finally` blocks.

Example 7.5.1. Assume we have an interface

```
interface Service
{
   void work();
   void close();
}
```

, a method

```
void closeService(Service s){
   if (s == null) throw new IllegalArgumentException("nil argument");
   s.close();
}
```

, and a method

```
void callService() throws NamingException
{
   Service s = null;

   try{
      s = (Service)new InitialContext().lookup("S");
      s.work();
   }finally{
      closeService(s);
```

```
    }
}
```

The intentions of the fictitious author of this code are the following:

- The method looks up a `Service` object by its name in JNDI, then uses it, then closes it.
- This closing has to be executed even if an error (`RuntimeException`) occurred using the `Service`, therefore it is done in a `finally` block.
- The statement `new InitialContext().lookup("S")` throws a `NamingException` if something is wrong with JNDI, in particular if no object is registered under the name `S`, and this `NamingException` is thrown out of the method since configuration or initialization errors like this are to be handled in another layer.
- The function `closeService` regards it as a precondition violation if its argument `s` is nil and throws an unchecked exception in this case.

All this sounds reasonable, but assume that `S` cannot be found in JNDI. This error situation is our E_1. It causes a second one, E_2, viz. that `s == null` in the `finally` block. The exception corresponding to E_1 is a `NameNotFoundException` containing the essential information that `S` could not be found in JNDI. The exception corresponding to E_2 is the `IllegalArgumentException`. The first of these exceptions explicitly states what was wrong, whereas the second one, if e.g. found in a log file, does not tell anything specific to someone who does not know the source code.

Now what happens at runtime? If the lookup of `S` fails, a `NamingException` occurs. The execution then goes into the `finally` block and here `s == null`, which leads to an `IllegalArgumentException` thrown out of the `finally` block. In contrast to what the author had imagined, the method throws the unspecific `IllegalArgumentException` instead of the specific `NamingException` if `S` does not exist.

In this simple case it is of course easy to deduce E_1 from E_2 if the stack trace of the `IllegalArgumentException` is known and the source code is inspected, but this is only due to the simplicity of the example. In reality it can be impossible to conclude backwards as here. Secondary error states can have more than one possible underlying primary error state, and these causes can be arbitrarily hard to reconstruct if the information about them is not saved.

7.6 Bugs

The situation that software detects bugs at runtime has been treated already in § 5.5.2. This section is devoted to the case where this detection occurs in a catch handler. Exceptions resulting from bugs, i.e. programming errors, in the software being developed, deserve special handling, guided by the principles that

(1) they do not occur in the release version

(2) they must not occur in the release version

As ridiculous as this may sound, it has serious consequences for exception handling:

(1) implies that these exceptions do not need to be caught

(2) implies that *if* they are caught, they should be handled with utmost priority, leave a maximal visible trace, up to stopping the runtime process, in order to increase the chance for getting fixed

Naturally, (1) applies only to unchecked exceptions, since checked exceptions have to be caught somewhere anyway.

Example 7.6.1. A local utility method for extracting a subarray shall serve as an example for a violation of principle (1). "Local" means that this method is not public, it is used exclusively by our own code.

```
/**
@param length length of the subarray
*/
int[] subarray(int[] a, int begin, int length){
   try{
      int[] r = new int[length];
      System.arraycopy(a, begin, r, 0, length);
   }catch(NegativeArraySizeException e){
      e.printStackTrace();
      throw e;
   }
   return r;
}
```

The intention is to log the error if the caller invokes this method with a negative `length` argument. But this is nonsense, because this would be a bug of the calling code, which is our own; this bug has to be fixed so the `NegativeArraySizeException` cannot occur anymore. Letting the `NegativeArraySizeException` simply hit the caller is exactly the right thing to do. The `try-catch` here only derogates performance.

Remark 7.6.2. If this method was public and used by foreign code, things would differ substantially. Then we could not simply fix the caller; we were in the situation of 5.5.3, thus we should take care of throwing something meaningful (assuming that the `NegativeArraySizeException` does not carry sufficient error information). By the way, catching `NegativeArraySizeException` is also an example of misusing exceptions for flow control (cf. 3.2) — the negativity of a number can be checked in a simpler and more natural way. The correct implementation in the non-local context would be:

```
/**
@param a the array to extract from
@param begin offset of the subarray
@param length length of the subarray
@pre a != null
@pre length >= 0
@pre begin >= 0
@pre begin + length < a.length
*/
public int[] subarray(int[] a, int begin, int length){
   if (length < 0)
      throw
         new PreconditionViolation(
            "negative subarray length " + length
         );
   if (begin < 0)
      throw
         new PreconditionViolation("negative offset " + begin);
   if (a == null)
      throw new PreconditionViolation("nil array argument");
   if (begin + length >= a.length)
      throw
```

```
            new PreconditionViolation(
                 "offset "
               + begin
               + " plus length "
               + length
               + " not less than "
               + a.length
               + " = length of a"
            );

      int[] r = new int[length];
      System.arraycopy(a, begin, r, 0, length);
      return r;
}
```

Let us now turn to (2). A phenomenon not so rarely encountered in Java code is illustrated by the following

Example 7.6.3. Exception that cannot occur:

```
try{
   Class c = Class.forName("org.acme.JdbcDriver");
   c.newInstance();
}catch(IllegalAccessException ignore){
   // cannot happen
}
```

The `IllegalAccessException` has to be caught because this method cannot declare it but `Class.newInstance` declares it. The programmer's reasoning might be that the `try` block cannot throw an `IllegalAccessException` (because
`org.acme.JdbcDriver` is known to possess a public nullary constructor), so there is no need to handle it. But I am criticizing the implementation of the catch handler. If we are inside this catch handler, we *have* caught an `IllegalAccessException`, and if we are "sure" that this cannot happen, this is a very severe error, isn't it? And severe errors, especially bugs in the software or the human brain, should not be ignored.

Example 7.6.4. Having received this lesson, the programmer improves the code:

```
catch(IllegalAccessException e){
   // this is a bug, must not be ignored
   myLogger.log(e);
}
```

... and has produced an example for the next bad exception handling practice. Logging an exception that is anticipated to occur in the release version, in the production environment, in this way can be correct. But only logging the software bug is not the maximal trace an error can leave. Does the programmer exactly know what the logger does with `e`? Maybe it is configured to do nothing for this category of log records. Maybe it sends an e-mail to a person who in the meantime has been fired on the grounds of poor error handling. Maybe it just creates a string object in a database. But even if the programmer knows exactly that the logger will log the exception in a reasonable way, say to the console or into a log file, who knows if and when somebody will read this message? The maximal alarm noise that can be produced in this situation is to raise an unchecked exception (we use a `Bug extends RuntimeException` here:

```
catch(IllegalAccessException e){
   // this is an outrage, exclaim
   throw new Bug(e);
}
```

Why? Because throwing is much more likely to produce a *noticeable* failure of the software functionality. Logging but not aborting, proceeding in the business logic as if nothing had gone wrong, has more unpredictable consequences. The erroneous behavior might show up at a later time, in an unexpected place, maybe by strange database contents or something like that, and the cause of the then visible error can be arbitrarily hard to find.

7.7 Tracing and Heisenbugs

In server-side software it is a custom to distinguish between logging and tracing. Tracing records information used only in the development phase for debugging or performance measurements and is switched off in production. Logging is a part of the business requirements; not only errors are logged, but also any information that might be useful to be recorded for future examination. Logging is always active. In practice this could be realized by a `Logger` interface having two methods `log` and `trace`, where the latter one does nothing unless a certain configuration property (for the logging verbosity) is set. Now a common bad practice with respect to error handling is to report (severe) exceptions, in particular exceptions that "cannot occur" (cf. 7.6), using the `trace` method, so they are not logged in the production environment. This can be a deadly mistake.

Example 7.7.1. A method for a money transfer in an EJB:

```
public void transfer(TransferRequest t){
   try{
      // do the JDO stuff
      ...
   }catch(JDOException impossible){
      myLogger.trace(impossible);
   }
}
```

The idea of the programmer is that logging should be avoided if it is not explicitely required in the business requirements, because too much logging is bad for performance and produces huge log files that are hard to examine, and if the error actually occurs, tracing can be switched on such that the exception is then logged and the error can be investigated. Unfortunately, if the error really occurs in the production system, the customer will show up with the urgent inquiry where all his money has gone, and this cannot be answered due to the missing error information. And even more unfortunately, the error having caused the exception cannot be found and fixed because one does not know how to reproduce it on the development system with tracing switched on.

The correct approach with respect to tracing is to distinguish between business errors (e.g. customer enters a wrong account number) and system-level errors (e.g. unexpected I/O error). Whether a business error is to be logged is (or at least should be) specified in the functional requirements. System errors (which business consultants do not have to be concerned with) have always to be logged (maybe into a dedicated log sink), and the worse the error, the more seriously this requirement should be taken. If this requirement really has a negative performance impact, this means that the frequency of these errors has reached a level where on the one hand the performance does not matter anymore in view of the general malfunction of the system, and on the other hand the system should be debugged instead of being run in production.

Logging errors always is especially important for catching so-called "heisenbugs", i.e. bugs manifesting themselves only in intermittent failures one is not able to reproduce in a deterministic way (the name alludes to W Heisenberg's Unschärferelation in quantum mechanics).

Chapter 8

Trying

8.1 Designing `try` Statements

Often one has multiple choices for grouping code the exceptions of which have to be caught into `try` statements. For the sake of readability, a solution with a smaller number of (consequently larger) `try` statements is preferable over a solution with a larger number of (consequently smaller) `try` statements. Lots of small `try` statements contradict the very purpose of exceptions, viz. separation of error handling from "good case" code (cf. 3.1) with the goal of making the code more maintainable.

Example 8.1.1. Multiple small `try` statements:

```
int n = 0;
boolean isFooSuccess = true;
try{
   n = foo();
}catch(FooException e){
   log(e);
   isFooSuccess = false;
}
if (isFooSuccess ){
   try{
      bar(n);
   }catch(BarException e){
      log(e);
   }
}
```

certainly make less comprehensible what the code really does, viz. executing `bar` with the result of the execution of `foo`, than one larger `try` statement, as shown below:

```
try{
   bar(foo());
}catch(FooException e){
   log(e);
}catch(BarException e){
   log(e);
}
```

There are two additional remarks to be made in connection with this little example. When refactoring code with respect of the partition into `try` statements one has to be very careful not to change the flow of control inadvertently. If I had omitted the `isFooSuccess` switch in the first code snippet, i.e. written it as

```
int n = 0;
try{
   n = foo();
}catch(FooException e){
   log(e);
}
try{
   bar(n);
}catch(BarException e){
   log(e);
}
```

a considerable percentage of people would not have noticed at first glance that the two code snippets were not logically equivalent (provided that `log(e)` does not throw, to be precise).

Another nasty effect of multiple `try` statements is the following: Why did I have to initialize the variable `n` as 0 in the first code snippet? If you inspect the code you will see that when the value of `n` is read, this variable would have a well-defined value (viz. the result of the `foo` call) anyway. The reason is that due to the catch block between the function calls your Java compiler might not recognize what we see, viz. that the `bar` call is never executed if the `foo` call does not complete normally, and might complain about reading an uninitialized variable.

The size of `try` blocks can be limited by the requirement to avoid the situation that

(*) some exception class can be thrown by multiple statements in the block, but its handling depends on where it had originated

Example 8.1.2. If we have a `File` object `file` and want to perform the following operations on it:

```
   file.createNewFile();
   new FileOutputStream(file).write(1);
```

where it is required to react to an `IOException` thrown by the first statement by reporting "file cannot be created" and to an `IOException` thrown by the second statement by reporting "cannot write", then these operations should not be contained in the same `try` statement catching `IOException`:

```
try{
   file.createNewFile();
   new FileOutputStream(file).write(1);
}catch(IOException e){
   ???: the difficulty
}
```

The conclusion is to make `try` blocks as large as it is possible without running into condition (*).

8.2 Nesting `try` Statements

Code stylists say `try` statements should not be nested since nested `try` statements make the code hard to read. In order to resolve a common misunderstanding: they do not mean to split nested `try` statements into lengthy non-nested constructs. They propose to define methods corresponding to inner `try` statements and to replace these blocks by the respective method calls.

Example 8.2.1. The following code with one `try` statement having an inner `try` statement (for understanding which one has to know that all the called methods except for those in the `catch` blocks can potentially throw `JDOException`):

```
try{
   Object id = y.jdoNewObjectIdInstance();
   y.jdoCopyKeyFieldsToObjectId(id);
   try{ x.setY((Y)persistenceManager.getObjectById(id, false)); }
   catch(JDODataStoreException e){
      throw new NoSuchObject();
   }
   persistenceManager.makePersistent(x);
}catch(JDOException e){
   // caller does not log this properly
   myLogger.log(e);
   throw e;
}
```

should not be rewritten in the form of 3 `try` statements

```
try{
   Object id = y.jdoNewObjectIdInstance();
   y.jdoCopyKeyFieldsToObjectId(id);
}catch(JDOException e){
   // caller does not log this properly
   myLogger.log(e);
   throw e;
}
try{ x.setY((Y)persistenceManager.getObjectById(id, false)); }
catch(JDODataStoreException e){
   throw new NoSuchObject();
}catch(JDOException e){
   // caller does not log this properly
   myLogger.log(e);
   throw e;
}
try{ persistenceManager.makePersistent(x); }
}catch(JDOException e){
   // caller does not log this properly
   myLogger.log(e);
   throw e;
}
```

but like this:

```
try{
   Object id = y.jdoNewObjectIdInstance();
   y.jdoCopyKeyFieldsToObjectId(id);
   x.setY((Y)getPersistent(id, persistenceManager));
   persistenceManager.makePersistent(x);
}catch(JDOException e){
   // caller does not log this properly
   myLogger.log(e);
   throw e;
}
```

with the extra method for the inner `try` statement:

```
Object getPersistent(Object id, PersistenceManager pm)
   throws NoSuchObject
   // and can also throw JDOException
{
   try{ return pm.getObjectById(id, false); }
```

```
   catch(JDODataStoreException e){
      throw new NoSuchObject();
   }
}
```

Remark 8.2.2. In order to keep the example above free from unessential ballast, the `throw` statement was chosen to be the simplest possible. But in real software, a caller getting an empty `NoSuchObject` exception would curse me because he would passionately like to know at least *which* object did not exist in the database. In view of the guideline 5.7 I should throw something like `new NoSuchObject("non-existent object " + y + " associated to " + x)`. (Supposing that we know that `e` means nothing more than the fact that `y` is not in our database, it is not needed to include `e` in the `NoSuchObject` since this information is already contained in the error message.)

8.3 Releasing Resources

A common situation is that for executing some operation multiple resources have to be acquired which have to be released explicitly after usage. The problem arising in this context is that one has to ensure that the resources are released even if an exception occurred. This is best clarified by an example that may look familiar to developers using JDBC or JMS:

Example 8.3.1. Let us start with one resource:

```
void method foo() throws FooException
{
   A a = myAFactory.getA();
   a.doSomething();
   a.close();
}
```

All the statements inside `foo` can throw `FooException`. This implementation is syntactically correct, but not semantically, because if `a.doSomething()` completes abruptly, the mandatory `close` statement will not be executed.

We therefore have to execute the `close` statement even if a `FooException` has occurred prior to this statement. This suggests to put `a.close()` into a `finally` block. (In C++ one would use a stack-based object for `a` and call `close` in its destructor.) The difficulty results from the fact that the `close` statement can also throw `FooException`. So if we get a `FooException`, and then get another `FooException` on closing `a`, we have collected two exceptions, but can throw only one. One solution is to log the less important second exception and to throw the first one. More elegant is it, however, to wrap the first exception in the second one (and not the other way around, since the caller expects the inner exception to be more original than the containing exception (4.2, Rem. 7.4.7)) and throw the second one, which is what we will do here. But inside a `finally` handler one does not know if an exception occurred prior to entering this handler, unless one has memorized this incident somehow.

A correct implementation would be

```
void method foo() throws FooException
{
   FooException error = null;
   A a = myAFactory.getA();

   try{ a.doSomething(); }
   catch(FooException e){
      error = e;
```

```
   }finally{
      try{ a.close(); }
      catch(FooException e){
         e.initCause(error);
         throw e;
      }
      if (error != null) throw error;
   }
}
```

Example 8.3.2. The real fun begins if we really have *multiple* resources, if for example the `myAFactory` also has to be closed. Wrong implementation without exception handling:

```
void method foo() throws FooException
{
   B b = myBFactory.getB();
   A a = b.getA();
   a.doSomething();
   a.close();
   b.close();
}
```

It is assumed that closing one of our resources `a`, `b` does not close the other one automatically.

Along the lines of the previous example a correct implementation is

```
void method foo() throws FooException
{
   FooException error = null;
   B b = myBFactory.getB();

   try{
      A a = b.getA();
      try{ a.doSomething(); }
      catch(FooException e){
         error = e;
      }finally{
         try{ a.close(); }
         catch(FooException e){
            e.initCause(error);
            error = e;
         }
      }
   }catch(FooException e){
      error = e;
   }finally{
      try{ b.close(); }
      catch(FooException e){
         e.initCause(error);
         error = e;
      }
      if (error != null) throw error;
   }
}
```

(If `foo` may not throw `FooException`, the code can have the same structure, but gets even lengthier.) Now at latest I hear you wonder why this is so damn complicated. The reason is that the factory methods for retrieving our resources

stand outside the `try` statements responsible for closing these resources. This is justified because the resource does not need to be released if its retrieval fails. We know that acquiring the resource succeeded because otherwise we would not have entered the `try` statement. But this information can also be obtained in another way: Successful retrieval of the resource sets the corresponding reference to a non-nil value. And this can be used to simplify the code structure:

```
void method foo() throws FooException
{
   FooException error = null;
   B b = null;
   A a = null;

   try{
      b = myBFactory.getB();
      a = b.getA();
      a.doSomething();
   }catch(FooException e){
      error = e;
   }finally{

      if (a != null){
         try{ a.close(); }
         catch(FooException e){
            e.initCause(error);
            error = e;
         }
      }

      if (b != null){
         try{ b.close(); }
         catch(FooException e){
            e.initCause(error);
            error = e;
         }
      }

      if (error != null) throw error;
   }
}
```

And if we have to handle more resources that have to be released, and do it in this way, the code certainly grows longer, but does not become more deeply nested.

Chapter 9

Catching

A `try` block b determines a set of checked exception classes T it can throw, namely the union of the sets of exceptions the individual methods called in this block declare to throw (if they are not caught within the block, of course) plus the checked exceptions raised in the block. The union C of the sets of exception classes caught by the catch handlers attached to b must contain T; this is enforced by the compiler. Should C coincide with T? It depends...

9.1 Fine-Grained Catching

T should coincide with C with respect to checked exceptions. Even stronger, this should hold on the abstract level in the sense of Def. 2.6, i.e. even in a future version of our software, which means that if new exception classes emerge and belong to C, they should also belong to T.

Example 9.1.1. Assume that our software system defines a checked exception class `B`, exactly one class `E` derived from `B`, and a method

```
void f() throws E;
```

Now consider the following `try` statement:

```
try{ f(); }
catch(B e){...}
```

T is the set of all exceptions derived from `E`, C is the set of all subclasses of `B`. On the concrete level, these sets coincide, they are both equal to {`B`, `E`}. But on the abstract level, every potential subclass of `B` that is not derived from `E` belongs to $C \setminus T$. The practical consequence is that if the next version of our software contains an additional class `F` derived from `B` but not from `E`, then the unchanged catch handler would also catch `F`.

What's the problem with that? The reason for my requirement that T and C should coincide on the abstract level is the desire for maintainability of the software. Let us be so good-hearted to assume that the present code, b and its catch handlers, is correct. In the next version of the software, one of the methods called within b changes by declaring a new exception class belonging to C but not to the former T, which is possible if T and C differ on the abstract level. The reason for declaring a new exception is usually (cf. 5.6.2) that the caller is supposed to handle the new exception in a specific way, different from the handling of other declared exceptions. And the reason for declaring exceptions at all is that the compiler can force the caller to take care of the exceptions. This concept is thwarted in our situation: b and its catch handlers are still syntactically correct; the compiler cannot notify the programmer of b about the essential change b has undergone.

Example 9.1.2. In this example, acme.org is writing a bug tracking system. The "client" group is responsible for writing the user interface (UI), the "server" group writes the middleware used by the UI. All checked exceptions thrown by the middleware are derived from a mother class `CheckedException`. The interface of the middleware contains a method for deleting a bug report:

```
public interface BugMeister
{
   /**
   @param b the bug report to be deleted
   @throws NoSuchObject if the database does
           not contain <code>b</code>
    */
   void delete(Bug b) throws NoSuchObject;

   ...
}
```

and here is how dangerously the client group uses it:

```
BugMeister m;
Bug b;
...
try{ m.delete(b); }
catch(CheckedException e){
   // We know that e is a NoSuchObject.
   // This is a severe logical error in the UI software
   // since b was obtained from the middleware earlier.
   displayToUser(
      "I am broken. Do not use me anymore. Try the next version."
   );
}
```

Some day, the chief software architect of the server group figures out that the reason for `NoSuchObject` can also be that while one user was studying the bug report another user deleted it. As he does not want to support long-term transactions the boundaries of which are set by the UI, he decides that in the announced next version the middleware should employ a lock service. The lock service throws a new exception class `LockException` derived from `CheckedException`. Before updating a bug report, the user has to lock it, so it cannot be modified by other users. But according to all guidelines in this paper, which the architect has carefully read (if he didn't even write it), this implies a change in the interface:

```
public interface BugMeister
{
   /**
   @param b the bug report to be deleted
   @throws NoSuchObject if the database does
           not contain <code>b</code>
   @throws LockException if the bug report is locked
           by another user
    */
   void delete(Bug b) throws NoSuchObject, LockException;

   ...
}
```

The unchanged client code is still compatible with this interface, but it is an error not to change it: if a `LockException` is caught, this is not a bug, rather

the user should be told which other user has locked the bug report, to update their copy of the bug report, to wait for some time or so.

If the catch handler in the client code had caught exactly what is thrown, this error could be prevented by force of the compiler.

Why would one catch more than is thrown? In fact, this is done very often in practice. Maybe the following example looks familiar to you:

Example 9.1.3. Programmer Dieter writes a block b, then the compiler tells him that b throws `InstantiationException`, `NoSuchMethodException`, `IllegalAccessException`, `InvocationTargetException`. Dieter knows that these exceptions cannot occur and in particular do not need to be handled on an individual basis, he therefore finds it a waste of time to write 4 catch handlers for exceptions with such long names, so he writes

```
try{ b }
catch(Exception e){ log(e); }
```

and although this approach is kind of well established, it is one of the worst exception handling practices. Dieter may be right in believing that the exceptions listed above cannot occur or do not require another handling than being only logged. Maybe Dieter is even right with respect to forthcoming versions of the software. The first criticism is the same as in the previous example: if the methods called in b change their `throws` clauses, Dieter gets no notification from the compiler. If in b a `RemoteException` is thrown (which did not occur in the earlier version not using RMI), Dieter's code logs it, but attempts to continue normal processing besides that. But worst of all, regardless of whether the software changes or not, the catch handler also catches all `RuntimeException`s, and under awkward questioning Dieter confesses that in the case that b throws a `SecurityException` or a `NegativeArraySizeException` he did *not* want the software to proceed as if nothing had gone wrong, which can hamper debugging enormously, as explained earlier.

Catching `RuntimeException`s is in fact in most cases unwanted (and, reversely, if one finds oneself often catching a certain `RuntimeException`, this is an indicator that this exception had better be a checked exception). Cf. 9.2 for reasons for catching `RuntimeException`s.

9.2 Catching Everything

Common reasons for catching unchecked `Throwable`s are

- the method has to avoid throwing whenever possible
- the process must not exit
- these incidents have to be logged, but would not if they were not caught

As usual in this paper, examples are provided for these situations:

Example 9.2.1. If the implementation of the operation `onMessage` in an implementation of `javax.jms.MessageListener` throws a `RuntimeException`, the message is not taken off the queue and `onMessage` is immediately called again, for the same message, which is often unwanted. Therefore an implementation of `onMessage` should make sure to catch all `RuntimeException`s occurring in it, reacting on the `RuntimeException` by something like logging but not rethrowing.

Example 9.2.2. In this example, we are going to write a servlet engine serving HTTP requests. One requirement is that no matter what weird stuff the possibly buggy servlets running in this engine put on, the engine has to stay alive, i.e. its `main` method must not exit, hence the `main` method must not throw an

exception. Therefore the web container encapsulating a servlet has to intercept exceptions thrown by the servlet. Even more, if a servlet causes, e.g., a `java.lang.LinkageError`, the server process might be rescued if the servlet is removed. The servlet invocation in the web container consequently should look like this (pseudo-code):

```
try{ servlet.doService(...); }
catch(ServletException e){
   // do what the specification prescribes
   ...
}catch(RuntimeException e){
   log(e);
   // send some error response
   // blaming the servlet to the HTTP client
   ...
   // do not rethrow
}catch(LinkageError e){
   log(e);
   // remove (undeploy) the servlet
   // send some error response
   // blaming the servlet's deployment to the HTTP client
   ...
}
```

Example 9.2.3. In a middleware system one has the natural desire to log errors. The more severe the error, the stronger the desire. As pointed out in 9.1 one rarely wants to really handle unchecked exceptions, but they should not leave the system without a trace. According to 7.3 the correct way to implement this is to let the unchecked exceptions inside one process fly up to the outmost local layer of the system, i.e. the first code being entered when a remote call occurs, and log and rethrow it there. Rethrowing the unchecked exceptions is the right thing to do since the application server follows regulations for handling these exceptions which the remote client relies upon. A business method in an EJB implementation class consequently ought to look like this:

```
try{
   // business logic, responsible for handling checked exceptions
   ...
}catch(RuntimeException e){
   // catch(Exception) would be wrong since we do not care about
   // the checked exceptions this method has declared to throw
   log(e);
   try{
      // this instance will be destroyed
      cleanUp();
   }catch(RuntimeException r){ log(r); }
   throw e;
}
```

Remark 9.2.4. As stated above, server systems often have to catch `Error`s, such as `OutOfMemoryError`, for which recovery can at least be tried. If one catches `java.lang.ThreadDeath`, however, no matter how one handles it otherwise, it *must* be rethrown so the dying thread can be finished.

Chapter 10

Remote Calls

10.1 RMI

Often Java programmers are uncertain about what happens to exceptions if they cross JVM boundaries. More precisely, the question is: What does the remote caller really get if the implementation of an RMI method throws a certain exception? To clarify this, let us investigate an RMI method

```
void foo() throws RemoteException, FooException;
```

declared in some remote interface. As the question is not completely covered by the RMI specification, the answer to the question can depend on the chosen RMI implementation, so be it stated here that the answers given below are valid for JDK 1.4.1 of Mac OS X 10.2.6. When speaking about "nested exceptions", we mean arbitrary associated `Throwable` objects, not only the exception attached to another exception via the `Throwable.initCause` mechanism of JDK $\geq$ 1.4.

Remark 10.1.1. What does the client get if the implementation of `foo` throws `FooException`?

Answer: A `FooException`, of course. The backtrace is in the client-side JVM, where the exception object is created on deserialization.

Remark 10.1.2. What does the client get if the implementation of `foo` throws `RemoteException`?

Answer: A `java.rmi.ServerException` (which is derived from `RemoteException`) containing the originally raised `RemoteException`. The backtrace of the nested `RemoteException` is that of the server JVM!

Remark 10.1.3. What does the client get if the implementation of `foo` throws a `RuntimeException` (the class of which is available for the client JVM)?

Answer: The same `RuntimeException`, but with client-side backtrace. Backtraces of nested exceptions, however, are the server-side ones.

Remark 10.1.4. What does the client get if the implementation of `foo` throws a `RuntimeException` the class of which is not in the client's classpath?

Answer: A `java.rmi.UnmarshalException` raised at the attempt of deserializing the exception object received from the server (cf. 10.2).

10.2 Serialization

Exceptions implement `java.io.Serializable`. If exceptions are really going to be serialized, one should make sure that all exception attributes are serializable or `transient`. This is of course not specific for exceptions. Deserialization of an object, however, does not only require it to be serializable, it also requires that its class be available at runtime. And this sometimes tends to be neglected with respect to exceptions.

Example 10.2.1. Caller C calls a method `foo` via RMI:

```
public interface Foo extends Remote
{
   void foo() throws RemoteException, FooException;
}
```

The implementation of `foo` internally uses `javax.jdo`, but C is independent of `javax.jdo`, i.e. C does not have `javax.jdo` classes in its classpath. Assume that the implementation of `foo` handles `javax.jdo.JDOException`s by wrapping them in `FooException`s, as proposed in 7.2.2:

```
try{ ... }
catch(JDOException e){ throw new FooException(e); }
```

Then when a `JDOException` occurs at runtime, C will not get a `FooException` containing a `JDOException`, as the implementor of `foo` had in mind, but instead C gets an unwanted `java.rmi.UnmarshalException` caused by the fact that the aggregated `JDOException`, hence the containing `FooException`, cannot be deserialized in C's JVM (cf. 10.1.4).

The great danger of wrapping exceptions the remote caller does not know into exceptions known to the caller is that this mistake cannot be detected by the compiler, it possibly shows up only at runtime (and probably not even in tests since error handling is seldom tested thoroughly).

Example 10.2.2. Regardless of the software design principle that implementation details like the usage of JDO should be hidden from client code, trying to solve the problem described above by putting `javax.jdo` into C's classpath is likely to lead to further problems since it is well possible that the real runtime type of the `JDOException` contained in the `FooException` is in fact in the namespace of the JDO provider rather than in `javax.jdo`. It could be, e.g., a `com.sun.jdori.query.JDOQueryException` if the JDO Reference Implementation from Sun Microsystems is used. And putting also the JDO provider's libraries into C's classpath would make C semantically (not necessarily syntactically) dependent even on the chosen JDO implementation...

10.3 EJB

The specification of exception handling of EJB containers in the EJB specification [3], Ch. 18, poses the following big problem to designers of EJB systems: How to cope with unchecked exceptions occurring in business methods of a session or entity bean? As usual the main principle is that error information must not get lost, especially in cases of severe errors corresponding to unchecked exceptions. So

1° one would like to keep a log record of the error information contained in the exceptions

2° the caller of the business method should get appropriate information about what happened

A reasonable "appropriate information" for a remote client is for example an ID that can be used for locating the entry corresponding to the respective exception in a log file.

But the specification [3], 18.3, prescribes only the following: If the EJB container detects an unchecked exception thrown by the business method, this is necessarily a non-application exception in the sense of [3], 18.1.1; therefore this exception has to be logged, and a
`javax.transaction.TransactionRolledBackException`,

`javax.ejb.TransactionRolledBackLocalException`, `java.rmi.RemoteException`, or `javax.ejb.EJBException`, depending on the execution context, has to be thrown at the client. And this means that our condition 2° is not guaranteed. The specification would already be satisfied if the EJB container threw an *empty* `RemoteException` at the caller (in case the caller was a remote one that did not start the transaction), without any error-specific information, let alone an ID for referral to the log record. And this implies that 1° is also not satisfied: the caller cannot log information not being provided to it, and the EJB container's logging cannot be relied upon since the phrase *"the Container logs the exception or error so that the System Administrator is alerted of the problem"* in the specification ([3], 18.3) leaves enough freedom for the EJB container provider to interpret this, for example, as printing only the exception class name to the console, which is certainly not the kind of error reporting a serious J2EE application requires. And even if the EJB container undertakes more effort for logging the unchecked exception, it is doubtful whether it logs every class of unchecked exceptions, including custom exceptions of our and third party software, in the proper way.

There is a simple procedure for achieving 1°: Catch all unchecked exceptions, the catch handler logs them using our own logging mechanism and throws an unchecked exception, preferrably the one that was caught. As the EJB container also does some logging, this will imply two logging activities for each unchecked exception, but the error information gets logged as it is required, independent of the concrete EJB container implementation.

Achieving 2° in a satisfactory manner seems to be impossible. Let us weigh some approaches against each other.

One idea is to throw a checked exception, say `InternalException`, containing the error information to be delivered to the caller when a business method catches an unchecked exception. This would work since the EJB container lets checked exceptions thrown by a business method pass to the client without modification. This `InternalException` would of course have to be declared in the enterprise bean's (remote or local) interface, which is why this idea violates principles of clean software design (5.5, 5.6.1).

The second idea is to check out what the EJB container implementation one is using really does with unchecked exceptions. Maybe it copies the message of the exception it throws to the caller from the message of the exception it has caught. Maybe it wraps the original exception in the exception thrown to the caller (an arguable idea because of the problems mentioned in 10.2). And if there is some data of the original exception that is transferred to the client, one can use this as an information channel for achieving 2°. The logic for encoding and extracting the "appropriate information" should be factored out to a one or two utility classes which are dependent on the EJB container and consequently may have to be adapted to each new EJB container version.

Example 10.3.1. The application server JBoss 3.2.1 (cf. [4]) throws a `java.rmi.ServerException` containing the unchecked exception coming from the EJB's business method to the remote caller. So if the information to be transmitted to the caller consists of a character string serving as an identifier for locating an entry in the server log file, we could define a non-nil field and accessors for it

```
private String id = "";

public String id(){
   return id;
}

public void setId(String anId){
   if (i == null) throw new Precondition("nil");
```

```
   id = anId;
}
```

in our basic unchecked exception class `org.acme.SystemException`. The EJB implementation now catches unchecked exceptions and handles them using the methods `ExceptionUtil.handle` from below (where we use a method `Logger.log` for logging the exception, returning the log record identifier)

```
/** Works for JBoss 3.2.1. */
public class ExceptionUtil
{
   public static void handle(Error e, Logger logger){
      logger.log(e);
      throw e;
   }

   public static void handle(RuntimeException e, Logger logger){
      RuntimeException result  =
         (e instanceof SystemException)?
               e
            :
               new SystemException();

      result.setId(logger.log(e));
      throw result;
   }

   /**
   @return the server log record ID if <code>t</code> is a
      <code>ServerException</code> containing a
      <code>SystemException</code>, nil otherwise
    */
   public static String getId(Throwable t){
      String result = null;
      if (t instanceof ServerException){
         t = t.getCause();
         if (t instanceof SystemException){
            result = ((SystemException)t).getId();
         }
   }
}
```

The client code would use `ExceptionUtil.getId` for retrieving the log record identifier from the exception.

One could as well define an interface with the methods `handle` and `getId` and a factory for getting an EJB-container-specific object implementing this interface...

10.4 Timeouts

This section is not specific for object-oriented software and exceptions.

From the point of view of error handling, method invocations (synchronous, blocking calls) are dangerous. This may sound paranoid — until the feared situation really occurs: cf. the examples below. The danger is that, if the implementation is not sufficiently well known to exclude this possibility, there is a potential for the method execution *never* completing, neither normally nor abruptly. This is certainly a severe error, but this error cannot even be detected by the current thread, which is then stuck.

Example 10.4.1. With a certain application server, I ran into the problem that RMI calls from a remote process to the server intermittently did not get a response, such that the client hung. Since the problem was so poorly reproducible (and we were short on time), I was not able to find the cause. But the client software was a CORBA server with very high availability requirements, hence this behavior was absolutely intolerable, and extra monitor threads had to be implemented for the purpose of detecting, killing, and replacing threads that were stuck due to RMI calls that did not return.

Example 10.4.2. An enterprise application in a big bank company used a Java applet for the graphical user interface. One day, suddenly, all bank employees in Germany could not work with the application anymore since their HTTP browsers froze on loading the applet. It took the emergency task force half a day to figure out the cause: As a measure for enabling central administration, the Java security settings on the users' workstations referred to some URL where the central security settings were to be read from an HTTP server. And over night somebody had closed the necessary port 80 on this server machine, so the Java plug-in of the browser did not get a response to its HTTP `GET` request for retrieving the security preferences and blocked the browser process indefinitely.

A little error message like "cannot access policy.url ... specified in java.security" would have saved the company a lot of money, time, efforts, anger.

The recommendation to be given here is not to avoid method invocations :-) What I want to point out is that thorough error handling should take care of the possibility of calls not getting a response. This possibility is a real one if remote calls are involved. The solution is of course to set a limit ("timeout") for the time to wait for a response and to handle it properly as an error if this time limit is exceeded. It is not the subject of this paper to present techniques for implementing such timeout mechanisms, this can in fact be pretty difficult and expensive: the usual trade-off between software quality and development costs.

Chapter 11

Testing Error Handling

As already mentioned, error-handling code (`throw` statements and catch handlers) is often poorly covered in tests, in particular in automatized unit tests. Apart from the fact that programmers seem to dislike to occupy themselves with errors, a common reason is that it can be objectively difficult to simulate error conditions. Passing a nil argument to some method and checking whether it throws a `PreconditionViolation` is easy, but how would you provoke an `IOException` on reading from a file? Or a `RemoteException` on invocation of a remote method? For the latter case, one could manually disconnect a network cable, but this is no solution for an automatic test that is to run without human interference.

Controlled failure of components can in most cases be achieved by using mock objects tailored for producing exceptions in place of the real ("domain") implementation.

Example 11.0.1. Let's take the challenge posed above. We want to make sure that a client method `bar` calling an RMI server behaves as it should if it receives a `RemoteException` from a remote object implementing

```
public interface Foo extends Remote
{
   void foo() throws RemoteException;
}
```

although the domain implementation of `foo` does never throw `RemoteException`. (The `RemoteException` would be raised by the RMI protocol layer.) `bar` reads the JNDI name "org/acme/Foo" of the RMI object `FooImp` from a system property "org.acme.Foo.name". It uses this name to look up the RMI object, retrieves a stub object for it, and invokes `foo` on it.

What we can do is the following: Since we want to test `bar` and not `FooImp`, we replace the server object. We write another ("mock") implementation `FooMock` of `Foo` with

```
public void foo() throws RemoteException
{
   throw new RemoteException();
}
```

and deploy it under some different name, say "org/acme/mock/Foo". (The RMI implementation of Sun Microsystems wraps the `RemoteException` into a `java.rmi.ServerException`, but this is derived from `RemoteException`). The test code now simply changes the system property "org.acme.Foo.name" to this name, executes `bar`, which then uses `FooMock` rather than `FooImp` and can show how it copes with the `RemoteException`. (And after that the test code should perhaps reset the system property to its original value in order to leave other test cases undisturbed.)

By the way, this example supplies an additional lesson: software design should also provide for making the software easy to test. If the implementation of `bar` had read the JNDI name only from a file or even hard-coded it, we would have had a harder time trying to let `bar` use our mock object without touching `FooImp`.

Bibliography

[1] J Gosling, B Joy, G Steele, G Bracha: *The Java Language Specification.* 2nd ed.

[2] http://jakarta.apache.org/log4j

[3] *Enterprise JavaBeans Specification, Version 2.0.* Sun Microsystems 2001

[4] http://www.jboss.org

[5] B Meyer: *Object-Oriented Software Construction.* Prentice-Hall, 1997