

BERUFSAKADEMIE MANNHEIM  
Information Technology International

Lecture Notes

Programming II  
Part 7 Errors and Robustness

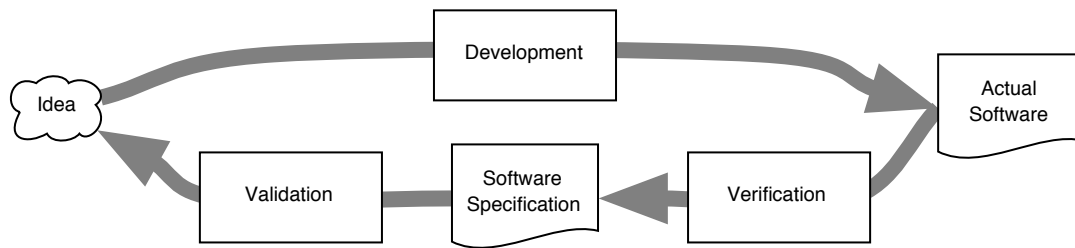
Dipl.-Betriebswirt (BA) VOLKAN YAVUZ  
{xmachina GmbH

November 2001–February 2002  
Classes: TIM01AGR && TIT01EGR  
Term: 1

**Contents**

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Correctness . . . . .	2
1.2	Robustness . . . . .	2
1.3	Quality Assurance . . . . .	3
<b>2</b>	<b>Errors</b>	<b>3</b>
2.1	Types of Errors . . . . .	3
2.1.1	Causes of Errors . . . . .	3
2.1.2	Symptoms of Errors . . . . .	4
2.1.3	Discovery of Errors . . . . .	4
2.2	The Lifecycle of Errors . . . . .	5
2.3	Typical Errors that affect Robustness . . . . .	5
2.3.1	Variable Declaration . . . . .	5
2.3.2	Arrays . . . . .	6
2.3.3	Pointers . . . . .	6
2.3.4	Memory Leaks . . . . .	6
2.3.5	Buffer Overflows . . . . .	6
2.4	Detecting and Handling Errors . . . . .	7
<b>3</b>	<b>Robustness through Exception Handling</b>	<b>7</b>
3.1	Robustness by Exception Handling . . . . .	7
3.2	Exception Handling Statements . . . . .	8
3.3	Advantages of Exceptions . . . . .	9
3.3.1	Separating Error Handling Code from Regular Code . . . . .	10
3.3.2	Propagating Errors up the Call Stack . . . . .	11
3.3.3	Grouping Error Types and Error Differentiation . . . . .	12
3.4	Catching or Specifying Exceptions . . . . .	13
3.5	Extending java.lang.Throwable . . . . .	13

**Figure 1:** Validation vs. Verification



**References**

**1 Introduction**

Computer programs tend to be faulty or unstable and very often exhibit unwanted symptoms like crashes and unpredicted behaviour. Although it is often not possible to *guarantee* that a program is free from errors, it is nevertheless possible – with the use of careful programming and well-designed programming tools – to keep the problems to a minimum.

**1.1 Correctness**

*Correctness* of a program is the property that it accomplishes the task it was designed to perform. Given an unambiguous *specification*, it is possible to *formally* prove the correctness of a computer program. This process is called *verification* (“does the program do the thing right?”). Unfortunately, this involves a very great amount of work, and it is only possible if the specification is present in an almost mathematically precise form. The other part, assuring that the specification is correct, is called *validation* (“does the program do the right thing?”) and is usually done by *inspection*, *review*, or *walkthrough* Balzert [1998].

**Verification**

- Ensure software meets specifications
- Internal consistency
- “Are we building the product right?”

**Validation**

- Ensure software meets customer’s intent
- External consistency
- “Are we building the right product?”

**Definition 1 (Correctness)** A *correct* program produces for *any* input that satisfies the *precondition* an output that satisfies the *postcondition*. Formally proving the correctness of a program is called *verification*. □

Validation and verification are usually performed after the specification or the program have been written, respectively, but careful work can yield results that already possess a very high degree of quality (see figure 1.)

**1.2 Robustness**

*Robustness* of a program is the property that it can handle illegal inputs and other unexpected situations in a reasonable way. It is a lot easier to write a program that works under ideal circumstances than making the program *robust*. One approach is to anticipate the problems that might arise and to include

test in the program for each possible problem and sensible ways to deal with and recover from these. There are many problems with this approach as it is difficult and sometimes impossible to anticipate all the possible things that might go wrong and also it is not always clear what to do actually when an error is detected. This also turn programs, that would normally be implemented in a straightforward way into a messy tangle where exceptional cases are detected and handled, thus obfuscating the implementation [see Eck, 2001, section 9.3].

While robustness of a program is an optional property, which greatly depends on who, how many other people and for how long it will be in use, correctness is an obligatory property. A program that performs incorrectly is almost useless.

### 1.3 Quality Assurance

Both correctness and robustness are aims of quality assurance. The quality of software products can be assured during construction via *constructive* measures, while *analytical* measures verify it. Although it is possible to detect many quality deficiencies during analysis, i. e. *after* the software has been implemented, and thus improve the quality, preventing deficiencies in quality is preferable, nevertheless.

- Constructive Quality Assurance
- Analytical Quality Assurance

## 2 Errors

Errors are unwanted behavior of software systems. Their cause lies in hardware defects, faulty systems or application software and also in data. A systematic error handling, i. e. detecting, logging, and handling errors are properties of a well designed and robust system [see Denert, 1992, pp. 297].

### 2.1 Types of Errors

The matrix in figure 2 on the next page describes the behavior of a software system by two criterions: whether that behavior is desired or not, and whether it is expected or not. The normal (error-free behavior) of software is both desired and expected. By *expecting* undesired behavior, the software is prepared to recognize those and performs appropriate actions and in consequence improves on robustness. In these cases, if an error has been detected the software will still be under control, i. e. it will be in a defined state. Undesired behavior that is not expected leads to “catastrophes” because the software is either not able to detect that behaviour or not capable to handle it in a reasonable way. The software enters an *undefined* state, where anything might happen (which typically means an abnormal termination).

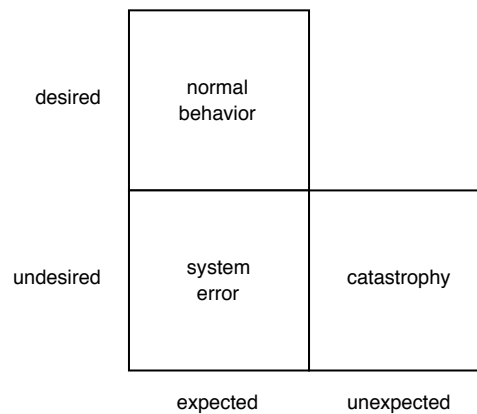
Errors by users (invalid input, requesting actions that are not sensible in the current state, etc.) are no system errors. They are categorized as “expected” and “desired”.

#### 2.1.1 Causes of Errors

While discussing undesired behavior, *symptom* and *cause* of an error have to be differentiated. Causes of errors might be

- Errors in design, e. g. interfaces.
- Implementation errors like implementation of a wrong algorithm or a faulty implementation of a correct algorithm.
- Errors in data, like faulty data that was received from an interface and assumed to be correct.
- Temporarily unavailable resources like database servers.

**Figure 2: Errors**



- I/O (Input/Output)-errors caused by (hardware) defects in peripheral devices.
- Hardware defects, like faulty memory.
- Errors in configuration of either the systems software or the application software.

### 2.1.2 Symptoms of Errors

The symptoms might be

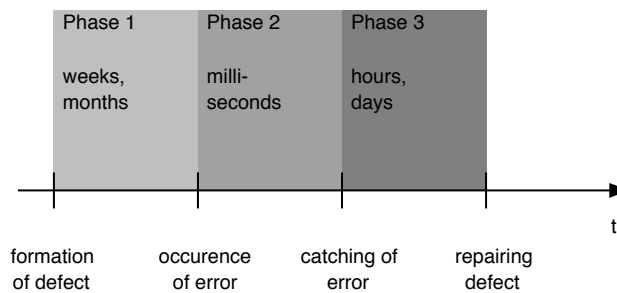
- Program flow diverting from the expected.
- Data corruption, or
- code corruption.

### 2.1.3 Discovery of Errors

Another way to differentiate errors is the time at which they occur or are being discovered:

- Syntactical errors are detected by the compiler at a very early stage during development and prevent the running of a program at all
- Warnings produced by the compiler during compilation. These are (heuristically) reported by the compiler because of untypical usage of language features, definition of variables that are never used, etc. They might be caused by implementation errors and should be investigated. Compilers can only detect a very limited subset of such errors.
- Runtime errors that are encountered at runtime after successful compilation. These are detected by the runtime environment and cause termination of the program with the production of a stack trace.
- Semantical errors occur during runtime but are not detected by the runtime environment. The program will continue execution, but produces faulty results.

**Figure 3:** Lifecycle of an Error



## 2.2 The Lifecycle of Errors

The lifecycle of errors can be roughly divided into three phases (see figure 3, [see Denert, 1992, p. 300]).

**Phase 1: Formation of a defect to occurrence of an error** The cause of an error is a defect, either in hardware, software or data, which well may already be in existence for a long period of time (weeks, months, or even years). Programming defects may stand unnoticed for a long time because a given constellation of input data has not occurred yet to bring the defect piece of code into execution.

**Phase 2: Occurrence of an error and its detection** The time between an error occurs and its detection should ideally be very short (and include as few instructions as possible) to prevent from traces getting lost. Ideally, the error is caught by detecting it, logging the relevant contextual information and neutralising its ramifications.

**Phase 3: Catching an error and repairing the defect** The repair defects that caused errors is the responsibility of system maintenance. Using the information gathered in phase 2, the programmer tries to identify the defect based on its symptoms and corrects it.

## 2.3 Typical Errors that affect Robustness

It is interesting to note that insufficient robustness is to a great deal caused by certain language features, and a great number of problems can be attributed to those language features.

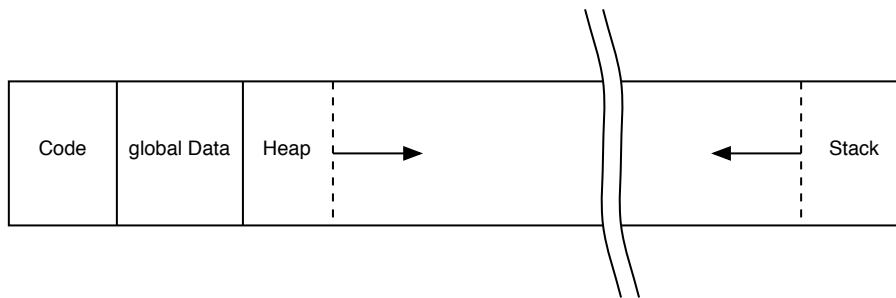
ACM:RISKS contains many reports about publicly known failures in computing systems. Errors typical to Java are discussed in van der Linden [2001]. The many darker corners of the C programming language are treated in van der Linden [1994].

### 2.3.1 Variable Declaration

Unlike Java, many programming languages don't require variables to be declared prior to usage. As this might seem convenient, it introduces the possibility, through a spelling mistake, that a new variable is introduced that had not been intended to be used.

Shortly after its launch on July 22, 1962, the Mariner I Venus probe veered off course and had to be destroyed by mission control officials. The problem was later traced to a single character error in the controlling software.

In FORTRAN, a command like 'D0 20 I = 1,5' is the first statement of a loop. As spaces are insignificant in FORTRAN, the above statement is equivalent to 'D020I=1,5'. Unfortunately, the programmer made a typing error and substituted the comma with a period, yielding a statement like 'D020I = 1.5', which is an absolutely valid assignment to a variable 'D020I'. If variables had to be declared, the compiler could have detected this error.

**Figure 4:** Typical Memory Layout

### 2.3.2 Arrays

An array stores uniform objects, that can be accessed by an integer index. Unlike Java, many programming languages do not check whether the index used to access the array is valid and actually within the bounds of the created array. An attempt to read or even store data beyond the array results in C or C++ results in unpredictable behaviour, since it cannot be told what is contained in those memory locations. In Java, this kind of error is detected by the runtime system and an 'ArrayIndexOutOfBoundsException' is thrown.

### 2.3.3 Pointers

*Pointers* are used in many programming language to make it possible to create dynamic data structures, using a variable sized area of the memory. Unfortunately, those programming languages do not check whether a pointer is still in a valid state. It is often possible to let that pointer reference any part of the memory, with the result that critical pieces might be overwritten. In Java, the corresponding concept is implemented by *references*, and the compiler is able to detect improper use of references. During runtime, it is impossible to let references directly reference parts of the memory.

### 2.3.4 Memory Leaks

The use of pointers causes another problem, namely *memory leaks*. Whenever an area of memory is allocated for further usage and referenced by a pointer, it is the programmers responsibility to free that area of memory when it is not needed, anymore. However, most programming languages cannot make sure that all allocated memory is released when not needed, anymore. In Java, the *garbage collector* makes sure that any unreachable object (occupying memory) is discarded whenever more memory is needed.

### 2.3.5 Buffer Overflows

*Buffer Overflows* are another typically made error leading to unexpected behaviour in program, and that may also be exploited causing severe security issues. During runtime (a typical memory layout is shown in figure 4), local variables are stored on the runtime stack, along with the return address of calling functions. If data is to be read in a fixed sized local memory location is longer than the reserved memory, it will overwrite the address of the calling function. As the stack now contains random data, program execution will continue at some random point in memory, even in locations, where no valid machine code will be available. Transmitting valid code as data and deliberately causing *Buffer Overflows* gives a potential intruder to let a remote machine execute code. Errors of this type cannot occur in Java, because the runtime does neither allow direct access to random memory locations, nor will it pass program execution to regions of memory that are not known to carry valid machine instruction.

## 2.4 Detecting and Handling Errors

The objectives while handling errors are:

- Preventing catastrophes, i. e. to expect almost all behavior (albeit undesired) and to be able to handle these conditions in a reasonable way.
- Detecting errors as soon as possible.
- Handling errors as gracefully as possible, thus preventing abnormal program termination or crashes.
- Collecting as much information as possible about the causes and the circumstances to be able to recover or repair.
- Performing tests with access to this information as a guide to debugging.
- And in general, improving overall system robustness by means of a consequential error handling.

## 3 Robustness through Exception Handling

### 3.1 Robustness by Exception Handling

There are many possible ways to react whenever an error has been detected by the program:

- Each method checks for errors itself after each operation that may result in an error. This results in bloated programs that are impossible to maintain.
- Immediately terminating the currently running method and returning an appropriate value to the calling method indicating the type of error detected. The responsibility to handle the error is delegated to the calling function. It is problematic that the return value might not be able to contain enough information about the error. Additionally, each return value has to be checked for the case that it may be a value indicating an error.
- The Unix C library traditionally returns a value of `-1` to indicate an error and sets the global variable `errno` to a value describing the error. It is not guaranteed that return codes are checked and the value of `errno` evaluated properly. Also, global variables violate the most fundamental principles of programming style. Global variables also introduce new problems in multithreaded environments.
- As soon as an error is detected, a message is produced and program execution continues. This might lead to unpredictable behavior.
- After producing an appropriate message stating the detected error condition, the program is terminated at once.

Exception handling is used to detect and possibly recover from errors at runtime. While exception handling in itself is not sufficient to improve robustness, it provides some mechanics to do so in a more organized way. It also allows the separation of normal behavior from exceptional behavior, of error detection from error handling. The latter case is especially relevant to libraries. While the client cannot detect errors, but knows how to handle them, only the library can detect errors, but would not know how to handle them.

Exceptions can be used to help write robust programs. They provide an organized and structured approach to robustness. Without exceptions, a program can become cluttered with `if` statements that test for various possible error conditions. With exceptions, it becomes possible to write a clean implementation of an algorithm that will handle all the normal cases. The exceptional cases can be handled elsewhere, in a `catch` clause of a `try` statement [see Eck, 2001].

Most of the material discussed here is contained in “The Java Tutorial”, Trail “Essential Java Classes”, Lesson “Handling Errors with Exceptions” [Campione et al., 2000] and in Eck [2001].

The Java language uses exceptions to provide error-handling capabilities for its programs. The term *exception* is shorthand for the phrase “exceptional event”.

**Definition 2** An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions. □

While the compiler is able to detect syntactical and some semantical errors at compile time, handling errors that occur at runtime (i. e. during the execution of a program) fall into the responsibility of the programmer and the runtime environment. Runtime errors can be caused by many problems, including

- hardware failure,
- temporarily unavailable resources like database servers,
- formally invalid input, like a filename containing invalid characters,
- semantically invalid input, like “february, the 30th” as a date,
- any simple programming errors, like accessing an array out-of-bounds,
- etc.

Whenever such an error occurs, an exception object is created (*thrown*) and handed off to the runtime system. Execution of the program is terminated, and the runtime system looks for some code to handle the error. This piece of code is called an *exception handler*. In the search for an appropriate exception handler, the runtime system searches backwards through the call stack, beginning with the method in which the error occurred, until it finds a method that contains an appropriate exception handler. If the runtime system exhaustively searches all of the methods on the call stack without finding an appropriate exception handler, the runtime system (and consequently the Java program) terminates.

### 3.2 Exception Handling Statements

‘try’	The ‘try’ statement identifies a block of statements within which an exception <i>might</i> be thrown and also states that exception handling will take place for the enclosed block of statements.
‘catch’	The ‘catch’ statement identifies a block of statements that can handle a particular type of exception. The statements are executed if an exception of that particular type occurs with the ‘try’ block.
‘finally’	The ‘finally’ statement identifies a block of statements that is guaranteed to be executed regardless of whether or not an exception has been thrown within the try block.

Shown below is the general form of the ‘try’-statement and its ‘catch’ and ‘finally’ clauses. In a ‘try’ statement, at least a ‘catch’ or ‘finally’ clause has to be given. There can also be more than one ‘catch’ clause, but the ‘finally’ clause can appear at most once.

```
1 try {  
2     <statement>  
3 } catch (<ExceptionType> <name>) {  
4     <statement>  
5 } catch (<ExceptionType2> <name>) {  
6     <statement> ]  
7 } finally {  
8     <statement>  
9 }
```

The following paths of execution are possible:

1. No exception gets thrown in the 'try'-block:
  - The 'catch'-block is ignored.
  - The 'finally'-block gets executed, if present.
  - Program execution continues immediately after the 'try'-statement.
2. An exception gets thrown within the 'try'-block:
  - An exception might be thrown either by the JVM or explicitly by an 'throw' statement.
  - If an exception is thrown, execution will continue at the end of the 'try'-block.
  - For each following 'catch'-block, the class of the exception to be caught is compared with the exception thrown.
    - a) There exists a matching 'catch'-block following the 'try'-block:
      - A 'catch'-block matches the thrown exception if it declares the class of the thrown exception or a direct or indirect parent class thereof. It is necessary that the most specific exception classed are caught *before* more general exception classes.
      - When no other exception gets thrown within the 'catch'-block, the exception is assumed to have been handled.
      - The 'finally'-block, if present, will be executed.
      - Execution will continue immediately after the 'try'-statement, *not* at the statement following the place the exception got thrown.
    - b) There does not exist a matching 'catch'-block following the 'try'-block:
      - The 'finally'-block, if present, will be executed.
      - Program execution will continue at the immediately enclosing 'try'-block, either in the current method or the directly or indirectly calling method.
      - A matching 'catch'-block will be searched.
      - In case no matching 'catch'-block was found, program execution will terminate.

### 3.3 Advantages of Exceptions

In traditional programming code that detects, reports, handles, and recovers from errors often leads to confusing code, where the original flow of logic gets obfuscated. Shown below is an example (in pseudo-code) of a program that reads a whole file into memory for further processing:

```
10 readFile {
11     open the file;
12     determine its size;
13     allocate sufficient memory;
14     read the file into memory;
15     close the file;
16 }
```

That example clearly shows how the program is intended to work, and it seems simple enough to be implemented without much difficulties. But, many potential errors are not considered:

- What happens if the file can't be opened?
- What happens if the length of the file cannot be determined?
- What happens if there is not enough memory available?
- What happens if the read fails?

- What happens if the file cannot be closed?

To be able to detect and try to recover from these potential problems, much code has to be added, yielding a program that might look like this:

```
17 readFile {
18     open the file;
19     if (theFileIsOpen) {
20         determine its size;
21         if (gotTheFileLength) {
22             allocate sufficient memory;
23             if (gotEnoughMemory) {
24                 read the file into memory;
25                 if (readFailed) {
26                     errorCode = -1;
27                 }
28             } else {
29                 // !gotEnoughMemory
30                 errorCode = -2;
31             }
32         } else {
33             // !gotTheFileLength
34             errorCode = -3;
35         }
36         close the file;
37         if (theFileDidntClose && errorCode == 0) {
38             errorCode = -4;
39         } else {
40             errorCode = errorCode and -4;
41         }
42     } else {
43         // !theFileIsOpen
44         errorCode = -5;
45     }
46 }
```

### 3.3.1 Separating Error Handling Code from Regular Code

Given the exception handling support of Java, the same could be implemented as follows:

```
47 readFile {
48     try {
49         open the file;
50         determine its size;
51         allocate sufficient memory;
52         read the file into memory;
53         close the file;
54     } catch (fileOpenFailed) {
55         doSomething;
56     } catch (sizeDeterminationFailed) {
57         doSomething;
58     } catch (memoryAllocationFailed) {
59         doSomething;
60     } catch (readFailed) {
61         doSomething;
62     } catch (fileCloseFailed) {
63         doSomething;
64     }
65 }
```

### 3.3.2 Propagating Errors up the Call Stack

Suppose that the ‘readFile’ is the fourth method in a series of method calls.

```
66 method1 {
67     call method2;
68 }
69
70 method2 {
71     call method3;
72 }
73
74 method3 {
75     call readFile;
76 }
```

It might be that ‘method1’ is the only method interested in possible error conditions that might have occurred in ‘readFile’. In that case, all methods in between have the responsibility to propagate an eventual error code up the call hierarchy.

```
77 method1 {
78     error = call method2;
79     if (error)
80         doErrorProcessing;
81     else
82         proceed;
83 }
84
85 method2 {
86     error = call method3;
87     if (error)
88         return error;
89     else
90         proceed;
91 }
92
93 method3 {
94     error = call readFile;
95     if (error)
96         return error;
97     else
98         proceed;
99 }
```

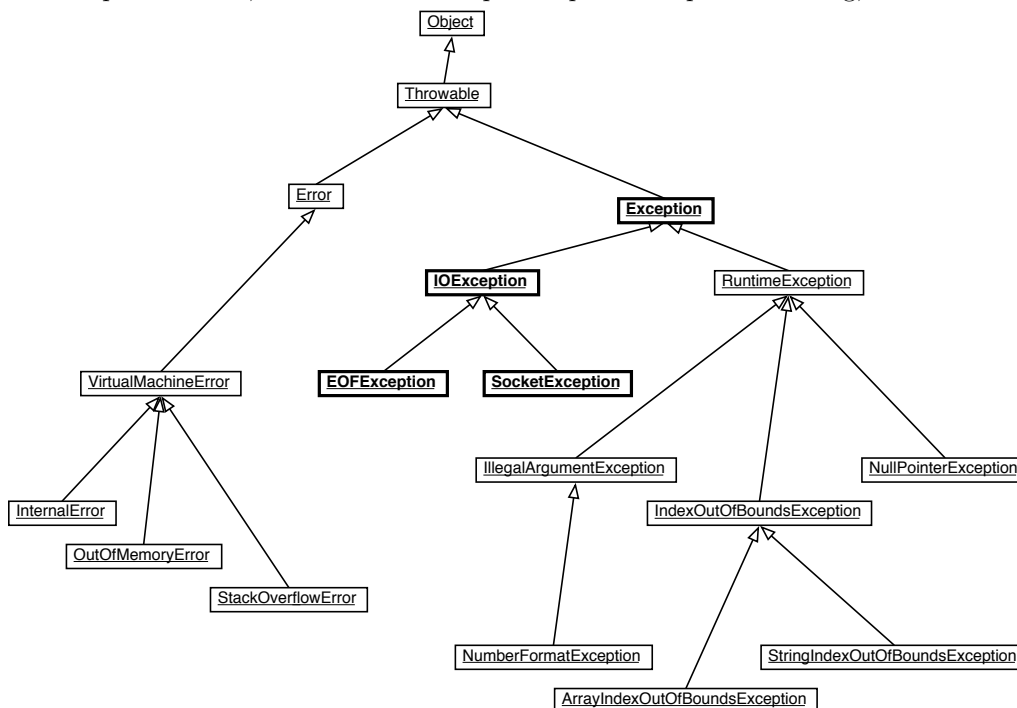
The two previous examples also have shown that the classical error handling is flawed in that it is susceptible to two types of problems [Warren and Bishop, 1999, pp. 63–65]:

1. It cannot be ensured that a client actually checks a return code that reports errors and handles accordingly.
2. The error condition might have been double-checked, both by the implementor and the user of the method. From a correctness point of view this is better, but obviously will lead to inefficient code.

As the Java runtime system searches backwards through the call stack to find an appropriate exception handler, propagation of error conditions is performed automatically, and does not overly affect pieces of code that do not intent to handle those error conditions. As those “middleman” methods participate in error propagation, they declare the checked exceptions thrown. Additionally, the ‘throws’-clause is part of a method’s public programming interface.

**Figure 5:** Exception Classes

Checked exception classes, i. e. classes that require explicit exception handling, are shown in bold.



```

100 method1 {
101     try {
102         call method2;
103     } catch (exception) {
104         doErrorProcessing;
105     }
106 }
107
108 method2 throws exception {
109     call method3;
110 }
111
112 method3 throws exception {
113     call readFile;
114 }
    
```

### 3.3.3 Grouping Error Types and Error Differentiation

As exceptions in Java are first-class objects, the concept of inheritance can be used to group closely related types of exceptions under one common inheritance hierarchy. Whenever a specific class identifier is used in the 'catch'-clause of the 'try'-statement, all exceptions of that class *and* any subclasses can be caught by that 'catch'-clause.

Some of the many predefined exception classes are shown in figure 5.

### 3.4 Catching or Specifying Exceptions

Java requires that a method either *catches* or *specifies* all *checked exceptions* that can be *thrown within the scope of the method*.

**catch or specify** Whenever a method is called that might throw an exception, the programmer has to either

- catch that exception, or to
- specify explicitly that this method might throw that exception.

As any exception that can be thrown by a method is really part of the method's public programming interface, callers of a method must know about the exceptions that a method can throw in order to intelligently and consciously decide what to do about those exceptions [see Gosling et al., 2000].

**checked exceptions** Java knows of many different exception classes, all of which fall into one of three categories:

- **'Error'**, and subclasses thereof are *unchecked exceptions* and indicate serious problems that a reasonable application *should not* try to catch. Most such errors are abnormal conditions, and no method is required to declare any subclasses of **'Error'** in its **'throws'** clause. Additionally, they can occur at many points in the program and recovery from them is difficult or impossible. A program declaring such exceptions would be cluttered, pointlessly.

Summarized, **'Error'**s are conditions whose cause lies not in the responsibility of the programmer, and from which the programmer is unlikely to be able to recover.

- **'RuntimeException'**, and subclasses thereof are *unchecked exceptions*, too, and indicate error conditions during the normal operation of the Java Virtual Machine, that can arise at many places within the program. However, from the programmers point of view, those error conditions are symptoms of flaws in his code, so the original cause should be identified and fixed. **'RuntimeException'**s are also exempted from compile-time checking because, having to declare such exceptions would not aid significantly in establishing the correctness of programs.

Summarized, **'RuntimeException'**s are usually caused by flaws in the program (e. g. improper use of other classes and their methods), and their *prevention* is the responsibility of the programmer.

- Any other subclass of **'Exception'** are *checked exceptions*, i. e. that the compiler ensures that these exceptions are either *caught* or *specified* in the **'throws'**-clause. These exceptions represent useful information about the operation of a legally specified request that the caller may have had no control over and that the caller needs to be informed about, so she might be able to recover from that condition.

Summarized, *checked exceptions* are caused by circumstances outside the control of the caller, and they provide valuable information for the recovery from these conditions.

**thrown within the scope of the method** means any exceptions, that

- are thrown directly by the method using the **'throw'** statement, and
- are thrown indirectly by the method through calls to other methods.

### 3.5 Extending java.lang.Throwable

As soon as it is not immediately possible to handle exceptional conditions, the programmer may want to **'throw'** an exception, either by using the already defined exception classes (e. g. **'IllegalArgumentException'** or **'IOException'**) or by declaring new exception classes if no proper representation of the actual error

condition is present. The new class must extend ‘`java.lang.Throwable`’ or any of its subclasses. If the programmer does *not* want mandatory exception handling, ‘`RuntimeException`’ (or any of its subclasses) have to be extended. If mandatory exception handling is required, any of the other subclasses of ‘`Exception`’ or ‘`Exception`’ itself can be extended.

**Throwable(String message)** creates a new ‘`Throwable`’ with the given error message.

**public String getMessage()** returns the message string.

**public String getLocalizedMessage()** returns a localized version the message string.

**public String toString()** returns a short description of this throwable object.

**public void printStackTrace()** prints the stack backtrace of this throwable (at the time the throwable was created) to ‘`System.err`’.

**public void printStackTrace(PrintStream s)** prints the stack backtrace of this throwable (at the time the throwable was created) to the given ‘`PrintStream`’.

**public void printStackTrace(PrintWriter s)** prints the stack backtrace of this throwable (at the time the throwable was created) to the given ‘`PrintWriter`’.

**public Throwable fillInStackTrace()** can be used whenever a caught exception is thrown again (maybe after inspection it has been determined that it cannot be handled immediately). Returns an instance will the stack backtrace information filled in with the current stack context. For example:

```

115 try {
116     a = b / c;
117 } catch (ArithmeticThrowable e) {
118     a = Number.MAX_VALUE;
119     throw e.fillInStackTrace();
120 }
```

**List of Tables**

**List of Figures**

1	Validation vs. Verification . . . . .	2
2	Errors . . . . .	4
3	Lifecycle of an Error . . . . .	5
4	Typical Memory Layout . . . . .	6
5	Exception Classes . . . . .	12

**List of Acronyms**

I/O . . . . . Input/Output

**References**

ACM:RISKS. Forum on risks to the public in computers and related systems. news:comp.risks. URL <http://catless.ncl.ac.uk/Risks/>. Peter G. Neumann (Moderator).

- Helmut Balzert, editor. *Lehrbuch der Software-Technik*. Lehrbücher der Informatik. Spektrum Akademischer Verlag, Heidelberg, Berlin, 1998. ISBN 3-8274-0065-1. URL <http://www.swt.ruhr-uni-bochum.de>.
- Mary Campione, Katy Walrath, and Alison Huml. *The Java Tutorial: A Short Course on the Basics (With CD-ROM)*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 3rd edition, January 2000. ISBN 0-201-70393-9. URL <http://web2.java.sun.com/docs/books/tutorial/>.
- Ernst Denert. *Software-Engineering – Methodische Projektentwicklung*. Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo, Hong Kong, Barcelona, 1992. ISBN 3-540-52404-0.
- David J. Eck. Introduction to programming using java, 2 2001. URL <http://math.hws.edu/javanotes/>.
- James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 2nd edition, June 2000. ISBN 0-201-31008-2. URL <http://java.sun.com/docs/books/jls/index.html>.
- Hans-Werner Six, Christina Esser, Wilfried Lange, Detlef Meier, Thomas de Ridder, and Mario Winter. *Konzepte imperativer Programmierung*. FernUniversität - Gesamthochschule in Hagen, 10 1998. URL <http://www.fernuni-hagen.de/LVU/guidedtour/inhalte/2-tour/5-kursangebot\%/inf11612/index.html>. 1612.
- Peter van der Linden. *Expert C Programming – Deep C Secrets*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1994. ISBN 0-13-177429-8.
- Peter van der Linden. Frequently asked questions (with answers) for programmers using the java language, 2001. URL <http://www.afu.com/javafaq.html>.
- Nigel Warren and Phil Bishop. *Java in Practice – Design Styles and Idioms for Effective Java*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1999. ISBN 0-201-36065-9.
- James A. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, 2000. URL <http://www.computer.org/spftware/so2000/pdf/s1070.pdf>.